AN ABSTRACT OF THE THESIS OF

Byoungchul Ahn for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on May 3, 1989.

Title: The Analysis and Synthesis of a Parallel Sorting Engine

Redacted for Privacy

Abstract approved: ______________________
John M. Murray

This thesis is concerned with the development of a unique parallel sort-merge system suitable for implementation in VLSI. Two new sorting subsystems, a high performance VLSI sorter and a four-way merger, were also realized during the development process. In addition, the analysis of several existing parallel sorting architectures and algorithms was carried out.

Algorithmic time complexity, VLSI processor performance, and chip area requirements for the existing sorting systems were evaluated. The rebound sorting algorithm was determined to be the most efficient among those considered. The rebound sorter algorithm was implemented in hardware as a systolic array with external expansion capability.

The second phase of the research involved analyzing several parallel merge algorithms and their buffer management schemes. The dominant considerations for this phase of the research were the achievement of minimum VLSI chip area, design complexity, and

logic delay. It was determined that the proposed merger architecture could be implemented in several ways. Selecting the appropriate microarchitecture for the merger, given the constraints of chip area and performance, was the major problem. The tradeoffs associated with this process are outlined.

Finally, a pipelined sort-merge system was implemented in VLSI by combining a rebound sorter and a four-way merger on a single chip. The final chip size was 416 mils by 432 mils. Two micron CMOS technology was utilized in this chip realization. An overall throughput rate of 10M bytes/sec was achieved. The prototype system developed is capable of sorting thirty two 2-byte keys during each merge phase. If extended, this system is capable of economically sorting files of 100M bytes or more in size. In order to sort larger files, this design should be incorporated in a disk-based sort-merge system. A simplified disk I/O access model for such a system was studied. In this study the sort-merge system was assumed to be part of a disk controller subsystem.

The Analysis and Synthesis of a Parallel Sorting Engine

by

Byoungchul Ahn

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed May 3, 1989

Commencement June 1989

Approved:

Redacted for Privacy

Associate Professor of Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented May 3, 1989

Typed by Byoungchul Ahn for Byoungchul Ahn

ACKNOWLEDGEMENTS

First of all I would like to express my gratitude to my advisor, Dr. John Murray, who has provided me with generous support, constant encouragement, and advice throughout this research study. I would also like to thank Drs. James Herzog, Theodore Lewis, and Thomas Dietterich for serving on my thesis committee and for their dedication in reading my manuscripts and offering valuable feedback. In addition, my very special thanks to Dr. Roy Rathja for helping me to get started with this study and encouraging my research work, as well as serving on my thesis committee. I would also like to thank Mr. Richard Lary who first suggested the double buffering technique used in conjunction with the four-way merger.

Finally, I am ever grateful to my wife, my son, and my parents for their patience during this long period of study. In particular, my wife and son are due my special thanks for enduring the Oregon hay fever season each spring.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

APPENDICES

LIST OF FIGURES

LIST OF FIGURES (continued)

Figure Page

LIST OF TABLES

Table Page

LIST OF APPENDIX FIGURES

LIST OF APPENDIX TABLES

THE ANALYSIS AND SYNTHESIS OF A PARALLEL SORTING ENGINE

1. INTRODUCTION

Recent advances in very large scale integration (VLSI) technology have resulted in a significant rate of increase in general purpose computer performance. One consequence of this increase in performance has been a corresponding increase in the amount of data requiring computer processing. The development of VLSI systems for performing specialized processing functions on large volumes of data has lagged behind the development of more general purpose computing systems. The development of special purpose VLSI systems is expected to provide high performance in certain application areas at a reasonable cost. One application area, that of sorting large files, appears amenable to a hardware solution. VLSI parallel sorters have the potential for sorting very large files efficiently and economically.

This study includes the development of a parallel VLSI sort-merge system suitable for implementation on silicon. An analysis of parallel sorting architectures and algorithms is also accomplished. A new parallel sorting algorithm is not developed, *per se*. Rather, the unique design of a VLSI sort-merge system is accomplished by the interconnection of a number of new sorting subsystems. These include a parallel hardware sorter, merger, and associated support hardware. Through the combination of these subsystems, a high performance sort-merge system was obtained. This strategy saved substantial design time and effort over that involved in the creation and implementation of a conceptually new sort-merge algorithm. The resulting sort-merge system can be extended to accommodate a wide range of needs for sorting applications ranging from 10^3 to 10^{12} bytes in size. The system has the ability to provide increased performance in executing many of the basic operations required by various specialized processing systems at relatively low cost and with high performance.

1.1 Motivation and Objectives

Sorting has been one of the most widely studied areas of computer data processing and a problem area recognized as long ago as 1945 when Von Neumann and Goldstine collaborated on the development of the EDVAC (electronic discrete variable automatic computer). Since 1973, the most complete reference on sequential sorting is Knuth's three volume study, *The Art of Computer Programming*. In his third volume, *Sorting and Searching* [26, p.3], Knuth indicated that:

> Computer manufacturers estimate that over 25 percent of the running time on their computers is currently being spent on sorting, when all

> their customers are taken into account. There are many installations in which sorting uses more than half of the computing time.

In 1985, Linderstrom pointed to the importance of sorting operations in a discussion of the amount of sorting time required for bank applications [31]. He estimated that when using conventional disk merge-based methods, major banks spent two or more hours each night sorting demand deposit accounts. These accounts must be processed prior to opening on the next business day. The sorted files in question were on the order of several hundred megabytes (10^6 bytes) in size, a figure which is expected to increase by an order of magnitude before the end of this decade. In this case, even with the current generation of faster disks, an optimized merge sort would take 10 to 15 hours. Currently, larger banking applications encompass sorting files as large as 800 million bytes, which may include 2 million records, with 400 bytes per record, and 50 bytes per key. Within this decade file sizes in applications other than banking are expected to reach more than 10 gigabytes (10^9 bytes), or 10 million records with 1000 bytes per records and 100 bytes or more per key. Therefore, the need for the development of faster and more cost effective sorting methods is obvious.

Conceptually, it is possible that the demand for special purpose rapid sorting systems could be diminished due to the recent increases in general purpose computing performance and low cost, high capacity memory systems. However, for the foreseeable future, the problem of sorting will remain as a dominant need in computer information processing.

There are several reasons for this conclusion. First, there is an increasing demand for database operations which can accommodate the large amounts of data required by some applications, including office automation processes, computer-aided design and computer-aided manufacturing (CAD/CAM), graphics, and image processing systems [20,38,42]. For these applications, data continues to be stored in peripheral disks or tape. As the amount of data, the number of records, the record lengths, and field sizes continue to grow, the necessity for sorting will be unavoidable despite improvements in processor speed and memory capacity. Fast and efficient database operations involving large amounts of data are primarily concerned with performing general purpose sorting functions and relational operations. Efficient sorting computations are the most effective method of completing merge operations that eliminate duplicate records as well as performing other data base operations [10,19,32,34,37].

Second, many parallel sorting algorithms have been developed to improve the sorting performance compared with serial sorting on traditional sequential machines. Many of these parallel sorting algorithms concentrate on massively interconnected networks or processors. These algorithms focus on sorting operations, without regard to the means of data transfer to and from the sorting hardware [4,7,28,35,40,44]. When parallel sorting algorithms are evaluated, the costs of reading and writing data to peripheral devices should be incorporated in their formulation, rather than ignored, *i.e.*, there is no guarantee that a parallel sorting algorithm of the O(log N) time complexity is optimal when the data input and output transfer rate is O(N^2). Thus, the issue of data input and output time is equally as important as the actual sorting operation itself when very large files are involved.

Finally, VLSI developments, particularly the availability of small geometry CMOS devices and the emergence of high level VLSI CAD technology, has encouraged the design of inexpensive and fast sorters. The fact that dynamic and static RAM capacities have continued to increase and costs decrease has also helped make special purpose VLSI sort-merge system solutions attractive. The type of system under consideration can be incorporated as part of either a database machine, or an independent sorting system with a high capacity RAM memory, plus a serial disk, fast parallel disk, or a RAM disk as required. The VLSI sort-merge system in this context will eliminate a part of the computational burden on the general purpose central processing unit (CPU) contained in almost all computing system configurations, allowing it to concentrate upon more valuable general purpose computing tasks.

To solve the above problems, special purpose parallel hardware should be designed which can utilize the advantages inherent in VLSI technology rather than conducting sorting operations by software on traditional computers or on parallel computers. The performance capabilities of an N processor parallel sorting algorithm implemented in VLSI can be linearly proportional to the number of items to be sorted. Indeed, several VLSI sorting algorithms have shown impressive performance, performance significantly greater than that obtained in sequential systems or parallel systems [1,5,36,39,50].

Most parallel VLSI algorithms are of O(N) or O(log$N$) time complexity with N processors. In contrast, sorting algorithms for traditional sequential computers provide O($N\log N$) to O(N^2) performance in sorting N items [26]. When parallel VLSI architectures are used for sorting, the size of the files which can be sorted is often limited by the number of processing elements and the amount of local memory available. Performance will be highest if a large number of processing elements and a substantial amount of high speed local memory are available. For reasons stated at the beginning of this study, external sorting techniques are generally a necessity for sorting very large files [9,12,21,24]. In 1988, Beck and Francis evaluated the application of parallel external sorting to parallel computers and were able to show significant improvements in sorting performance [6,22]. However, they used a merge-sort algorithm rather than a special purpose hardware sorter. To date, practical high capacity VLSI-based external sort-merge systems have not been realized.

Therefore, the objective of this study is the design of a special purpose parallel VLSI sort-merge system in which efficient sorting and merging algorithms can be implemented to solve high performance parallel sorting problems. The proposed sort-merge system includes several VLSI subsystems. These subsystems not only provide pipelined sorting capability, but also provide the initial presorting when sorting very large files. The merge function is designed to utilize pipelining capabilities for the improvement of overall sorting performance, based on the use of O(log$N$) processors.

The most efficient sorting algorithm was selected for implementation in VLSI following the analysis and comparison of several parallel sorting algorithms. The VLSI system design process included the high level simulation of the proposed algorithm, as well as the functional simulation and verification of the resulting chip. These steps were performed in order to assure the correctness of the design. A hierarchical design methodology and careful documentation process were utilized. The system was implemented using a commercial silicon compiler. Although a parallel VLSI sorter of O(N) time complexity was developed, it can be shown to be both complex and expensive to build an isolated N-processor VLSI sorting engine for large values of N. Therefore, a VLSI merger was incorporated with the VLSI sorter to improve the overall sorting capacity while maintaining reasonable hardware costs. The merger plus presorter combination performs not only the internal sorting of small files into sorted strings, but also provides an external sort capability for sorting extremely large files in conjunction with external memory structures and/or peripheral storage devices. The external sorting performance of the sort-merge system is closely related to the input and output operations of the mass storage devices utilized, *e.g.*, magnetic disks. For external sorting, I/O processing is a significant performance factor because the data must be transported between main memory and the storage devices a number of times during sorting operations. Therefore, it is necessary to carefully define an appropriate disk device I/O model for sort-merge functions. Such a model will provide information regarding collective system performance. The sort-merge system developed may be utilized in a variety of system configurations. A general application is illustrated in Figure 1.1.

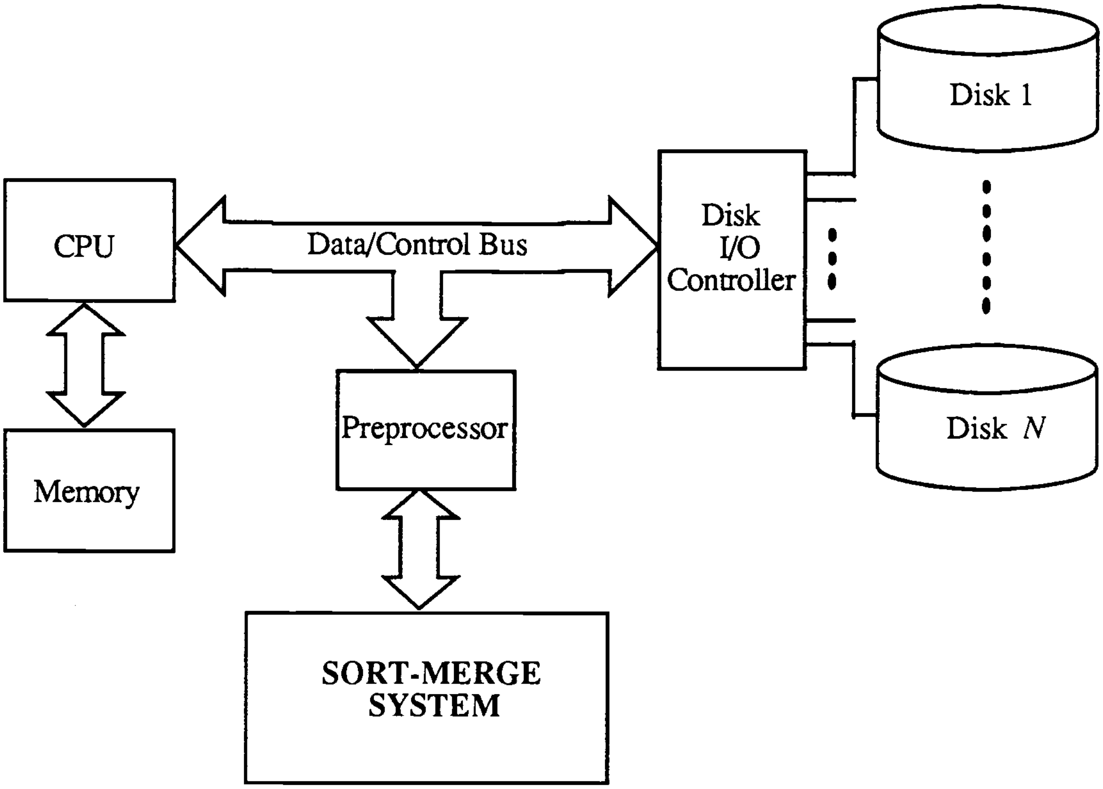


Figure 1.1 Sort-Merge System Application

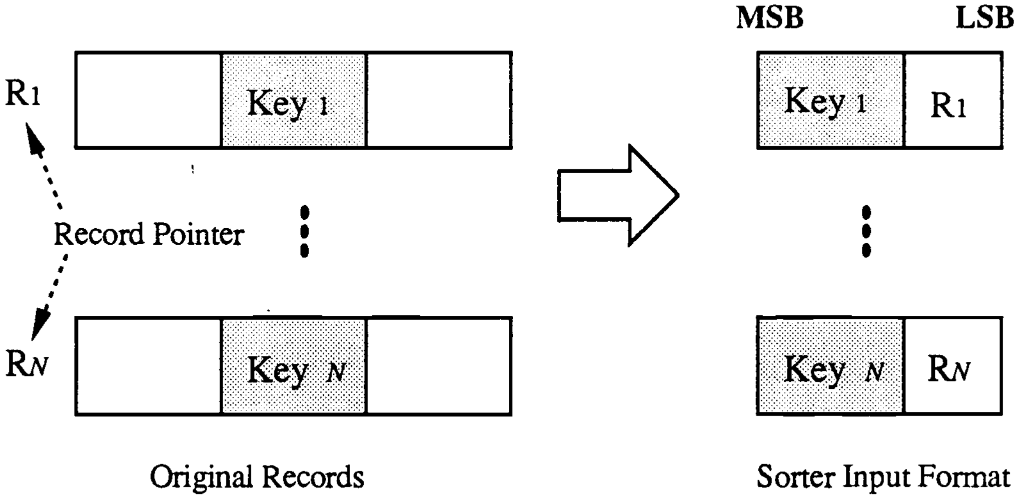


Figure 1.2 Input Data Structure

1.2 Statement of the Problem

Typically a data file is divided into records, the basic units of data processing. Each record is composed of one or more fields and an ordered subset of the fields is designated as the key field or key. Sorting is defined to be the process of rearranging records in ascending or descending order by the relative magnitude of the key fields. When a file to be sorted is loaded into the memory of a local processing unit and the results of the sorting activity are placed in memory as well, the sort is defined to be an internal sort; otherwise, it is an external sort.

The input data structures for a typical hardware sorter are indicated in Figure 1.2. The key fields are extracted from each record and the keys plus associated pointers are compared. When the key field is extracted the record pointer is added to the key so that the original record order may be preserved. This procedure is accomplished by a sort preprocessor, which retrieves the keys and record pointers for each record before sorting, and restores the original records following sorting, if desired.

The most common methods of organizing disk files are sequential, partitioned, indexed sequential, and direct organization [13]. The disk I/O access model is dependent upon the file organization, the I/O blocking factor, the disk scheduling method, and the disk access time. The output of an external merge sorter is a sequence of records in ascending or descending order. Records are generally distributed on the disk, prior to sorting, in random order. For all practical purposes, file organization can be defined as a partitioned sequential organization since the output of the external merge sort is a sequence of sorted records. The final output will have one partitioned sequence which corresponds to the entire file, while the initial input of an external merge sort will have one record per partitioned sequence. A partitioned sequential file can be stored and retrieved in a indexed sequential order for each input and output operation.

1.3 Definitions and Notation

The following is a summary of the notation and terms used throughout this study. When additional notation and terms specific to individual sections and/or chapters are introduced, they are defined at the point of introduction.

1.3.1 Definitions

$\log x$ = $\log_2 x$ logarithm based on 2,

$\lceil x \rceil$ = the least integer greater than or equal to x,

$\lfloor x \rfloor$ = the greatest integer less than or equal to x,

$[a,b]$ = $\{x \mid a \leq x \leq b\}$ or the range of values that lie between a and b inclusively,

(a,b) = $\{x \mid a < x < b\}$ or the range of values that lie between a and b exclusively.

1.3.2 Notation

B = buffer size in bytes,

N = number of records in the input file,

P = number of processors,

S = number of a presorted sequence,

r = record size in bytes,

m = order of merge,

α = disk assess time,

β = average rotational latency time (= 1/2 disk rotation time),

θ_r = disk read time,

θ_w = disk write time,

δ = physical disk block size in bytes,

γ = disk cylinder size in bytes,

τ = total number of blocks (= rN/δ),

$p(n)$ = complexity of processor,

$t(n)$ = complexity of time,

$b(n)$ = complexity of buffer size,

$c(n)$ = total cost (= $p(n) \times t(n)$).

1.4 Organization of the Study

This study is presented in six chapters, including the introductory chapter. Chapter 2 provides an analysis of existing parallel sorting algorithms. Several previously implemented VLSI sorters are also briefly considered. In addition, three VLSI sorting algorithms with similar structural characteristics are compared and their architectural complexities and concepts are explained. The implementations of these sorters in VLSI are compared in terms of chip area and processing speed. In order for the comparison to be meaningful, they were all implemented with the Genesil silicon compiler [23] in 2-micron CMOS technology.

Chapter 3 presents design methods for the implementation of a rebound sorter originally conceived by Chen *et al.* [18]. The rebound sort algorithm constitutes the most efficient type of algorithm with respect to chip area and processing time found. Improved and modified features, as well as pipelining control capability and expandability, were added to Chen's original concept. Functional simulation, timing, floorplanning, and routing were performed and are discussed.

In Chapter 4, several merge sort algorithms are analyzed in terms of VLSI implementation. These algorithms include a parallel binary merge sort algorithm [9,21] and a pipelined merge-sort algorithm [25,47]. Since the disadvantage of external mergers is the requirement of large external buffers, buffer management schemes for reducing on-chip and external RAM size are discussed. Two comparator implementation methods, the first an indirect four-way comparator and the second a direct four-way comparator, are provided for the improvement of processing speed. A VLSI four-way merger is implemented using the indirect four-way comparator technique.

Chapter 5 presents an integrated sort-merge system which combines two rebound sorters and the four-way merger. The system is designed to reduce merger stages and disk I/O access time, based upon the analysis provided in chapter 4. The sort-merge system is designed for implementation on a single chip. This system may be modified if a larger sort-merge system is desired. Additional modular sort-merge systems, which connect individual sorter and merger chips for implementation of more efficient systems are examined. These include an iterative sort-merge system, a tree sort-merge system, and a disk based sort-merge system. The simple disk I/O access model is defined, analyzed, and applied to the external sort-merge system.

Chapter 6 summarizes the results of the study and suggests interesting avenues for further research.

2. PARALLEL VLSI SORTERS

The theory of parallel sorting algorithms has been explored in a number of studies. Some of these algorithms have been implemented in hardware. A few parallel sorting algorithms have been implemented with parallel processors and their performance has been demonstrated [6,22,43]. In this chapter, several sorting algorithms are discussed briefly. Criteria related to the time complexity of each algorithm and its implementation feasibility are presented. According to Akl [3] and Bitton *et al.* [11], parallel sorting algorithms may be broadly classified in three categories: (1) network sorting algorithms, (2) shared memory sorting algorithms, and (3) parallel file sorting algorithms.

Most of these algorithms require a large number of processors. The number of processors is proportional to N or, at a minimum, log$N$ when sorting N keys. Furthermore the size of the keys to be sorted is limited by the internal memory size. The sort performance and cost can be improved when N special VLSI processors are used instead of general purpose processors. Moreover, when sorting very large files, the data I/O transmission time between main memory and the storage devices cannot be ignored. Therefore, external sorting is more desirable than the internal sorting for sorting very large files. The most interesting aspect of parallel internal sorting algorithms is whether or not they can be implemented in VLSI and whether or not the input and output times can be overlapped with the sorting time.

Five parallel sorting algorithms can be considered, which meet the above criteria: (1) The enumeration sort proposed by Yasuura *et al.* [50], (2) the VLSI sorter proposed by Miranker *et al.* [36], (3) the rebound sorter proposed by Chen *et al.* [18], (4) pipeline merge sort proposed by Todd *et al.* [47], and (5) the hardware sort-merge system proposed by Takagi *et al.* [45]. The first three algorithms are analyzed and compared in this chapter, while (4) is discussed in Chapter 4 and (5) is discussed in Chapter 5.

2.1 VLSI Sorters

Sorting algorithms for magnetic bubble memory systems were proposed by Chung *et al.* [16] and Lee *et al.* [29] in 1980 and 1981, respectively. Earlier a number of sorting algorithms for other technologies were proposed by Chen and Todd in 1978. VLSI sorting systems which have been implemented were developed by Carey *et al.* in 1982, Yasuura in 1982, Miranker in 1983, Rajgopal in 1985, and Bate in 1988.

Carey *et al.* implemented a parallel version of the classic bubble sort algorithm, using a stack-based microarchitecture on silicon [15]. This system has O(N) time complexity and uses a chip area of O(N). The chip is not pipelined and requires 2N cycles to sort an N-element data stream in which 16-bit records, 8-bit integer keys and 8-bit record pointers, are pushed into a single stack. When records are popped from the stack, they are returned in ascending order. The core chip area was 7000 microns by 7000 microns (280 mils by 280 mils), excluding the power and I/O pads. Each chip has the capability of producing 32 16-bit sorted records at its output. The maximum estimated speed achieved by this system was about 0.84 MHz. The chip , which can be cascaded, may be used as a disk block buffer for a database machine.

Rajgopal *et. al* implemented Yasuura's parallel enumeration sort [39]. This implementation was also not pipelined. Its time complexity were approximately the same as the sorter developed by Carey *et al.* Rajgopal's implementation requires $2N$ cycles to sort an N element data stream and a single chip can sort 8 16-bit keys. Following the sorting activity, the index of the sorted key value is returned. Given the 8-key limit of each chip, 32 chips can be directly cascaded to sort up to 256 keys. The total die area was 9200 microns by 7900 microns (368 mils by 316 mils) in 4-micron NMOS technology. No speed performance estimates were given for this chip.

Bate implemented the rebound sorter in pipelined VLSI architecture [5]. This implementation achieved $O(N)$ time and area complexity, requiring $2N$ cycles to sort an N element data stream. A single chip is capable of sorting 16 16-bit keys and cannot be chained. The total area of the chip is about 10000 microns x 10000 microns (400 mils x 400 mils) in 2-micron CMOS technology and it achieves speeds up to 5.7 MHz. The chip pipeline controller design is very complex.

Segal described a content-addressable, sequential data management chip for the performance of sort and search operations in 1986 [42]. The data management chip consists of an articulated (adjustable) first-in, first-out (FIFO) register and a binary search engine. By implementation of an articulated FIFO register to manipulate data files, sorting times are up to 500 times faster than software sorting routines and the time complexity and area were almost proportional to *N*. With 1024 bytes of internal RAM, the chip treats each key as if it had two constant width fields, *i.e.*, a key field of 1 to 255 bytes and a pointer of 0 to 255 bytes. Up to 256 of these chips may be cascaded, enabling the sorting of 256 Kilo(10^3) bytes of data. The chip is fabricated with 1.6 micron CMOS technology and data speeds of 2.6M bytes per second at a clock rate of up to 16 MHz have been achieved. It is a powerful commercial sorting chip, but has 8-bit data bus to transfer data to and from the chip.

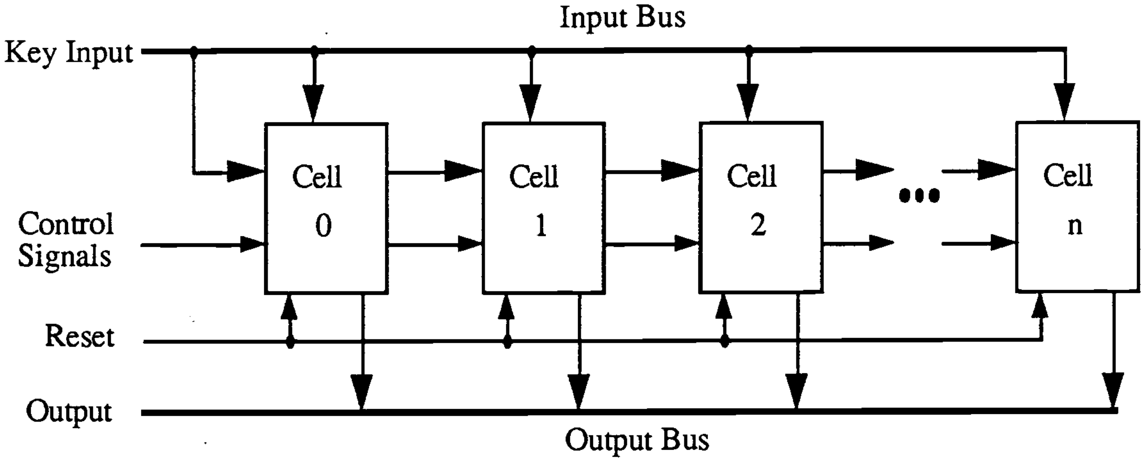
2.2 Analysis of VLSI Parallel Sorters

Many parallel sorting algorithms have been proposed for hardware or software implementation [2,4,7,14,18,36,40,44,47,50]. Few of these algorithmic architectures are suitable for implementation in VLSI [27]. The VLSI implementations must be small and capable of high performance in order to be commercially feasible. Many of the proposed implementations do not satisfy these criteria. Thompson [46] discussed the time-area complexity of VLSI sorters for 13 algorithms in order to find the existence of an area x time^2 tradeoff sorting problem. The area x time^2 performance figures for most of the designs were close to the theoretical minimum value, $O(N^2 \log N)$. In addition, Leighton and Bilardi conducted a study of the upper and lower bounds for this problem [8, 30]. These studies concentrated on the time-area complexity of the number of circuits, including the wires and gates, number of bits, and the time required to sort N keys. Possible restrictions on input and output times were removed.

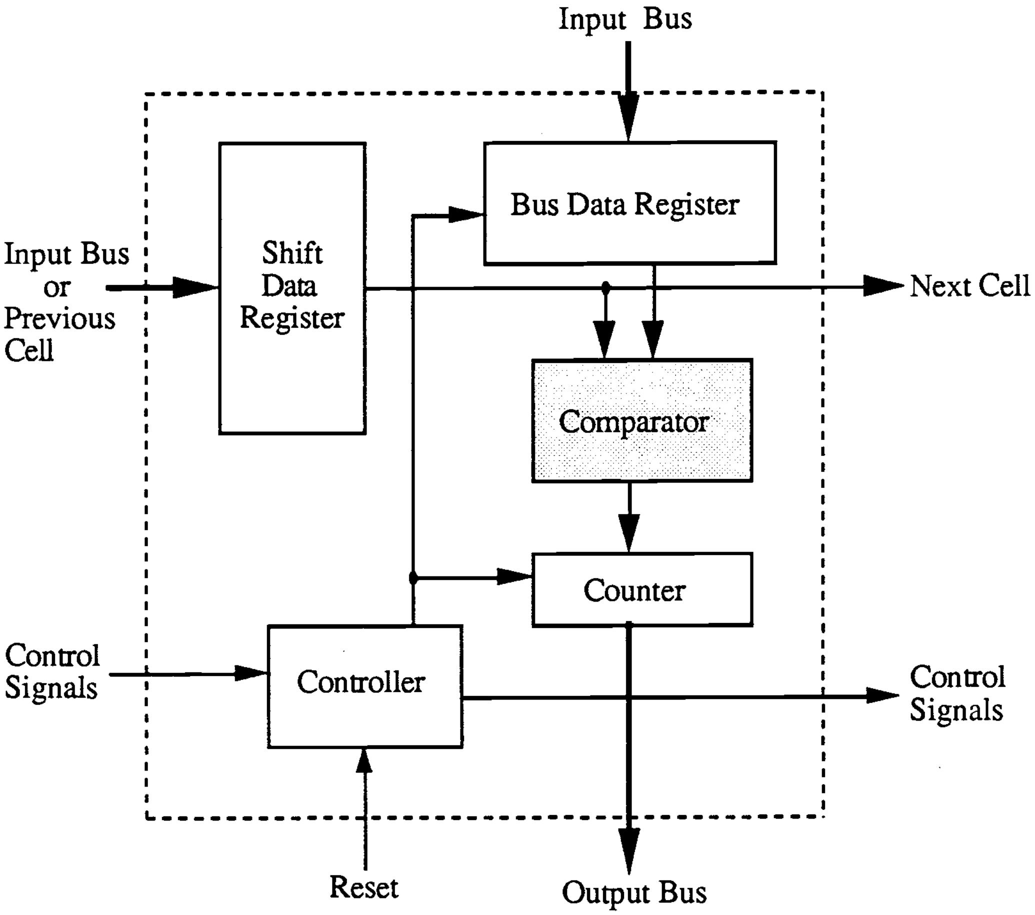
In the following sections, three other algorithms, each requiring N processors for sorting N keys, having an area proportional to N, are considered in more detail from the standpoint of time and processor complexity: (1) the enumeration sorter [50], (2) the VLSI sorter [36], and (3) the rebound sorter [18].

2.2.1. Enumeration Sorter

The enumeration sort can be regarded as two separate procedures, an ordering process and an arranging process. Ordering is the process of determining the order of each key in the set of keys, $K = \{k_1, k_2, \ldots, k_n\}$. When k_i is the i^{th} smallest key in the set of keys, K, the result returns the order c_i equal to i-1 corresponding to k_i. The order c_i is calculated by counting the results of comparisons between k_i and all of the keys in K. In turn, rearranging is performed by software, through the host computer or a memory device with special mechanisms. A block diagram for this algorithm is shown in Figure 2.1 and an example of the enumeration sorter operation is shown in Figure 2.2.



a. Parallel Enumeration Sorter Block Diagram



b. Enumeration Sorter Cell Block Diagram

Figure 2.1 Enumeration Sorter Block Diagram

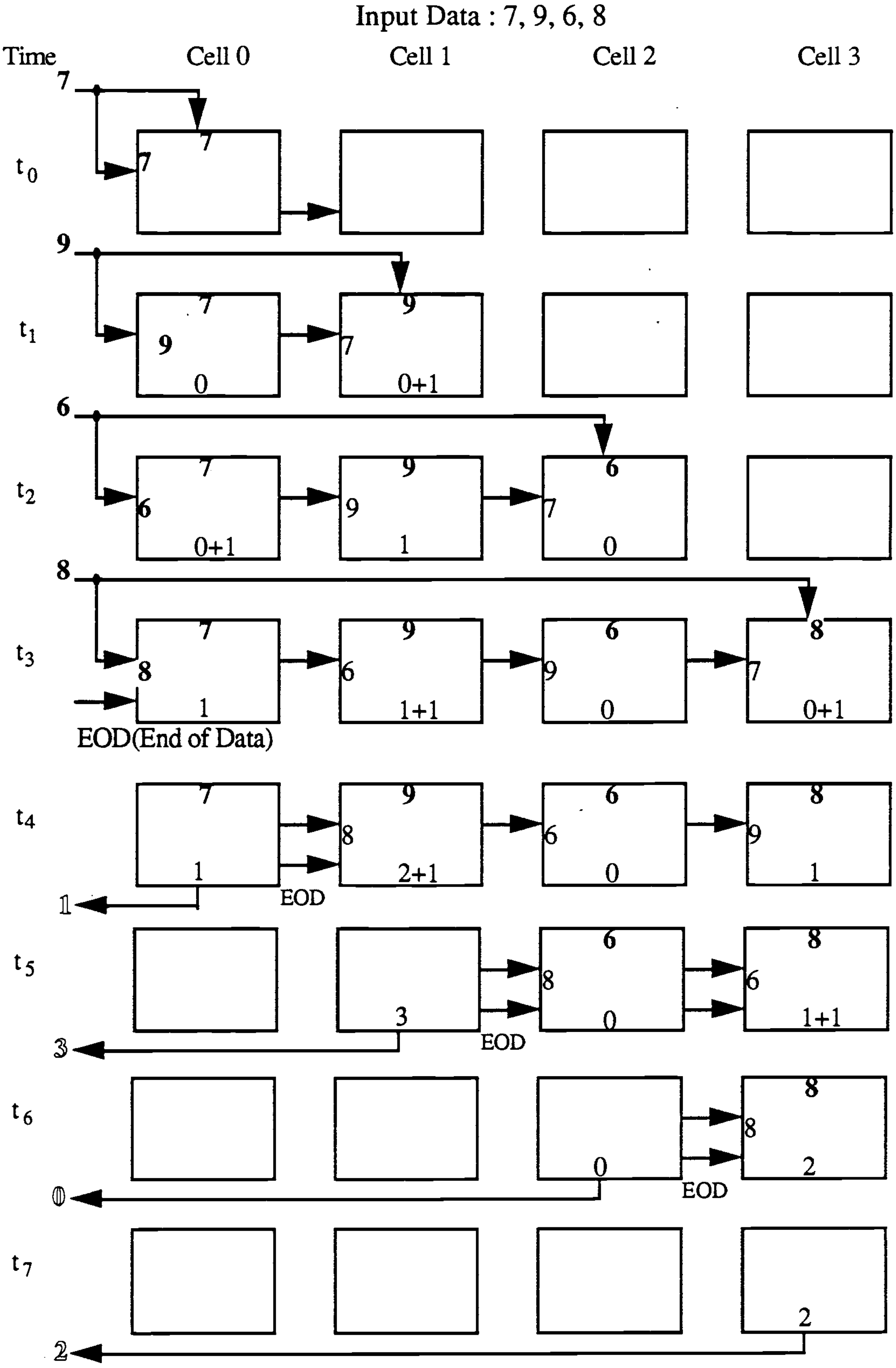


Figure 2.2 Enumeration Sorter Operation

```
begin
for $i$ = 1 to $N$ do in parallel
    $c_i = 0$
    for $j$ = 1 to $i$-1 do          /* $k_i = k_i$ and $i < j$ -> $c_i < c_i$ */
        if $k_i \geq k_j$ then
            $c_i = c_i + 1$
        end if
    end for

    for $j = i$ to $N$ do
        if $k_i > k_j$ then
            $c_i = c_i + 1$
        end if
    end for
end for
end
```

Figure 2.3 Enumeration Sort Algorithm (Algorithm 2.1)

Algorithm 2.1 shown in Figure 2.3 assures that two or more keys are not stored in the same location and that if $k_i = k_j$ and $i < j$, then $c_i < c_j$. N processor elements execute Algorithm 2.1 and compute orders in parallel. Keys and their orders are transmitted serially between the memory device and a sorting circuit which performs the ordering process. Input/output time completely overlaps the sort processing time.

The number of processors, p(N), for Algorithm 2.1 is N. The total time required is O(N^2). The time complexity for N processors is,

$$\mathrm{p}(N) = N,$$
$$\mathrm{t}(N) = \frac{\mathrm{O}(N^2)}{\mathrm{O}(N)} = \mathrm{O}(N),$$

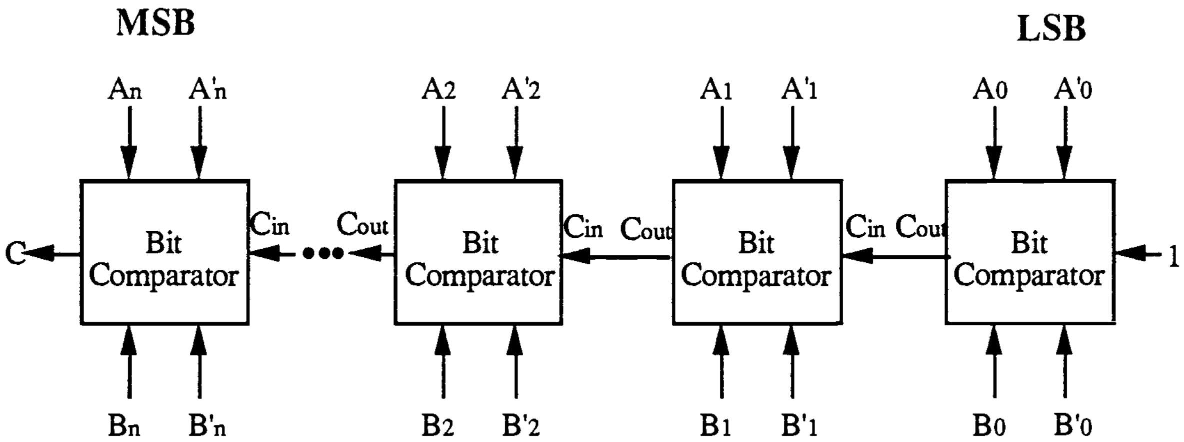
and the total cost, c(N), is

$$\mathrm{c}(N) = \mathrm{p}(N) \text{ x } \mathrm{t}(N) = \mathrm{O}(N^2)$$

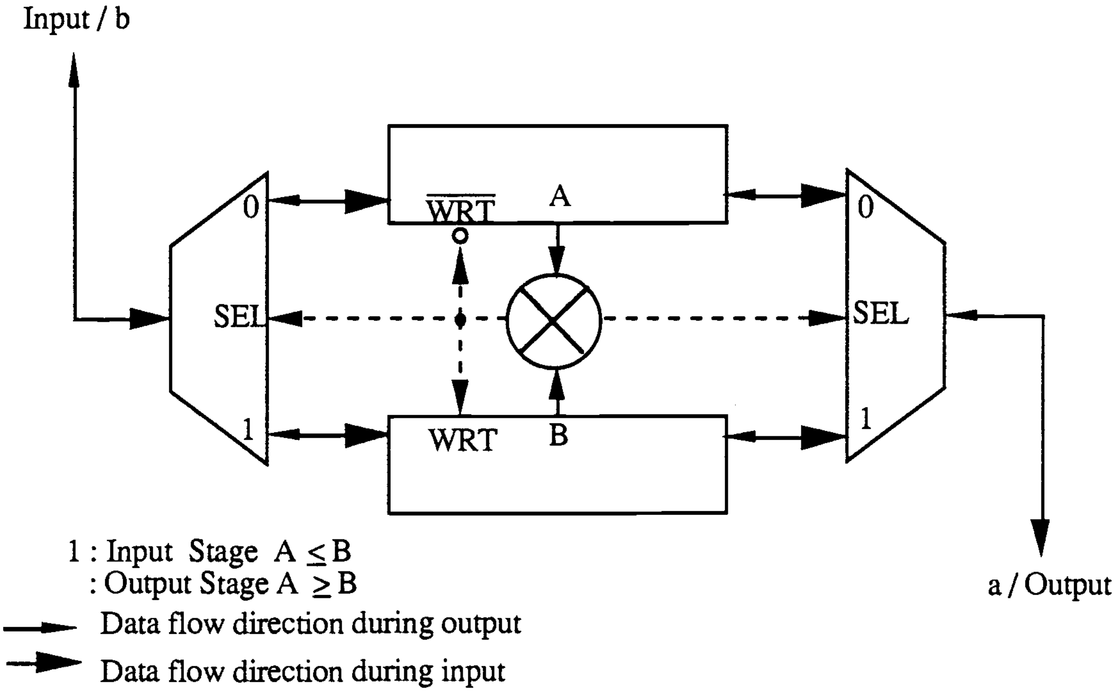
The ordering process requires 2N cycles. The algorithm requires a host computer rearranging process following completion of the ordering process. The reordering process is of O(N) time complexity.

2.2.2. The VLSI Sorter

The VLSI sorter proposed by Miranker *et al.* [36] are similar to the basic operations of the bubble sorter proposed by Lee *et al.* [29]. However, the cell microarchitecture is considerably different. This sorter consists of a linear array of $\lceil N/2 \rceil$ cells, each of which has an array of 1-bit comparators and memory elements. For instance, n-bit basic cells, *dibit cells*, are connected with each other to sort n-bit key fields. Each cell is connected to the cells above and below it in order to establish data communication and proper data movement during sorting. The block diagram of the basic dibit cell and the overall comparator structure is shown in Figure 2.4 and a sorting example of the VLSI sorter is provided in Figure 2.5.



a. Comparator Structure



b. Dibit Cell Block Diagram

Figure 2.4 Basic Cell of the VLSI Sorter

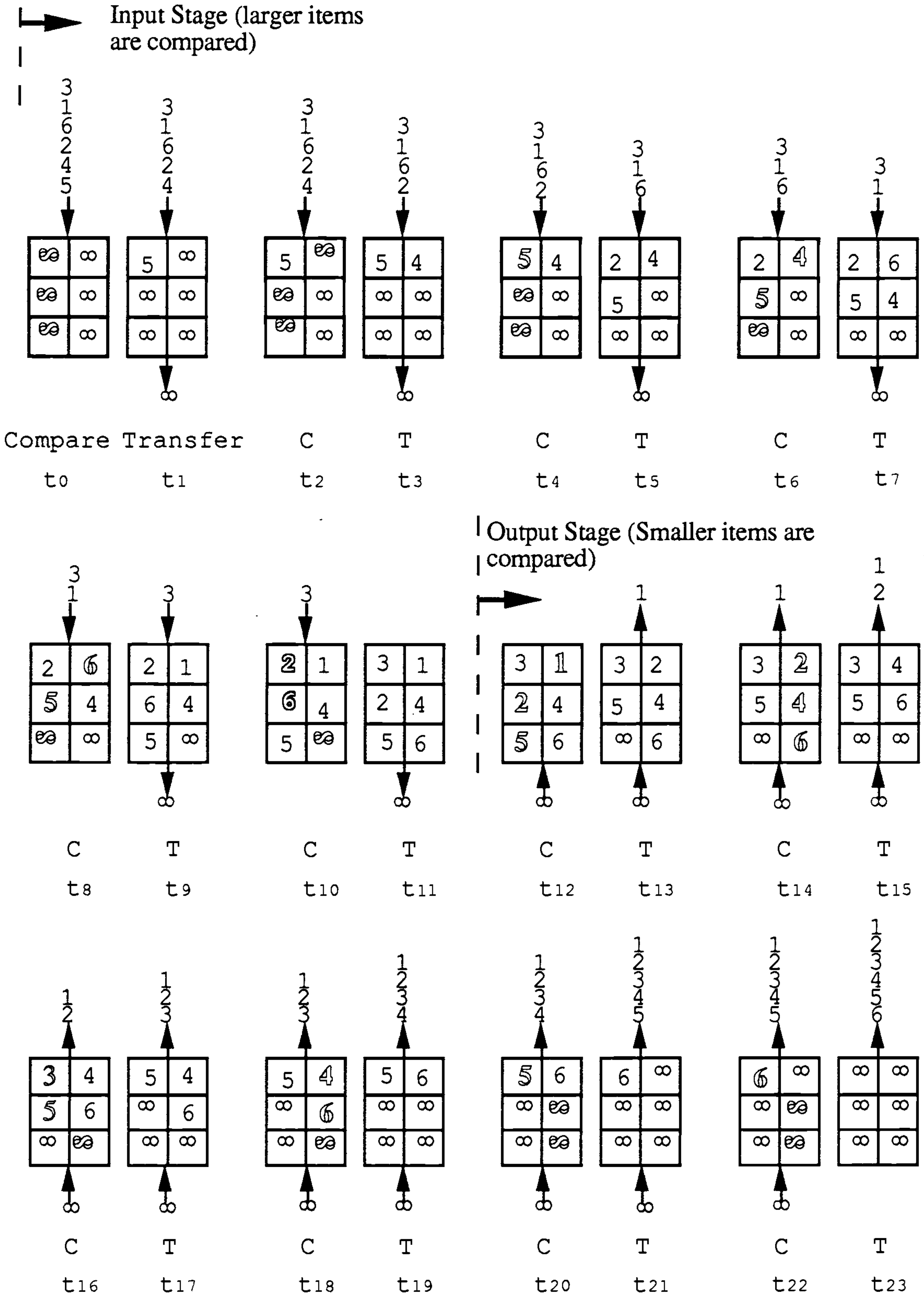

Input Stage (larger items are compared)
Output Stage (Smaller items are compared)
Compare
Transfer
C
T
t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
t13
t14
t15
t16
t17
t18
t19
t20
t21
t22
t23

Figure 2.5 The VLSI Sorter Operation

The VLSI sorter can sort keys in ascending or descending order by modifying the data flow direction. Each sorting step has two phases:

1. Compare Phase: Two keys in each cell are compared each other.
2. Transfer Phase: Depending on the result of the comparison, one of the two keys is transferred to a neighboring cell and the original cell receives an item from its remaining adjacent cell.

The VLSI sorter cell contains two stages, an input stage and an output stage. In the input stage, the larger of the two keys in each cell is transferred downward so that the smallest key is located in the top cell and the largest key is located in the bottom cell. In general, the i^{th} smallest key must be in one of the top i cells. In the output stage, the smaller of two keys in each cell is transferred upward (Figure 2.6, Algorithm 2.2).

The time required for Algorithm 2.2 is $O(N^2)$. When $N/2$ processors are used, the time complexity is $O(N)$:

$$p(N) = N/2 = O(N) ,$$

$$t(N) = \frac{O(N^2)}{O(N)} = O(N) ,$$

and

$$c(N) = p(N) \times t(N) = O(N^2) .$$

This algorithm can be used to sort one ascending order and one descending sequence. The total sorting time is $4N$ cycles to sort N keys with $N/2$ processors.

```
begin
for $j$ = 1 to $N/2$ in parallel
    for $i$ = 1 to $N$              /* Input stages */
        if $a_j \le b_j$ then       /* larger data are transferred */
            $a_{j+1} = b_j$ or $b_{j+1} = b_i$
            $b_j = \max(a_{i-1}, b_{i-1})$
        else
            $a_{j+1} = a_j$ or $b_{j+1} = a_i$
            $a_j = \max(a_{i-1}, b_{i-1})$
        end if

    end for
    for $i$= $N$+1 to 2$N$          /* Output Stage */
        if $a_i < b_i$ then         /* smaller data are transferred */
            $b_{i-1} = a_i$ or $a_{i-1} = a_i$

            $a_i = \min(a_{i+1}, b_{i+1})$
        else
            $a_{i-1} = b_i$ or $b_{i-1} = b_i$
            $b_i = \min(a_{i+1}, b_{i+1})$
        end if
    end for
end for
end
```

Figure 2.6 The VLSI Sorter Algorithm (Algorithm 2.2)

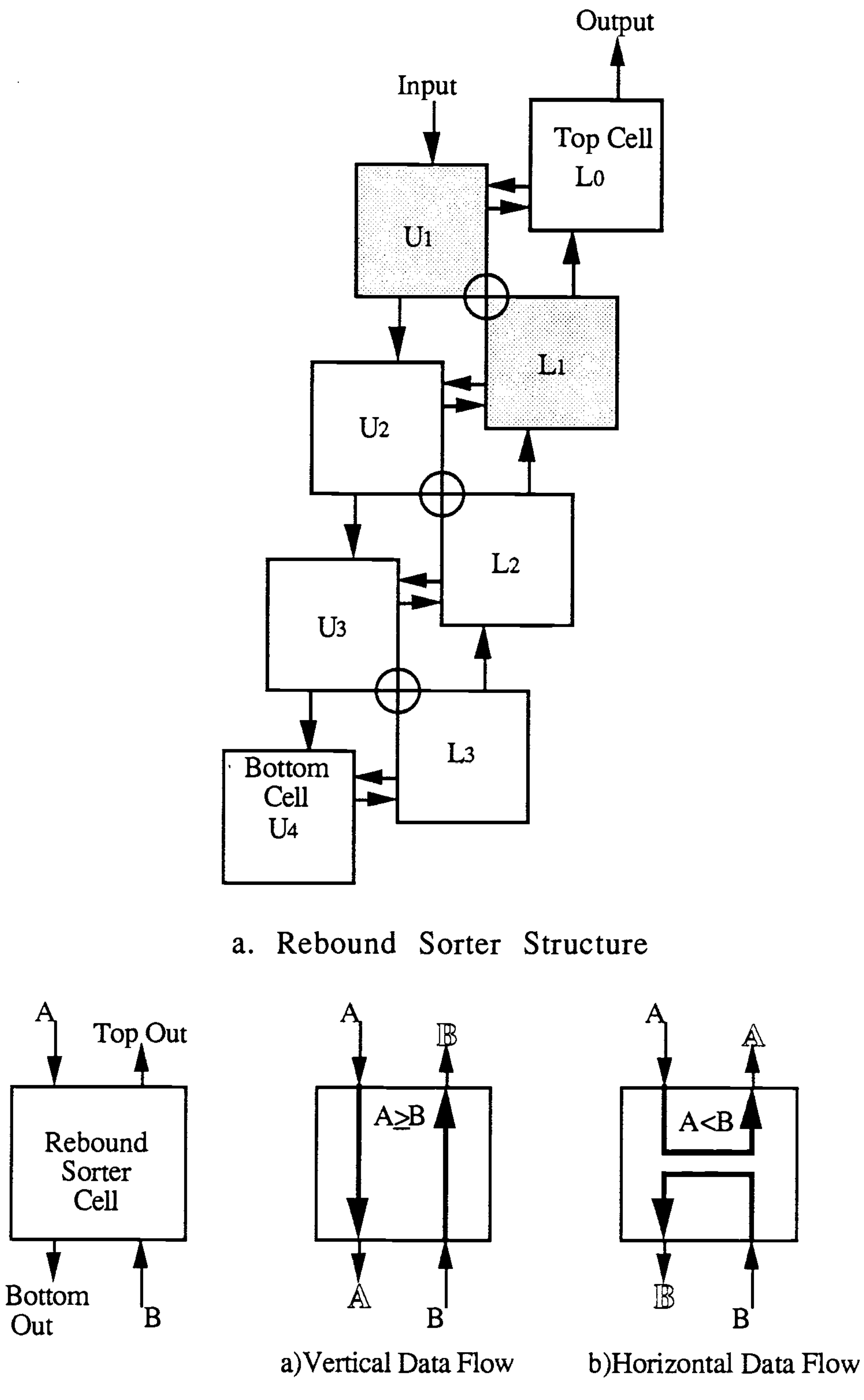
2.2.3. Rebound Sorter

The rebound sorter is a new sorting scheme based upon an improvement of the uniform ladder sorter [48], which is an N-loop shift register structure capable of holding N keys with one key per loop [17]. This storage structure is suitable for hardware implementation. The algorithm is a variation of the odd-even transposition sort algorithm. If the time for circulating data within a loop is called a period, then N keys stored in the ladder can be sorted in $(N+1)/2$ periods, using $(N-1)$ comparators.

Input and output time completely overlap sorting time. The rebound sorter cell consists of a controller, memory elements, and a data steering unit. Multiple instances of the basic cell are used to compose the complete sorter. The basic building block known as the steering unit, contains an upper-left cell, U, and a lower-right cell, L, as shown in Figure 2.7. A sorter for N keys is assembled by connecting $(N-1)$ steering units, plus the top and bottom cells. An example of the rebound sorter operation is shown in Figure 2.8.

In the rebound sorter, keys are divided into two parts: the most significant and the least significant parts. The most significant part of a key is fed into the sorter and the least significant part of the key follows after one clock period. Thus, in order to sort an N key sequence, $4N$ cycles are required.

The sorter alternates between two states: the decision state and the continuation state. The latter state may be a pass data vertical state, a pass data horizontal state, or a second decision state. In the decision state, each steering unit compares the key stored in its upper-left (U_i) and lower-right (L_i) cells, steering the halves of the keys either vertically or horizontally, depending on the comparison results. In the continuation state, each steering unit steers the keys in a direction determined from the previous decision state.



a. Rebound Sorter Structure

b. Rebound Sorter Cell Data Flow

Figure 2.7 Basic Structure of the Rebound Sorter

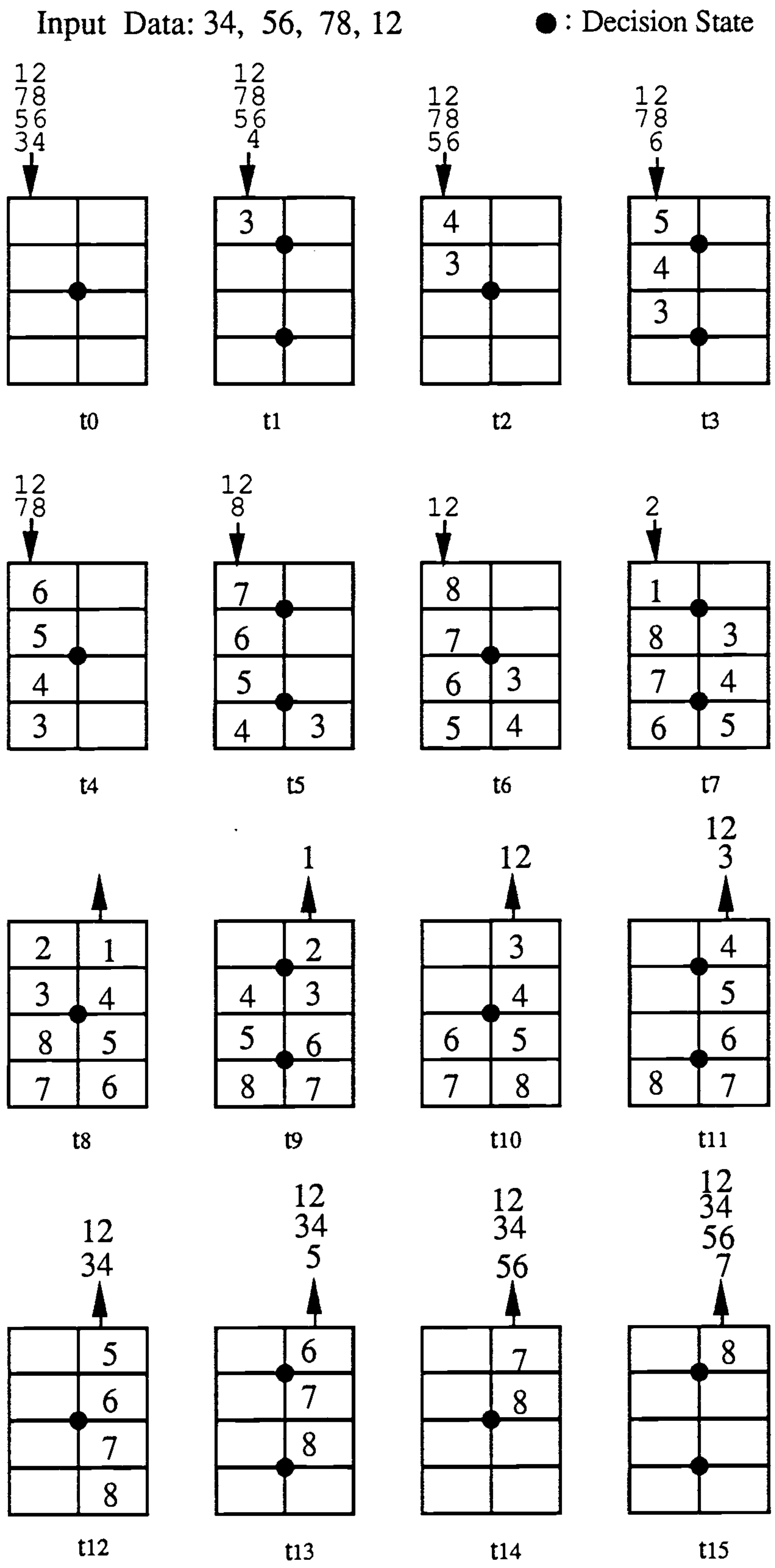


Figure 2.8 Rebound Sorter Operation

```
begin
for $j$ = 1 to $N$ in parallel
        for $i$ = 0 to $2N$-1 do
                if mod($i$,2) = 0 then          /* Even processors start */
(1)                     if $L_j \geq U_j$ then
                                $L_{j+1} = L_j$         /* vertical data movement */
                                $U_{j-1} = U_j$
                        else if $L_j < U_j$ then
                                $L_{j+1} = U_j$         /* horizontal data movement */
                                $U_{j-1} = L_j$
                        end if
                else
(2)                     if $L_j < U_j$ then         /* $L_j = U_j$ at cycle ($i$-1) */
                                $L_{j+1} = U_j$
                                $U_{j-1} = L_j$
                        else                        /* Odd processors start */
(3)                             $L_{j+1} = L_j$         /* $L_j > U_j$ at cycle ($i$-1) */
                                $U_{j-1} = U_j$
                                or
                                $L_{j+1} = U_j$         /* $L_j < U_j$ at cycle ($i$-1) */
                                $U_{j-1} = L_j$
                        end if
                end if
        end for
end for
end
```

1. Item (1) is the first decision state, item (2) is the second decision state and item (3) is the continuation state.
2. Even processors start in the first decision state(1) and odd processors start in the pass vertical state(3)

Figure 2.9 Rebound Sorter Algorithm (Algorithm 2.3)

Algorithm 2.3 (Figure 2.9) requires $O(N^2)$ time. The time complexity for N processors is

$$p(N) = N - 1 = O(N),$$

$$t(N) = \frac{O(N2)}{O(N)} = O(N)\ ,$$

and

$$c(N) = O(N^2)\ .$$

The algorithm requires $4N$ cycles to sort N keys, but this number can be reduced to $2N$ cycles if the rebound sorter is pipelined. When the number of comparisons are considered, the rebound sorter exhibits two extreme cases. For the best case, each comparator requires N comparisons to sort N keys; for the worst case, $2N$ times are required for each comparator. When the values of the half-key are the same, *i.e.*, 2411, 2412 *etc.*, the worst case can be applied. Although the sorter operates according to the worst case, the total time complexity is $O(N)$ with N processors, including the input and output times. No time penalty is incurred for the additional comparison in the hardware implementation of this algorithm. If the algorithm were implemented in software, a time penalty would be incurred.

The time complexity for sorting, in addition to data input/output time is $O(N)$ since pipelining introduces latency in the system only for a particular key. The record division and merge time for the rebound sorter is also absorbed in the sorting time through the employment of a unique buffer management scheme (analyzed in Section 4.3).

2.3 Comparison of Parallel VLSI Sorters

In the previous section, three sorting algorithms amenable to implementation in VLSI were analyzed. These algorithms can be further analyzed from the point of view of total chip area and system performance. Table 2.1 provides the performance and chip area data for the three parallel VLSI sorters [1]. Representative internal blocks from each of these sorters were synthesized on a silicon compiler in order to obtain these area and performance estimates.

The VLSI sorter (Algorithm 2.2) of Miranker *et al.* [36] has the smallest overall chip area, using $N/2$ cells to sort N keys. The parallel enumeration sorter (Algorithm 2.1) developed by Yasuura *et al.* [50] requires the largest chip area. Though the chip area of the rebound sorter is slightly larger than that of the VLSI sorter, its performance is much better, *i.e.*, the processing speed of the rebound sorter is twice as great as the VLSI sorter.

Among all of the parallel VLSI sorters analyzed to implement using silicon compilation technology, the comparators determined the critical timing path. The comparators were built using subtracter blocks included within a parallel datapath, which is a fundamental microarchitectural element contained in the silicon compiler's building block library. Other comparator structures, such as those implemented with programmable logic arrays (PLA) and random logic (or logic gates), used excessive chip area and their performance were poor due to the excessive wiring requirement.

Table 2.1 Comparison of VLSI Sorters

	Functional Blocks of 1 Sorter Cell	Area of Each Functional Block (mils x mils)	Total Area of N Cells (mils x mils)	No of Clock Cycles	Maximum Clock Frequency
Enumeration Sorter	2 16-bit data register 1 16-bit comparator 1 8-bit counter	2× 3 .60 x 115.19 52.24 x 11 .81 16.67 x 27 .98	*N* x 1903.31	2*N*	6 MHz
The VLSI Sorter	2 16-bit data register 1 16-bit comparator 2 16-bit 2-way multiplexer	2× 3 .60 x 115.19 52.24 x 11 .81 2× 3.96 x 46 .80	*N*/2 × 1807.54	4*N*	6 MHz
Rebound Sorter	2 8-bit data register 1 8-bit comparator 2 8-bit 2-way multiplexer	2x 3 .60 x 62 .75 30.89 x 11 .81 2x 3 .96 x 24 .83	*N* x 1013.27	2*N* (pipelined)	17 MHz

Note: Assumes equal area for controllers

The 8-bit comparator utilized by the rebound sorter is much faster than the 16-bit comparators used in the other sorters. In order to increase the processing speed for the 16-bit comparators, the area must be increased to twice that shown in Table 2.1. The estimated maximum frequency for the resulting high speed comparators is 6 MHz, when static carry technology is employed and 14 MHz when a precharged carry chain is used. When a precharged carry chain is implemented, it is necessary to add additional circuitry to overcome the significant timing problems which result. Therefore, the rebound sorter is the most feasible structure for VSLI implementation in terms of performance and overall chip area.

2.4 Conclusion

Three parallel VLSI sorting algorithms which have very similar characteristics were analyzed for time complexity. Although there were slight differences, each achieved O(N) time complexity with O(N) processing elements. The enumeration sorter proposed by Yasuura *et al.* [50] fails to consider the internal delay of the counter for each cell and the rearranging process time. The more the total number of processors is increased, the greater the counter delay time and the larger the counter area must be in order to reduce delay time.

The VLSI sorter proposed by Miranker *et al.* [36] has a suitable algorithm for VSLI implementation since it requires fewer processors, $N/2$ processors, than the others. However, its speed is twice as slow as the others. Considered by itself, the VLSI sorter comparator achieves only one-half the speed of its rebound sorter counterpart. To improve comparator speed, other design methods or the application of lower level VLSI CAD tools should be considered. Consequently, the rebound sorter is the most efficient in terms of processing speed and chip area. Processing speed is twice that of the others and the chip area is one-half of the enumeration sorter area and only slightly larger than the VLSI sorter. However, for the rebound sorter an external preprocessor is required to decompose the records and merge them following the sort. Implementation methods for merging two half-keys are discussed in Chapter 4.

3. THE REBOUND SORTER

In the previous chapter, three parallel VLSI sorting algorithms [18,36,50] with O(N) time complexity were analyzed in terms of processing speed and chip area and the rebound sorter was determined to be the most cost effective. In this chapter, processing elements, pipeline control, and the expandability of the VLSI implementation of the rebound sorter are discussed. Detailed design methods, including floorplanning and timing issues, are discussed as well as the functional simulation of the chip for purpose of design verification.

To build each module of the sorter, several possible design methods were considered and compared. The smallest chip area and the lowest delay times were selected and a basic processing element was designed and replicated in order to build the complete rebound sorter with the capability of sorting eight 16-bit keys. For VLSI design, delay time considerations of block modules were an important factor since the delay time affects overall system chip size and performance.

3.1 Design Considerations

To formulate a simple design process the following assumptions were necessary:

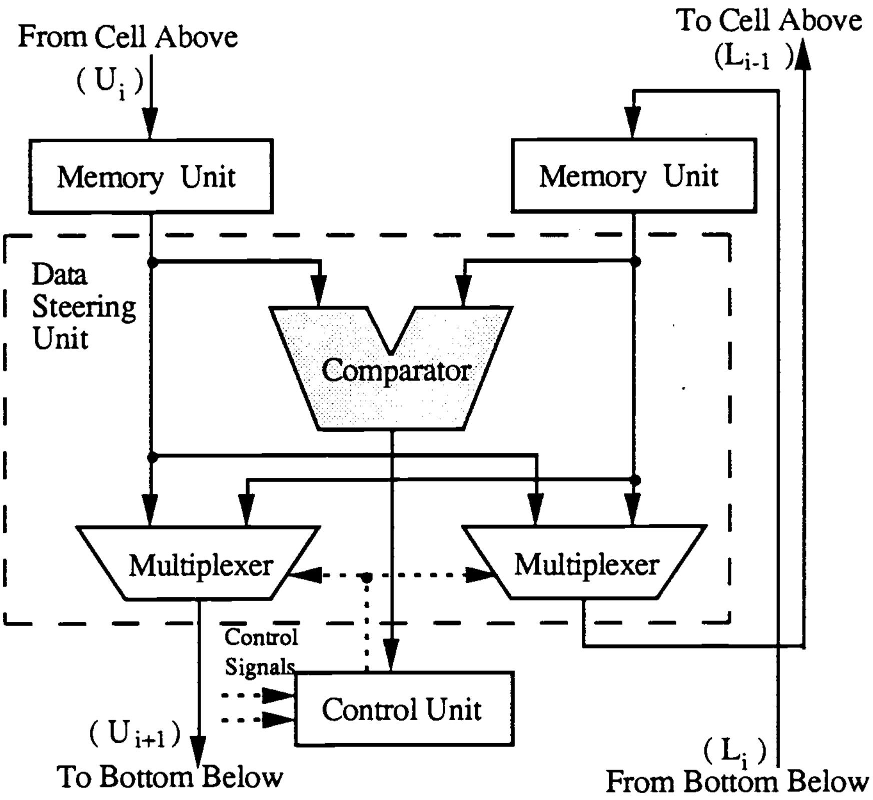
1) The order of sorting is performed such that keys emerge from the sorter in ascending order. Larger keys will descend to the bottom of the sorter upon entry, while smaller keys ascend toward the top.
2) Keys to be sorted are directed to the rebound sorter's input port and sorted keys exit through the output port.
3) The size of each key is 16 bits. In general, key size is arbitrary and this requirement can be made variable by hardware modification (refer to Section 3.9).
4) If the number of keys exceeds the capacity of the sorter, additional sorting chips can be chained; alternatively, an external merger can be used to merge numbers of sorted strings for sorting more than N keys.

To sort N keys, the rebound sorter includes N identical processing elements and one pipeline controller. Each processing element is composed of two memory units, two data steering units, and a control unit. A block diagram of the processing element is shown in Figure 3.1.a [1]. The memory unit, the comparator, and the data steering unit are designed as a single datapath module, using silicon compilation technology, to minimize chip area and reduce interconnections among the separate modules. The control unit is a finite state machine (FSM) built using a programmable logic array (PLA).

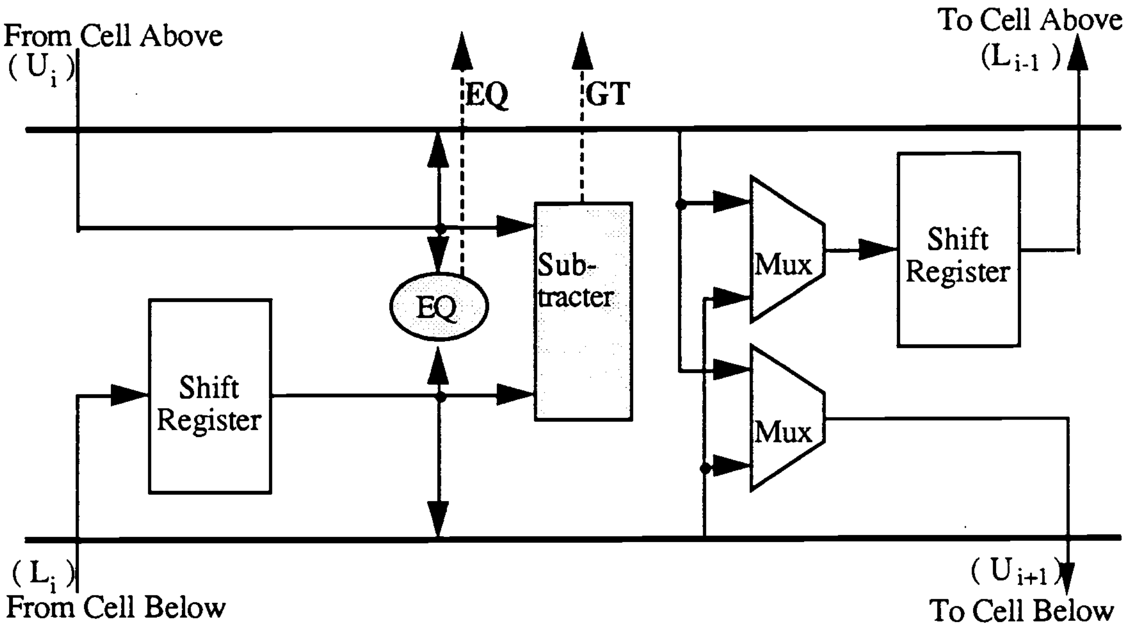
3.2 Processing and Memory Elements

The steering and memory units for the processing element are included within the same datapath module as indicated in Figure 3.1.b. The memory unit consists of two transparent latches, which retain the data for a complete clock cycle. New data is loaded into the first latch on phase A and into the second latch on phase B. The comparator is composed of a subtracter plus an equal flag unit within the datapath module. Comparator output signals are "greater-than" and "equal-to" and are provided as inputs to the PLA FSM controller during the same clock phase.

The steering unit, consisting of a comparator and two multiplexers, compares two keys, one-half of a key at a time. In functional terms, the most significant half-key of a particular key from the sorter cell above is applied to the upper left input (U_i) of the cell in question, while the most significant half-key of a second key from the cell below is applied to the lower right input (L_i) of the cell. The steering unit moves the input half-keys through the cell vertically or horizontally, depending upon their relative magnitudes as determined by the comparator. The smaller value of the two half-keys ascends to the upper right output (L_{i-1}), while the larger value of the two half-keys descends to the left lower output (U_{i+1}). If the values of the two half-keys are equal, data movement is vertical and in the next clock cycle the least significant half-keys which follow are compared. Data comparison and movement of the least significant half-keys in the second clock



a. Processing Element of Rebound Sorter Cell



b. Datapath Implementation

Fig 3.1 Block Diagram of a Processing Element

cycle are conceptually identical to those for the most significant half-keys. All controller signals are dependent upon the comparison results and multiplexers are used to change data flow direction.

3.3 Controller

The controller for the processing element consists of a finite state machine. Three possible implementation methods were considered for this unit: A pure combinational logic implementation, the use of a read only memory (ROM), and the use of a programmable logic array. The PLA approach allowed for rapid design, due to the existence of a PLA generator in the silicon compiler. The controller receives inputs from the comparator and, when pipelining is implemented, from an external controller. Based on these inputs, the controller determines the next state of the state machine, and provides multiplexer outputs to steer incoming half-keys in the proper direction during each clock cycle.

The controller state table is shown in Table 3.1 and its state diagram is shown in Figure 3.2 [1]. In Figure 3.2, each node in the diagram represents a controller state and each edge represents the state transition and its associated inputs and outputs as illustrated. The left and the right sides of the "/" are, respectively, input and output, while the dash, "-" indicates a "don't care" condition. The input signals are reset, GT (greater than) and EQ (equal to) and the output signal which controls the multiplexers is pass-vertical if output = 1 and pass-horizontal if output = 0.

Table 3.1 Controller State Table

	Next State, Output *			
Present State	GT•EQ 00	01	11	10
D1 (State-0)	**PH**,0	**D2**,1	X	**PV**,1
PV (State-1)	**D1**,1	**D1**,1	X	**D1**,1
PH (State-2)	**D1**,0	**D1**,0	X	**D1**,0
D2 (State-3)	**D1**,0	**D1**,1	X	**D1**,1
Reset	Even PE → **D1** Odd PE → **PV**			

D1: First Decision
PV: Pass Vertical
PH: Pass Horizontal
D2: Second Decision
X: Don't cares

* 1: Pass Vertical Output
0: Pass Horizontal Output

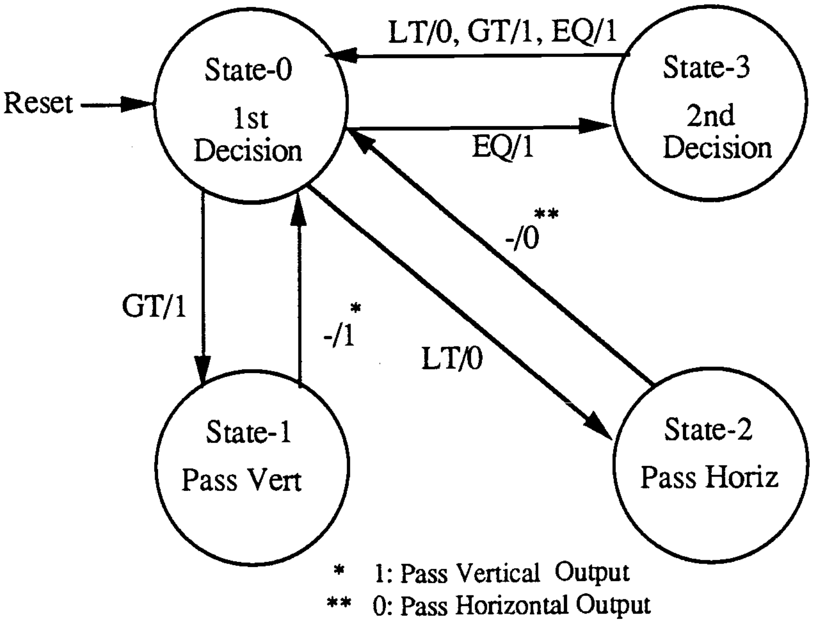


Figure 3.2 Controller State Diagram

The controller has four states as follows:

1) At State-0 (State-D1), the initial decision state, comparison of the two half-keys is performed, *i.e.*, the most significant halves of the pair of keys in question. One half-key is provided at the upper left input (U_i), while the other half-key appears at the lower right input (L_i). These are moved vertically if the upper left half-key (U_i) is greater than or equal to the lower right half-key (L_i). When the upper left input (U_i) is less than lower right input (L_i), the half-keys are moved horizontally.
2) State-1 (State-PV) is entered when the half-key (U_i) is greater than the half-key (L_i). In this case, the second or least significant half-keys are also passed vertically, a process which occurs because the initial decision has already determined which half-key is larger and no additional magnitude decisions are required.
3) State-2 (State-PH), in which the second or least significant half-keys are passed horizontally, is entered when the upper left input half-key *(L_i)*, is less than the lower right half-key *(U_i)*.
4) State-3 (State-D2) is entered if the upper left input half-key (U_i), is equal to the lower right half-key (L_i). This is a second decision state, which compares the least significant half-keys to determine proper data flow direction.

All processing elements operate synchronously. The PE control units, shown in Figure 3.3, are initialized as follows:

1) Even PEs from the input stage, *e.g.*, PE0, PE2, are initially set to State-D1 when the reset signal is received.
2) Odd PEs from the input stage, *e.g.*, PE1, PE3, are initially set to State-PV when the reset signal is received.

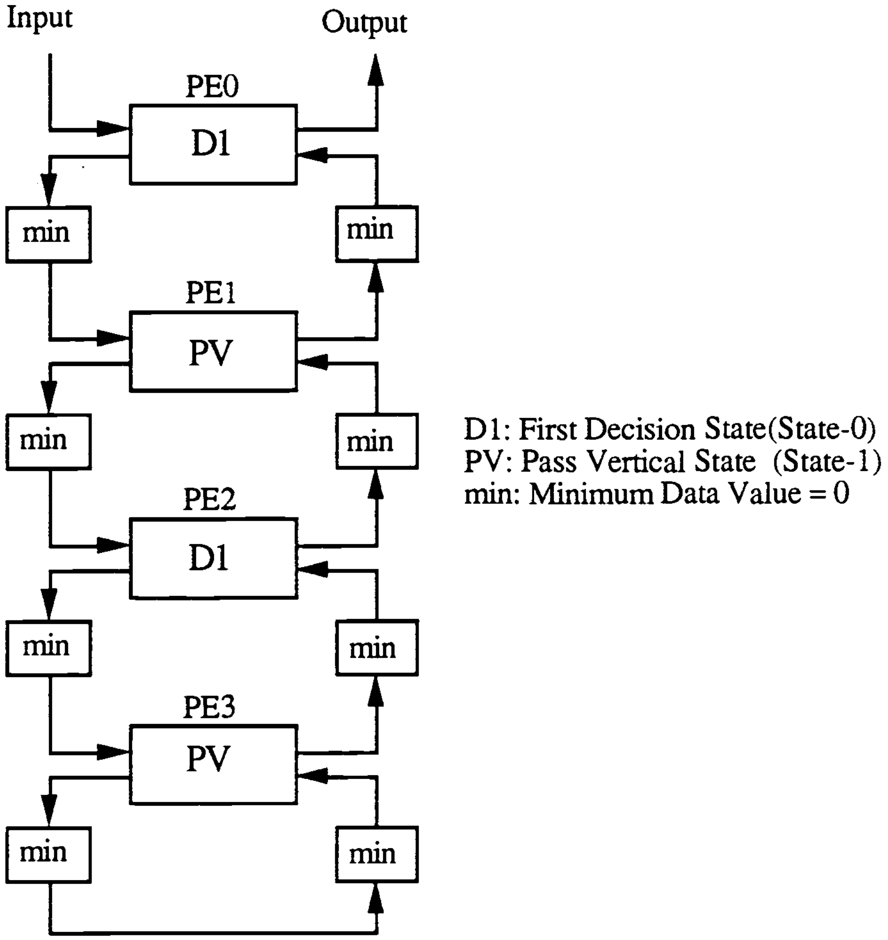

Input
Output
PE0
D1
min
min
PE1
PV
min
min
PE2
D1
min
min
PE3
PV
min
min
D1: First Decision State(State-0)
PV: Pass Vertical State (State-1)
min: Minimum Data Value = 0

Figure 3.3 Initial State of the Rebound Sorter

3.4 Pipeline Control

The rebound sorter is capable of holding and processing *N* keys, producing sorted strings of the same length at its output. In order to maximize performance, input and output operations must take place simultaneously. This requires presentation to the sorter of new, unsorted keys while the previously sorted keys associated with earlier input data are emerging at the output. This process is referred to as pipelining. An example of the pipelined rebound sorter operation is provided in Figure 3.4.

In the pipelined rebound sorter, it is assumed that the sorter is filled with data currently in the process of being sorted. New unsorted keys enter as input and descend through the system without being sorted for 2*N* clock cycles. On completion of 2*N* cycles, the new half-keys are compared, while the remaining half-keys within the sorter associated with an earlier input sequence continue to emerge, undisturbed, from the sorter's output.

A specialized controller provides the necessary control signals to each sorter cell for the correct implementation of pipelining, which is implemented as follows:

1) First, the incoming *N* keys are isolated from the current set of *N* keys

2) When the new incoming keys descend to the bottom cell of the sorter, comparison is initiated and the previously sorted data ascends vertically to the output port.

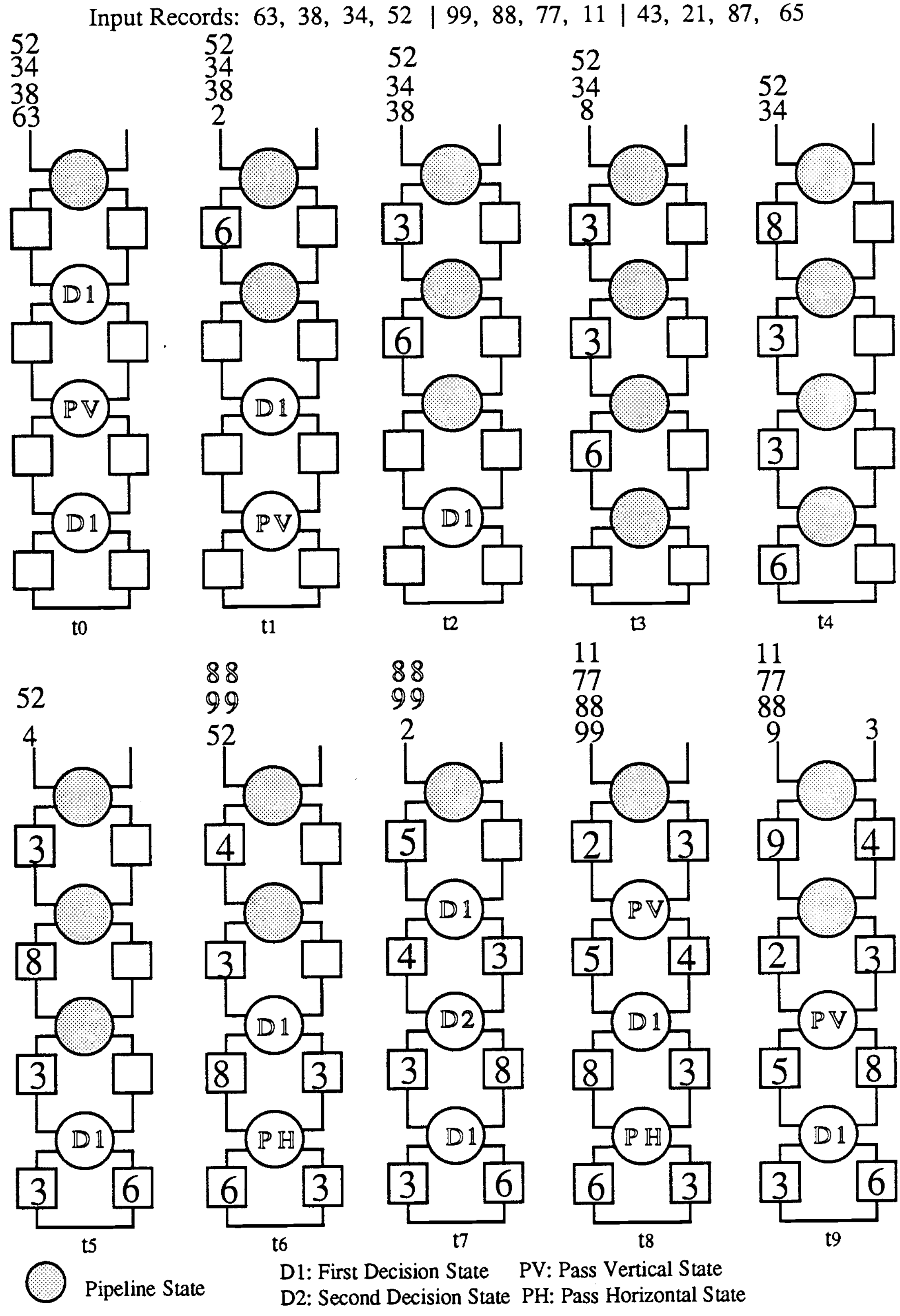


Figure 3.4 Pipelined Rebound Sorter Operation

Input Records: 63, 38, 34, 52 | 99, 88, 77, 11 |43, 21, 87, 65

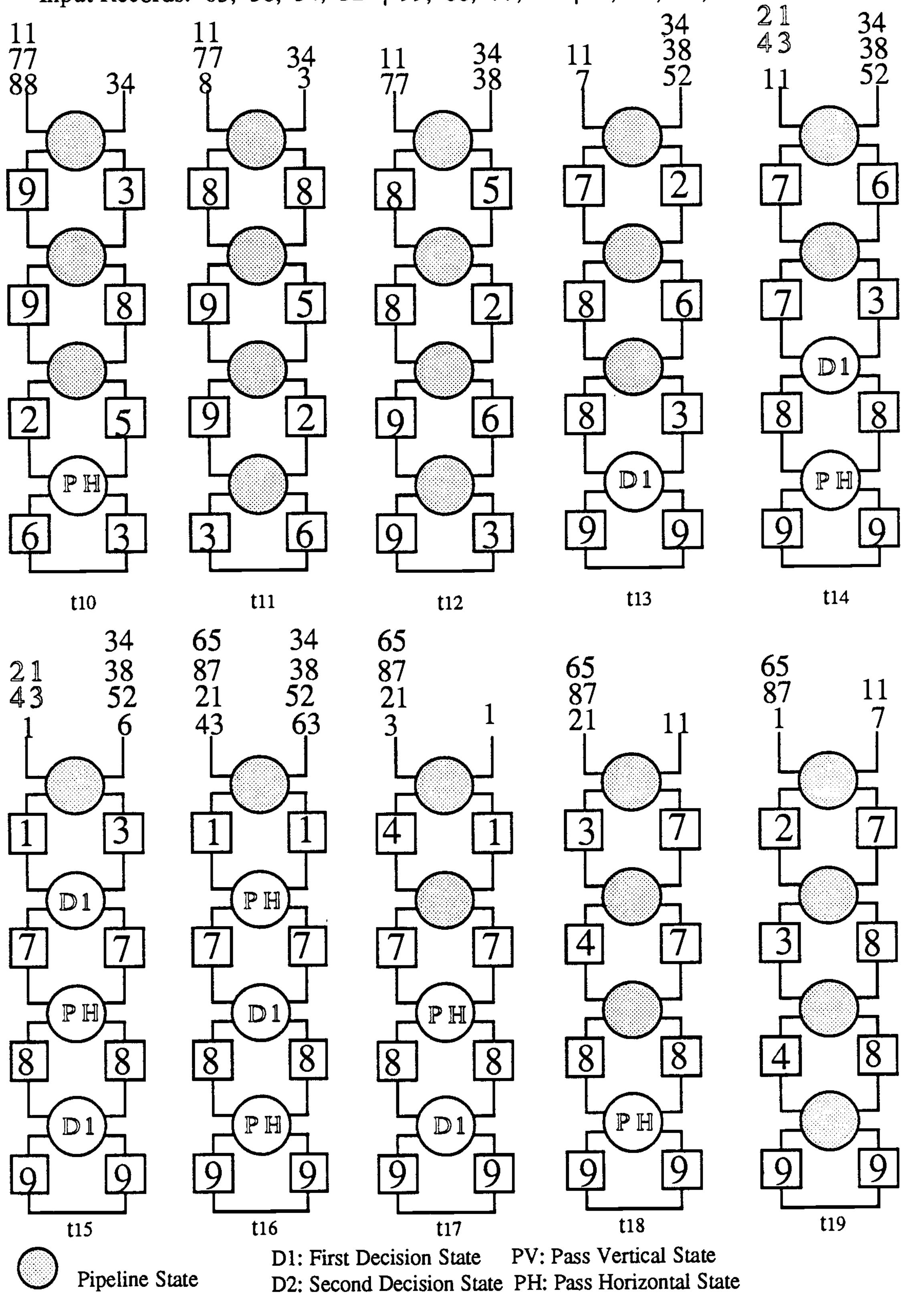


Figure 3.4 Pipelined Rebound Sorter Operation (*Continued*)

3) The pipeline control signals produced by the specialized PLA FSM to assure correct operation during pipelining are illustrated in Figure 3.5.a.

The PLA pipeline control signals are implemented in two stages. First a pipeline FSM, as indicated in Figure 3.5.b produces a control sequence. A pipeline pattern generator then translates this sequence into the appropriate pipeline control pattern. For ease of design and maintenance, this method is preferable to the combination of two blocks in a single FSM. The pipelined state machine controller may be expanded if it is desired to chain several sorters together. The pipelined FSM state table and diagram are shown in Table 3.2 and Figure 3.6, respectively.

3.5 System Expansion

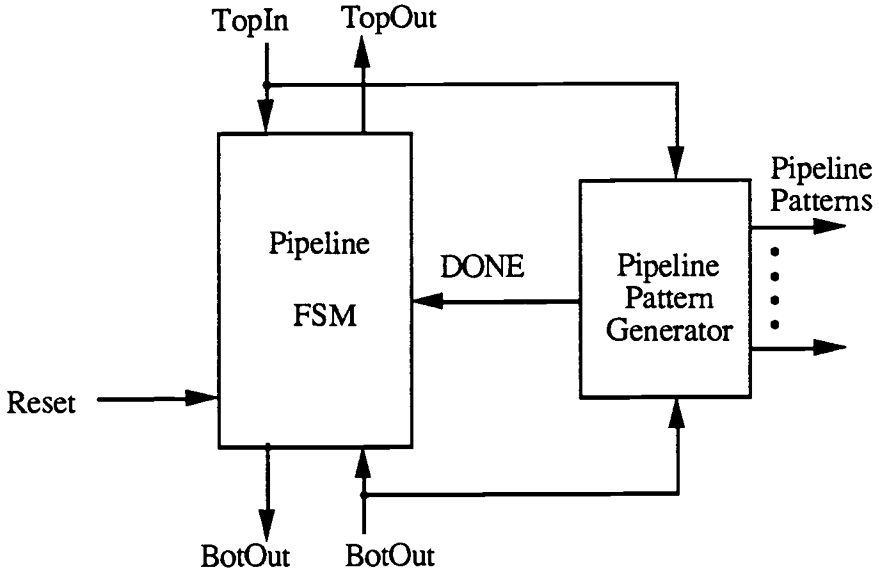
Since the sorter chip is capable of maintaining only *N* keys internally, large files must be sorted using multiple sort chips. To provide this expansion capability, the rebound sorter of Chen *et al.* [18] was modified to encompass multiple chips chained in series. The design requirements resulting from this decision include the following [1]:

1) Pipeline control signals have the capacity to pass across multiple chained chips.

	t0	t1	t2	t3	t4	t5	t6	t7
PE0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
PE1	0	**1**	**1**	**1**	**1**	**1**	**1**	0
PE2	0	0	**1**	**1**	**1**	**1**	0	0
PE3	0	0	0	**1**	**1**	0	0	0

1: Pipeline control signal
0: Normal operation

a. Pipeline Control Signal Patterns

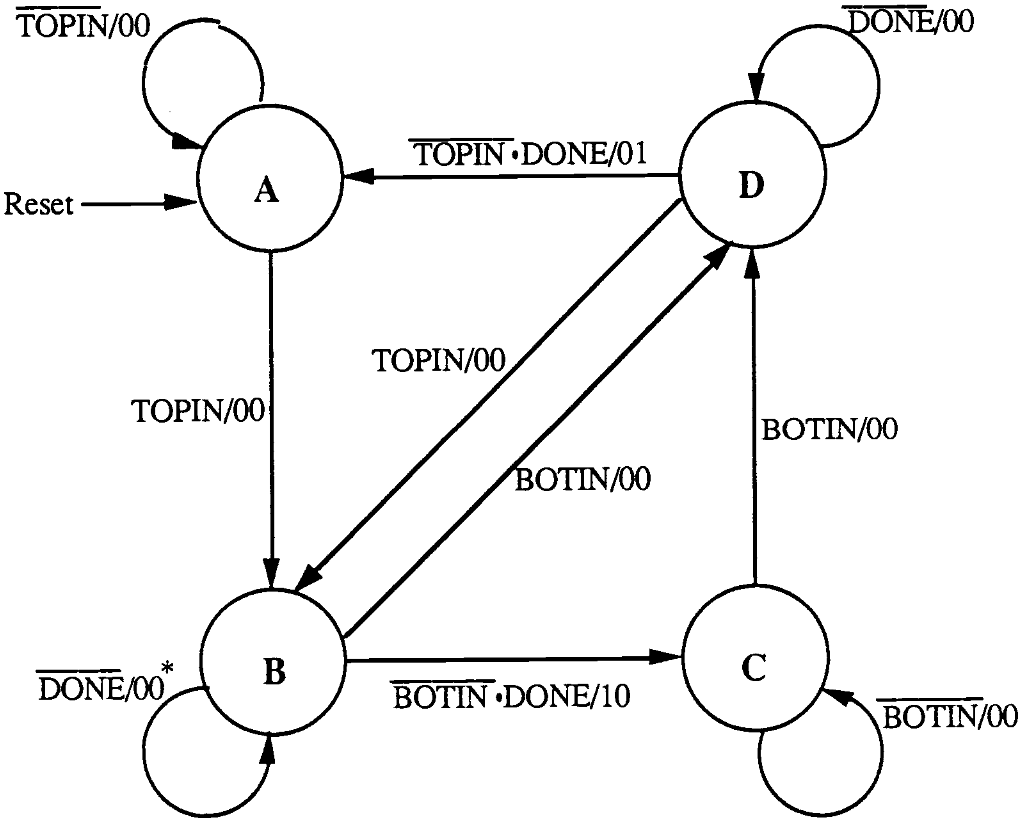


b. Pipeline Control FSM Block Diagram

Figure 3.5 Pipeline Control Block Diagram and Signal Patterns

Table 3.2 Pipeline Control FSM State Table

	Next State, Output (BOTOUT, TOPOUT)							
	DONE = 0				DONE = 1			
	TOPIN •BOTIN				TOPIN •BOTIN			
Present State	00	01	10	11	00	01	10	11
A	A,00	A,00	B,00	B,00	A,00	A,00	B,00	B,00
B	B,00	D,00	B,00	D,00	C,10	D,00	C,10	D,00
C	C,00	D,00	C,00	D,00	C,00	D,00	C,00	D,00
D	D,00	D,00	B,00,	B,00	A,01	A,01	B,00	B,00


$\overline{\text{TOPIN}}$/00
$\overline{\text{DONE}}$/00
$\overline{\text{TOPIN}}$•DONE/01
Reset
A
D
TOPIN/00
TOPIN/00
BOTIN/00
BOTIN/00
$\overline{\text{DONE}}$/00*
B
C
$\overline{\text{BOTIN}}$•DONE/10
$\overline{\text{BOTIN}}$/00

*Ouput Signals = BOTOUT •TOPOUT

Figure 3.6 Pipeline Control FSM State Diagram

2) All chained chips are aware of their own position in the chain, *i.e.*, chips at the top of the chain perform I/O operations in correct order and chips at the bottom of the chain automatically connect their lower-left output to their lower-right input.

A block diagram of the complete VLSI rebound sorter chip with expansion capability is shown in Figure 3.7 [1]. The control unit recognizes the expansion by triggering the control signals, which are described as follows:

1) The reset signal initializes all chip controllers simultaneously, including the pipeline control FSM.
2) The expansion control signal determines the source of the data applied to the lower processing element (cell) of the sorting chip. Data input originates either from another chip or from its own lower-left output port.
3) The top/bottom pipe control expansion input/output signals are used to enable pipelined pattern generation for a chain of chips.

The input and output of the data bus is processed through an expansion multiplexer as follows:

1) When the expansion control signal is low, the lower cell input is routed directly from the lower-left chip cell.
2) When the expansion control signal is high, lower cell input is routed from another chip.

3) When the sorting chip is the lowest of the chained chips, the bottom input and output of the pipe control expansion signals are connected directly.

4) When the sorting chip is the highest chained chip in a series, the top input and output of the pipe control expansion signals are connected directly.

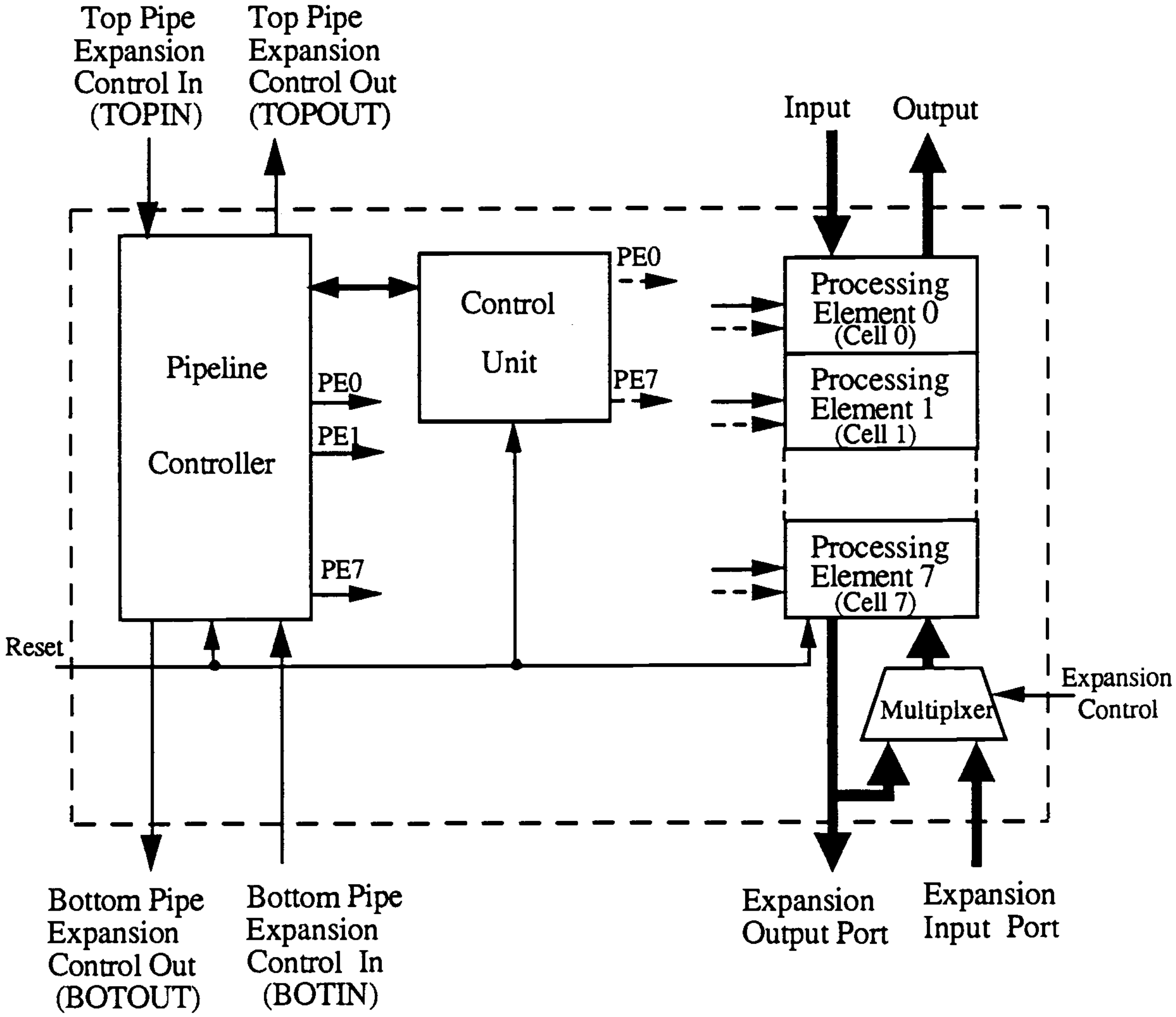

Top Pipe Expansion Control In (TOPIN)
Top Pipe Expansion Control Out (TOPOUT)
Input
Output
Pipeline Controller
Control Unit
PE0
PE1
PE7
Processing Element 0 (Cell 0)
Processing Element 1 (Cell 1)
Processing Element 7 (Cell 7)
Reset
Multiplxer
Expansion Control
Bottom Pipe Expansion Control Out (BOTOUT)
Bottom Pipe Expansion Control In (BOTIN)
Expansion Output Port
Expansion Input Port

Figure 3.7 Rebound Sorter Block Diagram

3.6 Silicon Compilation

The Genesil silicon compiler is the computer-aided design (CAD) tool for VLSI design used in implementing this architecture. Silicon compilation allows a hierarchical design methodology to be employed whereby large systems are decomposed into datapath modules, general modules, blocks (*e.g.* RAM, ROM, PLA, I/O Pads, *etc.*) and random logic. Both timing verification and sophisticated logic simulation capabilities are provided by this CAD tool.

The rebound sorter shown in Figure 3.7 is composed of the Genesil silicon compiler building blocks [23]. The basic sorter processing element, steering unit and memory unit, was implemented with a parallel datapath module, resulting in the reduction of chip area and in increased performance in comparison to other possible logical implementations. The memory units were also included within the datapath module to reduce chip area, and the sorter cell and the pipeline controllers were implemented with PLA finite state machines.

The total die area, including the power ground, clock, and input and output data pads consumes 5801 microns by 5283 microns (228 mils by 208 mils) in 2-micron CMOS technology. This chip can be packaged in a 48-pin pin-grid array, although only 40 of the pins are utilized. Since the chip core area is 159.2 mils by 146.8 mils, the chip potentially could be expanded to include up to 32 processing elements within an area of 400 mils by 400 mils.

3.7 Functional Simulation and Timing

Simulation was performed on functional models constructed from the compiler block specification tools. Successful design verification required that both switch-level and functional models pass the same set of test vectors. Using a synchronized two-phase clocking system, the individual modules, PLAs, and blocks were simulated, verifying their functionality and performances. When a basic module or block was designed, functional verification was used to debug each element in the design hierarchy prior to moving to a higher design level. Top-down architectural definition and decomposition into an appropriate microarchitecture, followed by bottom-up implementation and verification, proved to be a successful methodology.

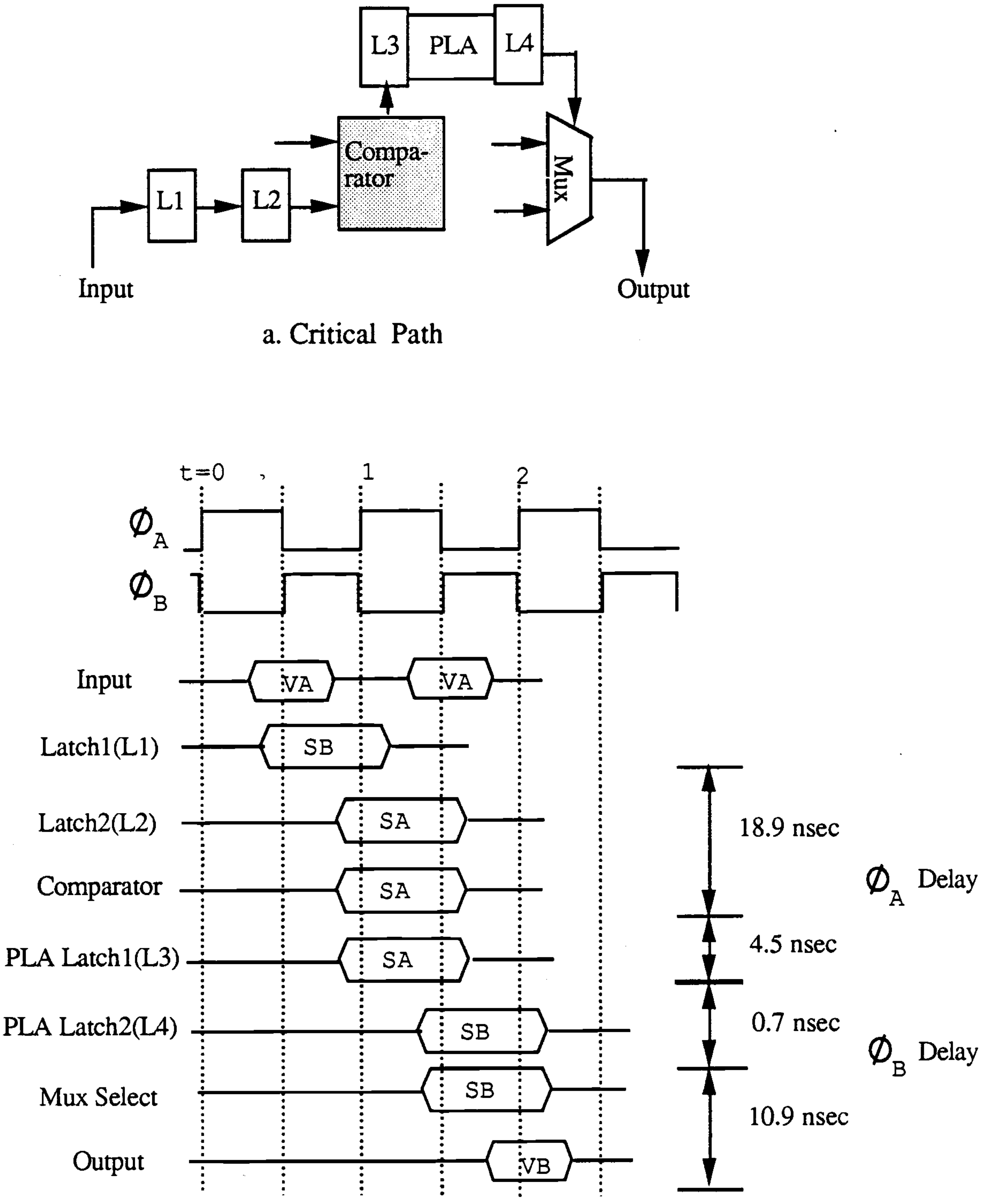
Four test verification procedures were conducted for: (1) A complete sorted input sequence (ascending order); (2) A complete sorted sequence (descending order); (3) An unsorted random sequence;, and (4) A sorted sequence consisting of the most significant half-keys of keys with the same digits, *e.g.*, 1123 and 1122. The test sequences were recorded in a set of test vector files. All of the above sequences verified the operation of the rebound VLSI sorter at the chip level.

Timing analysis provided a design consistency check (type checking) against the block level timing attributes used in the logic design process. Using a two phase, non-overlapping clocking method, the timing references for all logically associated clocked devices were checked. All of the data and control signals for a block instance were classified as either valid, stable, or propagate with respect to the phase pair associated with the specific block. This verification tool also automatically calculated worst-case signal propagation delays through all timing paths, plus the input setup and hold times. It also provided the maximum attainable system clock rate. The rebound sorter design achieved a performance rate of 17M bytes/sec. The critical path and major block delay times of the rebound sorter chip are shown in Figure 3.8.

3.8 Floorplanning and Routing

Floorplanning consists of the placement of design objects on the chip, the assignment of routing channels, and the implementation of appropriate external connections. Floorplanning was conducted in three steps: placement, fusion, and pinout. Placement specified the initial location of objects, fusion defined the wiring channels, and pinout specified the external inputs and outputs.

Design objects for the rebound sorter consist of eight processing elements, one controller, and one pipeline controller. To reduce chip area and propagation delays and minimize wiring complexity, the 10 modules were given an adjacent placement and aligned with their respective signal ports. From this position they may be turned, rotated, and/or flipped. A number of trial-and-error attempts were repeated in order to obtain the final placement. Upon completion of floorplanning, the core size without pads was 159 mils by 146 mils.



a. Critical Path

b. Major Delays

Figure 3.8 Critical Path and Major Delays of the Rebound Sorter

3.9 Architectural Extension

The rebound sorter has an 8 keys, 16-bit per key sorting capacity, as noted previously. In the architecturally expanded version of the rebound sorter, the key length can be increased in 16-bit increments. This scheme is shown in Figure 3.9. The performance of the expanded sorter increases because of the additional parallelism provided. At the same time, the clock rate decreases slightly with expansion due to the carry propagation across multiple chips. The only special design requirement is the addition of an external controller. In order to use variable key lengths, a dynamic method of configuration may be considered. In systems where a small number of large keys are required, the sorter subsystems would operate in parallel; in systems where a large number of small keys are to be sorted, the sorter subsystems could be configured serially.

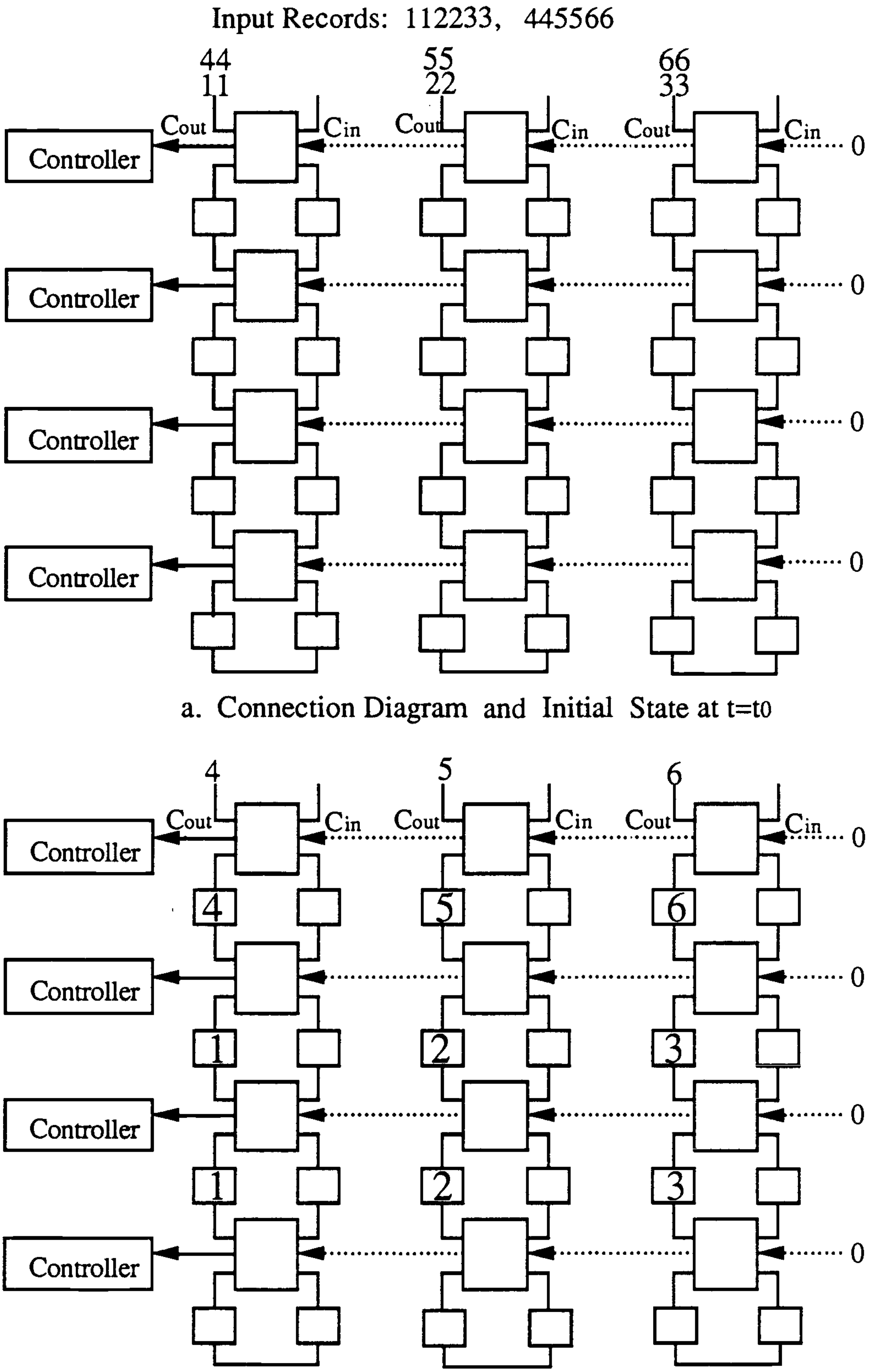
3.10 Conclusion

The proposed design is only one illustration of several possible methods of implementation of the rebound sorter architecture, as modified and extended. The feasibility of implementing the proposed sorting engine using silicon compilation technology has been demonstrated.

A simple and efficient processor element design for this sorter has been achieved. The pipeline control unit consists of a pipeline

FSM and a pipeline pattern generator, a methodology which makes for ease of maintenance and provides for future expansion. The overall size is less than one-half the area of Bate's design [5], achieving a speed which is three times as fast.

The proposed chip is capable of sorting eight 16-bit keys. It can readily be expanded to handle larger files, through either the use of smaller geometry technology or chaining, or in conjunction with an appropriate external hardware merger structure. The increase of the key length was discussed. The proposed design is based upon 16-bit data, but the length of the key can be extended to much larger sizes, dependent upon specific processor application speeds.



a. Connection Diagram and Initial State at t=t0

b. State after t=t3

Figure 3.9 Architectural Extensions

4. EXTERNAL PARALLEL SORTING

External sorting is a process based on the repeated merging of a number of sorted strings. External sorting is appropriate when the files to be sorted are too large to fit in main memory. Mass storage devices such as disks or magnetic tapes are used to hold the large volumes of data involved in the process. The basic objective of the merge sort, a form of external sorting, is the generation of a single large sorted string from two or more sorted data strings. The algorithms and techniques for merging two sorted strings are less complicated than those for sorting single files since the merge input are a sequence of sorted data. The parallel sorters analyzed in Chapter 2 use N processors to sort N records, where as the external sort-merge sort uses only $\log_m N$ processors, where m is the order of merge.

A number of performance evaluation studies of serial and parallel external sorting have been conducted [11,12,21,22,24,25,47]. Performance modeling and analysis of either the serial or parallel external merge sort is a complex process since access time, disk scheduling, and the file distribution and organization of the mass storage devices is often necessary. External sorting algorithm analysis must also consider the architectural characteristics of the mass storage devices used in order to obtain satisfactory results. A simple I/O access model is analyzed in Chapter 5.

This chapter begins with an analysis of external sorting techniques. In particular, the characteristics of the external parallel merge sort are considered since high performance algorithms for the internal merge sort have already been thoroughly analyzed for use with parallel computers [6,22]. Several parallel merge sort algorithms are discussed to implement and methods of memory management are considered. Following this analysis, the VLSI implementation of a 4-way merge sorter is considered

4.1 VLSI Merge Sort

Even [21] proposed two methods for parallel tape-sorting based upon a serial, two-way, external merge sort algorithm. In the first method, all processors must initiate the merge simultaneously, sorting a group of files in parallel. In the best case, this method performs $(N/P)\log N + 3N$ unit times using $P \leq \log N$, where N is the number records to be sorted and P is the number of processors. For the second method, processors were added one at a time to implement a pipelined sorting algorithm. This resulted in the use of $\log P$ processors and 4 x $\lceil \log P \rceil$ tape units for the sort of $3.2\lceil \log N \rceil$ write cycles, where $N > P$. These sorting methods were complicated by the necessity of rewinding the tapes before they could be read.

Todd [47] formulated a pipelined merge sort technique based on Even's algorithm [21]. More sophisticated hardware, including bubble memory and faster RAM of less complex design, was used. The algorithm allowed $\log N$ processors to sort N records in just over

$2N$ write cycles. This pipelined merge sorter was implemented by Kitsuregawa *et al.* [25] in 1983, sorting data streams at 3M bytes/sec.

In 1982, Bitton [9, 11] analyzed a variety of parallel sorting algorithms and architectures. The best performance was achieved by a parallel binary merge algorithm, which was a modified parallel two-way merge sort based upon the Even algorithm. In 1985, Takagi [45] applied Todd's pipelined merge sort algorithm to disks with initial sorters. The idea was to provide IBM disk storage systems with internal intelligent disk processor subsystems capable of performing sorting directly. Takagi noted a number of limitations of these subsystems, including problems associated with the synchronization of data transmissions and avoidance of I/O latency time.

4.2 Analysis of the External Sortings

The merge sort, a form of external sorting, is suitable for hardware implementation. This approach may readily be pipelined and can achieve a throughput rate is O[N]. The processing overhead for this system is only that associated with pipeline latency. Separate input and output ports are assumed. The most difficult factor in implementing the merge sort is buffer management. Buffering is required since it is used for storing sorted data between each of the merge steps. In order to produce an economical hardware system, it is important to synthesize an efficient scheme of merge buffer management. Techniques for efficient buffer management methods are discussed in Section 4.3. For purposes of analysis, it is assumed that the total number of records to be sorted is always a power of m, for an m-way merge. If the number of records is less than a power of m, dummy records are added to satisfy this requirement. Two merge sort algorithms are analyzed: Bitton's [9] parallel binary merge sort and Todd's [47] pipelined merge sort.

4.2.1 Parallel Binary Merge Sort

The algorithm proposed by Bitton for implementation on disk requires a binary tree connection between processors, as shown in Figure 4.1 [9,11]. The disk storage system was modified by simplification of a moving head disk. In 1988, Beck [6] applied this algorithm to a backend multiprocessor built around a fast packet bus, resulting in a system which was judged to be more cost-effective.

The binary merge sort utilizes a binary tree of merge processors with a sequence of N length and P leaf processors, where is N and P are a power of 2. The binary tree has a total of $2P-1$ processors and $1+\log P$ levels. The algorithm has three stages: suboptimal, optimal and post-optimal. Each processor has only two record buffers and the input sequence of length N is initially assigned to P leaf processors with subsequences of $N/2P$ length. When the leaf processors generate an initial record, it is transferred to the parent processor, whereupon the parent processor merges the record and transfers it to its own parent processor. Processors for i^{th} levels function in 2^i merge operations. The sequence is shown Figure 4.2 (Algorithm 4.1).

The following assumptions were observed for the performance analysis of a synchronous and pipelined parallel design: (1) All processors at the same level of the tree execute their tasks concurrently; (2) all processors, with the exception of the root processor, send each record directly to their parent processors, rather than writing a record to a mass storage device; and (3) all processors at the same level of the tree start and stop merge operations simultaneously. The operational sequence is as follows:

1) An N record input sequence is distributed to P leaf processors, each processor holding $N/2P$ records. This step takes N time units.
2) Each processor of P leaf processors performs $\log(N/2P)$ phases of the serial merge operations for N/P records during the suboptimal stage. The merging time for each processor is N/P.
3) At the i^{th} level, every processor merges two sorted runs of the size 2^i records into single sorted runs of the size 2^{i+1} records.

The total time for one synchronous merge stage of size 2^{i-1} is 2^{i+1}-2 time units and $N/(2^iP)$ merge pairs are required for synchronous processing. For example, processors at the second level merge $N/(2P)$ pairs of two-record size into a single four-block run with $N/(4P)$ pairs and six time units. Thus, the entire process takes

$$\sum_{i=1}^{\log(\frac{N}{2P})} \frac{N}{(2^iP)}(2^{i+1}-2) = \frac{2N}{P}\log\frac{N}{2P} - \frac{2N}{P} + 4 .$$

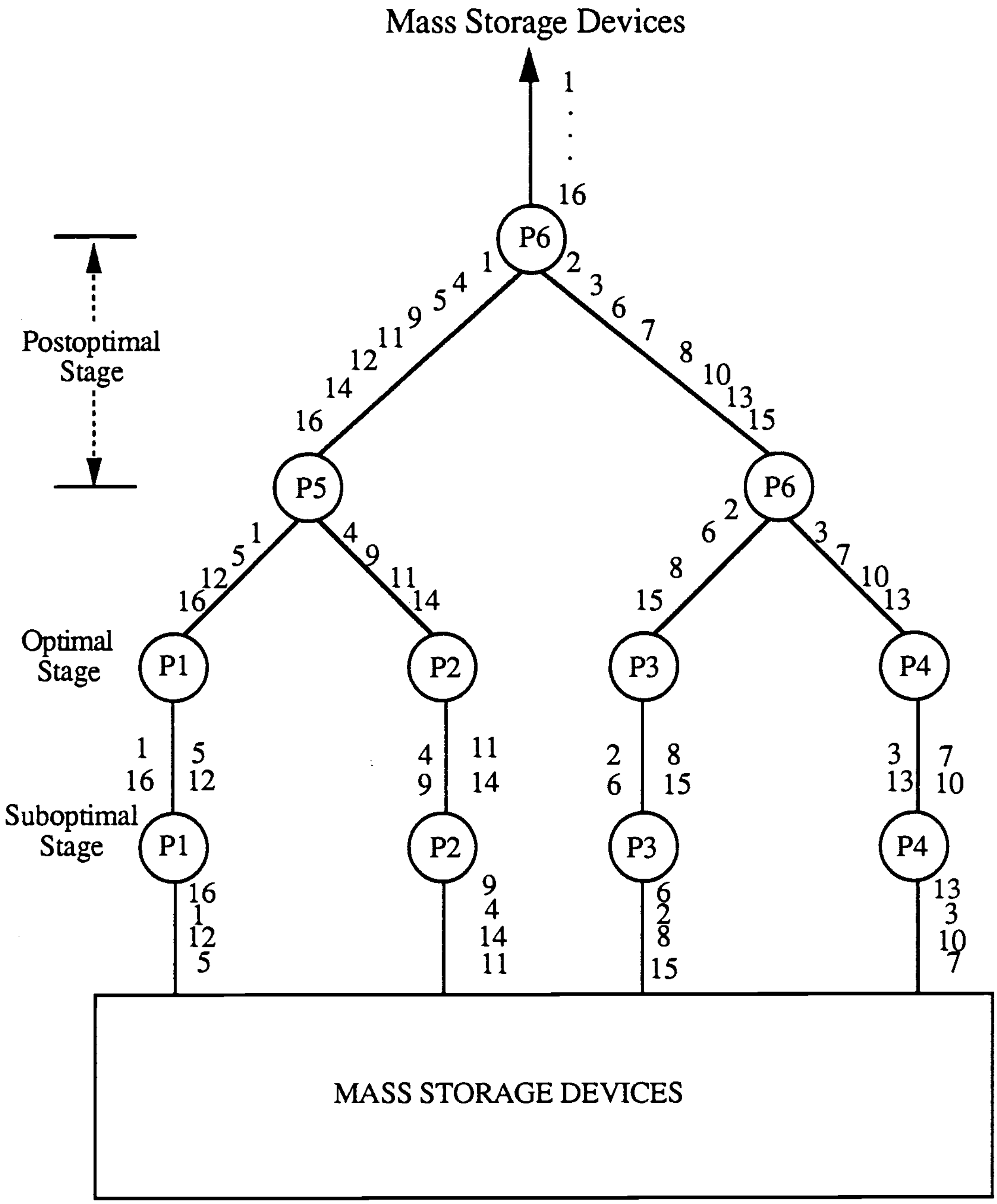


Figure 4.1 Binary Merge Sort

```
begin
Divide $N$ records into $N/2P$ blocks.
for $i$ = 1 to log $N/2P$ do in parallel
    for all processors at level i do in parallel
        merge two sorted runs of size $2^{i-1}$
        into a single sorted run of size $2^i$
    end for
end for
end
```

Figure 4.2 Binary Merge Sort Algorithm (Algorithm 4.1)

The total time is

$$
\begin{aligned}
t(N) &= N + \frac{2N}{P} \log \frac{N}{2P} - \frac{2N}{P} \\
&= \frac{2N}{P} \log \frac{2N}{P} + N - \frac{2N}{P} \\
&= O(N) + O(\frac{N}{P} \log \frac{N}{P}),
\end{aligned}
$$

and the number of processor is

$$p(N) = 2P - 1 .$$

When the input sequence distribution buffers are ignored in the first step, the number of buffers is

$$b(N) = 2(2P-1) .$$

Thus, the binary merge sort is

$$c(N) = (2P - 1)\,(O(N) + O(\frac{N}{P}\log\frac{N}{P}))$$

$$= O(NP) + O(N\log N)\,.$$

When the binary merge sort has $P = \log N$ processors, total complexity is optimal. The maximum delay time requires the same as the number of tree levels after the initiation of the merge step.

4.2.2 Pipelined Merge Sort

The algorithm proposed by Todd [47] for implementation in bubble memory or fast RAM was based on of Even's second algorithm [21] for improved sorting performance. In contrast to the connected binary tree structure of the parallel binary merge sort algorithm considered in the previous section, this algorithm is a cascaded connection of merge processors. In 1985, Takagi [45] applied this algorithm to a hardware sort-merge system intending to add sorting capability to a disk subsystem. First the two-way merge is analyzed and an *m*-way merge operation is then discussed. A block-merge sort algorithm is also discussed.

4.2.2.1 Pipelined Multi-Way Merger

Assume that each processor within a structure of $\lceil\log N\rceil + 1$ processors can conduct two-way merge operations. A serial two-way merge operation is carried out in $\lceil\log N\rceil + 1$ steps, and a sorted sequences of records created at each step as follows:

1) Upon completion of the first step, N records are split into two groups, each consisting of $N/2$ records.
2) Individual records are merged into streams of paired records.
3) The pairwise records are merged into four-record streams.
4) Following the i^{th} steps, the length of the data stream is 2^{i-1}.
5) Following $\lceil \log N + 1 \rceil$ steps, all N records are merged in a single sorted stream.

The structure of the merge sorter is shown in Figure 4.3, an example of the pipelined merge sort operation is illustrated in Figure 4.4, and the algorithm is presented in Figure 4.5.

For this algorithm, input and output time are absorbed into the merge operations. To sort $N= 2^k$, $k+1$ processors $(P_0,P_1,\ldots,P_k)$ are required. The output from the processor P_i is 2^i records with 2^{k-i} sorted sequences. Processors P_i and P_{i+1} are connected by two buffers, B_{2i+1} and B_{2i+2}. Processor P_{i+1} initiates merging operations as soon as there is an input sequence of the length 2^i on one of its two input lines, and a sequence of length one on the other, *i.e.*, 2^i+1 cycles after P_i. If the processor P_0 is initiated in cycle 1, then P_i initiates merge processing of its first input

$$1 + \sum_{j=0}^{i-1} (2^j+1) = 2^i + i$$

cycles later.

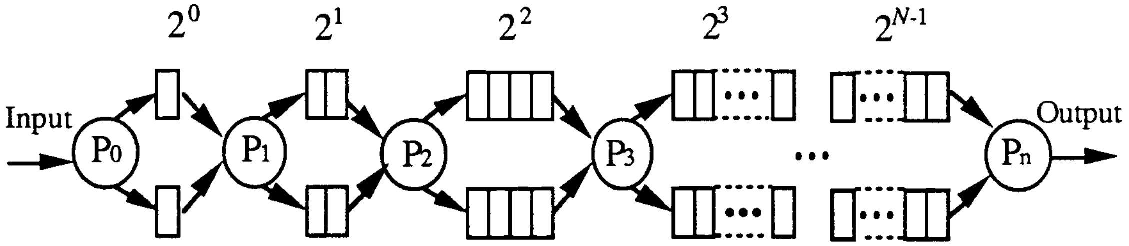
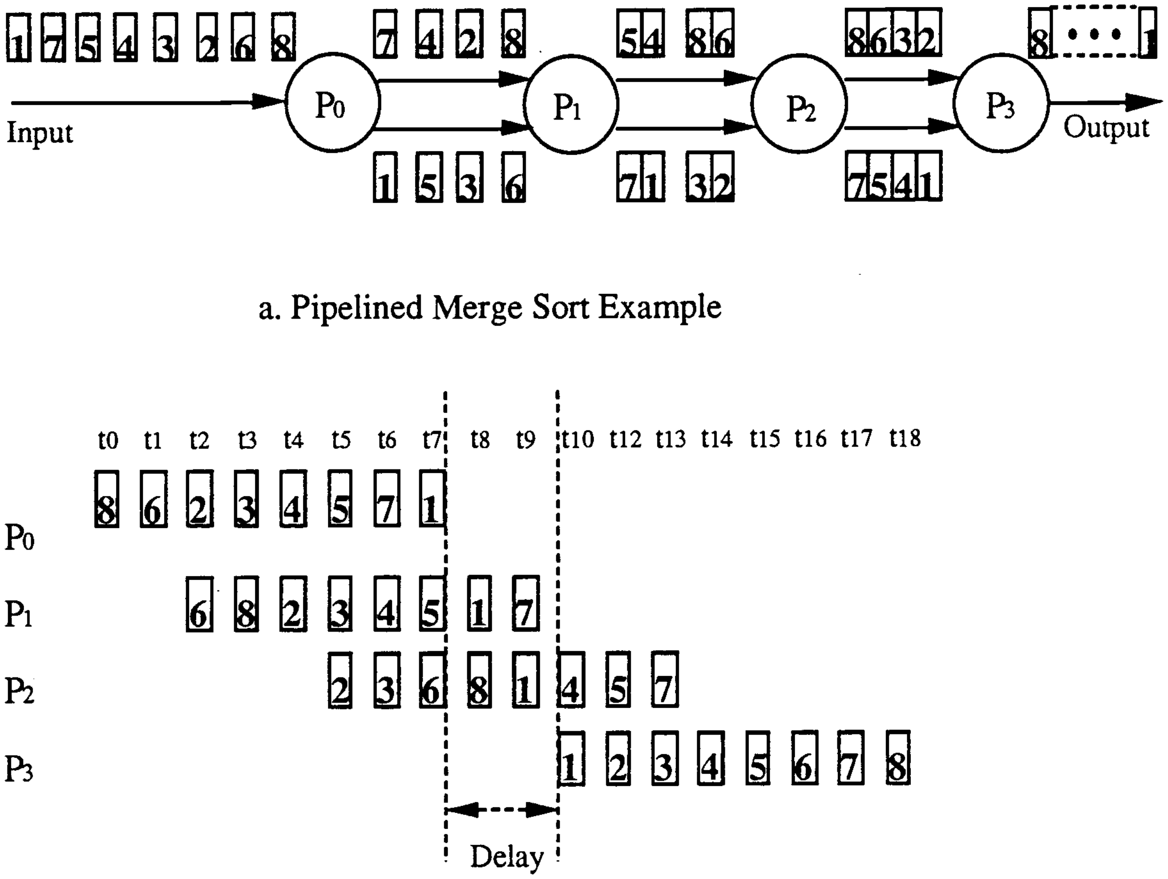


Figure 4.3 Pipelined Merge Sort Structure



a. Pipelined Merge Sort Example

b. Pipeline Timing

Figure 4.4 Pipelined Merge Sort Operation

```
begin
for $i$ = 1 to $N$
        split records into 2 alternate groups
end for
for $i$ = 2 to log $N$   in parallel
        while $k \leq N$ do
                if the 1st buffer in $P_{i-1}$ contains $2^{i-2}$ records and
                        the 2nd buffer contains 1 record
                then for $j$ = 1 to $2^{i-1}$ do
                                compare two records in the buffer and
                                move the smaller one to a Buffer.of $P_i$
                        end for
                else
                        move the record to the first buffer of $P_{i-1}$
                end if
        end while
        If the 1st buffer in $\mathrm{P}_{\log N}$ contains $2^{\log N-1}$ records
                        and the 2nd buffer contains 1 record
        then for $j$ = 1 to $2^{\log N}$
                        $\mathrm{P}_{\log N+1}$ , the processor compares the two
                                records and outputs the smaller record
                end for
        end if
end for
end
```

Figure 4.5 Pipelined Merge Sort Algorithm (Algorithm.4.2)

The processor P_i merges for N cycles and completes merging following $N + 2^i + i - 1$. The sorting procedure completes the operations at processor P_k in the cycle

$$N + 2^k + k - 1 = 2N + \log N - 1 .$$

Hence the merge sort algorithm is

$$p(N) = O(\log N) ,$$

$$t(N) = O(N) + O(\log N) ,$$

$$c(N) = O(N \log N) ,$$

and

$$b(N) = \sum_{i=1}^{\log N} (2^i+1) = O(N) .$$

The output buffer length of the i^{th} processor can be calculated. The processor P_{i+1} begins to merge as soon as there is an input sequence of length 2^i on one of its two input lines and a sequence of length one on the other. Thus, P_{i+1} initiates merging operation after the processor P_i has written 2^i+1 records into the buffers. Thereafter, when a record is written by processor P_i, it is read by the processor P_{i+1} since P_i completes each write cycle only when the processor P_{i+1} reads the record. Therefore, the maximum buffer length of the processor P_i is $2^i + 1$ and the total number of buffers is $2k - 1$, where $k= \log N$

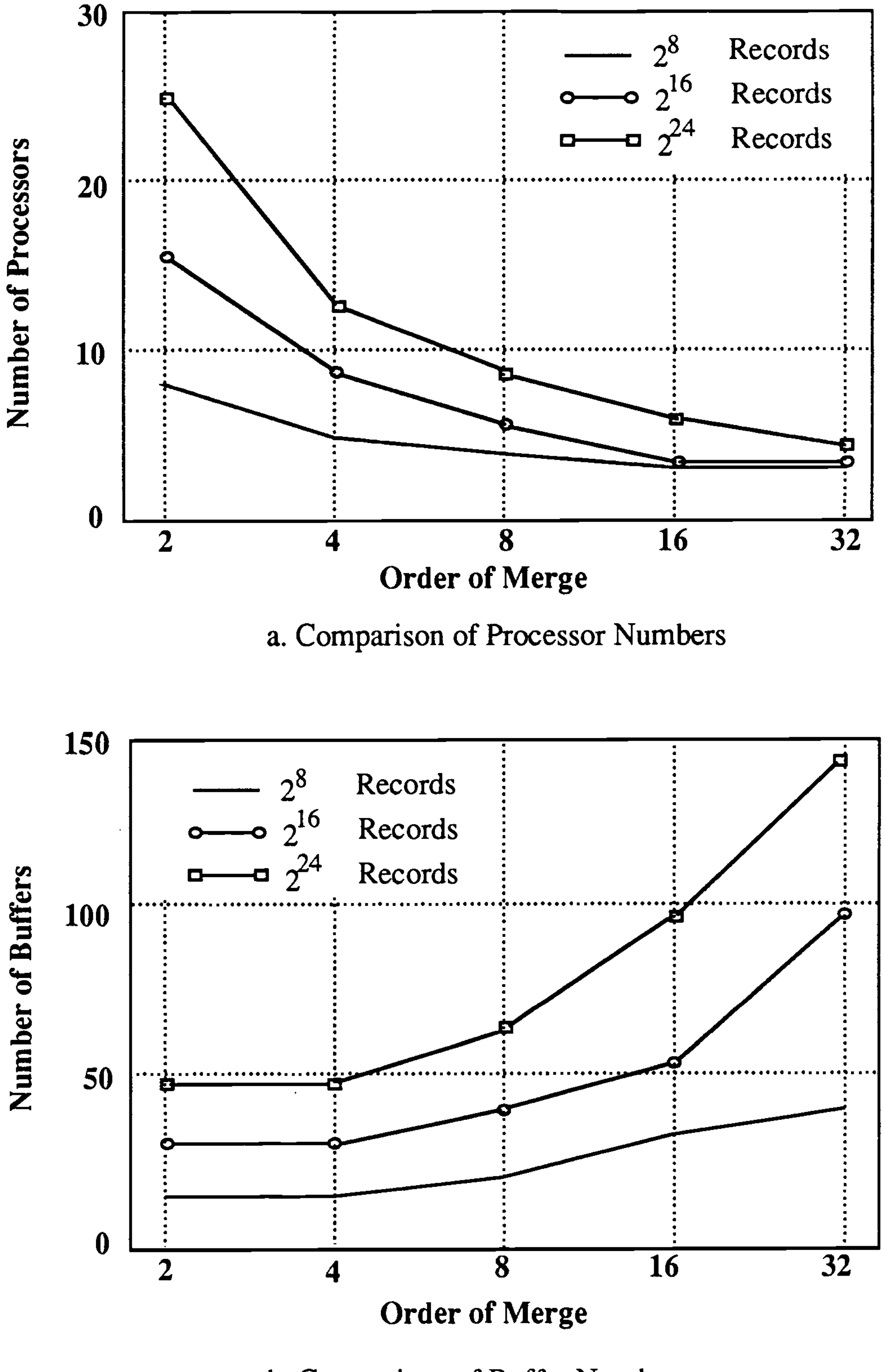
This two-way merge operation can be extended to an m-way merge operation. For an m-way merge sort, $k+1 = \lceil \log_m N \rceil + 1$ processors are required to sort N records in $2N + k - 1$ cycles. Thus, the time complexity of an m-way merge sort is $2N + \log_m N - 1$, where $2N$ represents the input and output time of the data stream and $\log_m N - 1$ represents the flush time for the pipeline. The processor speed for m-way mergers should be faster than that for two-way mergers since each processor cycle of an m-way merger includes $O(\log_2 m)$ comparisons. When each processor is able to perform m-way merge operations, the i^{th} processor, P_i, which has a buffer for the storage of $(m-1)m^{i-1}$ records outputs m^i data streams. Thus, the total number of buffers is $(k-1)m + \lceil N/m^{k-1} \rceil$. To sort N records, the m-way merge sorter has $O(N)$ time complexity and $O(N)$ buffer space for $\lceil \log_m N \rceil + 1$ processors.

4.2.2.2 Block-Merge Sort

To sort N records with single record input to P_0 in a single step, the first few processors merge only very short record streams. If the input to the first processor, P_0, is a presorted record sequence, S, it is possible to economize on the number of processors and the amount of buffers and sorting time. The input to processor P_0 then consists of S record sequences. The processor P_i produces S x m^{i+1} records and only $\lceil \log_m (N/S) \rceil$ processors are required for sorting N records. The number of processors and buffers required for a m-way merge sort and four-way merge sort using presorted sequences are illustrated in Figure 4.6 and Figure 4.7 respectively.

When presorting methods are used, sorting time is $2N + \lceil \log_m (N/S) \rceil - S$ cycles, plus S read cycles prior to the first write cycle. The required number of buffers for the processor P_i is $(m-1)S$ x m^i.

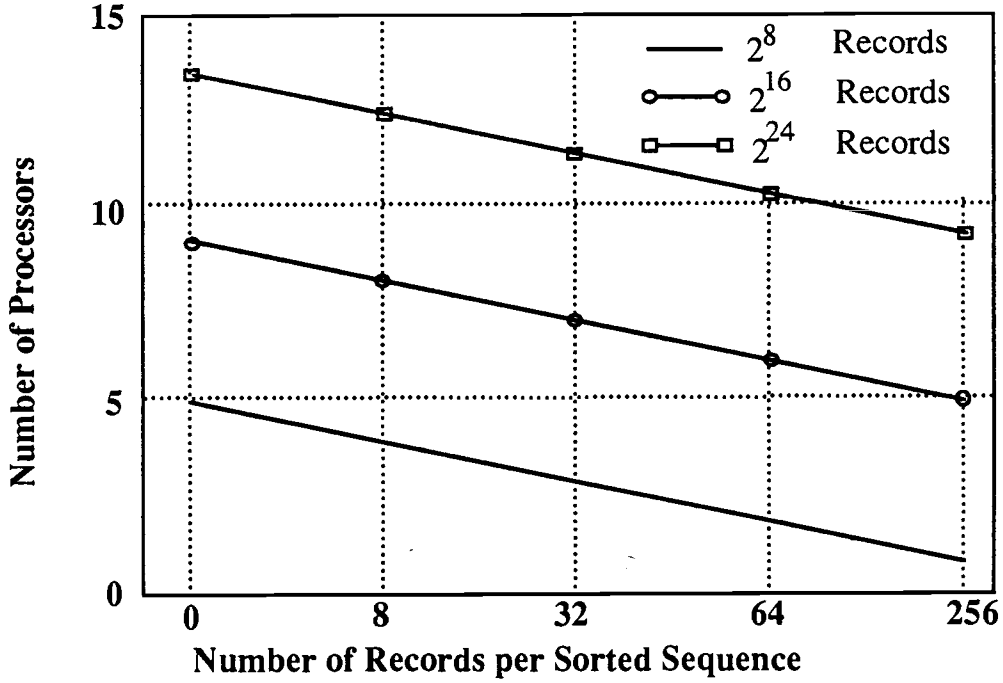
Depending upon the particular hardware implementation, the record block-merges improve sorting performance. Blocking causes a delay in record transfer from processor P_i to process P_{i+1} and slows



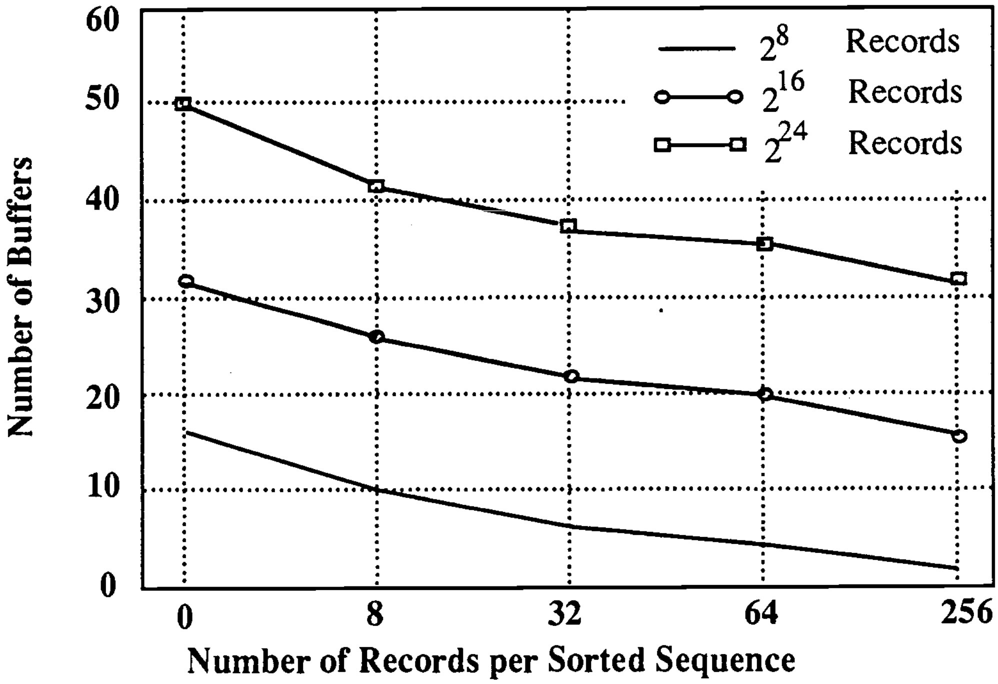
a. Comparison of Processor Numbers

b. Comparison of Buffer Numbers

Figure 4.6 Comparisons of *m*-Way Mergers



a. Comparison of Processor Numbers



b. Comparison of Buffer Numbers

Figure 4.7 Comparisons of Four-Way Merger

down the sort. However, by using presorted records, the blocking method improves merge performance when it is combined with such peripheral devices as disks or tape. Sorting time is then $2N + S \lceil \log_m(N/S) \rceil -1$. If S records are input in a block size B, the processor P_i buffer size is $B(m\text{-}1)m^i + B$.

4.3 Buffer Management

4.3.1 General Implementation

To store and retrieve sorted data from the buffer of each processor, using the pipelined merge sort algorithm discussed in the previous section, a buffer or queue management scheme is required. The size of buffer is dependent upon the efficiency of the design scheme. Three memory management algorithms [25] are considered: (1) the double memory method, (2) the pointer method, and (3) the block division method. The double memory method is quite simple, but is limited to memory efficiency of 50 percent. The pointer method achieves relatively higher efficiency, but its limitation lies in the requirement for more sophisticated processor design. The block division method reflects a midpoint of design complexity and processor efficiency. Its memory space is divided into fixed-size blocks, each of which can store several records. The size of a block can be determined as a multiple of the record length.

For an m-way merge, the double memory method requires $2\text{x}(m\text{-}1)$ data sequence buffers [25]. In turn, the block division method requires $(m-1) \times B + m$ buffers, B is the size of each block, and the pointer method requires m buffers. The pointer method is the best method to reduce chip area size for VLSI implementation. However, it requires complex controls to maintain the sorted sequences. The double memory and block division methods are difficult to implement for m-way merge sorts in VLSI. The problem with this technique is that dynamic allocation of buffers is required to execute merging operations. The buffer management controller must determine when each buffer is empty and allocate available buffers to allow for the input of new data streams. When all of the processors are operated at identical speeds, or if they are operated synchronously, even more buffer space is required. When the dynamic allocation method is not implemented, buffer space requirements consist of $2 \times m$ data sequence buffers.

4.3.2 Rebound Sorter and Buffer

The merger used for the finite-length strings from the rebound sorter must be compatible with the sorter operations. Some of the more important design considerations in connecting the merge unit and the sorter are as follows:

1) The output data for the rebound sorter consists of half-records, which are merged into whole records prior to the initiation of merge sort operation.
2) Storage of the input merge data from the sorter in the merger's memory buffers requires $2N$ cycles, while N cycles are required to merge data from the buffers. Buffer

management and merge operations are asynchronous operations.

3) The output sequence of the rebound sorter consists of S presorted data strings. Each buffer should have the capacity to hold an entire presorted sequence.

The mergė sort buffer for the output sequence of the rebound sorter can be implemented using the double block buffer method shown in Figure 4.8. There are $2m$ buffers of m^i size, increased by O(N) complexity. Using double block technique, the merge process is independent of processor speed, is simple to implement, and requires only a small controller area. In operation, data is initially loaded in the first half of the buffer and merge operations may begin as soon as it is filled. While merging takes place, the remaining half of the buffer is loaded with another data stream.

4.4 VLSI Implementation Considerations

To reduce the length of the merge stage, a four-way merge structure was designed for sorting large files. In the four-way merger, the number of merge stages can be reduced by a factor of two and still maintain the performance associated with the two-way merger. Therefore, the four-way merge sort is preferable to the two-way merge sort since the number of merge stages and the number of buffers can be reduced. The four-way merger consists of three two-way comparator and multiplexers shown in Figure 4.9.a

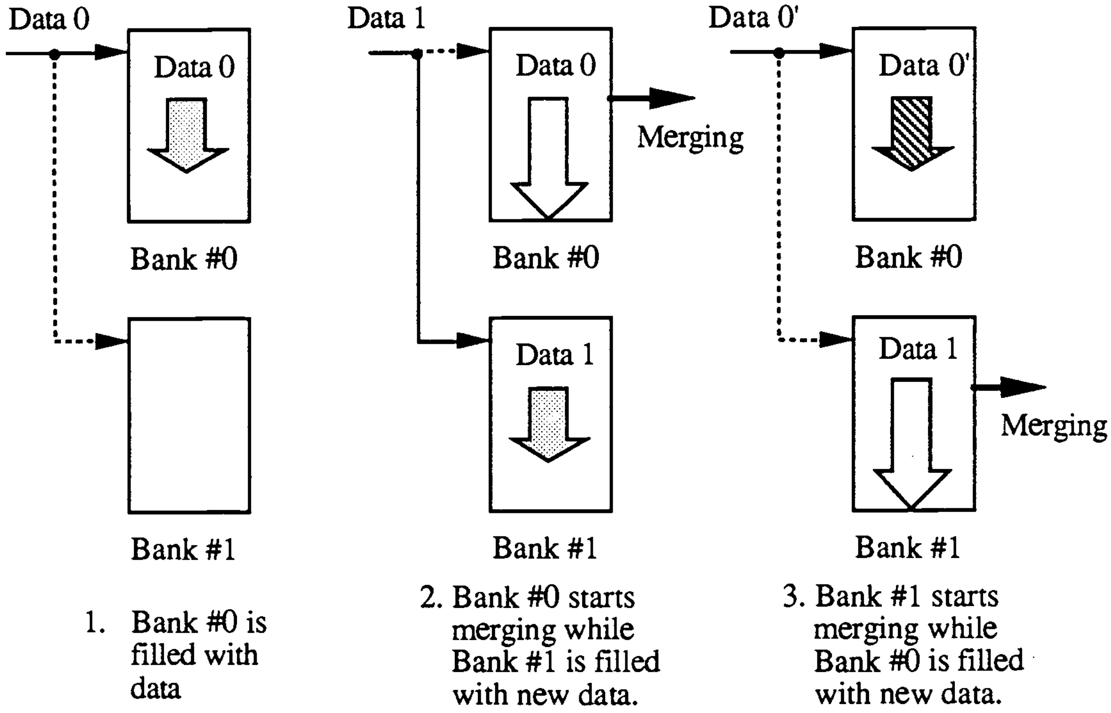
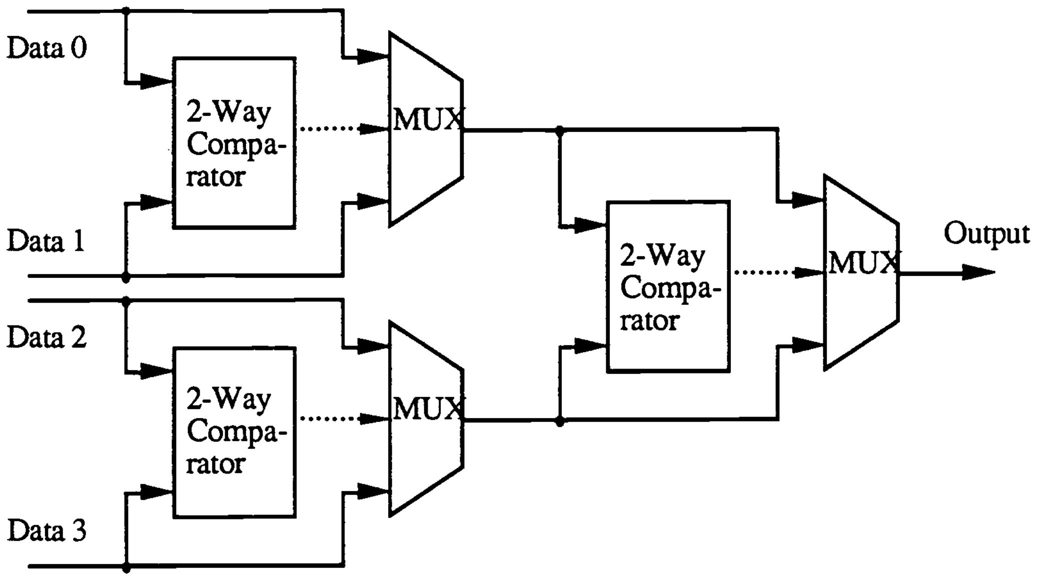

Data 0
Data 0
Bank #0
Bank #1
1. Bank #0 is filled with data
Data 1
Data 0
Merging
Bank #0
Data 1
Bank #1
2. Bank #0 starts merging while Bank #1 is filled with new data.
Data 0'
Data 0'
Bank #0
Data 1
Merging
Bank #1
3. Bank #1 starts merging while Bank #0 is filled with new data.

Figure 4.8 Double Block Buffer Method

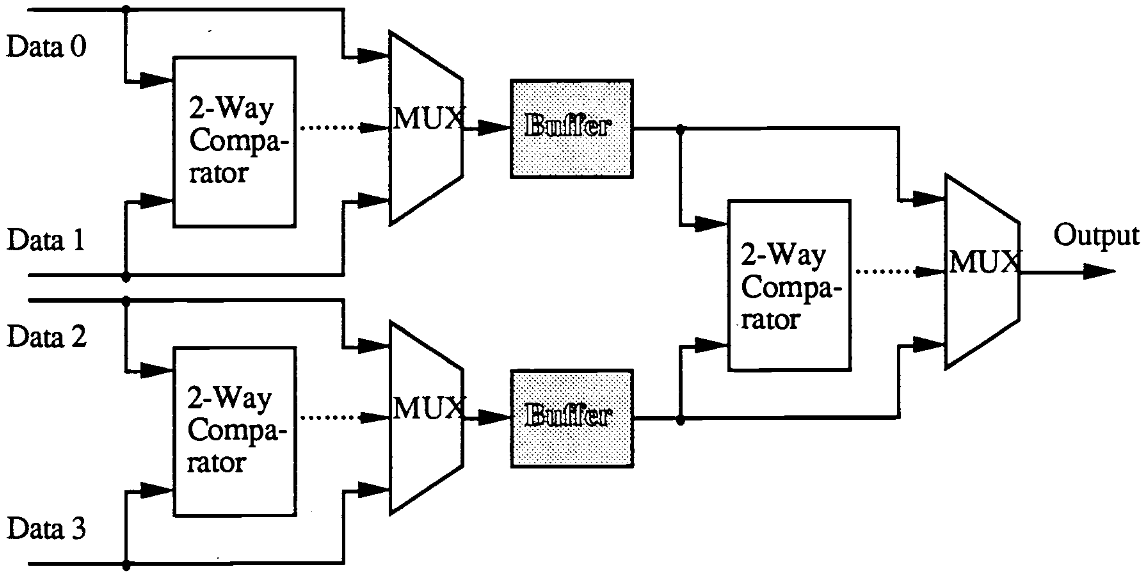
In this section, two methods of comparator design are considered. In general terms, an m-way comparator is required, based upon log$m$ stages and m-1 two-way comparators. A four-way comparator can be designed in two stages or as three two-way comparators. To connect the first and second stages, either the direct connection method or the indirect connection method consisting of an intermediate buffer connection may be used. These implementation schemes are shown in Figure 4.9. The direct four-way comparator yields a smaller chip area than the indirect four-way comparator, but is twice as slow. The critical timing paths for these two connections are shown in Figure 4.10. Processing speeds achieved with the direct connection method are only one-half that of the two-way merge sort. Therefore, there is no performance advantage to adaption of this method.

In the indirect method, total delay time is dependent upon the delay time of either the first stage or the second stage and is separated by intermediate buffers. Thus, the delay time for the four-way comparator is equivalent to the first stage delay time in combination with the delay time required to write data into the buffer, or to the data access delay time in combination with second stage delay time. A comparison of two methods is indicated in Table 4.1. The four-way merger with the indirect four-way comparator method is similar to the two-way merge, with the additional feature that it economizes on the number of buffers, controllers, and merging processing delays required to process the data stream..

The loading and access times of the intermediate buffer are small and the buffer requires only a small chip area (57 mils by 44 mils). This area increases as the number of merge stages are increased. That is, the buffer size of the i^{th} merge stage is $2m^{i+S}$. Merge processing time and area requirements reflect a trade-off effect. By using an intermediate buffer, the merge processing delay time requires only one clock cycle. As soon as the data streams for two merge sequences are stored in the buffer, the second comparator stage initiates operations with an overall delay of one cycle. Other advantages in the use of the indirect comparison method is that it is simple to control and requires only limited chip area for the controller and data paths. In the Genesil silicon compiler, RAM area size for buffers is proportional to the capacity of the memory.

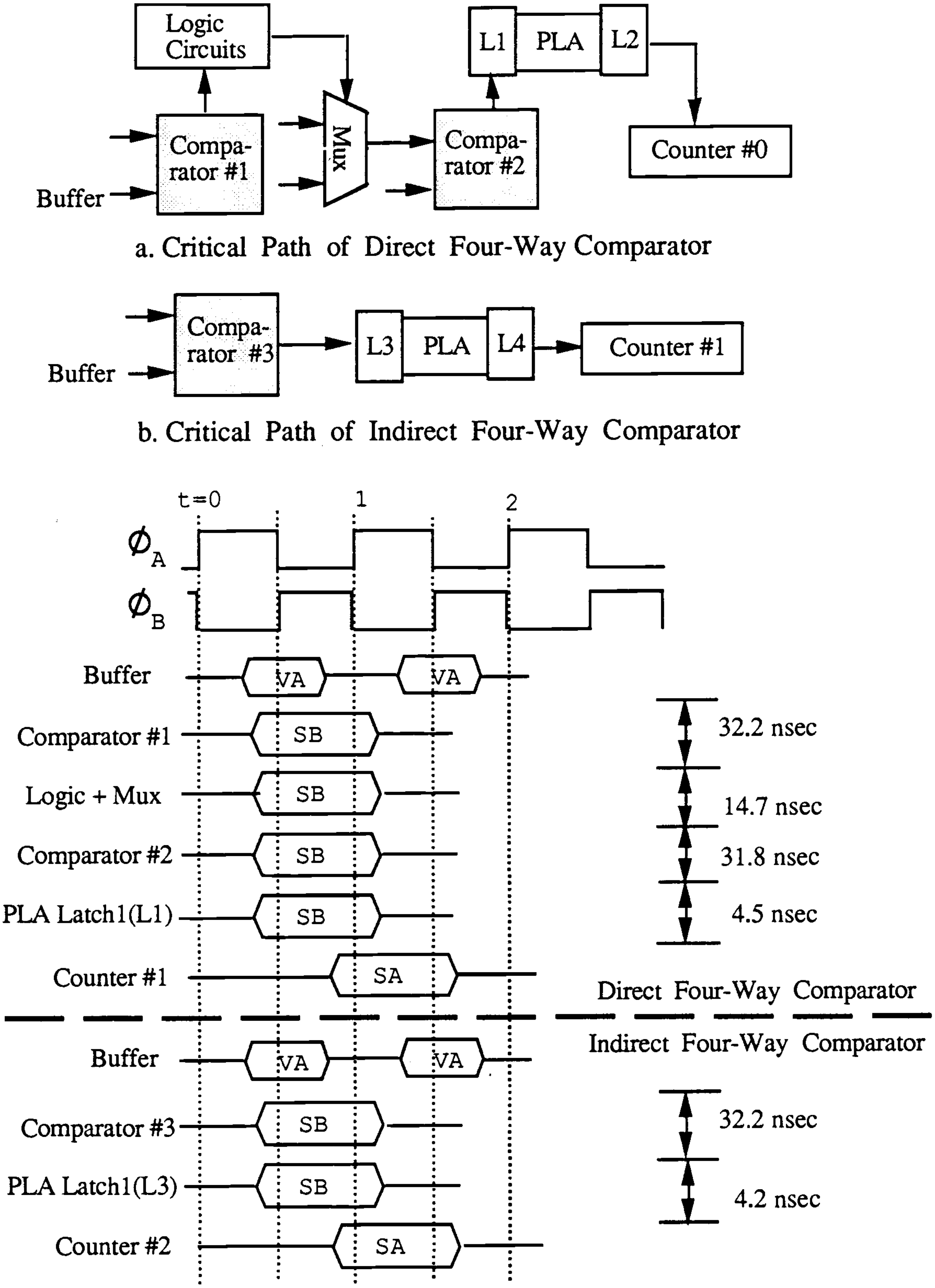


a. Direct Four-Way Comparator



b. Indirect Four-Way Comparator

Figure 4.9 Four-Way Comparator Structures



a. Critical Path of Direct Four-Way Comparator

b. Critical Path of Indirect Four-Way Comparator

c. Delay Time Comparison

Figure 4.10 Timing Comparison of Four-Way Comparators

Table 4.1 Comparison of Four-Way Comparators

Unit : mils

	3 Comparators + 3 Mutiplexers + 2 Data Buses	Input Buffers (8 x 16-bit)	Output Buffer (32 x 16-bit)	Intermediate Buffer (16 x16-bit)	Controller	Total Area	Speed
Direct Four-Way Comparator	51.54 x 130.78 = 6740.32	**8** x 54.43 x 46.24 = 2516.84	92.44 x 45.74 = 4228.21	**None**	54.18 x30.07 = 1629.19	**32732.44** = 181 x 181	**5.6**MHz
Indirect Four-Way Comparator	51.54 x 130.78 = 6740.32	**8** x 54.43 x 46.24 = 2516.84	92.44 x 45.74 = 4228.21	**2** x 57.48 x 44.20 = **2540.62**	54.18 x30.07 = 1629.19	**35273.06** = 188 x 188	**10.9**MHz

4.5 Four-Way VLSI Merger

The four-way merger includes intermediate buffers to improve performance. After buffer halves are filled with presorted data streams, the merger is initiated. After the initial loading delay, the merger input and output operations are overlapped completely with merging operation and pipelined.

4.5.1 Buffers and Controller

Buffers for the four-way VLSI merger can be implemented using several techniques, including FIFO, RAM , and the use of shift registers. The use of RAM provides very fast access in the smallest possible chip area. For the actual merger buffers, RAM memory was selected and the buffers are shown in the block diagram in Figure 4.11. The buffers were designed to perform as follows:

1) The buffers act like FIFO queues, storing data while merge operations are performed. Dual port RAM was used for simultaneous read and write capability.
2) Half-records are merged into complete records in the buffer as follows: The merge input data from the presorter consists of a sequence of half records, the most significant and the least significant, of which are combined to construct a complete record.
3) The buffers achieve fast access time in comparison to the merge processor since data read and write access times can be absorbed into the data transfer time.

The RAM can be used as a FIFO queue through the use of two counters which act as memory pointers, one for input and the other for output. To reduce routing, netlisting, and the chip area required for buffer control, the counters are imbedded in a PLA in conjunction with a controller. For an eight word buffer containing 16 bits per word, a 4-bit buffer input counter and a 3-bit buffer output counter are required. Sixteen 8-bit sorted data input streams must be stored in the eight word buffer. The first 3 bits of the 4-bit input counter are used to address the last bit of the four-bit input counter in order to merge two 8-bit data streams in one 16-bit complete record. The Genesil RAM allows "slice" read and write data operations, which provide the designer with access to the desired locations for slicing RAM inputs and outputs into fractional words rather than full word widths. This technique removes extra logic from this application and RAM access time is less than 15 *nsec*.

For the first stage of the merger, asynchronous control methods are used. The number of cycles for the input port is twice that for the output port. The controller controls data routing and control signals (Figure 4.11), thereby generating reads and writes to and from the RAM buffers. The control signal sequence for the RAM buffers is as follows:

1) The switch bank signal selects one of two buffer banks. While one buffer bank is read by a merge processor, the other buffer bank is stored with data from the presorter without sending data to the merge processor.

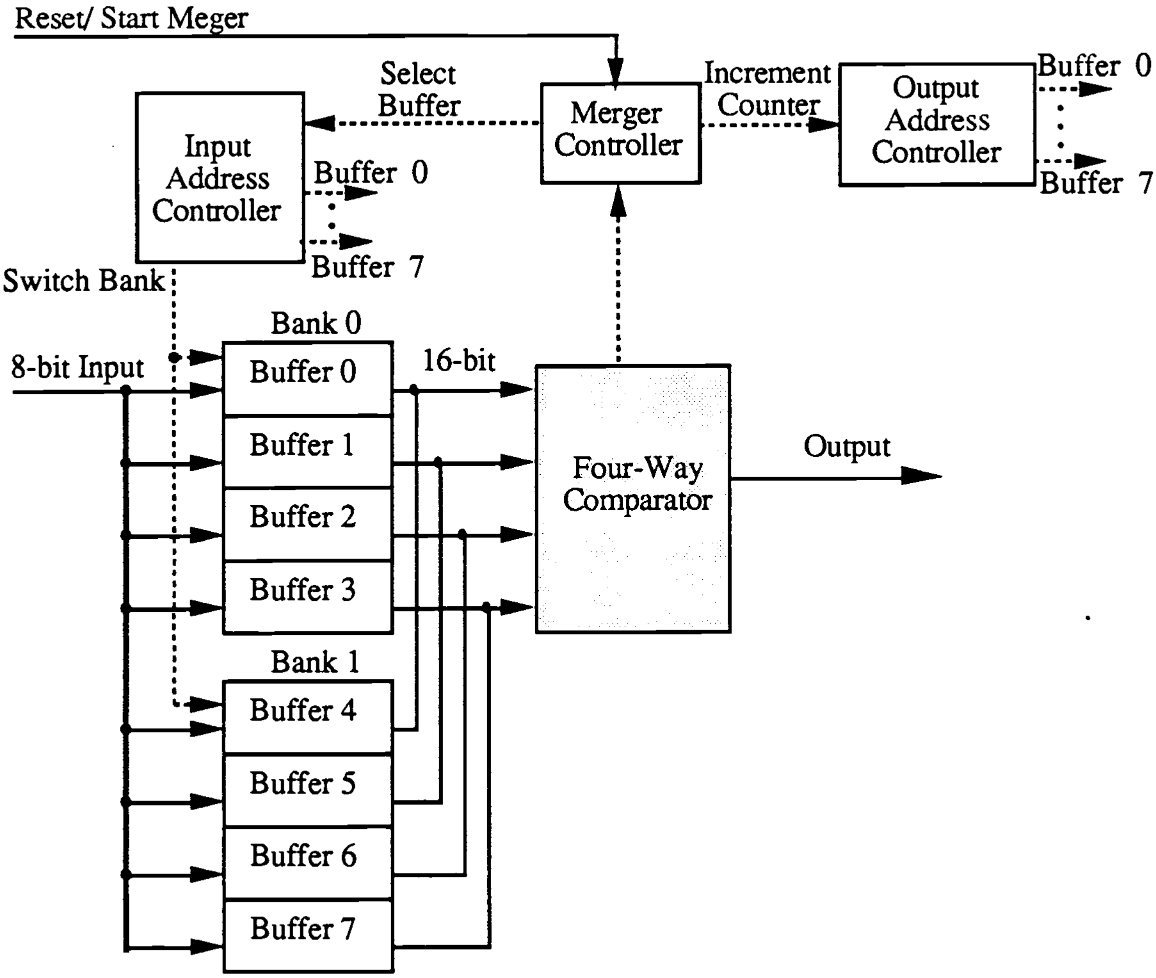


Figure 4.11 Four-Way Merger Block Diagram

2) The select signal generates RAM write enable signals for the buffers of two banks. Only one signal must be enabled in order to store a data sequence in a given RAM buffer.

3) The start merger signal enables the initialization of merge processing operations, causing the system to send output data to intermediate buffers in the next stage.

4) The increment counter signal causes the buffer output counters to be incremented. Only one counter is incremented at a time when buffer data is selected to be moved.

4.5.2 Simulation and Timing

For purpose of simulation, it was assumed that the first stage of a merger was connected to the output of a rebound sorter. The simulation input data were 16 half-records, which were merged into 8 complete records in the buffers. Variable test sequences were applied and the functionality of the system was verified. The total response delay time of the merger in simulation was 65 clock cycles.

The merger can achieve speeds up to 10.8 MHz, assuming a symmetric clock cycle. To increase merge processing speed, the indirect method was implemented. The use of RAM slices for this implementation saved time when merging half-records into complete records. The critical timing path of the four-way merger is shown in Figure 4.10.c. The comparator caused the longest delay path and when more efficient comparators are designed by the customer design, the total delay time may be reduced by a factor of four in comparison to the delay times achieved with the present merger.

4.6 Conclusion

Merge algorithms and their associated time complexities were analyzed in this chapter. The intent of the analysis was to develop an optimal design for an efficient VLSI merge algorithm. The parallel binary merge and the pipeline merge sort algorithms had time complexities on the order of O(log$N$). The performance and parallelism of the parallel binary merge were superior to those of the pipelined merge sort [9]. The buffer size requirements of the former were modest in comparison to those of the latter. The number of processors required for an efficient parallel binary merge sort (O(N)) is much greater than for the pipelined merge sort (log$N$). However, the processors for the parallel binary merge sort could not be operated synchronously, implying that a child processor could be blocked when its parent processor is not ready to receive new records. In general, it is necessary to consider the bandwidth of the processors, along with the limits of tree levels. Performance of the pipelined merge processor could be improved by the implementation of a block-merge sort algorithm, which would increase the input length of the presorted input sequences. When presorted sequences of the length S are used, the block-merge algorithm requires log(N/S) processors.

Buffer management methods for the merge sort were discussed with the objective of reducing RAM sizes and optimizing the design process. The double block buffer method was implemented for the VLSI merger, resulting in a buffer memory size increase of O(N) complexity. The buffer management scheme adopted requires connection of the merge sorter in a chain to improve the sorting performance for very large files.

In order to implement a four-way comparator with three two-way comparators, both direct and indirect methods were examined. The direct four-way comparator area requirements were modest in comparison to those of the indirect four-way comparator, but at the same time the former achieved only one-half the speed of the latter. Consequently, there was no advantage to the use of a four-way merger based upon a direct four-way comparator in place of the two-way merger. The four-way comparator is faster, but presents the disadvantage of a slight increase in chip area. However, the total area requirement of the four-way merger based upon the indirect four-way comparator is less than that of the two-way merger chip area, and better performance is achieved without the loss of speed.

Utilizing the indirect method, a four-way merger was implemented in VLSI. Its maximum speed was 10.8 MHz. The chip area, including power, ground, clock, and input and output pads, was 9013 microns by 9009 microns (354 mils by 354 mils). The basic merge structure was based on block-merge microarchitecture with presorted inputs. This format can be used in several applications, including a cascaded connection of tree structures with external RAM or mass storage devices. This application will be considered in Chapter 5.

5. VLSI SORT-MERGE SYSTEM

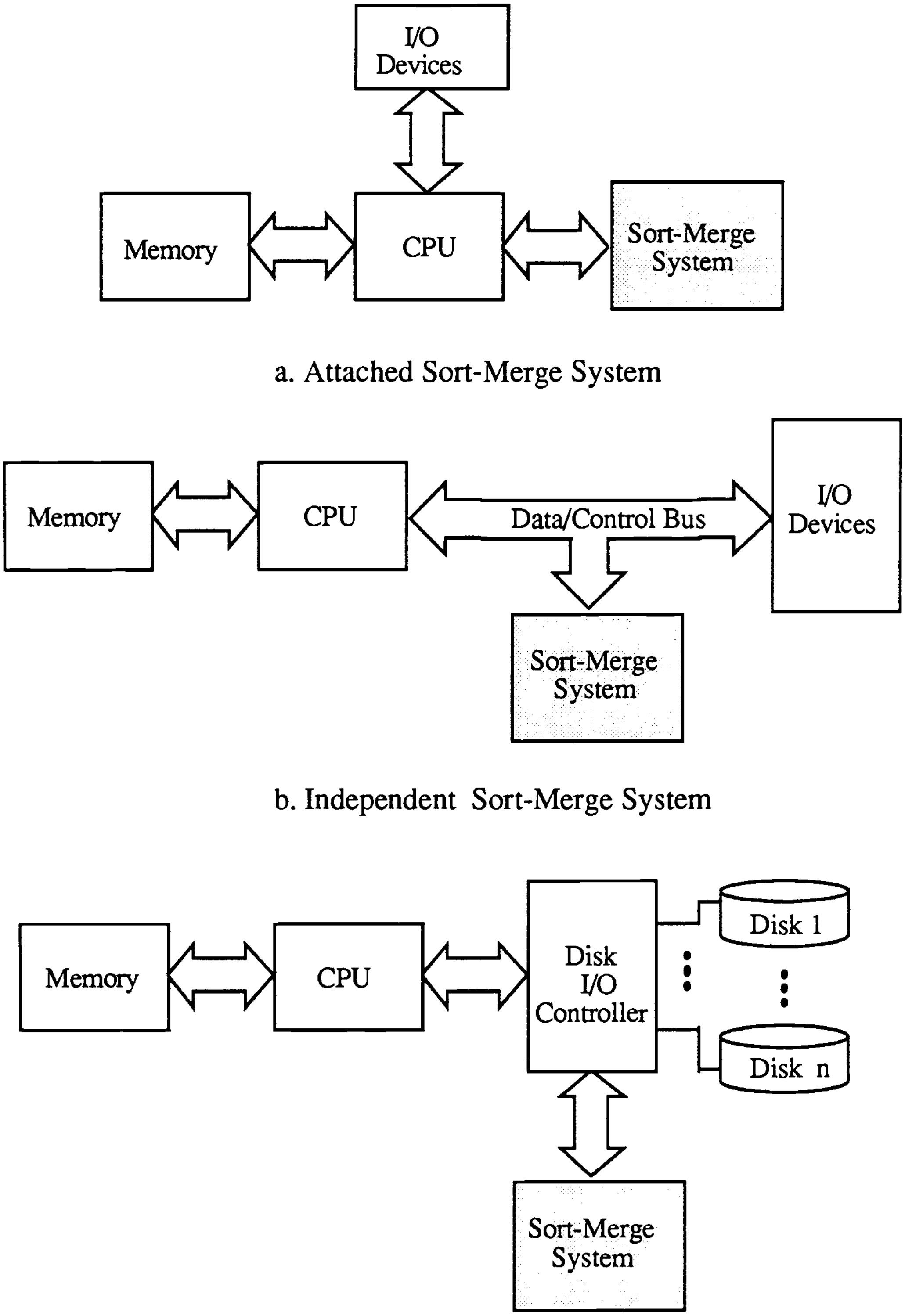
In the preceding chapters, several VLSI sorters and mergers were compared. The architecture and design of both a high performance rebound sorter and a four-way merger were proposed. In order to achieve adequate sorting capacity for very large files as well as maximum performance, it was determined that the sorter and the merger should be combined in a single sort-merge system. It was also established that the use of presorted sequences allows the use of fewer merger stages.

In this chapter the development of a VLSI pipelined parallel sort-merge system is presented. The system consists of two pipelined rebound sorters connected to a unique, pipelined four-way merger. The proposed system uses two rebound sorters for the generation of sorted strings of a fixed length, and a four-way merger for the combination of individually sorted strings into a single, continuous string of sorted records. The rebound sorter input data passes continuously through the input ports. This process generates sorted strings which are stored in a 8 x 16-bit dual-port RAM buffer. When the first buffer is filled, a pipelined merge is initiated while the sorter simultaneously fills the second buffer. Sorting and merging are synchronized and operate concurrently. The two RAM buffer banks can alternate merge and storage functions. Overall performance for the proposed sort-merge system is excellent since input and output operations are completely overlapped with sorting operations. In general, this system consists of a series of pipelined mergers with appropriately sized buffers. The system may be used to extend the basic sorter or merger discussed previously, or it may be connected in the tree-structured sort-merge system proposed by Bitton [9].

In addition, the system may be attached directly to a disk controller or to an I/O bus between main memory and disk storage. It may also be attached to a general purpose computer CPU as a sort-merge coprocessor. These applications are illustrated in Figure 5.1. When a disk controller is combined with several high capacity disk systems with fast access times, the sort-merge system can be imbedded in the disk controller to provide internal sorting capability, or it may be shared by a network file system. High speed low-level sorting and searching units may be used for the execution of selected database functions, *e.g.*, join, member, select, and sort [10,19,32,34].

5.1 Sort-Merge System Architecture

The basic performance of the sort-merge system is extended by a factor of two with the sort-merge configuration shown in Figure 5.2. This system includes a first stage presort in which multiple rebound sorters are used for the initial sorting of input sequences. Merge operations are conducted in the following stage.



a. Attached Sort-Merge System

b. Independent Sort-Merge System

c. Disk-Based Sort-Merge System

Figure 5.1 Sort-Merge System Applications

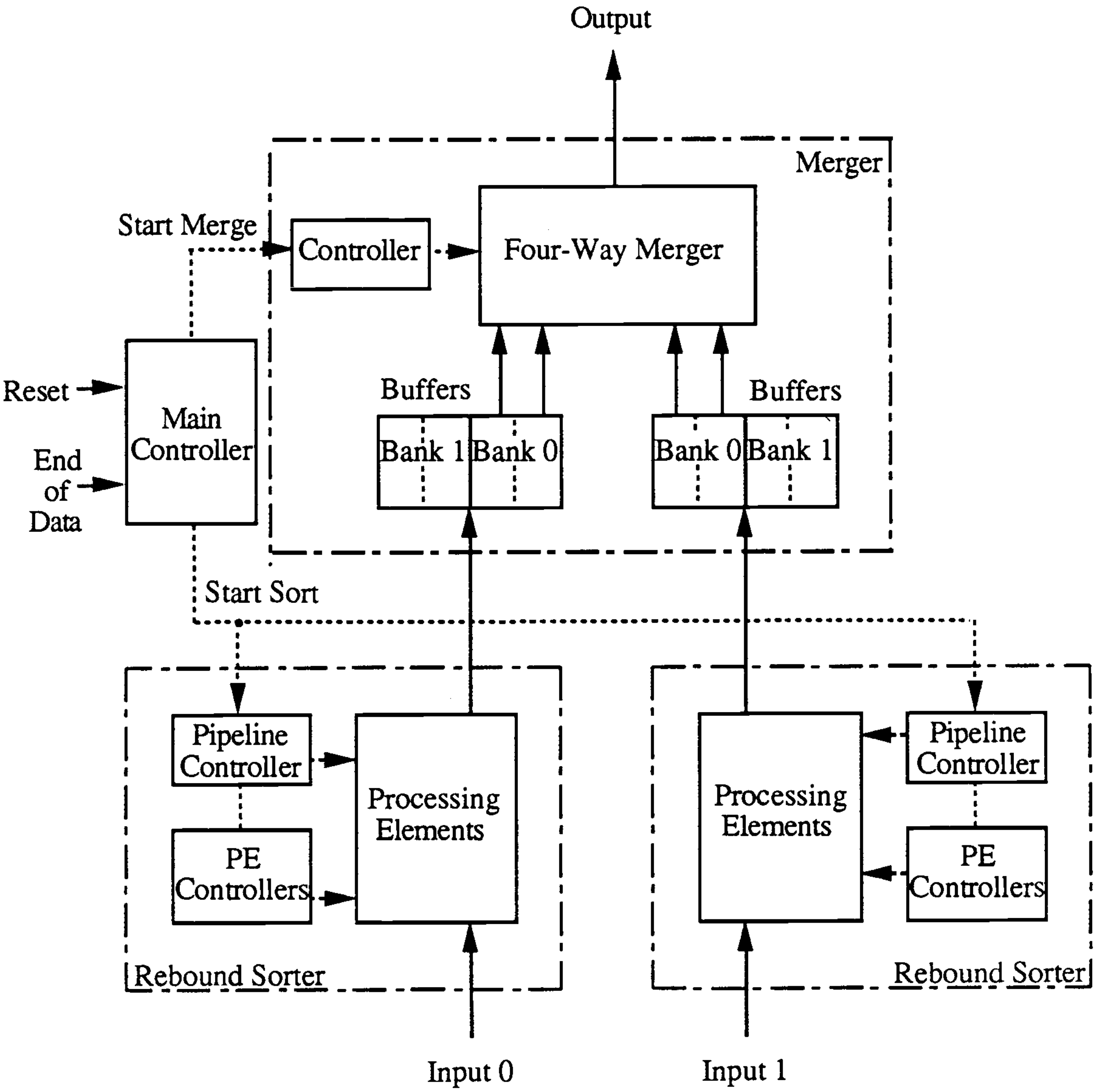


Figure 5.2 Sort-Merge System Block Diagram

When the presorted outputs are sorted sequences of length S, the merger performs an m-way merge operation which generates sorted sequences of the length m x S. The i^{th} merge produces a data stream of the length m^i x S, and the final merge processor ($\log_m(N/S)^{th}$) produces sorted sequences of length m^k x S, where k =

$\log_m(N/S)$. Buffer implementation methods were described in detail in section 4.3. The buffers can be implemented directly in VLSI or on a printed circuit board (PCB) using commercially available RAM, depending on the number of records to be sorted and the nature of the application.

In an alternative application, a sort-merge system consisting of multiple rebound sorters, mergers, and RAM may be designed for incorporation on a printed circuit board (PCB). Such a system provides additional flexibility to the system designer. Current VLSI technology is sufficient to support either approach.

5.2 Sort-Merge Algorithm

The basic parallel merge sort algorithm was discussed in Chapter 4. If the presorter is implemented with a rebound sorter, each record is divided into two half-records. To compensate for this disadvantage, two rebound sorters are utilized. Each sorter generates S records every $2S$ time units. These sorters are required to implement the algorithm presented in section 4.2.1. Four buffers are connected to the first stage processors, P_0 and P_1, of the four-way merger. The merger contains two processors, P_0 and P_1, in the first stage and one processor, P_2, in the last. The sorter stores sorted sequences of length S in two buffers of length $2S$ in $4S$ time units. Initially, the merger's first stage processors are idle for $2S$ time units. After $4S$ time units, the processor P_0 starts merge operations. In the last stage, P_2 initiates merge operations as soon as the processors P_0 and P_1 output data to the intermediate buffer. The total time for P_2 processing is $4S$, with a total delay time through the merger of $4S+1$. The delay in the operation of processors P_0 and P_1 is compensated by the four-way merge operation. Given that the sort-merge system is designed as a pipelined merge sorter, and that the system accepts sorted strings of length S, the following complexity measures can be derived:

$$\begin{aligned}
\mathrm{t}(N) &= 2S + S\lceil \log_4 \frac{N}{S} \rceil + 2N \\
&= \mathrm{O}(S \log \frac{N}{S}) + \mathrm{O}(N), \\
\mathrm{p}(N) &= 3 \log_4 \frac{N}{S} + 2S \\
&= \mathrm{O}(\log \frac{N}{S}) + \mathrm{O}(S), \\
\mathrm{c}(N) &= (\mathrm{O}(N) + \mathrm{O}(S \log \frac{N}{S}))\ (\mathrm{O}(\log \frac{N}{S}) + \mathrm{O}(S)) \\
&= \mathrm{O}(NS) + \mathrm{O}(S^2 \log \frac{N}{S}) + \mathrm{O}(N \log \frac{N}{S}) + \mathrm{O}(S\,(\log \frac{N}{S})^2),
\end{aligned}$$

and

$$\begin{aligned}
b(N) &= 2Sm \sum_{i=1}^{N} m^i \\
&= 2Sm \frac{m^{\log_m (N/S)} - 1}{m - 1} \\
&= \mathrm{O}(N)
\end{aligned}$$

When S is selected properly, $\mathrm{c}(N)$ can be optimal. When $S = N$, $\mathrm{c}(N)$ is maximum complexity of $\mathrm{O}(N)^2$.

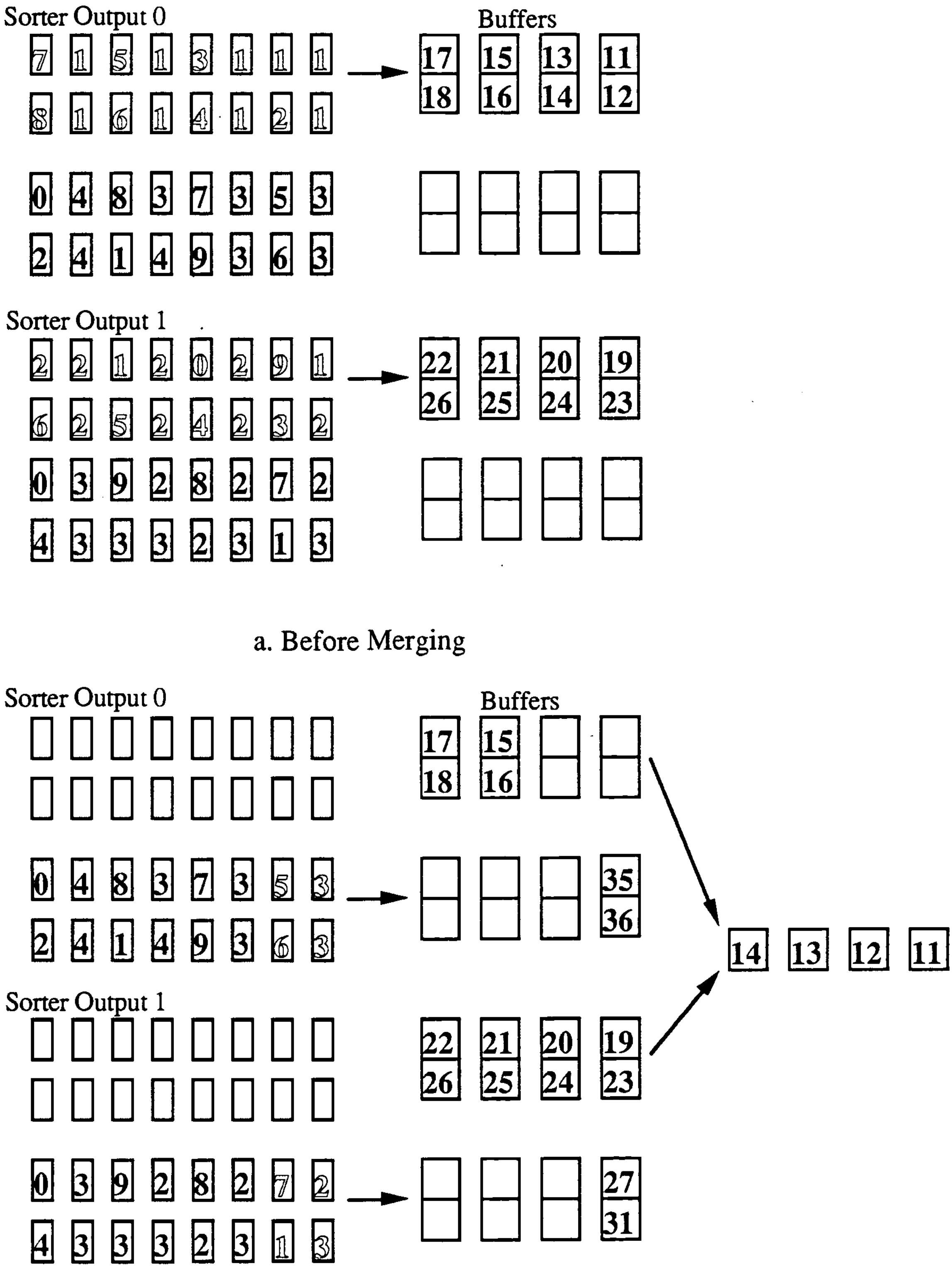
5.3 System Implementation

Two rebound sorters and a four-way merger were combined in a single sort-merge system on a single chip, the block diagram of which is shown in Figure 5.2. The system is simple in design, but is capable of sorting 10^6 records in 0.2 second when connected to 15 merge stages. To reduce the number of merge stages, the initial sorter capacity can be increased. For example, if the sorter can simultaneously process 64 data sequences, only 10 merge stages are required for sorting 10^6 records. This implementation is similar to the sorter and merger designs presented in the previous chapters. An example of the VLSI sort-merge system operation is shown in Figure 5.3.

5.3.1 Controller and System Integration

To control the sort and merge stages, an overall system controller is required. This controller generates the proper control signals for each sort and merge stage. A hierarchical control strategy was employed. The system controller's primary function is to pass appropriate control signals to controllers at lower levels in the controller hierarchy. The controller was implemented as a PLA finite state machine.

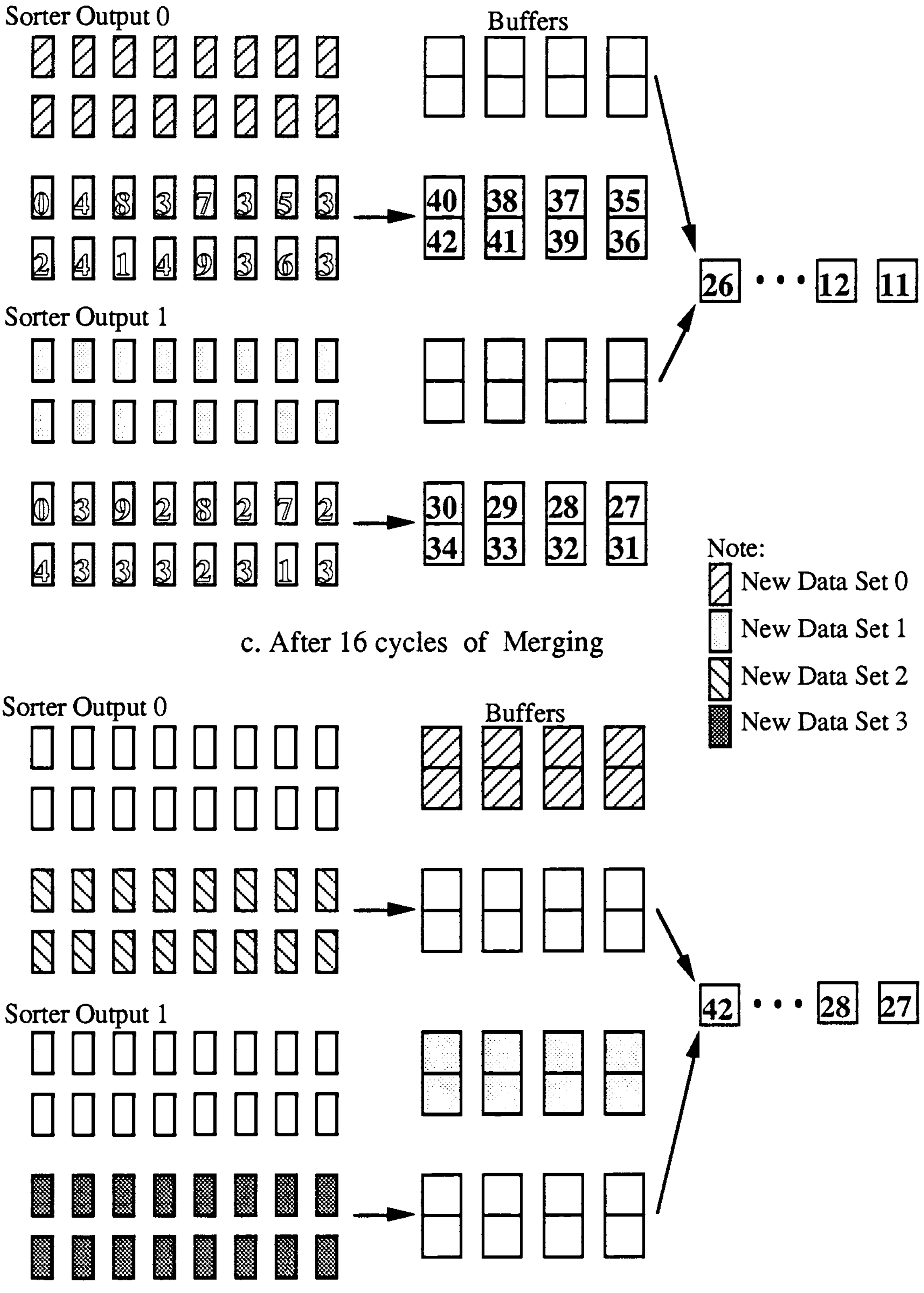
The overall system controller generates the signals shown in Figure 5.2. The signals perform the following functions:



a. Before Merging

b. After 4 cycles of Merging

Figure 5.3 Sort-Merge System Operation



c. After 16 cycles of Merging

d. After 32 cycles of Merging

Figure 5.3 Sort-Merge System Operation (*continued*)

1) The start sort signal initiates sorting operations.

2) The start merge signal initiates the first stage of four-way merge operations.

3) The end of data signal is used to halt sorting and merging operations.

To integrate the sort-merge operations of the individual modules, system input and output data flow must be considered. Two input sequences are simultaneously applied to each of the rebound sorters, while unsorted records are distributed in a balanced method by an external preprocessor. It is assumed that $N = 2 \times S^i$, where S is the length of the input sequence and N represents the number of records to be sorted. When N is an arbitrary integer, dummy records with maximum values are added to the input file and the presort and merge stages are synchronized.

The sort-merge system, consisting of two sorters, a merger, a controller, and input and output pads was built on a single chip. The chip is area is 10400 microns by 10800 microns (416 mils by 432 mils). It was assumed that the system output is stored in a separate, external RAM.

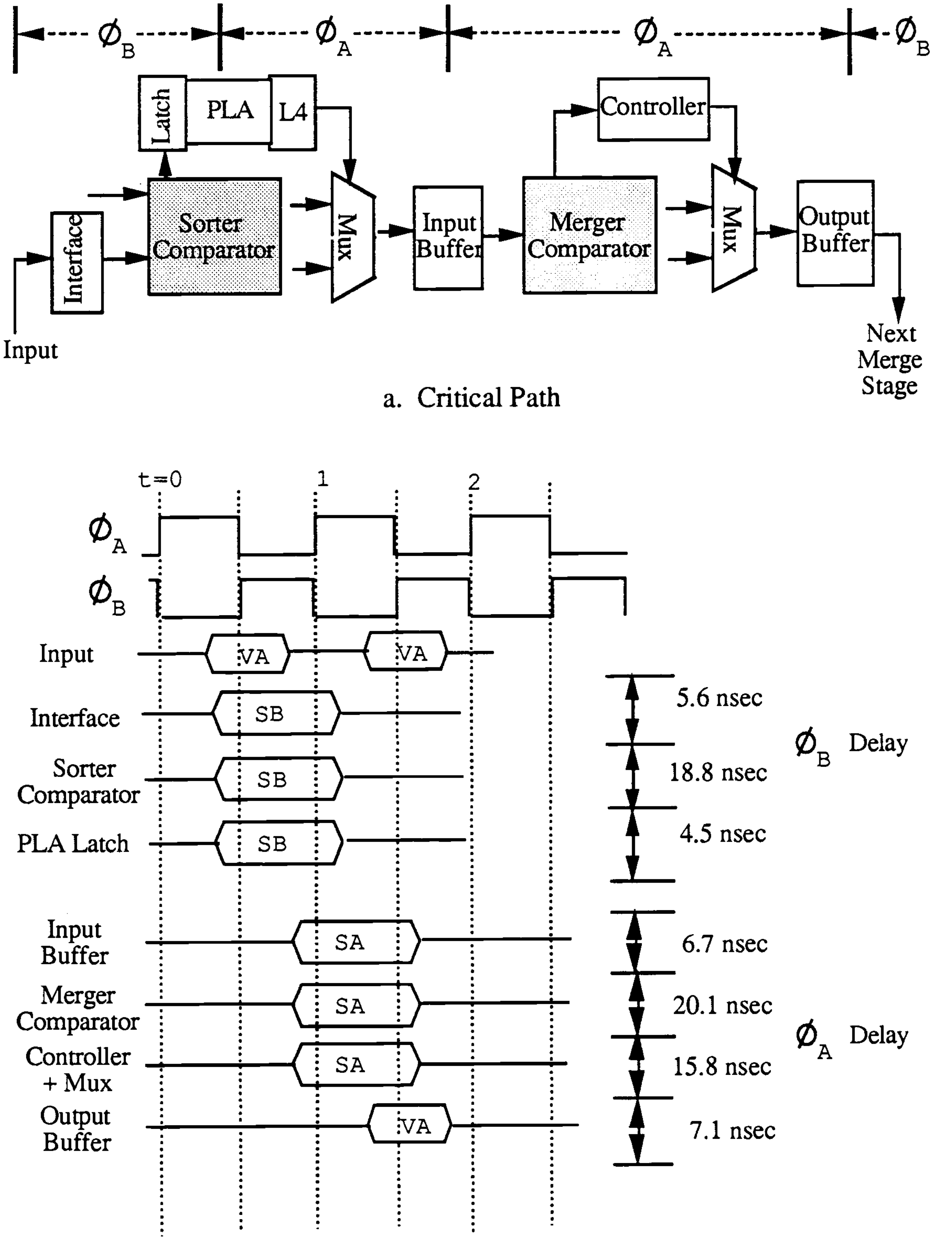
5.3.2 Simulation and Timing

A variety of input sequences were simulated to verify the functionality of the sort-merge system. Two sets of 16-bit data sequences were prepared. Each record was divided into two half-records for input to the two sorters. After applying eight records to the input of each sorter, a second sequence of eight records was also loaded into the sorter, *i.e.*, $N/2$ records were loaded into a sorter, where $N/2$ is a multiple of eight records. Dummy records of maximum value were added to complete the eight-record sequence, when required.

Timing verification was performed to establish the total input/output delay time and the maximum symmetric clock frequency for the system. The critical timing path is shown in Figure 5.4. The sort-merge system operated at a maximum throughput rate of 10.8 MHz. Overall system performance was less than that of the high performance (17MHz) rebound sorters due to the additional logic delays associated with the sort-merge system, plus parasitic delays incurred due to additional wiring. The longest delay path occurred in the 16-bit comparator's carry chain. If the comparator is optimized using custom VLSI implementation techniques, it may be possible to increase the system performance while reducing the overall chip area.

5.3.3 Floorplanning and Routing

Floorplanning is defined as the process of moving building blocks around on a chip in order to minimize area and delay time. Efficient floorplanning was achieved in this case via manual methods. This process was difficult given the varying shapes of the system building blocks. Automatic techniques for performing these floorplanning functions in an optimal way are not yet available. The final floorplan is shown in Figure 5.5. The input ports are located on east and west sides of the chip and the output port is located on the



b. Major Delays

Figure 5.4 Critical Path and Major Delays of the Sort-Merge System

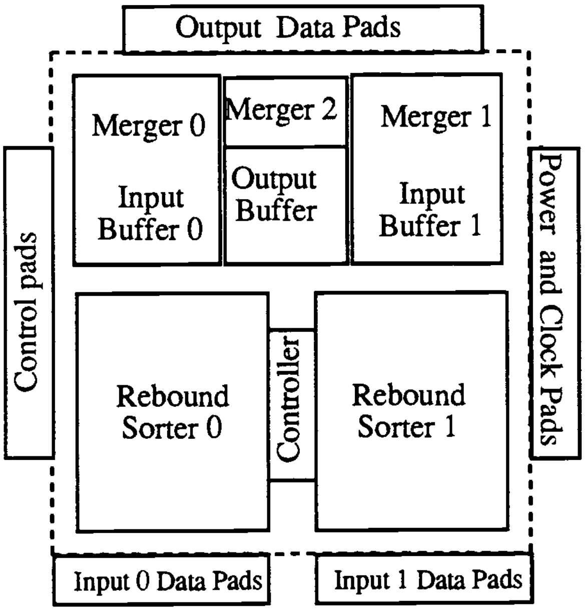


Figure 5.5 Sort-Merge System Chip Floorplan

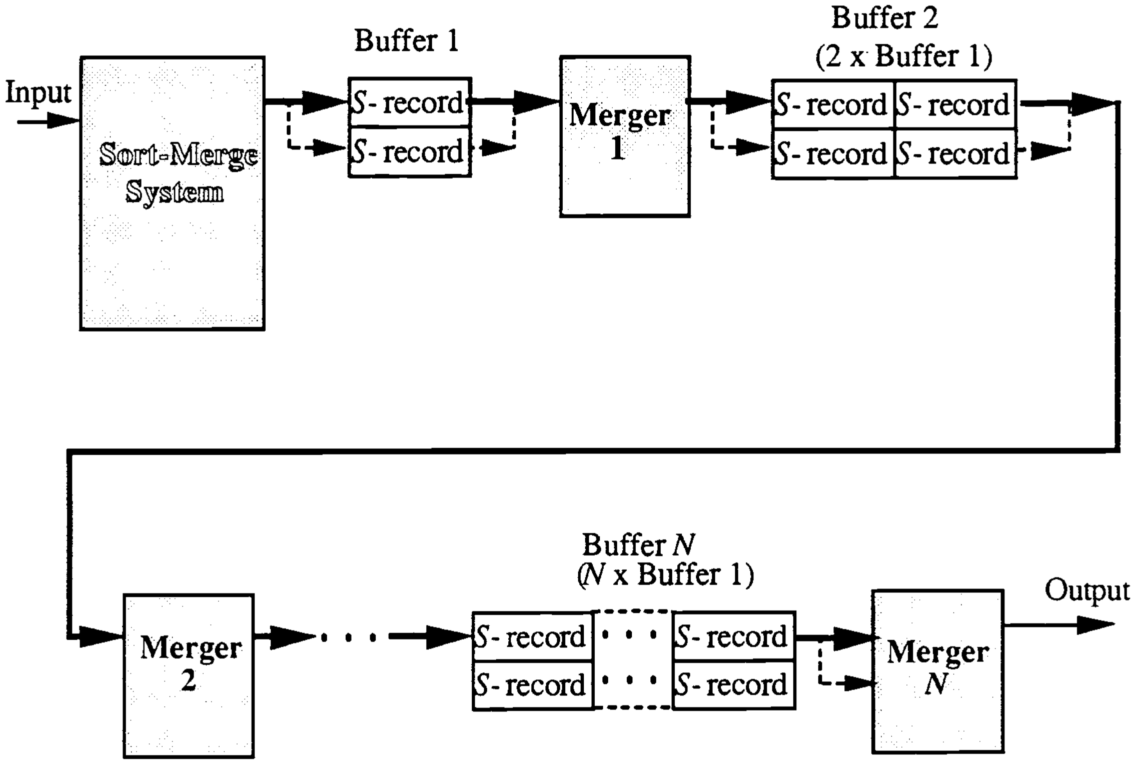
south side. The main controller is located in the center of the chip to reduce the length of the routing lines used for control signals.

5.4 Additional Modular Sort-Merge Systems

In the previous section, a sort-merge system built on a single chip was described. System architectures incorporating individual sort and merge chips composed of the subsystems described in previous chapters may also be implemented, if desired. Among these are the iterative sort-merge system, the tree sort-merge system, and the disk-based sort-merge system.

5.4.1 Iterative Sort-Merge System

The structure of an iterative merge system which uses individual rebound sorter chips, merger chips, or sort-merge chips is similar to the pipelined merge sorter proposed by Todd [47]. The first stage of the iterative merge system may consist of a rebound sorter or a sort-merge chip. Sorted data streams from the first stage are applied to the next stage. Multiple buffers are required to obtain high performance. The block diagram for this system is shown in Figure 5.6.



Note: S is the length of the sort-merge system output sequence.

Figure 5.6 Iterative Sort-Merge System

5.4.2 Tree Sort-Merge System

The tree sort merge system can be configured in a variety of methods. These ways are dependent upon the nature of the application and thedesired system processor speed. The binary tree structure is one of the most desirable structures since it has the advantage of requiring only a small buffer. If the processing speed of a higher level processor in such a system is twice as fast as its lower level processors, a powerful sort-merge system can be achieved. However, each processor has its own speed limitations. For a tree structure of h levels, the processing speed of the root should be 2^h times that of its leaf processors. When the binary tree structure is combined with a block-merger, the binary tree structure can be built with processors of identical speeds. With a buffer size of B, all processors may operate at identical processing speeds so long as inter-level communications are asynchronous. A block diagram for this system is shown in Figure 5.7.

5.4.3 Disk-Based Sort-Merge System

The sort-merge system may be incorporated in a disk controller. In this case, the system does not require a high speed processor due to the I/O rates associated with standard disk controllers (3M bytes/sec). Sort-merge system inputs and outputs may be sent directly to the disk. Alternatively, they may be passed through appropriate buffers to reduce the number of disk accesses and increase overall system performance [24, 45]. A simple disk I/O model is analyzed in the following section. The VLSI sort-merge system discussed in Section 5.3 plus a disk controller are incorporated in the design of a system capable of handling very large files.

5.4.3.1 Disk I/O Access Model

Disk I/O access time can be divided into three components: seek time, rotational latency time, and input and output data transfer time [13,33,41,49]. Seek time is dependent upon the distance the disk heads move from one cylinder to the next cylinder. Modeling seek time appropriately is a complex process, given the application dependent seek distributions [49]. Rotational latency is defined as the time spent reading or writing to a specific cylinder block and is proportional to the number of blocks placed on each cylinder. Input and output data transfer time is the time required to perform data read and write operations, plus data transfer time between buffers and heads. To model a disk system adequately for the sort-merge system, the following assumptions were made:

1) The disk system contains multiple disks and each cylinder on the each disk is composed of several tracks.
2) Head movement for accessing the records is random, *i.e.*, heads remain in place following an I/O disk access, thus alternating record accesses require head repositioning for each seek.
3) Average rotational latency time is one-half of one complete disk revolution.

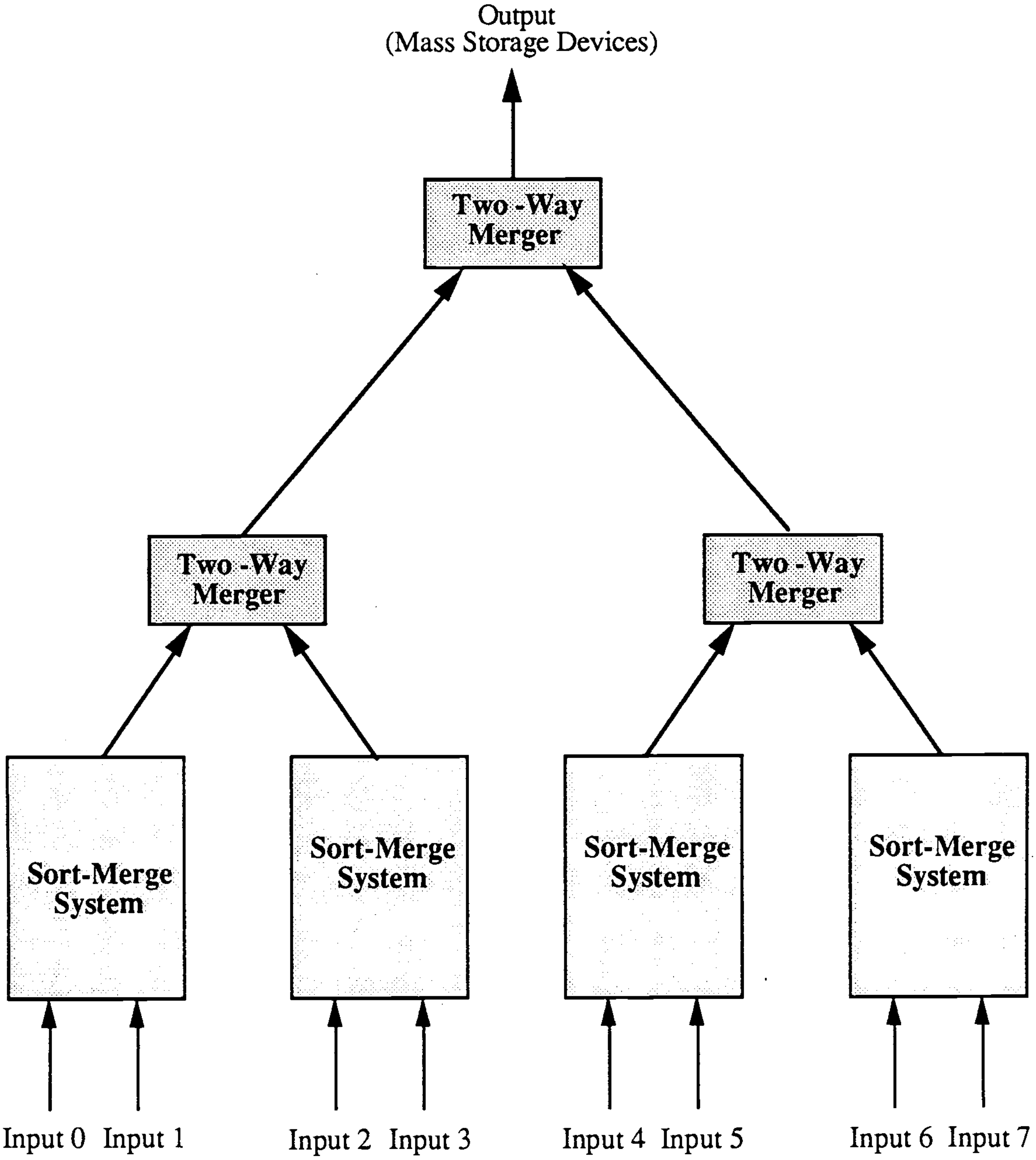


Figure 5.7 Tree Sort-Merge System

4) Blocks indicate physical blocks rather than logical blocks and several logical blocks may be contained within a single physical block.

5) Average input and output times are used.

6) Record length is fixed.

Based upon these assumptions, the following theory was developed:

Lemma 5.1: The expected head movement for accessing N records randomly distributed on disk is $N/3$.

Proof: Let a set of records, $r_1, r_2, \ldots, r_N$, be randomly distributed on disk cylinders with the same fixed record lengths. Their access probabilities are $p_1, p_2, \ldots, p_N$ and $\sum_{i=1}^{N} p_i = 1$. The expected distance traveled by the head from one requested position to another is:

$$\mathrm{E}(d_i) = \sum_{i=1}^{N} \sum_{j=1}^{N} p_i p_j \, d(i,j)$$

$$= \sum_{i=1}^{N} \sum_{j=1}^{N} p_i p_j \, |d_i - d_j| ,$$

where $d(i,j)$ denotes the distance between records r_i and r_j. Since all records are accessed only once during each merge phase, the probability of accessing each record distributed unformly on the disk is

$$p_1 = p_2 = \ldots = p_N = \frac{1}{N}$$

and

$$E(d_i) = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} |d_i - d_j|$$

$$= \frac{1}{N^2} \sum_{i=1}^{N} [\sum_{i=1}^{j} -1(j-i) + \sum_{i=j+1}^{N} (i-j)]$$

$$= \frac{1}{N^2} \sum_{i=1}^{N} [j(j-1)/2 + (N-j)(N-j+1)/2]$$

$$= \frac{1}{N^2} (\frac{N^3}{3} - \frac{N}{6})$$

$$= \frac{N}{3} - \frac{1}{6N}.$$

When N is large, the expected distance traveled by the head is proportional to $\frac{N}{3}$.

To read a record and move to another record in a randomly distributed file system, the expected movement to access N records is proportional to $N/3$. From the lemma, the disk total transaction time can be calculated for an individual record as follows:

T = seek time x (input record size/cylinder size)
+ rotational latency time
+ input record read time
+ seek time x (output record size/cylinder size)
+ rotational latency time
+ output record write time,

where: α = disk assess time,

β = average rotational latency time (= 1/2 disk rotation time),

θ_r = disk read time,
θ_w = disk write time,
δ = physical disk block size in bytes,
γ = disk cylinder size in bytes,
τ = total number of blocks (= rN/δ),
N = number of record in a file,
r = record size in bytes,
m = order. of merge

It was assumed that a block of sorted records was read from and written to contiguous cylinders to save disk access time in a partitioned sequential file system [13]. For example, a block from the i^{th} sort-merge stage may be written to the same cylinder or to the next contiguous cylinder when the size of the block is too large to write in a single cylinder. During the first sort-merge pass, the seek time is the product of α and τ and the number of blocks is equal to the number of records to be sorted, or $\tau = rN/\delta$. Thus, the seek time for the first input stage is $\alpha\tau$ x $(N/3)$. Rotational latency time is the product of β and τ. The read time is the product of θ_r and τ, where θ_r is the disk read time. After reading data from the disk, the merger writes the block of merged records to the disk. Thus, seek time has been decreased by a factor of m since individual records are merged into N/m merge blocks, and the total seek time is reduced by a factor of m. Disk write time is the product of θ_w and τ, where θ_w is the disk write time.

Table 5.1 Disk I/O Access Model Analysis

Merge Stage	Seek Time	Rotational Latency	Read Time	Seek Time	Rotational Latency	Read Time
1	$\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_r$	$\frac{N}{m}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_w$
2	$\frac{N}{m}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_r$	$\frac{N}{m^2}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_w$
3	$\frac{N}{m^2}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_r$	$\frac{N}{m^3}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_w$
	$\frac{N}{m^3}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_r$	$\frac{N}{m^4}\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_w$
	⋮	⋮	⋮		⋮	⋮
k - 1	$m^2\mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_r$	$m\ \mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_w$
$k = \log_m^N$	$m\ \mathrm{x}\frac{N}{3}\mathrm{x}\alpha\tau$	$\tau\beta$	$\tau\theta_r$	$\alpha\tau$	$\tau\beta$	$\tau\theta_w$

$$\text{Total Acess Time} = \alpha\tau + \frac{N\,\alpha\tau}{3} + 2\frac{N\,\alpha\tau}{3} + (\frac{1}{m} + \frac{1}{m^2} + \cdots + \frac{1}{m^{k-1}}) + k\tau(2\beta + \theta_r + \theta_w)$$

$$= \alpha\tau + \frac{N\,\alpha\tau}{3} + \frac{2N\,\alpha\tau}{3}(\frac{1}{m-1}\ \frac{N-m}{N}) + k\tau(2\beta + \theta_r + \theta_w)$$

$$= 2\alpha\tau + k\tau(2\beta + \theta_r + \theta_w)$$

$$= O(N^2)$$

In the second and subsequent passes through the merger, all the timing parameters become proportional to the number of merge stages. Table 5.1 shows the results for an m-way merge of N records. The total I/O access time for the merge phase is O(N^2). The sort-merge process, in isolation, is O(log$N$), while the disk I/O time is O(N^2). Therefore, disk access time is much greater than sort-merge time and the overall sort-merge process in a single disk based system is O(N^2). To overcome the problem of disk I/O access, two methods are available. First, when the records are arranged in sequential rather than random order, the total I/O time can be reduced to O(N). To arrange the records in sequential order is not practical in a real application. Parallel disks may be used to improve disk I/O performance. Second, the sort-merge system has the ability to perform disk I/O access efficiently as long as the sort-merge system has enough buffers and merge stages. Detailed observation of Table 5.1 indicates that the sort-merge process using a single disk, will be improved by removing the repeated read/write operations. If $\log_m N$ merge stages are used, the disk I/O time is O(N) and the overall sort-merge process for a single disk is O(N).

5.4.3.2 Disk-Based Sort-Merge System

In 1985, Takagi *et al.* [45] proposed a hardware sort-merge system which used a key-pointing sorter as the initial sorter and a pipelined m-way merger, as shown in Figure 5.8. The merger consisted of several intelligent disks, each of which had a simple processor and $m+1$ two-bank buffers. Problems with the synchronization of data transmissions and excessive latency times were experienced. The $m+1$ two-bank buffers were used in the disk system for an m-way merge, setting each bank at a size equal to the disk track size to avoid latency time problems. However, chip architecture, details of implementation, and the merger structure used by Takagi are unknown. In the present study a new VLSI sort-merge system including multiple disks is proposed.

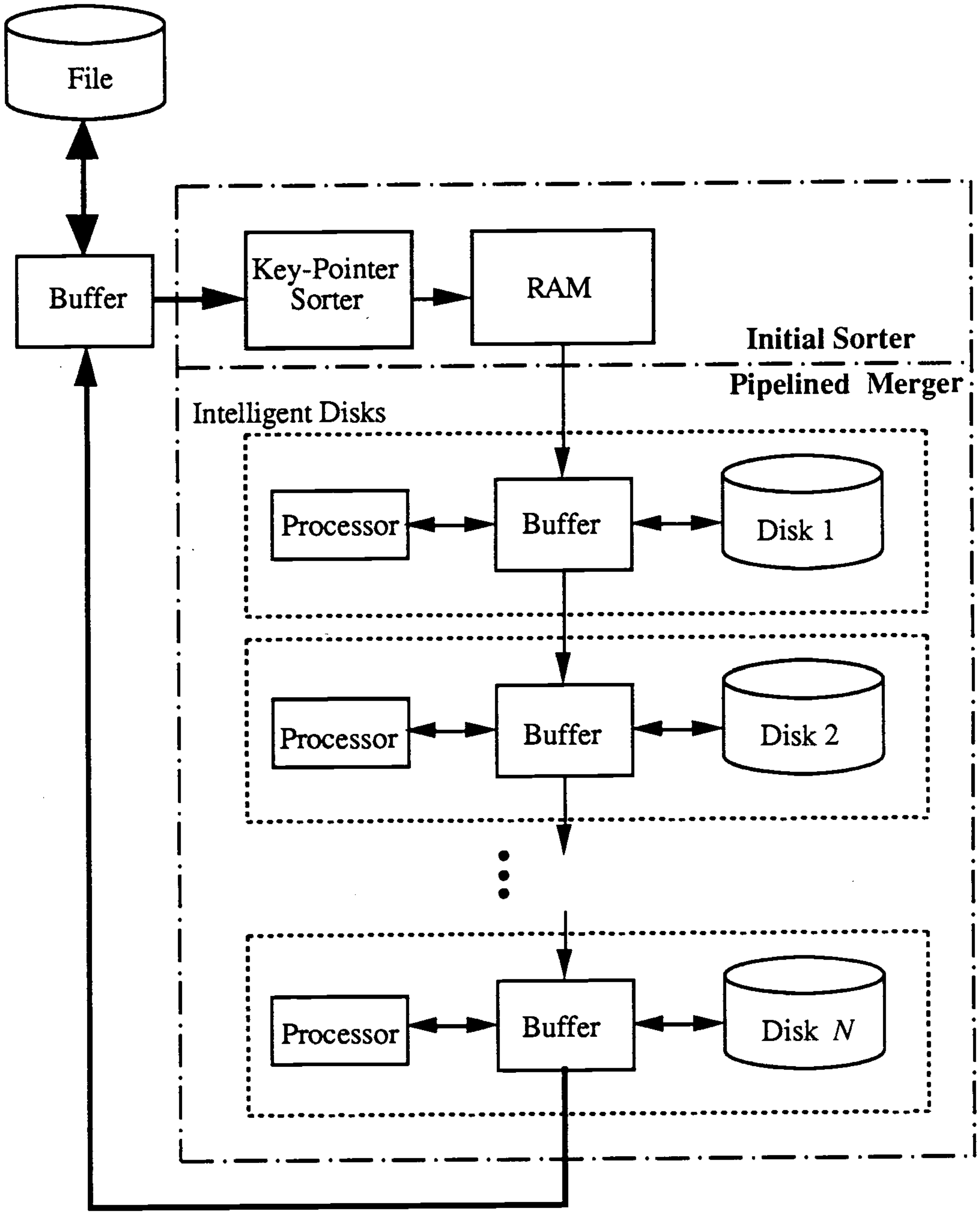


Figure 5.8 Takagi's Pipelined Sort-Merge System

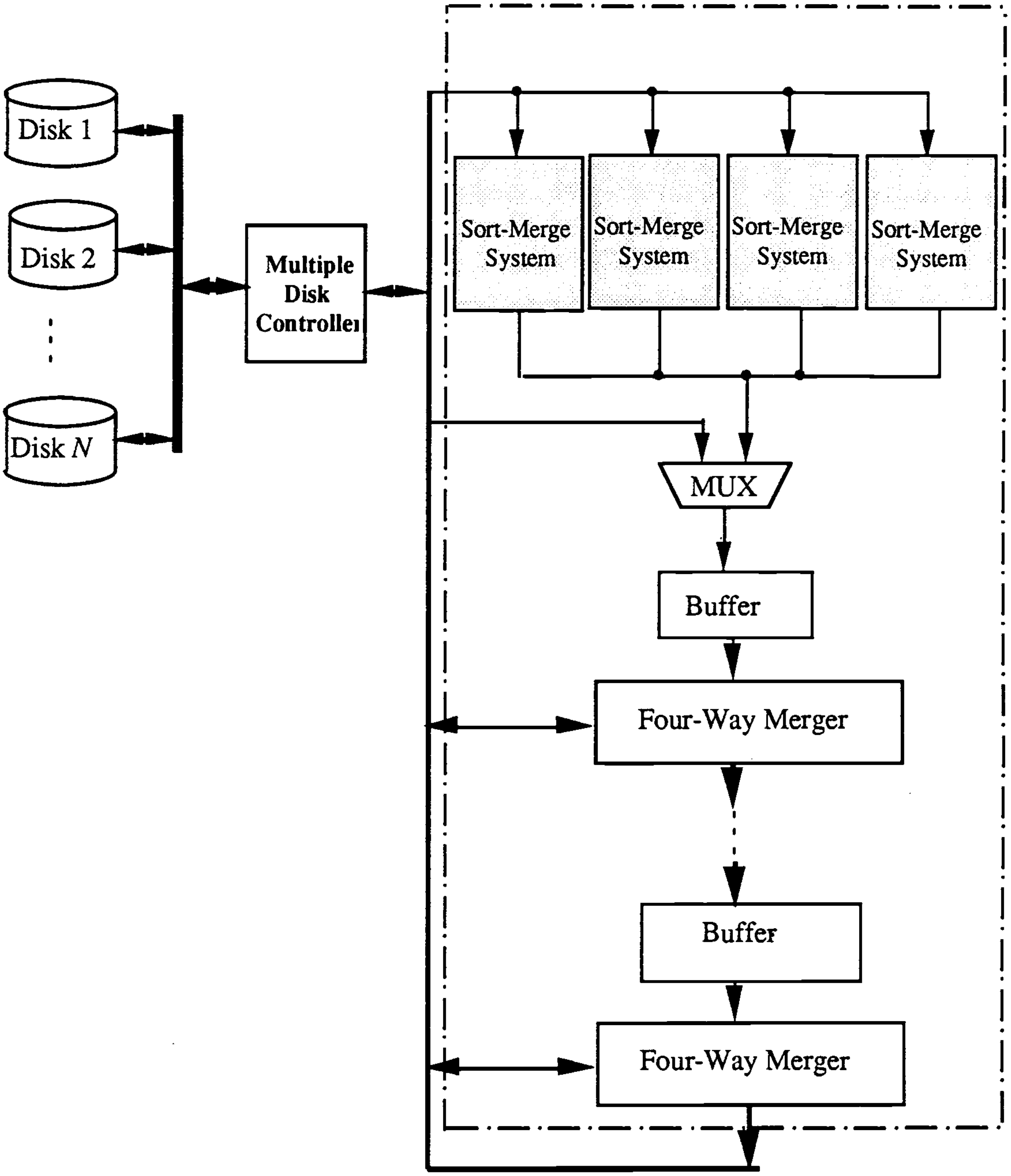


Figure 5.9 Disk-Based Sort-Merge System

The parallel sort-merge system in this section is composed of multiple sort-merge systems, each of which contains multiple rebound sorters and mergers. The system, with buffers, is attached directly to one or more disks operating in parallel. One or more disk controllers may be used in such a system. Sorting may be performed in the disk controller rather than in the CPU of the host system. The system must have sufficient buffer space to store sorted data temporarily, and the processing speed should match the disk I/O access time. In the previous section, the disk I/O access time was of O(N^2) time complexity, in contrast to O(N) time complexity for the merger when a partitioned sequential file organization was used. Thus, sorting time is faster than the disk I/O time and the disk controller performance will not be degraded during data transfer to and from the sorter. The proposed system configuration is shown in Figure 5.9.

5.5 Conclusion

The proposed single-chip sort-merge system was implemented in VLSI. The resulting chip size was 416 mils by 432 mils and the throughput rate achieved was 10.8 MHz. The single chip version contains two rebound sorters, plus a four-way merger with intermediate buffers. The single chip sort-merge system accepts inputs in the form of 8-bit half-records every clock cycle, and outputs a complete 16-bit sorted string. Various input data were applied during simulation and correct system functionality was verified. Performance data was also obtained through the use of a timing verifier. The overall time complexity achieved was O(N) and the chip area was also O(N). The system, as currently implemented, does not have the expansion capabilities described in the study.

Moreover, it requires a preprocessor to retrieve a key from each record and to add the required pointers.

Three additional methods for creating sort-merge systems were also discussed, the iterative sort-merge system, the tree sort-merge system, and the disk-based sort-merge system. These sort-merge systems can be built by combining rebound sorter, merger, or sort-merge chips for specific applications. The disk storage subsystem is the most desirable method of implementing the sort-merge system as a special processor for the rapid sorting of larger files. Large files can be initially sorted by rebound sorters and then repeatedly merged by the pipelined merger.

The disk I/O access model for the iterative pipelined merger was analyzed, resulting in an I/O complexity proportional to the number of records to be sorted. Other method has been reviewed by Kwan [24], who noted that selection of a large merge order is not optimal for files composed of a very large number of records. When a partitioned sequential file organization was used, disk I/O access time was found to be of $O(N^2)$ time complexity in contrast to the $O(\log N)$ time complexity of the sort-merge system. Disk I/O performance can be improved by increasing the number of merge stages with appropriate buffers. Although the simulated model was simplified in contrast to the complexities of actual disk I/O, the results indicate that the sort-merge system can be used with a disk subsystem. Through the implementation of efficient disk scheduling methods, disk I/O access time can be improved.

Future research based upon the proposed system should examine techniques to better model the disk storage subsystem.

Additional research should allow for the optimization of file distributions across several disks to provide maximum I/O performance.

6. CONCLUSIONS

In this chapter, key research contributions are summarized, and directions for future research are suggested.

6.1 Research Contributions

The first research objective of this study was to develop a unique parallel sort-merge system capable of sorting large files at a high throughput rate. Suitability for implementation in VLSI was a key requirement. This objective was achieved and constitutes a key research contribution. In the process, a unique 4-way merger and high performance sorter were also synthesized. The second objective, that of analyzing current parallel sorting architectures and algorithms, particularly those in which I/O functions are overlapped with sorting, was also successfully achieved. The process associated with the achievement of these objectives is summarized in the following paragraphs.

Although extensive studies have been conducted in the area of parallel sorting, few of the algorithms proposed have overlapped sorting time with input and output times. Three algorithms have achieved O(N) time complexity for implementation in VLSI. The VLSI sorter proposed by Miranker *et al.* [36] requires the smallest chip area, but achieved only relatively slow processing speed. The rebound sorter proposed by Chen *et al.* [18] was the fastest of the three algorithms and was implemented in a comparatively small chip area. The enumeration sorter of Yasuura *et al.* [50] has the largest chip area and achieved only modest processing speeds. Each of these algorithms reflected O(N) area complexity.

Following chip area and performance analysis, the rebound sorter was selected for implementation. The algorithm was modified and improved by the implementation of specialized pipelining techniques and the inclusion of expansion capability. As modified, the rebound sorter can process N 16-bit records using N processing elements. The chip designed for this system consists of eight processing elements and controllers, one pipeline controller, and one expansion multiplexer. Sequences of 8-record data streams are applied, then sorted in ascending order. The proposed chips can be chained to sort large files. File size is only limited by clock distribution and economic considerations. The core area of this sorter is 159 mils by 146 mils and the total chip area, including I/O pads, is 228 mils by 208 mils. Several sorters can be built on a single chip. For example, 72 16-bit sorting chips can be placed on a single 540 mils by 510 mils chip. The sorter processing speed was 17 MHz. Methods to expand the rebound sorter were also examined. With a special controller, the key length can be extended up to 256 bytes by the horizontal connection of parallel rebound sorters.

By itself, the rebound sorter can handle only small files economically. Compared to the cost of the merger hardware, it is also expensive to build. The merger described uses an iterative process [47] for external sorting, whereas the rebound sorter cannot. The merger algorithm and merge stages were examined for implementation in VLSI and the parallel binary merge sort and pipelined merge sort algorithms were compared. The parallel binary merge sort achieves better performance with smaller buffers than the pipelined sort-merge, but it presents a processor bandwidth problem when implemented synchronously. Therefore, processor speed limitations of the parallel binary merge sort must be considered in any hardware implementation..

A four-way block merge algorithm using presorted input data was selected for implementation. This method requires sufficient buffers for the accommodation of intermediate processing results. Buffer management schemes were studied and a double buffer method was implemented in order to reduce design costs. To implement a four-way merger in VLSI, two methods were compared: direct and indirect four-way merging. The indirect method was faster than the direct merger, and its chip area was less than that required for the two-way merger. An indirect four-way merger was implemented in VLSI at a size of 354 mils by 354 mils, including internal buffers. The processing speed achieved was 10.8 MHz.

In summary, a sort-merge system was designed and implemented on a single chip, 416 mils by 432 mils in area. The maximum frequency for this system was 10.8 MHz. The chip has not yet been fabricated. All implementation results are based on silicon compiler simulation data. The performance achieved is substantial relative to other architectures described in the literature. The system can be configured in several ways. These include an iterative sort-merge system, a tree sort-merge system, or as a disk-based sort-merge system. To implement the disk-based system, a simplified I/O access model was studied. The disk I/O access time achieved, (O(N^2) time complexity) is less than that of the sort-merge system when partitioned sequential files are used. However, the disk model can be reduced to O(N) time complexity for this organizational structure by a variety of means, including the use of a multiple disk system and methods to increase the number of sort keys available for processing.

6.2 Research Recommendations

Some of the problems and limitations of sorting which were not examined during the course of this study are considered in this section. General recommendations for future research in the area of VLSI sorting systems are also provided.

The proposed sort-merge system requires preprocessors for sorting records. The system has the potential of serving as a dedicated accelerator for special and general purpose processor hardware systems. For example, a number of parallel sorting and database hardware systems have been studied or proposed [19,32,34], but to date no functional systems which can perform both sorting and database operations have been developed. Many of the problems specific to such systems have not been addressed in the literature.

The buffer management scheme developed for this study has been discussed for merging purposes only. The buffers used in the proposed sort-merge system could be used, for example, as a disk cache. For these more general purpose activities, a dynamic allocation scheme for buffer management, such as a partitioned buffer, would have to be developed.

In disk I/O modeling, input and output data are uniformly distributed on the disk model. However, in the real world the data may be distributed in a variety of ways, such as Gaussian [49], Poisson, or binomial distributions. It is important that the statistical characteristics of the system be measured in accordance with the file organization method in use. The various disk I/O access methods and head movements reviewed by Wong [49] should be applied to the development of more practical disk I/O modeling. To evaluate the disk I/O complexity of external sorting, it is necessary to develop realistic and practical disk models, as well as analysis techniques for various file organization and disk scheduling methods. Further study should incorporate more experimental observations of disk seek times in real application environments.

The system developed in the present study is ready for fabrication. It should be physically fabricated and tested, in addition to being simulated.

BIBLIOGRAPHY

[1] B. Ahn and J. M. Murray, "A Pipelined, Expandable VLSI Sorting Engine Implemented in CMOS Technology," ISCAS'89 Proc., pp. 134-137, May 1989.

[2] S.G. Akl and H. Schmeck, "Systolic Sorting in a Sequential Input/Output Environment," Parallel Comput. vol.3, pp. 11-23, 1986.

[3] S.G. Akl, *Parallel Sorting Algorithms*, Orlando, FLA:Academic Press, 1985.

[4] K.E. Batcher,"Sorting Networks and Their Applications," AFIPS Proc. Spring Joint Comput. Conf., vol.32, pp. 307-314, Apr. 1968.

[5] J.G. Bate, "The Architecture and Design of a Parallel Sorting Engine," MS Thesis, Oregon State Univ., Corvallis, Oregon, 1988.

[6] M. Beck, D. Bitton and W.K. Wilkinson, "Sorting Large Files on a Backend Multiprocessor," IEEE Trans. on Comput. vol.37, No.7, pp. 769-778, July 1988.

[7] G. Bilardi and F.P. Preparata, "An Architecture for Bitonic Sorting with Optimal VLSI Performance," IEEE Trans. on Comput. vol.C-33, No.7 pp. 646-651, July 1984.

[8] G. Bilardi, "The Area-Time Complexity of Sorting," Ph.D Dissertation, Univ. of Illinois, Urbana, Illinois, 1985.

[9] D. Bitton, "Design, Analysis and Implementation of Parallel Sorting Algorithms." Ph.D Dissertation, Univ. of Wisconsin, Madison, Wisconsin, Dec. 1981.

[10] D. Bitton and D.J. DeWitt, "Duplicate Record Elimination in Large Data Files," ACM Trans. Database Sys. vol.8, No.2 pp. 255-265, June 1983.

[11] D. Bitton, D.J. DeWitt, D.K. Hsiao ans J. Menon, "A Taxonomy of Parallel Sorting," Computing Surveys, vol.16, No.3, pp. 287-318, Sep. 1984.

[12] N.A. Black, "Optimum Merging from Mass storage," Comm. ACM, vol.13, pp. 745-749, Dec. 1970.

[13] M. Bohl, *Introduction to IBM Direct Access Storage Devices*, Chicago, ILL:Science Research Associate, 1981.

[14] M.A. Bonuccelli, E. Lodi, and L. Pagli, "External Sorting in VLSI," IEEE Trans. Comput., vol.C-33, pp. 931-934, Oct. 1984.

[15] M.J. Carey, P.M. Hansen and C.D. Thompson, "RESST: A VLSI Implementation of a Record-Sorting Stack," Report No. UCB/CSD 82/102, U. of California, Berkeley, Apr 1982.

[16] K.M. Chung, F. Luccio and C.K. Wong, "On the complexity of sorting in Magnetic Bubble Memory Systems," IEEE Trans. on Comput. vol.C-29, No.7, pp. 553-563, July 1980.

[17] T.C. Chen and C. Tung, "Storage Management Operations in Linked Uniform Shift Register Loop," IBM J. of Res. Develop., vol.20, pp. 123-131, Mar. 1976.

[18] T.C. Chen, V.Y. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large Files," IEEE Proc., pp. 342-318, June 1978.

[19] Y. Dohi, A. Suzuki, N. Matsui, "Hardware Sorter and Its Application to Data Base Machine," Proc. 9th Annual Symp. Comput. Arch.(ACM Sigarch), vol.10, pp. 215-225, 1982.

[20] J.E. Dorband, "Sort Computation and Conservative Image Registration," Ph.D. Dissertation, Pennsylvania State Univ., Pennsylvania, Dec. 1985.

[21] S. Even, "Parallelism in Tape-Sorting," Commun. ACM vol.17, No.4, pp. 202-204, Apr. 1974.

[22] R. Francis and I.D. Mathieson, "A Bench Mark Parallel Sort for Shared Memory Multiprocessors," IEEE Trans. on Comput. vol.37, No.12, pp. 1619-1626, Dec. 1988.

[23] Silicon Compiler Inc., *Genesil System Compiler Library*, vol. I, II and III, 1988.

[24] S.K. Kwan, "External Sorting: I/O Analysis and Parallel Processing Techniques," Ph.D Dissertation, Univ. of Washington, Seattle, Washington, 1986.

[25] M. Kitsuregawa, S. Fushirra, H. Tanaka, T. Moto-oka, "Memory Management Algorithms in Pipeline Merge Sorter," Database Machines, Berlin:Springer-Verlag, pp. 208-232, 1985.

[26] D.E. Knuth, *The Art of Computer Programming:Sorting and Searching, vol.3*, Reading, MA:Addison-Wesley, 1973.

[27] H.-W. Lang and M. Schimmler, "A Method for Realistic Comparisons of Sorting Algorithms for VLSI," IEEE Proc. of the Int'l Conf. Kyoto, Japan, pp. 236-40, May 1985.

[28] H.-W. Lang, M. Schimmler, H. Schmeck, and H. Schroder, "Systolic Sorting on a Mesh-Connected Network," IEEE Trans. Comput. vol.C-34, pp. 652-658, July 1985.

[29] D.T. Lee, H. Chang and C.K. Wong, "An On-chip Compare/Steer Bubble Sorter," IEEE Trans. Comput. vol.C-30, pp. 396-405, June 1981.

[30] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting." IEEE Trans. on Comput. vol.C-34, No.4, pp. 344-353, Apr. 1985.

[31] E.E. Linderstrom and J.C. Vitter, "The Design and Analysis of Bucket Sort for Bubble Memory Secondary Storage," IEEE Trans. on Comput., vol.C-34, No.3, pp. 218-232, Mar. 1985.

[32] M. Maekawa, "Parallel Sort and Join for High Speed Database Machine Operations," AFIPS Conf. Proc:Natl Comput. Conf, pp. 512-520, 1981.

[33] R.E. Matick, *Computer Storage Systems and Technology,* New York, NY:John Wiley & Sons, 1977.

[34] J. Menon, "Sorting and Join Algorithm for Multi-processor Database Machines," Database Machines Berlin:Springer-Verlag, pp. 289-322, 1986.

[35] F. Meyer auf der Heide and A. Wigderson, "The Complexity of Parallel Sorting," IEEE Proc., pp532-540, Apr. 1985.

[36] G. Miranker, L. Tang, and C.K. Wong, "A 'Zero-Time' VLSI Sorter," IBM J. of. Res. Develop. v-27, pp. 140-148, Mar. 1983.

[37] M. Negri and G. Pelagatti, "Join during Merge: An Improved Sort Based Algorithm," Inf. Proc. Letters, vol.21, No.2, pp. 11-16, July 1985.

[38] R.J. Offen, *VLSI Image Processing,* New York, NY:McGraw-Hill, 1985.

[39] S. Rajgopal, S. Ghatak, J. McNair, S. Kumar, and D. Bouldin, "A VLSI Implementation of the Parallel Enumeration Sort Technique," IEEE VLSI Tech. Bulletin, vol.1, No.3, pp. 35-42, Dec. 1986.

[40] L. Rudolph, "A Robust Sorting Network," IEEE Trans. Comput. vol.C-34, pp. 326-335, Apr. 1985.

[41] M. Satyanarayanan, *Modeling Storage Systems,* Ann Arbor, MICH:UMI Research Press, 1986.

[42] M. Segal, "Hardware Sorting Chip Steps up Software Face," Electronic Design, pp. 85-91, June 1986.

[43] P.H. Singgih, H.B. Demuth, M.T. Hagan, and R.L. Wainwrite, "Parallel Merge-Sort Algorithms on the HEP," ACM Fourteenth Annual Comput. Science Conf.: CSC '86 Proceedings, pp. 237-244, Feb. 1986.

[44] H.S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. Comput., vol.C-20, pp. 153-161, Feb. 1971.

[45] N. Takagi and C.K. Wong, "A Hardware Sort-Merge System," IBM J. of Res. Develop. v-29, pp. 49-67, Jan 1985.

[46] C.D. Thompson, "The VLSI Complexity of Sorting," IEEE Trans. Comput. vol.C-32, pp. 1171-1184, Dec. 1983.

[47] S. Todd, "Algorithm and Hardware for a Merge Sort using Multiple Processors," IBM J. of Res. and Develop. vol.22, No.5, pp. 509-517, Sep. 1978.

[48] C. Tung, T.C. Chen, and H. Chang, "Bubble Ladder for Information Processing," IEEE Trans. on Magn., vol. Mag-11, No.5, pp. 1163-1165, Sep. 1975.

[49] C.K. Wong, *Algorithmic Studies in Mass Storage Systems,* Rockville, MD:Computer Science Press, 1983.

[50] H. Yasuura, N. Takagi and A. Yajima, "The Parallel Enumeration Sorting Scheme for VLSI," IEEE Trans. Comput. vol.C-31, pp. 1192-1201, Dec. 1982.

APPENDICES

APPENDIX A

LAYOUT AND ROUTE PLOTS

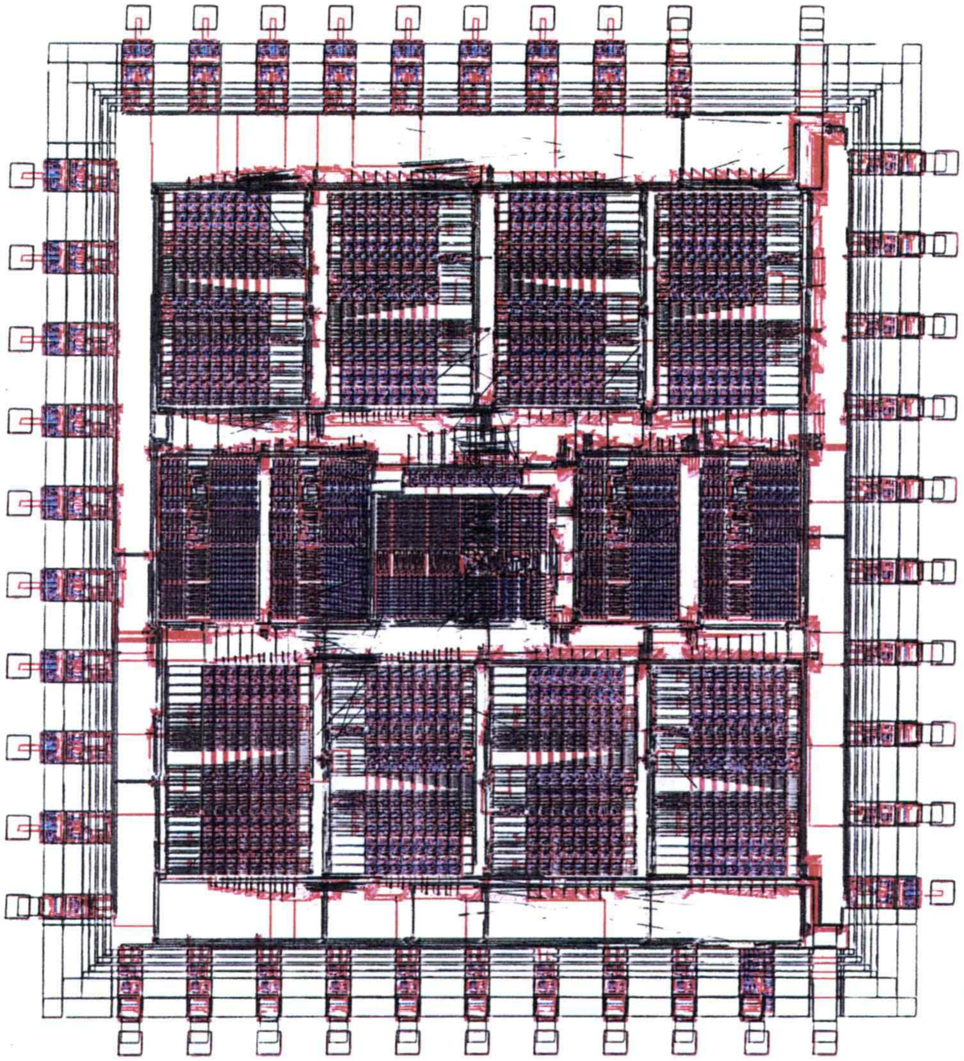
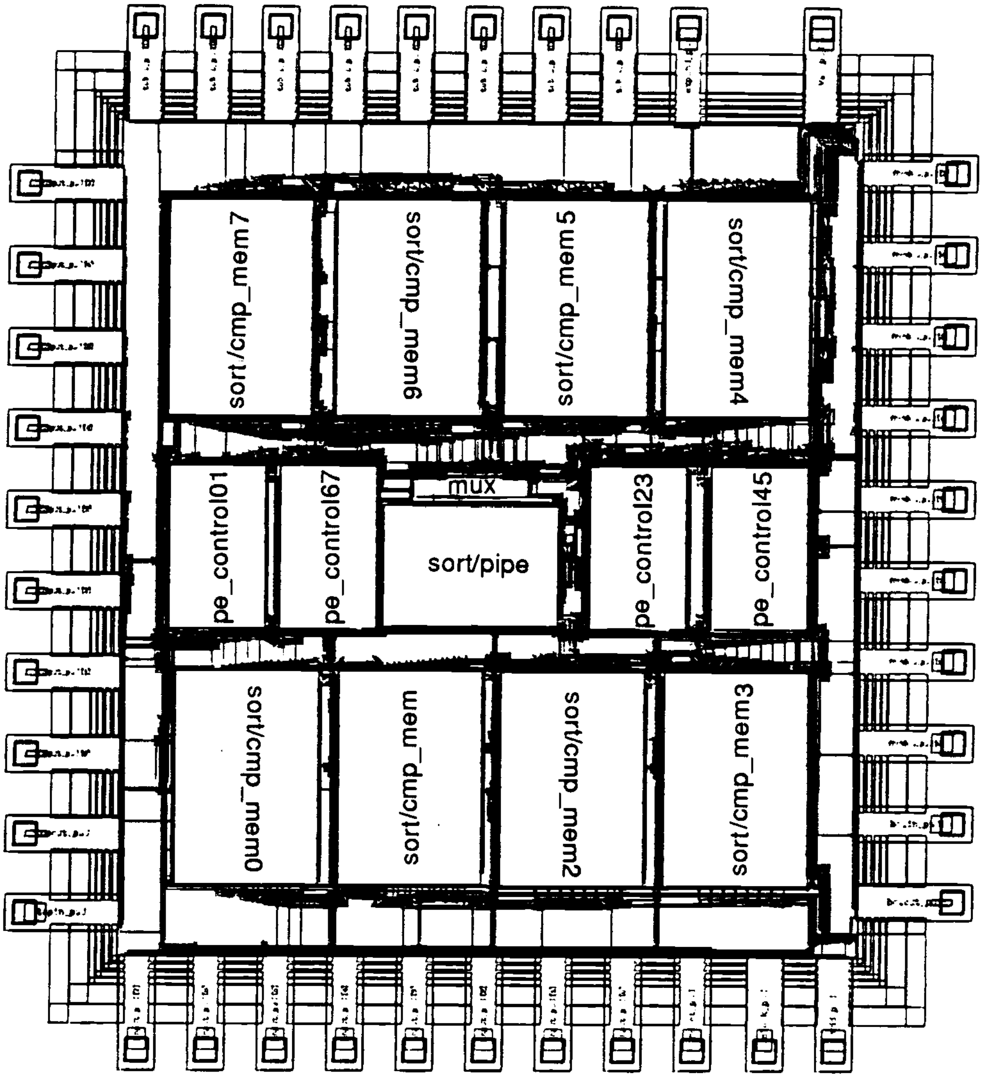


Figure A.1 Rebound Sorter Layout Plot



Silicon Compiler Systems

Object: sort

Date: Apr 23 89 8 35

Account: genahn

Sub-account: dhn

Scale: 26.19

Page: 1 of: 1

Object size (mils): 228.40 x 208.02

Window limits (mils): None

Figure A.2 Rebound Sorter Route Plot

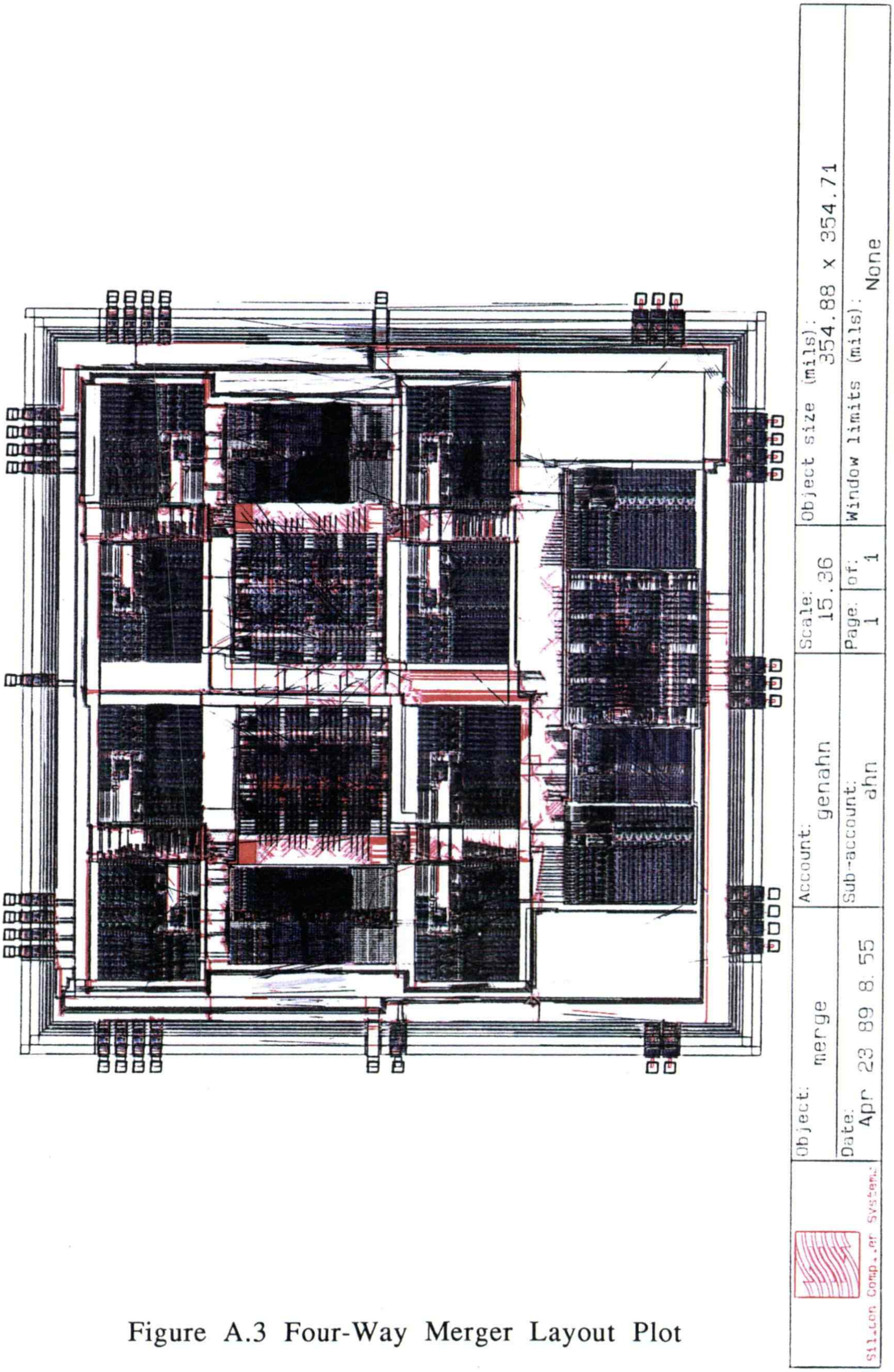


Figure A.3 Four-Way Merger Layout Plot

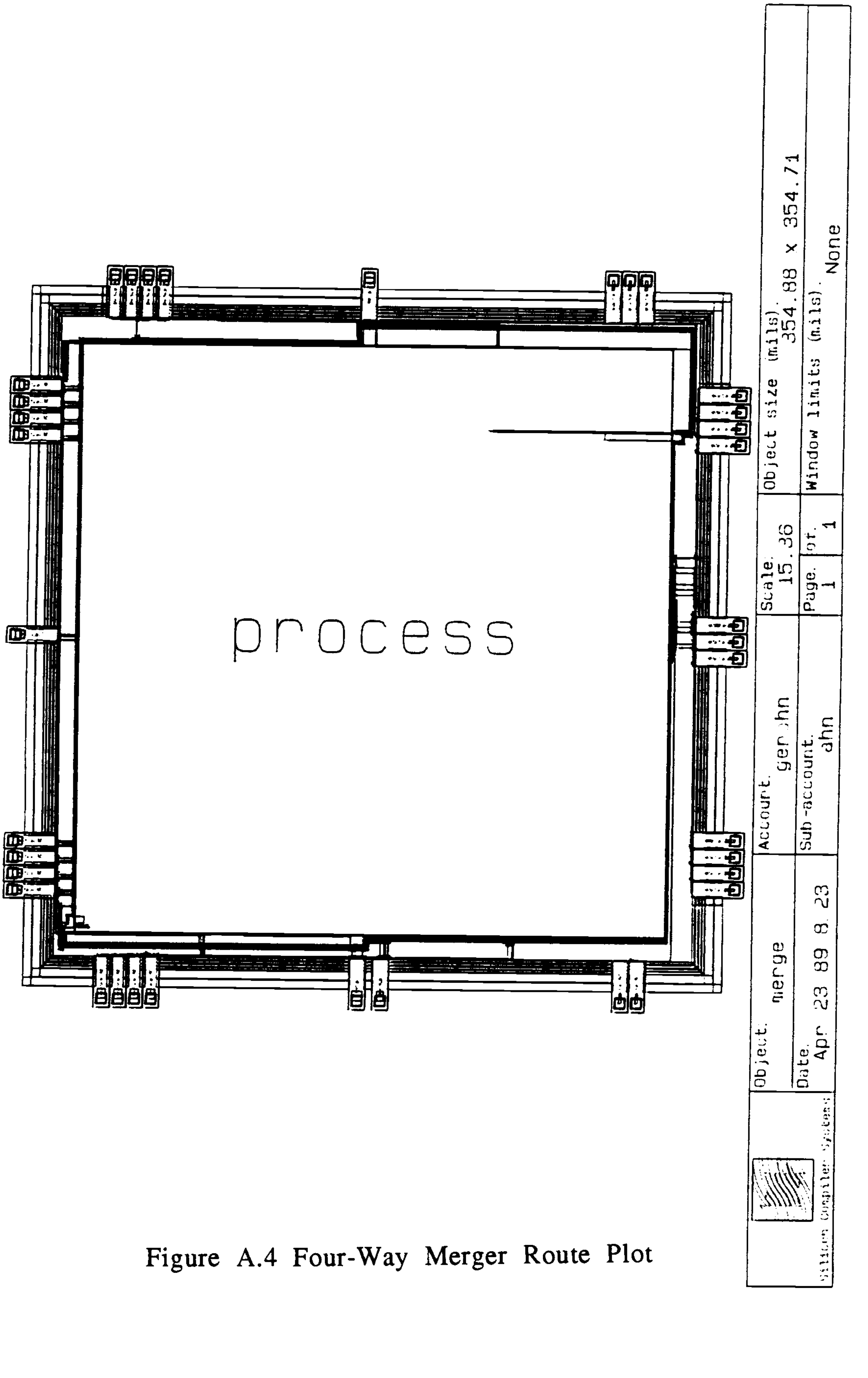


Figure A.4 Four-Way Merger Route Plot

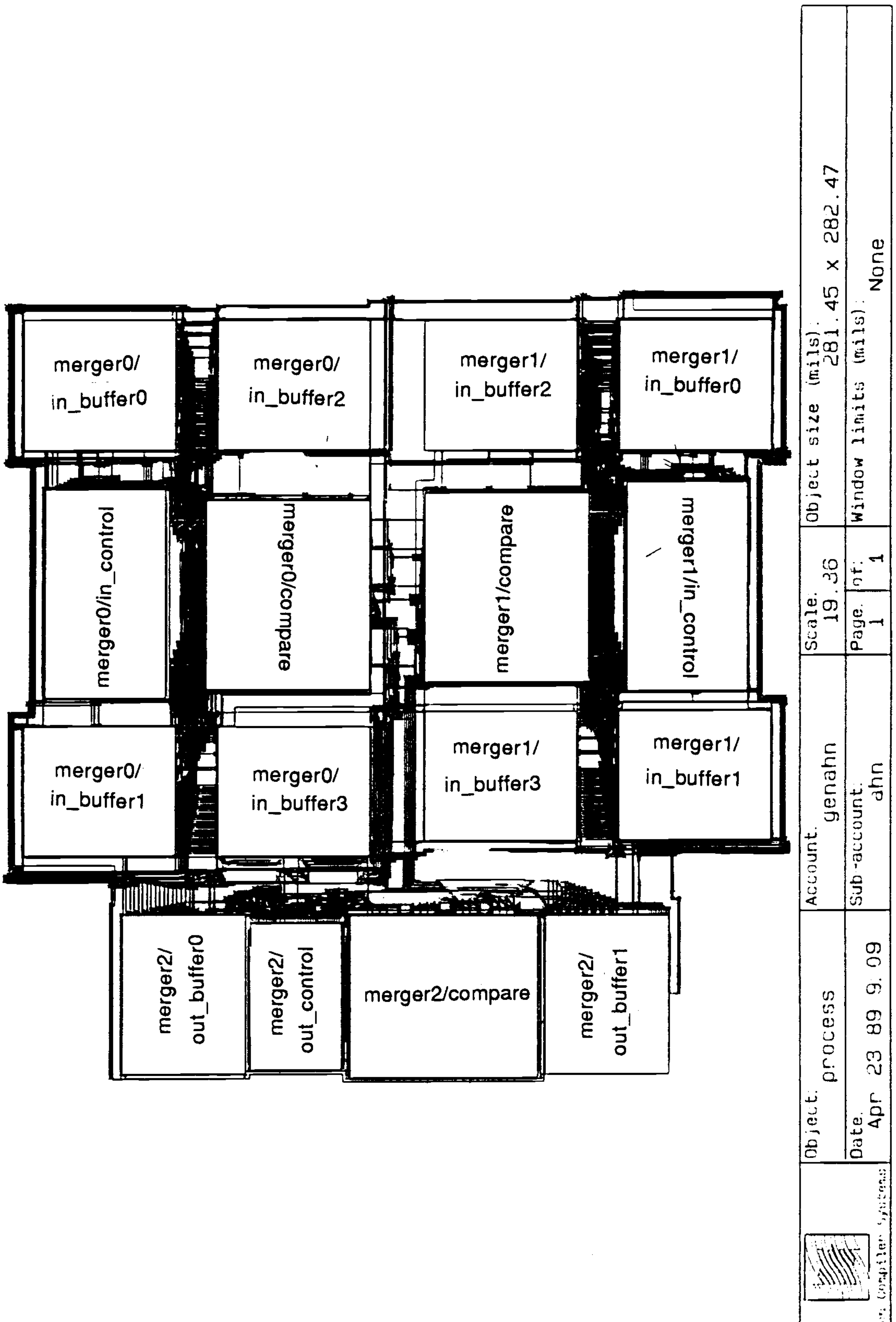
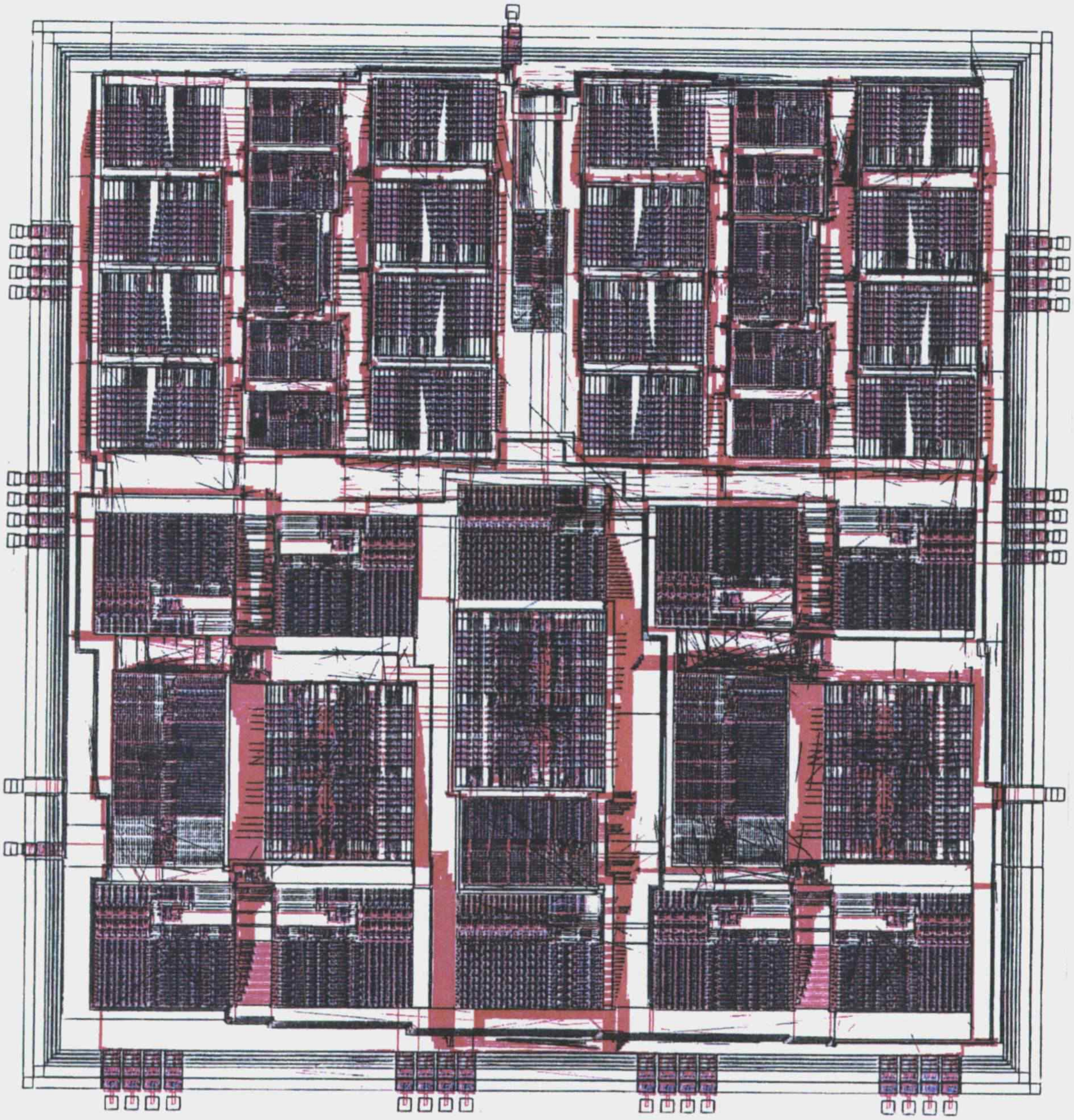
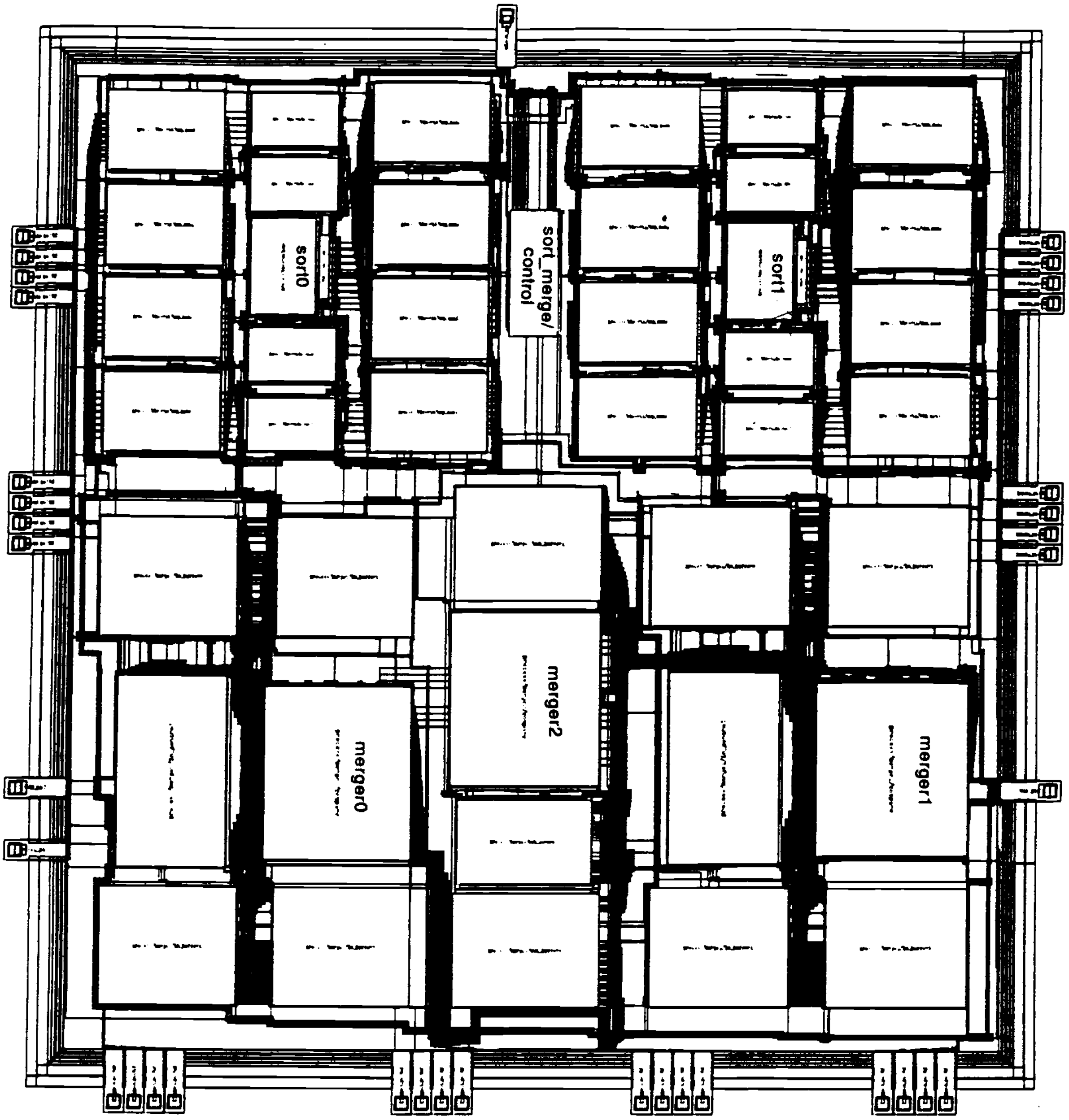


Figure A.4 Four-Way Merger Route Plot (*continued*)



Silicon Compiler Systems	Object: sort_merge	Account: genahn	Scale: 18.50		Object size (mils): 415.37 x 430.95
	Date: Apr 23 88 13:05	Sub-account: ahn	Page: 1	of 1	Window limits (mils): None

Figure A.5 Sort-Merge System Layout Plot



	Object: sort_merge	Account: genahn	Scale: 18.50		Object size (mils): 415.97 x 430.99
	Date. Apr 23 89 17:12	Sub-account: ahn	Page: 1	of: 1	Window limits (mils): None

Figure A.6 Sort-Merge System Route Plot

APPENDIX B

TIMING ANALYSIS

Table B.1 Rebound Sorter Timing Analysis

```
**************************************************************************
                Genesil Version v7.0 -- Thu Nov 10 22:44:31 1988
Chip: ~genahn/ahn/rebound                                 Timing Analyzer
**************************************************************************
CLOCK REPORT MODE
--------------------------------------------------------------------------
Fabline: VTI_CN20A                        Corner: TYPICAL
  Junction Temperature:75 degree C        Voltage:5.00v
  External Clock: clockpad
 Included setup files: default setup file
--------------------------------------------------------------------------
CLOCK TIMES (minimum)
Phase 1 High:    26.4   ns               Phase 2 High:    29.9   ns
              ----------                              ----------
Cycle (from Ph1):    53.1   ns           Cycle (from Ph2):    19.8   ns
                  ----------                              ----------
Minimum Cycle Time:    56.3   ns         Symmetric Cycle Time:    59.9  ns
                     ----------                                ---------
--------------------------------------------------------------------------
                             CLOCK WORST CASE PATHS
Minimum Phase 1 high time is    26.4   ns set by:
                             ----------
  ** Clock delay: 4.4ns (30.8-26.4)
   Node                               Cumulative Delay      Transition
   process/cmp_mem0/(internal)             30.8                rise
   process/cmp_mem0/port0_EXT2[1]          28.6                fall
   process/cmp_mem1/port9_EXT2[1]          28.6                fall
   <ocess/cmp_mem1/port9_EXT2[1]'          28.5                fall
   process/cmp_mem1/inter7_SEL2            17.7                fall
   process/pe_fsm01/V_P1                   17.7                fall
   process/pe_fsm01/V_P1'                  17.0                fall
   process/pe_fsm01/PHASE_B                 5.3                rise
   clock_pad/phasea                         2.0                rise
   clockpad                                 0.0                rise

Minimum Phase 2 high time is    29.9   ns set by:
                             ----------
  ** Clock delay: 5.3ns (35.2-29.9)
   Node                               Cumulative Delay      Transition
   process/pe_fsm45/(internal)             35.2                fall
   process/pe_fsm45/GT0                    30.7                fall
   process/cmp_mem4/sub5_COUT              30.7                fall
   process/cmp_mem4/sub5_COUT'             30.4                fall
   process/cmp_mem4/port0_EXT1[0]          11.9                fall
   process/cmp_mem3/port9_EXT1[0]          11.9                fall
   <ocess/cmp_mem3/port9_EXT1[0]'          11.3                fall
   <ocess/cmp_mem3/inter8_VAL1[0]           9.1                fall
   process/cmp_mem3/PHASE_B                 6.3                rise
   clock_pad/phaseb                         3.7                rise
   clockpad                                 0.0                fall
```

Table B.1 Rebound Sorter Timing Analysis (*Continued*)

```
Minimum cycle time (from Ph1) is   53.1    ns set by:
                                 ----------
  ** Clock delay: 6.1ns (32.6-26.5) cycle_sharing disabled
   Node                           Cumulative Delay      Transition
   process/pe_fsm01/(internal)         59.0                fall
   process/pe_fsm01/GT0                54.5                fall
   process/cmp_mem0/sub5_COUT          54.5                fall
   process/cmp_mem0/sub5_COUT'         54.0                fall
   process/cmp_mem0/inter4_IV2[1]      36.6                fall
   <ocess/cmp_mem0/inter2_VAL2[1]      35.5                rise
   *process/cmp_mem0/(internal)        31.9                fall
   <ocess/cmp_mem0/inter1_VAL2[1]      31.3                rise
   process/cmp_mem0/port0_EXT2[1]      28.4                rise
   process/cmp_mem1/port9_EXT2[1]      28.4                rise
   <ocess/cmp_mem1/port9_EXT2[1]'      28.2                rise
   process/cmp_mem1/inter7_SEL2        17.7                fall
   process/pe_fsm01/V_P1               17.7                fall
   process/pe_fsm01/V_P1'              17.0                fall
   process/pe_fsm01/PHASE_B             5.3                rise
   clock_pad/phasea                     2.0                rise
   clockpad                             0.0                rise

Minimum cycle time (from Ph2) is   19.8    ns set by:
                                 ----------
  ** Clock delay: 5.3ns (15.2-9.9) cycle_sharing disabled
   Node                           Cumulative Delay      Transition
   process/cmp_mem4/(internal)         31.3                rise
   <ocess/cmp_mem4/inter1_VAL2[1]      30.4                fall
   process/cmp_mem4/port0_EXT2[1]      27.6                fall
   process/cmp_mem5/port9_EXT2[1]      27.6                fall
   process/cmp_mem5/inter7_SEL2        16.3                rise
   process/pe_fsm45/V_P1               16.3                rise
   process/pe_fsm45/V_P1'              15.9                rise
   *process/pe_fsm45/(internal)        13.2                fall
   process/pe_fsm45/PHASE_B             4.4                fall
   clock_pad/phasea                     2.0                fall
   clockpad                             0.0                fall
-------------------------------------------------------------------------
```

Table B.2 Four-Way Merger Timing Analysis

```
*************************************************************************
                Genesil Version v7.0 -- Sat Apr 22 21:13:23 1989
Chip: ~genahn/ahn/merge                                  Timing Analyzer
*************************************************************************
CLOCK REPORT MODE
-------------------------------------------------------------------------
Fabline: VTI_CN20A                       Corner: TYPICAL
  Junction Temperature:75 degree C       Voltage:5.00v
  External Clock: clock
 Included setup files: default setup file
-------------------------------------------------------------------------
                             CLOCK TIMES (minimum)
Phase 1 High:    46.4   ns              Phase 2 High:    31.4   ns
              ----------                              ----------
Cycle (from Ph1):   19.5   ns           Cycle (from Ph2):   38.7   ns
                 ----------                              ----------
Minimum Cycle Time:   77.8   ns         Symmetric Cycle Time:    92.8  ns
                    ----------                                ---------
-------------------------------------------------------------------------
                             CLOCK WORST CASE PATHS
Minimum Phase 1 high time is    46.4   ns set by:
                              ----------
  ** Clock delay: 18.3ns (64.7-46.4)
   Node                               Cumulative Delay      Transition
   <merge2/out_buffer0/(internal)         64.7                 rise
   <ge2/out_buffer0/out_datain[3]         57.7                 fall
   </merge0/compare/port7_EXT1[3]         57.6                 fall
   <merge0/compare/port7_EXT1[3]'         57.2                 fall
   <ss/merge0/compare/inter5_SEL1         41.5                 rise
   <cess/merge0/compare/sub3_COUT         41.5                 rise
   <ess/merge0/compare/sub3_COUT'         41.3                 rise
   </merge0/compare/port0_EXT1[6]         21.8                 rise
   <merge0/in_buffer2/outdatal[6]         21.8                 rise
   <erge0/in_buffer2/outdatal[6]'         21.2                 rise
   <cess/merge0/in_buffer2/phaseb         15.2                 fall
   clock_pad/phaseb                        2.6                 fall
   clock                                   0.0                 rise

Minimum Phase 2 high time is    31.4   ns set by:
                              ----------
  ** Clock delay: 15.1ns (46.5-31.4)
   Node                               Cumulative Delay      Transition
   </merge0/in_buffer1/(internal)         46.5                 rise
   </merge0/in_buffer1/address[2]         33.2                 fall
   <erge0/in_control/count_out[3]         33.2                 fall
   <rge0/in_control/count_out[3]'         29.8                 fall
   <cess/merge0/in_control/phaseb         17.7                 rise
   clock_pad/phaseb                        3.7                 rise
   clock                                   0.0                 fall
```

Table B.2 Four-Way Merger Timing Analysis (*Continued*)

```
Minimum cycle time (from Ph1) is   19.5    ns set by:
                                 ----------
  ** Clock delay: 17.6ns (27.4-9.8) cycle_sharing disabled
   Node                            Cumulative Delay     Transition
   </merge1/in_control/(internal)       31.4                fall
   <erge1/in_control/out_addr0[1]       30.3                rise
   <rge1/in_control/out_addr0[1]'       28.1                rise
   *<merge1/in_control/(internal)       25.4                fall
   <cess/merge1/in_control/phaseb       15.3                fall
   clock_pad/phaseb                      2.6                fall
   clock                                 0.0                rise

Minimum cycle time (from Ph2) is   38.7    ns set by:
                                 ----------
  ** Clock delay: 7.5ns (46.2-38.7)
   Node                            Cumulative Delay     Transition
   <merge2/out_control/(internal)       46.2                rise
   process/merge2/out_control/GT2       42.7                fall
   <ess/merge2/compare2/sub3_COUT       42.7                fall
   <ss/merge2/compare2/sub3_COUT'       42.1                fall
   <merge2/compare2/port0_EXT2[6]       22.9                rise
   <2/out_buffer1/out_dataout0[6]       22.9                rise
   </out_buffer1/out_dataout0[6]'       22.6                rise
   <ess/merge2/out_buffer1/phasea        6.2                fall
   clock_pad/phasea                      2.0                fall
   clock                                 0.0                fall
--------------------------------------------------------------------------
```

Table B.3 Sort-Merge System Timing Analysis

```
************************************************************************
              Genesil Version v7.0 -- Tue Jan  3 23:20:19 1989
Chip: ~genahn/ahn/sort_merge                              Timing Analyzer
************************************************************************
CLOCK REPORT MODE
------------------------------------------------------------------------
Fabline: VTI_CN20A                       Corner: TYPICAL
  Junction Temperature:75 degree C       Voltage:5.00v
  External Clock: clockpad
 Included setup files: default setup file
------------------------------------------------------------------------
                            CLOCK TIMES (minimum)
Phase 1 High:    46.2   ns              Phase 2 High:    36.8   ns
              ----------                              ----------
Cycle (from Ph1):    41.7   ns          Cycle (from Ph2):    37.4   ns
                 ----------                               ----------
Minimum Cycle Time:    83.0   ns        Symmetric Cycle Time:    92.3  ns
                    ----------                                ---------
------------------------------------------------------------------------
                            CLOCK WORST CASE PATHS
Minimum Phase 1 high time is    46.2   ns set by:
                             ----------
  ** Clock delay: 22.7ns (68.9-46.2)
   Node                              Cumulative Delay      Transition
   <merge2/out_buffer1/(internal)        68.9                 rise
   <ge2/out_buffer1/out_datain[9]        61.8                 fall
   </merge1/compare/port7_EXT1[9]        61.8                 fall
   <merge1/compare/port7_EXT1[9]'        61.4                 fall
   <ss/merge1/compare/inter5_SEL1        45.6                 rise
   <cess/merge1/compare/sub3_COUT        45.6                 rise
   <ess/merge1/compare/sub3_COUT'        44.9                 rise
   </merge1/compare/port0_EXT2[6]        25.5                 rise
   <merge1/in_buffer1/outdata1[6]        25.5                 rise
   <erge1/in_buffer1/outdata1[6]'        24.8                 rise
   <cess/merge1/in_buffer1/phaseb        18.8                 fall
   clock_pad/phaseb                       2.6                 fall
   clockpad                               0.0                 rise

Minimum Phase 2 high time is    36.8   ns set by:
                             ----------
  ** Clock delay: 13.9ns (50.7-36.8)
   Node                              Cumulative Delay      Transition
   <s/rbsort1/pe_fsm45/(internal)        50.7                 fall
   process/rbsort1/pe_fsm45/GT0          46.2                 fall
   <ss/rbsort1/cmp_mem4/sub5_COUT        46.2                 fall
   <s/rbsort1/cmp_mem4/sub5_COUT'        45.9                 fall
   <bsort1/cmp_mem4/port0_EXT1[0]        27.4                 fall
   <bsort1/cmp_mem3/port9_EXT1[0]        27.4                 fall
   <sort1/cmp_mem3/port9_EXT1[0]'        26.8                 fall
   <sort1/cmp_mem3/inter8_VAL1[0]        24.6                 fall
   <cess/rbsort1/cmp_mem3/PHASE_B        21.8                 rise
   clock_pad/phaseb                       3.7                 rise
   clockpad                               0.0                 fall
```

Table B.3 Sort-Merge System Timing Analysis (*Continued*)

```
Minimum cycle time (from Phl) is     41.7    ns set by:
                                   ----------
  ** Clock delay: 21.3ns (42.2-20.8) cycle_sharing disabled
   Node                             Cumulative Delay       Transition
   <s/rbsort0/pe_fsm01/(internal)        56.1                 fall
   process/rbsort0/pe_fsm01/GT0          51.6                 fall
   <ss/rbsort0/cmp_mem0/sub5_COUT        51.6                 fall
   <s/rbsort0/cmp_mem0/sub5_COUT'        51.0                 fall
   <bsort0/cmp_mem0/inter4_IV2[7]        44.3                 rise
   <sort0/cmp_mem0/inter2_VAL2[7]        43.0                 fall
   *</rbsort0/cmp_mem0/(internal)        40.9                 rise
   <sort0/cmp_mem0/interl_VAL2[7]        40.0                 fall
   <bsort0/cmp_mem0/port0_EXT2[7]        37.2                 fall
   <bsort0/cmp_meml/port9_EXT2[7]        37.2                 fall
   <sort0/cmp_meml/port9_EXT2[7]'        37.1                 fall
   </rbsort0/cmp_meml/inter7_SEL2        26.3                 fall
   process/rbsort0/pe_fsm01/V_P1         26.2                 fall
   process/rbsort0/pe_fsm01/V_P1'        25.5                 fall
   <cess/rbsort0/pe_fsm01/PHASE_B        13.8                 rise
   clock_pad/phasea                       2.0                 rise
   clockpad                               0.0                 rise

Minimum cycle time (from Ph2) is     37.4    ns set by:
                                   ----------
  ** Clock delay: 19.8ns (57.2-37.4)
   Node                             Cumulative Delay       Transition
   </mergel/in_buffer2/(internal)        57.2                 fall
   </mergel/in_buffer2/address[2]        37.6                 fall
   <ergel/in_control/count_out[3]        37.5                 fall
   <rgel/in_control/count_out[3]'        34.1                 fall
   <cess/mergel/in_control/phaseb        22.0                 rise
   clock_pad/phaseb                       3.7                 rise
   clockpad                               0.0                 fall
-----------------------------------------------------------------------------
```