A hardware sort-merge system

by N. Takagi
C. K. Wong

A hardware sort-merge system which can sort large files rapidly is proposed. It consists of an initial sorter and a pipelined merger. In the initial sorter, record sorting is divided into two parts: key-pointer sorting and record rearranging. The pipelined merger is composed of several intelligent disks each of which has a simple processor and some buffers. The hardware sort-merge system can sort files of any size by using the pipelined merger repeatedly. The key-pointer sorting circuit in the initial sorter requires only unidirectional connections between neighboring cells, instead of the usual bidirectional ones. The initial sorter can also generate sorted sequences longer than its capacity so that the number of merging passes can be reduced. A new data management scheme is proposed to run all merging passes in a pipelined fashion.

1. Introduction

Sorting is one of the most important operations in data processing systems. Much research has been carried out in this regard [1, 2]. However, it still takes a great deal of time to sort large files. For example, major banks currently spend two hours or more every night to sort large files (on the order of several megabytes) using large computers, in order to process their demand deposit accounts [3]. It is estimated that within this decade files to be sorted will become more than ten times larger, and then each sort will take 10–15 hours [3]. Thus it is necessary to develop a hardware sorting system which can sort large files more rapidly.

In view of advances in VLSI technology, various hardware sorters have been proposed [4–13]. However, most of them are for internal sorting only, and the size of files which can be sorted is limited by their capacity. To sort large files, external sorting is a necessity.

In this paper, we propose a hardware sort-merge system, *the pipelined sort-merger*, which consists of an initial sorter and a pipelined merger. The initial sorter is continuously fed records to be sorted from a secondary memory device and outputs sorted sequences consecutively to the pipelined merger, which in turn merges them into a single output sequence. In the initial sorter, we divide record sorting into key-pointer sorting and record rearranging. The sorting operation itself is completely overlapped by the input/output of the records. Furthermore, it can sort different sequences in a pipelined way. More specifically, while one sorted sequence is being output, a new sequence can be input (and sorted).

The pipelined merger is composed of several intelligent disks. Each disk has some buffers and a simple processor. All merging passes are run in a pipelined fashion and each pass is supported by a separate intelligent disk.

The idea of pipelined merging was first proposed by Even [14], who used tapes. Later Todd [9] adapted it to RAM and bubble memories. Here we consider disks since sorting involves only relatively simple operations, and current disk storage systems, such as the IBM 3380 and its controller, the IBM 3880, provide enough intelligence to perform sorting on them directly, freeing the CPU for other processing.

However, to build the merger using disks involves some difficult problems, such as synchronizing data transmissions and avoiding latency time. To resolve these, we attach $m + 1$ two-bank buffers to each disk (for an m-way merge) and let

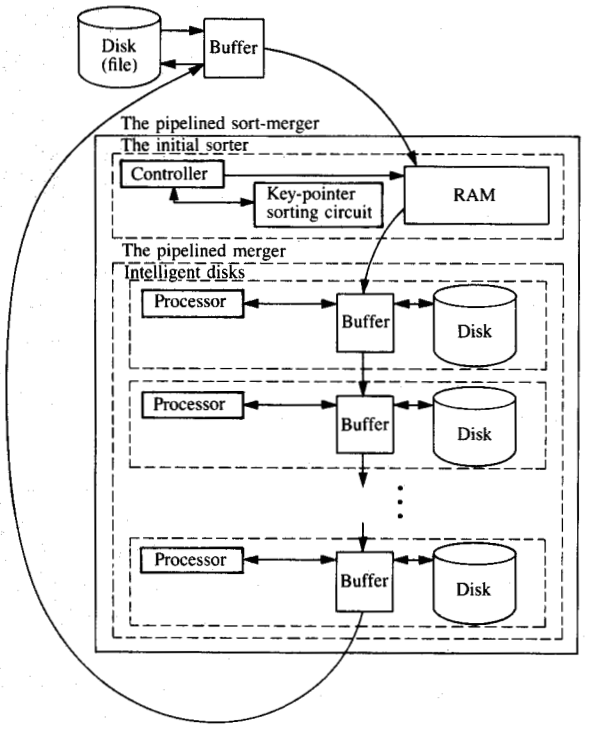


Figure 1

The pipelined sort-merger.

each bank size be the track size of the disk to avoid latency time. Finally, a new data management scheme is developed to run all merging passes in a pipelined fashion.

It should also be pointed out that our key-pointer sorting circuit is simpler than similar sorting circuits in that we require only two unidirectional connections between a cell and its neighbors instead of bidirectional connections. Also our initial sorter is, to our knowledge, the first one which can generate sorted strings which are longer than the capacity of the sorter so that the number of merging passes can be decreased. In fact, it can produce, on the average, sorted sequences of $2 \times n$ records, where n is its capacity.

When the pipelined merger is composed of k intelligent disks and an m-way merge is performed, it can merge m^k initial sorted sequences at a time. Therefore, the pipelined sort-merger can sort about $2n \times m^k$ records. We also show how to sort a file which contains more records by using the merger repeatedly.

In the next section, after some preliminary remarks, we describe the overall structure of the sort-merger. In Section 3, we discuss the initial sorter in more detail. We also present the modifications needed to generate sorted sequences longer than its capacity. In Section 4, we are concerned with the merger.

2. Overall structure

For convenience, we call each package of information a *record*. Each record contains a special field called a *key*. A set of records forms a *file*. Sorting means to rearrange the original file so that the records are ordered by their key values. We call a sorted record sequence a *run*.

Let the number of records in a file be N (N is very large). Also, let the size of each record and each key be L and l bytes, respectively. For ease of discussion, we first assume that L and l are fixed. Later, we consider the general case. Files to be sorted are stored in secondary memory devices and are transmitted to the hardware sort-merge system serially.

• *The pipelined sort-merger*

The pipelined sort-merger consists of the initial sorter and the pipelined merger. **Figure 1** is a schematic of the sort-merger. The initial sorting pass is supported by the initial sorter, and each merging pass is supported by a separate intelligent disk. A file to be sorted is transmitted from a secondary memory device to the sort-merger, where it is sorted, and is transmitted back to the secondary memory device. In the sort-merger, data transmission for several passes is done in parallel. Furthermore, input/output and sorting are overlapped, and output starts almost immediately after the sort-merger is filled. Thus, the total sorting time is reduced. (See **Figure 2**.) Note that not only are all the passes run in an overlapped fashion, but also the initial sorting time is reduced by using our initial sorter.

It should be pointed out that the sorter proposed in [13] also does a merge-sort, except that the sorter just takes the place of the CPU and main memory in conventional merge methods; it does a 2^r-way merge in one merging pass. Since it requires 2^r buffers (each at least the track size of a disk to avoid latency time) to do a 2^r-way merge, r is bounded by the RAM size. Thus the file to be sorted must be transmitted several times between the sorter and the disks.

3. The initial sorter

The initial sorter is a hardware internal sorter which is continuously fed records to be sorted and outputs sorted sequences continuously to the merger. If n is the capacity of the sorter, then each sorted sequence has n records. (Generation of longer sequences is discussed later.)

In the initial sorter, although whole records are processed, they are not moved after each comparison. Instead, we divide record sorting into key-pointer sorting and record rearranging. The initial sorter is composed of the key-pointer sorting circuit, a RAM, and a controller. Key-pointer sorting is done in the key-pointer sorting circuit, and record rearranging is done according to the output of the key-pointer sorting circuit and is overlapped with the output of records.

Since the key-pointer sorting circuit processes only key-pointer pairs, it is small and simple. Furthermore, as shown later, it has a regular linear array structure and is therefore suitable for VLSI implementation. The RAM can be easily built. The controller only needs to perform very simple operations. Thus the whole initial sorter is easily realizable.

In the remainder of this section, we first discuss the components of the initial sorter, then its timing, and finally the modification needed to generate longer sequences.

The key-pointer sorting circuit

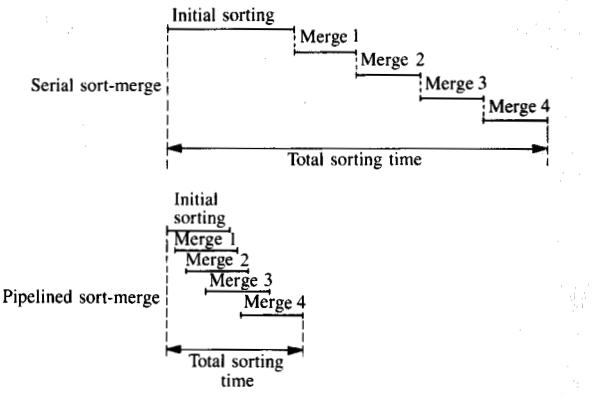
The key-pointer sorting circuit is fed a sequence of n key-pointer pairs serially and outputs them serially according to the order of the key values. In the sorting circuit, the sorting time is completely overlapped with the input/output time. It has complete parallel operation and processes key-pointer pairs in a pipelined fashion. Furthermore, it can overlap the sorting time for two consecutive input sequences.

Its basic algorithm is a pipelined version of the odd-even transposition sort [1, 15]. (The odd-even transposition sort is a parallel version of the bubble sort [1].) It is similar to the up-down sorter of Lee et al. [5], the zero-time sorter of Miranker et al. [6], the weave sorter of Mukhopadhyay [7], and the RESST of Carey et al. [8], whose basic algorithms are all pipelined versions of the odd-even transposition sort. In our sorting circuit, unlike the others, all inner connections are unidirectional. It should also be pointed out that the systolic priority queue proposed in [12] cannot sort different sequences in a pipelined fashion as we can.

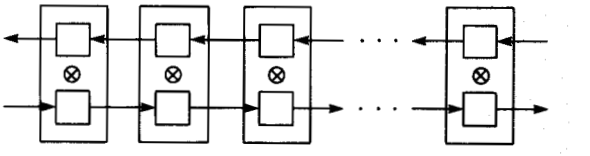
The key-pointer sorting circuit consists of a linear array of $n/2$ cells (we assume that n is even), each of which has two registers and a comparator (**Figure 3**). Each cell can store two key-pointer pairs and can exchange them according to their key values. There are only two unidirectional connections between a cell and its left and its right neighbor cell.

Here n key-pointer pairs are serially input to the lower register of the leftmost cell by n right-shift (input) steps and serially output from the upper register of the leftmost cell by n left-shift (output) steps. One right-shift (left-shift) step of the sorting circuit consists of a right-shift (left-shift) phase followed by a compare-exchange phase. (The removed pair at the rightmost cell goes out of the array in a right-shift step.)

Figure 4 shows an example of the sorting of the key sequence "5, 3, 2, 9, 1, 7" ($n = 6$) in ascending order. (Pointers are not shown.) Initially, each register contains $+\infty$. In the right-shift phase of each input step, key-pointer pairs in lower registers are shifted to the right. In the left-shift phase of each output step, pairs in upper registers are shifted to the left, and $+\infty$ is entered into the upper register of the rightmost cell. In the compare-exchange phase of any step, in each cell, two keys are compared, and the pair with the smaller key value goes to the upper register. At the end of operation, the circuit is filled with $+\infty$'s. Note that at the end of any step, the pair with the smallest key value in the circuit at that time must be in the upper register of the leftmost cell and the second smallest must be in either the lower register of the leftmost cell or the upper register of the second leftmost cell. In general, the pair with the ith smallest key value must be in one of the left i cells.



A comparison of serial sort-merge and the pipelined sort-merge.



Construction of the key-pointer sorting circuit.

The same principle applies to the descending sort. We have only to replace $+\infty$ with $-\infty$ and interchange "smaller" and "larger." In order to distinguish between the ascending and the descending sort, we only need a single control line. In the remainder of this section, we consider the ascending sort only.

The sorting circuit not only processes the key-pointer pairs of a given sequence in a pipelined fashion, but also can sort different sequences in a pipelined way; i.e., while one sorted sequence is being output, a new sequence can be input from the other end of the circuit. In order to distinguish sequences, we attach tag 0 to each key-pointer pair input from the left end and tag 1 to each pair input from the right end. The tags are not compared.

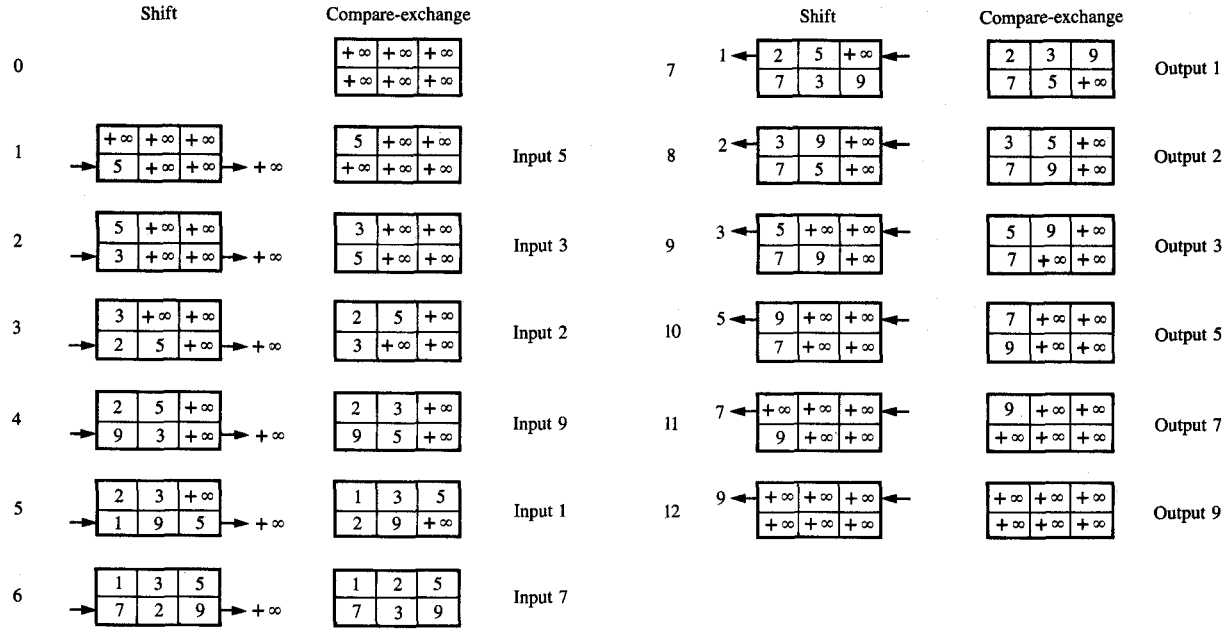


Figure 4

An example of key-pointer sorting using key-pointer sorting circuit. The key sequence is "5, 3, 2, 9, 1, 7." Only keys are shown.

Figure 5 shows ascending sorting for four sequences in a pipelined way. (Pointers are not shown.) Whenever two pairs with different tags meet at a cell, they are exchanged (no comparison is performed). When two pairs input from the right end (1-tagged) meet at a cell, the smaller one goes to the lower register. As we can see in the third sequence of the example, the sorting circuit still works when a sequence contains equal keys. However, if we require that two pairs with equal keys be output in the same order as they are input, i.e., first in first out, we can attach a counter value to each key which indicates its input number in a sequence. Of course, $\lceil \log_2 n \rceil$ extra bits are then needed for each register of the key-pointer sorting circuit. (Counter values are not shown in the figure.) After input is completed, $+\infty$'s are input. $+\infty$'s input from the left (right) end are tagged with 0 (1). As shown in the last sequence of the example, the sorting circuit still works in a pipelined way when the length of the last sequence is smaller than n. The output of the last sequence starts immediately after the output of the second from the last sequence is complete. Thus pairs are continuously output.

Initially, the sorting circuit must be filled with $+\infty$'s. After several sequences have been sorted (see Step 28 of Fig. 5), the sorter is filled with $+\infty$'s with different tags. However, to sort the next batch of sequences, no reinitialization is needed because all 0-tagged (1-tagged) $+\infty$'s must reside in the left (right) part of the sorter, which is sufficient to guarantee that the later sequences are correctly sorted.

In this paper, we do not discuss the detailed logic design of the key-pointer sorting circuit. Suffice it to say that it can be implemented either in a bit-serial or a bit-parallel fashion. Of course, parallel operation is faster but requires more hardware. Since in the initial sorter the key-pointer sorting circuit needs to perform only one step of an operation (shift and compare-exchange) during each transmission of a record and since the record transmission time is much longer than the operation time, serial operation would be enough.

- *The RAM and the controller*

The RAM has a capacity of $(n + 2) \times L$ bytes and can store $n + 2$ records. (The need for the extra 2 will become clear after the timing discussion, which is next.) It has block addresses from 0 to $n + 1$. (In the case that L is a power of 2, a block address consists of some most significant bits of the corresponding byte address.) When a block address is given by the controller, the RAM outputs the record which is stored there and concurrently stores a newly input record there.

The controller controls the RAM by giving it block addresses. It also controls the key-pointer sorting circuit. The controller gets a key-pointer pair from the key-pointer sorting circuit and then gives a block address to the RAM

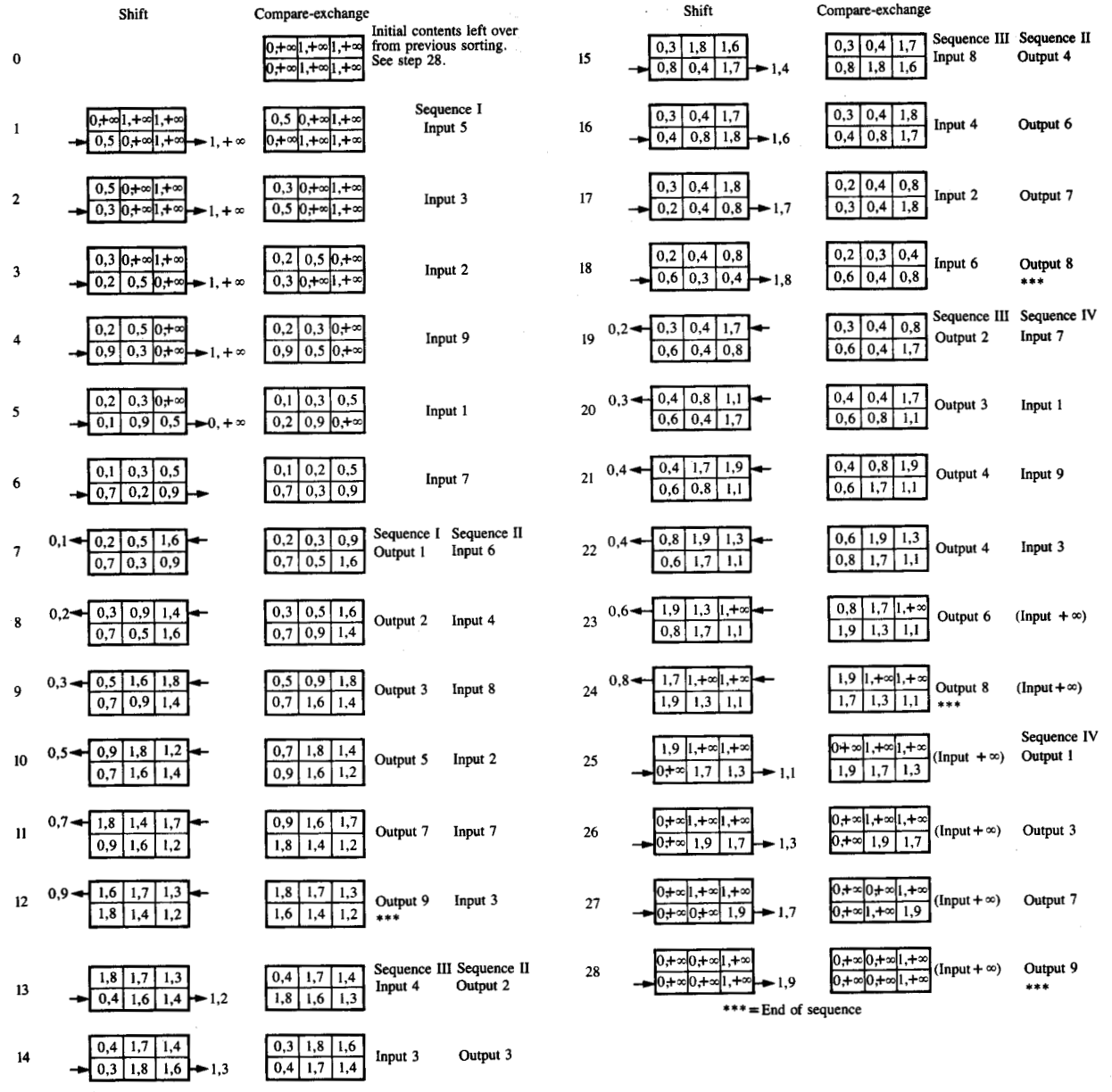


Figure 5

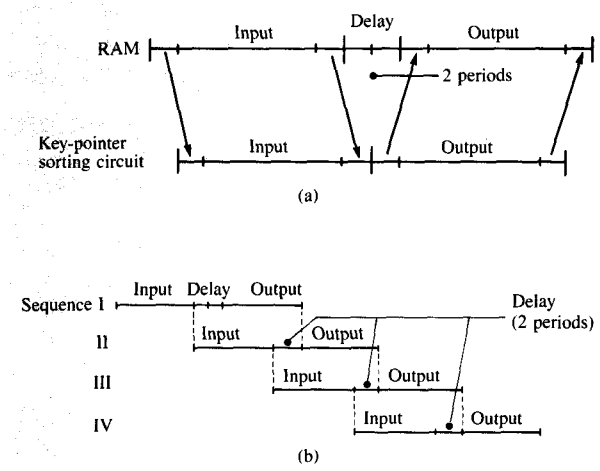
An example of key-pointer sorting in a pipelined way. There are four key sequences: "5, 3, 2, 9, 1, 7"; "6, 4, 8, 2, 7, 3"; "4, 3, 8, 4, 2, 6"; "7, 1, 9, 3." Only tags and keys are shown.

according to the pointer. As the record which is stored in the block indicated by the block address is output, a newly input record is stored there. Then the controller inputs a copy of the key of the newly input record and the pointer to the key-pointer sorting circuit. Of course, to get a copy of the key, the controller has to know the byte address where the key is stored. The pointer indicates the block address where the newly input record is stored.

Three operations—output of a record, input of a record, and one step of the key-pointer sorting circuit—are performed concurrently in one period.

- *Timing*

We call the time period to transmit one record simply a *period*. As mentioned in the previous subsection, three operations are done in one period.



Timing chart. (a) I/O timing of sequence; (b) timing chart of initial sorting for $N = 4n$ records.

An input record is immediately stored into the RAM. In the next period, the controller inputs a copy of the key of the record and the pointer to the key-pointer sorting circuit. The pointer indicates the block address where the record is stored. Upon being fed a key-pointer pair, the key-pointer sorting circuit operates for one step and outputs another key-pointer pair from the other end. (Recall the example of Fig. 5.)

During this period, another record is input and stored in another block of the RAM. In the next period, the record that was pointed to by the pointer which was output from the key-pointer sorting circuit in the last period is output. As the record is output, another record is input and stored. During this period, the key-pointer pair for the record which was input in the last period is input to the key-pointer sorting circuit, and simultaneously another pair is output.

Figure 6 shows a timing chart for initial sorting. Output for a sequence starts with a delay of two periods after its input is completed. Immediately after input for a sequence is completed, input for a new sequence can start. Thus, the RAM must have the capacity to store $n + 2$ records at any time. Although in general N is not divisible by n, initial sorting for N records is always completed in $N + n + 2$ periods.

Figure 7 shows the flow of initial sorting by an example in which $n = 6$ and $N = 22$. For example, the first input record, #0(5), is input to the RAM at time 0, and the key-pointer pair for record #0(5) is input to the key-pointer sorting circuit at time 1. And the key-pointer pair for the first output record, #4(1), is output from the key-pointer sorting circuit at time 7, and record #4(1) is output from the RAM at time 8. The total initial sorting is completed in $22 + 6 + 2 = 30$ periods.

• *Generating longer runs*

Although the initial sorter can handle only n records at a time, it can generate runs of $2 \times n$ records on the average. The principle is the same as that of the replacement-selection sort [1]. Instead of a selection tree, we use the key-pointer sorting circuit.

In the previous method, we always input a key-pointer pair for the newly input record from the other end of the key-pointer sorting circuit. In the current method, we input a key-pointer pair to the sorting circuit from either end according to the following condition: If the key value of the input record is larger than or equal to that of the record to be output next, we input the key-pointer pair for the input record from the same end where the current output is carried out; otherwise we input the pair from the other end. In the former case, the newly input record is added to the currently processed sequence. In the latter case, it belongs to the next sequence.

The timing is as follows. In a certain period, a record is input and stored in the RAM. In the next period, first the key-pointer sorting circuit operates for one step and outputs a key-pointer pair for the record to be output in the next period. In this step, $-\infty$ is input from the other end. Then the key value of the record input in the last period is compared with that of the record to be output in the next period. If the key value of the input record is larger than or equal to that of the record to be output, the key-pointer pair for the input record is input from the same end. Otherwise, it is input from the other end. To input the key-pointer pair from the other end, we have first to remove the previously input $-\infty$. Thus, in this period, the key-pointer sorting circuit operates for two or three steps.

Figure 8 shows an example of initial sorting for the same record sequence as in Fig. 7, using the new method. In the key-pointer sorting circuit, pairs with the same key value are sorted according to their attached counter values (which are not shown in the figure), when their input order must be preserved. However, in such a case, more than $\lceil \log_2 n \rceil$ extra bits are needed for each register of the sorting circuit since the length of the run is at least n. (On the average, it is $2 \times n$. In the best case, it could be the length of the whole input sequence.) If only p extra bits per register are used, the length of a run will be bounded above by 2^p.

• *Further considerations*

Throughout the previous discussions, we assumed that the records and the keys had fixed size, namely, L and l bytes, respectively, and that they could always fit in the initial sorter. However, this may not always be true. Suppose that the size of each block of the RAM and the size of each register for a key in a cell of the key-pointer circuit are L' and l', respectively.

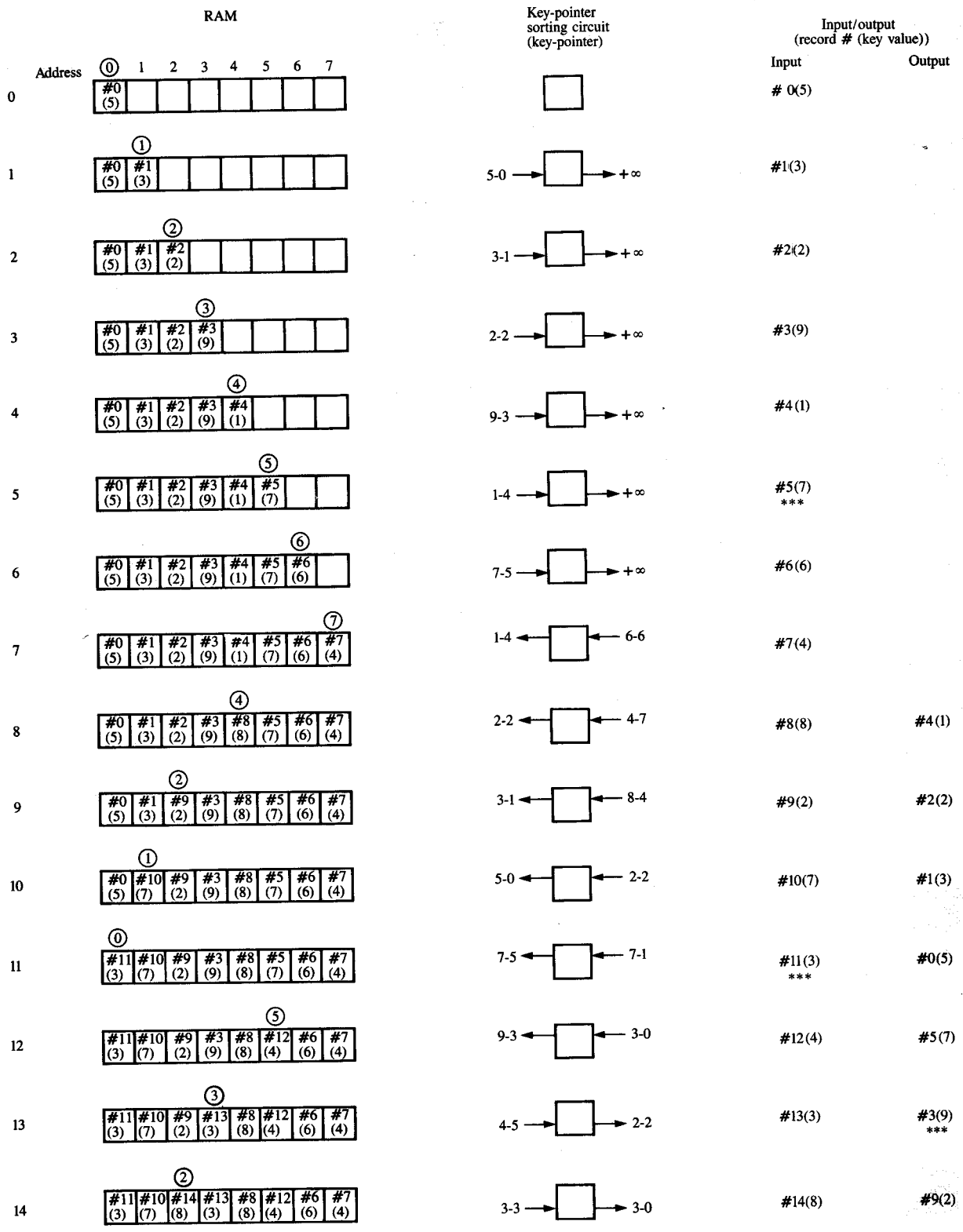


Figure 7

Flow of initial sorting of 22 records with key values of 5, 3, 2, 9, 1, 7, 6, 4, 8, 2, 7, 3, 4, 3, 8, 4, 2, 6, 7, 1, 9, 3.

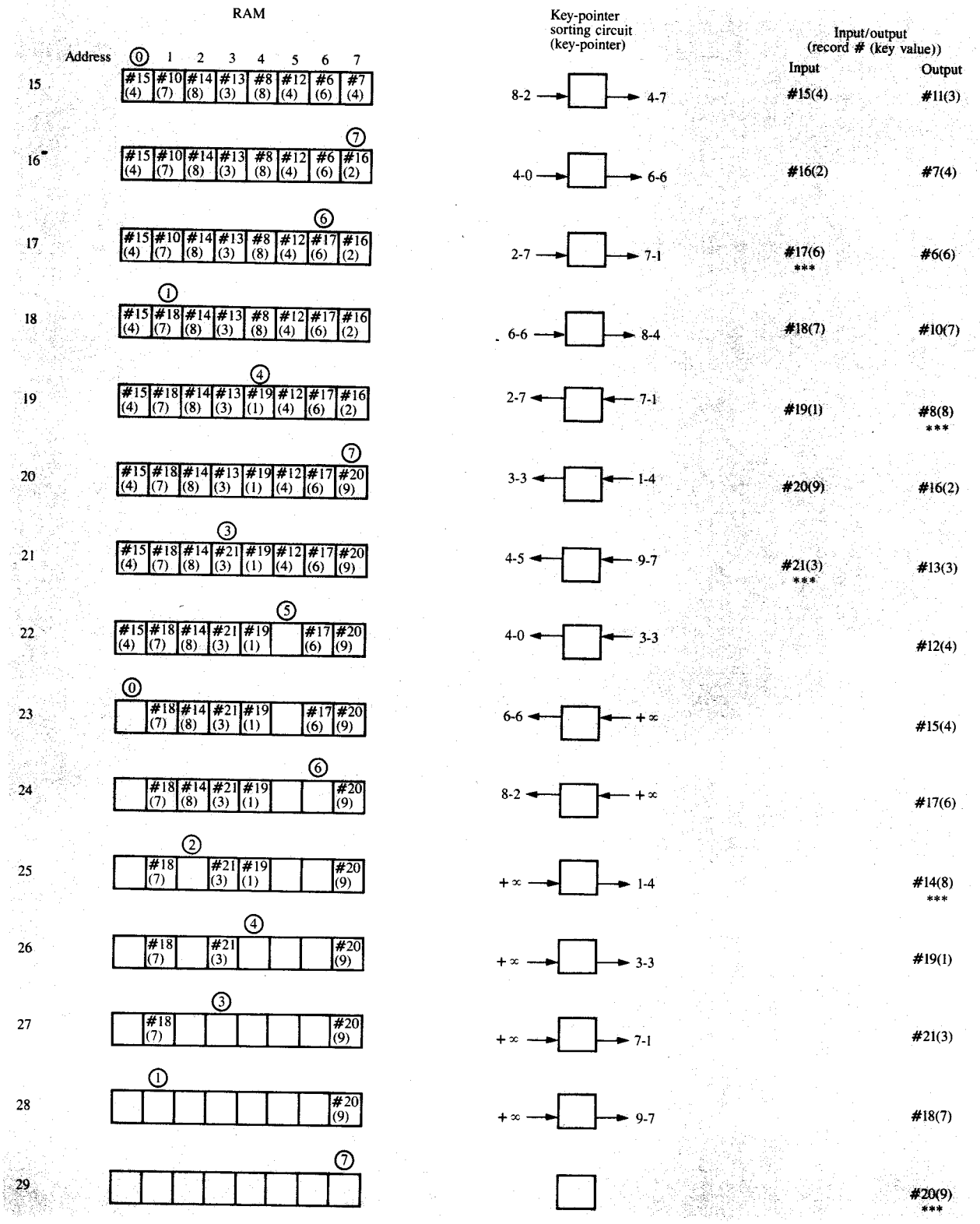


Figure 7 Continued

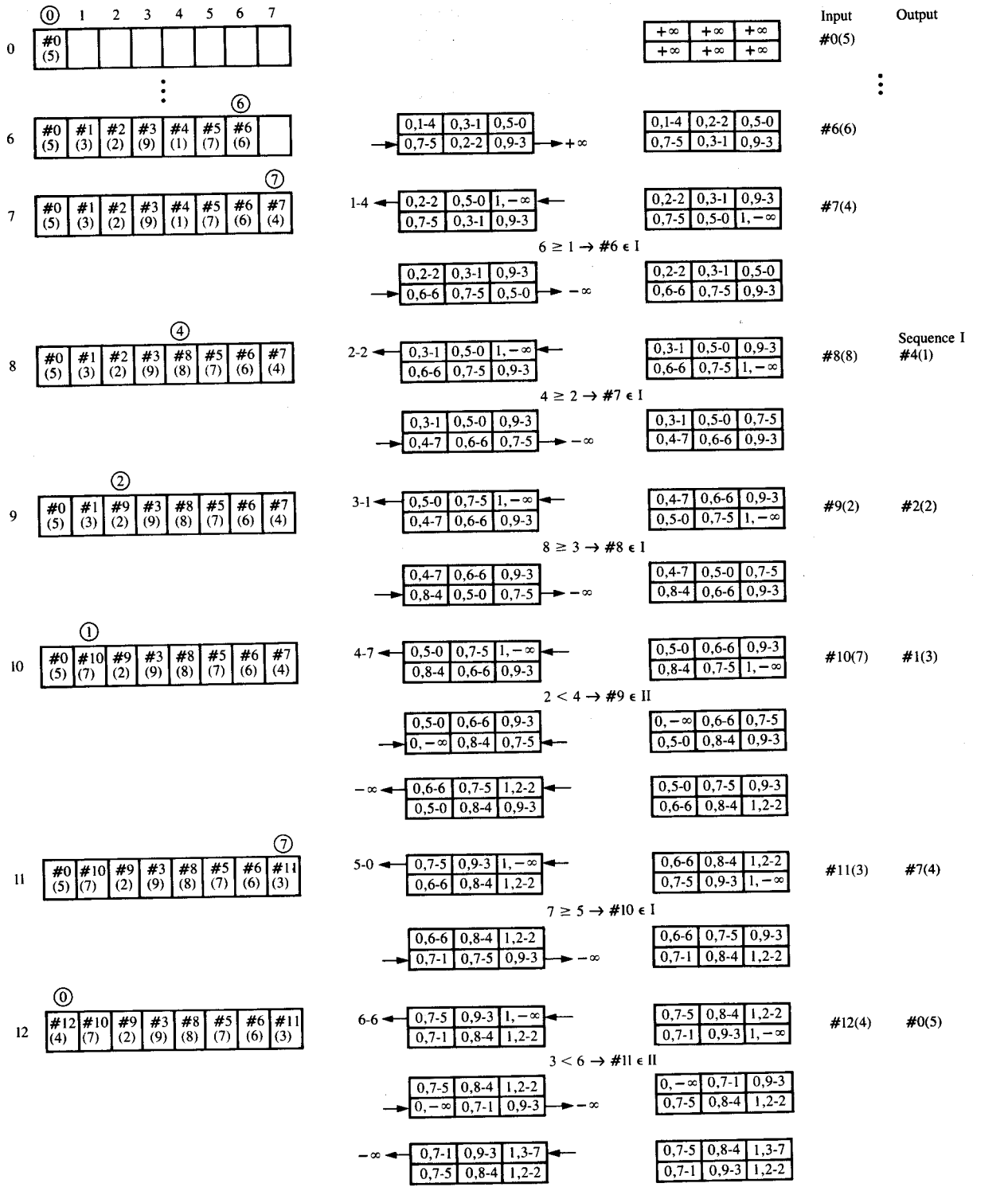


Figure 8

Flow of initial sorting using the method of generating longer runs.

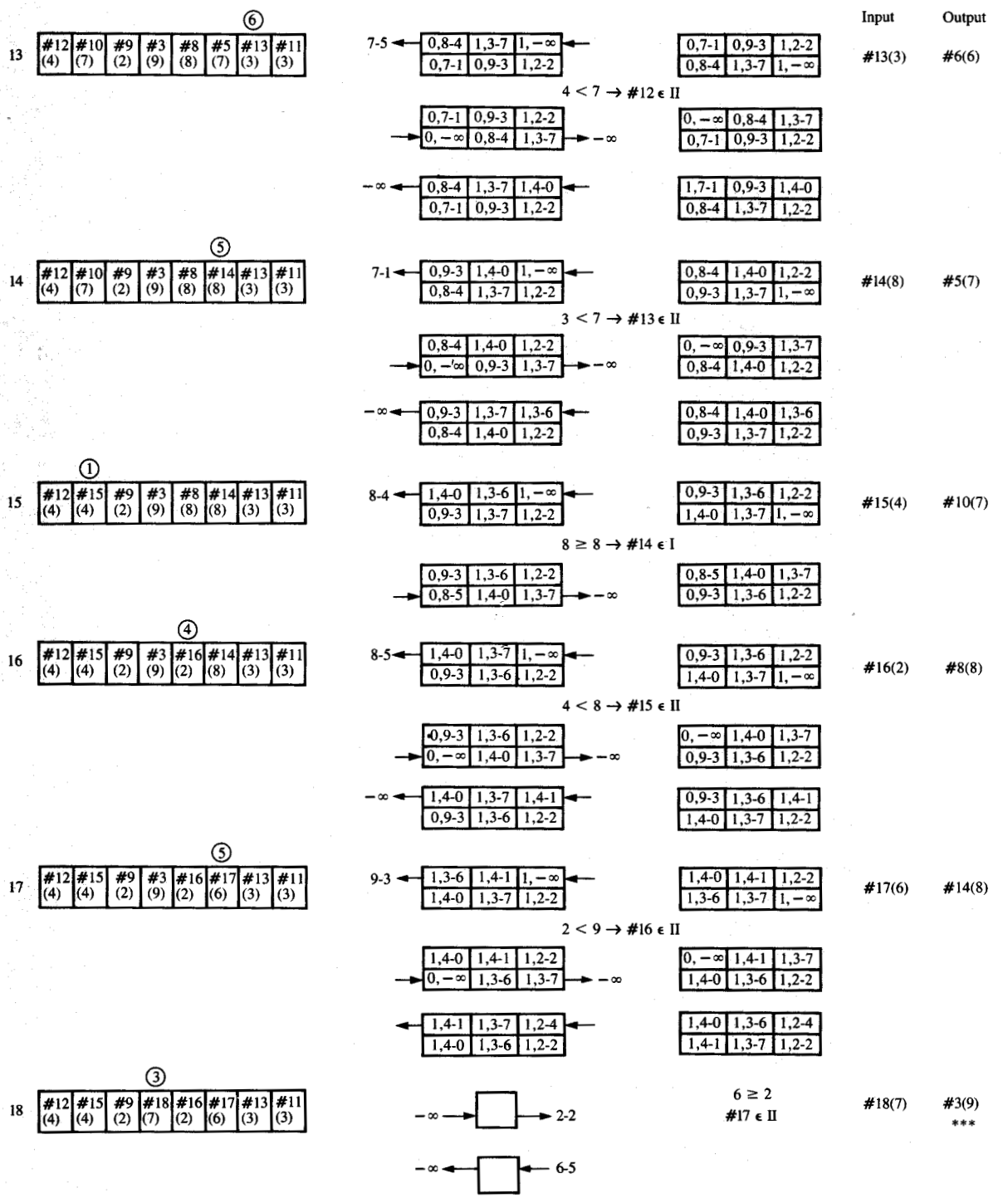


Figure 8 Continued

First, consider the record size. If L is fixed and $L \leq L'$, there is no problem. Even though L is not fixed, as long as max $(L) \leq L'$, it is still all right. However, if L is fixed and $L > L'$ or L is not fixed and max $(L) > L'$, we must use two or more blocks of the RAM for sorting each record. In this case, the initial sorter will have fewer than $n/2$ records at a

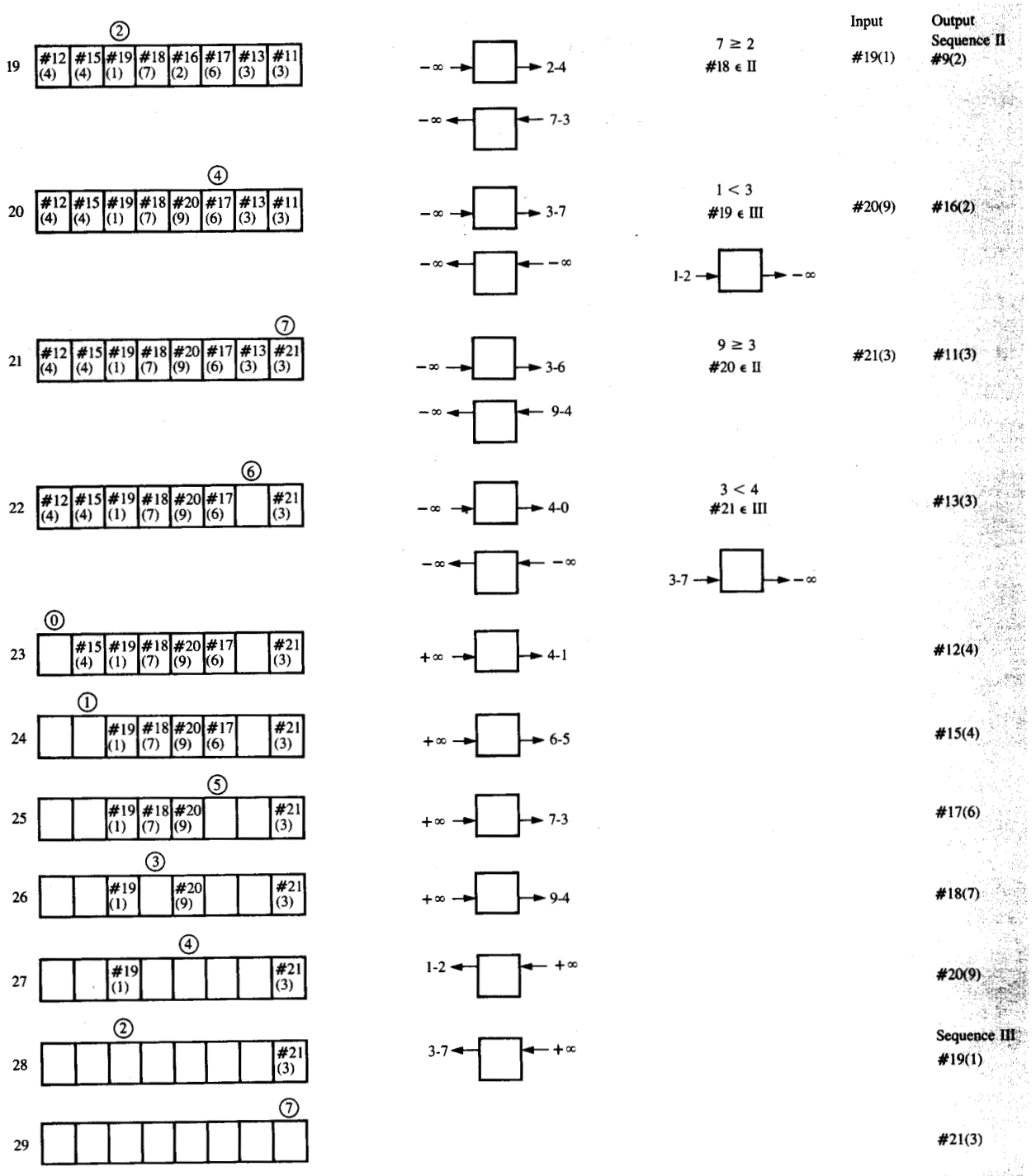


Figure 8 Continued

time, and in any period, more than $n/4$ cells of the key-pointer sorting circuit must be idle.

Next, consider the key size. When l is not fixed, we temporarily let all key lengths be max l by attaching 0's. (If

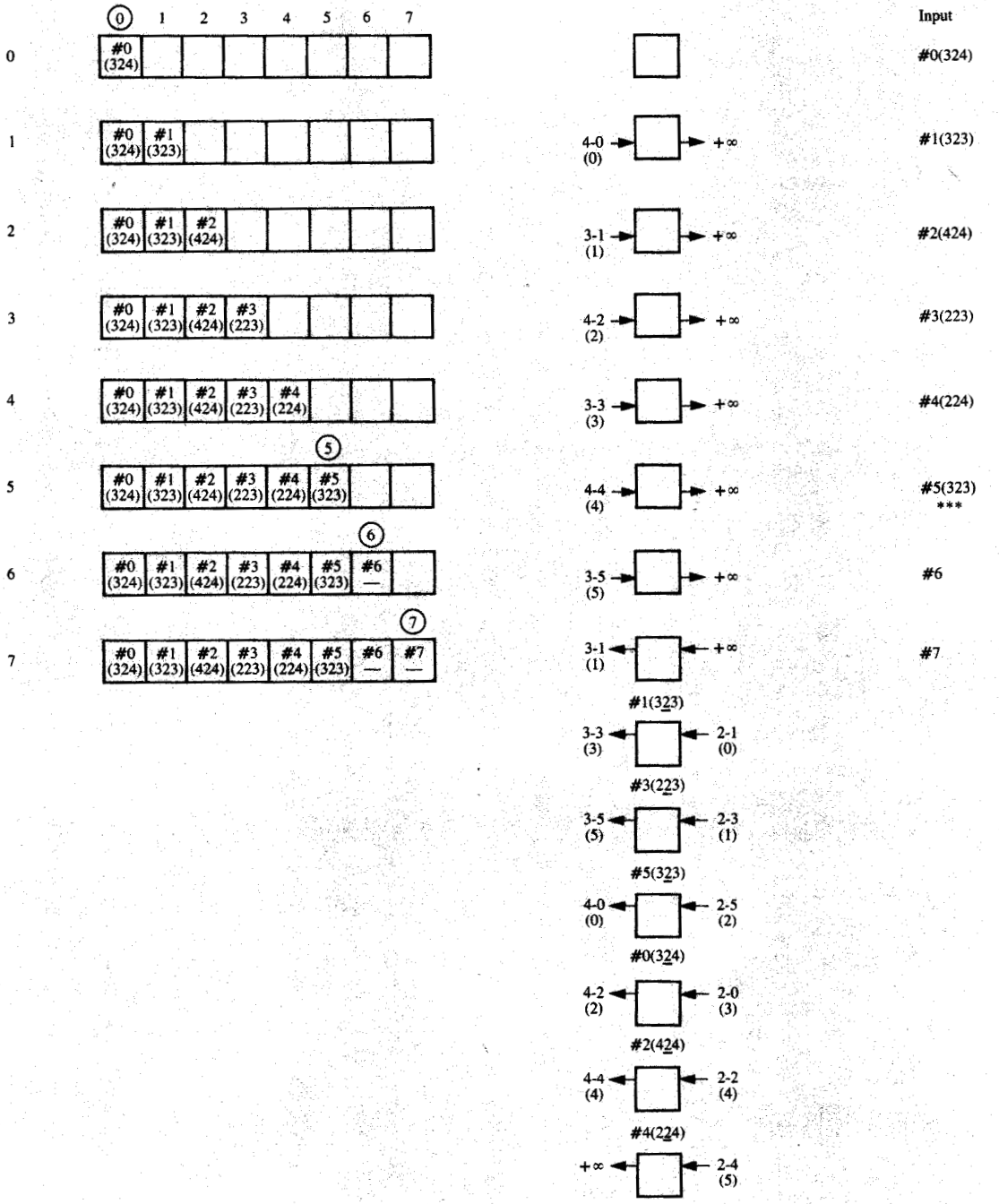


Figure 9

Sorting of records with long keys.

the keys are characters, we attach 0's to their least significant parts. If the keys are numbers, we attach 0's to their most significant parts.) If max $l < l'$, we attach more 0's to each key so that its length becomes l'. If max $l > l'$, we divide each key into several parts so that the size of each part becomes l' (if l is not divisible by l', we attach 0's) and sort key-pointer pairs by a method such as the least-significant-digit-first radix sort [1] using the key-pointer sorting circuit $\lceil l/l' \rceil$ times. Namely, first we sort pairs according to the values of the least significant parts of the keys, next

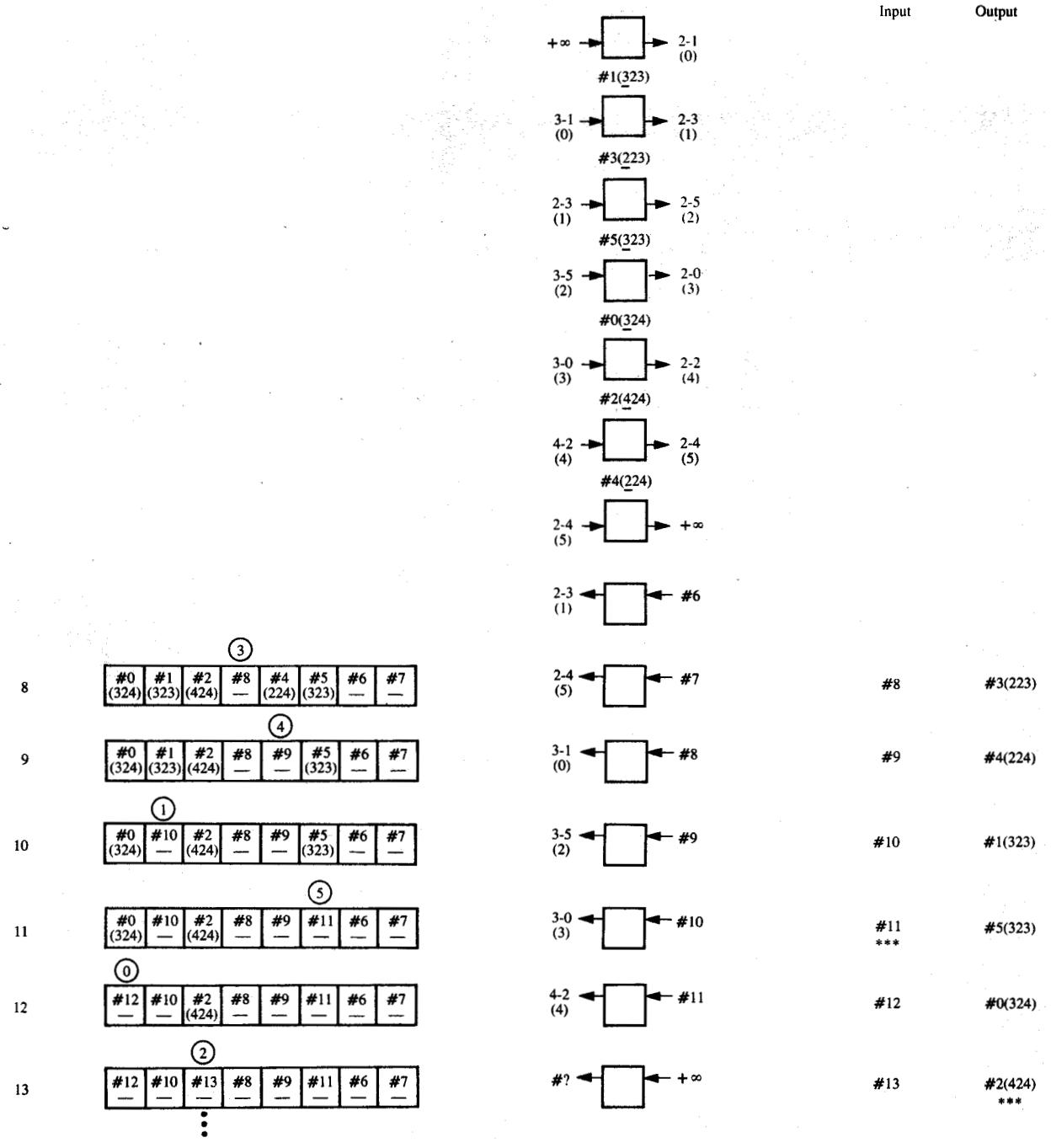


Figure 9 Continued

according to the values of the second least significant parts, and so on, and finally sort them according to the values of the most significant parts.

Figure 9 shows an example of a case when $l = 3l'$. In this method, the input order is important, and counter values must be attached to the parts of the keys in order to distinguish pairs with the same key-part value. Of course, the method delays the initial sorting time. However, since records are transmitted only once, overall sorting is still very efficient. This method clearly applies to multi-key sorting. On the other hand, it is difficult to incorporate the method of generating longer runs mentioned previously.

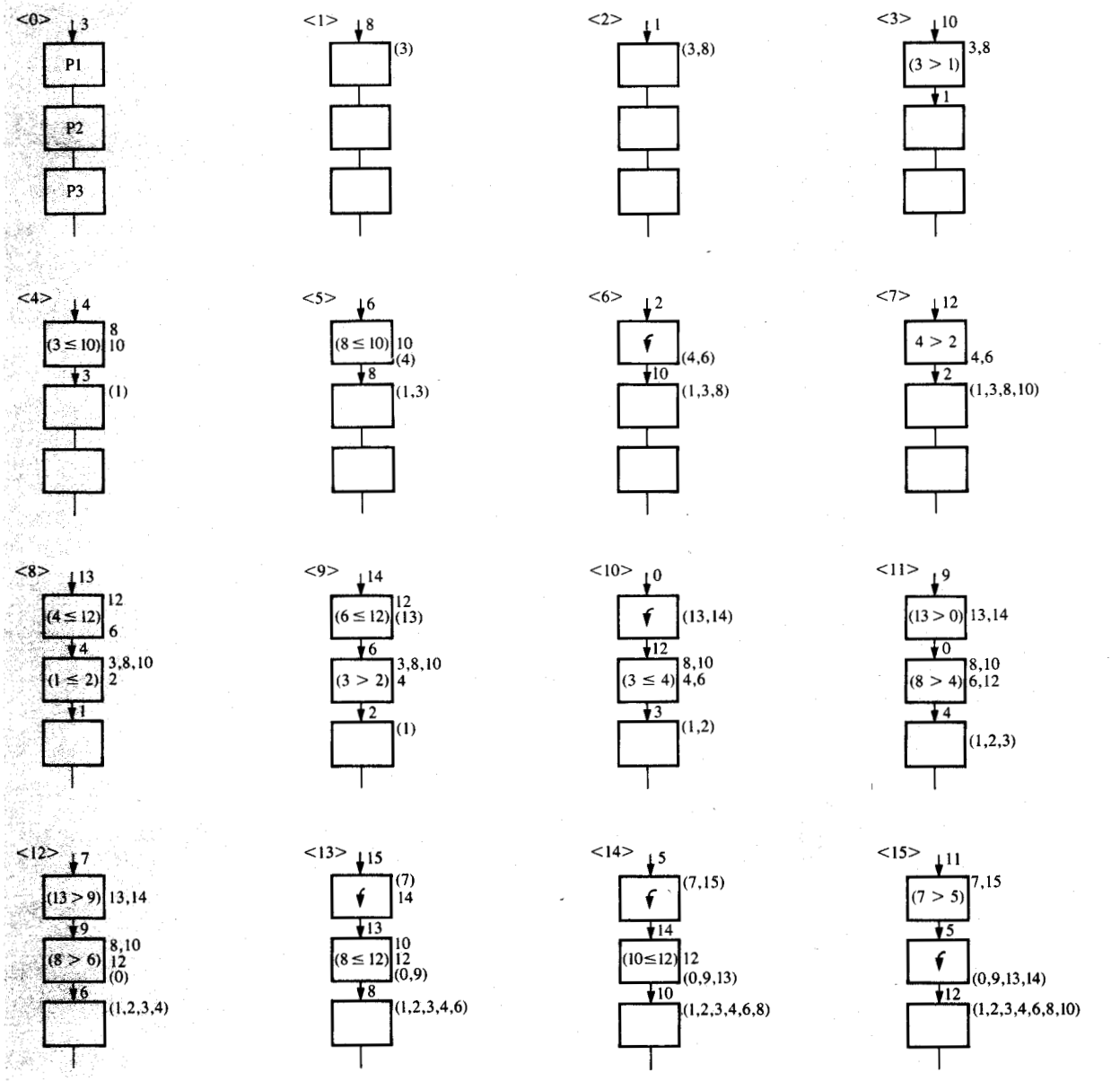


Figure 10

Flow of the ideal pipelined merge (3, 8/1, 10/4, 6/2, 12/13, 14/0, 9/7, 15/5, 11). ( ) indicates waiting queue.

4. The pipelined merger

The pipelined merger is fed initial runs consecutively from the initial sorter, merges them into a single run, and outputs it serially. The pipelined merger is composed of several intelligent disks, each of which has a simple processor and several two-bank buffers.

The basic scheme is a pipelined version of an m-way merge. The serial m-way merge operates in several passes, with each pass generating longer runs by merging every m input runs. For simplicity, we assume that the initial sorter generates runs of length exactly n, as described in Section 3. Merging of longer initial runs is discussed later. The first pass is fed N/n runs of n records each and generates $N/(m \times n)$ runs of $m \times n$ records. The second pass merges these runs into runs of $m^2 \times n$ records. After i passes, the runs have length $m^i \times n$. After $\lceil \log_m (N/n) \rceil$ passes, all N records are in one run. In this scheme, all merging passes are run in an overlapped way, rather than serially.

First, we describe the principle through a two-way merge. **Figure 10** shows an example of an ideal pipelined merge for

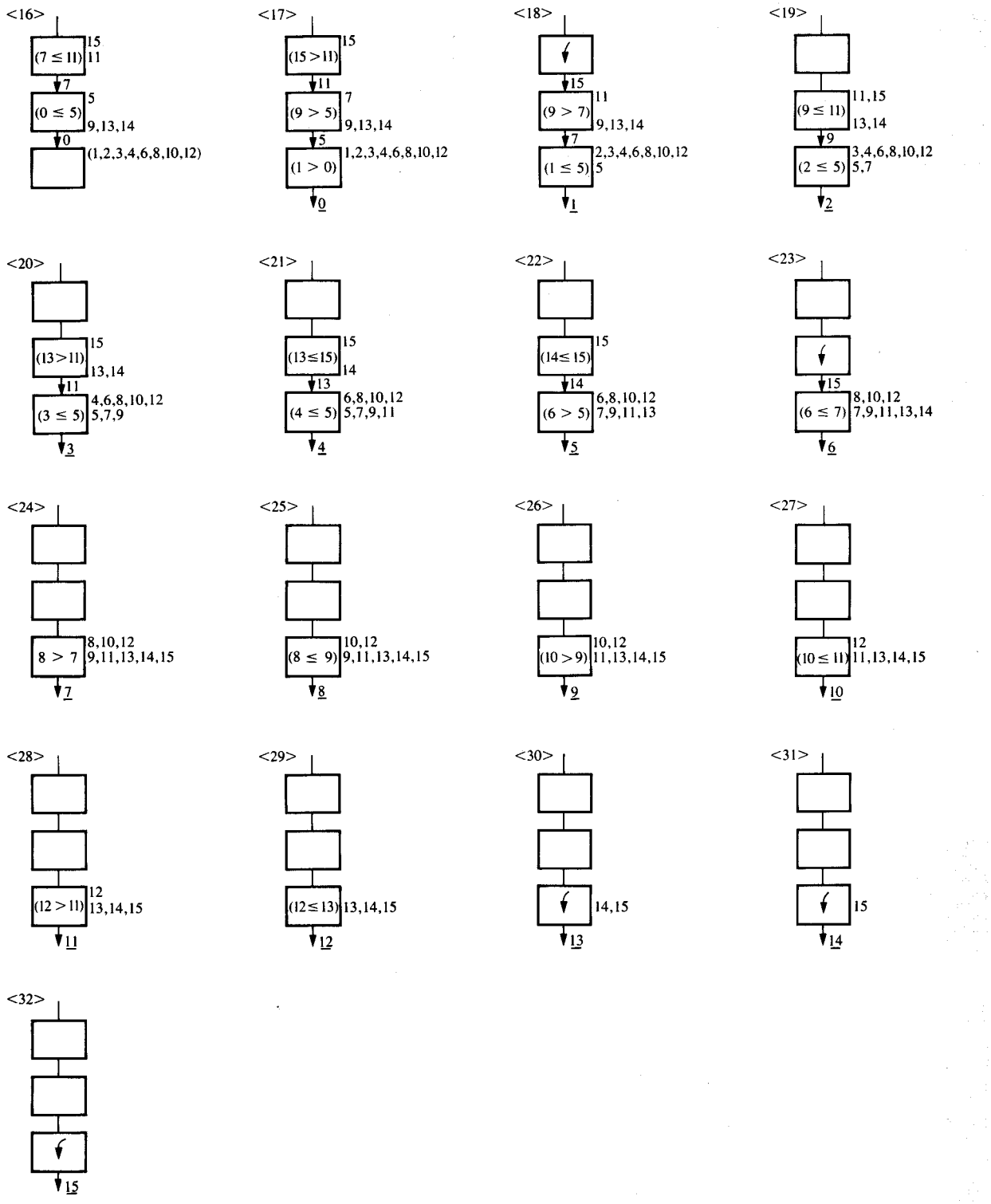


Figure 10 Continued

8 initial runs of 2 records (i.e., $N = 16$ and $n = 2$). $\lceil \log_2 8 \rceil = 3$ processors operate in parallel. Each processor manages three FIFO queues. Two of them are for runs which are currently merged, and the other is for a run to be merged

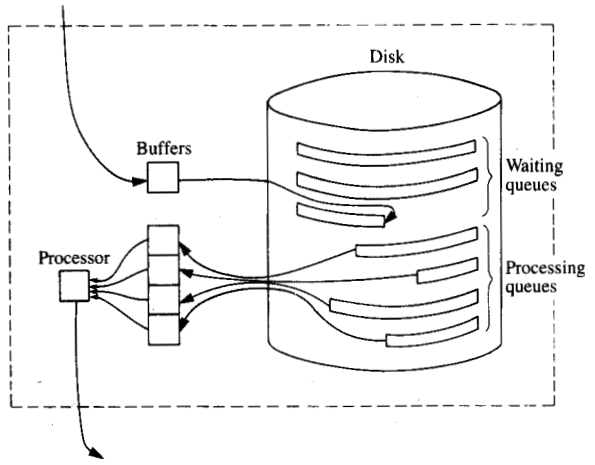


Figure 11

Management of FIFO queues.

next. The maximum length of each FIFO queue of the ith processor is $2^{i-1} \times n$, and the total length of all three queues is also $2^{i-1} \times n$.

When an m-way merge is carried out, $(2m - 1)$ FIFO queues are required for each processor. m of them are for runs which are currently merged, and the others are for runs to be merged next. The maximum length of each queue of the ith processor is $m^{i-1} \times n$, and the total length of all $(2m - 1)$ queues is $(m - 1) \times m^{i-1} \times n$.

In the ideal pipelined merge, all data transmission is done synchronously and all processors are synchronized by their input cycles. However, when we use disks, it is difficult to synchronize data transmission. Furthermore, the data access in the ideal pipelined merge is rather random and not suitable for disks. We must modify the ideal pipelined merge to suit the disk-based implementation. In our proposed pipelined merge scheme, we use the blocking technique mentioned by Todd [9], i.e., we deal with records in blocks rather than individually. In order to adopt the blocking technique, we attach $m + 1$ buffers to each disk. m of them are for the front blocks of the FIFO queues for runs which are currently merged, and the other is for input. (See **Figure 11**.) Note that although there are $m - 1$ waiting queues, since input is done serially, only one buffer is required for the rear block of the FIFO queue which is currently input. For parallel processing and data transmission, we divide each buffer into two banks and use them alternately; i.e., while one of them is used for data transmission to or from the disk, the other can be used for processing. To avoid the latency of the disk, we let the bank size be the track size of the disk. We also let the block size be the bank size and do data transmission to and from disks in blocks. The blocking delays the transmission of records between processors, since the processor has to wait for a block of data before merging starts. Consequently, it slightly slows down the merging.

It is also difficult to synchronize the processors. We just let them run asynchronously, so that a processor may have to wait for a neighbor processor to complete its operation. However, when the operations of the processors are faster than the data transmission, i.e., they are not the bottleneck, then the performance is comparable to that of the synchronous case.

Figure 12 shows an example of the flow during merging at a certain intelligent disk of the pipelined merger. There each 4 input runs of 8 records are merged. Each block contains two records. As shown in stages ⟨16⟩ and ⟨17⟩, the first few blocks of the last run of m runs to be merged can sometimes be directly transmitted from the input buffer to one of the work buffers instead of via the disk. This is due to the availability of the appropriate work buffer at that moment.

To summarize, the pipelined merger is composed of several intelligent disks, each of which has a simple processor and $m + 1$ two-bank buffers. The processor is only required to do an m-way merge and to manage the buffers and the disks. The bank size of buffers should be the track size of the disk (which is 4.8×10^4 bytes for the IBM 3380 disk). m buffers are for the front blocks of the FIFO queues for the runs currently merged, and the other is for the rear block of the queue for the run currently input. Other parts of the queues are maintained in the disks by linking tracks with pointers.

- *Other considerations*

If the pipelined merger is composed of k intelligent disks, it can merge m^k initial runs at a time. Of course, it can merge fewer than m^k initial runs. In such a case, the final run is directly output from the $\lceil \log_m R \rceil$th intelligent disk, where R is the number of initial runs.

Since the processor may operate asynchronously, the pipelined merger can merge initial runs of variable lengths, such as those generated by the method in Section 3. It can also handle records of variable size. In both cases, however, each processor may have a longer waiting time.

The best m for maximum performance depends on the ratio of the speed of the processors and the memories. Of course, a large m makes it possible to merge more initial runs at a time but more buffers are needed for each disk.

In order to reduce merging time, one may employ disks which have separate read and write heads and can concurrently read and write from/to different tracks. In this case, the data transmission speed is twice as fast.

- *Merging more runs*

Although the pipelined merger can merge no more than m^k initial runs at a time, we can merge more than m^k initial runs using the merger repeatedly. We can merge R $(>m^k)$

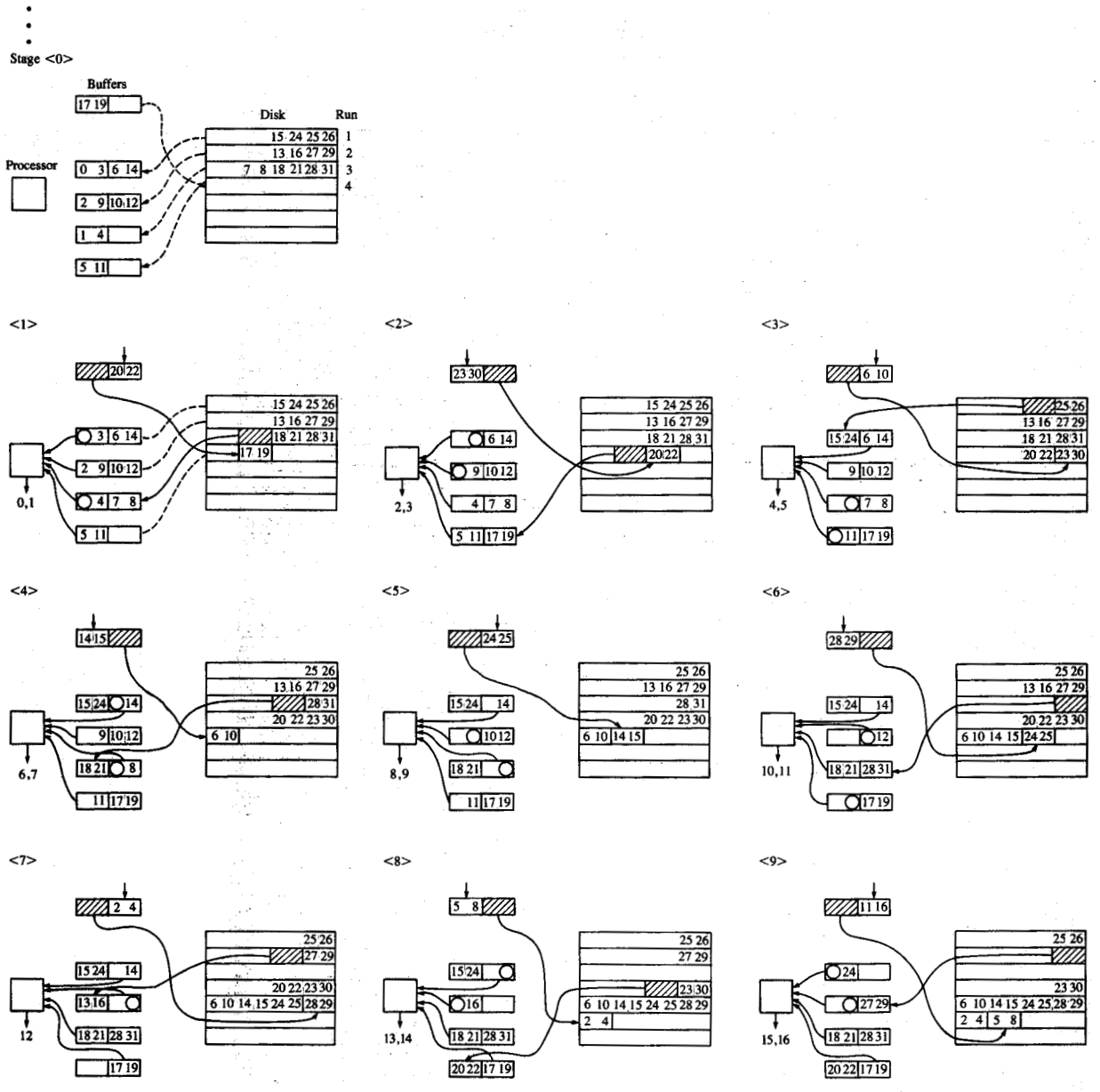


Figure 12

An example of the flow of merging at a particular intelligent disk of the pipelined merger.

runs using the merger $\lceil \log_{m^k} R \rceil$ times. In this case, we can overlap the beginning part of a certain pass with the end part of the previous pass.

Therefore, we can sort any large file using the pipelined sort-merger, i.e., using the initial sorter once and the pipelined merger several times. Of course, the size of the file which can be sorted using the pipelined sort-merger is bounded by the capacity of the intelligent disks. However, it is very large.

5. Conclusion

We have proposed a hardware sort-merge system which can sort large files rapidly. Since the initial sorting pass and the merging passes of the sort-merger are run in an overlapped way rather than serially, the total sorting time is dramatically reduced.

Our hardware sort-merge system consists of the initial sorter and the pipelined merger, both of which can be easily implemented by today's technology. We can sort any large

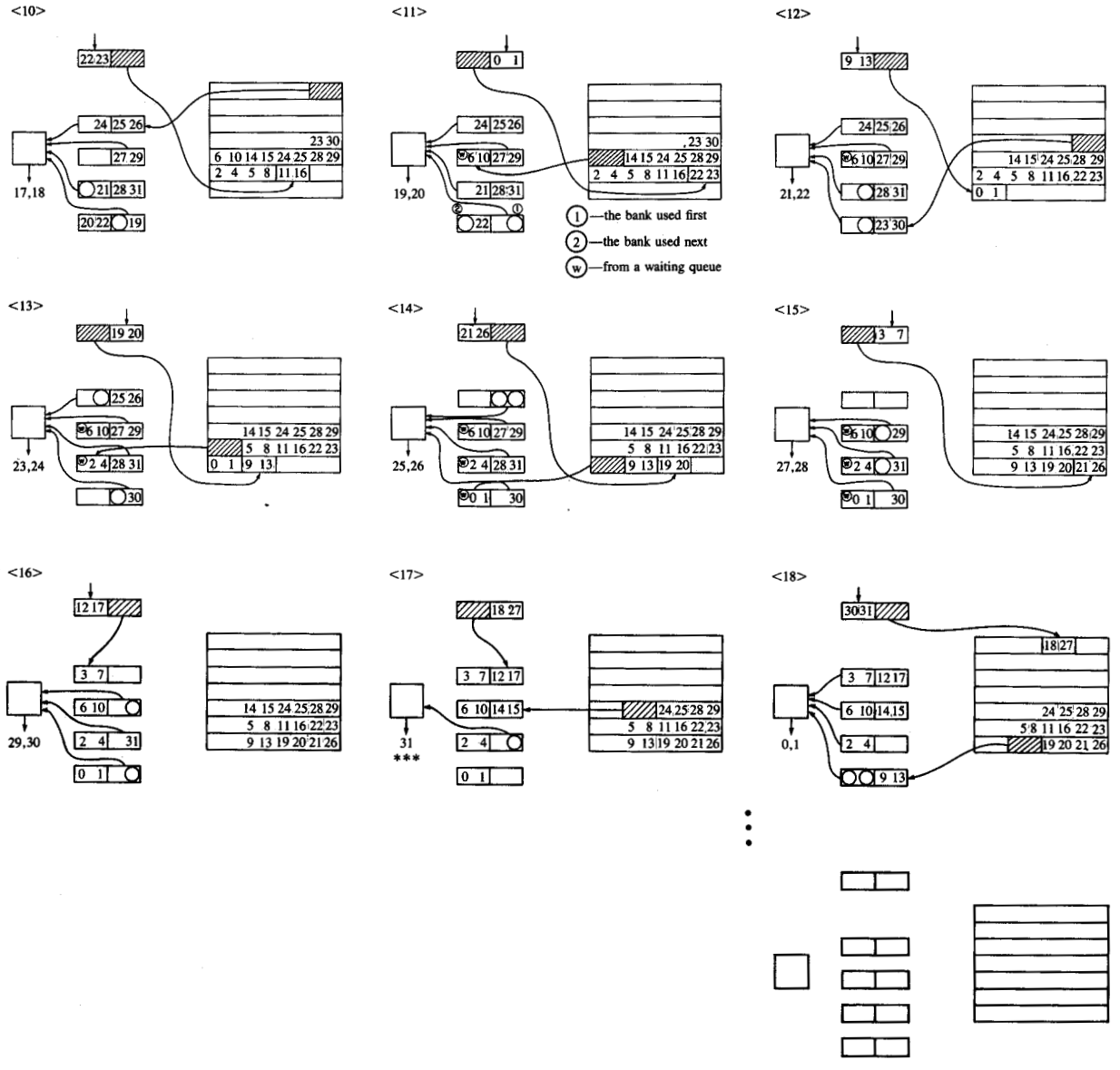


Figure 12 Continued

file using the initial sorter once and the pipelined merger repeatedly. Note also that the RAM of the initial sorter and the intelligent disks of the pipelined merger can be efficiently used for other purposes when they are not doing sort-merge. For example, the RAM can be used as a buffer or a part of the main memory and the disks can be used as work disks. Conversely, the pipelined sort-merger can be easily realized by attaching a few facilities to a conventional computer system.

References

1. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley Publishing Co., Reading, MA, 1973.
2. H. Lorin, *Sorting and Sort Systems*, Addison-Wesley Publishing Co., Reading, MA, 1975.
3. E. E. Lindstrom and J. S. Vitter, "Analysis of BucketSort for Bubble Memory Secondary Storage," *Technical Report ZZ20-6458*, IBM Scientific Center, Palo Alto, CA, October 1982.
4. T. C. Chen, V. Y. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large Files," *Proceedings of the 4th International Conference on Very Large Data Bases*, September 1978, pp. 312–318.
5. D. T. Lee, H. Chang, and C. K. Wong, "An On-Chip Compare/Steer Bubble Sorter," *IEEE Trans. Computers* **C-30**, No. 6, 396–405 (June 1981).
6. G. Miranker, L. Tang, and C. K. Wong, "A 'Zero-Time' VLSI Sorter," *IBM J. Res. Develop.* **27**, No. 2, 140–148 (March 1983).
7. A. Mukhopadhyay, "WEAVESORT—A New Sorting Algorithm for VLSI," *Technical Report TR-53-81*, Computer Science Department, University of Central Florida, Orlando, 1981.

8. M. J. Carey, P. M. Hansen, and C. D. Thompson, "RESST: A VLSI Implementation of a Record-Sorting Stack," *Technical Report UCB/CSD 82/102,* Computer Science Division (EECS), University of California, Berkeley, April 1982.
9. S. Todd, "Algorithm and Hardware for a Merge Sort Using Multiple Processors," *IBM J. Res. Develop.* **22,** No. 5, 509–517 (September 1978).
10. Y. Tanaka, Y. Nozaka, and A. Masuyama, "Pipelined Searching and Sorting Modules as Components of a Data Flow Database Computer," *Proceedings of IFIP '80,* October 1980, pp. 427–432.
11. H. Yasuura, N. Takagi, and S. Yajima, "The Parallel Enumeration Sorting Scheme for VLSI," *IEEE Trans. Computers* **C-31,** No. 12, 1192–1201 (December 1982).
12. L. J. Guibas and F. M. Liang, "Systolic Stacks, Queues, and Counters," *Proceedings of the 1982 Conference on Advanced Research in VLSI,* Massachusetts Institute of Technology, Cambridge, 1982, pp. 155–164.
13. Y. Dohi, "Sorter Using PSC Linear Array," *Proceedings of the 1983 International Symposium on VLSI Technology, Systems, and Applications,* Taipei, Taiwan, 1983, pp. 255–259.
14. S. Even, "Parallelism in Tape-Sorting," *Commun. ACM* **17,** No. 4, 202–204 (April 1974).
15. A. Mukhopadhyay and T. Ichikawa, "An *n*-step Parallel Sorting Machine," *Technical Report 72-03,* Computer Science Department, University of Iowa, Iowa City, 1972.

Received October 5, 1983; revised June 18, 1984

Naofumi Takagi *Kyoto University, Kyoto, Japan.* Mr. Takagi received the B.E. and M.E. degrees in information science from Kyoto University, Kyoto, Japan, in 1981 and 1983, respectively. He is a graduate student at Kyoto University and is working for his Ph.D. degree. During the summer of 1983, he visited the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and performed the work reported in this paper. His current interests include VLSI algorithm design and analysis, logic design, computational complexity, and computer architecture. Mr. Takagi is now doing research on VLSI algorithms for arithmetic operations.

Chak-Kuen Wong *IBM Research Division, P. O. Box 218, Yorktown Heights, New York 10598.* Dr. Wong joined IBM in 1969 as a member of the Computer Sciences Department at the Thomas J. Watson Research Center. His current interests include VLSI design algorithms, abstract and concrete computational complexity theory, optimization problems related to data allocation, magnetic bubble memory structures, the theory of fuzzy sets, and satellite switching/time domain multiple-access systems. Dr. Wong received a B.A. in mathematics from the University of Hong Kong in 1965 and an M.A. and a Ph.D. in mathematics from Columbia University, New York, in 1966 and 1970, respectively. For the academic year 1972 to 1973, he was a Visiting Associate Professor of Computer Science in the Department of Computer Science at the University of Illinois, Urbana. For the academic year 1978 to 1979, he was a Visiting Professor of Computer Science in the Department of Electrical Engineering and Computer Science at Columbia University, New York. Dr. Wong received an IBM Outstanding Invention Award in 1971 for a new family of sorting methods, as well as three IBM Invention Achievement Awards. He holds three U.S. patents and one pending. He is the author of the book *Algorithmic Studies in Mass Storage Systems,* published in 1983 by Computer Science Press. Dr. Wong is a member of the Association for Computing Machinery and a senior member of the Institute of Electrical and Electronics Engineers. He is also an editor of the *IEEE Transactions on Computers,* an Advisory Editor of the international journal *Fuzzy Sets and Systems,* and a Foreign Editor of the Chinese journal *Fuzzy Mathematics.*