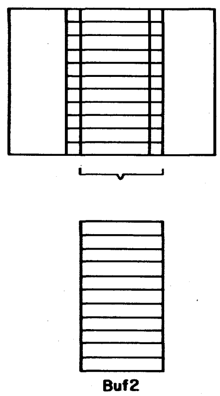

Fig. 8. Moving Buf2 East by updating one-dimensional transforms.

and-merge procedure," in *Proc. Conf. on Computer Graphics, Pattern Recognition and Data Structure,* Beverly Hills, CA, May 1975.

[3] J. K. Hawkins, "Textural properties for pattern recognition," in *Picture Processing and Psychopictorics,* Lipkin and Rosenfeld, eds. New York: Academic, 1970.

[4] R. M. Pickett, "Visual analyses of texture in the detection and recognition of objects," in *Picture Processing and Psychopictorics.* New York: Academic, 1970.

[5] R. Bajcsy, "Computer description of textured surfaces," in *Proc. 3rd Int. Joint Conf. on Artificial Intelligence,* pp. 572–579, Aug. 1973.

[6] D. Marr, "Early processing of visual information," MIT A.I. Memo 340, Dec. 1975.

[7] A. R. Hanson and E. M. Riseman, "Preprocessing cones: a computational structure for scene analysis," COINS Tech. Rep. 74C-7, Univ. of Massachusetts Amherst, Sept. 1974.

[8] R. Bajcsy, "Computer identification of textured surfaces," *Computer Graphics and Image Processing,* vol. 2, pp. 118–130, 1973.

[9] J. H. Halberstein, "Recursive, complex Fourier analysis for real-time applications," *Proc. IEEE,* p. 903, June 1966.

Optimal Sorting Algorithms for Parallel Computers

GÉRARD BAUDET AND DAVID STEVENSON

***Abstract*—The problem of sorting a sequence of n elements on a parallel computer with k processors is considered. The algorithms we present can all be run on a single instruction stream multiple data stream computer. For large n, each achieves an asymptotic speed-up ratio of k with respect to the best sequential algorithm, which is optimal in the number of processors used.**

***Index Terms*—Comparison/exchange, optimal speed-up ratio, parallel algorithms, parallel computers, sorting algorithms.**

INTRODUCTION

From the advent of parallel computers, much attention has been paid to the design of efficient algorithms for these machines. In this paper, we are interested in sorting algorithms for those parallel computers referred to as single instruction stream multiple data (SIMD) stream machines [4]. In these parallel computers all processors execute the same instruction at the same time, but an instruction may be processing different data for different processors. Illiac IV is such a parallel computer [1].

We assume 1) that the machine is composed of k processors $(P_1,P_2,\cdots,P_k)$ and that each processor P_i has its own memory M_i which can be accessed directly by P_i; 2) that a special instruction, the route instruction, allows the processor P_i to read one cell of the memory of an *adjacent* processor (where adjacent depends upon the interconnection strategy) and store its contents in its own memory, M_i; and 3) that each processor has the capability of inhibiting the execution of the current instruction by setting an appropriate indicator. These assumptions are not restrictive. In particular, Illiac IV has these properties.

The central idea of this paper is the following: given any algorithm using only comparison-exchanges for sorting k elements with k processors, there is a corresponding algorithm for sorting rk elements with k processors where every comparison in the first algorithm is replaced by a merge-sorting of two ordered lists of r elements in the second. Since the time for merging two ordered lists is also a lower bound for routing the information involved (i.e., both are linear), it follows that the time complexity of routing will determine the complexity of the sorting algorithms in this family, provided the initialization time for sorting the lists of r elements can be neglected.

Because of the effect of routing, we consider three methods for connecting the processors: a linear order; a two-dimensional array scheme (as is used in the Illiac IV); and a "perfect shuffle." For each interconnection pattern there is an rk-element/k-processor sorting algorithm that is optimal in the sense that for suitable choices of r, the speed-up of the parallel sorting over the optimal sequential sorting algorithm is the number of processors used, which, of course, is the maximal possible increase using parallelism. When the maximum speed-ups for the three interconnection methods are expressed in terms of the number of elements sorted, say n, they are, respectively: $\log n$; $(\log n/\log\log n)^2$; and $2^{\sqrt{\log n}}$. This is also the number of processors used.

Previous studies in parallel sorting (see [10]) have frequently equated the number of elements to be sorted with the number of processors. These studies concluded that the speed-up was logarithmic in the number of processors, or, using a special interconnection scheme, $n/\log n$. No algorithm that had a speed-up that was linear in the number of processors was known. While the equation of the number of processors with the number of elements may be reasonable for sorting networks (i.e., an ensemble of logical circuits to sort n inputs), it is likely that users of parallel computers will want to employ more than one memory location per processor. Indeed, one interpretation of the results we derive is that they are an indication of the optimal ratio of memory size to the number of processors (with respect to the interconnection scheme used).

The family of sorting algorithms we present is in itself very simple but, so far, no optimal sorting algorithm was known for parallel computers. The best known result was an implementation of Quicksort for sorting a sequence of n elements on a parallel computer with $\log n$ processors, leading to an average complexity of $O(n)$, as in our first algorithm, but with a worst case complexity of $O(n^2/\log n)$ ([11]). In contrast, our family of parallel sorting algorithms provides optimal speed-up for problems which are large relative to the number of available processors.

I. A NEIGHBORHOOD SORT FOR A LINEARLY CONNECTED SYSTEM

We begin this section by giving some notation and definitions that are used throughout the paper. An r-sequence, S, is an ordered set of n elements with $n = rk$, such that the first r elements, S_1, are stored in M_1, the next r elements, S_2, are stored in M_2, etc. It will be denoted by $S = S_1;S_2;\cdots;S_k$. An r-sequence $S = S_1$;

Manuscript received May 10, 1975; revised January 9, 1976. This work was supported by the National Science Foundation under Grant GJ-32111 and the Office of Naval Research under Contract N00014-67-A-0314-0010, NR 044-422.

G. Baudet is with Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA 15213.

D. Stevenson was with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. He is now with the Institute for Advanced Computation, Sunnyvale, CA 94086.

0018-9340/78/0100-0084$00.75 © 1978 IEEE

$S_2;\cdots;S_k$ is *partially sorted* if each S_i is a sorted sequence. An r-sequence $S = S_1;S_2;\cdots;S_k$ is *sorted* if the whole sequence $S_1,S_2,\cdots,S_k$ is sorted.

The reader should note that this definition of a sorted sequence coincides with the standard notion of a sorted sequence (i.e., as memory address increases the sequence is monotone) only when each memory M_i contains consecutive memory locations. Another way of addressing memory is to have in memory M_i all addresses which modulo k are congruent to i (as is done for example in the Illiac IV). For this case, a sorted r-sequence will not be sorted with respect to memory location, but a post-processing step which accomplishes this reordering can be performed in time linear in the number of elements sorted. Since this does not affect the nature of our asymptotic results, we shall not treat the matter further.

In this section we will present and analyze a sorting algorithm for a machine for which the processors adjacent to P_i are P_j where $j = i - 1 \bmod k$ or $j = i + 1 \bmod k$. The algorithm takes, as input, an arbitrary r-sequence, and gives as output the corresponding sorted r-sequence. It is illustrated in Fig. 1.

The algorithm is based upon a generalization [6, p. 241] of the *odd–even transposition sort* (see for example [6]) and will be described in Section I-A. In Section I-B we will give an implementation of this algorithm for the machine we consider. An analysis of the algorithm will be presented in Section I-C. Taking into account both the computational cost, measured as the number of comparisons executed, and the routing cost, the cost for transferring the data between different memories, it will be shown that the algorithm takes time $(n \log n)/k + O(n)$. Therefore, when $k < \log n$ the asymptotic speed-up ratio is k and this is optimal in the number of processors used.

A. The Algorithm

The odd–even transposition sort can be considered as an algorithm for sorting k elements using k processors in k steps of parallel "comparison exchanges." The algorithm is the following. Let $a_1,a_2,\cdots,a_k$ be the sequence to be sorted. In the first step, for $i = 1,3,\cdots,2\lfloor k/2\rfloor - 1$, processor P_i compares elements a_i and a_{i+1} and if $a_i > a_{i+1}$ the two elements are exchanged. In the second step, the same comparison exchanges are executed for $i = 2,4,\cdots,2\lfloor(k-1)/2\rfloor$. Steps 3,5, $\cdots$ are the repetitions of Step 1. And Steps 4,6, $\cdots$ are the repetitions of Step 2. A proof that this algorithm leads to a completely sorted sequence after at most k steps is given in [5] and is based on the observation that the distance between the position of an element in the sequence after p steps and its final position is bounded by $k - p$.

Now we give a generalization of this algorithm to partially sorted r-sequences. In the first step, for $i = 1,3,\cdots,2\lfloor k/2\rfloor - 1$, processor P_i merges the two subsequences S_i and S_{i+1} and then assigns to S_i the first half of the resulting merged sequence (i.e., the r smallest elements) and assigns to S_{i+1} the second half (i.e., the r largest elements). For the second step, the same operations are executed but for $i = 2,4,\cdots,2\lfloor(k-1)/2\rfloor$. Again Steps 3,5, $\cdots$, are repetitions of Step 1, and Steps 4,6, $\cdots$, are repetitions of Step 2. This is illustrated in Fig. 2.

A proof that the algorithm leads to a sorted r-sequence in at most k steps can be found in [6] (merging network theorem). Another proof based only on the induction principle was given by Habermann [5] for the odd–even transposition sort (neighbor sort) and can be adapted to the generalization to partially sorted r-sequences.

B. Implementation of the Algorithm

Assume we know how to merge two sorted sequences A_i and B_i of equal lengths using processor P_i, and that it can be done in parallel for all processors (that is, the control structure of the algorithm must depend upon data-dependent masks rather than upon data-dependent branches). In particular, the merge sorting algorithm (see for example [6, p. 159]) is an ideal method for completing the first phase, that is, for initially sorting the S_i. The second phase, sorting the partially sorted r-sequence using the k steps of parallel "merge splitting" as described above, can also be done easily on an SIMD machine, as can be seen in the program given below.

(a)	(b)
$P1 \leftrightarrow M1$: 43 63 54	$P1 \leftrightarrow M1$: 17 25 28
$P2 \leftrightarrow M2$: 28 79 72	$P2 \leftrightarrow M2$: 32 43 47
$P3 \leftrightarrow M3$: 32 47 84	$P3 \leftrightarrow M3$: 54 63 66
$P4 \leftrightarrow M4$: 66 25 17	$P4 \leftrightarrow M4$: 72 79 84

Fig. 1. Effects of the algorithm on the 3-sequence 43,63,54;28,79,72;32,47,84;66,25,17. (a) Input. (b) Output.

The program is written in an Algol-like language using an additional feature for describing the parallelism. Namely, when a statement is followed by a mask, only those processors P_i for which bit i of the mask is set to 1 will execute the instruction, the other processors being inhibited. When no specification follows a statement every processor is to execute the corresponding instruction.

The data structure involved in the program includes in memory M_i, for $i = 1,2,\cdots,k$, an array $S_i[1{:}r]$ containing the current subsequence S_i of the r-sequence to be sorted, and three arrays $A_i[1{:}r+1]$, $B_i[1{:}r+1]$, and $C_i[1{:}2r]$ used as working areas. For simplicity we assume that k, the number of processors, is even. We use two masks: in EVEN the bit i is set to 1 only for $i = 2,4,\cdots,k-2$, and in ODD only for $i = 1,3,\cdots,k-1$. These masks are used only for the implementation of the second phase.

The procedure MERGE, when executed by processor P_i with parameter p, merges the two subsequences $A_i[1{:}p]$ and $B_i[1{:}p]$ (that is the first p elements in both arrays A_i and B_i) and stores the resulting merged sequence in $C_i[1{:}2p]$.

```
MERGE(p) =
begin
A_i[p + 1] ← w; B_i[p + 1]
  ← w;
a ← 1; b ← 1;
for c ← 1 step 1 until 2p do
  if A_i[a] > B_i[b]
  then begin
    C_i[c] ← B_i[b]; b ← b +
      1;
    end
  else begin
    C_i[c] ← A_i[a]; a ← a +
      1;
    end;
end
```

Comment: w is added as a sentinel at the end of both sequences A_i and B_i so that no supplementary test is required for checking if any sequence becomes exhausted. (w is supposed to be greater than any element that A_i and B_i can ever contain).

Comment: We use implicitly two masks which are set according to the result of the comparison between $A[a]$ and $B[b]$. (This is easily done if there exists an instruction which allows a processor to set its inhibition indicator according to its condition code.)

```
Phase 1:
for p ← 1 step p until r − 1
      do
        for j ← 1 step 2p
      until r do
          begin
            A_i[1:p] ← S_i[j:j
      + p − 1]
            B_i[1:p] ← S_i[j +
      p:j + 2p − 1];
            MERGE(p);
            S_i[j:j + 2p − 1]
        ← C_i[1:2p];
          end
```

Comment: The phase 1 as it is programmed here is able to partially sort only those r-sequences for which r is a power of 2. This restriction can be easily removed if we add artificial elements.

Comment: The phase 1 is completed at this point and $S_1[1{:}r],S_2[1{:}r],\cdots,S_k[1{:}r]$ contain the partially sorted r-sequence.

	Step 1:		Step 2:		Step 3:		Step 4:	
$S1$: 43 54 63	$\xrightarrow{P1}$	28 43 54		28 43 54	$\xrightarrow{P1}$	17 25 28		17 25 28
$S2$: 28 72 79		63 72 79	$\xrightarrow{P2}$	17 25 32		32 43 54	$\xrightarrow{P2}$	32 43 47
$S3$: 32 47 84	$\xrightarrow{P3}$	17 25 32		63 72 79	$\xrightarrow{P3}$	47 63 66		54 63 66
$S4$: 17 25 66		47 66 84		47 66 84		72 79 84		72 79 84

Fig. 2. Four steps of parallel "merging-splittings" for sorting the partially sorted 3-sequence 43,54,63;28,72,79;32,47,84; 17,25,66.

```
Phase 2:
for j ← 1 step 2 until k − 1 do
   begin
   A_i[1:r] ← S_i[1:r];               mask ODD;    Comment:
   B_i[1:r] ← S_{i+1}[1:r];           mask ODD;    This statement,
   MERGE(r);                          mask ODD;    e.g., implicitely
   S_i[1:r] ← C_i[1:r];               mask ODD;    uses the route
   S_{i+1}[1:r] ← C_i[r + 1:2r];      mask ODD;    instruction.
   A_i[1:r] ← S_i[1:r];               mask EVEN;
   B_i[1:r] ← S_{i+1}[1:r];           mask EVEN;
   MERGE(r);                          mask EVEN;
   S_i[1:r] ← C_i[1:r];               mask EVEN;
   S_{i+1}[1:r] ← C_i[r + 1:2r];      mask EVEN;
   end
```

Comment: After those k steps of parallel "merging-splittings," $S_1[1:r], S_2[1:r], \cdots, S_k[1:r]$ contain the sorted r-sequence, and our goal is achieved.

C. *Analysis of the Algorithm*

We first evaluate the routing time. Routing is introduced only by instructions of the form $B_i[1:r] \leftarrow S_{i+1}[1:r]$ and $S_{i+1}[1:r] \leftarrow C_i[r + 1:2r]$ in the second phase. Such instructions are executed k times each and, in both cases, r route instructions are required, therefore the total number of route instructions is

$$R = 2rk = 2n.$$

For the computational cost, we count only the number of comparison instructions executed but, clearly, the total cost excluding routing has the same order of magnitude. Comparison instructions occur only in the MERGE procedure and $2p$ comparisons are required then the procedure is called with parameter p. Looking at Phase 1, the first for-loop is executed $\log r$ times (when r is a power of 2) for $p = 1,2,4, \cdots, r/2$ and the inner loop is executed $r/2p$ times for each of the values of p. Each of the inner loops requires $2p$ comparisons, therefore the total number of those instructions is

$$C_1 = [2p(r/2p)] \log r = r \log r = (n/k) \log (n/k).$$

For the second phase, the number of comparisons is simply

$$C_2 = 2rk = 2n.$$

Therefore the total number of comparisons is

$$C = C_1 + C_2 = (n \log n)/k - (n \log k)/k + 2n.$$

Assume, now, that the time for executing a comparison instruction is a unit of time. Let λ be the time for executing a route instruction involving an adjacent processor. Then, when we consider only comparisons and routing, the total execution time of the algorithm is

$$T = C + \lambda R = (n \log n)/k - (n \log k)/k + 2n + 2\lambda n$$

or simply

$$T = (n \log n)/k + O(n).$$

But we know that the minimum number of comparisons required for sorting a sequence of n elements on a sequential computer is asymptotically $n \log n$. Therefore, when k is smaller than $\log n$ (in order of magnitude), the asymptotic speed-up ratio of our algorithm over the optimal sequential algorithm is k, which is optimal. In particular, when $k = \log n$, the ratio of this parallel algorithm to the optimal sequential algorithm is of order $\log n$, the number of processors.

On the other hand, when k is greater than $\log n$, the total execution time required for our algorithm is asymptotically linear in n; but this is the best we can achieve with the constraints of the machine we consider here, as can be seen from the following remark. In case all elements whose final destination is memory M_k are initially in the same memory M_i, the total routing must be at least $r(k - i)$ (or $\min\{r(k - i), ri\}$ if we consider processors P_1 and P_k adjacent). If we choose, for example, $i = k/2$ then the total routing must be at least $n/2$ and consequently the total execution time must also be at least linear in n. Therefore, in this case our algorithm is within a constant factor of the optimum time.

II. Other Optimal Algorithms

It seems unlikely that systems with many processors will have the routing time between P_i and P_j linear in $|i - j|$. Indeed, this is not the case with Illiac IV, which has sixty-four processing elements. In this section we examine two other schemes for connecting processors which are in some sense more efficient in routing information among the processors. The first, a two-dimensional array, is the technique used in Illiac IV: the second, based on the "perfect shuffle," is a design that has been frequently advocated for interconnecting networks of arithmetic processors. We begin with some general observations about our family of parallel sorting algorithms before considering the particular impact of the various interconnection structures.

Our linear-connected parallel algorithm is based on the simple idea of replacing the sequence of parallel "comparison-exchanges" of the initial neighbor sort by a corresponding sequence of parallel "merging-splittings." This idea can be easily generalized. For any algorithm (based exclusively on comparison exchanges) for sorting a sequence of k elements on a parallel computer with k processors, there corresponds another parallel sorting algorithm for sorting a sequence of $n = rk$ elements on a parallel computer with comparable interconnections. In addition, if the initial algorithm requires $C(k)$ parallel comparisons and a total routing $R(k)$, the running time of the new algorithm is, assuming that comparison takes one unit of time and that λ is an average for the time required for a unit step of routing:

$$T(n) = n \frac{\log n}{k} + n \frac{2C(k) - \log k}{k} + \lambda n \frac{R(k)}{k}.$$

And, provided that $C(k)$ and $R(k)$ are less than $\log n$ (in order of magnitude), the asymptotic speed-up ratio is k which is optimal.

For example, this scheme could be applied to Batcher's sorting algorithm [2]. The resulting algorithm would lead to an improvement over the neighbor algorithm for the number of comparisons ($C(k)$ is reduced from k to $(\log k)^2$) but, for a parallel computer with a linear interconnection scheme, the total routing becomes $2n \log k + O(n)$ as opposed to $2n$ for the neighbor sort algorithm and thus Batcher's sorting algorithm is not optimal. Hence, the computational efficiencies of this algorithm are lost in the routing inefficiencies. This illustrates the often overlooked aspect of parallel computing: that it is the environment in which an algorithm will be executed that determines whether the computational aspects will be realized.

If we take into account other specific interconnection schemes for a parallel computer, it is possible to decrease the time for

TABLE I
Comparison of the Total Routing on a Square Array for the Neighbor Sort and Batcher's Algorithms

k =	4	16	64	256	1024
$RN(k)$ =	2	2	2	2	2
$RB(k)$ =	1.50	2.25	2.03	1.47	0.99

sorting n elements and hence to increase the speed-up of parallel sorting over sequential sorting. In Illiac IV, the interconnection scheme is such that each processor can access in one route step not only the memory of an adjacent processor but also the memory of a processor at a distance of $\sqrt{k}$. In this case simple algebra shows (see for example [7]) that the total routing time for Batcher's algorithm is given by

$$R(k) = 4\sqrt{k} \log k - 8\sqrt{k} - \log k + 8.$$

This shows that asymptotically (for large values of k) an adaptation of Batcher's sorting algorithm on a two-dimensional array computer will give a better result both for the total number of parallel comparisons (for the second-order term only) and for the total routing than will the neighbor sort algorithm on a linearly connected processor. However, Table I shows that for small values of k (the number of processors), the total routing of both algorithms are very close. In this table we have reported the quantities $R(k)/k$ for both algorithms, namely $RN(k) = (2k)/k = 2$ in the case of the neighbor sort and $RB(k) = (4\sqrt{k} \log k - 8\sqrt{k} - \log k + 8)/k$ for the adaptation of Batcher's sorting algorithm to exploit the two-dimensional connections.

With this two-dimensional interconnection scheme, the adaptation of Batcher's sorting algorithm achieves an asymptotic speed-up ratio of k as long as $C(k)$ and $R(k)$ are less than $\log n$ (in order of magnitude), that is, as long as $\sqrt{k} \log k < \log n$. This holds for k less than $(\log n/\log \log n)^2$. Thus, using this processor configuration, the maximum possible speed-up over the optimal sequential sorting algorithm is greater than $\log n$ (the case for the linear interconnection scheme). This improvement in speed-up is due, of course, from using proportionately more processors, and is still optimal in the number of processors used.

And in fact for the actual configuration of Illiac IV, with $k = 64$ and λ approximately $3/4$, we find that for $r = 1024$, Batcher's sort is almost twice as fast as the neighbor sort, counting only comparisons and routing.

Finally, we briefly consider the effect of using a "perfect shuffle" to connect the processors [9]. In this scheme the processors "adjacent" to P_i and P_j where $j = 2i \bmod k$ and $2j = i \bmod k$. For k a power of 2, a sort of k elements can be done with $C(k) = R(k) = (\log k)^2$ [9]. As above, the corresponding merge-splitting algorithm has linear speed-up (in the number of processors) as long as k is less than $2^{\sqrt{\log n}}$.

Acknowledgment

We wish to thank J. F. Traub and D. E. Heller, CMU, for comments on the manuscript, and H. T. Kung, CMU, for suggesting to the first author that sorting on a parallel computer would be an interesting problem to study, and for many comments and discussions.

References

[1] G. H. Barnes *et al.*, "The ILLIAC IV Computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.
[2] K. E. Batcher, "Sorting networks and their applications," in *Spring Joint Comput. Conf., AFIPS Proc.*, vol. 32, 1968, pp. 307–314.
[3] W. J. Bouknight *et al.*, "The ILLIAC IV system," *Proc. IEEE*, vol. 60, pp. 369–388, Apr. 1972.
[4] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948–960, Sept. 1972.
[5] A. N. Habermann, "Parallel neighbor sort," Carnegie-Mellon University, Comput. Sci. Dep. Rep., Aug. 1972.
[6] D. E. Knuth, *The Art of Computer Programming—Sorting and Searching*, vol. 3. Reading, MA: Addison-Wesley, 1973.
[7] S. E. Orcutt, "Computer organization and algorithms for very high speed computations," Ph.D. dissertation, Comput. Sci. Dep., Stanford University, Sept. 1974.
[8] D. L. Slotnick *et al.*, "The Soloman computer—A preliminary report," in *Proc. 1962 Workshop on Computer Organization.* Washington DC: Spartan, 1963, p. 66.
[9] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.
[10] ——, "Problems of parallel computations," in *Proceedings of the Symposium on Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub ed., New York: Academic Press, 1973.
[11] ——, private communication, 1975.

On an Ordering of Walsh Functions

H. BUTIN

***Abstract*—In addition to the automorphism between the sequency and the Gray-code-of-sequency ordering for Walsh functions there is also one for Kacsmarz ordering. This is shown by using a matrix definition suggested by K. W. Henderson some years ago [4].**

***Index Terms*—Sequency, Walsh–Kacsmarz functions, Walsh matrix.**

I. Introduction

The widely used definition of Walsh functions in terms of binary expansion of an ordering parameter s and an argument t, which play symmetrical roles, is often limited to the nonnegative values of s and t.

Handling negative arguments can be done, as noticed in [1], but some little difficulties arise when s or t are dyadic rational numbers [2]. This can be avoided, and the Walsh functions can be defined for every s and t real, in terms of their binary expansion, via a particular form, sometimes called the Walsh–Kacsmarz form [3]. Therefore it will be denoted here $\mathrm{wal}_k(s,t)$. So defined, the Walsh functions are sequency-ordered, and naturally separate into cal and sal functions, for their sequency equals the absolute value of the ordering parameter. Moreover, there is an easy way to pass from $\mathrm{wal}_k(s,t)$ to other well-known forms of the Walsh functions.

II. The Walsh–Kacsmarz Form of Walsh Functions

Let a and d be two integers. Define $s \in [-2^{a-1},2^{a-1})$, i.e., $-2^{a-1} \leqslant s < 2^{a-1}$, and $t \in [-2^{d-1},2^{d-1})$ by their binary expansion, in 2's complement notation, as follows:

$$\sum_{-\infty}^{a-1} 2^m \cdot s_m = \begin{cases} s & \text{if } s_{a-1} = 0 \\ 2^a - s & \text{if } s_{a-1} = 1 \end{cases} \tag{1}$$

and likewise for t, changing s_m into t_m, and a into d. Here, only finite binary representations are being considered, if s or t are dyadic rational.

Manuscript received June 17, 1976; revised January 5, 1977.
The author is with Thomson-CSF, Gennevilliers, France.