Bit-Sliced VLSI Algorithms for Search and Sort

Yuzuru Tanaka

Department of Electrical Engineering
Hokkaido University
Sapporo, 060 JAPAN

Abstract

For the high speed processing of databases, it is fundamental to introduce various VLSI architectures to the processing of basic functions. Especially, sort and batch search requires high speed modules. The VLSI algorithms of them must make use of the time necessary for the transfer of a large amount of data to and from the modules. These modules should be nonprogrammable in order to avoid serious overheads. However, they should be able to extend their capacity and wordlength by the connection of them.
This paper solves the problem of how to extend the wordlength of search and sort hardware modules. It proposes bit-sliced architectures of an interval search engine and a two-way-merge sorter. The slicing of these engines does not cause excessive overheads. The decrease of the slice length decreases the hardware complexity, and increases the flexibility of the modules. Therefore, it increases the feasibility of the VLSI implementation of these hardware modules.

1. Introduction

The arrival of high speed relational database machines with large capacity is prompted by the users application requirements for very large databases and a wide range of database activities. The use of moving head disk units is innevitable to provide a sufficiently large storage space. While the provision of an associative search mechnism to disk units might enable some database processing to be directly performed on disk tracks, data transfer to and from the storage modules is innevitable to cope with more sophisticated database processing. Besides, the parallel processing of database operations requires the separation of storage modules and processing modules. The arrival of high performance database machines requires innovations in the follwing technologies:

1. VLSI architectures for high-speed processing of basic functions.

2. Hierarchical memory organization for database processing.

3. Control mechanism for cooperative coordination of massively parallel processes.

This paper concerns the first item. Since the transfer of a large amount of data is innevitable in database processing, VLSI architectures for basic functions must be able to make much use of the transfer time for their processing.

The relational model of databases provides a set of database operations as listed below:

set operations :	union, intersection, set difference.
relational operations :	projection, selection, restriction, join, division.
aggregate operations :	count, sum, average, maximum, minimum.
others :	sort.

Suppose that no relations is sorted apriori with respect to some attribute, nor provided with an auxiliary files such as inverted files or link files. Suppose also that the size of each relation is proportional to a single parameter n. Then the time complexity of each operation above is either O(n) or O(n*logn). They are classified as follows:

O(n) :	selection, restriction, count, sum, average, maximum, minimum.
O(n*logn) :	union, intersection, set difference, projection, join, division, sort.

Operations in the first class can be executed by

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

The table is stored in an engine preceding to the batch search processing, in which m search keys are sent to the engine one after another as a stream ($k_0, k_1, \ldots, k_{m-1}$). For each input key k, the ISEE outputs an interval (A^L, A^R) of table addresses. A Search Engine (SEE) that was previously proposed can output only A^L. The addresses A^L and A^R are the minimum addresses that satisfy respectively the following two conditions:

$$T(A^L) \geq k \quad \text{and} \quad T(A^R) > k.$$

These two table addresses are respectively called the left address and the right address of the search key k in the table T. Obviously, their difference $A^R - A^L$ is equal to the number of keywords in T that are equal to k.

In an ISEE, a search table is represented by a binary tree called a left-sided binary tree. This is similar to an SEE. The height of trees is determined by the capacity of the hardware implementation. The i+1st keyword T(i) is stored in the node labelled i in Fig. 2.1. An ISEE with L levels can store a table with no more than 2^L-1 keywords. The number of nodes at each level of a tree that are loaded with keywords is referred to by the load factor of this level. This is denoted by LOAD(j), where j denotes the level number.

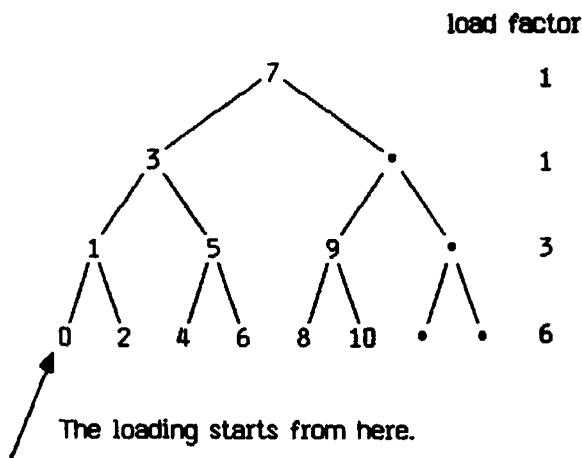

Fig. 2.1. A half-loaded left-sided binary tree.

An ISEE has a hardware configuration as shown in Fig 2.2. It has multiple levels. Each level has dedicated logic circuits and a dedicated memory bank. The memory bank at the i-th level has 2^{i-1} words. The nodes at the i-th level of the left-sided binary tree are stored from left to right in the memory bank at the i-th level. The i-th node from the leftmost one at each level is stored in the address i-1, which is referred to by the intra-level address of this node.

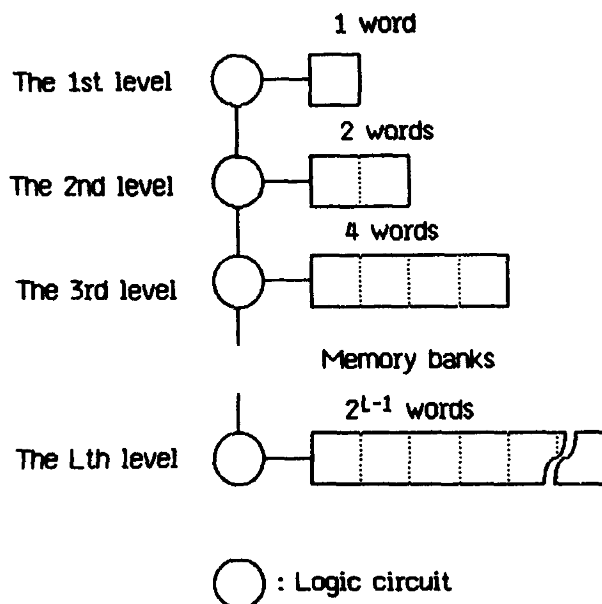

Fig. 2.2. The hardware configuration of an ISEE.

The interval search of a left-sided binary tree for a search key k starts from the root of the tree, procedes downward, and outputs (A^L, A^R) of k from the bottom level. The logic circuits at each level j get a search key k and a pair of addresses ($w^L(j)$, $w^R(j)$) from the upper level and outputs k and ($w^L(j+1)$, $w^R(j+1)$) to the next level j+1. The values of $w^L(0)$ and $w^R(0)$ are assumed to be always zero. Let $T^j(i)$ denote the keyword stored in the node at the intra-level address i in the level j. The addresses $w^L(j+1)$ and $w^R(j+1)$ are calculated as follows:

$$\begin{aligned} & w^L(j+1) \\ &= \text{if } k \leq T^j(w^L(j)) \text{ or } w^L(j) > \text{LOAD}(j)-1 \\ &\qquad \text{then } 2*w^L(j) \\ &\qquad \text{else } 2*w^L(j)+1, \\ & w^R(j+1) \\ &= \text{if } k < T^j(w^R(j)) \text{ or } w^R(j) > \text{LOAD}(j)-1 \\ &\qquad \text{then } 2*w^R(j) \\ &\qquad \text{else } 2*w^R(j)+1. \end{aligned}$$

The search result (A^L, A^R) for a search key k is obtained as the output addresses from the bottom level, i.e., the following properties always hold.

Property 2.1.
Suppose that an ISEE has L levels. The addresses $w^L(L)$ and $w^R(L)$ that are output from the bottom level as the search result for a key k are respectively equal to the left address and the right address of k in this search table, i.e.,

$A^L = w^L(L)$ and $A^R = w^R(L)$.

The intra-level addresses $w^L(j)$ and $w^R(j)$ in each level j are called respectively the left boundary and the right boundary of the search for k at level j. Figure 2.3 shows an example interval-search process.

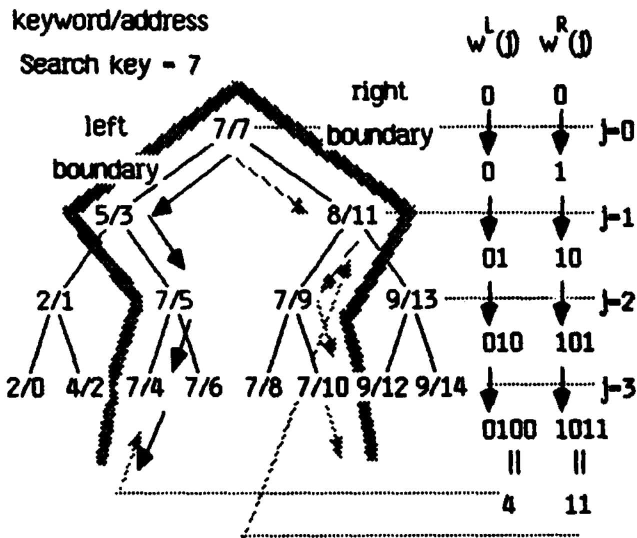

Fig. 2.3. An example of an interval search.

Although the above description of operations has concerned a single key search, an ISEE actually searches a stream of keywords in a pipeline fashion. Every time after having output a pair $(w^L(j+1), w^R(j+1))$ for a search key k_i, each level j gets a next search key k_{i+1} and its boundaries $(w^L(j), w^R(j))$ from the level j-1, and begins the comparison to output $(w^L(j+1), w^R(j+1))$ for k_{i+1} to the level j+1. At every instance, an ISEE treats as many keys as its levels. The flow of keys forms a stream and it flows downward through all the levels of the ISEE.

2.2. Bit-sliced interval search engine

Now, let us consider how to realize a bit-sliced architecture of an ISEE. First we shall modify an ISEE to have two bits of output signals at each level. The values of these signals are defined as follows:

$CO^L(j)$

= if $k \le T^j(w^L(j))$ or $w^L(j) > LOAD(j)-1$

then 0

else 1,

$CO^R(j)$

= if $k < T^j(w^R(j))$ or $w^R(j) > LOAD(j)-1$

then 0

else 1.

Such a modified ISEE is referred to by an MISEE.

A bit-sliced ISEE with n-bit wordlength is defined as a module that is connected to an MISEE with m-bit wordlength in a way as shown in Fig. 2.4 to form a new MISEE with (n+m)-bit wordlength. Each of its levels has two output lines $CO^L(j)$, $CO^R(j)$, and two input lines $CI^L(j)$, $CI^R(j)$. These two input lines $CI^L(j)$ and $CI^R(j)$ are respectively connected to $CO^L(j)$ and $CO^R(j)$ of the corresponding level of the preceding m-bit MISEE.

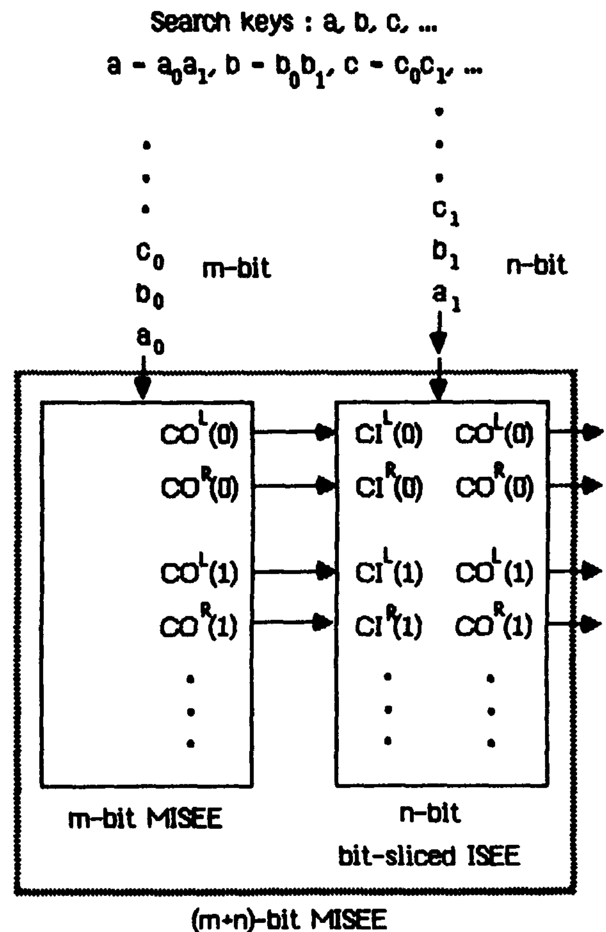

Fig. 2.4. Connections of an m-bit MISEE and an n-bit bit-sliced ISEE.

Suppose that all keywords and all search keys have a common leftmost m bits. Then the (m+n)-bit MISEE in Fig. 2.4 may be considered same as an n-bit MISEE since the leftmost m bits of each value are insignificant. This n-bit part is actually treated by the n-bit bit-sliced ISEE, which, in this case, works as an n-bit MISEE. In this case, the input signals $CI^L(j)$ and $CI^R(j)$ at

each level are always set equal to zero and one respectively by the preceding m-bit MISEE. Therefore, an n-bit bit-sliced ISEE with each $CI^L(j)$ and $CI^R(j)$ respectively set to zero and one should work in a same way as an n-bit MISEE.

A bit-sliced ISEE is designed to have a similar hardware configuration as an ISEE. The node where T(i) is stored in an ISEE stores the rightmost n bits of T(i), which is denoted by $T^R(i)$. The bit-sliced ISEE in Fig. 2.4 can not see the boundaries $w^L(j)$ and $w^R(j)$ calculated at each level of its lefthand MISEE. However, we first assume they are visible. They are denoted by $W^L(j)$ and $W^R(j)$ for the distinction from those of the bit-sliced ISEE. Since the rightmost n bits of keywords and search keys are used to refine the search result obtained by the leftmost m-bit processing, the following relations must hold at each level j:

$$W^L(j) \le w^L(j) \le w^R(j) \le W^R(j).$$

These relations also hold at the next level. However, the addresses at the level j+1 is related to those at the upper level j as follows:

$$W^L(j+1) = 2*W^L(j)+CI^L(j),$$

$$W^R(j+1) = 2*W^R(j)+CI^R(j),$$

$$w^L(j+1) = 2*w^L(j)+CO^L(j),$$

$$w^R(j+1) = 2*w^R(j)+CO^R(j),$$

where CI and CO signals of the bit-sliced ISEE are used. Therefore, the following relations must hold:

$$2*W^L(j)+CI^L(j) \le 2*w^L(j)+CO^L(j)$$

$$\le 2*w^R(j)+CO^R(j) \le 2*W^R(j)+CI^L(j).$$

Considerations on these relations gives the logic to determine $CO^L(j)$ and $CO^R(j)$. For simplicity, two signals $CO^L(j)$ and $CO^R(j)$ are considered separately.

As to the calculation of $CO^L(j)$, four possible cases should be considered:

(1) Case X^L : $W^L(j) = w^L(j) = W^R(j)$,

(2) Case L^L : $W^L(j) = w^L(j) < W^R(j)$,

(3) Case N^L : $W^L(j) < w^L(j) < W^R(j)$,

(4) Case R^L : $W^L(j) < w^L(j) = W^R(j)$.

The signal $CO^L(j)$ is determined as shown in Table 2.1 (a). This table can be made based on the following analyses.

Case X^L
In this case, $W^L(j)$, $w^L(j)$, and $W^R(j)$ coincide.
◇If $CI^L(j)=CI^R(j)$ then $W^L(j+1)$ and $W^R(j+1)$ will also coincide, and hence $w^L(j+1)$ must be equal to them.
◇If $CI^L(j)=0$ and $CI^R(j)=1$, then the search direction is determined by the comparison result.
◇Since $W^L(j+1) \le W^R(j+1)$, $(CI^L(j), CI^R(j))$ can not

Table 2.1. Calculations of $CO^L(j)$ and $CO^R(j)$.

(a) Calculations of $CO^L(j)$.

Condition $COND^L$:
$k \le T^{Rj}(w^L(j))$ or $w^L(j) > LOAD(j)-1$

Case	$CI^L(j)$	$CI^R(j)$	$COND^L$	$CO^L(j)$
X^L	0	0	-	0
	0	1	F	1
	0	1	T	0
	1	0	-	-
	1	1	-	1
L^L	0	-	F	1
	0	-	T	0
	1	-	-	1
N^L	-	-	F	1
	-	-	T	0
R^L	-	0	-	0
	-	1	F	1
	-	1	T	0

(b) Calculations of $CO^R(j)$.

Condition $COND^R$:
$k < T^{Rj}(w^R(j))$ or $w^R(j) > LOAD(j)-1$

Case	$CI^L(j)$	$CI^R(j)$	$COND^R$	$CO^R(j)$
X^R	0	0	-	0
	0	1	F	1
	0	1	T	0
	1	0	-	-
	1	1	-	1
L^R	0	-	F	1
	0	-	T	0
	1	-	-	1
N^R	-	-	F	1
	-	-	T	0
R^R	-	0	-	0
	-	1	F	1
	-	1	T	0

be (1, 0).

Case L^L
In this case, $W^L(j)$ and $w^L(j)$ coincide and $W^R(j)$ is different from $w^L(j)$.
◊If $CI^L(j)=1$, then the search can not proceed to the left son, i.e., $w^L(j+1)$ must be equal to $W^L(j+1)$.
◊If $CI^L(j)=0$, then the search direction is determined by the comparison result.

Case N^L
In this case, the search process will never encounter the interval boundaries determined by the MISEE. The search direction is determined by the comparison result.

Case R^L
Similar to L^L.

As to $CO^R(j)$, the possible cases are

(1) Case X^R : $W^L(j) = w^R(j) = W^R(j)$,

(2) Case L^R : $W^L(j) = w^R(j) < W^R(j)$,

(3) Case N^R : $W^L(j) < w^R(j) < W^R(j)$,

(4) Case R^R : $W^L(j) < w^R(j) = W^R(j)$,

and the calculation is as shown in Table 2.1 (b), which can be obtained based on similar analyses as before.

Although we have assumed that $W^L(j)$ and $W^R(j)$ are visible, actually they are not. This problem is solved by introducing states of a search process. For each search key, its search process changes its state as it moves from the top level to the bottom level. The following combinations of the cases that are respectively chosen from Table 2.1 (a) and (b) can be considered as all the possible states of search processes, i.e.,

(X^L, X^R) : $W^L(j) = w^L(j) = w^R(j) = W^R(j)$,

(L^L, L^R) : $W^L(j) = w^L(j) = w^R(j) < W^R(j)$,

(L^L, N^R) : $W^L(j) = w^L(j) < w^R(j) < W^R(j)$,

(L^L, R^R) : $W^L(j) = w^L(j) < w^R(j) = W^R(j)$,

(N^L, N^R) : $W^L(j) < w^L(j) \le w^R(j) < W^R(j)$,

(N^L, R^R) : $W^L(j) < w^L(j) < w^R(j) = W^R(j)$,

(R^L, R^R) : $W^L(j) < w^L(j) = w^R(j) = W^R(j)$,

At the top level, each search process initializes its state to (X^L, X^R). The state transition is specified by two automata, each of which specifies transitions among either the set $\{X^L, L^L, N^L, R^L\}$ or the set $\{X^R, L^R, N^R, R^R\}$. These automata are described in Table 2.2 (a) and (b). These tables can be obtained by similar analyses made for $CO^L(j)$ before. A part of Table (a) can be obtained as follows.

Table 2.2 The state transition of search processes.

(a) the state transition among X^L, L^L, N^L, and R^L.

Condition $COND^L$:
$k \le T^{Rj}(w^L(j))$ or $w^L(j) > LOAD(j)-1$

Case	$CI^L(j)$	$CI^R(j)$	$COND^L$	Next State
X^L	0	0	-	X^L
	0	1	F	R^L
	0	1	T	L^L
	1	0	-	-
	1	1	-	X^L
L^L	0	-	F	N^L
	0	-	T	L^L
	1	-	-	L^L
N^L	-	-	-	N^L
R^L	-	0	-	R^L
	-	1	F	R^L
	-	1	T	N^L

(b) the state transition among X^R, L^R, N^R, and R^R.

Condition $COND^R$:
$k < T^{Rj}(w^R(j))$ or $w^R(j) > LOAD(j)-1$

Case	$CI^L(j)$	$CI^R(j)$	$COND^R$	Next State
X^R	0	0	-	X^R
	0	1	F	R^R
	0	1	T	L^R
	1	0	-	-
	1	1	-	X^R
L^R	0	-	F	N^R
	0	-	T	L^R
	1	-	-	L^R
N^R	-	-	-	N^R
R^R	-	0	-	R^R
	-	1	F	R^R
	-	1	T	N^R

Case X^L
◊If $CI^L(j)=CI^R(j)$, then $W^L(j+1)$ and $W^R(j+1)$ will also coincide, i.e., the state will stay at X^L.

◇If $CI^L(j)=0$ and $CI^R(j)=1$, $W^L(j+1)$ and $W^R(j+1)$ will separate, and $w^L(j+1)$ will become equal to either $W^L(j+1)$ or $W^R(j+1)$ depending on the compaison result. Hence, the next state will be either L^L or R^L.

Other transitions in the table can be similarly understood.

An example search process with a 1-digit MISEE and a 1-digit bit-sliced ISEE is shown in Fig. 2.5.

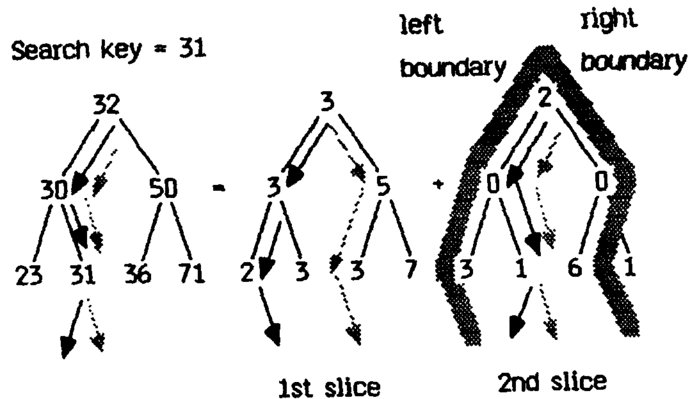

Fig. 2.5. An example search process with two bit-sliced ISEE.

A set of bit sliced ISEE with a same width, when concatenated, forms a wider ISEE with arbitrary wordlength. The i-th slice of a search key must be input to the i-th module at the time (i-1) steps after the input of the first slice. The result of each serach process is output from the rightmost module as its $w^L(L)$ and $w^R(L)$.

A bit-sliced ISEE with L levels and 1 bit width requires 6L+1 pins; 1 pin for data input both in table loading and in batch searchessing, 4L pins for the connections between consecutive slices, and 2L pins for the output of $w^L(L)$ and $w^R(L)$. For L=12, It becomes 73 pins. However, $w^L(L)$ and $w^R(L)$ need not be output by each slice. Actually, they can be constructed by a simple circuit from CO^Ls and CO^Rs output from the least significant slice. Therefore, the pin complexity can be decreased to 4L+1. For L=12, this is 49 pins, which seems to be acceptable.

The slicing method proposed above causes no serious overheads, nor it excessively increases the pin complexity. Therefore, the width of each slice can be decreased to a single bit without causing any problems, which decreases the hardware complexity and increases both the flexibility of the module and the feasibility of its VLSI implementation.

3. Bit-Sliced Sort Hardware

3.1. Two-way-merge sorter

This section describes briefly the operations performed by the two-way-merge sorter proposed by S. Todd [TODD78]. He proposed a sorting algorithm that repetitively applies merge operations to every two sorted runs in an input stream to increase the length of sorted runs. The initial input stream is considered as a sequence of sorted runs of length one. Each stage merges every two runs from the head of the input stream to output a sorted run of double length. The sequence of these output runs becomes an input stream of the next stage. In order to perform these repetitive merge operaions in a pipeline fashion, a two-way-merge sorter has a hardware configuration similar to that of an ISEE in Fig. 2.2, however their logic circuits at each level are different. An input stream flows into this module at the top level, and flows out from the bottom level. The logic circuits at each level performs the following operations. At every time when the next two input runs arrives at a stage, it begins to merge these two runs to output a merged run to the next level. Each input element is first stored in the memory bank at this stage before it is manipulated. A hardware module with L levels outputs from the bottom level a sorted run of length 2^L, and hence it can sort a set of no more than 2^L elemens.

A bit-sliced architecture of this module can be easily designed if we can find out how to slice a merger used at each level of this module. Let L and R denote two sorted streams of same length, and L(i), R(i) their i+1st elements. The stream L is referred to by the left stream, while R the right stream. The logic circuits at each level of a two-way-merge sorter receive two streams L and R one after another and merge them. The circuits can be decomposed into two parts. The first part receives an input element at every step and stores it at a proper address of the memory bank at this level, while the second part merges two streams, whose next elements are always guranteed to have been already stored in the memory bank by the first part of the circuits. As to the merge operation, the following property holds.

Property 3.1
At every time, say, j steps after the arrival of the head of the second stream R, the merger can see L(i) and R(i) if the index i is less than or equal to j and if they have not been output yet.

The design of a bit-sliced merger applicable to a two-way-merge sorter should take this property into consideration.

3.2. Bit-sliced merger applicable to a two-way-merge sorter

A bit-sliced merger has two 1-bit input lines LI and RI, and two 1-bit output lines LO and RO. These lines are used to connect multiple bit-sliced mergers to form a single merger with an arbitrary wordlength as shown in Fig. 3.1. The operation of the i-th slice module is delayed (i-1) steps from that of the leftmost module.

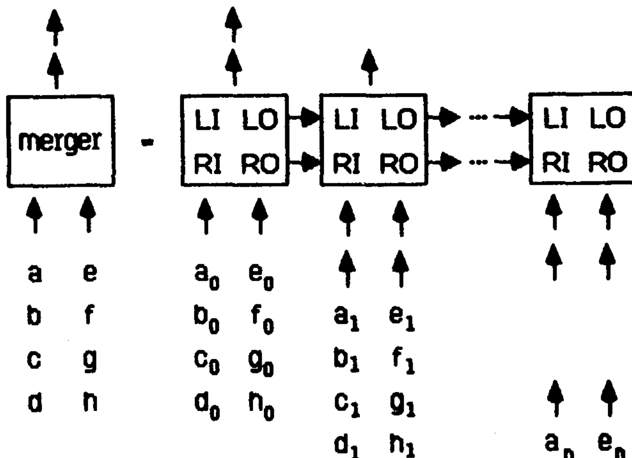

Fig. 3.1. Connections of bit-sliced mergers.

The slicing of a merger can not be achieved in a straight forward way because of the following reason. Suppose that the next elements of the two input streams have a common value at the leftmost slice. Then we can not decide which one to take out. Suppose that we have chosen the left one. Suppose also that, in this case, the next element of the right stream is less than that of the left stream at the second slice. Then the second slice should choose the right head to output. However, this makes it insignificant to continue further comparison operations at these two slices because the pairs of the elements compared next at these two slices do not correspond with each other.

This problem can be solved as follows. Let us introduce two pointers pointing to the next elements of the input streams. The left pointer lp points to the next element of the left stream, while rp points to that of the right stream. If L(lp) and R(rp) are equal at the leftmost slice, we will advance both of the two pointers, and make the module at this slice to output one value that is equal to both L(lp) and R(rp). The output signals LO and RO are both set to one. The signal LO denotes the advance of the left pointer, while RO the advance of the right pointer. They are set to one if their corresponding pointers are advanced. Otherwise, they are set to zero.

Suppose that the next i elements of L and the next j elements of R have a same value v at the leftmost slice. Assume that i is less than or equal to j. Then the first (i+j) output values from this module should be equal to v. During the first i steps, both pointers will be simultaneously advanced, and the value v will be output i times. After these operations, the module is said to have reached the left boundary. During the following (j-i) steps, only the pointer rp will be advanced. Finally, both L(lp) and R(rp) will become greater than v. At this time, the module is said to have reached both the left boundary and the right boundary. The module has already output v j times, however there still remain i elements to output the value v. Therefore, the module must output v during the following i steps. During this period, the comparison of the next elements is suspended, and hence neither of the pointers is advanced.

Now let us consider the second slice of a merger. Suppose that, in the leftmost slice, the left boundary and the right boundary are respectively located at the i-th next element and at the j-th next element as before. The second slice of a merger operates in a similar way as the leftmost one does unless it reaches either the left boundary or the right boundary of the leftmost slice. If it reaches, say, the left boundary, it must stop the advance of the left pointer. The following output must be selected from the right stream untill the right pointer also reaches the right boundary of the leftmost slice.

From these considerations, a bit-sliced merger works as the leftmost slice of a merger if both of its connection input LI and LO are set to one. The following classification of the processing status will make it easier to describe the operations of a bit-sliced merger. Let us first introduce several notations:

C : A counter that counts how many times the two pointers are simultaneously advanced. Initially zero.

v_0 : The previous output; initially zero.

D^L : The difference of lp between the current slice and the preceding slice. It becomes zero when the module reaches the left boundary of the preceding slice. Otherwise, it is kept positive. Initially zero.

D^R : The difference of rp between the current slice and the preceding slice. It becomes zero when the module reaches the right boundary of the preceding slice. Otherwise, it is kept positive. Initially zero.

$D^{L\prime} = D^L + LI.$

$D^{R\prime} = D^R + RI.$

The classification and the operations in each case are described below:

Case 1. $D^{L\prime} = D^{R\prime} = C = 0$: nonexistent.

Case 2. $D^{L'} = D^{R'} = 0$, $C\ (= n) > 0$

The two pointers lp and rp have reached the boudaries specified by the lefthand slices, however, there are still n elements with a value v_0 that remains to be output during the following n steps. In this case, neither of the pointers should be advanced. The value v_0 is output and C is decreased by one.

$(LO, RO) \leftarrow (0, 0)$; output v_0; $C \leftarrow C-1$;

$D^L \leftarrow D^{L'}$; $D^R \leftarrow D^{R'}$.

Case 3. $D^{L'} = 0$, $D^{R'} > 0$

The pointer lp has reached the left boundary specified by the lefthand slices.

Case 3.1. $C = 0$ or $R(rp) = v_0$

There remains no elements equal to v_0 that are to be output before the next comparison. Or R(rp) is equal to v_0. In this case, R(rp) is output, and rp is advanced.

$(LO, RO) \leftarrow (0, 1)$; output $R(rp)$;

$v_0 \leftarrow R(rp)$; $rp \leftarrow rp+1$; $D^L \leftarrow D^{L'}$;

$D^R \leftarrow D^{R'}-1$.

Case 3.2. $C \neq 0$ and $R(rp) \neq v_0$

The pointer rp has reached the right boundary of this slice. In this case, the advance of rp and lp should be suspended. The remaining v_0 is output, and hence C is decreased by one.

$(LO, RO) \leftarrow (0, 0)$; output v_0; $C \leftarrow C-1$;

$D^L \leftarrow D^{L'}$; $D^R \leftarrow D^{R'}$.

Case 4. $D^{L'} > 0$, $D^{R'} = 0$

Similar to Case 3. The roles of left and right are interchanged.

Case 4.1. $C = 0$ or $L(lp) = v_0$

L(lp) is output, and lp is advanced.

$(LO, RO) \leftarrow (1, 0)$; output $L(lp)$;

$v_0 \leftarrow L(lp)$; $lp \leftarrow lp+1$; $D^L \leftarrow D^{L'}-1$;

$D^R \leftarrow D^{R'}$.

Case 4.2. $C \neq 0$ and $L(lp) \neq v_0$

The advance of the pointers should be suspended. The remaining v_0 is output, and hence C is decreased by one.

$(LO, RO) \leftarrow (0, 0)$; output v_0; $C \leftarrow C-1$;

$D^L \leftarrow D^{L'}$; $D^R \leftarrow D^{R'}$.

Case 5. $D^{L'} > 0$, $D^{R'} > 0$

Neither of the pointers has reached the boundary specified by the lefthand slices.

Case 5.1. $\min(L(lp), R(rp)) \neq v_0$ and $C \neq 0$

Both of the pointers have reached the boundaries specified by this slice. In this case, the advance of rp and lp should be suspended. The remaining v_0 is output, and hence C is decreased by one.

$(LO, RO) \leftarrow (0, 0)$; output v_0; $C \leftarrow C-1$;

$D^L \leftarrow D^{L'}$; $D^R \leftarrow D^{R'}$.

Case 5.2. $C = 0$ or $\min(L(lp), R(rp)) = v_0$

There remains no element equal to v_0 that are to be output before the next comparison. Or the smaller of L(lp) and R(rp) is equal to v_0.

Case 5.2.1. $L(lp) < R(rp)$

The next element of the left stream is less than that of the right one. L(lp) is chosen to output, and lp is advanced.

$(LO, RO) \leftarrow (1, 0)$; output $L(lp)$;

$v_0 \leftarrow L(lp)$; $lp \leftarrow lp+1$; $D^L \leftarrow D^{L'}-1$;

$D^R \leftarrow D^{R'}$.

Case 5.2.2. $L(lp) > R(rp)$

The next element of the right stream is less than that of the left one. R(rp) is chosen to output, and rp is advanced.

$(LO, RO) \leftarrow (0, 1)$; output $R(rp)$;

$v_0 \leftarrow R(rp)$; $rp \leftarrow rp+1$; $D^L \leftarrow D^{L'}$;

$D^R \leftarrow D^{R'}-1$.

Case 5.2.3. $L(lp) = R(rp)$

The two next elements are same. Their common value is output. Both of the pointers are simultaneously advanced, and C is increased by one.

$(LO, RO) \leftarrow (1, 1)$; output $L(lp)$;

$v_0 \leftarrow L(lp)$; $C \leftarrow C+1$; $lp \leftarrow lp+1$;

$rp \leftarrow rp+1$; $D^L \leftarrow D^{L'}-1$; $D^R \leftarrow D^{R'}-1$.

An example merge process with two bit-sliced

mergers is shown in Appendix.

The slicing method described above is applicable to a two-way-merge sorter. The operation of this bit-sliced merger is consistent with Property 3.1 of the mergers that are used in a two-way-merge sorter. The bit-sliced two-way-merger thus obtained does not suffer from the serious increase of the pin complexity. A module with L levels and 1 bit width requires (4L+1) pins, 1 pin for data input and data output, and 4L pins for connections between consecutive slices. For L=12, this is 49 pins, which seems to be acceptable. The hardware complexity is much decreased by the bit-slicing, while its flexibility is increased. Its VLSI implementation becomes far more feasible than a nonsliced two-way-merge sorter.

4. Conclusion

For the high speed processing of databases, it is fundamental to introduce various VLSI architectures to the processing of basic functions. Especially, sort and batch search that are fundamental in database processing require high speed modules. The VLSI algorithms of them should make much use of the time necessary for the transfer of a large amount of data to and from the modules. Besides, these modules should be nonprogrammable in order to avoid serious overheads that are likely to be caused by the introduction of the programmability. However, they should be able to extend their capacity and wordlength by the connection of them.

This paper has solved the problem of how to extend the wordlength of search and sort hardware modules. It has proposed the bit-sliced architectures of an interval search engine and a two-way-merge sorter. The slice of these engines does not cause excessive overheads. The decrease of the slice length decreases the hardware complexity, and increases the flexibility of the modules. Therefore, it increases the feasibility of the VLSI implementation of these hardware modules.

The two different bit-sliced VLSI architectures respectively for search and sort both require (4L+1) pins per each chip. For L=12, i.e., if the capacity of each module is restricted to 4095 words, their pin complexity becomes 49+α, where α pins are necessary for power supply, clock supply, and mode control. This number seems to be acceptable. The bit-sliced ISEE with L levels and 1 bit width consists of 2^L-1 memory cells (, for L=12, 4095 cells) as a whole, and two 4 state automata at each level, while the bit-sliced sorter with L levels and 1 bit width consists of $2(2^L-1)$ memory cells as a whole, and a simple logic circuit with several registers at each level. Therefore, even if we use static RAM technology, each module with L levels and 1 bit width requires less than 10^5 transistors. This number ensures their feasibility.

Bit-slicing is fundamental for the flexibility increase. The arrangement shown in Fig. 4.1, for example, can reconfigure itself to cope with various wordlength. Consecutive modules in the array of bit-sliced ISEEs are separated by a control signal line $control_i$. When $control_i$ is set to one, the subarray of modules to the left of this control signal line and the remaining right subarray can work independently. If all control signals are set to zero, the whole array of modules can work as a single ISEE. The same kind of reconfigurable arrangement is also possible for bit-sliced two-way-merge sorters.

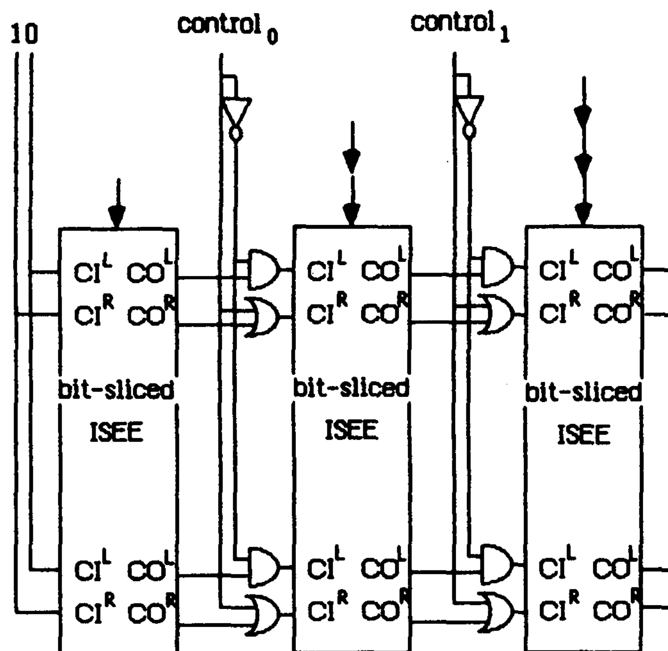

Fig. 4.1. Wordlength-controllable connections of bit-sliced ISEEs.

Referrences

[BATC68] Batcher, K.E., 'Sorting networks and their applications,' Proc. SJCC, Apr. 1968, pp.307-314.

[HIRS78] Hirschberg, D.S., 'Fast parallel sorting algorithms,' CACM, vol. 21, no. 8, Aug. 1978, pp.657-661.

[MULL75] Muller, D.E. and Preparata, F. P., 'Bounds for complexity of networks for sorting and switching,' JACM, vol. 22, no. 2, Apr. 1975, pp.195-201.

[NASS79] Nassimi, D. and Sahni, S., 'Bitonic sort

on a mesh connected parallel computer,' IEEE Trans. on Computers, vol. c-27, no. 1, Jan. 1979, pp.2-7.

[PREP78] Preparata, F.P.,'New parallel sorting schemes,' IEEE Trans. on Computers, vol. c-27, no. 7, July 1978, pp.669-673.

[STON71] Stone, H.S., 'Parallel processing with the perfect shuffle,' IEEE Trans. on Computers, vol. c-20, no. 2, Feb. 1971, pp.153-161.

[TANA80] Tanaka, Y., Nozaka, Y. and Masuyama, A., 'Pipeline searching and sorting modules as components of a data flow database computer,' IFIP Congress 80, Oct. 1980, pp.427-432.

[THOM77] Thompson, C.D. anf Kung, H.T., 'Sorting on a mesh connected parallel computer,' CACM, vol. 20, nó. 4, Apr. 1977, pp.263-271.

[TODD78] Todd, S., 'Algorithm and hardware for a merge sort using multiple processors,' IBM J. R&D, vol. 22, no. 5, May 1978.

APPENDIX An example merge process with two bit-sliced merger.

input stream : L = (00, 00, 01, 11, **) (* : delimiter)
R = (00, 01, 10, 11, **)

1st slice : L = (0, 0, 0, 1, *) R = (0, 0, 1, 1, *)

2nd slice : L = (0, 0, 1, 1, *) R = (0, 1, 0, 1, *)

operations

time	0	1	2	3	4	5	6	7	8
1st slice									
lp	0 1	1 2	2 3	3 3	3 3	3 4	4 4	4 4	
rp	0 1	1 2	2 2	2 2	2 2	2 3	3 4	4 4	
L(lp)=R(rp)	T	T	F	T	T	T	F	T	
$min(L(lp),R(rp))=V_0$	-	T	T	F	F	F	T	F	
C	0 1	1 2	2 2	2 1	1 0	0 1	1 1	1 0	
output	0	0	0	0	0	1	1	1	
LO	1	1	1	0	0	1	0	0	
RO	1	1	0	0	0	1	1	0	
2nd slice									
LI		1	1	1	0	0	1	0	0
RI		1	1	0	0	0	1	1	0
lp		0 1	1 2	2 2	2 3	3 3	3 3	3 4	4 4
rp		0 1	1 1	1 1	1 2	2 2	2 3	3 4	4 4
L(lp)=R(rp)		T	F	T	T	F	F	T	T
$min(L(lp),R(rp))= V_0$		-	T	F	F	T	T	F	F
C	0	0 1	1 1	1 0	0 1	1 0	0 0	0 1	1 0
D^{L}'	0	1 0	1 0	1 1	1 0	0 0	1 1	1 0	0 0
D^{R}'	0	1 0	1 1	1 1	1 0	0 0	1 0	1 0	0 0
output		0	0	0	1	1	0	1	1
case #		5.2.3	5.2.1	5.1	5.2.3	2	5.2.2	5.2.3	2

output

1st slice	0	0	0	0	0	1	1	1	
2nd slice		0	0	0	1	1	0	1	1
concatenation		00	00	00	01	01	10	11	11