CMU-CS-80-114

Systolic (VLSI) Arrays for Relational Database Operations

H. T. Kung and Philip L. Lehman

October 1979

[Last revised March 1980]

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-80-114

Systolic (VLSI) Arrays for Relational Database Operations

H. T. Kung and Philip L. Lehman

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

October 1979

[Last revised March 1980]

To be presented at the 1980 ACM/SIGMOD International Conference on Management of Data, Los Angeles, California, May 14-16, 1980.

This research was supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551, the National Science Foundation under Grant MCS 78-236-76, and the Office of Naval Research under Contracts N00014-76-C-0370 and N00014-80-C-0236.

Abstract

This paper proposes the use of VLSI technology to perform relational database operations directly in hardware. It is shown that relational computations, such as *intersection*, *remove-duplicates*, *union*, *join*, and *division*, can all be pipelined elegantly and efficiently on networks of processors having an array structure. These (systolic) processor arrays are readily and cost-effectively implementable with present technology, due to the extreme simplicity of their processors, and the high regularity of their interconnection structures.

Table of Contents

1. Introduction

LSI technology allows tens of thousands of devices to fit on a single chip; VLSI technology promises an increase of this number by at least one or two orders of magnitude in the next decade. This paper proposes one method of exploiting this technology advance: the construction of special-purpose VLSI chips for relational database operations. These special-purpose chips are to be attached to a conventional host computer, or used as a component in a larger special-purpose system, such as a database machine. (We suggest one such database machine at the end of this paper.)

In [5] a structure called a systolic array[1] is proposed for implementation in VLSI. These arrays of processors have the following desirable properties:

1. They can be designed and implemented with only a *few* different types of *simple* cells.

2. The array's data and control flow is *simple* and *regular*, so that cells can be connected by a network with local and regular interconnections. Long distance or irregular communication is not needed.

3. The array uses extensive *pipelining* and *multiprocessing*. Typically, several data streams move at constant velocity, over fixed paths in the network, interacting where they meet. In this fashion, a large proportion of the processors in the array can be kept active, so that the array can sustain a high rate of data flow.

VLSI designs based on systolic arrays tend to be simple (a consequence of property 1), modular (property 2) and of high *performance* (property 3) -- for more discussion of the attractiveness of the systolic array approach, see [3]. In the present paper we illustrate the use of systolic arrays in performing relational database operations.

In section 2 we give details concerning the notion of systolic arrays, and present some concepts and notation for discussing relational database operations. In section 3, we describe the basic building block of several of our systolic arrays: a systolic processor array to compare two tuples. Section 4 includes a detailed systolic example: an array to rapidly perform the intersection (or difference) operation on two relations. In section 5 we use an array identical to the intersection/difference array, to remove duplicates from acollection of

[1] The word "systole" was borrowed from physiologists, who use it to refer to the rhythmically recurrent contractions of the heart, which pulse blood through the body. For a systolic array, the action of a cell or processor is analogous to that of the heart. Each cell regularly pumps data in and out (performing some short computation before each "contraction"), so that a regular flow of data is kept up in the network. Many systolic arrays have been designed recently, and are surveyed in [7].

tuples, and to perform the operations of union and projection on relations. In sections 6 and 7 we detail relational operations (join and division) that are substantially different from the intersection-like operations, but still lend themselves to simple implementation with systolic arrays. Section 8 remarks on some implementation and performance aspects of the systolic arrays proposed in this paper. Section 9 discusses the architectural issues of an integrated system capable of using many types of systolic arrays.

2. Systolic Arrays and Relational Database Considerations

2.1 Systolic Arrays

Regular geometric structures are typically used in systolic arrays. For the present paper we use predominantly orthogonally and linearly connected arrays of processors (both of which are shown in figure 2-1), although hexagonally connected arrays as in [5] would work as well in many instances.

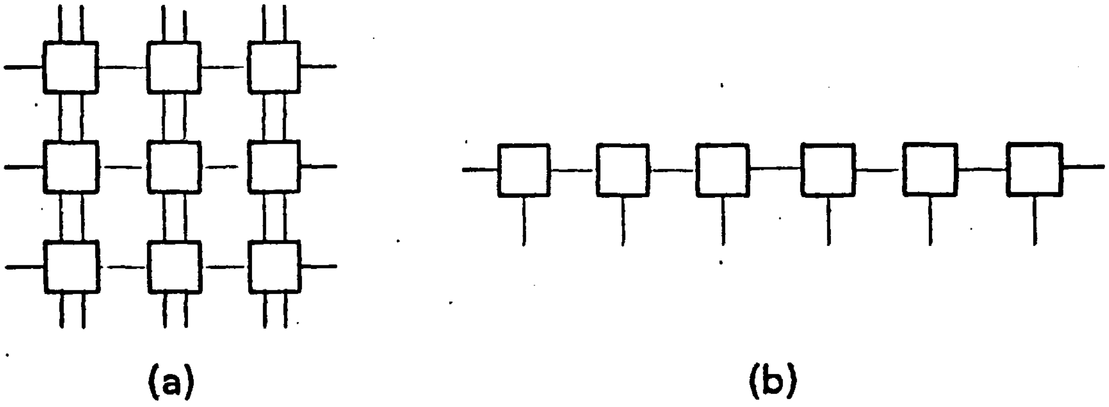

Figure 2-1: Orthogonally and linearly connected processor arrays.

We find that these arrays facilitate many relational database operations by allowing swift interaction among the tuples of two relations, with a set of temporary results also traveling through the array. Typically, the relations move top-to-bottom and bottom-to-top, and the temporary results move left-to-right. All of the data in the array moves synchronously. As a piece of data passes through a processor, it may have some computation performed on it; then it is passed on to the next processor. The final results of the array are sent out a side of the array.

2.2 Processors

In figure 2-2 we show the prototype for the processor used in the orthogonally or linearly connected systolic structure. The processor has three input lines and three output lines. For each "pulse" of the systolic array, inputs come in on the input lines, and outputs leave the processor on the output lines. In the intervening time, all of the work (computation) of the processor is performed -- the processor computes some simple transformation on the data which it has just received, in preparation for shipping it out at the next pulse. VLSI arrays

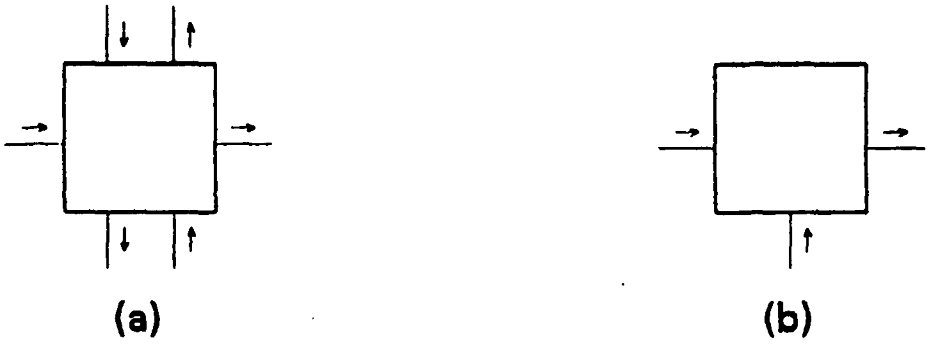

Figure 2-2: Orthogonal and linear processor prototypes.

are greatly simplified if most processors in the array are identical. This is the case for the arrays presented in this paper. Given the orthogonally or linearly connected array structure, and the processor prototype described here, it is the algorithm actually executed by each processor that determines the function of the array. Therefore, to define a systolic array to perform a specific relational operation, we specify the algorithm for the processors in a systolic array. The sections below consist of such specifications and an explanation of how they actually produce the desired result.

2.3 Representation of Relations

In the following discussion, we assume some familiarity with the basics of relational database theory (see, for example, [1, 2]). A relation is a set of tuples. Each tuple consists of an ordered sequence of elements. It is these elements that are fed through our systolic arrays. The tuples in a relation, however, are not necessarily ordered in any particular fashion.

In a relation, an element can be of any data type: an integer, a boolean value, a string, etc. We wish to give all of these a uniform representation, in order to simplify the design of systolic arrays to process relations. The assumption we make is a common one in the implementation of relational database systems. We assume that the elements from any particular column in a relation are selected only from one underlying domain. Each member of the domain is uniquely and reversably encoded into an integer. These integer encodings are the form in which the elements are stored in the relations, and the list of encodings is stored separately. Whenever necessary, the integers are decoded into the appropriate value; however, encoding and decoding are usually only necessary for input or output: that is, for use by humans. Most relational operations are logically the same whether they operate on integers or, say, strings or calendar dates. Since -- for our purposes -- integer operations are more convenient, we assume that relations are stored as tuples of integers (and we are not concerned with encoding and decoding).

2.4 Union-Compatibility

Certain relational operations such as union and intersection can only be performed between relations that are *union-compatible*. Two relations are said to be union-compatible if the following two conditions hold:

- They have the same number of columns (and thus tuples from the two relations have the same number of entries).

- Corresponding columns from the two relations have entries drawn from the same underlying domain.

This definition is an attempt to capture the informal notion that a tuple from one relation *could legally be a member of the other relation*, in that the respective columns of the two relations are defined on the same domains.

2.5 Multi-relations

A *multi-relation* is an extension of the concept of a relation in which duplicate tuples are allowed. (This is by analogy with the term "multi-set," since a relation can be viewed as a set of tuples.) This is a notion that we will find useful later in the paper. Multi-relations are usually generated as the intermediate results of relational operations. For example, suppose we remove a few columns from a relation (which is the projection operation). The intermediate construct we obtain before we remove duplicate tuples to produce the new (result) relation is a multi-relation.

2.6 Notation

We briefly summarize the notation used in the remainder of the paper. Relations and multi-relations are denoted by capital letters: A, B, C. Tuples that are members of these are denoted by subscripted lower-case letters. The *i*th tuple of A is denoted by a_i, or by $a_i \in A$, if we wish to indicate membership. In turn, elements in tuples are double-subscripted: a_{ik} is the *k*th element of a_i, and the whole tuple can be exhibited as $a_i = <a_{i,1}, a_{i,2}, \ldots, a_{im}>$. The letter n is usually used to denote the number of tuples in a relation (the cardinality of the relation, since a relation is a set): $|A| = n$. The letter m usually designates the number of elements in a tuple in the relation in question.

Letter T represents a boolean matrix that contains results of logical operations. The (*i*,*j*)-th entry of T, t_{ij}, is usually used to denote the result of a comparison between the *i*th tuple of a relation and the *j*th tuple of another. Where we wish to display the formation of t_{ij} over time, we use the notation t_{ij}^{k} for the result after the *k*th time step; $t_{ij}^{initial}$ and t_{ij}^{final} denote specific instances (the first and the last) of t_{ij}^{k} (When no confusion will thereby result, we use the same notation t_{ij} to refer to t_{ij}^{k} for any *k*.) Finally, the notation t_{i} is used to designate the result of some logical operation on all of the members of the *i*th row of T, for example, the *OR* or *AND* of t_{ik}, for all *k*.

3. Arrays for Tuple Comparison

In several of the relational operations described below, it is necessary to test for equality between a pair of tuples, one from each of two relations. (Two tuples, $a_i \in A$ and $b_j \in B$, where A and B are union-compatible relations or multi-relations, are said to be equal if and only if element a_{ik} equals element b_{jk} for $1 \le k \le m$.) For example, in the intersection operation, the intersection of two relations, say A and B, consists of those tuples which are in both A and B. Forming this intersection, then, requires many tests for equality between tuples, $a_i \in A$ and $b_j \in B$. In this section, we first describe a linear systolic array of processors capable of performing one such comparison. We then combine many copies of this basic structure to form a two-dimensional systolic array that can pipeline many tuple comparisons.

3.1 Linear Comparison Array for Performing One Tuple Comparison

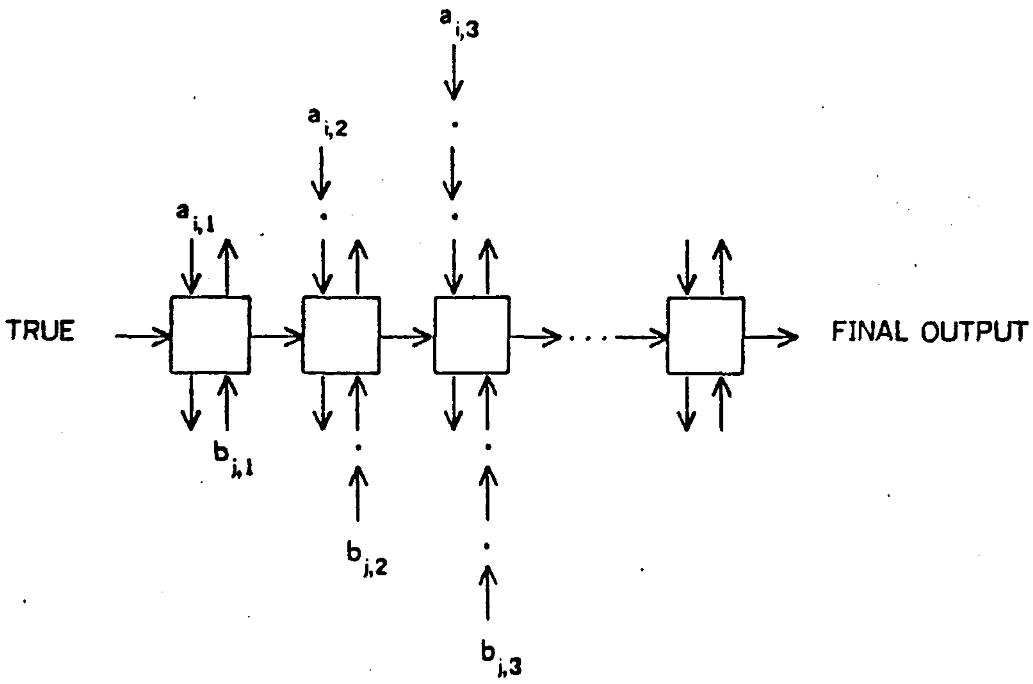

Figure 3-1: Tuple comparison array.

A tuple comparison can be done by the linear array of processors in Figure 3-1. A single processor from the array is shown in more detail in Figure 3-2. One can see that the processor array in Figure 3-1 is able to compute the *AND* of the comparison results from all of the individual element comparisons. More precisely, at each step the *k*th processor (from

$$t_{OUT} \leftarrow t_{IN} \wedge (a_{IN} = b_{IN})$$

$$a_{OUT} \leftarrow a_{IN}$$

$$b_{OUT} \leftarrow b_{IN}$$

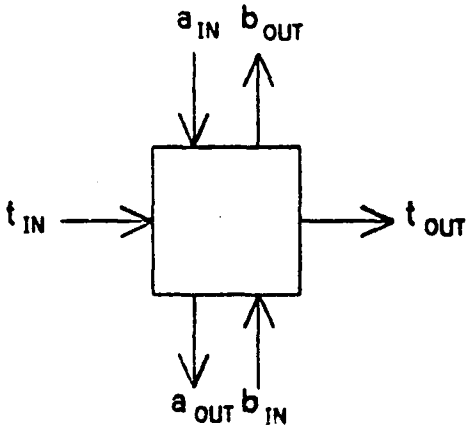

Figure 3-2: Individual comparison processor.

the left) in the array compares the two elements a_{ik} and b_{jk}, and outputs on its output line t_{OUT} the *AND* of this comparison result with the input to the processor on input line t_{IN} (which is the output of the (k-1)st processor). Thus, if the input to the left-most processor is the value *TRUE*, then, by induction, after m time steps the output at the right-most processor of the processor array will be a bit indicating whether the two tuples are equal. That is, this output will be *TRUE* if and only if all of the comparisons of individual elements produced *TRUE*. (Notice also that if the initial input is *FALSE*, then the output at the right side of the array is guaranteed to be false. Surprisingly, this fact will be useful in later sections of the paper.)

To make this all work, all of the data must be in the right place at the right time. This is why the inputs to the individual processors are "staggered" (as shown by the "slanted" input tuples in figure 3-1) so that elements a_{ik} and b_{jk} arrive at the kth processor and are compared at the kth time step. Also at that time the *AND* of the results of previous comparisons arrives at the same processor, so that it can be *AND*ed with the new comparison result at the processor.

We summarize the function of the linear comparison array shown in figure 3-1. This array compares two tuples (presumably one from each of two relations), and forms the result of the comparison by propagating intermediate versions of that result to the right through the array. By staggering entries from the tuples one can assure that the output from the right-most processor of the array will be the result of the equality test on the two tuples.

3.2 Two-Dimensional Comparison Array for Pipelining Many Tuple Comparisons

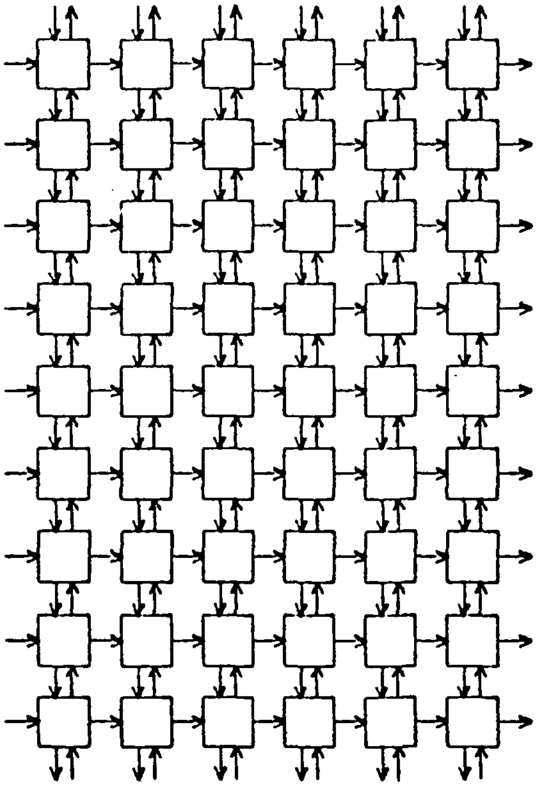

Figure 3-3: Two-dimensional (orthogonal) comparison array.

We concatenate, vertically, several of the linear comparison arrays described above, to form a 2-dimensional processor array, as shown in Figure 3-3. This orthogonally connected, 2-dimensional processor array can perform many tuple comparisons in parallel. To accomplish this, we feed the relations A and B into the array, from the top and bottom, respectively.

- We feed the relations at times such that the elements of any given tuple, say a_i, are "staggered," so that the element a_{ik} enters the array one time step before the element $a_{i,k+1}$. This has the effect of staggering the inputs to each of the component linear arrays, so that it will perform exactly as the single linear array described above.

- We pipeline tuples in each relation through the orthogonal processor array, in such a way that each tuple is two steps behind the tuple that preceeded it into the array. This assures that any particular pair of tuples $a_i \epsilon A$ and $b_j \epsilon B$ will eventually cross each other. More specifically, first $a_{i,1}$ will meet $b_{j,1}$ in the

> left-most processor of some row in the processor array. These two elements will be compared, and the result of this comparison will be *AND*ed with the initial input to that row of processors (*TRUE* for our present purposes). At the next time step, as the tuples ripple through the array, element $a_{i,2}$ will meet $b_{j,2}$ in the processor to the right, in the same row. They will be compared there, and the result of the comparison will be *AND*ed with the output from the first processor to produce the output of the second processor. Processing continues in this fashion, and the intermediate boolean result of the *AND*s propagates to the right through that particular row of processors, until -- as discussed above -- the right-most processor outputs a boolean value that indicates whether tuple a_i equals tuple b_j.

In Figure 3-4, the t_{ij} represent intermediate values for the results of comparing tuples a_i with tuples b_j. (Note that in the figure, the initial value for $t_{3,3}$ is just about to enter the processor array.)

3.3 Matrix Notation

For convenience in discussion, we express the results produced by a comparison array in the form of a matrix T. The elements of the matrix are defined as follows:

$$t_{ij} = \left| \begin{array}{ll} \mathit{TRUE} & \text{if } t_{ij}^{initial} = \mathit{TRUE}, \text{ and } a_{ik} = b_{jk} \text{ for all } 1 \le k \le m, \\ & \\ \mathit{FALSE} & \text{otherwise.} \end{array} \right.$$

We see that it is these t_{ij} that are produced at the right-most column of the array described in Section 3.2.

In the following sections, we add additional processors which manipulate these t_{ij}'s after they leave the comparison array. These manipulations will be shown to produce the equivalent of relational operations.

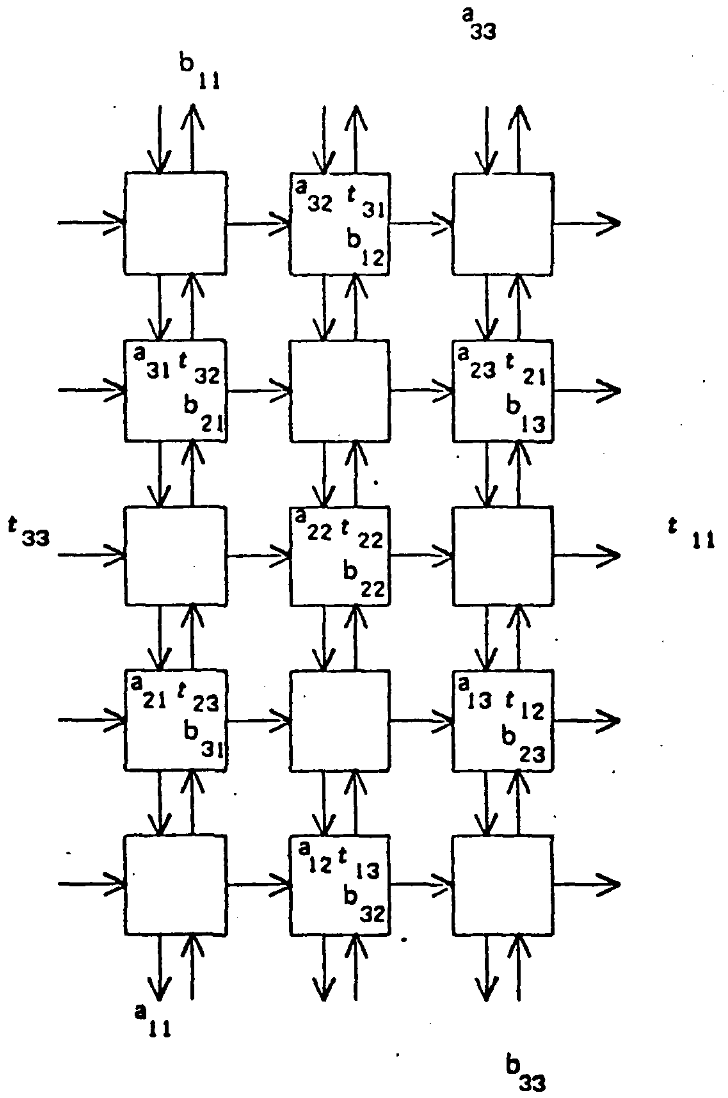

Figure 3-4: Data moving through the comparison array.

4. Arrays for Intersection -- A Detailed Example

In the preceding section, we saw how we could use a systolic comparison array to quickly do pairwise comparisons on sets of tuples. The results of these comparisons (t_{ij}) are sent out from the right side of the array. By examining a particular relational operation, namely intersection, in some detail, we illustrate how these individual results are combined in applications.

4.1 The Intersection Operation

Consider the operation of finding the intersection of two union-compatible relations

$$C = A \cap B.$$

The relation C consists of those tuples that are in both relation A and relation B. This is exactly the same as finding those tuples in A which are also in B. Thus we need only examine the tuples in A for membership in B. This is the basis for our "intersection array." We compare each tuple $a_i \epsilon A$ pairwise with each tuple $b_j \epsilon B$. For each a_i if a_i matches ***some*** b_j, then a_i is a member of the intersection. This is where the comparison array described in the preceeding section comes in handy.

4.2 The Intersection Array

The *intersection array* for performing the intersection operation consists of a (two-dimensional) comparison array on the left and a (linear) accumulation array on the right (see figure 4-1). The comparison array performs comparisons between tuples in A and tuples in B, to produce the matrix T, whereas the accumulation array accumulates t_{ij} to form:

$$t_i = OR_{1 \le j \le n}\, t_{ij} \qquad (4.1)$$

One can easily see that a tuple $a_i \epsilon A$ is a member of the intersection, *i.e.* a_i matches some $b_j \epsilon B$, if and only if t_i is true.

Figure 4-1 illustrates how the intersection array computes the intersection of two 3 x 3 relations. Processors in the accumulation array are called accumulation processors; their function is as follows. At each time step, an accumulation processor takes its left input (some t_{ij} from the comparison array), *OR*'s that with the top input (some t_i), and passes on the result as its output (the updated t_i) to the processor below. More specifically, a t_i is formed in the accumulation array in the following manner. First $t_{i,1}$ reaches an accumulation processor from the comparison array on the left. At the next time step, this value is sent to the accumulation processor below. During the same time step, $t_{i,2}$ is sent into that accumulation processor from the left, and is *OR*ed with $t_{i,1}$. Similarly, at the next time step, the result of this *OR* is sent down one processor, and is *OR*ed with $t_{i,3}$, which is just arriving

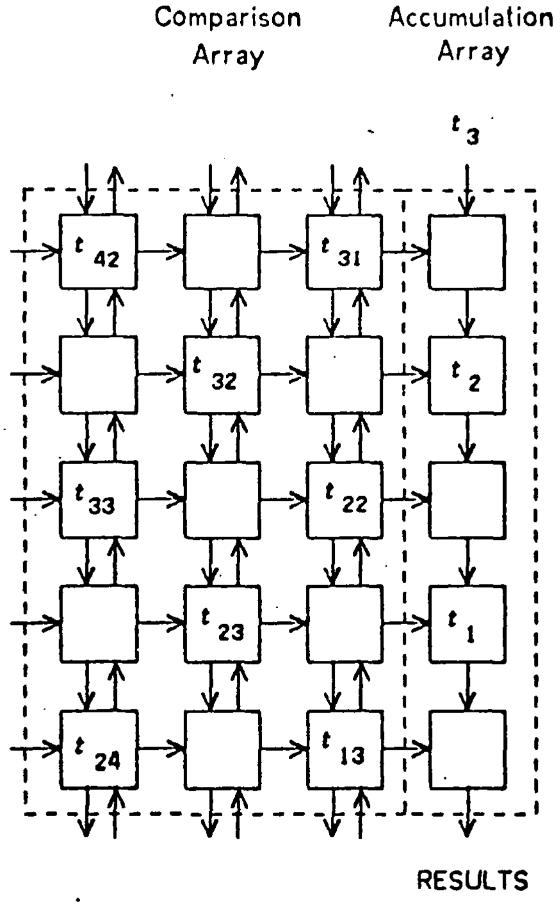

Figure 4-1: Intersection array, consisting of two modules: (2-dim) comparison array on the left, and (1-dim) accumulation array on the right.

from the left. In an implementation, the first accumulation processor can be identical in function to the others, provided we initialize the value moving down through the accumulation array as *FALSE* (*i.e.*, $t_i^{initial}$ = *FALSE*; in the figure, t_3 is about to enter the array with its initial value). This value is successively *ORed* with all of the t_{ik}, for all k, and when it leaves the bottom of the accumulation array, it takes on the value t_i, specified in equation (4.1). This t_i designates whether a_i is a member of the intersection C, and it is then a simple matter to use the t_i's to generate C from A.

At any time step, accumulation processors that aren't busy (i.e. that have no t_{ij} coming in from the left) simply pass on the t_i that they have. It takes less than the length of the accumulation array to produce a t_i, but different t_i are produced in different sub-arrays.

4.3 Remark

We have illustrated the use of the so-called accumulation array at the right of the comparison array to implement a desired relational operation, namely, the intersection operation. In general, as shown in the rest of the paper, only simple changes in the accumulation array or in the input data are required to alter the output of the array to produce other useful functions. The main "hardware" -- the comparison array -- is sufficiently general that it need not be changed at all.

As an illustration, we see that after a slight modification the intersection array can be used to perform the ***difference*** operation on two relations. The ***difference***, C, of two union-compatible relations A and B, denoted C = A - B, consists of those tuples that are members of A, but are not members of B. When we compute the intersection with the intersection array, we notice that t_i is *TRUE* for any tuple a_i that is in both A and B (***i.e.***, A ∩ B). We can also see that t_i is *FALSE* for any a_i that was in A, but ***not*** in B, which is precisely the condition for a_i being in the *difference*. Therefore, to form A - B, we can use the intersection array, with the modification that the tuples in the resulting relation correspond to those t_i's which are *FALSE*, instead of *TRUE*. (Alternatively, we could just put an inverter on the output line of the accumulation array.)

5. Arrays for Removal of Duplicate Tuples

The operation *remove-duplicates* transforms a multi-relation (defined in section 2.5), A, into a relation, A', which contains all of the tuples in A, except that no tuple is duplicated in A'. The systolic array used for intersection in the last section can also be used for the operation *remove-duplicates*. Instead of comparing relation A to relation B, we compare relation A to *itself*, by feeding it into both the top and bottom of the array. (Note that A is union-compatible with itself.) By doing so, we produce a matrix, T, whose elements are:

$$t_{ij} = \begin{cases} TRUE & \text{if } t_{ij}^{initial} = TRUE, \text{ and } a_{ik} = a_{jk} \text{ for all } 1 \le k \le m, \\ FALSE & \text{otherwise.} \end{cases}$$

Our strategy for eliminating duplicate tuples from A is to remove all tuples that are preceeded by another tuple that equals it. For example if tuples a_5, a_{19}, and a_{73} are all equal, then in producing A', we wish to remove a_{19} and a_{73} from A, leaving a_5 in A' (not necessarily as a'_5 because, for example, a_3 might equal a_4). In our matrix notation, the problem is then that of removing any tuple a_i, where there exists a t_{ij}=*TRUE*, for $j<i$. This is equivalent to saying that we wish to remove any tuple corresponding to a row in the matrix T which contains a "*TRUE*" in the lower triangle (left of the main diagonal). We could find the appropriate a_i by *OR*ing across each row of T, as far as (but not including) the main diagonal. Alternatively, we could set the main diagonal and the upper triangle all to *FALSE*, and then take the *OR* across the whole row. This second scheme is what we will do.

For those t_{ij} on the main diagonal and in the upper triangle ($i \le j$), we set $t_{ij}^{initial}$ to *FALSE*. This implies that t_{ij} will be *FALSE* for $i \le j$, since the comparison array works by *AND*ing each individual comparison result with the current value of t_{ij}. The accumulation processors in the *remove-duplicates* array act identically to those in the *intersection* array. They form the *OR* of each row of the matrix T. To produce A', we eliminate from A any row where the resulting t_i is *TRUE*, and keep the rest. (This is the opposite of the *intersection* operation, where we keep those rows with *TRUE* t_i).

Our *remove-duplicates* array can be used to implement the following relational operations:

Union

The union C = A ∪ B of two union-compatible relations, A and B, is the relation containing all tuples in *either* A or B, without duplicates. It is straightforward to form A ∪ B by applying the *remove-duplicates* operation to the concatenation A+B of A and B:

C = *remove-duplicates*(A + B).

In practice, this means that we first form the concatenation of A and B as we retrieve them. We then put the concatenation through both sides of the *remove-duplicates* array, and what comes out is a bit-string, indicating which tuples of the concatenation should be in the *union*.

Projection

The projection operation is similarly easy, with our *remove-duplicates* operation. We speak of the projection of a relation A over a column, or list of columns, f. (Usually, f is of the form "first column, second column, fifth column," or "name column, salary column, children column.") The projection is produced by first finding for each tuple $a_i \epsilon A$, the corresponding (smaller) tuple $a_{i,f}$, which contains only those columns from a_i that have been specified in f - this can be done conveniently during the time when the original tuples are retrieved from storage. The set A_f -- a multi-relation in general -- of the resulting smaller tuples is then transformed into a relation by removing duplicate tuples. This is precisely the function performed by our *remove-duplicates* array. (Duplicates may occur in A_f since we are taking the projection of a relation which may contain tuples that differ only in columns that are not in f.)

6. Arrays for Join

6.1 The Join Operation

We illustrate the join operation by describing a special case: the join over a single column. The more general case is sketched later in this section. The join, C, of two relations, A and B, over columns C_A and C_B, respectively, is written C = A $J_{\{C_A,C_B\}}$ B. The join, C, is the set of tuples, c_k, such that $c_k = a_i \,|_{\{C_A,C_B\}}\, b_j$, where $a_{i,C_A} = b_{j,C_B}$, for $a_i \in A$ and $b_j \in B$. (For the join to be well-defined, columns C_A and C_B must be drawn from the same underlying domain.) The operator "$|_{\{C_A,C_B\}}$" is defined to be the concatenation of its two arguments, with the exception that only one of a_{i,C_A} and b_{j,C_B} is included in the concatenation.[2]

Intuitively, we check all pairs of tuples, a_i and b_j, taken from relation A and B, respectively. Where they match in the columns specified by C_A and C_B, we concatenate the two tuples. After removing *one* of the two matching columns (to eliminate redundancy), we add the concatenation to the join, relation C.

6.2 The Join Array

We can formulate the results of a join again in terms of a matrix. Let the matrix T be defined as

$$t_{ij} = \left| \begin{array}{ll} \mathit{TRUE} & \text{if } a_{i,C_A} = b_{j,C_B} \\ & \\ \mathit{FALSE} & \text{otherwise.} \end{array} \right.$$

That is, t_{ij} *is true if and only if* a_i *and* b_j *match in the specified columns.*

If we have the matrix T, it is straightforward to generate the relation C. For each t_{ij} that has the value *TRUE* (and for only those t_{ij}), we simply retrieve a_i and b_j, and concatenate them, removing the redundant column. The size of the join, |C|, might be as large as the product |A||B|. (This happens in the degenerate case where all tuples in A match all tuples in B in the specified columns.) However, for most applications the number of *TRUE* t_{ij}'s in T is far less than this product. Therefore, we can usually generate C fast, provided we can produce T quickly. A fast way of producing T is the concern of this section.

Consider the linear array of processors in figure 6-1. We use this array to produce the

[2] Actually, authors differ as to whether the redundant column appears in the *join*. For example, Date [2] includes it, but Codd's original paper [1] omits it.

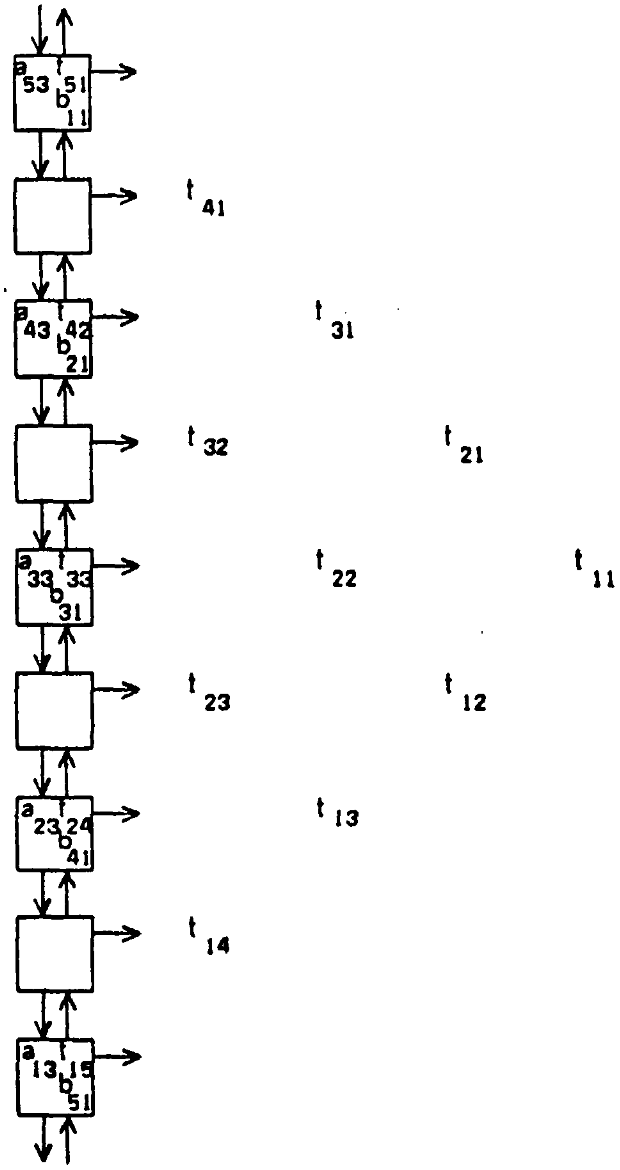

Figure 6-1: Join array.

matrix T. The column C_A of relation A (column 3 in the example in the picture) is input to the processor array from its top, and moves down. Similarly, the column C_B of B (column 1 in the example) is sent through the array from bottom to top. As the two columns "pass through" each other, each a_{i,C_A} will meet each b_{j,C_B}. (We send the columns through the array in such a way that each element follows its predecessor after two time steps so that *all* pairs of a_{i,C_A} and b_{j,C_B} meet.) When a_{i,C_A} meets b_{j,C_B}, a simple comparison suffices to determine the value of t_{ij}. These t_{ij} are collected at the right of the array. (In the figure, the t_{ij} are shown coming out from the array.) Unlike some of the operations discussed earlier, here we are interested in the t_{ij} individually, and do not perform further accumulation operations on them.

6.3 General Case

6.3.1 Join Over More Than One Column

In the general case, C_A and C_B specify more than one column. Their specifications are constrained in the following way:

- the number of columns specified by C_A must be the same as that specified by C_B, and
- the respective columns in the specifications must be based on the same underlying domains (up to a permutation, which can easily be handled).

Given this, c_k (= $a_i|_{\{C_A,C_B\}}b_j$) $\in$ C only if $a_{i,C_A} = b_{j,C_B}$, which means that tuple a_i must match tuple b_j in *all* of the columns specified by C_A and C_B. The concatenation operator "$|_{\{C_A,C_B\}}$" is defined analogously: the concatenation includes only one copy of the columns over which A and B are being joined.

The corresponding modification to the processor array in figure 6-1 is simple. Instead of having one column of processors in the array, we have several columns: one for each relational column over which A and B are to be joined. Each processor column is responsible for comparing a_i and b_j in some particular column pair, and the result t_{ij} is propagated to the right, in essentially the same way as in the *intersection* array. When they reach the right side of the processor array, the t_{ij}'s are used directly, without an intervening accumulation array.

6.3.2 Non-Equi-Join

The join operation we have been considering so far in this section is usually referred to as the *equi-join*, since the join is performed on tuples for which the values in columns C_A *equal* those in columns C_B. This notion can be generalized to allow any sort of binary comparison (e.g. ≤, >, etc.) to be done between the relevant columns of the two tuples.

The processor array to perform such an operation is easy to construct. For greater-than-join, say, processors in the array would simply perform that comparison between C_A and C_B. The particular operation to be performed might be encoded in a few bits, and passed along with the a_{ij} and t_{ij}. Or, it might be preloaded into the array of processors. This illustrates that some degree of programability can often be provided to a processor array at the expense of additional logic.

7. Arrays for Division

Division is an operation between two relations (the dividend and the divisor) which produces another relation (the quotient) as its result. The notation "C = A $\div_{\{C_A, C_B\}}$ B" means that C is the result of dividing A by B over the columns C_A of A and C_B of B.

We show how to perform the division operation by a processor array for a restricted case of division: A is a binary relation and B is a unary relation. Further, C_A and C_B specify only single columns. The extension from this to the general case is straightforward (as in the preceding section on the join).

Let the dividend A have columns A_1 and A_2 and let the divisor B have column B_1, and let A_2 and B_1 be defined on the same underlying domain (which makes their elements comparable). Then the divide operation C = A $\div_{A_2, B_1}$ B produces a quotient C, having column C_1 defined on the same domain as A_1; a value x will appear in C_1 if and only if the pair (x,y) appears in A for *every* value y appearing in B_1 [2]. An example of the division operation is shown in figure 7-1.

A	A_1	A_2
	i	a
	j	a
	k	b
	i	b
	j	c
	i	c
	k	a
	i	d
	j	b
	k	c
	k	d

÷

B	B_1
	a
	b
	c
	d

=

C	C_1
	i
	k

Figure 7-1: Example of relational division

Our systolic array for performing relational division consists of two modules: a dividend array and a divisor array. Figure 7-2 illustrates how the division array works on the example given in figure 7-1. The left-hand column of the two columns of processors in the dividend array stores (distinct) elements appearing in column A_1, one element to a processor. (These elements -- {i, j, k} for this example -- can be identified by the *remove-duplicates* array.) Similarly, elements appearing in the divisor B_1 are preloaded into each row of processors in the divisor array. In the figure, circled elements represent those elements

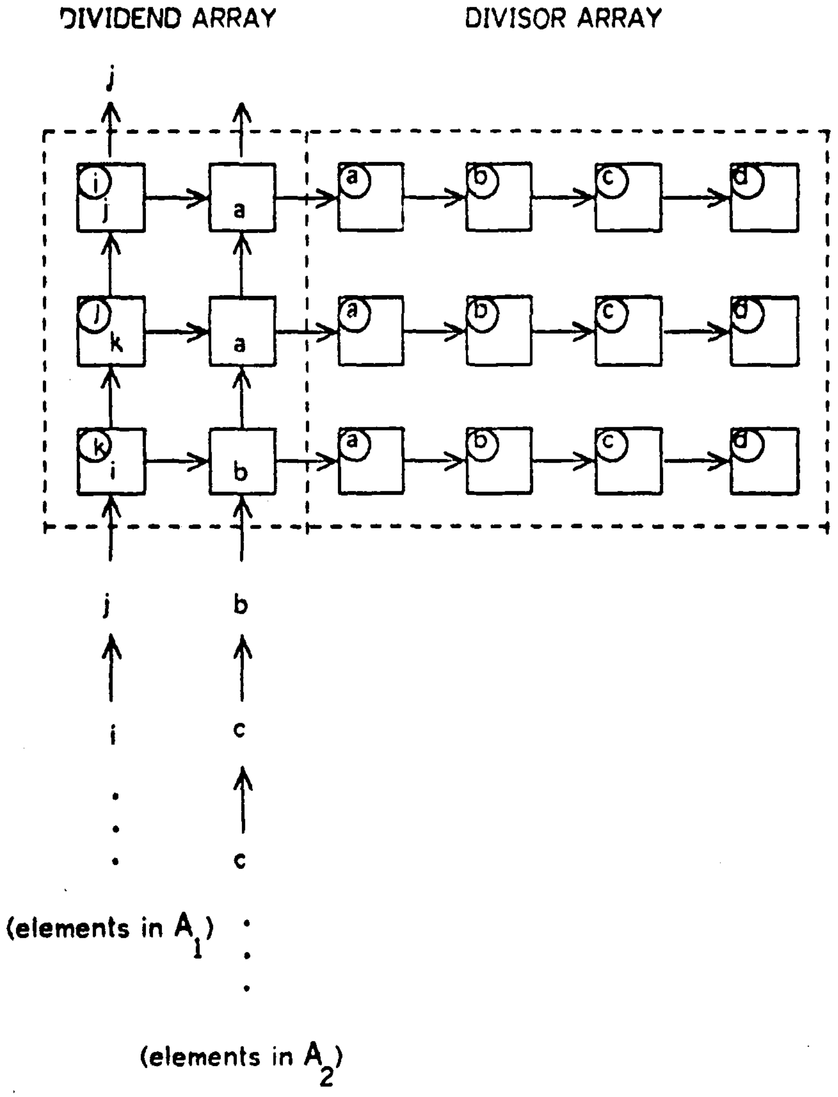

Figure 7-2: Division array (in operation).

which are stored at processors.

The dividend array computes for each element x appearing in A_1 the set of y such that $(x,y) \in A$. It works as follows. We take each pair $(z,y) \in A$, and pass it into the dividend array from the bottom; the z into the left column and y into the right column. At each time step, the z will be in the same processor as some preloaded element x, and the y will be following one step behind it, in the column to the right. We compare z to x, and if they match, we output a *TRUE* from the right side of the processor; otherwise, we produce a *FALSE*. This boolean value t arrives at the processor in the right column, just as the associated y arrives there. If t is true, then y is output from the right side of the processor. Otherwise, some null value is output.

Thus for each x appearing in A_1, the non-null values, output from the dividend array at the row whose left processor has x stored, are those y's such that $(x,y)\epsilon A$. We see that if these y's include all the elements in B_1, then x belongs to C_1. This is checked by the corresponding row of processors, in the divisor array, which takes the y's as inputs. More precisely, each processor of the row checks if the element it is storing matches any of the y's passing from left to right along the row. If every processor of the row finds at least one such match (which is checked by doing an *AND* across the row after the dividend passes through the array), then the y's contain a, b, c, and d, and thus x belongs to C_1. This is the essential idea behind the division array. One can already see that the division array provides the same kind of rapid computations (using simple and regular structures) as other arrays discussed earlier.

8. Remarks on Implementation and Performance

During the past year, we have designed prototypes of several special-purpose chips at CMU. These include a pattern-match chip [3], an image-processing chip [6], and a tree processor for database applications [9]. The pattern-match chip can be viewed as a scaled-down version of the comparison array in Section 3. (This chip has been fabricated, tested, and found to work.) The following comments and projections are based partly on our experience with the pattern-match chip.

In some of the schemes presented in this paper, it is the case that only half of the processors in a systolic array are busy at any one time. This inefficiency can be avoided in the following implementation: rather than marching two relations against each other along the systolic array, we let only one relation move while the other remains fixed. Also, for simplicity, we have so far assumed that processors in systolic arrays operate on words. In implementation, each word processor can be partitioned into bit processors to achieve modularity at the bit-level. A transformation of a design from word-level to bit-level is demonstrated in [3]. In general, many variations on the systolic arrays suggested are possible. All of these are equivalent, and differ only in implementation details.

Below, we give figures for a reasonable array size for implementation. While such an array would be large enough for many applications, it is also possible to use the array to solve problems that will not fit entirely on it. This calls for the technique of decomposing problems. The technique is best illustrated by a simple example. In the intersection problem, consider the matrix, T, of results. For a large problem, one can simply partition this matrix into sub-problems small enough to fit on the array; each of these sub-problems would generate a piece of the matrix.

Intersection is one of the most computationally demanding relational operations, since it requires *full* tuple comparisons between *all possible* pairs of tuples. We examine the speed with which systolic arrays can perform intersection.

We make the following assumptions concerning the size of a typical relation:

- A tuple is of size 1500 bits (or about 200 characters).
- A relation is of size 10^4 tuples.

The following (conservative) estimates are typical of results that have been achieved with present NMOS technology:

- A bit-comparator, the fundamental workhorse unit of our arrays, is about 240µ x 150µ in area. The comparison is performed (very conservatively!) in about 350ns, including time for on-chip and off-chip data transfer.

- With present technology, chips are about 6000μ x 6000μ in area. Division gives us about 1000 bit-comparators per chip. (Notice that this calculation is realistic only if the design is repetitively regular, which is the case for our systolic arrays.) We can assume that none of the comparators on a chip incurs delay due to pin limitations; since the time for a comparison is large relative to off-chip transfer time ($\leq$30*ns*), we can multiplex about 10 bits on a pin during a single comparison.

- It is practical to construct devices involving a few thousand chips. We assume 1000 chips. This gives us the capability of performing 10^6 comparisons in parallel.

Based on these assumptions, we can make the following performance predictions for intersection. The intersection requires a total of 1.5×10^{11} bit comparisons, since we need 1500 bit-comparisons for each of the $(10^4)^2$ tuple comparisons. The time to perform intersection, therefore, is:

$(1.5 \times 10^{11}$comparisons) x (350*ns* / 10^6comparisons),

which is about 50*ms*. We believe that this estimate is extremely conservative, even with existing technology. If we assume instead, for example, 200*ns*/comparison, and 3000 chips, we derive a figure of about 10*ms*.

The processing speed obtainable from these systolic arrays can keep up with the data rate achievable with the fast mass storage devices available in present technology. For example, a moving-head disk rotates at about 3600 r.p.m., or about once every 17*ms*. Assume that we can read an entire cylinder in one revolution, as in some of the proposed database machines (for a survey of these machines, see [4]). This is a rate of about 500,000 bytes in 17*ms*. In a comparable period of time, our systolic array can process (for example, can intersect) two relations, each of about 2 million bytes.

9. Remarks on the Organization of an Integrated Systolic System

Systolic arrays introduced in preceeding sections are capable of rapid processing of individual relational database operations. To process all of the operations required in a single transaction or a set of transactions, an integrated system containing several systolic arrays is needed. Many strategies are possible for the interconnection of the systolic devices. To decide which interconnection strategy to choose, one must consider the system requirements:

- High capacity for data transfer. As described in the last section, it is feasible that a systolic array may process hundreds of thousands of bytes per millisecond.

- Flexibility and generality. The execution order of systolic devices varies greatly from one transaction to another transaction. Relations may have to be decomposed to fit the (fixed) sizes of systolic arrays. Results from subrelations must be stored outside the systolic arrays before they are finally combined.

One organization that seems to match the system requirements is the crossbar switch interconnection depicted in Figure 9-1. Typically, the system works as follows. Initially, the relevant relations are read from disks into memories. (Disks with "logic-per-track" capabilities [8] can of course be incorporated into the system, so that some simple queries never have to be processed outside the disks.) Then the crossbar switch is configured so that the relevant memories are connected to the systolic array that will perform the first operation of the transaction in question. The data is pipelined from the memories through the switch and through the processor array. The output of the array is pipelined back into another memory. This is repeated for each relational operation in the transaction. Due to the crossbar structure, several operations may be run concurrently. The final results are eventually returned to the disk (or a user's terminal, or printer, etc.) from the memory in which they reside.

In the future, we plan to perform a detailed analysis of the crossbar scheme and a comparison of this scheme with other alternative structures. For example, Song [9] has suggested the use of a tree machine for database applications. The leaf nodes of the tree machine are responsible for data storage, and for a limited amount of processing of the data. The tree structure itself is used to broadcast instructions and data, and to combine results of low-level computations on the data. This same tree machine is capable of performing all database operations. A detailed comparison of these and other database machine structures is needed in order to understand their relative merits.

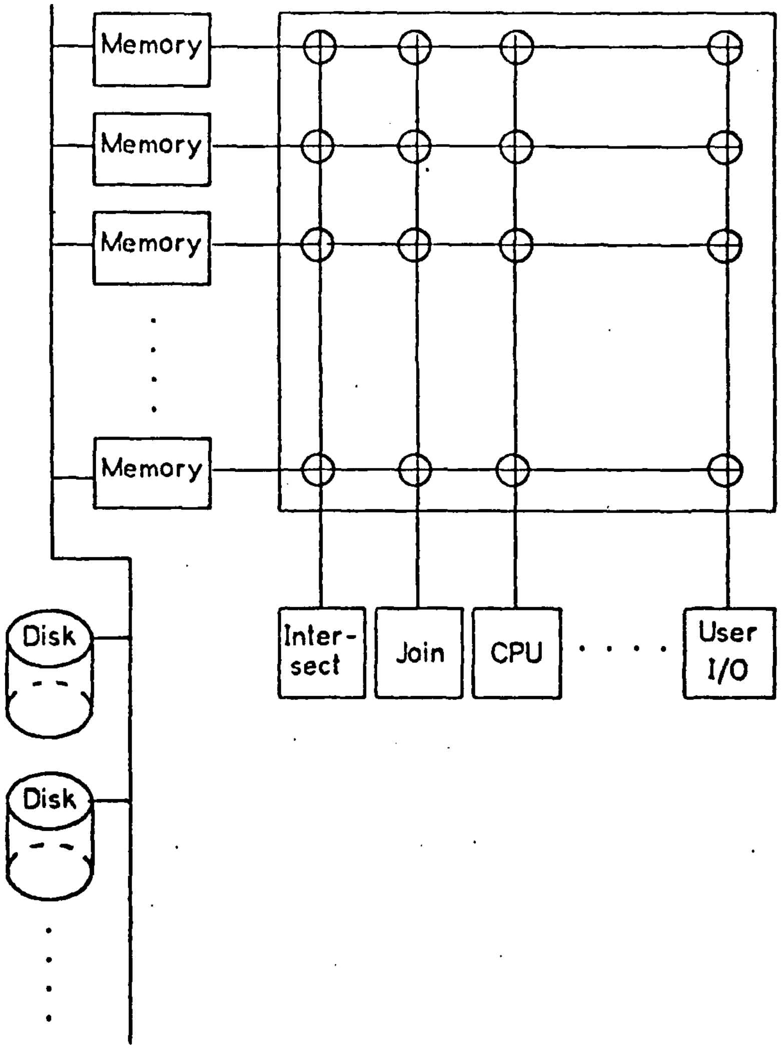

Figure 9-1: Systolic database system using crossbar switch.

References

[1] Codd, E.F.
A Relational Model of Data for Large Shared Data Banks.

Communications of the ACM 13(6):377-387, June, 1970.

[2] Date, C.J.
An Introduction to Database Systems.
Addison-Wesley, Reading, Mass., 1977.

[3] Foster, M. J. and Kung, H.T.
The Design of Special-Purpose VLSI Chips.
Computer Magazine 13(1):26-40, January, 1980.
An early version of the paper, entitled "Design of Special-Purpose VLSI Chips: Example and Opinions", is to appear in *Proceedings of the 7th International Symposium on Computer Architecture,* La Baule, France, May 1980.

[4] Hsiao, D.K.
Database Computers.
In Yovits, M.C., editor, *Advances in Computers, Vol. 19.* Academic Press, New York, 1980.
To appear.

[5] Kung, H.T. and Leiserson, C.E.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W., editor, *Sparse Matrix Proceedings 1978,* pages 256-282. Society for Industrial and Applied Mathematics, 1979.
A slightly different version appears in *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Section 8.3.

[6] Kung, H.T. and Song, S.W.
A Systolic Array Chip for the Convolution Operator in image processing.
Technical Report, VLSI Document V046, Carnegie-Mellon University, Department of Computer Science, 1980.

[7] Kung, H.T.
Let's Design Algorithms for VLSI Systems.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication,* pages 65-90. California Institute of Technology, January, 1979.
Also available as a CMU Computer Science Department technical report, September 1979.

[8] Slotnick, D.L.
Logic per Track Devices.
In Tou, J., editor, *Advances in Computers, Vol. 10,* pages 291-296. Academic Press, New York, 1970.

[9] Song, S.W.
A Highly Concurrent Tree Machine for Database Applications.
Internal working document, Carnegie-Mellon University, Department of Computer Science, March 1980.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-80-114	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SYSTOLIC (VLSI) ARRAYS FOR RELATIONAL DATABASE OPERATIONS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) H.T. KUNG & PHILIP L. LEHMAN		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE October 1979 13. NUMBER OF PAGES 32
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)