TR-82-002

A Hardware Architecture for Computing Relational Joins and Projections

By

Yang-Chang Hong

Institute of Information Science

Academia Sinica, Taipei, R.O.C.

This work was supported by National Science Council Grant NSC70-0404-E001-02

February 1982

ABSTRACT: A hardware architecture is described which provides powerful join and projection capabilities to currently proposed relational associative database machines (RADMs). The main feature of the hardware is an RAM, consisting of bit- and word-addressable stores, which can rapidly remember or recall data. This data might be join column values/tuples selected from one relation, in which case the RAM helps on performing the joining of these values/tuples with the tuples in the second relation. Alternatively, the RAM can help eliminate repetitive tuples within the columns selected. The hardware uses a memory bank and an array of queue servers for the joining of two relations in which their join columns do not satisfy the referencial integrity constraint. Great efficiency of this architecture is achieved by much parallelism in the cross referencing, giving a considerable performance improvement over existing RADMs.

Content

1. Introduction

The relational model [5] has, more than any other data models during the past decade, attracted and held great interest of the database researchers and database management community. Its tabular representation of data is very suitable to ordinary users and it provides a high-level, nonprocedural data language for users to interact with the database and satisfies the requirements of data independence. The model is structurally and behaviorably far from the storage organization and primitive operators of conventional computer hardware and, thereby, is very difficult to be efficiently implemented by conventional hardware.

Several approaches to implementing a relational database by means of specialized hardware have been proposed [2,3,4,6,7,8,9,10,11,14,15,17]. Currently, the design of direct hardware support for the joining of relations (or files) has been to concentrate on a form so called the "implicit" join [1,3,11, 15]. This join does not create a derived relation; instead the values of the columns being joined, called the join columns, from the selected tuples in one relation are transferred to select tuples in the second/same relation that have the same those values in their join columns. The algorithms designed for explicit joins (as opposed to implicit joins), in which more data other than those join column values in one relation are needed for computing the join with the second relation, are mainly carried out by the host computer [1,9,10]. The LEECH and CAFS machines use a filter for selecting tuples needed for the join. The selected tuples are sent to the host to form the concatenated tuples of the join. The only difference of these two machines is the design of the filter. RARES prevodes a hardware-support algorithm for dividing the tuples of the relations being joined into buckets according to different value-intervals of their join columns. The tuples within each bucket are sorted in the main memory and the sorted buckets are used for computing the join by the host. They will not be very effective if the number of tuples being joined is large. The work by Tanaka et.al[16] proposed a totally hardware-support join algorithm based on pipeline searching and sorting engines. One disadvantage of this work is that the algorithm requires a considerable amount of logic to be implemented.

Except CAFS[1], little is said about how to implement the projection operation in hardware. CAFS uses a pre-compiled index for each combination of columns for projection and each index value is associated with one bit position in the single bit array store. For a relation that has n columns, 2^n pre-compiled indices are generally needed for supporting all the possible projections on this relation. An alternative way of projection proposed by CAFS is to use a set of functions to hash concatenated values of columns to be projected to a value corresponding to one bit position in the store. This method might lose information because different values may not be able to be distinguished by the set of hashing functions provided.

Most existing relatinal associative database machines (RADMs), e.g., CASSM, RAP, RARES, DBC, etc., are based on the parallel processing of the segmented sequential search and cannot efficiently support join and projection operations by means of their parallelism. This is because (possibly) a great amount of cross checking is involved in this type of operation, which breaks the parallelism. They, thus, are not alone sufficient to make a high-performance database machine. New hardware which can perform a large amount of cross checking in parallel has to be sought to cope with a join- and project-dominating database application.

This paper describes a hardware architecture which can provide powerful join and projection capabilities to currently proposed RADMs. The main feature of the hardware is an RAM extending the concept of single bit array stores, as suggested by CASSM and CAFS, for remembering or recalling data needed for joins. Through this extention, the queries demanding the joining of relations which satisfy the referencial integrity constraint [5] can be efficiently executed.

Our observation reveals that a majority of joins are via the columns which have the referencial integrity property. This means the extention of single bit array stores to an RAM is essential. For those joins in which the join columns do not satisfy this integrity rule, we suggest a memory bank for storing tuples of one relation being joined and an array of queue servers for performing the concatenated tuples of the join, in parallel, from the bank and the array of queues associated. Alternatively, the hardware can be used to eliminate any repetitive tuples within the columns being projected. It will be seen that this hardware design emphasizes on much parallelism in the cross referencing, giving a considerable performance improvement over existing RADMs.

The body of the paper is divided into three parts. In the first part the hardware architecture is described. The second part is concerned with the algorithms for computing the joins. The third part is concerned with the multiple column projection algorithms. This is followed by a summary and further research.

2. Hardware

The hardware (see Figure 1) described here provides join and projection capabilities to an RADM. It accepts a sequence of column values/tuples from the RADM where data are searched in parallel by the search logic. The command and control processor (CCP) receives data requests from the host computer; it translates them into commands for the RADM and the hardware, distributes commands for execution, receives the data transferred out of the RADM and the hardware, and outputs the data to the host computer. We will assume that data stored in RADM are in coded form and the encoding and decoding process are done by CCP.

The hardware consists of five major components — IP, MB, RAM, S, and CP — as shown in Figure 1 where

(1) IP is an input processor which accepts column values/tuples from RADM and stores them in queue Q. The queue Q acts as a buffer between the RADM and

the hardware. Associated with Q are two registers T and H and one flag F_Q. The T- and H-registers are used to hold the locations of the last and first entries of the queue. The setting of the flag F_Q indicates the queue Q is full. This will also notice the RADM to stop outputting data to the hardware. The IP will start its processing once the flag F_Q is clear.

(2) MB is a memory bank for holding the tuples of relations being joined or projected. It consists of p memory modules M(i), $1 \leqq i \leqq p$, and each module has q words, where p and q are design parameters.

(3) The RAM is composed of one or several single bit, directly addressed stores rA, rB,..., and an array r of words. It is used to hold intermediate results. The single bit array store is addressed by encoded column values of the database. The array r of words is also addressed by encoded values and can hold encoded values, counters, and pointers (to be detailed later). If a pointer word is concerned, it can point to a specific word of a specific memory module. Such a word can be regarded as consisting of two parts : one storing a value pointing to a particular memory module and the other storing a value pointing to a particular word within the module. Thus, the array r of words can be seen as consisting of two arrays r' and r" of words and the sizes of the word in r' and r" are $\lceil \log_2 p \rceil$ and $\lceil \log_2 q \rceil$, respectively, where $\lceil x \rceil$ is the least integer greater than or equal to x.

(4) S is a set of queue servers S_i each associated with a queue Q_i. The Q_i's are served to hold incoming tuples of the second relation being joined. Like queue Q, each Q_i has two registers T_i and H_i and one flag F_i. Each queue sever S_i is designed to read data from its queue Q_i and the memory module M(i). Thus, there are as many S_i's as M(i)'s. A buffer is provided for each S_i for holding the results produced which are either output to the host computer or stored back to the RADM for further processing. The data transfer is accom-

plished by an output mechanism.

(5) CP is a central processor which fetches column values/tuples from Q and uses them as indices to address the bit array store for setting to 1 or 0 or testing for being 1 or 0, or to locate desired words in r for various purposes (to be detailed later). The CP also serves to allocate storage space in MB for storing tuples of the relation being joined or projected. The registers T, D, and BR(i), $1 \leqq i \leqq P$, are provided for storage space allocation. They are best explained when used.

3. Implementation of Relational Joins

This section shows how the hardware performs relational joins. We will first consider the implementation of implicit joins and explicit joins in which their join columns satisfy the referencial integrity rule and then consider those explicit joins in which their join columns do not satisfy this integrity rule.

3.1 Queries Involving the Implicit Joins of Relations

Implementing an implicit join by the single bit array stores is best explained by means of an example.

Example 1.

Print all the green items sold by the D1 department.

To answer this query, a simplified database with tables SALES and TYPE is assumed in Figure 2. This query can be implemented by various ways. One way is to apply the selection process to the table SALES to select the items sold by the D1 department. The selected items are then transferred to the table TYPE as a disjunctive condition to retrieve all the green items. The procedure can be implemented by using the store rA, which is outlined below :

(1) Clear the single bit array store rA.

(2) Scan the table SALES by RADM and output the items sold by the D1 department to the input processor IP. The items fed to IP are then queued in Q, which

will be used as indices to address the store rA and then recorded in rA by CP.

(3) Scan the table TYPE by RADM and output all the green items and store them in Q. Any item in Q is output to the host computer if it has been recorded in rA; i.e., it is an item sold by the D1 department.

Here we assume that the reader is familiar with the data search performed by the RADM. What is not made clear is the function of the single bit array store rA; how the CP records the items in rA and how it determines which green items are to be output to the host.

Recall that data are encodedly stored in the RADM. Each bit position in the array can be made corresponding to an encoded value. With this technique, the addressed bit can be set to 1 or 0, or tested for being 1 or 0. We give a real example to illustrate this technique.

Assume that BOLT is encoded as 0, i.e., <BOLT> = 0, and <CAM> = 1, <COG> = 2, <GEAR> = 3, <NUT> = 4, and <SCREW> = 5. At the end of step (2), the bit pattern of rA will be (0,1,0,1,0,..., 0). This pattern would record the list of items CAM and GEAR. In step (3), items <BOLT> and <GEAR> are selected and stored in Q for examination. Since rA(<BOLT>) = rA(0) = 0, <BOLT> is discarded. Similarly, rA(<GEAR>) = rA(3) = 1, <GEAR> is output to the host computer. Before <GEAR> is output, it is decoded by the encoding and decoding unit (EDU) in CCP. Of course, values D1 and GREEN in the query have to be encoded by EDU before the query is executed. (We neglect the detailed encoding and decoding processes here.)

The discussion above assumes that all the encoded ITEM values are within the address space of rA. If not, they are divided into buckets; the values in the first bucket lie between 0 and $2^t - 1$; the values in the second bucket lie between 2^t and $2^{t+1} - 1$; and so forth, where t is the number of bits required in the address space. Each bucket is then evaluated by repeatedly applying the same procedure being described.

The idea of using the single bit array store to remember or recall data is the same as those used in CASSM and CAFS. CASSM uses a single bit array store per cell (consisting a memory element and a processing logic) and one logical single bit array store, consisting of the concatenation of single bit array stores of cells, addressable by each processing logic. To address a bit in the logical array store requires passing the bit address (i.e. the encoded value) from one logic to another. Moreover, only one cell is allowed to address the logical array store at one time. If two cells want to address the store simultaneously, one of the two cells must wait for the subsequent revolution. This means additional memory revolutions are required in addressing the bit array store. Our approach, like CAFS, uses a central processor CP to set or test a single bit array store, thereby eliminating memory addressing contention. Because of the use of an RADM, which acts as a filter, less data than CAFS are fed to the CP for setting or testing the bit array store.

When the values selected from the second relatin are to be transferred to select tuples in the third relation that have the same those values in their join columns, in this case two single bit array stores are needed. In general, two stores are sufficient and can be alternatively used for a query involving a chain of implicit joins.

3.2 Explicit Joins With the RAM

By extending the concept of single bit array stores to an RAM, some type of explicit join can be implemented as effectively as the implicit join. The idea is that more data other than these join column values can be kept in the RAM for computing that type of join. Two examples below are used to illustrate this point.

Example 2.

Find the names of the employees who make more than their department managers.

The query is directed at the table

EMPLOYEE(NAME,SALARY,DEPT,MGR)

where the managers are also employees — i.e., the values in the MGR column also appear in the NAME column.

One way to answer this query is first to scan the MGR column and output unique managers. Next select EMPLOYEE tuples where NAME = 'one of the selected managers' and then join the tuples being selected with EMPLOYEE tuples that have the same those names in their MGR column. Finally, scan the joined relation and output the employee names whose salaries are greater than their managers. This method performs an implicit join followed an explicit join and a selection operation. It is insufficient to use single bit array stores to remember the manager names and their salaries for being used to select those employee names who make more than their managers. Our approach uses the single bit array store rA and the array r of words for storing the manager names and their salaries, respectively, which is outlined below :

(1) Clear the single bit array store rA.

(2) Scan the EMPLOYEE table by RADM and output the entires in the MGR column and store them in Q. The entries stored in Q are then fetched and used to set the single bit array store rA by CP.

(3) Scan the EMPLOYEE table again and output the employee names and their salaries and store them in Q. The pairs (<name>,<salary>) stored in Q are then fetched, one by one,to test if rA(<name>) is 1 or not. If rA(<name>) = 1, then the <salary> is stored in the corresponding word in r, i.e., r(<name>) ← <salary>. Otherwise, discard the pair.

(4) Scan the EMPLOYEE table again and output the employee names, salaries, and managers and store them in Q. Each triple (<name>,<salary>,<manager>) stored in Q are then fetched to test if r(<manager>) < <salary> or not. If yes, output the <name>. Otherwise, discard the triple being held.

We notice that the encoded values in the SALARY column should have the same order as they originally have. This procedure combines one explicit join and one selection operation to a single process where the manager names and their salaries are recorded in RAM and each incoming EMPLOYEE tuple is virtually concatenated to a proper entry in RAM so that the qualified employee names can be determined immediately. Our observation concludes that this technique can generally be applied to the explicit joining of any two relations in which their join columns satisfy the referencial integrity constraint. This rule is rewritten bellow:

Let D be a primary domain, and let R1 be a relation with a column A that is defined on D. Then, at any given time, each value of A in R1 must be either (a) null, or (b) equal to V, say, where V is the primary key value of some tuple in some relation R2 (R1 and R2 not necessarily distinct) with primary key defined on D.

The example 2 illustrates the case where R and R2 are not distinct. The example 3 bellow is the case where R1 and R2 are distinct.

Example 3.

Join the tuples of the SALES table with those TYPE tuples having items whose price is greater than 4P and output the DEPT, ITEM, and COLOR columns. This is a typical explicit join of two relations. The column ITEM in table SALES is sometimes called the foreign key. This join can be realized by the following procedure:

(1) Clear rA.

(2) Scan the TYPE table and output the items and their color if their price is greater than 4P and store them in Q. Each pair (<item>, <color>) stored in Q is fetched and recorded in RAM. That is, $rA(<item>) \leftarrow 1$ and $r(<item>) \leftarrow <color>$.

(3) Scan the SALES table and output the SALES tuples to Q. Each tuple (<department>, <item>) stored in Q is fetched to test if rA(<item>) = 1 or not. If yes, read r(<item>) and concatenate it to the tuple being held and output the new tuple (<department>, <item>, <color>) to the host computer. Otherwise, discard the tuple. If further processing is needed each new tuple is stored back to the RADM.

If the join of SALES and TYPE is considered, it can at least be implemented by two ways. The first way divides this join into two subjoins: one is to join SALES and TYPE$^{(1)}$(ITEM, COLOR) over ITEM, denoting the resulting table as R1, and the other is to join R1 and TUPE$^{(2)}$(ITEM, PRICE) over ITEM. Each subjoin follows the same procedure descirbed above. The second way is to treat each word in the array r as a pointer word containing a value pointing to the starting address of a block of words in MB in which a TYPE tuple, except for its identifier, is stored. This approach would modify the step(2) of the above procedure as

(2') Output TYPE tuples and store them in Q. Read each tuple and store it, except for its identifier t-id, say, in a block of words in MB. The starting address of the block is then stored in r(<t-id>) and the tuple identifier is recorded in rA(<t-id>). The addresses recorded in r are provided for step(3) for locating TYPE tuples.

It is quite obvious that the latter approach is generally better than the former one. Note, however, that not all the join columns satisfy the referencial integrity constraint. We refer to the explicit join shown in Example 2 or 3 as the Type I explicit join and otherwise the type II explicit join.

3.3 Implementation of Type II Explicit Joins

3.3.1 General Description

An explicit join of two relations in which the join columns do not satisfy the referencial integrity constraint may make the implementation rather costly in time and storage. To reduce the cost in storage, "large" relations being joined are divided into buckets, according to their join column values in such a way that the first bucket of the first relation is to join with the first bucket of the second relation; the second bucket of the first relation is to join with the second bucket of the second relation; and so forth. A relation is "large" if it satisfies one of the following conditions : (1) The range of the encoded values of the join column is over the size of the address space of RAM and (2) It cannot be fitly stored in the memory bank MB. The processing time is decreased by increasing the parallelism of computing the concatenated tuples of the join of buckets.

This parallelism is achieved by further dividing the tuples of buckets being joined into sub-buckets. The first sub-bucket of the ith bucket is then joined with the first sub-bucket of the i'th bucket; the second sub-bucket of the ith bucket is joined with the second sub-bucket of the i'th bucket; and so forth, where ith and i'th buckets have the same value-interval. The join of the pairs of sub-buckets is done in parallel by the array of servers in our approach. It will be seen that, in logical effect, the join is implemented as joining the tuples from two "sorted" sub-buckets.

In our design, the division of the large relations being joined into buckets is relegated to the RADM, similar to RARES [9], so that fewer tuples are output to the hardware. The pairs of buckets are then sent to the hardware, one by one, for computing the join. In computing the join of two buckets, the hardware uses two single bit array stores rA and rB for filtering out the irrelevant tuples of the join since the join column values in one bucket may not appear in another. We will see that it is worthy of doing so, especially when a large number of irrelevant tuples are involved. The array r of words, except for helping with Type I explicit joins on remembering or recalling data as described previously, can also help with type II explicit joins on dividing tuples of each bucket into sub-buckets. For two buckets being joined, the sub-buckets of one bucket are first stored in the memory modules M(i) of the memory bank, one per module. Each incoming tuple of the second bucket is then stored in the corresponding queue Q_i; that is, the first queue Q_1 accepts only those incoming tuples whose join columns have the same value-interval as those stored in M(1); the Q_2 accepts only those incoming tuples whose join columns have the same value-interval as those stored in M(2); and so forth. This arrange permits each queue server S_i $(1 \leqq i \leqq p)$ to produce the concatenated tuples of the join from its queue Q_i and the M(i) in parallel, without any memory addressing contention. What is not made clear here is how each S_i can know which tuples in M(i) are concatenated to the tuple being fetched from Q_i. This can be seen from the following algorithm.

3.3.2 Algorithm for Explicit Equi-Joins of Two Buckets

Let us denote the two buckets being joined as R_A and R_B of relations A and B with $A = (X_1, X_2, \ldots, X_u)$ and $B = (Y_1, Y_2, \ldots, Y_v)$, respectively, where $X_i (1 \leqq i \leqq u)$ and $Y_j (1 \leqq j \leqq v)$ are column names. Assume that columns X_a and Y_b are of the same underlying domain. Compute the join of buckets R_A and R_B over $(X_a = Y_b)$. The resulting table consists of the set of tuples t, where t is the concatenation of a tuple t' belong to R_A and a tuple t" belong to R_B and $x_a = y_b$ (x_a being the X_a-component of R_A and y_b being the Y_b-component of R_B). The algorithm for the join is outlined below :

(1) Initialization : Clear rA, rB, and r.

(2) Output X_a-components of R_A and set the rA and increment the corresponding counter words of r : Clear Registers T and H and the flag F_Q of Q. Scan the relation A by RADM and output the sequence of X_a-components x_a's of R_A (in encoded form) to IP. The IP accepts each component x_a and deposits it into Q. The x_a's in Q are then fetched, one at a time, by CP and used as indices to address the bits in rA and the corresponding counter words in r. The addressed bits of rA are set and the corresponding counter words are incremented by one. For example, if x_{ai} is fetched, then $rA(x_{ai}) \leftarrow 1$, and $r(x_{ai}) \leftarrow r(x_{ai}) + 1$. At the end of step(2), the word $r(x_{ai})$ contains a value indicating the number of R_A tuples with the component X_{ai} in their join columns. The discussions which follow use $[r(x)]$ to denote the contents or value of the word in r addressed by the component x.

(3) Output Y_b-components in R_B, set the rB, and allocate memory space in MB for R_A : Clear registers T and H and the flag F_Q of Q and BR(i), $1 \leqq i \leqq p$, and $D \leftarrow 1$. Scan the relation B by RADM and output the sequence of Y_b-components y_b's of R_B to IP. The IP accepts each y_b and deposits it into Q. The y_b's in Q are then fetched by CP and used to test if the corresponding bits in

rA and rB are set or not.

Cases : (i) if $rA(y_{bj}) = 0$, i.e., the y_{bj} does not appear in the join column of R_A, then ignore the component y_{bj}.

(ii) if $rA(y_{bj}) = 1$ and $rB(y_{bj}) = 0$, i.e., the y_{bj} is first encountered, then $rB(y_{bj}) \leftarrow 1$ and allocate memory space in MB for storing R_A tuples with $X_a = y_{bj}$.

The setting of $rB(y_{bj})$ will prevent the subsequent incoming $y_b = y_{bj}$ from re-allocating memory space in MB for those R_A tuples having $X_a = y_{bj}$. The memory allocation is done as follows : (Initially, registers $BR(k)$, $1 \leqq k \leqq p$, are cleared and $D \leftarrow 1$, i.e., each $BR(k)$ points to the starting address of k-th module and D points to the first memory module.)

(a) $T \leftarrow r(y_{bj})$, i.e., the value of word $r(y_{bj})$ is saved in T-register, which is a temporary register.

(b) $r'(y_{bj}) \leftarrow D$ and $r''(y_{bj}) \leftarrow BR([D])$, where $[D]$, the contents of D-register, is used to index one of $BR(k)$, $1 \leqq k \leqq p$.

(Remember that r may be regarded as consisting of r' and r".)

(c) $BR([D]) \leftarrow BR([D]) + (T + 1)$ and $D \leftarrow (D + 1)$ module p, and if $D = 0$, $D \leftarrow P$. The former statement denotes that the current module will be allocated following the logical location $BR([D]) + T + 1$ if it is to be allocated again. We add one extra logical word for each allocation (to be described in next step.) The later one indicates that the next allocation will be assigned to the module next to the current one.

The above three statements allocate a block of $(T+1)$ logical words (each can hold a tuple) in the module specified by D-register (before updating) for storing R_A tuples having the join column value equal to y_{bj}. For the case in which $rA(y_{bj}) = 1$ and $rB(y_{bj}) = 1$, this means that block allcation for R_A tuples with $X_a = y_{bj}$ has been done.

(4) Output R_A tuples and store the relevant tuples of the join in the

allocated memory : Clear registers T and H and the flag F_Q of Q. Scan the relation A by RADM and output the sequence of R_A tuples and store in Q. The tuples stored in Q are then fetched by CP. The join column values x_a's of each fetched tuple are extracted and used as indices to address the corresponding bits in rB. The contents of each addressed bit are tested for being set or not.

Cases : (i) if $rB(x_{ai}) = 0$, i.e., the R_A tuple being processed by CP is irrelevant to the join since the join column value x_{ai} of the tuple does not appear in the join column of R_B, then ignore the tuple.

(ii) if $rB(x_{ai}) = 1$, then

1. if $rA(x_{ai}) = 1$, then $rA(x_{ai}) \leftarrow 0$ and $MB([r(x_{ai})]) = MB([r'(x_{ai})]).[r''(x_{ai})]) \leftarrow 1$, where the part $r'(x_{ai})$ points to a particular memory module and $r''(x_{ai})$ points to a particular logical word in the module specified.
2. $T \leftarrow r(x_{ai}) + MB([r(x_{ai})])$ and $MB([T]) \leftarrow$ the tuple being held by CP, and $MB([r(x_{ai})]) \leftarrow MB([r(x_{ai})]) + 1$.

From (ii), we know that each logical word in MB pointed by the contents of the word $r(x_{ai})$ is a word containing a value indicating the number of R_A tuples with $X_a = x_{ai}$ that have been stored. At the end of this step, the word will contain a value one larger than the number of R_A tuples with $X_a = x_{ai}$. This information is important to each server S_i where new tuples are formed.

(5) Output R_B tuples, deposit the relevant R_B tuples of the join into the proper queues Q_i, and produce the concatenated tuples of the join : Clear T and H registers and the flag F_Q of Q, and T_i and H_i registers and the F_i flag for $1 \leqq i \leqq p$. Scan the R_B tuples by RADM and output them to Q. The tuples in Q are fetched and their join column values y_b's are extracted by CP. The extracted y_b's are used to test if the corresponding bits in rB are set or not.

Cases : (i) if $rB(y_{bi}) = 0$, i.e., the tuple being held is irrelevant to the join, ignore the tuple.

(ii) if $rB(y_{bj}) = 1$, then fetch the $r(y_{bj})$ consisting of two fields $r'(y_{bj})$ and $r''(y_{bj})$, and deposit the concatenated R_B tuple, consisting of $r''(y_{bj})$ and the R_B tuple being held, into the queue specified by the word $r'(y_{bj})$. The $r''(y_{bj})$ holds an address pointing to the starting address of a block of R_A tuples to which the R_B tuple will be concatenated.

Each server S_i will start its joining of tuples from the Q_i and the corresponding module M(i) once Q_i is not empty (i.e., the contents of registers H_i and T_i in Q_i are not equal). After the join of two buckets is completed, the join of next bucket-pair follows and so forth, until all the bucket-pairs have been processed. Logically, we can say that each S_i produces the concatenated tuples of the join from two "sorted" such buckets. Teh concatenated tuples of the join in each buffer will be either output to the host or stored back to the RADM for further processing.

So far, only a single join column is involved in the join operation. For joins of relations on multiple columns, the addressing bits or words of RAM using single encoded values have to be modified. One way is to associate the multiple join column values with a pre-compiled index, as suggested by E. Babb [1], whenever such a join is concerned. Another way is to dynamically encode each multiple join column value with a unique value.

3.3.3 An Illustration Example

This section shows how the above algorithm works by means of an example. Consider the same database as given in Figure 2. As an example, we consider the equi-join of table SALES on column ITEM with table TYPE on column ITEM, though this join can be computed without creating a derived relation. We will follow the steps of the above algorithm to illustrate how this join is computed.

Step 1. Clear RAM, i.e., rA, rB, and r.

Step 2. Clear registers T and H and the flag F_Q of Q. Assume that the sequence of ITEM-components of table SALES that are input to Q is <CAM>,<GEAR>, <CAM>,<NUT>,<CAM>, and <NUT>, as they appear in the SALES table of Figure 2. These components will be used as indexes to set the rA and update the corresponding counter words of r. At the end of this step, the bit pattern of rA will be (0,1,0,1,1,0,..,0) and their corresponding values of the array of words r will be (0,3,0,1,2,0,...,0)(Figure 3(a)) — i.e., there are three SALES tuples with ITEM-component = <CAM>, one SALES tuple with ITEM-component = <GEAR>, and two SALES tuples with ITEM-component = <NUT>.

Step 3. Clear T, H, and F_Q in Q, and BR(i), $1 \leq i \leq p$, and set D to 1. Assume that the sequence of ITEM-components of table TYPE input to Q is <BOLT>, <CAM>,<COG>,<GEAR>,<NUT>, and <SCREW>. These values are used as indexes to test the rA bits for being 1 or 0. Since rA(<BOLT>) = rA(0) = 0, ignore the <BOLT>. Since rA(<CAM>) = rA(1) = 1 and rB(<CAM>) = rB(1) = 0 (initially, rB is cleared), set rB(1) = 1 and allocate memory space in MB for those SALES tuples with ITEM = <CAM>. Since r(<CAM>) = r(1) = 3, thus, 4 logical words must be allocated. The allocation will do the following :

(a) Save the contents of word r(<CAM>), now being 3, in T.

(b) Store the contents of D-register, now being 1, in r'(<CAM>) and the contents of BR(D) = BR(1), now being 0, in r"(<CAM>). The word r(<CAM>) = r(1) now is a pointer word pointing to the starting address of the first module. (In fact, the setting of rB bits can be used to distinguish pointer words from counter words.)

(c) (i) Increment BR(D) = BR(1) by 4 (= T + 1) so that if there is any memory allocation assigned to the first module, it will be allocated starting from the fifth logical word (i.e. logical address 4).

(ii) Increment the D-register by 1, indicating that next allocation, if any, will be assigned to the module next to the current one.

The third incoming value is <COG>. Since rA(<COG>) = rA(2) = 0, ignore the value. The same procedure is repeatedly applied to other values. At the end of this step, the bit array store rB and r will be (0,1,0,1,1,0,...,0) and (0, 1·0,0,2·0,3·0,0,...,0)(Figure 3(b)), where r(1) = 1:0 = r'(1) · r"(1)(i.e., concatenation) and the contents of all registers in CP are shown in Figure 3(b).

Step 4. Clear rA and registers T and H and the flag F_Q of Q. Assume that the sequence of SALES tuples input to Q is the same as that of SALES tuples appearing in Figure 2. Any tuples with rB(x) = 1 (x being the ITEM-component of SALES) will be stored in the logical location in MB pointed by r(x). The first incoming tuple with rB(<CAM>) = 1 is stored in the logical location 1 of the first module M(1). (After this, the logical location 0 of M(1) has the value 2, which is initially set to 1 and incremented by 1 when a tuple is stored.). The second incoming tuple with rB(<GEAR>) = 1 is stored in the logical location 1 of M(2); the third incoming tuple with rB(<CAM>) = 1 is stored in the logical location 2 of M(1); and so forth, until the sixth incoming tuple which is stored in the logical location 2 of M(3). At the end of this step, the logical locations 0 of M(1), M(2), and M(3) have the values 4,2, and 3, respectively. Figure 3(c) shows the contents of RAM and the first three modules M(1), M(2), M(3).

Step 5. Assume that the sequence of TYPE tuples input to Q is the same as that of TYPE tuples appearing in Figure 2. Any tuples with rB(y) = 0 (y being the ITEM-component of TYPE) are ignorant. Those tuples with rB(y) = 1 will be dispatched into the queues $Q_i (1 \leqq i \leqq p)$ determined by r'(y). They are concatenated to the contents of r"(y) before dispatching into the proper queues. The first incoming tuple is ignored since rB(<BOLT>) = rB(0) = 0; the second one concatenated to the contents of r"(CAM) = r"(1) = 0 is dispatched into the first queue Q_1 since r' (<CAM>) = r'(1) = 1; the third tuple is ignored; the fourth one concatenated to the contents of r"(<GEAR>) = r"(3) = 0 is dispatched into Q_2 since r'(<GEAR>) = r'(3) = 2; the fifth one concatenated to the contents of r"(<NUT>) = r"(4) = 0 is dispatched into Q_3 since r'(<NUT>) = r(4) = 3; the sixth tuple will be ignored. Since each tuple dispatched is associated with a pointer pointing to the beginning of a block of tuples to which the dispatched tuple is concatenated, each server S_i thus can produce the concatenated tuples of the join from each TYPE tuple in Q_i and the block of SALES tuples in M(i), without memory addressing contention problem.

4. Implementation of Relational Projections

A major implementation issue is how to effectively remove any repetitions within the columns selected. For a projection in which the primary key is one of the columns of a relation being projected, it can be implemented solely by RADM because there is no repetition involved. Others can be implemented by means of the proposed hardware. This section will be devoted to the illustration of how the hardware helps remove the repetitive tuples within the columns projected.

4.1 Projection Using Single Bit Array Stores

A single column projection can be implemented as follows :

(1) Initially rA is cleared.

(2) Scan the column being projected and output the column values to IP which stores them in queue Q. The values v's in Q are then fetched and used as indices to address the rA. If rA(v) = 0, i.e., the value v is not yet output, then v is output and rA(v) $\leftarrow$ 1. If rA(v) = 1, discard the value v. This step continues until all the column values have been processed.

By making use of the single bit array store, the repetitions can be removed in one scan of the values of the column to be projected. This algorithm requires that the size of rA should be large enough so that each column value has one bit position in rA corresponding to it. If not, the same procedure should be repeatedly invoked one for each value interval.

In the case of multi-column projection, the values of the columns being projected are concatenated to a single value, say cv. If the range of cv's are still within the address space of rA, then the same algorithm used in the single column projection can be applied. If the value range of cv's is larger than the address space of rA, it is divided into value intervals and the above algorithm has to be repeatedly invoked one for each value interval. However, it happens quite often that the range of cv's is rather large, while the number of values cv's in each value interval is low. This generally requires a great number of algorithm invocations in order to eliminate any repetitive values cv's.

4.2 Projection Using Hashing Logic

One way to effectively project a relation is to hash the concetenated values cv's to address the bit store instead of using cv's directly. The values cv's which hash to the same bit positions are formed subfiles which are stored in the memory bank MB. The values cv's in each subfile are then separated by the array of servers. The detailed algorithm is shown below :

(1) Initialization : Clear rA, rB, and r.

(2) Hashing and Counting : Scan the relation being projected by RADM and output the values of the columns selected to IP. The values come from the same tuple are concatenated to a single value cv which is stored in Q. Each cv in Q is then fetched and hashed with a hashing function to a value hv. The value hv is used as an index to locate r and increment r(hv) by one. After this step, each entry of r will contain the number of cv's that hash to that entry.

(3) Output and Allocation : Scan the relation again. Each cv stored in Q is read and hashed. If r(hv) = 1, output the cv. Otherwise, if rA(hv) = 0 allocate the memory space in MB for storing the corresponding subfile and rA(hv) ← 1 which indicates that the allocation has been made. (The memory allocation is referred to section 3.3.2.)

(4) Scan the relation again. Each cv in Q is read and hashed. If rA(hv) = 1,

store the cv in the allocated space in MB (referring to Section 3.3.2).

(5) Processing : Each server takes the cv's from its corresponding memory module and eliminates repetitions and output the results to the host computer. Note that each server used convention algorithms to remove repetitions and rB is used in memory alocation not explicitly shown in the algorithm.

Two problems may be associated with the algorithm : one being that if one subfile is considerable large, the server which processes it will take long time to finish and the other being that if the relation being projected is too large to be stored in MB, the projection cannot be proceeded without dividing it into subtasks.

The first problem can be solved by limiting the size of subfiles. For a subfile which contains more than, b, say, concatenated values, it will be by passed in the current process; no allocation for the subfile is made. The cv's in the subfile are just marked. The marked cv's are then processed in the subsequent process by the same algorithm with another hashing function. The algorithm is repeatedly applied until the projection is complete. We note that there are two cases which may cause a large subfile :

(1) The hashing function used causes a great deal of collisions.

(2) The number of duplicated tuples within the columns selected is really large.

The case (1) can be avoided by choosing a proper hashing function each time the algorithm is invoked. In fact, the step (2) of the algorithm can be modified as hashing the concatenated values of subset of the columns being projected instead of hashing each cv as a whole. Different subsets of the columns being projected are considered each time the algorithm is invoked. This can avoid the complexity of hashing logic employed. For case (2), the size of the subfile can no way be reduced. In such case, the algorithm can never be terminated. To avoid this situation, we have to limit the total number of times allowed for the algorithm to be invoked during projection.

What is not made clear here is the problem of how a tuple that has not been processed (i.e., not yet been output or discarded) can be marked a tuple within the columns selected once output to the hardware for projection can never be returened to the RADM. If has to be marked somewhere in the hardware. One way to tell if a tuple has been processed or not is to use an additional bit array store rC for remembering those tuples processed. The algorithm thus is modified as follows :

(1) Initialization : Clear rA, rB, and r.

(2) Hashing and Counting : Scan the relation being projected by RADM and store the concatenated values cv's of the columns selected and their associated tuple identifier x's in Q. Each cv with x is then fetched. If rC(x) = 1 (i.e., the corresponding cv has been processed), discard the cv. Otherwise, the value cv is hashed to a value hv which is then used as index to locate r and increment r(hv) by one. After this step, each entry of r will contain the number of cv's that hash to that entry.

(3) Output and Allocation : Scan the relation again. For each cv with x, if rC(x) = 1 then the value cv has been processed and will be discarded. Otherwise, if r(hv) = 1, output the value cv immediately and set the rC(x) to 1. If 1 < r(hv) < b, use the contents of r(hv) to allocate memory space in MB for storing the corresponding subfile and set both rA(hv) and rC(x) to 1. The setting of rA(hv) indicates that the allocation has been completed.

(4) Scan the relation again. Each value cv in Q is read and hashed. If rA(hv) = 1, store the cv in the allocated space in MB.

(5) Processing : Each server takes the cv's from its corresponding memory module and eliminates repetitions and outputs the results to the host.

The algorithm requires that the number of bits in rC is the same as that of tuples in the relation being projected. Another way to tell if a tuple has been processed or not is to introduce n extra bit array stores $rC^{(1)}$, $rC^{(2)}$, ..., $rC^{(n)}$, each of which has the same size as rA, if n times of invoking the algorithm are considered. Instead of checking the contents of rC(x), we record the value cv that is processed in the ith invocation on the corresponding $rC^{(i)}(hv)$ by setting $rC^{(i)}(hv)$ to 1. The checking of the value cv if it has been processed or not in the previous i-1 invocations is done by examining the stores $rC^{(1)}(hv)$, $rC^{(2)}(hv)$,..., $rC^{(i-1)}(hv)$ for being 0 or 1. The setting of one of the stores indicates that the corresponding cv has been processed and cannot be counted in the subsequent process.

5. Summary and Further Research

We have shown how the RAM helps perform implicit joins, type-I explicit joins, and single column projection. We have also shown how the hardware helps perform type-II explicit joins and multi-column projections. Through the use of the RAM, the joining of relations in which their join columns satisfy the referencial integrity rule can be implemented as effectively as the explicit joins. Since a majority of joins of relations are accomplished via the relationships defined by the rule, it appears that the computing of joins in our approach is fast.

The hardware provides powerful join and projection capabilities to existing RADMs and, thereby, gives a considerable performance improvement over existing RADMs, especially when a join- and project-dominating application is involved. We believe that this extention is adapted to current VLSI technology and has the important characteristics of being applicable with little or without modification to currently proposed RADMs' hardware. The new version of an RADM allows that data search is performed by the search logic, while join and projection operations are carried out by the extended hardware.

Although the design of join and projection algorithms has been completed, there are still many issues to be investigated. For example, performance of the hardware based on different design parameters such as the length of queues, the sizes of the RAM and the memory bank, etc. needs to be worked out. The functions of IP, CP, and each server have to be defined. Since different types of operations can obtain different performance results, this implies that different, but equivalent, execution orders of operations involved in a query may result in different performance results. This leads to a study of a scheme which allows any query expressions to be optimally executed on the new version of RADM.

6. References

[1] Bobb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol.4, 1, March 1979, pp.1-29.

[2] Banerjee, J., and Hsiao, D. K., "DBC — A Database Computer for Very Large Databases," IEEE Trans. on Computers, Vol.C-28, 3, 1979.

[3] Chang, H., "On Bubble Memories and Relational Data Base," Proc. 4th Int'l Conf. on VLDB, West Berlin, 1978, pp.207-229.

[4] Chen, T. C., Lum, V. W., and Tung, C., "The Rebound Sorter : An Efficient Sort Engine for Large Files," Proc. 4th Int'l Conf. on VLDB, West Berlin, 1978, pp.312-315.

[5] Date, C. J., An Introduction to Database Systems, Addison-Wesley, Reading, Mass., Third ed., 1981.

[6] Edelberg, M., and Schissler, L. R., "Intelligent Memory," Proc. 1976 NCC, Vol.45, AFIPS Press, Montuale, N. J., pp.691-701.

[7] Hong, Y. C., and Su, S. Y. W., "Associative Hardware and Software Techniques for Integrity Control," ACM TODS, Vol.6, 3, Sept. 1981, pp.416-440.

[8] Hong, Y. C., and Su, S. Y. W., "A Mechanism for Database Protection in Cellular-Logic Devices," Paper under review for the IEEE Trans. on Software Engineering.

[9] McGregor, D. R., Thomson, R. G., and Dawson, W. N., "High Performance for Database Systems," Systems for Large Databases, North-Holland Publishing Co., 1976, pp.103-116.

[10] Lin, C. S., Smith, D. C. P., and Smith, J. M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS, Vol.1, 1, March 1976, pp.53-65.

[11] Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "RAP — an Associative Processor for Database Management," Proc. 1975 NCC, Vol.44, AFIPS Press, Montvale, N. J., pp.379-387.

[12] Smith, D. C. P., and Smith, J. M., "Relational Database Machines," IEEE Computers, Vol.12, 3, March 1979, pp.28-37.

[13] Su, S. Y. W., "On Logic-Per-Track Devices : Concepts and Applications," IEEE Computers, Vol.12, 3, March 1979, pp.11-25.

[14] Su, S. Y. W., and Lipovski, G. J., "CASSM: A Cellular System for Very Large Databases," Proc. Int'l Conf. on VLDB, Sept. 1975, pp.456-472.

[15] Su, S. Y. W., Nguyen, L. H., Eman, A., and Lipovski, G. J., "The Architectural Features and Implementation Techniques of the Multicell CASSM," IEEE Trans. on Computers, Vol. C-26, 6, June 1979, pp.430-445.

[16] Tanaka, Y., Nozaka, Y., and Masuyama, A., "Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer," Proceedings of IFIP Congress 80, pp.427-432.

[17] Todd, Stephen, "Hardware Design for High Level Databases," IBM United Kingdom Scientific Center, Peterlee, TN 49.

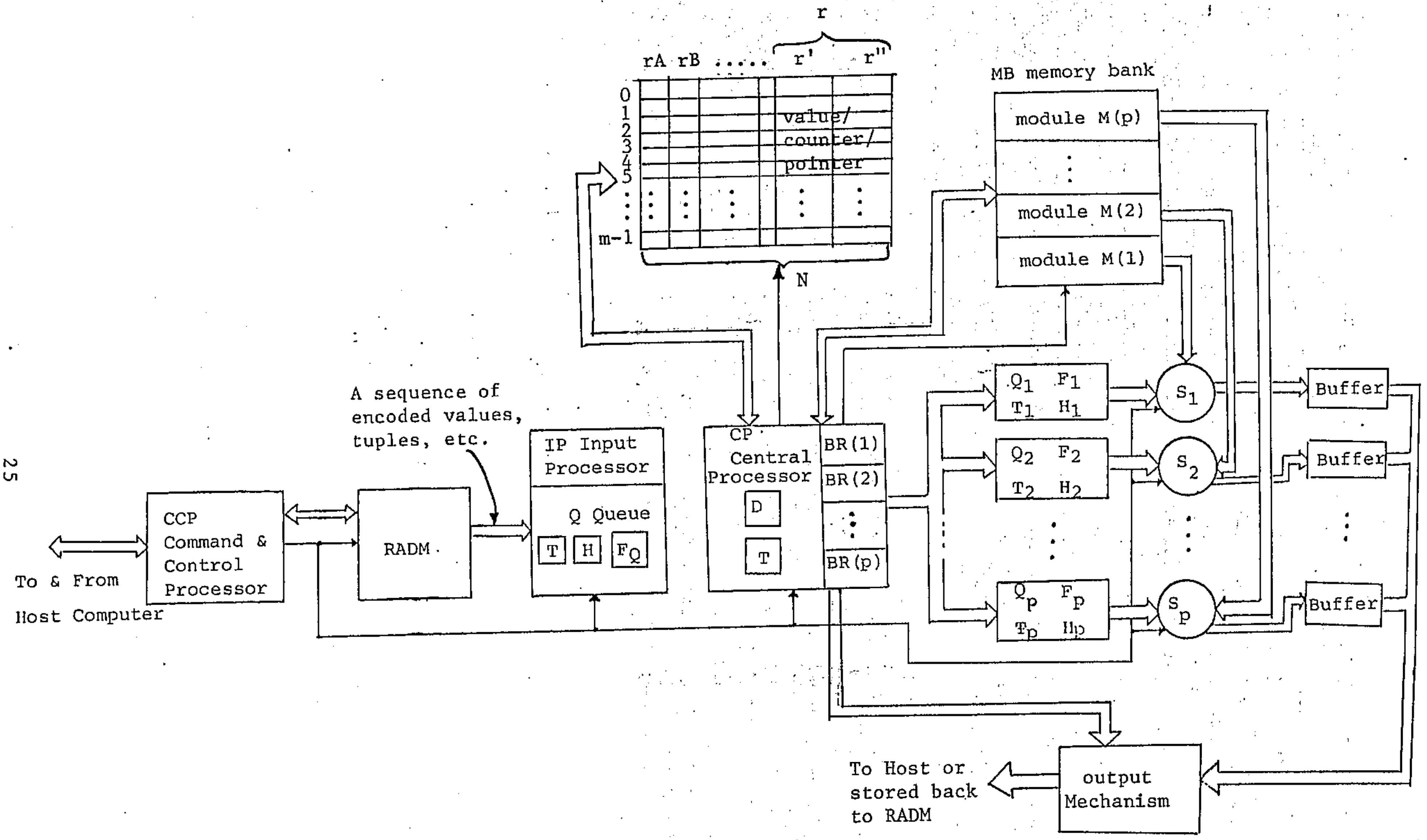

Figure 1. Hardware Architecture

SALES

DEPT	ITEM
<D1>	<CAM>
<D1>	<GEAR>
<D5>	<CAM>
<D5>	<NUT>
<D8>	<CAM>
<D10>	<NUT>

TYPE

ITEM	COLOR	PRICE
<BOLT>	<GREEN>	<5p>
<CAM>	<RED>	<2p>
<COG>	<RED>	<4p>
<GEAR>	<GREEN>	<4p>
<NUT>	<BLACK>	<8p>
<SCREW>	<YELLOW>	<7p>

Figure 2. A Simplified Database With Two Tables SALES And TYPE Linked By ITEM.

RAM

	rA	rB	r
0	0		0
1	1		3
2	0		0
3	1		1
4	1		2
5	0		0
⋮	⋮		⋮
M-1	0		0

Figure 3(a)

RAM

	rA	rB	r'	r"
0	0	0	0	0
1	1	1	1	0
2	0	0	0	0
3	1	1	2	0
4	1	1	3	0
5	0	0	0	0
⋮	⋮	⋮	⋮	⋮
M -1	0	0	0	0

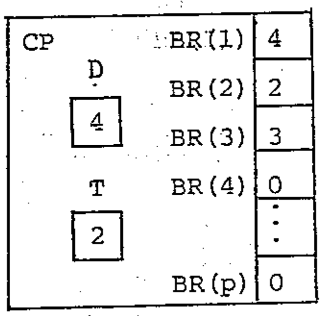

Figure 3(b)

RAM

	rA	rB	r'	r"
0	0	0	0	0
1	0	1	1	0
2	0	0	0	0
3	0	1	2	0
4	0	1	3	0
5	0	0	0	0
⋮	⋮	⋮	⋮	⋮
M-1	0	0	0	0

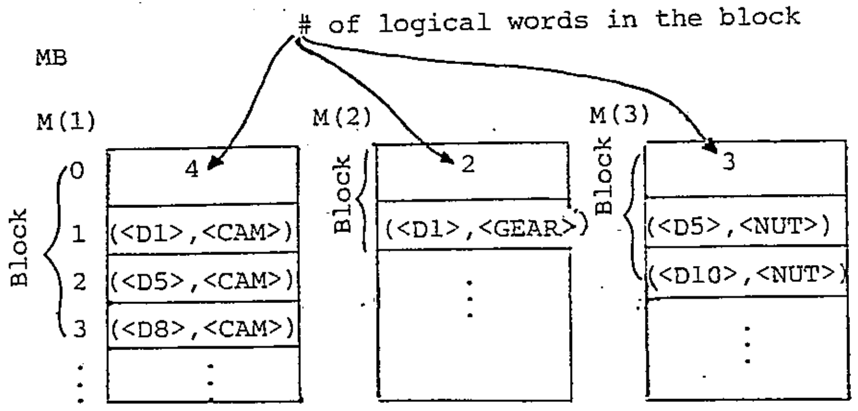

Figure 3(c)