Associative processing for general purpose computers through the use of modified memories*

by HAROLD S. STONE

Stanford Research Institute
Menlo Park, California

INTRODUCTION

The concept of the content-addressable memory has been a popular one for study in recent years, but relatively few real systems have used content-addressable memories successfully. This has been partly for economic reasons—the cost of early designs of content-addressable memories has been very high—and partly because it is a difficult problem to embed a content-addressable memory into a processing system to increase system effectiveness for a large class of problems.

In this paper, we describe a relatively inexpensive modification to the memory access circuitry of a general purpose computer that will permit it to perform some of the operations that can be performed in a content-addressable memory. The major restriction is that the memory must be a 2D memory. The modification results in a new access mode to memory, one which permits a bit slice from a number of different words in memory to be accessed. Memory can be viewed as a collection of N X N arrays of bits such that an access can be made to either the i-th row or the i-th column of an array. Although this capability is somewhat limited by comparison to the capability of large content-addressed memories as they are normally conceived, it is ideally suited to that class of problems that requires both conventional and associative processing. A good example of this type of problem is the Gauss-Jordan algorithm for matrix-inversion which involves a search through a matrix for the numerically greatest element.

In the next section we further describe the functional behavior of the modified memory and illustrate its use through a series of examples. The subject of the third section is the modification of a 2D memory in order to permit row-column access. The last section contains an evaluation of the technique and a summary.

Assoiciative processing with row-column operations

The functional behavior that is described in this section is not novel; it was first described nearly a decade ago. The reason for its resurrection is due to the ease with which it can be implemented in present technology using techniques described in the next section.

The basic idea is illustrated in Figure 1. Memory is viewed to be partitioned into multiple arrays of bits, each of size N X N, where N is a power of two. Memory accesses can be made in one of two modes, row mode or column mode. In either mode, a particular array in memory is selected by the high order bits of an effective address while the least significant log2 N bits select either a row or a column. This behavior is essentially the same as that of the horizontal-vertical computer described by Shooman. Row selection is equivalent to word selection in a conventional computer. The column selection mode has no counterpart in conventional computers and is the mode that supports a limited form of associative processing.

There is a major difference between Shooman's vertical-horizontal processor and the idea that is developed in this paper. Shooman conceives of two separate collections of registers and processing logic to be used in his processor, one for vertical processing and one for horizontal processing. An implementation of Shooman's idea is described in Ref. 12. What we describe here uses just one collection of registers and logic. This characteristic comes about because in the modified memory all data is transferred between memory and a common data register for both row and column operations.

To be more specific, Figure 2 shows an N X N array of bits and shows the contents of the memory data register.
Let $X$ and $Y$ be the base addresses of two $N \times N$ bit 0, 1 matrices in memory. Then rows of $X$ can be stored as columns of $Y$ by iterating the instructions below for different values of the index variables.

$$R_1 \leftarrow X[I];$$
$$Y[J] \leftarrow R_1;$$

If $I$ is equal to $J$ and the loop is repeated for $I = 1, 2, \ldots, N$, $Y$ will contain a copy of $X$ rotated a quarter turn. If $I = N-J$, then $Y$ will contain the transpose of $X$ where the transpose is taken about the minor diagonal.

To aid in search operations, we introduce the machine function NORMALIZE. The NORMALIZE instruction left shifts a specified register until a “1” appears in the left-most position or until $N$ shifts are performed if there are no “1”’s in the register. A count of the number of shifts is placed in another designated register. In our symbolic notation we use the form

$$R_2 \leftarrow \text{NORMALIZE}(R_1);$$

to mean that $R_1$ is normalized and the shift count is placed in $R_2$.

As an illustration of the use of the column operation for searching consider the problem of scanning a status vector array to find a vector with a “1” in the $i^{th}$ bit position. The pair of instructions that perform the search are

$$R_1 \leftarrow \text{STATUS}[I];$$
$$R_2 \leftarrow \text{NORMALIZE}(R_1);$$

$R_2$ contains $N-j$ where $j$ is the index of a vector with a “1” bit in the $i^{th}$ position. (For programming convenience, it would be wise to implement the NORMALIZE command so that the result left in $R_2$ would be $j$ instead of $N-j$.)

Now consider the problem of searching an array for a vector that contains a particular field matching a specified pattern. The technique that we use is to perform an iterative sequence of column operations on the pattern field. In the instruction sequence below, register $R_1$ holds the pattern, $R_2$ holds columns fetched from memory, and $R_3$ holds a 0, 1 vector that contains a 1 bit in a bit position if the corresponding word in the array has matched the pattern on all preceding operations. The sequence is initialized so that the left-most bit of the pattern lies in the left-most bit of $R_1$, and $R_3$ is initialized to all “1”’s. “NOT” and “AND” are full register logical operators.
R₂ ← X[I];  Fetch the next column of the pattern;

IF LEFTBIT(R₁) = 0 THEN
R₂ ← NOT R₂;
R₄ ← R₃ AND R₂;  Mask out bits in R₂ that disagree with pattern for this iteration.
R₁ ← LEFTSHIFT(R₁);

The sequence of instructions must be repeated for values of I ranging over the index field. After the final iteration, R₄ contains a "1" in positions that correspond to words with fields that match the pattern. A NORMALIZE command can be used to obtain the addresses of words that satisfy the search criterion.

Note that the search procedure effectively looks at N words but the number of fetches required is equal to length of the pattern. Hence, for short patterns, considerably fewer than N memory fetches will serve to search N words. Patterns that occupy a full word can be processed about equally well in row mode or in column mode.

Searches need not be made on "equality" matches. A slight modification of the instructions above is all that is required to implement a threshold search. We use one more register, R₄, that contains "1"s in positions that correspond to words with fields greater than the pattern. R₁, R₂ and R₃ are used as before. R₄ is initialized to all "0"s prior to executing the sequences below.

R₂ ← X[I];  Fetch next column of the pattern;

IF LEFTBIT(R₃) = 0 THEN
BEGIN R₄ ← R₄ OR (R₃ AND R₂);
R₂ ← NOT R₂;
END;
R₃ ← R₃ AND R₂;
R₁ ← LEFTSHIFT(R₁);

The statement immediately after the "BEGIN" updates the vector in R₄ such that a word is greater than the pattern if it had been greater on the previous iteration or had been equal before and is greater on the current iteration. At the close of a sequence of iterations of the program steps given above, the vectors in R₃ and R₄ uniquely identify all words that are either equal to the pattern or greater than the pattern. All of the remaining words are clearly less than the pattern.

Sets that satisfy any of the relations "=", "≠", "<", "≤", ">" and "≥" can be obtained easily by simple operations on the vectors in R₃ and R₄. "Between limits" or "outside limits" searches can be done by using three registers for each limit, i.e., one register to hold the limit pattern, and two that function as counterparts of R₃ and R₄. A number of other operations are also possible with row-column accessing. Many of these are given in Shooman7. The examples given here and in Reference 7 should suffice to illustrate the power of idea. It is appropriate at this point to consider the implementation.

The memory modification

The memory modification can be developed from a discussion of the functional requirements of row-column processing and the constraints of conventional memory technology. We begin by examining the N × N matrix in Figure 2.

In a conventional memory, the bits in the matrix are stored so that the columns of the matrix lie on separate and distinct sense lines of the memory. Because of constraints of conventional memory technology, during any memory cycle no more than one bit per sense line can be read or written. For row operations, selection circuitry activates all bits of a specified row, each of which lies on a different sense line. The physical portion of a core memory that is threaded by a sense line will be called a bit plane in the following material. For both 2D and 3D memory organizations, the entities that we call bit planes correspond to physical planes of memory stacks, but the correspondence is usually not true for 2D memories.

For column operations, the technology constraint is severely restrictive. With physical memory organized...
as shown in Figure 2, columns of arrays will lie wholly within bit planes, and thus no more than one bit per column could be accessed during any memory cycle under the assumed constraint. In order to overcome the constraints of memory technology, memory can be organized as shown in Figure 3. Each row in the array corresponds to a row in the array shown in Figure 2, but in Figure 3, the matrix is stored such that the $i^{th}$ row is cyclically shifted to the left by $i$ bit positions. Careful examination of the figure shows that both rows and columns of the matrix have the property that exactly one bit lies in each plane. The bits are scattered and shifted, however, so that the memory access circuitry must be constructed to take this into account.

Figure 4 shows how row and column selections must function when data is held in a skewed fashion. In Figure 4a, the first row is read into a data register, and must be cyclically shifted right one bit to place in a standard format. In general, an operation on the $i^{th}$ row requires a right cyclical shift of $i$ bits after a read cycle, and a left cyclical shift of $i$ bits before a write cycle.

Column operations are somewhat more complex. Examination of Figure 4b shows that $N$ different words must be accessed to obtain all of the $N$ bits in one column. Specifically, the memory planes must be able to support simultaneous selection of bits in different rows. We shall return to this point later. The planes in Figure 4 are numbered from left to right as $0, N, N-1, \ldots, 1$. The selection of the $i^{th}$ column is such that the $j^{th}$ plane must select the bit in word $(i+j) \mod N$. After access, the word must be cyclically shifted to the right by $N-1$ bits. The skewed storage technique has been used in the Illiac IV design, where Illiac IV memory modules correspond to bit planes here, and full word operands in Illiac IV correspond to bits.

Thusfar, the discussion has been functional in nature. Now we must take a closer look at memory technology. Among the conventional memory organizations only one can easily be adapted to permit accesses to different bits in different bit planes. This memory organization is the so-called "$2\frac{1}{2}D" memory, and its organization is shown in Figure 5.

In the figure, it is shown that addresses are split into two components, $X$ and $Y$, and are separately decoded. There is a single set of $X$ drivers for all planes, but there is an individual set of $Y$ drivers for each bit plane. The several sets of $Y$ drivers act in unison for selection purposes in the normal mode of operation. It is precisely the multiplicity of $Y$ drivers that permits the memory access circuitry to be modified to support the column mode of operation.
For column mode operations, it is necessary to modify the drive circuitry slightly in order to place independent control in each plane for the selection of the Y driver to be energized. The requirements are that each plane energizes either the driver corresponding to the decoded Y address (row mode) or the driver corresponding to the sum of decoded Y address and bit plane index (column mode).

Since the plane index is fixed for each plane, it is not necessary to use an adder in each plane to implement associative access mode. Consider, for example, the schematic diagram of the memory drive circuitry for a single bit plane as shown in Figure 6. In this figure, the X drive lines link all bit planes and effectively select a particular N x N bit array for interrogation. The Y drive lines select either a row or a column from the N x N array. The circuit in Figure 6 is intended to be identical to that in Figure 5 except for the two level logic circuit shown in dashed rectangle.*

The two level logic circuit has the following property. In row mode, signal A is false, and the output of the Y decoder is fed directly to corresponding drive lines in the selection matrix. Note that all planes act identically in this condition, so that each plane returns a bit from the same word address. When signal A is true, the output of the Y decoder is displaced cyclically by an amount i in the ith plane. Thus, each plane reports a bit that belongs to a different word of the N x N array, and in fact reports the bit indicated by Figure 4.

The signal displacement in column mode is "end-around." That is, if the decoded address is the jth address, then line (i+j) mod N will be activated in the ith plane.

In terms of logic circuitry, the two level circuit shown in Figure 6 is the only modification that is necessary for the access circuitry.

One possible adverse effect of the two level logic circuit is a small increase in the total memory cycle time. This increase is expected to be less than one percent in magnetic storage technologies, and is likely to be more than balanced by an increase in memory effectiveness. However, in newer technologies, such as integrated circuit memories, the increase in cycle time may be somewhat larger.

We turn our attention now to the problem of cyclically shifting data prior to entry to the memory and after retrieval from memory. The two shifts are called preshifts and postshifts, respectively. To solve the shifting problem note that the amount of a preshift or a postshift can be derived from the mode and the effective memory address. Let the least significant log_2 N bits of an effective address be called the s index (s for shift). Then, the shift direction and amount is given by the table below.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Preshift amount</th>
<th>Postshift amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td>s bits left</td>
<td>s bits right</td>
</tr>
<tr>
<td>column</td>
<td>N - s - 1 bits left</td>
<td>N - s - 1 bits right</td>
</tr>
</tbody>
</table>

Note that the value of N-s-1 can be computed by taking the 1's complement of s in a register with log_2 N bits. Consequently, it is extremely simple to compute the shift amount because it is equal to the lower address bits in row mode or the 1's complement of these bits in column mode.

A good candidate for the shift circuit is the barrel shifter shown in Figure 7. Shifters of this type have been...
implemented in several commercial computers. A shifter with \( \log_2 N \) stages can cyclically shift \( N \) bit words by any amount from 0 to \( N-1 \). Each stage shifts either 0 or \( 2^i \) bits depending on the \( i^{th} \) bit in the binary representation of the shift amount. For our purposes, it is most likely that two shifters are needed, one for postshifting and one for preshifting. The two shifters should be placed on the data paths between memory and processor, and not between the memory data register and the memory. The reason for the placement on the data paths is that there is a possibility for overlapping the shift operation with other operations, whereas if the shifters were placed between data register and drivers or sense amplifiers, the effect would be to increase the memory cycle.

A good example of masking the effect of shifting time through overlap of operations occurs for write cycles. Normally, write cycles are preceded by clear cycles to clear memory in the bit positions that are to receive new data. The clear cycle can proceed as soon as an address is available. While the clear cycle is active, data can be shifted into the proper format and be ready for storage when the write cycle begins. Post-shifting after read cycles will tend to increase the time required to access data but it will not increase the memory cycle time.

The barrel shifters thus are used to translate between a standard data format for the processor and one of \( N \) different storage formats. Because data within the processor are held in a standardized form, manipulation of data within the processor can be done in conventional ways. Conventional load and store commands can be issued from the central processor without regard for the fact that data might be cyclically rotated during the transfer. In essence, the physical form of data storage is completely invisible to the processor.

What has been gained, of course, with the barrel shifter, the modified memory, and the modified storage format, is the capability for performing associative operations in a general purpose computer. The net cost of the memory modification is two-barrel shift registers, and some logic in the \( Y \) address circuitry. The cost is undoubtedly a small fraction of the cost of the memory and processor. There may be some impairment of performance because of increased access time, but this is unlikely to be significant because the delays through a microelectronic barrel shift register are very small compared to the cycle time of a core memory.

SUMMARY AND CONCLUSIONS

The performance benefits to be derived from row-column addressing depend greatly on the effectiveness of column mode operations in reducing the number of memory accesses. The greatest benefit of column mode addressing is for those operations in which a small portion of a large number of words in memory must be accessed. Scaling and magnitude searches fall in this category. Take, for example, IBM System/360 long floating point words with 7-bit exponents and 56-bit mantissas. Scaling and magnitude searches over 64-word groups can be done with approximately eight accesses by accessing the exponent field of the groups in column mode. This is essentially how one would conduct the search for a pivot element in a Gauss-Jordan reduction.

Since the word-length effectively determines the number of different words that are accessed in a column operation, it is desirable to have as large a word length as possible. In present technology, it is feasible to implement memories with up to 64 or 128 bits/word. It appears that reasonably good performance improvement is possible with 64-bit words, so that a useful implementation of row-column accessing is possible within present technology.

Some analysis is required to determine the utility of column accessing for common operations such as symbol table searching and sorting. Hash-addressing used in combination with column mode access is one possible method for performing table searching. Hash-addressing normally involves a search when conflicts occur\(^9\),\(^11\) and this search might be speeded with column access. The biggest improvement would come when tables are nearly full, at which time there is a high probability that a search will take place after the computation of the hash-code.

Ultimately, the performance improvement to be derived from the techniques that have been described in this paper are determined by the applications. While we cannot predict what benefits might accrue at this time, the fact that row-column accessing can probably be achieved for little cost within present technology indicates that the benefits of an implementation of the technique will almost certainly outweigh the cost of implementation.

REFERENCES

1 A KAPLAN
   A search memory subsystem for a general purpose computer

2 RGEWING P M DAVIES
   An associative processor

3 B T MckEEVER
   The associative memory structure

4 A G HANLON
Content-addressable and associative memory systems—a survey
IEEE TEC Vol EC-15 No 4 pp 609–621 August 1966
5 T J GILLIGAN
2-1/2D high speed memory systems—past present and future
IEEE TEC Vol EC-15 No 4 pp 475–485 August 1966
6 J R BROWN JR
First and second order ferrite memory core characteristics and their relationship to system performance
7 W SHOOMAN
Parallel computing with vertical data
Proceedings of the EJCC Vol 18 December 1960
8 W SHOOMAN
Orthogonal computer

US Patent 3 277 449 October 4 1966
9 D L SLOTNICK
Unconventional systems
AFIPS Proc of the 1967 SJCC Thompson Book Co Washington
DC pp 477–481
10 W D MAURER
An improved hash code for scatter storage
CACM Volume 11 Number 1 pp 35–38 January 1968
11 R MORRIS
Scatter storage techniques
CACM Volume 11 Number 1 pp 38–43 January 1968
12 P A HARDING M W ROLUND
A 2-1/2 D core search memory
Fall Joint Computer Conference December 1968

From the collection of the Computer History Museum (www.computerhistory.org)