ilarly, the number of distinct delay row numbers associated with the states of the second block of θ_i corresponds to the number of blocks $\overline{K}(\theta_i)$ contains. Of course, the number of blocks in $J(\theta_i)$ and $\overline{K}(\theta_i)$ provides the information as to the amount of state variable dependency possessed by the respective J_i and $\overline{K}_i$ or K_i flip-flop input functions. Thus, the θ_i whose associated J_i and $\overline{K}_i$ or K_i input functions have the least combined variable dependency is chosen.

In conclusion, it is hoped that the several generalizations of the DM and WS methods presented in both parts of this study will permit the reader to choose that method which is most appropriate for his sequential machine design purposes.

References

[1] H. A. Curtis, "Systematic procedures for realizing synchronous sequential machines using flip-flop memory: Part I," *IEEE Trans. Computers*, vol. C-18, pp. 1121–1127, December 1969.
[2] T. A. Dolotta and E. J. McCluskey, "The coding of internal states of sequential circuits," *IEEE Trans. Electronic Computers*, vol. EC-13, pp. 549–563, October 1964.
[3] P. Weiner and E. J. Smith, "Optimization of reduced dependencies for synchronous sequential machines," *IEEE Trans. Electronic Computers*, vol. EC-16, pp. 835–847, December 1967.
[4] C. Harlow and C. L. Coates, "On the structure of realizations using flip-flop memory elements," *Inform. Control*, vol. 10, pp. 159–174, February 1967.

Short Notes

A Logic-in-Memory Computer

HAROLD S. STONE

Abstract—If, as presently projected, the cost of microelectronic arrays in the future will tend to reflect the number of pins on the array rather than the number of gates, the logic-in-memory array is an extremely attractive computer component. Such an array is essentially a microelectronic memory with some combinational logic associated with each storage element.

A logic-in-memory computer is described that is organized around a logic-enhanced "cache" memory array. Used as a cache, a logic-in-memory array performs as a high-speed buffer between a conventional CPU and a conventional memory. The effect on the computer system of the cache and its control mechanism is to make the main memory appear to have all of the processing capabilities and almost the same performance as the cache.

Operations within the array are naturally organized as operations on blocks of data called "sectors." Among the operations that can be performed are arithmetic and logical operations on pairs of elements from two sectors, and a variety of associative search operations on a single sector. For such operations, the main memory of the computer appears to the program to be composed of a collection of logic-in-memory arrays, each the size of a sector.

Because of the high-speed, highly parallel sector operations, the logic-in-memory computer points to a new direction for achieving orders of magnitude increase in computer performance. Moreover, since the computer is specifically organized for large-scale integration, the increased performance might be obtained for a comparatively small dollar cost.

Index Terms—Cache memories, computer architecture, logic-in-memory, microelectronic memories, unconventional computer systems.

I. Introduction

As the microelectronics industry has developed from integrated circuits into so-called medium-scale integration, and as the future promises to bring large-scale integration, it has become clear that the cost and complexity of microelectronic packages are only partly influenced by the number of gates or the number of transistors per package. Module cost is influenced primarily by the cost of packaging and testing, while module complexity is more strongly influenced by the number of pins per module. It is conceivable that component cost will become so heavily dependent on factors other than the number of gates per package that doubling or tripling the number of gates per package may not affect the cost of the package materially, provided that other factors—such as the number of pins per package, the process yield, and the cost of testing—remain the same. Such projections provide a challenge to the logic designer and system architect because it appears to be technically and economically feasible to obtain a substantial increase in hardware complexity of a computer system, provided that new computer system designs satisfy the restrictions and limitations of advanced microelectronic technology.

A natural starting point in meeting the challenge is the design of a memory system using microelectronic components. Memory is inherently cellular by nature; consequently, it is particularly well-adapted to realization in two-dimensional microelectronic arrays. The number of external connections required for a two-dimensional memory array is a linear function of the array dimensions, whereas the number of elements in the array is a quadratic function of the array dimensions. In view of pin limitations of microelectronics, large-memory arrays have favorable ratios of complexity to number of pins. It is also reasonable to attempt to enhance a microelectronic memory by including additional logic in the memory array to enable a certain amount of processing to be performed in the memory array, provided that the high complexity-to-pin ratio can be sustained. In this manner, the "logic-in-memory" array can be made extremely powerful by virtue of its

Manuscript received October 22, 1968. This work was supported at Stanford Research Institute by the Office of Naval Research, Information Systems Branch, under Contract Nonr-4833(00), Requisition NR048-206.

The author is with Stanford Research Institute, Menlo Park, Calif.

ability to perform logical operations in memory and the inherent potential for parallelism of operation.

It has been typical of the research in logic-in-memory arrays that each type of array has been visualized as a functional unit of a computer system. Most attention has been directed to the arrays themselves rather than to the computer system in which such arrays are embedded in order to gain a fundamental understanding of the characteristics of cellular logic-in-memory arrays and to establish the techniques for their design. One functional behavior that the logic-in-memory array could assume is that of an associative memory; several other forms have been investigated by Kautz *et al.* [1]–[4], including logic-in-memory arrays for performing threshold logic processes, permutation switching, error-correcting code implementation, and sorting. It is the purpose of this note to focus attention on the problem of embedding logic-in-memory arrays in a computer system. Specifically, we describe how a class of logic-in-memory arrays can be used as high-speed buffer memories as an extension of the approach used in the design of the IBM 360/85 [5]. That approach in turn owes its design to the Ferranti Atlas virtual memory [6] and to Wilkes' slave memory [7].

Section II is a brief review of the characteristics of the high-speed buffer memory as used in the IBM 360/85. Section III then describes the characteristics of a computer in which the high-speed buffer is a cellular logic-in-memory array. Section IV considers extensions to the approach of Section III, with particular emphasis on explicit (rather than implicit) control of the high-speed buffer memory.

II. A Computer Organization Containing a High-Speed Buffer Memory

In terms of today's economics, the cost per bit of storage is much cheaper for magnetic technologies than for microelectronic technologies. However, since microelectronic memories are orders-of-magnitude faster than magnetic memories, microelectronic memories are attractive for computer systems in spite of their relatively high cost. The economic picture is bound to change in the near future to make microelectronic memories even more attractive, although it is doubtful that the per-bit cost of microelectronic memories will drop below that of magnetic memories. Hence, it is reasonable to use both types of memories within one computer system in order to attain the high performance of microelectronics while deriving the cost benefit of magnetic devices. As an example of such a system, we consider here the IBM 360/85 computer which uses a microelectronic buffer memory 64 to 256 times smaller than main memory as an interface between the central processor and main memory.

The coupling of memories described here is similar to the coupling of a large rotating magnetic memory to a smaller and faster random-access memory so that the computer system performs as if the entire memory were the size of the bulk memory, yet were comparable in speed to the fast random-access memory. Bulk memory

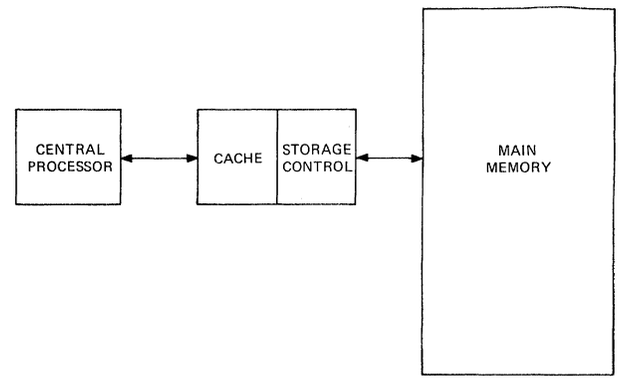

Fig. 1. The structure of a cache-organized computer.

is usually a magnetic drum or disk, and access times are 10^3 to 10^4 times greater than access time for the main memory. Clever memory management techniques have been developed for bulk memories that allow typical computations to proceed at rates only two to ten times slower than they would in a large high-speed memory [6], [8]. Since the memory management techniques have been well proven in practice, the designers of the 360/85 had a great deal of past experience to aid them in the design of their system.

Memory management algorithms are all based on the empirical observation that memory accesses tend to be highly correlated. Regions of active memory can be identified at particular points in time during the execution of a program; these regions of activity tend to change relatively slowly compared to the basic speed of program execution. The algorithms are designed so that active areas of memory tend to reside in high-speed memory, and as the activity changes, newly active regions of memory exchange places in high-speed memory with regions that have become dormant.

In the 360/85, a high-speed microelectronic memory, called the "cache," serves as a buffer between the main memory and CPU. The organization is shown in Fig. 1. The cache can hold 16 sectors of memory, each sector containing 1024 bytes. The cache is an 80-ns memory, while main memory operates anywhere between 750 and 8000 ns. All memory requests from the CPU are directed to the cache; as long as the requests can be granted by the cache, the computer system operates unhindered by the relatively slow speed of the main memory. When a request is directed to a sector of memory that is not resident in the cache, that sector of memory is brought into the cache for manipulation by the CPU.

To mechanize the operation of the cache, a 16-word associative memory (one word for each sector in the cache) identifies each sector in the cache by its address in main memory. When a memory request is issued, the address of the request is compared associatively with the addresses of the sectors in the cache to determine if the desired sector is in there. If a sector must be retrieved from main memory, the least recently used sector in the cache is replaced with the desired sector.

The interface between the cache and main memory is

arranged to expedite the transfer of data into the cache. Main memory is composed of independent memory modules, each module being capable of accessing 16 contiguous bytes in a memory cycle. Memory addresses are interlaced among groups of four modules so that contiguous regions of memory are divided into 16-byte "words," and the words at successive addresses are distributed cyclically among the modules. Data transfers between main memory and the cache are done in 4-word (64-byte) groups called "blocks," and 16 such transfers must be done to move an entire sector into the cache. A block can be moved into the cache in approximately the time it takes for a main memory cycle. The four 16-byte words in a block are retrieved from four different memory modules by directing memory requests to the modules at 80-ns intervals. Thus the transfer time required to move an entire sector into the cache is on the order of 10 main memory cycles.

While the performance benefit of the cache has not yet been reported for a real 360/85, it has been measured through simulation. Liptay claims that, on the average, over 95 percent of the CPU memory requests were directed to data sectors that were already resident in the cache for the mixture of representative programs that were simulated [5]. In terms of total performance, the simulated 360/85 performs computations at 80 percent of the speed of a similar machine whose entire main memory operates at the speed of the cache, yet the cache is at most a few percent of the size of main memory. The conclusion to be drawn here is that small microelectronic memories can be embedded effectively in computer systems as cache memories. Moreover, as long as the cost of microelectronic memories is greater than the cost of magnetic memories, it is not reasonable to construct large microelectronic memories because the performance of a large magnetic memory can be made almost equal to that of a large microelectronic memory by using a cache in conjunction with the magnetic memory.

III. A Logic-in-Memory Cache

The description of a cache in the previous section points out that the apparent memory speed of a computer system can be very close to the speed of the cache, even though main memory is very much larger and very much slower than the cache. The operation of the cache is invisible to programs, in the sense that memory requests by programs are always executed as if the request were directed to main memory. Actually, a majority of the memory requests are executed in the cache. The effect of the cache on the execution of the program is to make main memory appear to be endowed with the performance characteristics of the cache.

In the 360/85, the cache has the same functional characteristics as main memory, although it has different performance characteristics. A moment's reflection shows that the cache need not be limited to standard memory functions but can be capable of performing a wide variety of logic-in-memory operations that are not possible to perform in main memory. The effect of the cache and its control mechanism is to make main memory appear to be endowed with all of the capabilities of the cache as well as its performance.

Let us consider a cache-organized computer system in which each sector of the cache is an independent logic-in-memory array. We assume that data references cause sectors to be transferred from main memory into the cache if the sectors are not already there. Therefore, the CPU and the controlling program can always expect to find requested data in the cache, regardless of the location of the data when the request is issued. Consequently, the CPU can issue instructions for manipulation of data by a logic-in-memory array and the manipulation can be carried out in the cache. In this organization, main memory appears to a program as if every sector in main memory were an independent logic-in-memory array.

The types of operations performed by a logic-in-memory cache are best described by example. Suppose that each sector of the cache is an associative memory. For associative processing, the instruction repertoire of the CPU might include the instructions described below.

Search on Masked Equality: This instruction has two registers, designated as operands, and a memory address. The first operand register is assumed to contain a pattern to be matched; the second operand is a mask to be used in the search. The address is treated as the base address of a sector of memory. The execution of this instruction causes a search of the sector for a word that matches the pattern in those positions indicated by the mask. The first word found that matches the pattern replaces the pattern in the pattern register. All words in the sector that match the pattern will have a tag bit set to 1.

Search on Masked Threshold: This instruction is identical to the masked-equality search, except that the comparison need not be for exact equality but for "greater than" or "greater than or equal." A convention must be established for dealing with negative numbers to take into consideration the location of the sign bit and the representation of the negative number. Other search commands can easily be formulated to do combinations of arithmetic comparison, maximum and minimum searches, pattern inclusion, and pattern covering.

Copy Tag Bit: The parameters for this instruction are an indexable operand and a sector memory address. The operand is interpreted as a bit position; the execution of the instruction causes the tag bits in each word of the designated sector to be copied from their resident position to the indicated bit position. This operation allows tag bits to be stored after they are set by search operations.

Tag Bit AND: This instruction requires two indexable operands and a sector memory address as parameters. The two operands indicate the two bit positions in each word of the designated sector that are to be ANDed together. The result of the AND operation replaces the first operand bit. Other similar instructions are OR, EXCLUSIVE OR, and NOT.

Sector ADD: This instruction requires two sector memory addresses. Corresponding words of the two sectors are added together and the result replaces words in the first sector. Similar sector commands can be included to perform the sector multiply operation (inner product) and the sector copy operation.

Sector Scale: This instruction takes one operand and a sector address. The execution of this instruction causes each word in the specified sector to be multiplied by the operand. A "sector bias" operation is similar except that the operand is added to each word in the sector.

Two important aspects of the instruction repertoire given above bear further discussion. The first has to do with the utility of the instructions; the second concerns the cost of their implementation.

With respect to the utility of the instructions, it should be noted that the instructions given here are meant to be illustrative of the kinds of instructions that could be introduced into a computer system with a logic-in-memory cache. The instructions have been selected because they appear to be useful, i.e., they perform complex operations that are common to a variety of computer programs. The actual utility of these instructions, or of any other similar set of instructions, is a difficult quantity to measure, and is actually a problem for future research. We discuss this point further in Section IV.

The cost of implementation of the instructions depends very much on the cost and complexity of logic-in-memory arrays. Masked-equality searches require a few gates per memory bit in an array and are reasonable to implement today. Examples of such arrays have been exhibited in numerous papers [9]–[11].

The "sector add" command requires an adder for each word in a sector. However, only one bank of adders is necessary to serve all sectors in a cache if the adders are not embedded directly in the sector arrays. Fig. 2 shows how this might be done. The figure shows a "word slice" from a cache with a "sector add" capability. The two-bus system shown in the figure connects the ith word from any sector to the ith adder input, and the adder output to the ith word of any sector. To perform the "sector add" operation, the first operand is placed on the cache output bus and read into register R. The second operand is then placed on the output bus and gated through the adder, where it is added to the contents of R. The adder output is placed on the cache input bus and is gated into the appropriate sector. (It may be necessary to place a temporary register on the adder output to ensure proper timing.)

Since Fig. 2 shows only a "word slice," one must imagine the connection in the figure repeated for every word in the cache. Clearly, this calls for a good deal more complexity than is found in today's computers; however, it is reasonable for the future if the cost of microelectronics diminishes as projections promise. While still greater complexity is required to implement the "sector multiply" command, even this complexity is realistic. Multiplication can be done as a sequence of shifts and adds; the circuitry would possibly be two

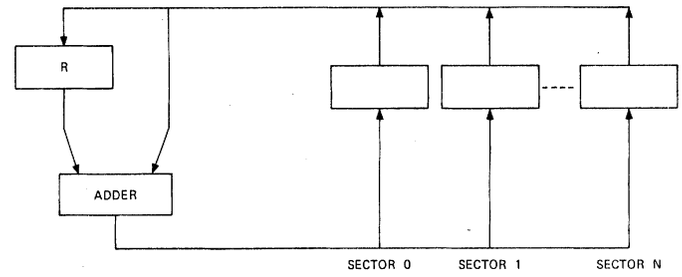

Fig. 2. A word slice of a cache logic-in-memory array with a "sector add" capability.

to three times as complex as that required for sector addition alone.

It is probably not necessary, or even desirable, to perform sector arithmetic operations in floating-point formats. The increased complexity of the floating-point units, as well as problems in the relative significance of results, lead one to expect that fixed-point formats should be used, and that "search for maximum" and "sector scale" instructions be used to scale data to ensure maximum significance of results. The main difficulty to be encountered in implementing the instructions described in this section (or similar instructions) will be in determining if the cost of the implementation is balanced by performance improvement. Clearly, if implementation cost drops sufficiently, the implementation will easily be justified.

To summarize this section, we have postulated that a cache can be implemented as a collection of logic-in-memory arrays. The effect of such an implementation is to make an ordinary magnetic memory appear to a program to be a collection of independent sectors, each of which is endowed with all of the capabilities and the performance of the logic-in-memory arrays in the cache. In this organization, programs do not issue commands for controlling the cache, but rather issue commands for manipulating data in main memory. A mechanism that is independent of the programs is responsible for transferring requested data to the cache as required.

In the next section we will consider the effects of giving explicit control of the cache to programs.

IV. Logic-in-Memory Caches Operating Under Program Control

One principal attribute of the cache as described in Section II is that the cache is completely invisible to programs. There are several advantages in operating a computer system with this assumption; the most important is that the system can make effective use of the cache without any explicit guidance from programs. However, in some circumstances, programs can provide information that will contribute to even greater effectiveness to the use of the cache. Among the control commands that might be issued by a program are those that help the computer system anticipate future requests for data. Data arrays and program segments might be identified as "sequential" to cue the computer to remove sectors containing these items from the cache

immediately upon access to the last item in the sector. Directives might be issued that establish priorities for sectors, that "hold" sectors in the cache (if space is available), and that "release" sectors when they are no longer needed.

Another interesting type of cache control is the control that instructs the computer system how to load the cache. Heretofore we have assumed that sectors in the cache correspond to sectors in main memory; however, this is not strictly necessary. Matrices can be allocated to main memory so that either rows or columns of data can be fetched to the cache. The technique is to skew the matrix in memory so that both rows and columns of data are interlaced among the memory modules as is done in the ILLIAC IV [12]. Then a row of data can be fetched in the usual way by accessing data at addresses S, $S+1$, $S+2$, $\cdots$, etc., or a column of data can be accessed by fetching data from the address S, $S+\Delta$, $S+2\Delta$, $S+3\Delta$, $\cdots$, where $\Delta \equiv 1 \bmod n$, and n is the number of memory modules in the address interlacing pattern. In conventional computers, rows and columns of matrices usually cannot be manipulated with equal ease when there is a potential for parallelism. In the computer organization outlined in the previous section, the rows of data can be treated in a highly parallel fashion, but columns of data essentially must be treated sequentially. With the addressing control mentioned above, columns and rows of data can be treated equally well in a parallel fashion. In fact, the operation of taking a matrix transpose in an $N \times N$ matrix can be done in N sector loads and N sector stores if there are at least N words in a sector. Most implementations of the Gauss–Jordan reduction for matrix inversion require a facility for interchanging pairs of rows and pairs of columns. The addressing control postulated here gives just that facility.

Thus far we have considered the use of a cache in conjunction with a conventional magnetic memory. It is possible that the main memory might be unconventional in some way and thereby lend even greater power to the computer system. Consider, for example, the effect of a new access mode in main memory. The new access mode allows a bit slice of data to be accessed, one bit from each word in a block of words, instead of all bits from one word. The number of bits accessed by any single module in bit-slice mode is the same as the number accessed in word mode. The bit-slice mode of access has been described in the context of a "horizontal–vertical" computer [13], and it appears to be economically feasible to modify a standard $2\frac{1}{2}$ D memory to support this mode of access [14].

With the bit-slice mode of access, a very large number of different words can be accessed simultaneously, and data from these words can be placed in a single sector of the cache. For example, let a sector contain m words, each with n bits. Now assume that we direct accesses to main memory modules in bit-slice mode, and control the accesses so that a bit slice, say the first bit, of $m \times n$ different words is placed in a sector of the cache. This extremely powerful feature allows us to manipulate, in part, data from n times as many words as could be obtained without the bit-slice mode. Mass arithmetic can be performed by obtaining successive bit slices and doing logical operations, by sector, on the data in the cache. This allows us to perform $m \times n$ simultaneous additions, for example, in a number of operations that depend linearly upon the number of bits in an operand; the only logic required is AND, OR, EXCLUSIVE OR, and NOT on corresponding bits of sectors in the cache. Similarly, many associative operations can be performed bit-sequentially as on many as $m \times n$ words of data at a time. A number of examples of algorithms for a bit-slice processing mode are given in [15].

It is possible to obtain greater parallelism if fewer than $m \times n$ words are to be manipulated. The cache can be loaded from memory in bit-slice mode so that it contains a slice of two or more bits from each word, instead of a single bit from each word. The number of different words that are accessed during a load of the cache is inversely proportional to the number of bits per word that are placed in the cache. The best bit-slice arrangement to use depends on the number of words to be manipulated and on the type of manipulation.

There are many ways in which the various cache controls can be implemented in a computer system. Detailed functional behavior of the controls can be formulated from the rough description above to help guide the implementation. The descriptions included here are sufficient for the purposes of this paper to illustrate the control and data manipulation processes in the logic-in-memory cache-organized computer.

V. Summary and Conclusions

Several facets of the cache-organized computer lend themselves naturally to a logic-in-memory implementation. It should be clear from our discussion that the resulting computer is very different from the conventional von Neumann organization, and is also remarkably different from other unconventional computers that have been proposed to perform processes that are well suited to the logic-in-memory computer. For example, compare the logic-in-memory computer with associative computers and Solomon-like computers.

It is also interesting to note that the logic-in-memory computer is general-purpose by nature, and can be justified as an economical general-purpose computer, provided that the cost of computational facilities in the cache remains small. Very much depends on the advancement of microelectronics technology, and on the development of algorithms that make good use of the functional power available in a logic-in-memory computer.

In Section III we raised the problem of evaluating the utility of a repertoire of operations. In order to evaluate the operations described in this note, or any other set of operations, one must not only determine where they might be used in existing programs, but must also project how new programs and algorithms could make use of the new instructions to perform processes that might otherwise be done differently. Even more important is

the fundamental problem of making the power of the instructions available to the user through high-level languages modeled after ALGOL or PL/I. Presently, high-level languages are ill-matched to the computers on which they are executed because the semantics of the language include processes that are inefficient to perform on computers, while the languages do not include semantics for many machine primitives such as the "execute" command, linked-list search instructions, and translation-table-oriented instruction.

Since there is currently a trend to write computer programs in high-level languages, the incorporation of powerful instructions in machine repertoires has largely failed to yield material advantages. The target language of most language translators is normally a subset of the instruction repertoire that is available to the user, and the instructions that are never used are precisely those "powerful" instructions whose use cannot be described in the high-level language. If the logic-in-memory computer—and for that matter, any computer organization with an unconventional instruction repertoire—is to become a reality, then there must be a high-level language for that computer that can describe processes in terms of the functional capabilities of the computer. This calls for a highly integrated parallel development of language and processor. In spite of our present sophistication in the design of computers and computer languages, we have barely scratched the surface in the problem of mating language with computer for conventional computers, to say nothing of the problems to be encountered in mating languages with unconventional computers.

References

[1] K. N. Levitt and W. H. Kautz, "Cellular arrays for the parallel implementation of binary error-correcting codes," *IEEE Trans. Information Theory*, vol. IT-15, pp. 597–607, September 1969.

[2] W. H. Kautz, K. N. Levitt, and A. Waksman, "Cellular interconnection arrays," *IEEE Trans. Electronic Computers*, vol. EC-17, pp. 443–451, May 1968.

[3] W. H. Kautz, "A cellular threshold array," *IEEE Trans. Electronic Computers* (Short Notes), vol. EC-16, pp. 680–682, October 1967.

[4] ——, "Cellular-logic-in-memory arrays," *IEEE Trans. Computers*, vol. C-18, pp. 719–727, August 1969.

[5] J. S. Liptay, "Structural aspects of the System/360 Model 85, Part II: The cache," *IBM Sys. J.*, vol. 7, pp. 15–21, 1968. See also, C. J. Conti, D. H. Gibson, and S. H. Pitkowsky, "Structural aspects of the System/360 Model 85, Part I: General organization," same issue, pp. 2–14.

[6] J. Fotheringham, "Dynamic storage allocation in the ATLAS computer, including use of a backing store," *Commun. ACM*, vol. 4, pp. 435–436, October 1961.

[7] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* (Short Notes), vol. EC-14, pp. 270–271, April 1965.

[8] J. Cohen, "A use of fast and slow memories in list-processing languages," *Commun. ACM*, vol. 10, pp. 82–86, February 1967.

[9] C. C. Yang and S. S. Yau, "A cutpoint cellular associative memory," *IEEE Trans. Electronic Computers*, vol. EC-15, pp. 522–529, August 1966.

[10] D. A. Savitt, H. H. Love, Jr., and R. E. Troop, "ASP: a new concept in language and machine organization," *1967 Spring Joint Computer Conf., AFIPS Proc.*, vol. 30. Washington, D. C.: Thompson, 1967, pp. 87–102.

[11] E. S. Lee, "Semiconductor circuits in associative memories," *1963 Proc. Pacific Computer Conf.*, pp. 96–108, March 1963.

[12] D. L. Slotnik, "Unconventional systems," *1967 Spring Joint Computer Conf., AFIPS Proc.* Washington, D. C.: Thompson, 1967, pp. 477–481.

[13] W. Shooman, "Parallel computing with vertical data," *1960 Fall Joint Computer Conf., AFIPS Proc.*, vol. 18. Washington, D. C.: Spartan, 1960.

[14] H. S. Stone, "Associative processing for general purpose computers through the use of modified memories," *1968 Fall Joint Computer Conf., AFIPS Proc.*, vol. 33, pt. 2. Washington, D.C.: Thompson, 1968, pp. 949–955.

[15] A. D. Falkoff, "Algorithms for parallel-search memories," *Commun. ACM*, vol. 9, pp. 488–511, October 1962.

Correspondence

A Modified Matrix Algorithm for Determining the Complete Connection Matrix of a Switching Network

***Abstract*—An efficient matrix algorithm is described which enables one to determine the complete connection matrix in only two steps.**

***Index Terms*—Boolean connection matrix, combinational switching network, product of operators.**

Let $\mathcal{C}$ be a combinational switching circuit with p nodes which is described by a Boolean matrix $A=\{a_{\alpha\beta}\}_{\alpha,\beta=1,\cdots,p}$ with $a_{\alpha\alpha}=1$, called a primitive connection matrix.

Note that in a primitive connection matrix, entry $a_{\alpha\beta}$ (for $\alpha\neq\beta$) is limited to a single bilateral element or a group of such elements in parallel.

It is shown [1], [2] that the complete connection matrix of the network is equal to the matrix $A^r=A^{r+1}=\cdots$, where $r\leq p-1$.

In the following we shall consider a more efficient matrix algorithm for determining the complete connection matrices of switching networks. The basic idea of this algorithm appeared in a previous paper by the author [3].

Manuscript received June 7, 1968; revised February 17, 1969.

Let T_{ij} be a matrix operator defined thus: $T_{ij}(A)=B$ where $B=\{b_{\alpha\beta}\}_{\alpha,\beta=1\cdots,p}$ with

$$b_{ij}=\{A^2\}_{ij}=a_{ij}\vee\bigvee_{\substack{k=1\\k\neq i,j}}^{p}a_{ik}a_{kj}$$

and $b_{\alpha\beta}=a_{\alpha\beta}$ for all $(\alpha,\beta)\neq(i,j)$.

Let us prove that $(T_{ij}(A))^{p-1}=A^{p-1}$. Obviously $A\subseteq T_{ij}(A)\subseteq A^2$, hence $A^{p-1}\subseteq(T_{ij}(A))^{p-1}\subseteq(A^2)^{p-1}=A^{p-1}$, and this implies $(T_{ij}(A))^{p-1}=A^{p-1}$. Let U be a product of T_{ij} for all pairs (i,j), where $i\neq j$ and $U_1=T_{1p}T_{1p-1}\cdots T_{12}$. We define $U_1^n(A)=U_1(U_1^{n-1}(A))$.

By recurrence we obtain $(U_1(A))^{p-1}=A^{p-1}$ and $(U(A))^{p-1}=A^{p-1}$.

For example, $(U_1(A))^{p-1}=(T_{1p}(T_{1p-1}\cdots T_{12}(A)))^{p-1}=\cdots=(T_{12}(A))^{p-1}=A^{p-1}$.

Theorem 1: $\{U_1^n(A)\}_{1\alpha}\supseteq\{A^{n+1}\}_{1\alpha}$ for all $\alpha=2,\cdots,p$ and if $U_1^r(A)=U_1^{r+1}(A)$, then the following relations hold: $\{U_1^r(A)\}_{1\alpha}=\{A^{p-1}\}_{1\alpha}$ for $\alpha=2,\cdots,p$.

Proof: If $\{U_1^n(A)\}_{1\alpha}\supseteq\{A^{n+1}\}_{1\alpha}$ for $\alpha=2,\cdots,p$, then by means of induction we obtain