

Cellular Logic-in-Memory Arrays

WILLIAM H. KAUTZ, MEMBER, IEEE

Abstract—As a direct consequence of large-scale integration, many advantages in the design, fabrication, testing, and use of digital circuitry can be achieved if the circuits can be arranged in a two-dimensional iterative, or cellular, array of identical elementary networks, or cells. When a small amount of storage is included in each cell, the same array may be regarded either as a logically enhanced memory array, or as a logic array whose elementary gates and connections can be “programmed” to realize a desired logical behavior.

In this paper the specific engineering features of such cellular logic-in-memory (CLIM) arrays are discussed, and one such special-purpose array, a cellular sorting array, is described in detail to illustrate how these features may be achieved in a particular design. It is shown how the cellular sorting array can be employed as a single-address, multiword memory that keeps in order all words stored within it. It can also be used as a content-addressed memory, a pushdown memory, a buffer memory, and (with a lower logical efficiency) a programmable array for the realization of arbitrary switching functions. A second version of a sorting array, operating on a different sorting principle, is also described.

Index Terms—Cellular logic, large-scale integration, logic arrays logic in memory, push-down memory, sorting, switching functions.

I. INTRODUCTION

PREDICTIONS for the development of integrated semiconductor electronics indicate the availability within the next decade of inexpensive and reliable circuit chips, each of which may contain hundreds or thousands of circuit elements. While many problems are unsolved regarding fabrication, testing, and interconnection of these chips, the outstanding problem is that of deciding just what kinds of large but useful networks should be designed. This may be regarded as the problem of how to decompose a large system into “modules” or “packages.” In seeking a solution to the problem, the following may be assumed.

- 1) The modules are potentially very complex.
- 2) The number of terminals per module is constrained to some value between 50 and 100.
- 3) A high premium is placed on the use of only a small number of different module types.
- 4) The modules are inherently nonrepairable, so that fault accommodation (on the chip, or in the system organization, or both) assumes a greater importance.

One aspect of this problem of “what to put on the chip,” concerns the class of applications centered around

digital computers and information handling systems. Advantages can be obtained by arranging the chip circuitry in the form of a cellular array—a two-dimensional iterative configuration of identical cells, each of which contains both logic and storage and is connected mainly to its immediate neighbors [4a]. Such an array, therefore, has the form of a memory array that is enhanced with logic at each digit position. Several features of these cellular logic-in-memory (CLIM) arrays are described. By way of an outstanding example, the balance of this paper then describes the structure, logical circuitry, operation, and use of one such CLIM array, a cellular sorting array, that has been found to be more versatile and efficient than several other types of cellular logic-in-memory arrays under study. This sorting array can be employed not only as a single-address sorting memory which keeps in order all words stored within it, but also as a pushdown memory, a buffer (queue) memory, a content-addressed memory, and even as an electronically programmable array for the realization of arbitrary combinational switching functions.

Several other types of cellular logic-in-memory arrays are described in other reports and technical papers. Some of these have already appeared (arrays for combinational logic [1]–[4], a cellular permutation array [5], [6], and a cellular threshold array [7]), and others are in preparation (improved arrays for combinational and sequential logic, arrays for the solution of problems that can be formulated in graphical terms, arrays for matrix inversion, arrays for encoding and decoding error correcting codes, an augmented content-addressed memory array, an arithmetic memory array, linear programming arrays, and others). The application of CLIM principles to the design of scratchpad memories, conventional associative memories, and programmable microprogram control arrays is straightforward, and probably not novel.

II. CELLULAR LOGIC-IN-MEMORY ARRAYS

Desirable features of the digital circuitry that is to be placed on an array or subarray are:

- 1) flexibility in function to reduce the required number of different types of subarrays,
- 2) testability,
- 3) fault accommodation,
- 4) subarray interconnectability (due to the limitations on the number of external terminals),
- 5) high logical performance (that is, a large processing or logical capability per chip),

Manuscript received September 4, 1968; revised April 8, 1969.

The research reported herein was supported by the Office of Naval Research, Information Systems Branch, under Contract Nonr-4833(00).

The author is with Stanford Research Institute, Menlo Park, Calif. 94025.

- 6) ease of logical design,
- 7) low power level and high speed, and
- 8) ease of functional decomposition of a system into chip-size "packages."

These features can be achieved simultaneously by organizing the logical circuitry on the chip in the form of a two-dimensional rectangular array of identical cells, each of which contains a relatively simple logic-and-storage circuit (10 to 50 gates). Each cell is connected only to its immediate neighbors with a small number of connections, although busing along rows and columns of the array is allowed, provided it is driven from some external source through terminals of the array, and not by internal gates. The fan-in and fan-out of each internal gate are held to small fixed values. Embellishments of this basic form of array, such as longer intercell connections, nonuniform cells, and non-rectangular (e.g., hexagonal) cell arrangements are certainly possible, but they do not appear to be necessary for most array functions.

The eight features listed above are achieved for such a CLIM array as follows.

1) *Functional flexibility*: For CLIM arrays principal function is logical, flexibility is obtained mainly by using the flip-flop storage in each cell to allow the cell logic to be "programmed" to a desired "mode" [8]. Thus, a cell that behaves as a full adder in one mode may act in other modes as a simple permutation switch, a register stage, a counter stage, or some other simple circuit. Through judicious selection of the set of cell modes that are employed in the cells of a particular array, the array as a whole can be made to exhibit a wide range of useful behavior, as each cell is programmed independently to one of its possible modes. In some types of arrays, particularly arrays whose principle function is storage, cell mode selection depends mainly upon the signals that are applied to the row and column buses through external terminals along the edges of the array.

2) *Testability*: Because of their regular structure, logical networks that are cellular should be substantially easier to test for faults and to diagnose than unstructured networks having the same number of elements. This conjecture is borne out not only by documented experience with fault detection and location procedures applied to one-dimensional iterative networks, and by theoretical results on two-dimensional arrays [9], [10], but also by experience with numerous examples of two-dimensional cellular arrays of various types and complexities [10], [11]. These examples strongly support the conjecture that test schedules for fault testing and diagnosis are both easier to derive and shorter in length for CLIM arrays than for arbitrary unstructured networks. In fact, in some of the cases studied, the length of the test schedule required for single fault detection was found to depend only upon the complexity of the cell, and not upon the number of cells in the array.

3) *Fault Accommodation*: Because of the flexibility claimed in 1), isolated faulty cells in an array may often be bypassed by programming the adjacent cells to avoid active connection to the faulty cell [8], [12]. In some cases, only one or two cells need be bypassed. In others, the entire row or column (or both) containing a faulty cell must be taken from use. In still other cases, appropriate reprogramming can utilize a cell in one mode, even though it has failed in another mode. While these techniques are not universal in either their applicability or their capability (that is, there will always be types of arrays and types of faults that are not amenable to fault avoidance), they nevertheless offer the designer an alternative to the usual requirement that all circuits employed in a network be absolutely perfect.

4) *Subarray Interconnectability*: Normally a cellular array will be realized as a macrocellular interconnection of cellular subarrays (chips), each having, say, between 20 and 200 cells. Since each cell has only a small number of connections to adjacent cells, the number of external inputs and outputs that connect to the edges of an array or subarray is relatively small. In any case, this number grows with the perimeter (that is, linearly) and not with the number of cells (quadratically) as the size of the array is increased. Typically, a 10-by-10 array, each cell of which has one horizontal bus, one vertical bus, one horizontal logic cascade, and one vertical logic cascade (such as the sorting array to be described in the next section), will have a total of about 64 terminals (including clock, reset, and power) around the periphery of the array.

5) *Logical Performance (Gate Utilization Efficiency)*: Array performance can be measured only by detailed analysis and evaluation of the capabilities of each separate array type. This is done for two arrays in the sections to follow, and for other arrays in other technical papers. Suffice it to say at present that past experience indicates that a high gate utilization efficiency is possible when realizing high speed memories of all types, as well as word-organized sequential circuitry such as registers, parallel accumulators, counters, etc. A somewhat lower efficiency results when realizing arbitrary combinational and sequential logic. Justification of this logical performance will probably become apparent only slowly, as more and more examples gradually improve the statistics. In any case, however, the cost per gate of integrated circuitry is expected to drop during the next decade.

6) *Ease of Logical Design*: Many common logical design techniques for realizing combinational and sequential behavior are still applicable when the network is constrained to be cellular. In addition, some special design procedures have been developed especially for cellular logic arrays [1]–[4]. These are even more readily programmed and applied than the familiar techniques, and some of them lead to efficient logical circuitry. Increasingly useful design approaches are still being developed.

7) *Low Power Level and High Speed*: Because of the low fan-in and fan-out of the gates within the cells, and the assumed constraints placed upon the length and number of intercell connections, individual gates in a CLIM array may operate at very low power levels and may be very small physically, without a sacrifice in the maximum achievable gate speed for a given device technology. If power drivers are needed, it is only on those signal lines that leave the subarray (chip). The logic diagrams of most arrays show logic cascades that pass successively through all cells in each row or each column. In some cases, as in a parallel adder or comparator, this cascading is unavoidable, and one must simply accept the resultant signal delays. In other cases, the cascade serves merely as a convenient diagrammatic means to buffer many signals together; the large OR gate may actually be realized electrically as many semiconductor elements feeding a single bus, so that the cell delays are not cumulative.

8) *Functional Decomposition*: The use of programmable cellular arrays frequently allows a digital system under design to be decomposed into functional blocks. Therefore, the number of interconnections required between these blocks is smaller than if a nonfunctional decomposition were used. The number of terminals per semiconductor array, while still a stringent constraint on the detailed design of the system, is less stringent than if some less structured decomposition were used.

In addition to these features, a few other advantages also accrue from the use of cellular subarrays.

9) *Insensitivity to Improvements in Device Technology*: The future availability of larger and larger subarrays does not imply a complete redesign of an array-organized digital system, but requires only that a smaller number of subarrays (chips) be used in building up full arrays.

10) *Simplified Circuit Design and Tooling*: Because each subarray consists of an iteration of a single, relatively simple cell, all design energies may be focused on the optimization of the circuitry of this cell with respect to speed, reliability, stability, required area, testing, tolerances, etc. Also, just a single set of initial masks is needed for fabrication.

11) *High Reliability*: This feature is inherent and implicit in the utilization of large-scale integrated circuitry. However, CLIM arrays have a minimum of long connections that cross other circuitry and other shorter connections, and the resultant reduction in the probability of short circuits will tend to increase the reliability and yield, if it has any effect at all upon these features.

It may be concluded from these arguments that a cellular approach to the selection of large-scale integrated modules for digital systems provides an alternative to customized arrays. This is true especially for those portions of systems that have a natural iterative structure, such as memories, registers, arithmetic units, counters, programmable microprogram arrays, (logic-

less) line switching circuits, decoders, and many special-purpose units that will undoubtedly find their place within very large multiprocessors of the future and in special-purpose computing systems. Indeed, many completely new and useful array types may be devised as the per-gate cost of arrays drops, and the pressure to replace software with hardware increases in the future. For these same reasons, the use of CLIM arrays for the realization of arbitrary combinational and sequential logic can also be expected to increase, despite its lower logical efficiency.

III. SORTING ARRAY I: BASIC OPERATION

We now describe a particular CLIM array¹ that behaves as a single-address, multiple-word memory that keeps in sorted order all data words that are fed into it. Words that are read out are obtained in order of size, with the largest word first (or alternatively, with the smallest first, as desired). The words are assumed to be of maximum length n . This memory could find use as a functional unit that can be attached to the central processor of a general-purpose computer, or to a special-purpose computing system for which sorting capability is needed. As explained later (Section V), the array may also be used for various other purposes having nothing to do with sorting.

The sorting array in question is a two-dimensional, iterative configuration of identical cells, each of which is a simple sequential digital logic circuit. Each cell is connected only to its immediate neighbors, and the cells around the edges of the array are connected to fixed signals or to registers of a conventional type that are assumed to lie outside of the array proper. Fig. 1 displays the elementary sorting array, the logic circuitry and equations of a typical cell, and the registers that would normally be used (an input-output register X , and a word selected register W). All cell input terminals on the right-hand side of the array are connected to a logical signal (z_0 , normally fixed at 1), and the outputs labeled z from the left edge of the array serve as inputs to the stages of the W -register.

It may be noted from the figure that each cell of the array contains one flip-flop, whose contents is designated y , so that the set of n flip-flops in any one of the N rows of the array may be employed to store one n -digit word. This set of words is assumed to be encoded in a uniform digit-weighted code, such as the conventional binary or binary-coded decimal number system, with the most significant digits at the left ends of the words, and a one's or two's complement representation for negative numbers, with the minus sign encoded as a 0 at the left end of the word. (The case in which sorting is performed with other representations or over only a portion of the digits in the word is treated later.) An entire

¹ A similar array based on cryogenic technology was described by Seeber [13] in 1960, but his elementary cell design is needlessly complex when converted to gate-type logic.

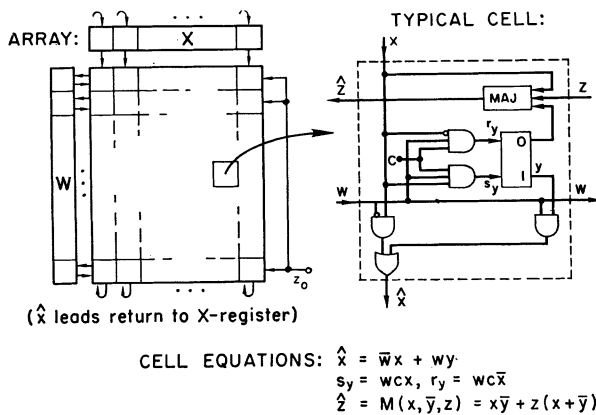


Fig. 1. Cellular sorting array I.

word is handled as a unit during the input, output, and sorting operations.

One cycle of operation of the array consists of two steps.

1) A *comparison* step, in which the word X in the X -register is simultaneously compared with all N words stored in rows of the array; a 1 is injected into the W -register in those rows whose words (including blank words) are smaller than or equal to the word X , and a 0 is injected into the W -register in those rows whose words are larger than the word X .

2) An *execution* step, in which: a) the set of all words that are stored in the subset of rows having a 1 in the W -register are collectively moved downward one row within the set, while the word in the X -register is copied into the uppermost such row; and b) the lowermost such word is copied into the X -register. Words in rows having a 0 in the W -register are not moved. Only one clock is needed, but if desired, the two substeps a) and b) can be executed with two separate clocks, so that readout may be carried out without concurrent write-in.

Sorting with this array is accomplished by maintaining a sorted file of previously entered words, with the largest at the top of the array, and the smallest and any blank rows (rows containing all 0's) at the bottom. Each new word that is to be sorted is inserted into this file in a single operational filing cycle. In step 1) every word smaller than or equal to the new word is marked with a 1 in the corresponding position of the W -register, and in step 2) all marked words are shifted down one word position, with the new word being inserted in the uppermost marked row. Unless the array is already full prior to the filing cycle, the X -register will contain all 0's at the end of the cycle. Otherwise, it will contain the smallest word in the array.

To read out in order the words in the file, largest to smallest, place a single 1 in the uppermost stage of the W -register. Now carry out step 2) of the cycle repeatedly, shifting the 1 downward in the W -register, one row with each step. If the words are desired in the opposite order, the single 1 may be started at the bottom of the W -register and shifted upward, although this

procedure will produce an initial string of all-0 words if the array is not full. (See the next section for a way to avoid this.) This sequence of operations gradually empties the file; if it is desired to merely copy the file into the output channel, without clearing it, then only step 2b) should be used.

An alternative method of readout is to enter repeatedly the number $(1\ 1\ 1\ \dots\ 1)$ from the X -register. This forces the contents of the array out of the bottom, one word at a time. This method avoids the use of the W -register as a shift register, but leaves the array full of 1's, which must then be cleared by some other means.

The detailed operation of the array during the comparison step 1) proceeds as follows. With reference to the cell circuitry and equations shown in Fig. 1, note that the majority gate at the top of the cell forms part of a chain of n such gates along each row of the array. The inputs x and \bar{y} of this gate allow it to play the role of a size comparator, so that the leftmost \hat{z} -output in each row takes on the value 1 when and only when the number represented on the set of x -lines entering this row is greater than or equal to the number represented in the cascade of y -flip-flops in the same row.

Normally, step 1) is carried out with the W -register initially empty, so that the w busses in all rows of the array carry the value 0. In this case the \hat{x} -output in each cell carries the same value as the x -input; $\hat{x} = x$. That is, the contents of the X -register is bussed downward to all rows of the array. As a result, the comparisons in step 1) are made between the word X in the X -register and every word Y stored in the array.

The detailed operation of the array during the execution step 2) proceeds as follows. Within each row for which the W -register contains a 0, we have $w = 0$, so that each cell in this row behaves according to

$$\hat{x} = x, \quad y' = y.$$

That is, the row is static, and behaves as if it were not even present. Within each row for which the W -register contains a 1, we have $w = 1$, so each cell behaves according to

$$\hat{x} = y, \quad y' = cy$$

where c is the clock. Thus, the word stored in the flip-flops in this row is transferred onto the set of \hat{x} -lines that pass downward from this row. Also, with the application of the clock, the word received on the x -lines from the row above is transferred into the flip-flops in this row. For the array as a whole, therefore, all words in the subset of rows consisting of the X -register and those rows marked with a 1 in the W -register shift down cyclically one row position within this subset. The contents of the X -register fills the top position, and the contents of the lowest marked row passes back into the X -register.

For some special uses of the array, it may be desired to use the W -register during step 1). Suppose that some one row containing word Y_i has been marked with a 1

in the W -register. The comparison process will then be modified so that all words Y in rows below the marked row will now be compared with the word Y_i in the marked row, instead of with the word X . If several rows are so marked, then each word in the array will be compared against the first marked word above it (or against the word X , if there is no marked word above it).

IV. SORTING ARRAY I: EMBELLISHMENTS AND SIMPLIFICATIONS

In most computation and data-processing problems, words are not sorted on the basis of just their relative magnitudes, but of their magnitudes taken over only a subset of the digits, usually called the *key*, of the words. That is, certain nonkey digit positions within the words carry auxiliary data (or pointers to the location of auxiliary data) that should not enter into the comparison process, but that must be retained for other computation purposes not related to the sorting. To inhibit the comparison in these digit positions, the array shown in Fig. 1 may be augmented, as depicted in Fig. 2, to include a *mask register* M , the stage outputs of which are bussed vertically along columns to all rows of the array. Each cell of the new array contains the additional gatery to inhibit the comparison operation in a column whenever $m=0$ for that column. When $m=1$, the cell behaves normally. With this arrangement, the position of the key within the data words may be selected externally by injecting a string of 1's into the proper positions in the M -register. In fact, multiple keys may be employed, and their positions need not be contiguous, but it is assumed as before that the significance of the digits in the keys increases from right to left.

A less flexible but simpler arrangement is shown in Fig. 3. Here the key and nonkey portions of the words are handled in two separate sorting arrays, the latter of which is simplified over the former by having its comparison circuitry (the majority gate and associated wiring) removed. Corresponding w -busses of the two arrays are directly connected, however, so that the two portions of each word undergo the same transfers.

When a mask register is used, it may be desirable to change to a new key for the entire set of words in an already ordered file, in order to re-sort the file on the basis of the new key. Probably the simplest way to achieve this re-sorting capability is to reserve the leftmost digit position within each word as a *tag* digit to indicate "re-sort" status. This digit will normally have the value 0. When the key is changed, the array is cycled ($N+1$) times, starting with the W -register full of 0's, but clearing it as usual at the end of each cycle. A 1 is held in the leftmost digit position in the X -register throughout the re-sorting. In this way, the file is gradually pushed out of the bottom of the array in successive cycles, and is reinserted into the top for re-sorting, one word at a time. The extra 1 causes all words in the new file to be treated as if they were larger than all words in the original file, thereby keeping the

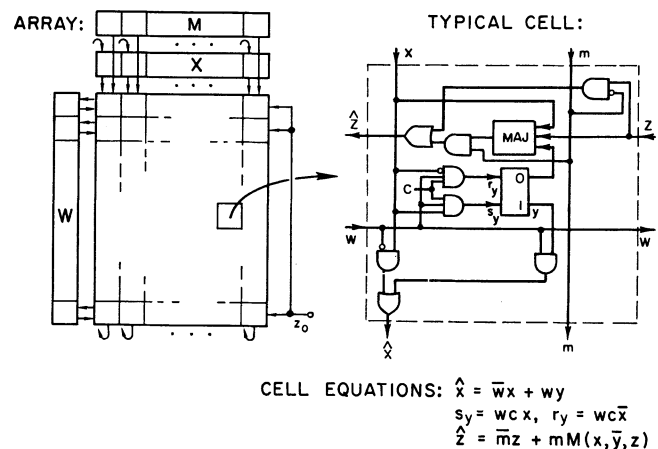


Fig. 2. Cellular sorting array I with masking.

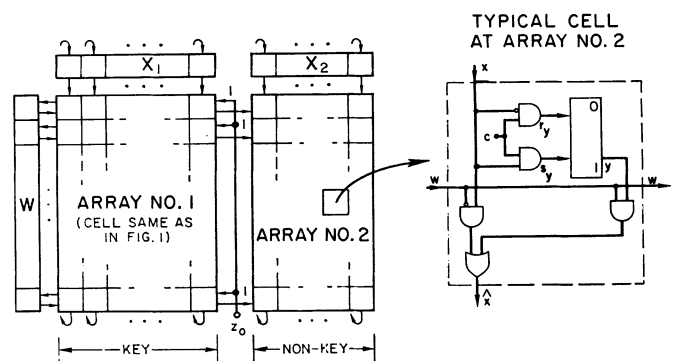


Fig. 3. Cellular sorting array I with separate nonkey array.

new file on top of the old one. The operation may be stopped as soon as the first word having an extra 1 in its most significant digit position appears at the output of the array.

If this process were going to be repeated with still another key, one could either: a) first circulate the file again for $N+1$ cycles, holding all 1's in the W -register but forcing the leftmost x -digit to be 0 before repeating the above process (in order to clear the leftmost column); or b) augment the array with a reset line attached to all flip-flops in the leftmost column, to clear this column before repeating the re-sort operation; or c) reserve a second digit position to indicate the next "re-sort" status.

If two or more words equal in magnitude are filed in the array described in the previous section, they will occupy adjacent rows in the array, with each entry located *above* all earlier equal-sized entries. If the opposite ordering of equal-sized words is desired for read-out, this may be achieved by changing the signal value on the boundary input z_0 on the right-hand edge of the array of Fig. 1 from 1 to 0. This modification causes the comparisons to be executed according to a strict inequality ($X > Y$) instead of a simple inequality ($X \geq Y$), so that a word equal in size to a previous word is treated as if it were larger rather than smaller than the previous word.

Floating-point numbers are handled with no special provisions required, provided only that the exponent is placed to the left of the mantissa, and the representation is normalized before being injected into the array proper. Negative numbers represented in a "magnitude-plus-sign" form must be complemented (in the X -register, for example) before comparison, and the sign must be moved to the left end of the word and complemented (to be 0 for a minus and 1 for a plus), if necessary.

Actually, it is sometimes possible to dispense with one or both of the X - and W -registers, depending upon the computing environment in which the sorting array is used. The X -register is used only as a buffer. If the signals and timing on the x - and \hat{x} -lines are compatible with those of the input-output channel connecting the sorting array to the rest of the digital system, then this register can be eliminated. Even during resorting, the \hat{x} -lines can be fed back directly to the x -lines. The W -register may also be eliminated by tying each \hat{z} -output directly to the x -bus in the same row. To see that this simplification is valid, recall that the effect of step 1), which is purely combinational and is unclocked, is to force w to take on the value 1 in a lower group of rows, each of whose words Y is less than or equal to the word X supplied to the top of the array. This change in w now changes the comparison in all of these rows (except the uppermost of them), so that each word Y is compared with the word immediately above it instead of with the word X . In an ordered file, however, the magnitude of the words decreases downward, so the value of \hat{z} , hence w , will never change to 1 and then back to 0 again, but in fact, will be held latched at the value 1, if it changes to 1 at all. Step 2) proceeds normally. When the input word X changes, the boundary between the 0 and 1 strings of w -values will ripple upward or downward until the insertion point for the new x -word is located. Consequently, this registerless array may operate somewhat more slowly, but still carries out the filing operation properly.

From a computing system point of view, this array is probably best treated as a single-address multiword memory having a storage capacity of N words of n -digits each, and having the property that a readout command will always retrieve the largest word in the memory. If a mask register is used, it should certainly be addressable as well. It might also be desirable to make the W -register partially addressable, so that prescribed blocks of words or individual words can be selected or inhibited during the sorting and readout operations. Even a small degree of external control of the W -register offers the possibility of employing some rather sophisticated selection criteria, such as: selection of all words whose magnitudes fall between given limits; selection of the k th largest word; selection on the basis of multiple keys, possibly improperly ordered within the words; and selection using combined inequality and equality testing.

If it is desired to use a sorting memory in which the roles of "smallest" and "largest" are interchanged, it is only necessary to modify the cell so that the equation for \hat{z} has the form:

$$\hat{z} = M(\hat{x}, y, z) = \hat{x}y + z(\hat{x} + y).$$

The cell required has substantially the same complexity as before.

If ordering based upon binary inclusion ($X \subseteq Y$) rather than size comparison is desired, only the second term in the above expression should be used—a rather special case since binary inclusion normally generates only a partial ordering.

V. OTHER USES OF SORTING ARRAY I

A. Switching Function Realization

This CLIM array may also be used for the realization of an arbitrary switching function of up to n variables if the W -register at the left side of the array is replaced by a single column of exclusive-OR gates, as shown in Fig. 4(a) or by some serial equivalent. In this role the words stored in memory locations within the array are used in programmed-logic fashion to set up the switching function to be realized. The write-in capability, including the w -busses and the clock, are then used only to program the array initially to the desired function, and the readout capability can be used for verifying the contents of the memory. The independent variables x_1, x_2, \dots, x_n of the switching function are entered into the top of the array, the right-edge boundary input z_0 is set at 1, and the output $f(x_1, x_2, \dots, x_n)$ is taken from the bottom of the exclusive-OR-gate chain, as shown in Fig. 4(a).

The words to be stored in the rows of the array (in any order) are the binary-number representations of those row-numbers of the function's truth table at which the function changes value [2]. For example, to realize the function whose truth table is

x_3	x_2	x_1	f	
0	0	0	0	
0	0	1	1	row 1
0	1	0	1	
0	1	1	0	row 3
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	row 7,

the rows of the array should store as binary words the row numbers

001
011
111

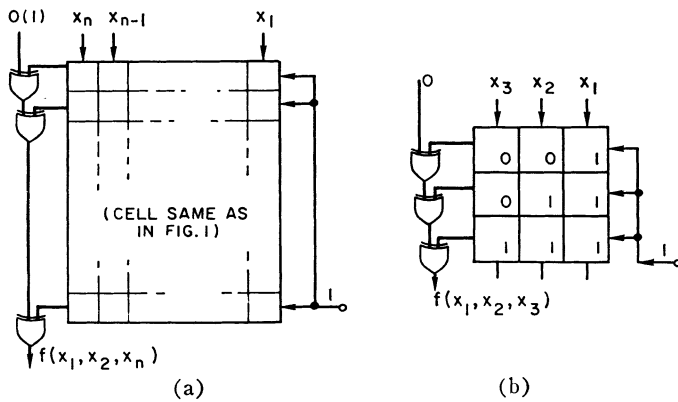


Fig. 4. Use of sorting array I for the realization of a switching function.

as depicted in Fig. 4(b).

The justification of this method is found in the fact that each row-function q_1, q_2, \dots realized in the sorting array, when expressed in truth-table form, consists of a string of 1's below a string of 0's. The row function q_k (that is, \hat{z}) equals 1 when and only when the binary number $X = (x_n, \dots, x_2, x_1)$ is equal to or greater than the binary number k (cf. Y) stored in that row. Thus, the transition from 1's to 0's occurs in row k of the truth table:

x_3	x_2	x_1	q_1	q_3	q_7	$f = q_1 \oplus q_3 \oplus q_7$
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	1	0	0	1
0	1	1	1	1	0	0
1	0	0	1	1	0	0
1	0	1	1	1	0	0
1	1	0	1	1	0	0
1	1	1	1	1	1	1.

The order in which the words are stored in rows of the array is arbitrary since the exclusive-OR operation is commutative. If the words are ordered with the largest at the top, however, the simpler downward-going left-edge logic $\hat{u} = \hat{z} + \bar{u}$ may be used instead of $\hat{u} = \hat{z} \oplus u$, if desired.

If $f(0, 0, \dots, 0) = 1$ instead of 0, the function \bar{f} should be realized instead of f , and a 1 injected into the top of the exclusive-OR-gate chain.

The number N of rows that are needed in the array is equal to one less than the total number of 0-strings and 1-strings in the f -column of the truth table of the desired function. As shown by Elspas and Short [14], a least upper bound for all functions of n variable is given by

$$N \lesssim 2 \cdot 2^{n/3},$$

when permutations and complementations of the input variables are permitted.

Actually, several other special types of arrays are known that can realize switching functions in a much more efficient manner than that illustrated for the sorting array [1], [2].

B. Pushdown and Buffer Memories

In another mode of operation, this sorting array may be employed as either a pushdown memory (last-in, first-out) or a buffer memory (first-in, first-out). This is done by allowing the W -register to have bidirectional shifting capabilities, so that it may serve as an address register for the set of words stored in the array. In this mode the array itself is employed only as a bank of memory registers having downward shifting capability, and the comparison logic in the cells is not used. A single 1 is held in the W -register to mark the row into which the next word should be entered. For use as a pushdown memory, the W -register is shifted one row downward (initially in row 1) after each entry and one row upward before each readout. For use as a buffer memory, the W -register is shifted one row upward (initially in row N) after each entry and one row downward after each readout, and in addition, its contents are complemented both before and after each readout. (In this way the entire contents of the array are shifted downward.) Several other configurations are possible for simulating these memory operations, but this one appears to be the simplest.

C. Content-Addressed Memory

Finally, the sorting array may also be used as a content-addressed (associative) memory. For inequality searching ($X \geq Y$ or $X > Y$, the selection between these choices depending upon the value of z_0), step 1) is carried out as first described. This leaves 1's in the W -register in just those rows containing words Y that satisfy the inequality. This entire subset of words may now be shifted out of the array by executing step 2) repeatedly with the input register empty until an all-0's word is encountered on the output lines from the array.²

If the circuitry shown in Fig. 1 is modified so that the \hat{z} -line inputs to the stages of the W -register are arranged to enter the *trigger* inputs rather than the *set* inputs of these flip-flops, then the associative search may be carried out on the basis of certain additional searching conditions without giving up any of the capabilities discussed so far. For example, assuming that the words in the array are in order, a search for all numbers Y in the range $X_1 \leq Y \leq X_2$ can be conducted as follows.

- 1) Load X_1 into the X register.
- 2) Compare $X_1 > Y$ (i.e., execute step 1) with $z_0 = 0$), apply clock to W -register.
- 3) Load X_2 into the X register.

² For nondestructive readout, start with the X -register full of 0's, but cycle back to the X -register the words read out, for reentry into the array. Stop when the all-0's word reappears in the X -register.

4) Compare $X_2 \geq Y$ (execute step 1) with $z_0 = 1$), and apply clock to W -register.

As a result, the W -register will contain a 1 in every row whose word Y satisfies *one* of the two tests, but not both (since two \hat{z} -signals will return the W flip-flop to its initial state). This can happen only when $X_1 \leq Y \leq X_2$. When $X_1 = X_2$, this constitutes an equality test, and step 3) is unnecessary. Readout of the selected words is conducted as before.

These three supplementary uses of the sorting array are offered merely as examples. Many other interesting uses can undoubtedly be devised by the ingenious designer. In particular, the multiple-comparison feature described at the end of Section III in conjunction with special control of the W and X registers could lead to procedures for handling very complex testing conditions involving several inequalities and equalities.

VI. SORTING ARRAY II

There also exist other possible configurations for cellular sorting arrays based upon different sorting algorithms [15]–[17], but the one described so far appears to be the simplest and most versatile. The array form and cell circuitry of the runner-up are shown in Fig. 5.

The cells of this sorting array are arranged in a "brick-wall" pattern, and the data words to be sorted are stored along rows, each semicell of which contains a one-bit data flip-flop. Very briefly, sorting is accomplished by shifting synchronously the entire bank of words to the left, making serial comparisons (most significant digit first) in each cell to determine whether or not the pair of words entering each cell should be row-interchanged as it passes through that cell. With the first shift, for example, the comparison is started in column 1 between each word in an odd-numbered row and the word just below it. For n -digit words (the width of the array is n), this comparison will last for n clock times (shifts), but as soon as the first digit has been shifted out of column 1, the second comparison may start in column 2 between each word in an odd-numbered row and the word just above it. While these two comparisons are still in progress, the third comparison may be started at the third clock time in column 3 between each odd-numbered word and the word below it, as in column 1, and so on. It is not difficult to show that, using only this elementary operation of successively interchanging the positions of words in adjacent rows on the basis of their relative sizes, a total of exactly N word comparisons ($N+n$ digit shifts) is necessary and sufficient to place into order a store of N arbitrary words. Thus, after $\langle N/n \rangle + 1$ complete word cyclings, the entire set of words in the memory is back in its proper digit phase, and is completely sorted.

The individual cell of this array has the form of a two-input serial sorter combined with the two single

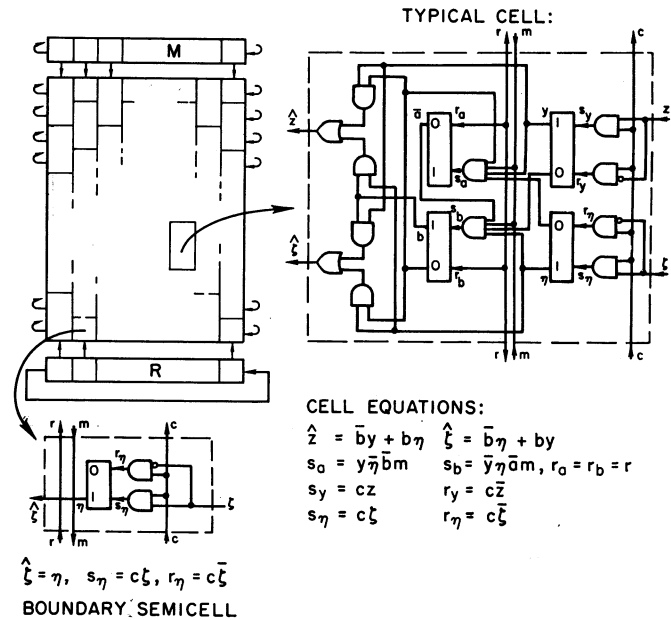


Fig. 5. Cellular sorting array II.

stages y and η of the shifting registers. Serial sorting of two words U and V requires the storage of three states per cell, and these states are represented in the two flip-flops a and b :

$a=0, b=0$: $U=V$ so far (that is, corresponding digits u and v are equal, $u=v$, in every digit encountered so far);

$a=1, b=0$: $U>V$ (that is, some digit position has been encountered for which $u=1, v=0$);

$a=0, b=1$: $U<V$ (that is, some digit position has been encountered for which $u=0, v=1$).

As soon as either the a or b flip-flop sets, setting of the other is blocked, inhibiting further digit comparisons. The R -register at the bottom of the array contains a single shifting 1, which resets at each clock the a and b flip-flops in that column that contains the ends of the words, so that serial comparison may recommence at the beginnings of the words at the next clock time. The mask register M at the top also shifts in synchronism with the data. It serves to inhibit comparisons in all columns not corresponding to the key, which is assumed for this type of sorting array to be located entirely to the left of the nonkey portion of the data word. The mechanism of filling and emptying the memory is not specifically shown in Fig. 5. Input and output can be accommodated by breaking into the data transfer lines or storage flip-flops at any convenient set of points, for example, at the register stages at the left or right side of the array.

This array has a cell complexity (per data digit) of about 15 elementary NOR gates per semicell (assuming static flip-flops), the same as for a single cell of the first sorting array (masking included). The number of external terminals per subarray is also the same. How-

ever, the versatility of Sorting Array I is completely absent in Sorting Array II, which operates more slowly and requires more precise clocking than does Sorting Array I.

VII. CONCLUSIONS

Cellular logic-in-memory (CLIM) arrays offer an attractive alternative to customized arrays in many applications of large-scale integration to the design of digital information-handling systems. At the very least, they offer many advantages to the designer of conventional and associative memories, and other computer circuitry that already has a natural iterative structure. At best, CLIM arrays can be used for the realization of arbitrary combinational and sequential logic circuitry. As the per-gate costs of integrated circuitry drop and the pressures for more special-purpose hardware within a processor increase, the advantages of a CLIM approach to logical design should become more attractive.

The cellular sorting arrays described in this paper are examples of CLIM arrays that achieve the advantages cited for the main purpose for which they were conceived (i.e., sorting). At the same time, the first sorting array has the flexibility to carry out certain secondary functions, albeit with a lesser gate utilization efficiency. Limited fault accommodation is also possible: except for certain short circuits, the row containing a defective cell may be taken from use by holding its w -bus at 0, and a defective column may be avoided by suitable connection of the array proper to the X -register. The array is readily tested by available testing techniques for combinational cellular arrays. (The flip-flops in the array are regarded merely as extra inputs since they may be directly set and reset by external signals.) The horizontal and vertical logical cascades involve only two elementary gates per cell, thereby introducing minimal delay in a situation where the cascades cannot be avoided if sorting is to be achieved at all. The other advantages listed in Section II are inherent to the cellular approach and are achieved automatically for the sorting array.

ACKNOWLEDGMENT

The author would like to express his appreciation to Dr. R. C. Singleton and Dr. M. C. Pease, III, of Stanford Research Institute, for their participation in an early survey of sorting methods to determine which might be realizable in cellular form, and to Dr. H. S. Stone for his comments on this paper.

REFERENCES

- [1] R. C. Minnick and R. A. Short, "Cellular linear-input logic," Stanford Research Institute, Menlo Park, Calif., Final Rept., Contract AF 19(628)-498, SRI Project 4122, February 1964.
- [2] R. C. Minnick, J. Goldberg, M. W. Green, W. H. Kautz, R. A. Short, H. S. Stone, and M. Yoeli, "Cellular arrays for logic and storage," Stanford Research Institute, Menlo Park, Calif., Final Rept., Contract AF 19(628)-4233, SRI Project 5087, April 1966.
- [3] R. C. Minnick, "Cutpoint cellular logic," *IEEE Trans. Electronic Computers*, vol. EC-13, pp. 685-698, December 1964.
- [4] —, "Cobweb cellular arrays," *1965 Fall Joint Computer Conf., AFIPS Proc.*, vol. 27, pt. 1. Washington, D. C.: Spartan, 1965, pp. 327-341.
- [4a] —, "Survey of microcellular research," *J. ACM*, vol. 14, no. 2, pp. 203-241, April 1967.
- [5] W. H. Kautz, K. N. Levitt, and A. Waksman, "Cellular interconnection arrays," *IEEE Trans. Computers*, vol. C-17, pp. 443-445, May 1968.
- [6] A. Waksman, "A permutation network," *J. ACM*, vol. 15, pp. 159-163, January 1968.
- [7] W. H. Kautz, "A cellular threshold array," *IEEE Trans. Electronic Computers (Short Notes)*, vol. EC-16, pp. 680-682, October 1967.
- [8] S. E. Wahlstrom, "Programmable arrays and networks," *Electronics*, pp. 91-95 December 11, 1967.
- [9] W. H. Kautz, "Fault testing and diagnosis in combinational digital circuits," *Proc. 1st Ann. IEEE Computer Conf.*, Chicago, Ill., September 6-8, 1967. See also *IEEE Trans. Computers*, vol. C-17, pp. 352-366, April 1968.
- [10] —, "Testing for faults in combinational cellular logic arrays," *Proc. 8th Ann. Symp. on Switching and Automata Theory*, Austin, Tex., October 1967, pp. 161-174.
- [11] J. Goldberg, M. W. Green, W. H. Kautz, K. N. Levitt, and J. B. Turner, "Techniques for the realization of ultra-reliable spaceborne computers," Interim Scientific Rept. III, Contract NAS12-33, SRI Project 5580, Stanford Research Institute, Menlo Park, Calif., June 1968.
- [12] R. C. Minnick [2], p. 82.
- [13] R. R. Seeber, "Associative self-sorting memory," *Proc. 1960 Eastern Joint Computer Conf.*, vol. 18, pp. 179-188.
- [14] B. Elspas and R. A. Short, "A bound on the run measure of switching functions," *IEEE Trans. Electronic Computers*, vol. EC-13, pp. 1-4, February 1964.
- [15] D. L. Shell, "A high-speed sorting procedure," *Commun. ACM*, vol. 2, no. 7, pp. 30-32, 1959.
- [16] R. C. Bose and R. J. Nelson, "A sorting problem," *J. ACM*, vol. 9, no. 2, pp. 282-296, 1962.
- [17] K. E. Batchner, "Sorting networks and their applications," *1968 Spring Joint Computer Conf., AFIPS Proc.*, vol. 32. Washington, D. C.: Thompson, 1968, pp. 307-314.