H. Fleisher
L. I. Maissel

An Introduction to Array Logic

Abstract: After a discussion of the reasons for choosing to implement logic in array form, a detailed description of the nature of array logic is given. Topics specifically discussed include general array structures and implementation, influence of decoder partitioning, design of logic arrays, output phase, "split" variables, feedback in logic arrays, and reconfiguration.

Introduction

Although there will always be a need for logic which offers the best possible performance regardless of cost, most logic necessarily falls into either the cost/performance class or the low cost/marginal performance class. Thus, an important aspect of the logic designer's job is his constant search for the best possible compromise between performance and unit cost. The kind of curve which determines the designer's decision is illustrated in Fig. 1. Two important quantities are shown; B, the base or threshold cost that shifts the curve to the right or left, and C, the knee of the curve.

In the past, when logic circuits were built of discrete components, or even from small unit logic blocks such as NAND or NOR gates, the knee of the curve was fairly easy to determine. Cost depended directly on the number of components used and, as it turned out, an average of three to six units of delay also coincided with a minimum number of circuits. Fewer levels of delay implied a significant increase in cost whereas adding delay gave little or no cost improvement. The base of the curve, B, was also fairly well defined, being determined by such things as total time spent in design and, to a lesser extent, by the number of different applications for a given cluster of logic. Assembly costs were essentially a function of the number of circuits, whereas the individual components cost the same in whatever circuits they happened to be used.

With the advent of large scale integration (LSI) the factors that determine this cost/performance curve have begun to change drastically. To begin with, cost must be considered in terms of total silicon area as opposed to the number of devices enclosed within that area. This fact determines the shape of the curve in Fig. 1 but not its displacement B. The latter is affected by a host of quantities including manual design time, automated design costs (DA), etc. In particular, the larger the number of chips actually manufactured for any given design, the smaller B becomes. In addition, once chips start to come off the line, we must consider the cost of testing them. Finally, as chips get larger their yield goes down (unless there is some kind of redundancy scheme), and the possibility of failure in subsequent life also increases with chip complexity.

The net results of the comments above are that in today's world we can get good value for our expenditures only if we make large quantities of any given chip. In practice, relatively few chips see the kind of widespread use that is necessary to make their cost really competitive. This arises because of engineering changes that are associated with design errors as well as with changes in philosophy as a machine is developed. The problem extends well past the release date of the machine because of late improvements, specific customer requirements, changes in philosophy, etc. This will be particularly true in the future with the coming emphasis on the translation of much of what is today's software into firmware and the replacement of some of our current firmware with hardware.

A major attempt to minimize this problem is the so-called "master slice" approach. In this scheme no chip is completely unique until after final metallization. Thus, the chip consists of an assemblage of unit-logic devices positioned where they are anticipated to be useful to each other. The final circuit is given its own specific

"personality" (or pattern of interconnections) via the top level of metallization. Such an approach is necessarily wasteful since it is impossible to optimize the placement of the unit-logic devices to suit all possible future circuits. This statement is supported by the fact that there are many master slice types available. These slices, of course, are then individually personalized into many more chip types.

Is there, then, any situation in which a single chip, once designed, receives extended usage? The answer is 'yes' if we consider semiconductor memory technology. It is further worth noting that several features that are associated with large chips designed for memory are absent in random logic chips of comparable size. These include easy testability and redundancy. One therefore wonders whether some of the distinct advantages that memory technology enjoys over logic technology might not be applied to the latter. The intent to do this is a strong motivation for array logic.

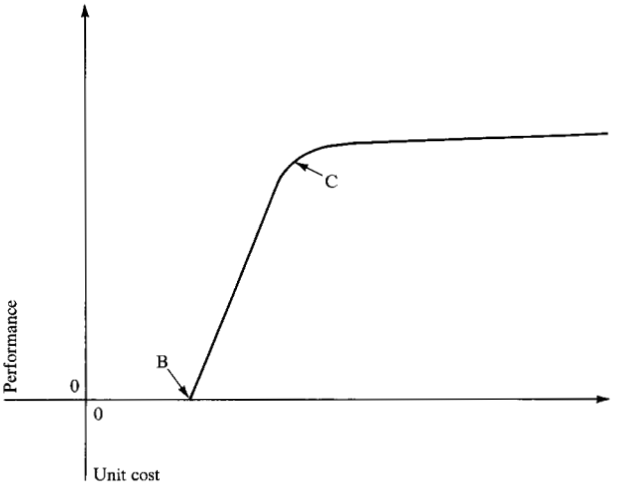


Figure 1 Cost-performance curve for logic circuits.

Array logic

In this paper "array logic" is defined as the use of memory-like structures for performing logic. Array logic should not be confused with cellular or iterative arrays [1] which have a closer structural relation to master-slice approaches embodying random logic.

Array logic functions by presenting the bits in the data path to the memory-like structures as "address bits." Decoding of this address starts the process whereby a pre-determined result is extracted from the array. Because the functions generated by such an array depend only on the "personality" (i.e., bit content) of the array, logic can, in principle, be changed at the same rate at which memory can be written. The idea of performing logic in this manner is by no means new. Small lookup tables can be found in many of today's machines and, in fact, the IBM 1620 worked in this fashion. We should also note that microcode, which has now become almost universal, is a form of lookup table, replacing conventional logic. The switch to microcode was dictated by considerations of cost rather than performance, as well as by the high frequency of change involved in the development of control logic. The latter is reflected in the growing popularity of writable control stores.

Array logic is embodied in a memory-like structure wherein the variables are used as address bits and the output bits that have been selected are combined to yield a single bit which is the value of the function stored in the array for the particular combination of input bits.

An associatively addressed memory array called Functional Memory [2] has been proposed as a substitute for conventional logic. However, this array had several deficiencies that become apparent later in this paper.

We now review some of the reasons why array logic was, until recently, insufficiently attractive to be widely adopted and discuss what has changed so as to make this picture rather different today.

Past problems

1. Arrays were too large to be competitive. With relatively small chips available, the amount of logic that could be placed on a single array chip was unacceptably small. This problem was aggravated by the fact that techniques for minimizing the size of such arrays were still undeveloped.

2. Array logic was unfamiliar to many logic designers. Aside from the normal conservatism with which any new technology is inevitably greeted, array logic could not offer the tremendous improvement in readability which made microcode so attractive as a replacement for hard-wired control logic. If anything, many designers feared that arrays might be more difficult to interpret than conventional logic.

3. Array logic was poorly matched with available device technologies. As already stated, array logic is a memory-like technology. However, technologies designed and optimized for memory purposes are not necessarily optimized for logic. Thus, for example, writing speed is equally as important as reading speed for memory; but for logic, writing is relatively infrequent compared with reading. Furthermore, the latter must be as fast as possible whereas writing speed need not be fast at all. In addition, some arrays are commercially available today in read-only technologies [3].

4. Although the logic performed by an array logic chip could easily be modified, this advantage was often offset

by an associated change in the connections between chips, sometimes called a "yellow wire" change. It is of relatively little value to be able easily to change the function of the array logic chip if this now requires us to make a time consuming and costly "yellow wire" change.

Why array logic makes sense today

1. The engineering change problem is going to get worse. As the amount of logic per chip increases, it is self-evident that the chance of an engineering change being associated with that particular chip is going to increase. As the complexity of machines increases and, as already noted earlier, more and more function is placed in hardware, it is likely that one might not wish to commit all hardware in a permanent manner. A good analogy to this is the current emphasis on writable control storage for microcode. Although the control store is not changed very often, it is changed often enough to make the rewrite feature advisable.

2. Design techniques for array logic have improved significantly. Computation of the personality for an array logic chip by conventional methods requires a high degree of skill and is very difficult for large arrays that handle complex functions. Fortunately, a number of new design aids are now available to ease this burden. These are discussed in a later section of this paper and have been considered in more detail by Hong, Cain and Ostapko [4].

3. Improved device technologies. Higher densities are currently achievable in semiconductor technologies ranging from read-only FET to high-speed read/write bipolar. Read-mostly technologies are approaching feasibility and there are exploratory efforts on the horizon that will make array logic highly competitive, because it provides low power and minimum space.

4. Additional interconnection flexibility between arrays is now possible. This technique will be discussed later.

5. There is evidence that designing in array logic, using the design aids mentioned in 2 above, materially shortens the design cycle. The impact here is clearly economic as well as providing more flexibility to the designer.

Figure 2 Truth table for the two-bit adder.

Truth table

A	B	C	D	K	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

$$\begin{array}{ll} AB & \text{Addend} \\ +CD & \text{Augend} \\ \hline KS_1S_0 & \text{Sum} \end{array}$$

Design of logic arrays

We use the two-bit adder (Fig. 2) as a vehicle to aid us in the following discussion. The two-bit adder accepts four input variables A, B, C and D and generates three outputs S_0, S_1 and K, which are the zero-order and first-order sums and a carry, respectively. Figure 2 shows the truth table for this piece of logic. One way to implement this would be in a conventional random access memory. If the memory were interrogated by some combination of four bits, the latter information would filter through a four-bit decoder which would then "find" the corresponding output line in the truth table. This type of table is limited to small problems because the number of rows is 2^N where N is the number of input variables. For as few as 16 inputs we would need in excess of 65 000 rows to contain the full problem.

Each term of the input table is referred to as a *minterm*, i.e., an AND function (product term) of all the variables. The Boolean equation for each output can be obtained as the OR of those minterms opposite which there is a one in the output table. Thus the Boolean expression for the carry, for example, contains the minterms 0111, 1010, 1011, 1101, 1110, and 1111 which are equivalent to $\bar{A}BCD$, $A\bar{B}C\bar{D}$, $A\bar{B}CD$, $AB\bar{C}D$, $ABC\bar{D}$, and $ABCD$.

After the usual algebraic manipulations of Boolean algebra, the three equations for zero- and first-order sums (S_0 and S_1) and for the carry (K) may be written as

$$S_0 = B\bar{D} \vee \bar{B}D \tag{1}$$

$$S_1 = \bar{A}\bar{B}C \vee A\bar{B}\bar{C} \vee \bar{A}C\bar{D} \vee A\bar{C}\bar{D} \vee \bar{A}B\bar{C}D \vee ABCD; \tag{2}$$

$$K = AC \vee ABD \vee BCD, \tag{3}$$

where the symbol $\vee$ represents the OR operation.

The same information that is stored as shown in Fig. 2 can be stored and accessed in a somewhat different manner. Instead of having one single decoder where four inputs expand to 16 rows, we could use four one-input decoders. Such decoders are sometimes called phase splitters or true-complement generators. They are illus-

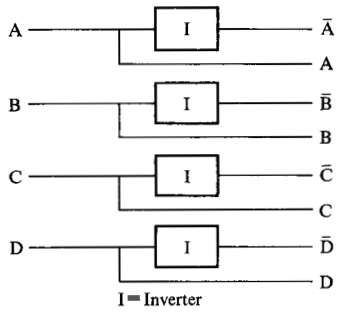


Figure 3 Four one-input decoders.

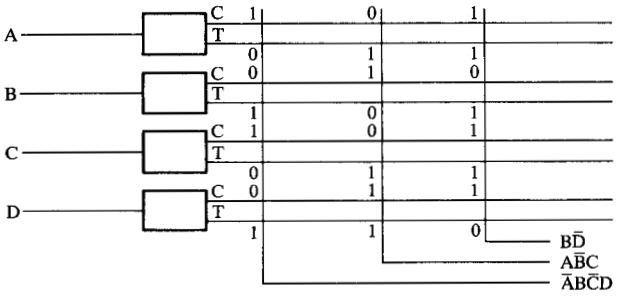


Figure 4 Some functions that can be generated in a single word by one-input decoders.

trated in Fig. 3, which shows that the number of rows needed for four inputs has been reduced from 16 to eight. It is clear that as the number of input variables grows larger, the saving in terms of number of rows becomes proportionately greater; as compared with 2^N rows for a conventionally addressed table, this structure requires $2N$ rows. For example, for a 16-input problem the number of rows would be reduced from over 65 000 to only 32.

However, there is a penalty for the tremendous compression obtained in the number of rows. Whereas a single decoder that accepts all inputs can generate any function of the input variables in a single column of the truth table, the one input decoder scheme generates only simple products of the input variables in a single column. Some examples are shown in Fig. 4. The function represented by any given column is obtained by selecting the appropriate output from each decoder and then ANDing all of these together. (The output of a decoder line is a 1 if the line is ON *and* the personality stored there is a 1.) For this reason we call this the AND array. We also note that the personality associated with any given variable can be $\binom{0}{1}$, $\binom{1}{0}$ or $\binom{1}{1}$. $\binom{0}{0}$ is not normally used, but it can be interpreted as an "inhibit" value assigned to the variable, because it forces that column to be 0 independently of the other variables. In the case where the personality (or state) is $\binom{1}{1}$ we have a DON'T CARE situation for that particular variable. This is functionally equivalent to eliminating the variable in a Boolean simplification operation because, with personality $\binom{1}{1}$ stored, the variable will not influence the AND operation for that column. Similarly, $\binom{0}{1}$ corresponds to $x = 1$, and $\binom{1}{0}$ corresponds to $x = 0$.

The neutral word "column" was used in the paragraph above to identify the group of bits containing the logic function. Note that in the case of a conventional truth-table the "column" of the text is mapped into a bit column of a word-addressed memory. On the other hand, the "column" of the partitioned array, regardless of the partitioning, is mapped into a word column, by extension of the use of "word" in identifying the readout of an associatively addressed memory.

Most functions are represented by the OR of several product terms, so the one-input decoder array requires, in general, a second array that ORs the various product terms together to obtain the desired function. This second array is referred to as an OR-array or OR-box.

Bearing this structure in mind, we can illustrate the "mapping" of a two-bit adder into a table based on this structure. This is shown in Fig. 5. To simplify the representation of information in the AND box, we adopt the following formalism: a 0 corresponds to the $\binom{1}{0}$ state

Figure 5 Mapping the two-bit adder into an array addressed through one-input decoders.

	In↓								*Out*↑			*In*↓			
	A		*B*		*C*		*D*								
	$\overline{A}$	*A*	$\overline{B}$	*B*	$\overline{C}$	*C*	$\overline{D}$	*D*	*K*	S_1	S_0	*A*	*B*	*C*	*D*
$B\overline{D}$	1	1	0	1	1	1	1	0			1		1		0
$\overline{B}D$	1	1	1	0	1	1	0	1			1		0		1
$\overline{A}B\overline{C}$	1	0	1	0	0	1	1	1		1		0	0	1	
$AB\overline{C}$	0	1	1	0	1	0	1	1		1		1	0	0	
$\overline{A}C\overline{D}$	1	0	1	1	0	1	1	0		1		0		1	0
$A\overline{C}D$	0	1	1	1	1	0	1	0		1		1		0	0
$\overline{A}B\overline{C}D$	1	0	0	1	1	0	0	1		1		0	1	0	1
$ABCD$	0	1	0	1	0	1	0	1		1		1	1	1	1
AC	0	1	1	1	0	1	1	1	1			1		1	
$AB\overline{D}$	0	1	0	1	1	1	0	1	1			1	1		1
$BC\overline{D}$	1	1	0	1	0	1	0	1	1				1	1	1
	Actual bit personality of the AND array											Equivalent representation of the bit pattern shown at the left			

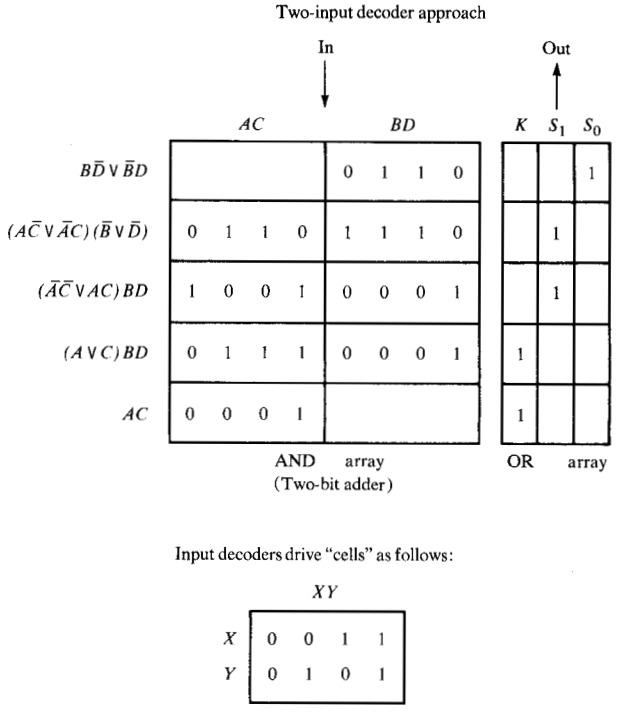


Figure 6 Mapping the two-bit adder into an array addressed through two two-input decoders.

in Fig. 4, a 1 corresponds to the $\binom{0}{1}$ state, and a blank (in the AND array) corresponds to the $\binom{1}{1}$ or DON'T CARE state. Note that the $\binom{0}{0}$ state here is assigned no meaning because the output of the word concerned would always be zero. The 1's stored in the OR-box show which product terms stored in the AND-array are to be ORed to generate the required functions. Note that a blank in the OR-box does *not* mean DON'T CARE. Instead, it means that no connection is made from the corresponding AND gate to the OR gate.

Ignoring for the moment the OR-box (on the right in Fig. 4), we note that each term in Eqs. (1) – (3) that defines the three output functions of the two-bit adder has been generated in an AND array table of 88 bits. If we now add the 33 bits of the OR-box we have a total of 121 bits, compared to only 48 bits that were needed to do the job in the conventional table lookup scheme shown in Fig. 2. Even so, this way of storing the information does have two important advantages. First, the totality of one-input decoders is considerably cheaper than a single large decoder. This cost spread becomes greater with a larger number of input variables. Second, inspection of Fig. 5 also shows that the original Boolean expressions which were used to generate the array personality are easily read out again. In other words, the personality for such an array is readily generated by a simple one-to-one translation of the Boolean expression directly into the table. The advantages of this may not be apparent in the current example, but for large problems (i.e., many variables) this is important because the generation of the full truth table is something which would be very difficult for the logic designer to do. These reasons explain why one-input decoder arrays of this type are becoming increasingly popular, several versions being commercially available [3].

• *Partitioning of inputs*

Returning now to the Boolean equations for the two-bit adder, we observe that Eqs. (2) and (3) can be factored as shown below.

$$S_1 = (A\bar{C} \vee \bar{A}C) \cdot (\bar{B} \vee \bar{D}) \vee (\bar{A}\bar{C} \vee AC) \cdot BD \qquad (4)$$

$$K = AC \vee (A \vee C) \cdot BD \qquad (5)$$

Equations (1), (4), and (5) show that we have been able to express each product term as products of functions of the variables A and C only or of variables B and D only.

This observation suggests that we might be able to derive some benefit by using only two decoders, rather than four, and feed variables A and C into one decoder and variables $B$and D into the other [5]. Using a format similar to that of Fig. 5, we map the factored Boolean expressions of Eqs. (1), (4), and (5) into the table shown in Fig. 6. (A blank here represents a DON'T CARE or 1111.) As in Fig. 5, we AND the personality bits selected by each decoder; these results in turn are presented to the OR-box. It is apparent that by using the two-input decoders (as opposed to the one-input decoders) the total number of decoder outputs has not been changed, but the number of words (rows in this example) has been reduced from 11 to five. Thus, including the OR-box, the total bit count has been reduced to 55, an amount nearly comparable to the table lookup case but still with the advantage of lower cost decoders.

The question arises as to whether the saving achieved here through the use of two-input decoders is universal or merely a special case for this particular example. Might not one-bit decoders prove to be superior in other cases? This is best answered by reference to Fig. 7 which compares the maximum and minimum number of bits needed to implement *any* function of 16 variables as a function of the bits per decoder or, conversely, the number of decoders. The OR-box is not included in the count. For simplicity, we consider only symmetric partitioning, although in principle there is no reason why the decoders should all be of equal size.

The maximum and the minimum number of bits needed to express any function addressed through a single decoder (table lookup case) are the same and are equal

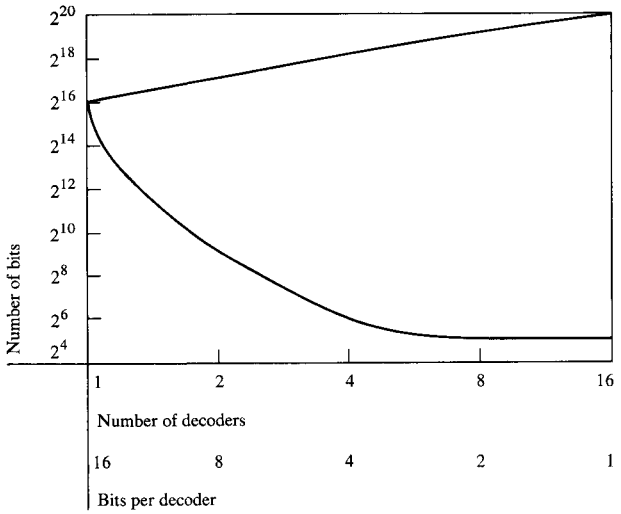


Figure 7 Maximum and minimum number of bits needed to implement any function of 16 variables as a function of decoder size.

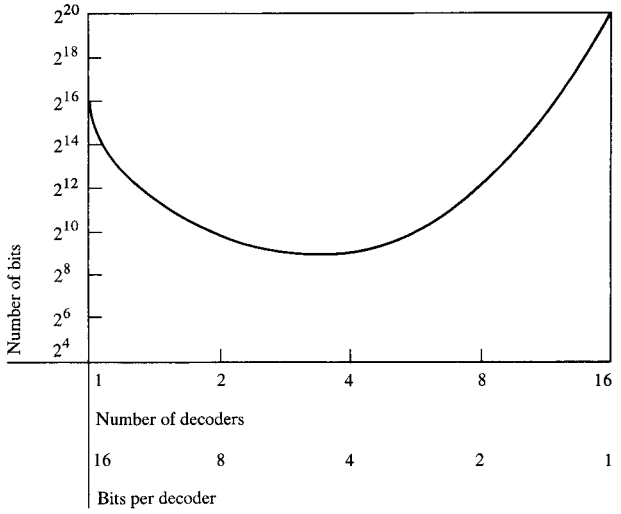


Figure 8 Number of bits needed to implement the EXCLUSIVE OR of 16 variables as a function of number of bits per decoder.

to 2^{16}. For any other partition, however, the discrepancy between the minimum and maximum number of possible bits can be very great. We also note that the maximum possible is 2^{20} ($\approx 10^6$) for the one-input decoder case whereas the minimum, $2^5 = 32$, *is the same for both the two-input and one-input cases.* (See Appendix A.) This latter observation is quite general so it follows that we can never lose (in terms of bit count) by going to two-input decoders and may, of course, gain significantly. The identity in minimum bit count for one-bit and two-bit decoders results from the fact that two one-bit decoders provide the same number of output lines as one two-bit decoder.

Thus, for many functions, the number of bits required for their implementation varies drastically as a function of the partitioning that is used. An example of this is the EXCLUSIVE OR whose dependence on the choice of decoder partitioning is illustrated in Fig. 8. This particular example was chosen because the numbers concerned are readily calculable from first principles. (See Appendix B.) For this particular case a minimum bit count is obtained when four decoders of four inputs each are used. The choice of partitioning can be critical because, in this example, the result varied from a minimum of 2^9 bits, through 2^{16} bits for table lookup, to 2^{20} bits for the case of one input per decoder.

Other factors influencing minimization of arrays

• *Choice of correct output phase*

Let us return for a moment to the Boolean expressions for the two-bit adder. The carry K and the first order sum S_1 were given in Eqs. (3) and (4), respectively. One form of the complement of the expression for K is

$$\overline{K} = (A\overline{C} \vee \overline{A}C) \cdot (\overline{B} \vee \overline{D}) \vee \overline{A}\overline{C}. \qquad (6)$$

It is immediately apparent that, by choosing the complement of the carry, we can express it so that it contains a term which also appears in the expression for the first order sum.

Thus, still using the partition $A, C-B, D$, we can map the two-bit adder into the array as shown in Fig. 9. This reduces the total bit count needed to implement a two-bit adder to 44, a figure that is actually less than for the table lookup case; in addition, we have the saving in the cost of the decoders.

• *Sharing of output columns*

We note in Fig. 9 that a row of the OR-BOX contains more than a single 1. Thus, in this example, the third word, representing the function $(A\overline{C} \vee \overline{A}C) \cdot (\overline{B} \vee \overline{D})$, is shared by two outputs, the carry complement and the first order sum. In general, any minimization technique must seek to take advantage of the possible sharing of words between outputs [6]. This implies that we cannot afford to generate personality separately for each output but must consider the total problem for all the outputs at one time. The minimization techniques to which we refer later are all of this type.

• *Split variables*

To this point we have assumed that each variable appears at one and only one decoder, although we have not restricted the size of these decoders. It is, however, also

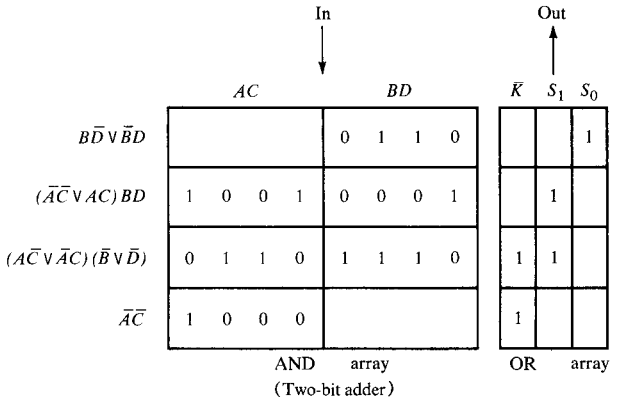


Figure 9 Mapping the two-bit adder into an array addressed through two two-input decoders but with the carry term complemented.

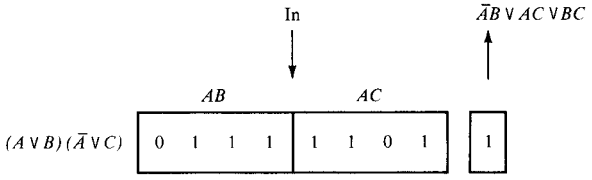


Figure 10 Minimization of a three-variable function by increasing effective number of variables to four, one of which appears twice.

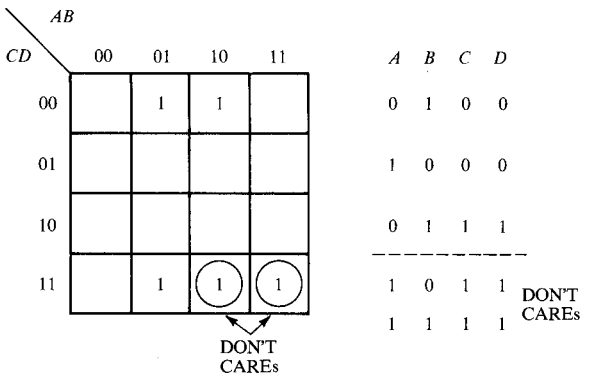
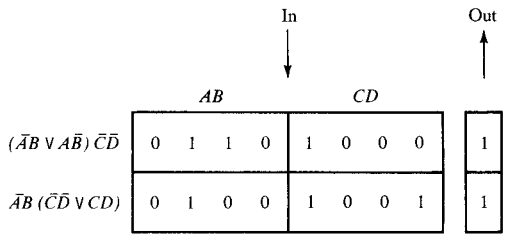


Figure 11 A function of four-variables containing three min-terms and two DON'T CARE min-terms.

Figure 12 Array implementation of function in Fig. 11 using only the min-terms.



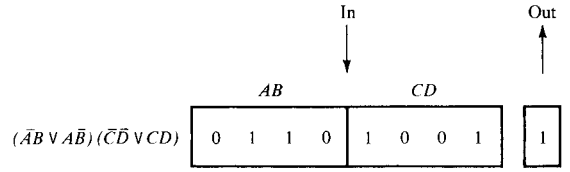


Figure 13 Array implementation of the function in Fig. 11 including one of the DON'T CARE min-terms.

possible that the same variable may be used as input to more than one decoder. It is not often that the addition of another variable to the problem actually leads to a smaller result, but such situations can arise. For example, consider the expression

$$\bar{A}B \vee AC \vee BC$$

which, by suitable factoring, becomes

$$(A \vee B) \cdot (\bar{A} \vee C).$$

By using two decoders, one accepting inputs A, B and the other accepting inputs A, C, this function maps into a single word (as shown in Fig. 10) for a bit count in the AND array of eight. Any other arrangement of decoders (e.g. AB, C) requires at least two words to implement the function for a bit count of at least 12.

The reason the split variable method can be powerful is that it creates a large number of DON'T CARE minterms. These are minterms about which we don't care whether or not they appear in any particular output. These minterms DON'T CARE are not to be confused with input variable DON'T CARE. As an example of the latter, in a function of A, B, and C, if we have the term AB, we say that C is a DON'T CARE since the function does not depend on the value of C.

• *Maximum use of minterm* DON'T CAREs

Usually when an engineer is formulating his logic design he writes down those input conditions for which certain outputs must be at 1 and it is tacitly assumed that for all other input conditions the outputs must be at 0. However, more detailed analysis can often show that there are many input conditions which are really DON'T CARES for certain outputs and the more of these that can be listed by the engineer, the better will be the minimization that can be obtained. The significance of these DON'T CARE conditions is that the minterms to which they correspond need not be accounted for when a solution to the problem is being generated. On the other hand, the availability of a DON'T CARE may make it possible to generate a word that would otherwise not be possible. We illustrate this in Fig. 11 where we show a function of four variables, partitioned AB, CD, which comprises three minterms and two DON'T CARE minterms. In the

absence of the DON'T CARES we would need two words to express the function as shown in Fig. 12, whereas by choosing to use one of the DON'T CARES, we are now able to generate the one-word solution shown in Fig. 13.

Physical implementation of logic arrays

In terms of what has been said so far, a logic array requires, in addition to the decoding function, a means for storing bit personality, a means for ANDing decoder outputs within each word and, finally, a means for ORing the results of these AND operations. This simplest way of implementing the AND and OR operations is shown in Fig. 14, where the bit personality is symbolically shown as opens and shorts (zeros and ones, respectively). Signals from various decoder lines of different partitions are thus driven down to the inverters (I) that lie just before the OR-box. A single short (logical one) is sufficient to power a word line and, because the signal is then inverted, this is seen to be logically equivalent to ANDing the complements of the outputs from the individual decoders (DeMorgan's rule).

For a read-only store the switches are readily implemented almost as shown in Fig. 14 but for read/write logic the ideal element to perform the job of these switches (directly as shown) is not yet available. The next best thing is a small memory cell which serves to latch each intersection to either 0 or 1. By also placing an AND gate at each intersection the same net effect as the switch and diode pairs of Fig. 14 is achieved. This device is illustrated in Fig. 15. An approximation to the ideal switch that is available at this time is found in MNOS technology where the switch could be replaced by the channel region of the MNOS device, as shown in Fig. 16.

In all cases, the additional circuitry needed to change the switches whenever desired adds to the total circuitry on the chip. We may also observe that in certain technologies (notably FET) a zero (short) may be a preferred state for the device because no voltage appears across the gate under this condition, whereas a device in the one (open) state has the potential for failure in the future. Hence minimization programs designed to give the maximum number of zeros in the minimum solution are desirable, and such techniques have been developed [4]. Conversely, in other technologies we might want just the reverse, and this can be done as well. For example, in an ideal technology such as that shown in Fig. 14, we still have to face the problem of loading in the various lines as current is drawn through them. In such a case the lower the number of ones stored in the array, the less will be the loading and hence the greater the speed. We may also note that techniques for doing dot ANDing and thus eliminating the need for an inverter step are also possible.

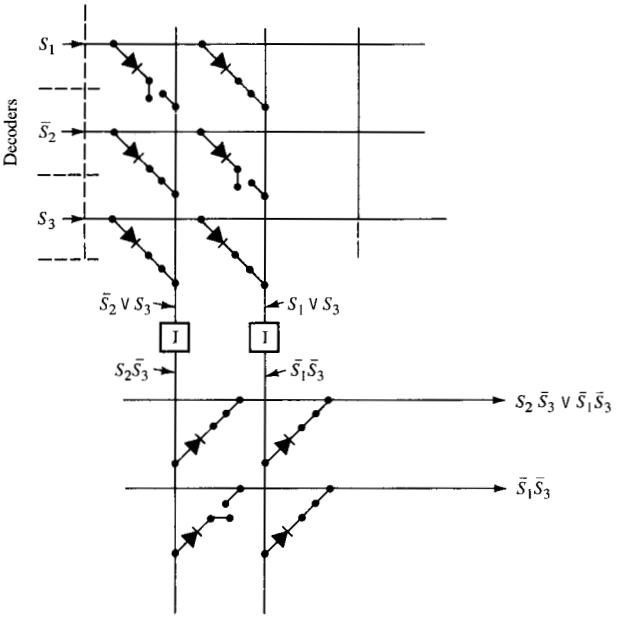


Figure 14 Idealized implementation of the AND and OR operations in a logic array.

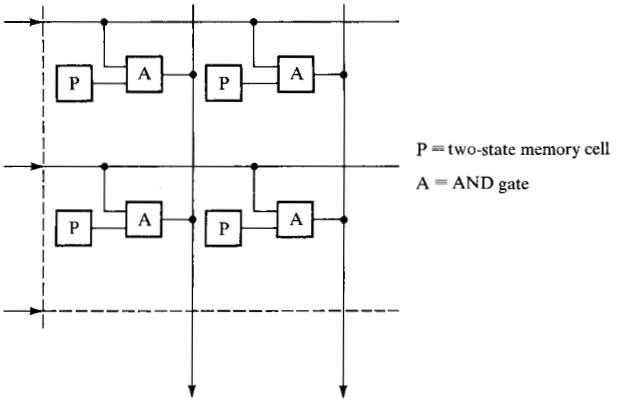


Figure 15 Use of small memory cells and an AND gate at each intersection to implement a logic array.

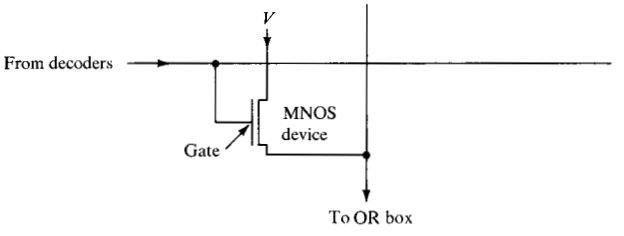


Figure 16 Use of MNOS devices to implement array logic.

Other possible candidates for array logic are CMOS (Complementary MOS) and chalcogenide-type bistable resistors.

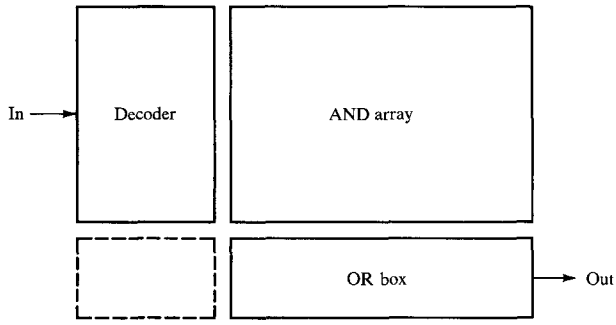


Figure 17 The three basic "boxes" of a logic array.

Personality changes and their effect on interactions between chips

To recapitulate, a logic array consists of three main sections as shown in Fig. 17. These are the decoder, the AND array and the OR-box. Up to this point we have assumed that decoding is done with conventional circuits. For example, each variable, as it comes in, is split into true and complement forms. The signals from these lines are then combined through a group of AND gates, providing an output line for each possible combination of values of the input variables.

There are several disadvantages to a conventional (fixed) decoder of this type:

1. As we have already noted, optimization of a given function or set of functions depends on the correct choice of partitioning. If, because of the fixed decoder, the partitioning is pre-committed, the number of functions that a given chip can be made to store may be limited.
2. When one logic array drives several others, the variables emerging from one array may be in a different order (permutation) from that required to input a second or third array. In this case, extra hardware must be interposed to obtain flexibility of interconnection.
3. If a failure should occur inside the decoder itself, the entire logic array is useless.
4. It is clear that if we have spare words present in the AND and OR arrays, we can deactivate a word line that contains bad bits and substitute one of the spares. However, this procedure would be wasteful if the bits were to lie along the same decoder line. We would then have to disable one word for every bad bit. It would be preferable if we could also have a few spare decoder lines. This option is not possible with a fixed decoder.

To eliminate these disadvantages it has been proposed to replace the conventional (fixed) decoder with an array decoder of the type shown in Fig. 18, building it from the same technology as the principal arrays themselves [7]. In the example shown in Fig. 18 the three variables A, B, C, are partitioned into (A, B) and C. Because it has been constructed from an array, this decoder can have several spare lines and the decoder outputs can be generated in any arbitrary order. For example, by simple personalization change we can interchange the $A\overline{B}$ and the $\overline{C}$ output lines without disturbing the order, A, B, C, in which the inputs enter the decoder.

Major disadvantages of this type of decoder might be that it is slower than a conventional decoder and it might use more space on the chip. However, as shown in Fig. 19, with the array decoder one could drive many other array chips without worrying about any changes in permutation since the variables coming in to a chip will always be in the same order. Furthermore, through the OR-box, we can permute the emerging variables into the same order.

By repersonalizing the decoder, we could, if we chose, use it to implement a conventional random access memory or an associative memory. Thus, in a system containing a large number of array chips where we might want some local storage, we would be able to use array logic technology directly to make these small local memories.

Feedback in logic arrays

Most pieces of logic involve considerable feedback, either within or between various blocks of combinational logic. In the former case, a considerable saving in chip pads can be achieved if the feedback is performed on the chip itself. In the most general case we would like to have the freedom to feed any signal from the OR-box back into the decoder. This can be done through a "feedback box" which is simply another OR-box located in the empty quadrant that can be seen in Fig. 17. To avoid race conditions, a register is placed between the OR-box and the feedback box to allow the gating of signals. A possible chip layout of this type is shown in Fig. 20.

In some commercially available array logic chips, feedback on the chip is hard-wired so that some of the outputs are permanently committed to being fed back. Also, hardware exists where, instead of an intervening register, each output that is fed back goes through a four-state (J-K) flip-flop. This arrangement has the advantage of storing the state of each output line rather than requiring it to be regenerated on each cycle, but it is somewhat more expensive.

Reconfiguration

We have thus far argued for array logic on the basis of significantly lower cost as a trade-off for a moderate drop in performance. There is, also, a unique feature of

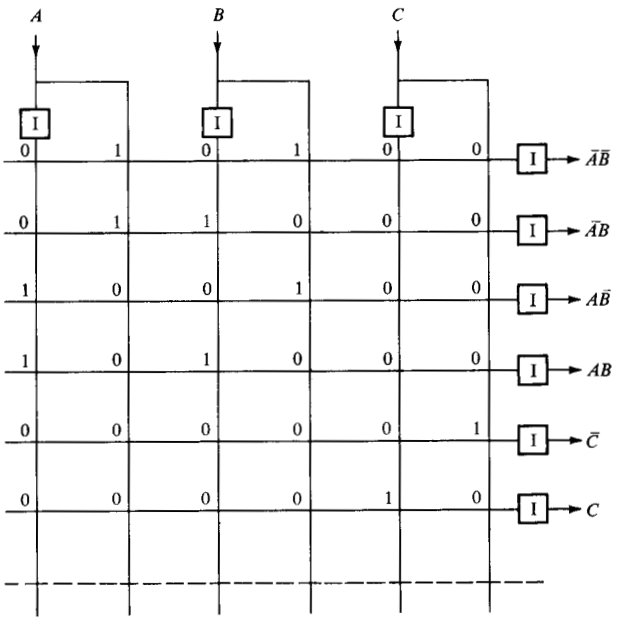


Figure 18 Personalized decoder.

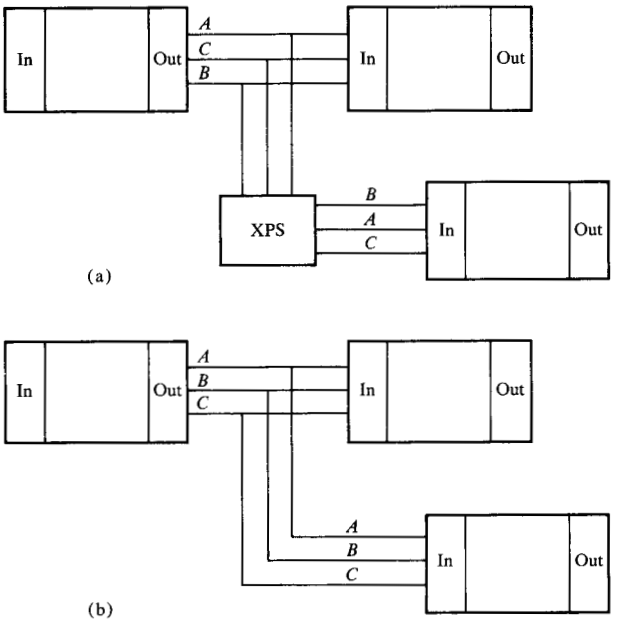


Figure 19 Array logic chips with personalized decoders can drive several other similar chips without requiring an intermediate cross-point switch. (a) Conventional arrangement; (b) same logic with personalized decoders.

array logic in that it can be changed *in situ*. Thus, in theory, we could store the personalities for a large number of logic arrays on a suitable medium (such as, for example, a Direct Access Storage Device (DASD)) and, as needed, write these into a machine that has been built with a relatively small number of chips.

Such an idea is reminiscent of the cache in a two-level memory, but we must bear in mind that the usefulness of the paging concept, which the memory cache embodies, depends on any given page being resident in the cache for time periods that are on the average substantially longer than the time needed to write a page into the cache. At the present state of the computer art it would be difficult to fulfill the equivalent condition for stored or virtual logic. However, we should bear in mind that many operations are currently performed in a highly sequential manner using a relatively small number of logic blocks. The latter are used in virtually all operations that the machine performs and it is only the sequencing that changes (under microprogram control) from one operation to the next.

As improvements in LSI allow us to use increasingly larger chips, it will become possible to reduce the extent to which various operations have to be performed as a series of small steps. This, in turn, will reduce the extent to which the same logic blocks can be shared by different sequences and virtual logic will provide a viable alternative, particularly under multiprogramming or time-sharing conditions where time spent in reconfiguring will not mean time lost to the CPU.

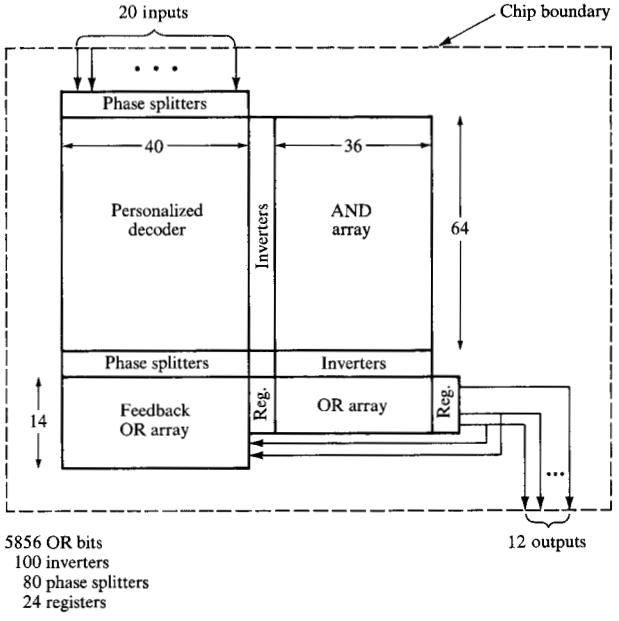


Figure 20 A chip layout with built-in, but alterable, feed-back.

Appendix A: Calculation of minimum and maximum number of bits needed to implement functions of 16 variables as a function of decoder size

The plots of minimum and maximum numbers of bits in the AND array needed to implement a function are illustrated in Fig. 7 for 16 variables, and show extreme

Table 1 Total number of bits needed to implement a 16-input EXCLUSIVE OR.

No. of decoders	*Inputs per decoder*	*Total number of bits in* AND *array*
1	16	$2^{16} = 65{,}536$
2	8	$2^{10} = 1024$
4	4	$2^{9} = 512$
8	2	$2^{12} = 4096$
16	1	$2^{20} = 1{,}048{,}576$

cases. That is, some functions of 16 variables will map into the minimum curve, some functions of 16 variables will map into the maximum curve, and the remainder will map into the space between the min- and max-curves.

Because we have restricted ourselves to symmetric partitioning, it is clear that if we have n variables and p parts (decoders), we will have $v = n/p$ variables per part (decoder). The number of output lines from each decoder then becomes $2^v = 2^{n/p}$. Therefore, the number of bits per column, $B_{\min}$, becomes

$$B_{\min} = p \cdot 2^v = p2^{n/p}.$$

Tabulating $B_{\min}$ for $n = 16$, and p ranging from $p = 1$ to $p = 16$, we have

P	$B_{\min}$
1	2^{16}
2	2^{9}
4	2^{6}
8	2^{5}
16	2^{5}

$B_{\min}$, plotted as a function of p, is shown as the lower curve of Fig. 7.

To obtain the plot of the maximum number of bits required to map a function, we must characterize the so-called "worst case" functions. Such functions, when mapped into an array of a given partitioning will require the maximum number of columns. These columns must have a different bit pattern in each partition when compared with bit patterns in the same partition for all other columns needed to implement the function.

For example, consider 16 variables partitioned into two groups of eight variables each. In this case, $p = 2$, and $B_{\min} = 2^9$. We can visualize any function of 16 variables being mapped into a Karnaugh map of 256 rows and 256 columns, with a "worst case" function being identified as having a different bit pattern in each of the 256 columns and rows of the Karnaugh map. Since each column of the Karnaugh map maps into a column of the AND array, it is clear that a "worst case" function of 16 variables, partitioned into $p = 2$, will require 256 columns at 512 bits per column. This may be written in power of two notation as $2^8 \times 2^9 = 2^1 \cdot 2^{16}$.

We continue this analysis for $p = 4$, 8, and 16, and write the general expression:

$$B_{\max} = p \cdot 2^n.$$

This is tabulated below for $n = 16$, and plotted as the upper curve in Fig. 7.

P	$B_{\max}$
1	2^{16}
2	2^{17}
4	2^{18}
8	2^{19}
16	2^{20}

Appendix B: Calculation of the number of bits needed to implement the EXCLUSIVE OR function of 16 variables as a function of decoder size

The EXCLUSIVE OR function may be stated as follows: The function has value 1 *iff* an odd number of inputs each have value 1. Because this must be true independently of partitioning, we must examine what effect, if any, partitioning the variables will have on mapping the EXCLUSIVE OR into an array.

For $p = 1$ (16 inputs into a single decoder), the EXCLUSIVE OR maps into a single bit column of 2^{16} ($\approx$ 65K) bits. For $p = 16$ (1 input per decoder), the number of columns needed is 2^{15}, since there are 2^{15} minterms in the canonical formulation of the 16 variable EXCLUSIVE OR. For this part, each column contains 2^5 bits of storage, so the total number of bits needed for mapping the function in this array is $2^5 \times 2^{15} = 2^{20}$ bits.

It is clear that the EXCLUSIVE OR function is a "worst case" function for one input per decoder partitioning.

When we examine the other parts (eight inputs per decoder, four inputs per decoder, and two inputs per decoder), we note that we can use the structure of the function to determine its mapping into these arrays.

Thus for eight inputs per decoder, we need two decoders for the 16 inputs. Because the function is symmetric, the inputs may be arbitrarily grouped: $(x_1 \cdots x_8)$, $(x_9 \cdots x_{16})$. We can then write the EXCLUSIVE OR of 16 variables as $\mathrm{XOR}(16) = \mathrm{XOR}(x_1 \cdots x_8) \cdot \overline{\mathrm{XOR}(x_9 \cdots x_{16})} \vee \overline{\mathrm{XOR}(x_1 \cdots x_8)} \cdot \mathrm{XOR}(x_9 \cdots x_{16})$. Each half of the right hand side of this equation is mapped into a single column

of the two-decoder driven array, and the function is obtained by ORing the two outputs. Since each column contains $512 = 2^9$ bits, the total number of bits needed is $2 \cdot 2^9 = 2^{10} = 1024$ bits. The argument used above can be readily extended to the other array configurations; for the four decoder driven array, the number of columns required is eight. Since there are $64 = 2^6$ bits per column, the total number of bits is $2^3 \times 2^6 = 2^9 = 512$. For the eight decoder driven array, we need $128 = 2^7$ columns. With 2^5 bits per column, we get $2^5 \cdot 2^7 = 2^{12} = 4096$ bits as the total number of bits. The results are tabulated in Table 1.

References

1. W. H. Kautz, "Programmable Cell Logic," *Recent Developments in Switching Theory*, Academic Press, Inc., New York, 1971, Chap. IX.
2. P. L. Gardner, "Functional Memory and its Microprogramming Implications," *Technical Report TR 12.091*, IBM Corporation, Poughkeepsie, New York, 1970.
3. W. N. Carr and J. P. Mize, *MOS/LSI Design and Applications*, Texas Instruments Electronics Series, McGraw-Hill Book Company, Inc., New York, 1972.
4. S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," *IBM J. Res. Develop.* **18,** 443 (1974).
5. H. Fleisher, A. Weinberger, and V. Winkler, "The Writable Personalized Chip," *Comput. Des.* **9,** No. 6, 59 (1970) and H. Fleisher, A. Weinberger, and V. Winkler, "Partitioning Logic Operations in a Generalized Matrix System," U.S. Patent #3,593,317, July 13, 1971.
6. H. Fleisher and L. I. Maissel, "Reconfigurable Machine," *IBM Technical Disclosure Bulletin*, IBM Corporation, Armonk, New York, March 1974.
7. S. Y. H. Su and D. L. Dietmeyer, "Computer Reduction of Two-level, Multiple Output Switching Circuits," *IEEE Trans. Comput.* **C-18,** 58 (1969).

Received January 17, 1974

The authors are located at the IBM System Products Division Laboratory, Poughkeepsie, New York 12602.