ITERATIVE DECOMPOSITION OF DIGITAL SYSTEMS AND ITS APPLICATIONS

V. DVORAK*

SCS-TR-90
MAY 1986

School of Computer Science
Carleton University
Ottawa, Ontario
KIS 5B6

*Department of Computer Science, Technical University of Brno,
61266 Brno, Bozetechova 2, Czechoslovakia

+This work was completed while the author was visiting the School of Computer Science, Carleton University, Ottawa, Canada. K1S 5B6

ITERATIVE DECOMPOSITION OF DIGITAL SYSTEMS AND ITS APPLICATIONS

ITERATIVE DECOMPOSITION OF DIGITAL SYSTEMS AND ITS APPLICATIONS

Vaclav Dvorak
Department of Computer Science
Technical University of Brno
612 66 Brno, Bozetechova 2, Czechoslovakia

Abstract

This report presents a new technique for synthesis of multiple-output combinational circuits as well as sequential circuits based on successive transformations of the given function table and on iterative decomposition. The resulting networks have a form of a cascade of combinational subcircuits (e.g. ROM's, PLA's or gate networks) or a time-varying sequential circuit. The former network may find applications in VLSI circuit design and in systolic arrays, whereas the later network in sequential controllers (including the microprocessor-based controllers) and as the processor in SIMD arrays and pipelined structures. The new method of synthesis gives a more general insight into a variety of implementations of combinational as well as sequential functions encountered in the logic design area.

1. INTRODUCTION

Design efforts to implement large digital systems in a form of regular arrays of cells started some twenty years ago [1] and continue until now. The research in this field has been motivated by such design features as simplicity of interconnections, modularity, testability, etc. As yet, programmable logic arrays (PLA's), read-only memories (ROM's), gate arrays and specialized iterative arrays of identical cells (binary/BCD converters, sorting networks, multipliers , dividers etc. [2]) are the most frequently used structures. A recent renewed interest in cellular arrays is expected to lead to results useful in design of VLSI circuits, systolic arrays,

parallel SIMD systems, pipelined structures and the like.

The aim of this report is to contribute to cellular array synthesis. It deals with the synthesis of one-dimensional array of uniformly interconnected cells. The cascade synthesis enables complex combinational and sequential functions to be decomposed either in time or space into more simple and manageable functions and simultaneously obtain a lower cost of system implementation measured by the chip area or memory capacity in bits. Individual cells in the cascade perform generally different functions and are easily realized in a form of PROM, PLA or using logic gates and larger modules.

Essentially the synthesis approach taken here is based on the iterative decomposition of the given function. The advantage of this approach is that we stay at the level of global description of partial functions by function tables and do not go down to logic equations and terms. Implementation of cell functions may be decided upon later on as well as transformation to the best implementation-dependent form. Further reduction of hardware can be obtained if we use one adjustable cell as a sequential processing unit (processor). Speed of processing is then , of course, reduced significantly.

In what follows, we shall introduce a figure of merit of cascade realization in terms of equivalent ROM capacity in bits, present theorems relevant to the cascade synthesis and illustrate the method by numerous examples.

2. A FIGURE OF MERIT OF COMBINATIONAL CASCADES

Let us consider a cascade implementation of a logic system at Fig.1a shown at Fig.1b. It is the k-rail cascade of B cells, each cell with k horizontal inputs and m vertical inputs being described by the function

$$H_i : \{0,1\}^{k+m} \dashrightarrow \{0,1\}^k \quad i = 1,2,\ldots,B-1$$
$$H_B : \{0,1\}^{k+m} \dashrightarrow \{0,1\}^r \qquad (1)$$

Such a cascade will be referred to as the k x m-cascade. It can be thought

of also as 1 x 1-cascade with K-valued and M-valued signals where k $= \lceil \log_2 K \rceil$, m = $\lceil \log_2 M \rceil$. Generally parameters k and m need not

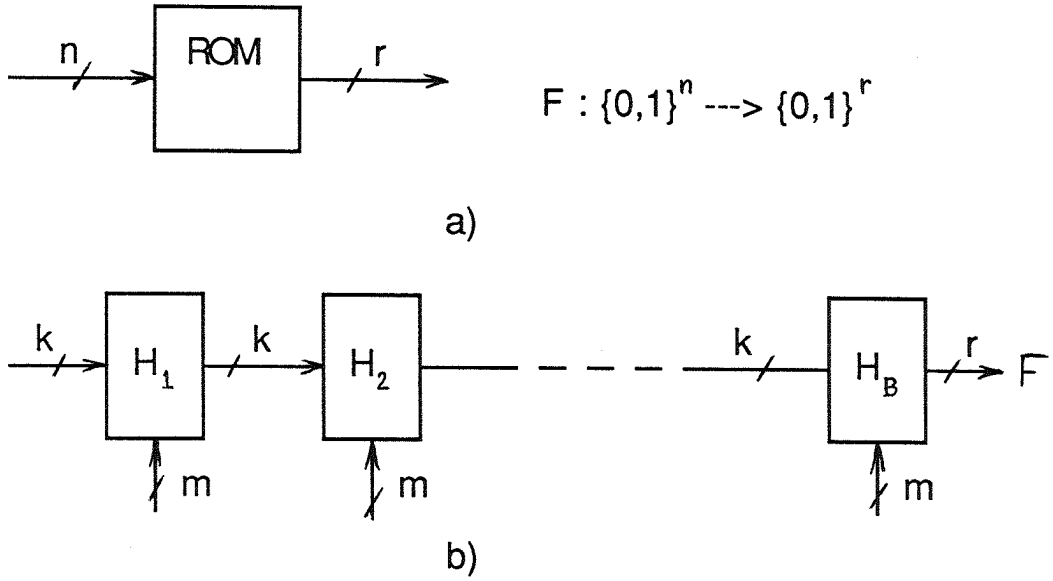


Fig.1. Implementation of r Boolean functions of n variables
a) ROM implementation b) k x m cellular cascade

be constant, they can vary along the cascade. Some cells, or in the extreme case all the cells in the cascade may be identical. Since we are going to investigate the nontrivial case B > 1, the condition

$$n > k + m \tag{2}$$

must hold. If all the cells in the cascade have disjoint groups of vertical inputs, we call the cascade nonredundant, otherwise if some binary signals enter more than one cell at vertical inputs, the cascade is redundant.

The most simple figure of merit of different cascades implementing the same logic system is a total number of bits of ROM's realizing the cell functions. The original ROM at Fig.1a requires r. 2^n bits, whereas the i-th cell, i < B, requires k . 2^{k+m} bits and the B-th cell r.2^{k+m} bits, so that we can realize saving if

$$r.\ 2^{n} \ > \ (B-1)\ k\ 2^{k+m} \ + r.\ 2^{k+m}. \qquad (3)$$

The cascades which satisfy inequality (3) will be referred to as cost-effective cascades. In the above comparison of ROM capacity we have made an implicit assumption that the cost of connections is negligible.

For a nonredundant cascade $B = \lceil (n-k)/m \rceil$, so that saving can be realized if

$$r > k\ [\ \lceil (n-k)/m \rceil - 1\ ]\ /\ (2^{n-k-m} - 1) \qquad (4)$$

Parameter limits of cost-effective nonredundant cascades for m=1 are shown at Fig.2. It is seen, that if $r \leq k$ the saving starts at as many as 6

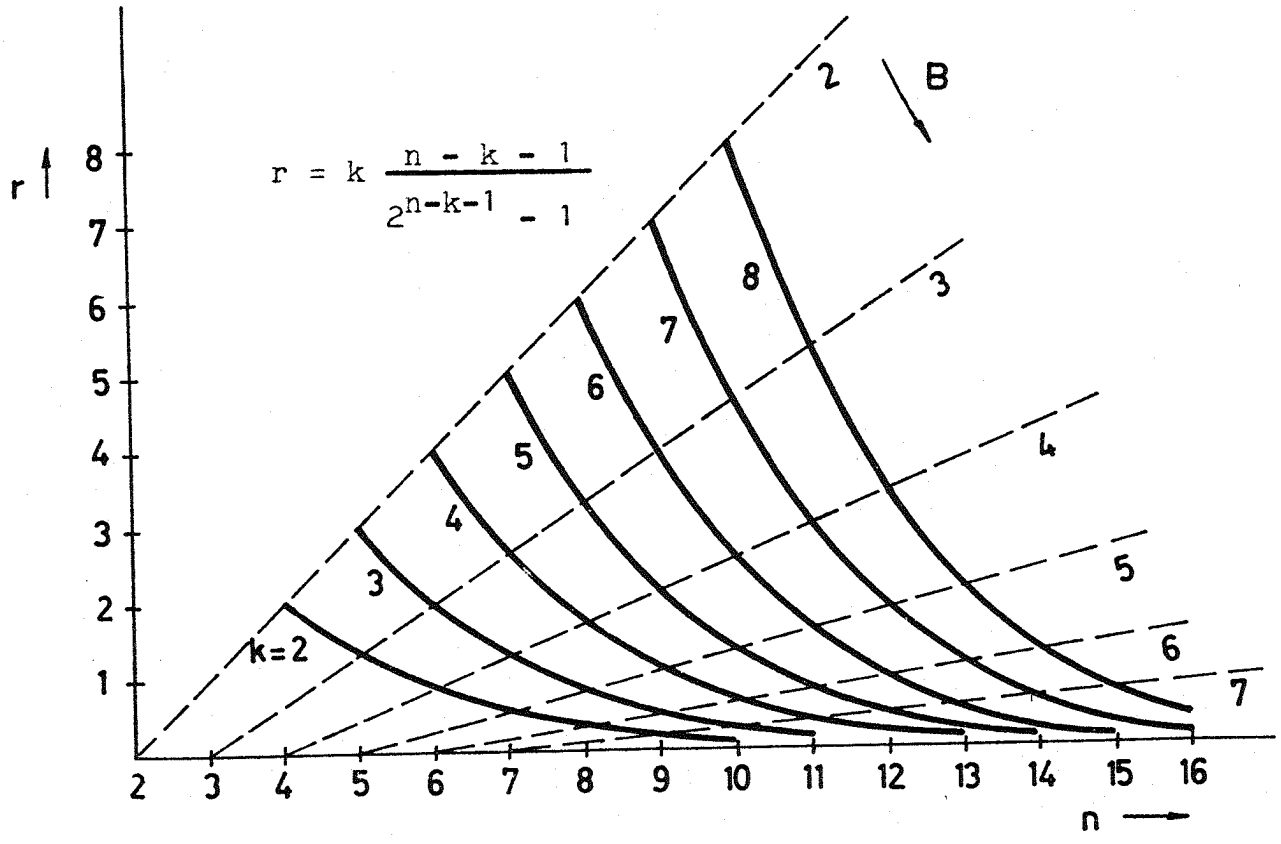


Fig.2.Parameter limits of cost-effective nonredundant kx1-cascades

variables (for r=1) or 5 variables (for r=2). Cascades with r > k are always cost-effective. Logic systems with $r \geq n-1$ can often be realized by means of several cascades, each of them implementing a partial function

$$F_i : \{0,1\}^n \text{--->} \{0,1\}^{r_i} \quad \text{where} \quad r_i < r \text{ and } \Sigma r_i = r.$$

For cost-effectiveness each r_i should satisfy condition (4). In the areas under the curves in Fig.2 cascades are not cost-effective. This , however, does not necessarily mean that the synthesis in these cases is useless. Instead of ROM's we can use gates, multiplexers or PLA's to implement cell functions and still get very efficient cascades in terms of ROM bits-per-gate ratio.

3. A FIGURE OF MERIT OF PROCESSOR-BASED PROGRAMMABLE LOGIC

The cascade of cells H_i in Fig.1b can also be replaced by a sequential circuit at Fig.3 with k D-type flip-flops and one adjustable combinational network G. The function of network G can be selected in the i-th clock period from the set of all distinct functions $\{ G_j \}$, $j = 1,2, \ldots, S$ using an integer mapping $j = \lambda(i)$, $\lambda : \{0,1, \ldots, B\} \text{--->} \{0,1, \ldots, S\}$, $S < B$, so that $H_i = G_{\lambda(i)} = G_j$. Similarly the subset of m input variables is selected out of all n input variables by means of a multiplexer net MX. We can describe such network as a time-varying sequential network (Moore type automaton):

$$Q^{t+1} = H_t (Q^t , X^t) \tag{5}$$

where Q is the vector of state variables and

X is the vector of m selected input variables.

The circuit at Fig.3 is a sort of a Boolean processor - it interprets and executes instructions from a program memory. Neglecting the cost of flip-flops, the multiplexer net and connections, we are left with two memories - a program memory with capacity of $B (a + \lceil \log_2 S \rceil)$ bits

(provided that to address the group of input variables a bits are required) and a union of cell ROM's denoted G with capacity of $k.S.2^{k+m}$ bits. In comparison to cascade implementation we can save $(B-S).k.2^{k+m}$ bits in the ROM G by not repeating identical functions, but on the other hand we have to add the program memory. This means that the sequential implementation will bring additional saving in ROM capacity with regard

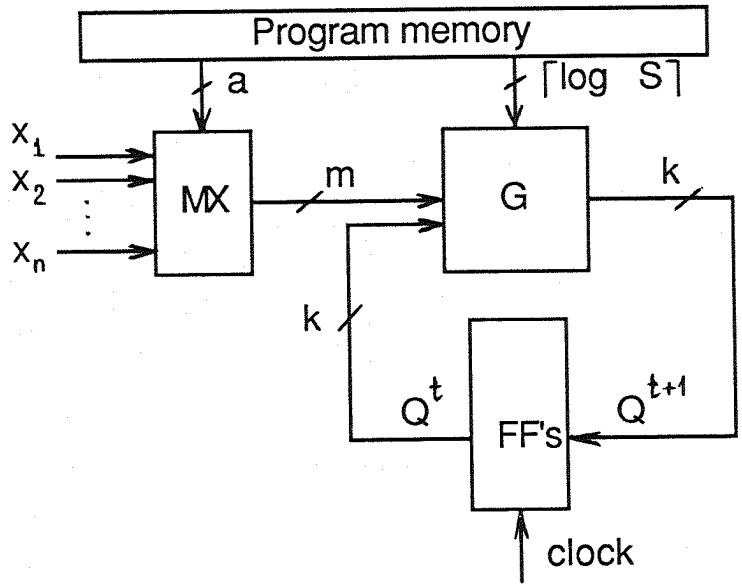


Fig.3. A sequential circuit equivalent to the cellular cascade of Fig. 1b.

to cascade implementation only if

$$(B - S)\, k\, 2^{k+m} \geq B\,(a + \lceil \log_2 S \rceil) \tag{6}$$

which comes to

$$\frac{S}{B} \leq 1 - \frac{a + \lceil \log_2 S \rceil}{k \cdot 2^{k+m}} \tag{7}$$

The high cost-effectiveness can be obtained especially in three cases:
-S is reduced drastically
-the Boolean processor is used to evaluate a larger set of functions

F_j: $\{0,1\}^{n_j}$ ---> $\{0,1\}^{r_j}$ with $n_j \leq n$, $r_j \leq r$

-the program memory is shared by the array of Boolean processors (e.g. a SIMD array).

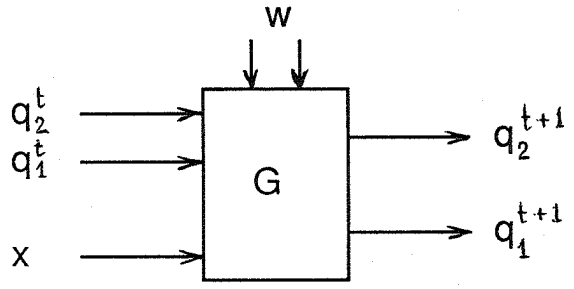
Let us comment on each of these arrangements.

If the lower processing speed does not matter, the number of operations S of a Boolean processor can be reduced a great deal. In the extreme case r = 1, k = 2, m = 1 as few as 4 operations will do, so that memory capacity of G is negligible (4 x 16 bits) and only the program memory matters. The example of a minimum set of operations is shown at Fig.4. The Boolean processor with these operations will evaluate Boolean functions of n variables, one at a time, using their disjunctive forms. The capacity of the program memory for one Boolean function of n variables will be less than that of a single ROM if

$$B \cdot (\lfloor \log_2 n \rfloor + 2) < 2^n \qquad (8)$$

which holds true for functions commonly used in practice, but may not be valid for selected "difficult" functions (such as the parity function).

The Boolean processor will be mainly cost-effective in cases when each of functions F_j: $\{0,1\}^n$ ---> $\{0,1\}^r$, that ought to be implemented, depends only on a small subset of input variables $n_j \leq n$. In this case we



w	q_2^{t+1}	q_1^{t+1}
0	0	1
1	q_2^t	$q_1^t \cdot x$
2	q_2^t	$q_1^t \cdot \bar{x}$
3	$q_1^t + q_2^t$	1

Fig.4. The set of operations of the simplest Boolean processor

better refer to operations of the Boolean processor completed in one clock period as to microoperations which are coded in microinstructions in a microprogram memory. Under the operation we shall understand evaluation of function F_j by means of a sequence of microoperations. The operation is invoked by the instruction from a processor at the higher level.

Let the number of distinct operations be P, distinct microoperations S and the number of microinstructions to specify operation F_j be B_j. If we do not consider the cost of a Boolean processor itself, the capacity of a microprogram memory will be less than that of the original single ROM when

$$\sum_{j=1}^{P} B_j \left(\lceil \log_2 S \rceil + a \right) < r P 2^n \qquad (9)$$

By introducing the average number of microinstructions B_{av} for a given set of operations, we obtain

$$B_{av} < \frac{r \cdot 2^n}{\lceil \log_2 S \rceil + a} \qquad (10)$$

Condition (10) is easily satisfied in practice. The potential applications of the Boolean processors are in the sequential controllers. If the next state of a controller is to be selected out of b distinct states depending on which Boolean function of input variables from the given set of disjunctive functions attains the value 1, the Boolean processor can evaluate a code of $\lceil \log_2 b \rceil$ bits for this multiway branch. At present, rather the alternative approach of executing conditional programs on binary decision machines is used [2]. In a special case $r = 1$, $m = 1$ the comparison of both the approaches has been made [3] with the result that the Boolean processor of a particular design outperforms the binary decision machine not only in memory capacity, but also in speed of processing. It is felt, though, that a more detailed comparison of both the approaches should be still done.

Extremely cost-effective applications of Boolean processors are found in parallel SIMD systems, since one central microprogram memory is shared by all processors. An array of Boolean processors executing the

same microprogram, even when working in the bit-serial mode, may outperform traditional monoprocessor systems in image processing, in solving graph-theoretic problems, matrix operations and other areas [4].

From the above analysis, it may be seen, that cascade synthesis of Boolean functions could be a fruitful tool in many areas of logic system design. The method of synthesis will be dealt with in the next section.

4. NONREDUNDANT CASCADE SYNTHESIS OF BOOLEAN FUNCTIONS

The derivation of functions at the outputs of the (B-1)-th cell, (B-2)-th cell, etc., down to 1st cell in Fig.1b can be looked at as the iterative decomposition of the overall function F. The theory of decomposition has been developed in the past [5], [6] and we shall use some of its results. First we review a concept of abstract decomposition. We are going to use multiple-valued logic functions in order to make the presentation concise. Let us consider an R-valued function of p binary variables, possibly with don't cares,

$$F : D \dashrightarrow \{0,1, \ldots, R-1\}, \qquad D \subset \{0,1\}^p. \tag{11}$$

We can split p binary variables into two disjoint groups, replace them with variables x (M-valued) and y(N-valued), and consider function F(x,y).

Definition 1. Let X,Y,Z, and W be arbitrary finite sets, $X = \{0,1, \ldots, M-1\}$, $Y = \{0,1, \ldots, N-1\}$, $Z = \{0,1, \ldots, R-1\}$, and $W = \{0,1, \ldots, K-1\}$. We say that function $F(x,y) : E \dashrightarrow Z$ where $E \subset X \times Y$ is decomposable with regard to variable x if there exist functions $G: X \times W \dashrightarrow Z$ and $f: Y \dashrightarrow W$ such that for all $[x,y] \in E$

$$F(x,y) = G [x, f(y)] \tag{12}$$

Decomposition is clarified by Fig.5.

To test whether the given function is decomposable, one could determine classes of compatible elements in Y [6] or use charts invented by Curtis [5]. For a larger number of binary input variables neither method is practical. We better reformulate the decomposition theorem as follows:

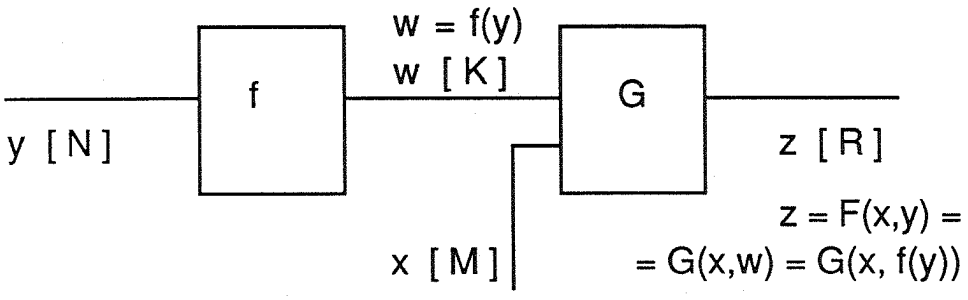


Fig.5. Decomposition of function F (multiplicity of signal values shown in brackets)

Theorem 1.(see notation of Df.1).

Function F(x,y): E ---> Z is decomposable with regard to the M-valued variable x, i.e. F(x,y) = G[x,f(y)], where variable y is N-valued and function f(y) is K-valued, K < N, if and only if there exist at most K distinct ordered M-tuples

$$[\ F(0,y),\ F(1,y),\ \ldots,\ F(M-1,\ y)\ ] \tag{13}$$

for all $y \in Y = \{0,1,\ \ldots,\ N-1\}$.

Proof. A. The necessary condition.

If the function F has the property stated in Theorem 1, then we can assign arbitrarily values 0,1, . . . , K-1 to distinct M-tuples. For all y in a certain subset Y_w of Y, $y \in Y_w \subset Y$, we get the same M-tuple that we denote w,

$$w \leftarrow [\ F(0,y),\ F(1,y),\ \ldots,\ F(M-1,\ y)\ ] \qquad w = 0,1,\ \ldots,K-1. \tag{14}$$

Y_w is one of K compatibility classes defined on Y: two elements y_i and y_j are compatible with respect to F if for all $x \in X$ such that $(x,y_i) \in E$ and $(x,y_j) \in E$, $F(x,y_i) = F(x,y_j)$. (For fully specified functions without don't cares compatibility is an equivalence relation.) Then the function f and G are easily found as

$$f(y)\ |_{y\ \in\ Y_w} = w \qquad\qquad G(x,w) = F(x,y)\ |_{y\ \in\ Y_w} \tag{15}$$

for all $w = 0,1, \ldots, K-1$.

B. The sufficient condition.

Let us consider a function $F(x,y) = G(x, f(y))$, Fig.5. If the function f is K-valued, then a relation of compatibility between elements of Y can be defined. Two elements y_i and y_j are compatible if $f(y_i) = f(y_j)$. Compatibility classes are denoted Y_w, $w = 0,1, \ldots, K-1$. For each $y \in Y_w$ we get the w-th M-tuple

$$[\ G(0,f(y)),\ G(1,f(y)),\ \ldots,\ G(K-1,\ f(y)\ )\ ] \tag{16}$$

and therefore for all $y \in Y$ at most K distinct M-tuples

$$[\ F(0,y),\ F(1,y),\ \ldots,\ F(M-1,\ y)\ ], \tag{17}$$

Q.E.D.

To find functions G and f for a given decomposable function F is a straightforward procedure. In fact there are many pairs of functions G and f equally suitable. We shall illustrate the procedure by an example.

Let us synthesize a 4-bit comparator of two 4-bit binary numbers $A = a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0$ and $B = b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0$. The output of the comparator is to detect one of three conditions $A = B$, $A > B$, and $A < B$ by its output value 0, 1, and 2. The synthesis procedure is shown at Fig.6. We begin with the full-size function table 16 x 16, the function is completely specified. If we choose $x = b_0$ and $y = \text{val}\,(b_3 b_2 b_1 a_3 a_2 a_1 a_0) \overset{df}{=} b_3 2^6 + b_2 2^5 + b_1 2^4 + a_3 2^3 + a_2 2^2 + a_1 2 + a_0$, we have $M = 2$, $N = 128$ and $K=4$ pairs [F(0,y), F(1,y)] . The function f(y) must be hence 4-valued and we can use two rails to carry its value. The table of function f(y) is obtained by shrinking the original table to half of its size, each distinct pair being denoted arbitrarily by one value $w = 0,1,2$, and 3. We have used

$$\begin{bmatrix}0\\2\end{bmatrix} := 0 \qquad \begin{bmatrix}1\\0\end{bmatrix} := 1 \qquad \begin{bmatrix}1\\1\end{bmatrix} := 2 \qquad \begin{bmatrix}2\\2\end{bmatrix} := 3 \qquad (16)$$

The inverse mapping (16) w ---> [F(0,y), F(1,y)] defines function G:WxX--->Z realized by the last cell in the cascade. Now the decomposition is repeated with function f with regard to variable $x = a_0$, etc. After each decomposition step we get a function of one less binary variables ; we terminate the procedure when the function of three binary variables is obtained. The resulting cascade has B=6 cells and is cost-effective (6x16 = 96 bits against $2 \cdot 2^8 = 512$ bits).

The implementation of the comparator in a form of a 6-cell cascade would have probably an excessive propagation delay in the most of applications. We can cut the number of cells to one half using a 2x2-cascade. Here K=3 and three distinct M-tuples selected are denoted

$$\begin{bmatrix}01\\20\end{bmatrix} := 0 \qquad \begin{bmatrix}11\\11\end{bmatrix} := 1 \qquad \begin{bmatrix}22\\22\end{bmatrix} := 2 \qquad (17)$$

We can terminate iterative decomposition after two steps, Fig.7, with the same M-tuples in each step. The resulting cascade consists of three identical cells and has the same ROM capacity of 96 bits as the previous 6-cell cascade. This is generally so: if both the cascades 2x1 and 2x2 exist, their cost-effectiveness is the same since we can combine each pair of neighbouring cells in the 2x1-cascade into one cell of the 2x2-cascade with the same memory requirements. For higher values of m, m>2, this is not true any longer (e.g. for m=3 memory capacity increases 4/3-times).

Finally we can decrease the number of levels and thus propagation delay even further using a 2x3-cascade. Here M=8 and K=4 since we can pick four distinct 8-tuples and rename them e.g.

$$\begin{bmatrix}0111\\2011\end{bmatrix} := 0 \qquad \begin{bmatrix}1111\\1111\end{bmatrix} := 1 \qquad \begin{bmatrix}2201\\2220\end{bmatrix} := 2 \qquad \begin{bmatrix}2222\\2222\end{bmatrix} := 3 \qquad (18)$$

The original table of the comparator is reduced through this new notation to the table of 5 variables that is realizable by the first cell in the cascade at Fig.8. The second cell function is defined by an inverse mapping (18). The total memory capacity is 128 bits.

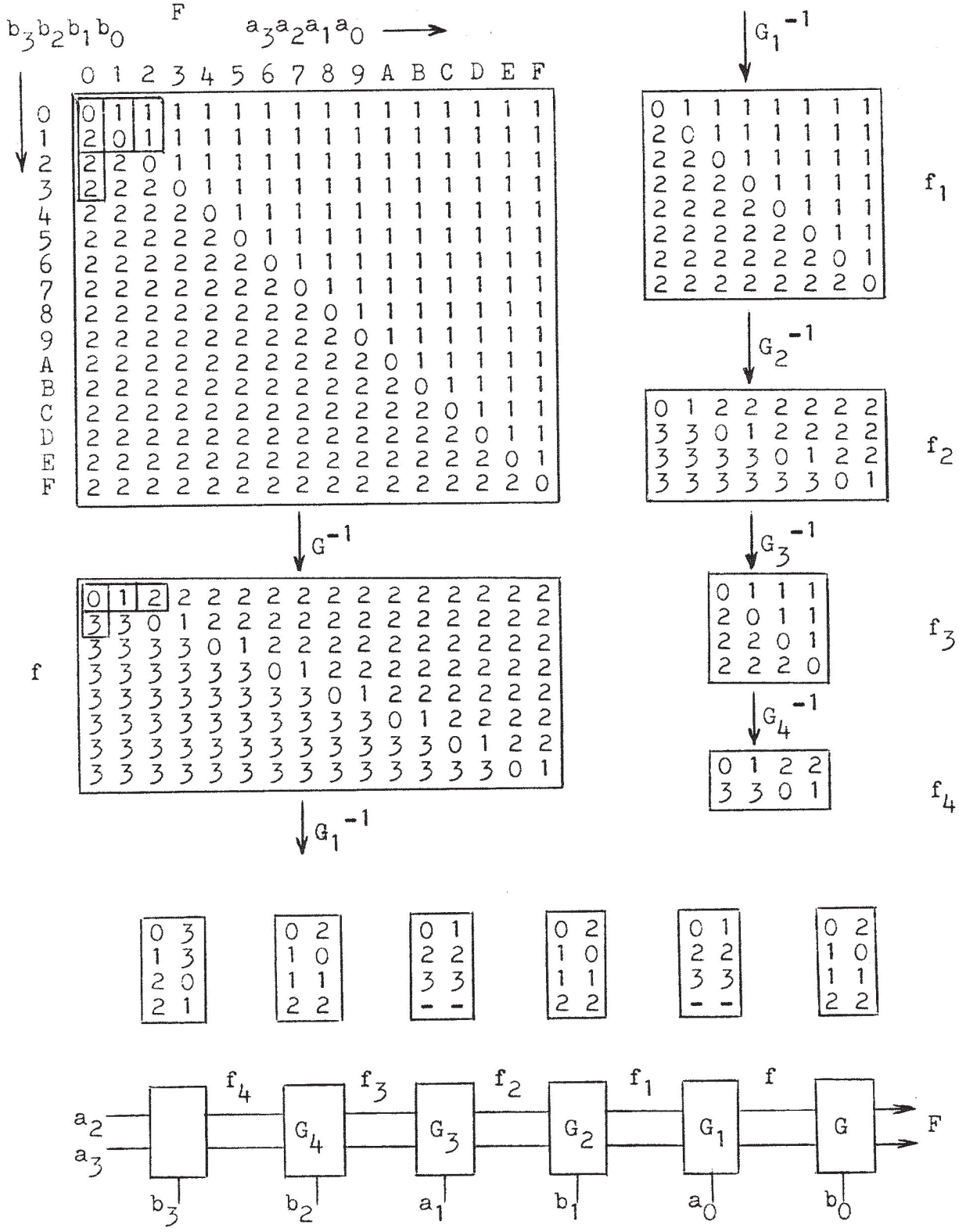


Fig.6. The 2x1-cascade synthesis of the 4-bit comparator

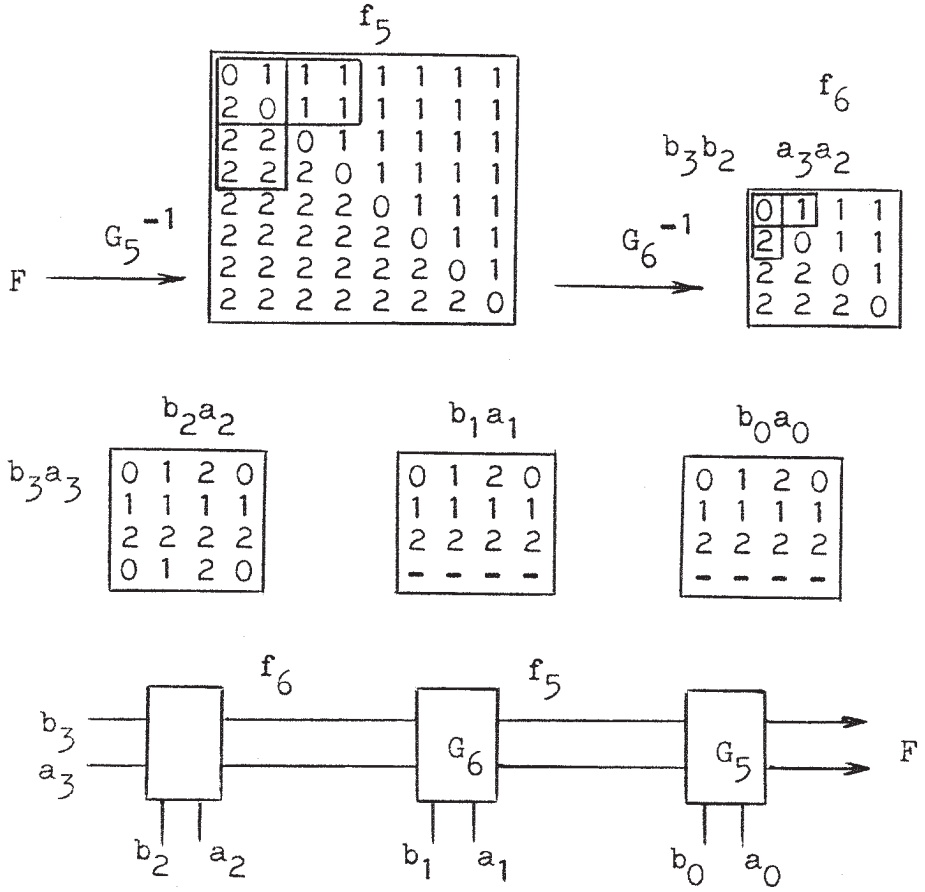


Fig.7. The 2x2-cascade implementation of the 4-bit comparator

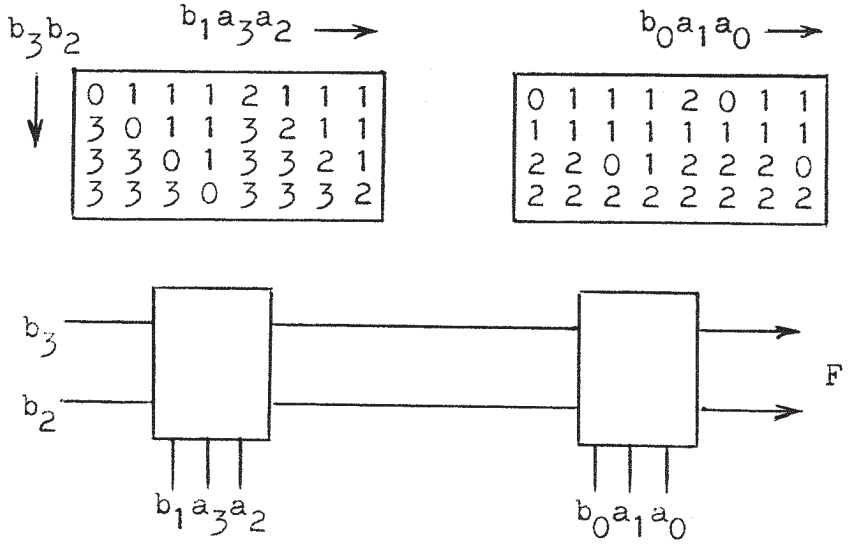
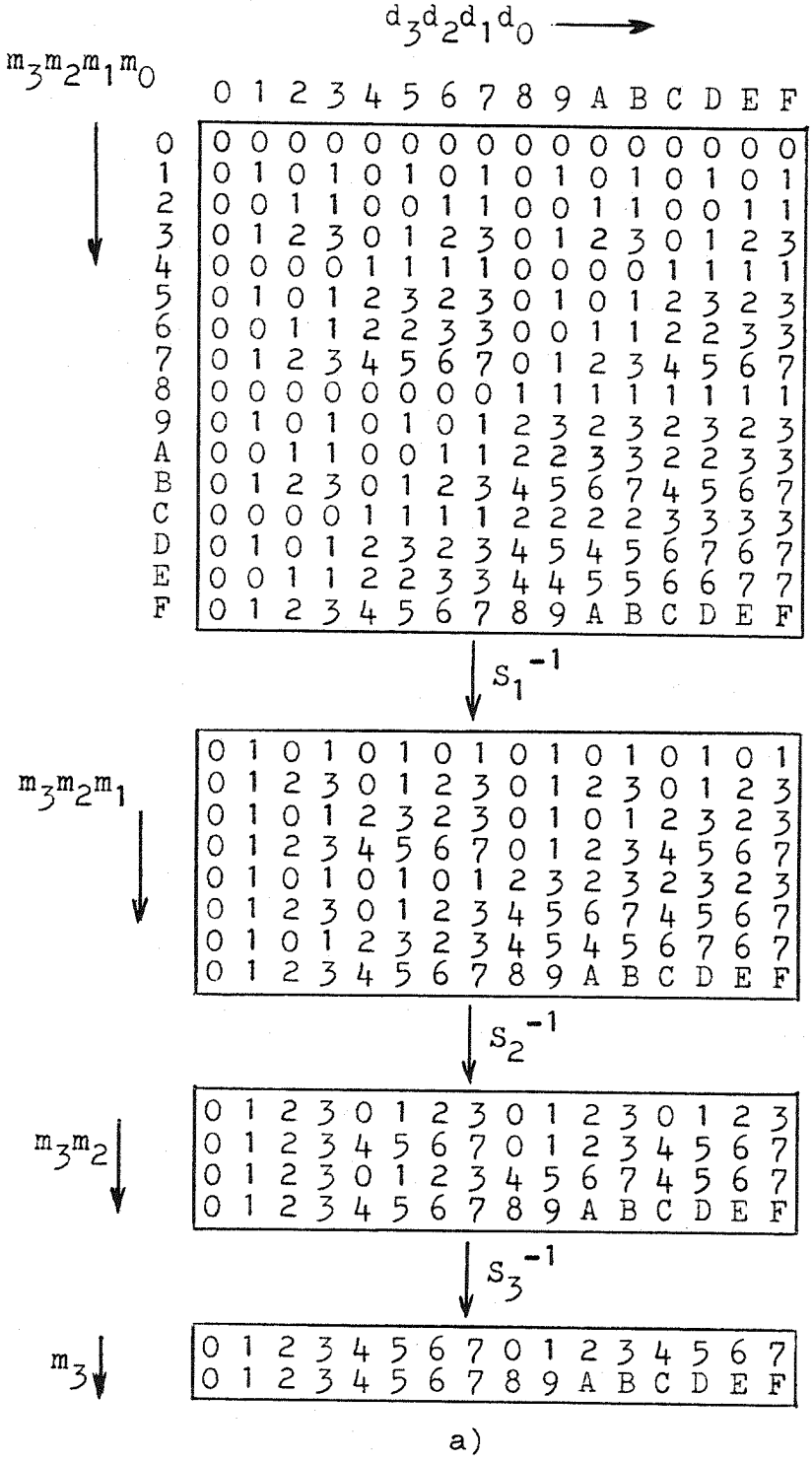


Fig.8. The 2x3-cascade implementation of the 4-bit comparator

Different decompositions of an R-valued function F: $\{0,1\}^n$ ---> {0,1, . . . , R-1} may come handy when we implement F in VLSI (decomposition of too large PLA's) or in microprocessor implementation (saving ROM capacity when speed is not critical). In practice, functions used most often are those which have sparse tables (many don't cares) and/or tables with repetitive patterns, that are easily synthesized and realized by cost-effective cascades.

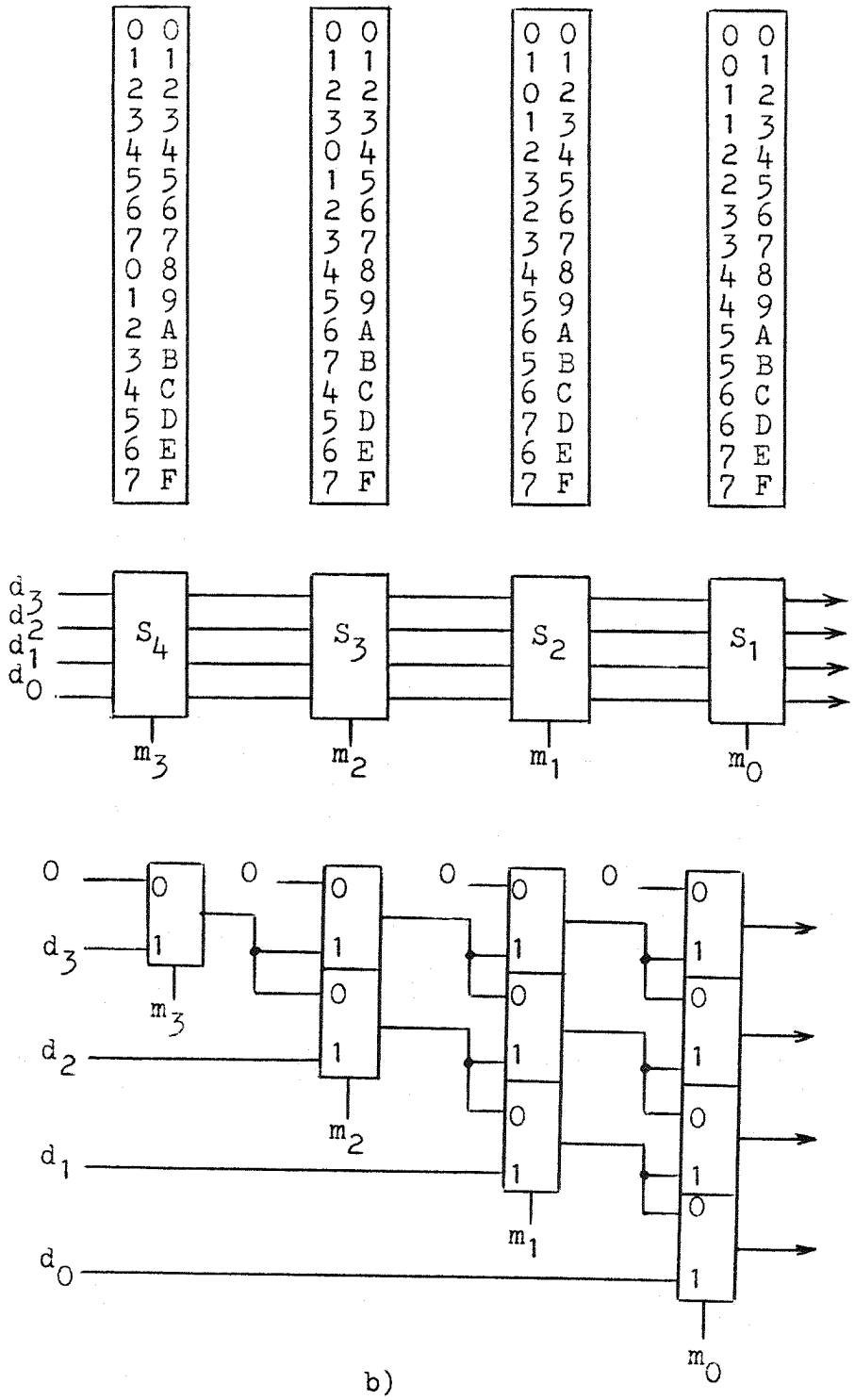
As another example let us take the Am 29803A 16-way branch control unit. It has four data inputs d_3, d_2, d_1, d_0 and four masking inputs m_3, m_2, m_1, m_0 . If $m_i = 0$, the corresponding value at data input d_i will not pass to any output. Data bits that are allowed by their masking bits to show up at outputs are rearranged to occupy the lowest positions on the 4-bit output bus. The full function table of this circuit is given at the top of Fig.9a. We can count as many as 16 distinct pairs with respect to variable m_0 (K=16), so that 4 rails are necessary. The table is shrinked three times in the vertical direction by one half using mappings S_i^{-1}, i = 1,2,3. The value of K remains in this particular example always 16 as at the beginning. The resulting table represents a function S_4 of five variables that together with mappings S_1, S_2, and S_3 describe functions of cells in the cascade, Fig.9b. By inspection of the cell functions one can easily find the possible implementation based on two-input multiplexers at the bottom of Fig.9b.

The next example is the 8-bit priority encoder defined by the function table in the upper left hand corner of Fig.10, which is expanded to the full size function table next to it. The later table contains 8 pairs $[F|_{x_0=0}, F|_{x_0=1}]$ and the last cell in the cascade transforms eight digits 0 to 7 into these pairs, e.g. 0-->08, 1-->99, and so on. Having replaced distinct pairs by their new symbols we obtain a new table, half of the size of the previous one. The procedure of re-labelling distinct pairs in the table is repeated four times until we get a function of four variables that is realized by the first cell in the cascade. Four other cells correspond to four table transformations - they are their inverse. Note that three cells in the middle of the cascade are identical.



a)

Fig.9. Synthesis of Am 29803A 16-way branch control unit
(4x1-cascade of 4 cells)
a) table transformations



b)

Fig.9. Synthesis of Am 29803A 16-way branch control unit
(4x1-cascade of 4 cells)
b) cascade implementation

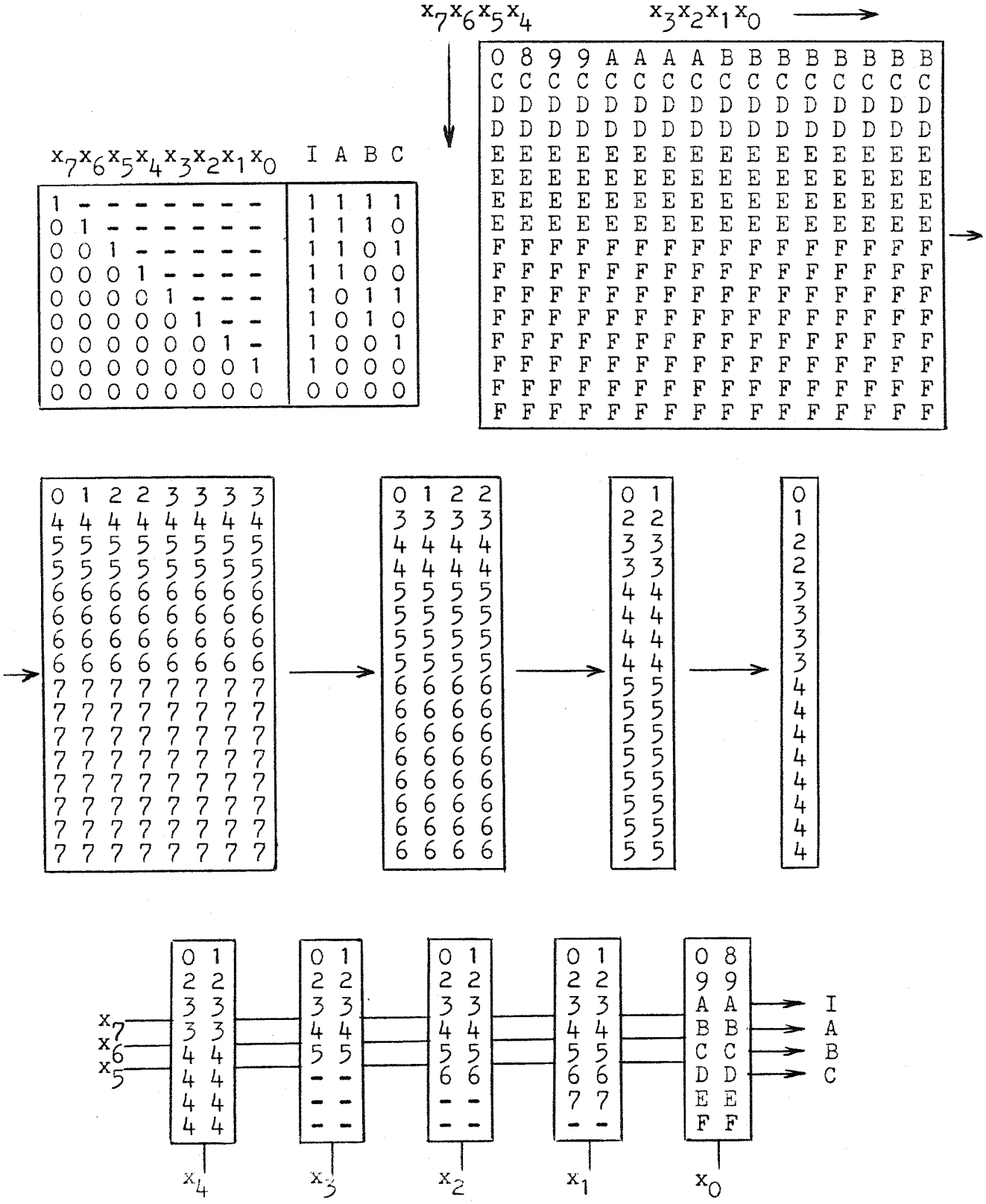


Fig.10. Cascade implementation of the 8-bit priority encoder

Among other solved examples let us name a 4-input binary adder (a kx2-cascade of 3 cells, k=2,3,4), the same adder with carry in, a multiplier in $GF(2^4)$, etc. The 4-by-4 bit multiplier is the example of a function which does not have the cost-effective cascade implementation. Its function table is neither sparse, nor has the repetitive patterns.

5. SYNTHESIS OF INCOMPLETELY SPECIFIED BOOLEAN FUNCTIONS

As we have already said, many functions used in practice can be implemented by cost-effective cascades. But still we would like to have at least a sufficient condition for realizability of the given function by the k x m-cascade. Such condition can be given for functions defined by sparse tables with many don't cares.

Let $F : E \rightarrow Z$ be a Boolean function, $E \subset \{0,1\}^n$, $Z \subset \{0,1\}^r$. An ordered sequence of 0's and 1's corresponding to a configuration of input variables $x_1, x_2, \ldots, x_n$ (1 for x_i, 0 for $\overline{x_i}$) will be called vertex and its decimal representation will be denoted e. We will assume that a function F of n variables is specified by a list of vertices $e_i \in E$ and by the associated function values $F(e_i)$. The set $\{0,1\}^n - E$ constitutes the don't care conditions.

Theorem 2. (The sufficient condition for realizability of incompletely specified functions).
Every R-valued function $F: \{0,1\}^n \rightarrow \{0,1, \ldots, R-1\}$ is realizable by the k x m-cascade if it is specified at not more than 2^k vertices.

Proof. First consider the case m=1. If the function F is defined in at most 2^k vertices e_i, then the number of pairs $[F|_{x_i=0}, F|_{x_i=1}]$, $i \in \{1,2, \ldots, n\}$ must be also less or equal 2^k since in the worst case each value of $F(e_i)$ will create just one distinct pair with don't care. If some pairs contain function values at two vertices $[F(e_i), F(e_j)]$, then the number of pairs will be even lower.

Generally, if we have $m > 1$, the number of distinct 2^m-tuples will be equal or less than the number of pairs, since function values at more than two vertices might be combined into one 2^m-tuple. Then, by Theorem 1, function F is decomposable with regard to the 2^m-valued variable x,

$$F(x,y) = G [x, f(y)]$$

where function f(y) is 2^k-valued. Denoting 2^k distinct 2^m-tuples by integers 0,1, . . . , 2^k-1, we again obtain a function defined at 2^k vertices, but with a reduced number of binary variables. Similarly we can go on until we obtain a function of k+m binary variables that is realizable by a single cell. The number of decomposition steps is apparently $\lceil (n-k)/m \rceil$. Thus the original function is realized with at most k rails entering each cell in the cascade, Q.E.D.

Corollary 2.1. Every R-valued function F: $\{0,1\}^n$ --->{0,1, . . . , R-1} is realizable by the k x m-cascade if F ≠ const at 2^k -1 vertices and F=const everywhere else.

Here the pairs or 2^m-tuples are created not by using don't cares, but using values const. The reasoning remains the same.

The multi-level partitioning is illustrated on a simple example at Fig.11. Function F: $\{0,1\}^6$--->$\{0,1\}^3$ is defined at 11 vertices. Since some of them can be combined into pairs, we can distinguish not more than four distinct pairs [$F(e_i)$, $F(e_j)$], namely [4,0], [-,3] , [1,2], and [3,5] . Thus we begin with two rails entering the last cell in the cascade. Using mapping G_1^{-1} defined at Fig.11 we obtain a function of 5 variables defined at 10 vertices. Again we can combine vertices into pairs in such a way that we end up with three distinct pairs [$F(e_i)$, $F(e_j)$] only and two rails into the last but one cell. Mapping G_2^{-1} reduces the number of variables to four and finally mapping G_3^{-1} to three, so that 2x1-cascade has four cells, Fig.11.

As the more complex examples two PLA's used in Intel's single-chip microcontroller MCS-51 have been investigated, see Table 1. The first PLA1 can be decomposed directly using Theorem 2. Since the number of vertices where F is specified is less than 256, we can use 8xm-cascade. E.g. the 8 x 1-cascade of 3 cells has a memory capacity of 2560 bytes, in contrast to 8192 bytes of a single ROM implementation. Generating three function tables for cascade cells would have to be done with the aid of a computer. The algorithm has already been explained on the smaller problem and will be precisely formulated later on. To improve cost-effectiveness of the cascade implementation, we could also use two

	No. of inputs	No. of outputs	No. of def. vertices	No. of distinct products after minimization
PLA1	13	8	175	31
PLA2	11	8	632	53

Table 1. Parameters of two PLA's in Intel's MCS-51

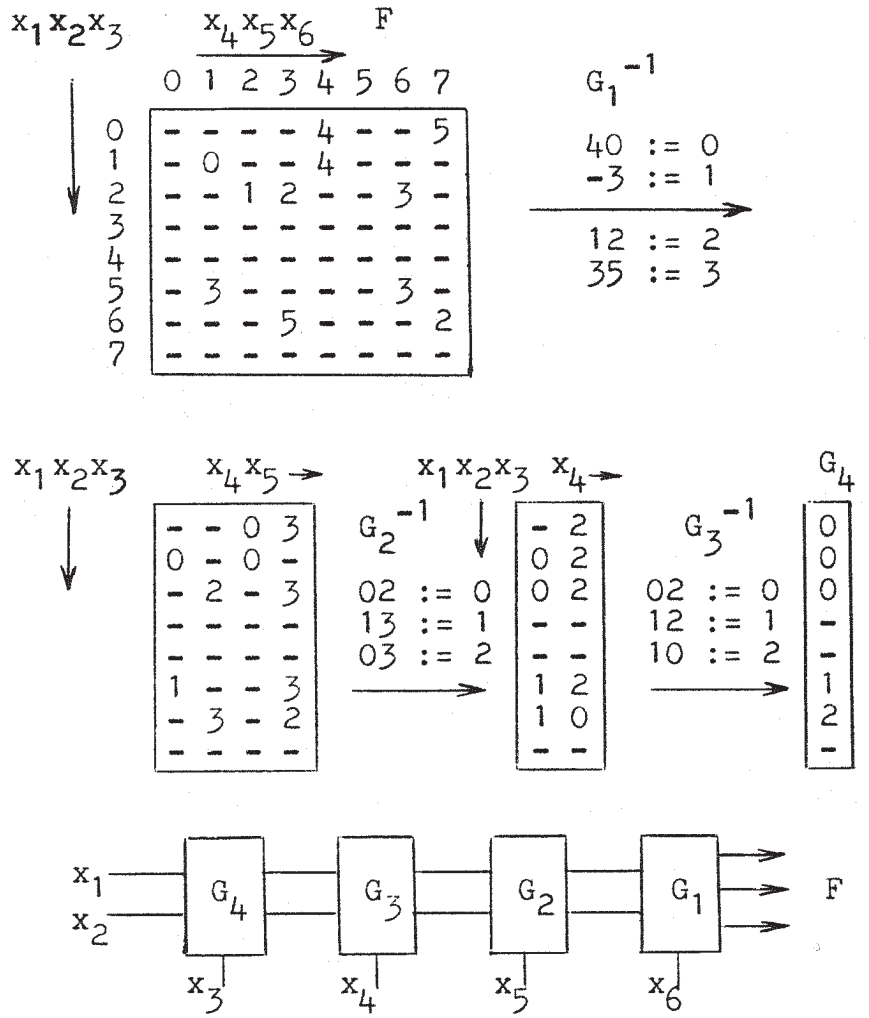


Fig.11. Cascade decomposition of the sparse function

cascades in parallel, each of them computing one function f_i :$\{0,1\}^{13}$---> $\{0,1\}^4$, i =1,2 or ultimately 4 cascades in parallel computing four functions g_j: $\{0,1\}^{13}$ --->$\{0,1\}^2$, j = 1,2,3,4. Don't cares in the functions should be used to minimize the number of 2^m-tuples and thus the number of rails, k. Generally we may have k and m varying along the cascade.

For the second PLA2 with 632 defined vertices k=10 rails would be sufficient according to Theorem 2. A related 10 x 1-cascade would not be cost-effective, though. There are, however, better cascade implementations. The author was able, without too much difficulty, find the decomposition by inspection of the function table. Using repetitive patterns a 2-cell, 4 x m-cascade was found with m=2 and 5 (m≠const). This cascade has memory capacity of 544 bytes, whereas the single ROM has 2048 bytes. To increase cost effectiveness even more (to 224 bytes) two parallel 4-rail cascades have been synthesized. The procedure could be done by hand for m=2 as the function table size was reduced 4 times in each decomposition step. Four cells of each cascade have been obtained in three steps. The number of distinct 4-tuples was slightly below 16 and in one step, by means of clever use of don't cares, just 16 - the upper limit for 4 x m-cascade. The solution found could thus serve as the good testing example for possible computer programs assisting decomposition.

Cascade decomposition of incompletely specified Boolean functions can be automated and the decomposition algorithm will now be given. Basically each Boolean function F: $\{0,1\}^n$ --->{ $0,1\}^r$ can be implemented by the k x m-cascade with possibly m and k varying along the cascade. The problem is, that some cascades will not be cost-effective. For example the number of rails entering the last cell can always be taken as min (n-m, $r.2^m$) . Then the function of n-m variables can again be decomposed similarly, etc.

The decomposition algorithm presented below will find only one cell in the cascade and iteratively the whole cascade. If there is a cost-effective cascade, it should find it. Otherwise it will signalize a trivial decomposition resulting in the cascade that is not cost-effective. The algorithm will not look for a system of parallel cascades.

The input to the algorithm is a Boolean function F: E --> Z, $E \subset \{0,1\}^n$, $Z \subset \{0,1\}^r$ specified by the list of vertices $e_i \in E$ and by the list

of associated function values $z_i = F(e_i)$. An ordered 2^m-tuple of vertices obtained by changing m input variables in all possible ways from 0 . . . 00 to 1 . . . 11 and leaving the remaining input variables at a certain constant values will be denoted $D_j = [e_0, e_1, \ldots, e_{2^m-1}]$. The corresponding 2^m-tuple of function values will be denoted $T_j = [z_0, z_1, \ldots, z_{2^m-1}]$. Our task is to cover the set of function values $z_i = F(e_i)$ by the least possible assortment of 2^m-tuples T_j, $j = 1,2, \ldots, K_{min}$.

Algorithm for decomposition.

Input: the list of vertices e_i and associated function values z_i, $[e_i, z_i]$, $i = 1,2, \ldots, i_{max}$; two parameters n, m : $n \geq 4$, $m \geq 1$.

Output: the list of pairs $[e_i, z_i]$ specifying the Boolean function of n - m variables.

1. Initialize i <--- 1, j <--- 1 .
2. Complete vertex e_i by other vertices to create a 2^m-tuple D_j. Substitute $z_i = F(e_i)$ into the proper position of the 2^m-tuple T_j, the remaining positions are don't cares ; i <--- i + 1 .
3. Investigate if e_i is covered by any 2^m-tuple D_k, $k = 1,2, \ldots, j$.
   If yes, substitute $z_i = F(e_i)$ into the proper position of the 2^m-tuple T_k instead of don't care. If $i = i_{max}$ then go to step 4 else i <--- i +1 and repeat step 3.
   If not, j <--- j +1 and go to step 2.
4. Find maximum compatible classes in the set of 2^m-tuples T_j.
5. Find the minimum set of maximum compatible classes covering the set $\{ T_j \}$. Let this lowest number of compatible classes be K.
6. Repeat steps 1 till 5 for other m-tuples of input variables and find the lowest value of K from all C(n,m) cases, K_{min} . The corresponding m-tuple of input variables will be denoted by a star. If
   $k_{min} = \lceil \log_2 K_{min} \rceil = n - m$,
   only the trivial decomposition exists.
7. Use a subset of $\{0,1, \ldots, 2^{k_{min}}-1 \}$ to mark arbitrarily all 2^m-tuples

T_j^* created. These will be values z_i of a new function of n-m variables. Replace each 2^m-tuple D_k^* by one vertex of n-m variables. (leaving out m variables changing in all possible 2^m ways) and you have a list of vertices where the new function is defined.

Computation time of this algorithm will grow with parameter m as the number of combinations C(n,m), so that m may be limited for reasons of economy.

At present the decomposition algorithm is being implemented on HP 21 MX computer at Technical University of Brno.

6. SYNTHESIS OF REDUNDANT CASCADES

Nonredundant cascade synthesis of Boolean functions described in sections 4 and 5 can be modified in such a way that some input variables are applied to more than one cell in the cascade. The transformation of a function table related to the redundant use of a certain variable does not change the size of the table and therefore does not represent decomposition. The only purpose of such a transformation is to reduce the number of M-tuples below the limit 2^k so that the next decomposition step becomes possible along the k-rail cascade. This can be done by transforming arbitrarily the map of the function in sectors corresponding to values x=0, x=1, . . . , x=M-1 with the only requirement that the inverse transformation be unique.

As an example, let us decompose the function $f:\{0,1\}^6 \rightarrow \{0,1\}$ defined by the map at Fig.12. This function is fully specified. Let us use M=2 (decomposition with respect to a single binary variable). There are 4 M-tuples (pairs) in the map, that are assigned new values arbitrarily,e.g.

$$\begin{bmatrix}0\\0\end{bmatrix} := 0 \qquad \begin{bmatrix}0\\1\end{bmatrix} := 1 \qquad \begin{bmatrix}1\\0\end{bmatrix} := 2 \qquad \begin{bmatrix}1\\1\end{bmatrix} := 3 \tag{19}$$

This mapping is denoted as D_1^{-1} at Fig.12. When we carry out mapping (19), variable x_3 is isolated and a quaternary function of 5 variables results.

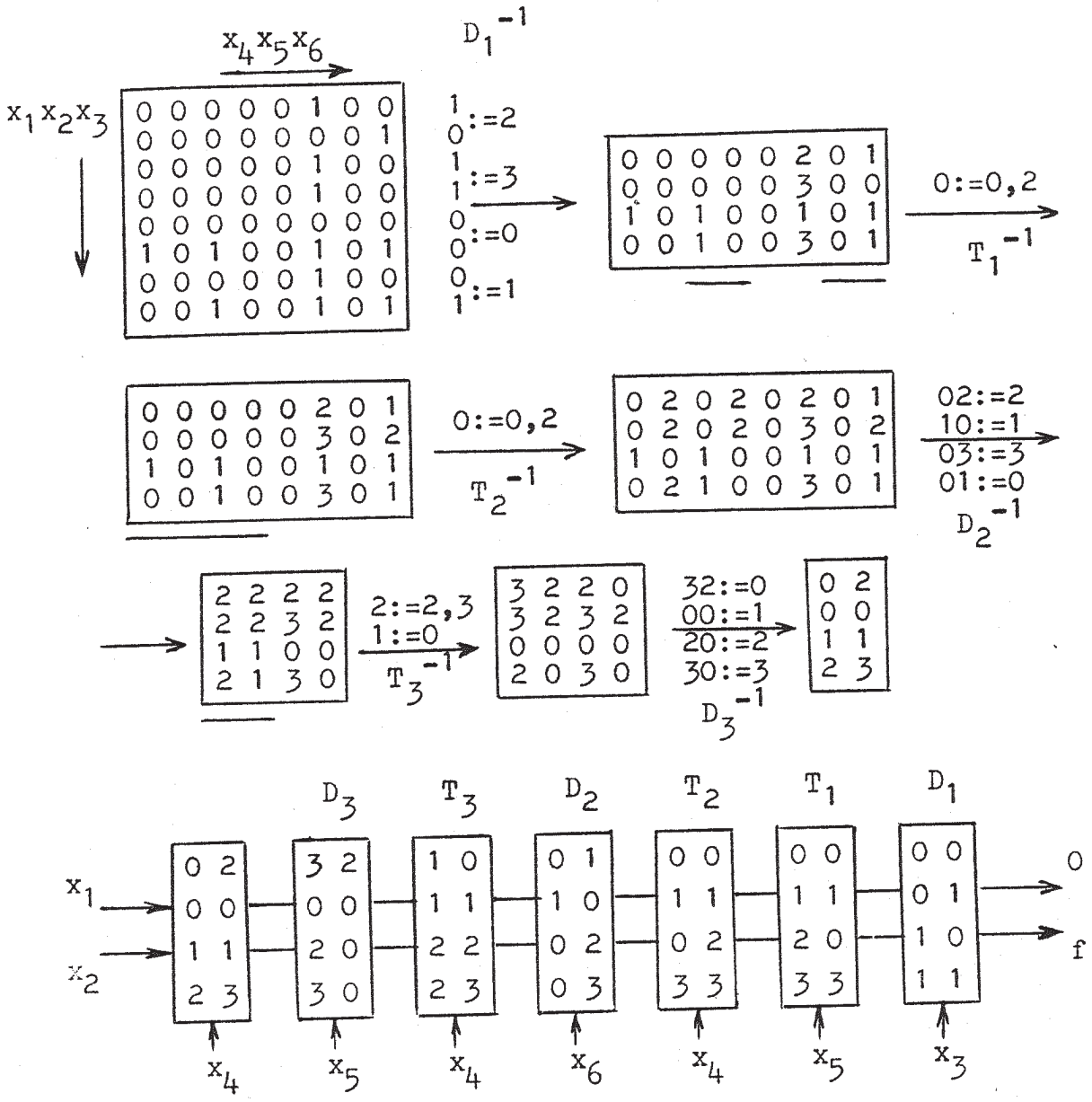


Fig.12. Iterative decomposition of a boolean function and the associate cellular cascade

Counting the number of pairs with respect to variable x_6, we can find 5 distinct pairs. This is more than 2^2 and therefore, to do with 2 rails in the cascade, we have to decrease this number at least to four. The idea is to unite two distinct pairs, in our case [0,0] and [0,2] into [0,2]. Two transformations are used for this purpose: T_1^{-1} changes one zero in the map segment $x_5=1$ (underlined) to 2 and T_2^{-1} five zeros in the map segment $x_4=0$ again to 2. These transformations can be applied as there are no 2's in segments $x_5=1$ and $x_4=0$ and inverse transformations are thus unique.

After carrying transformations T_1^{-1} and T_2^{-1} the number of distinct pairs of function values with respect to variable x_6 is four and we can use mapping D_2^{-1}, e.g

D_2^{-1} : [0 2] : = 2 [1 0] : = 1 [0 3] : = 3 [0 1] : = 0

and separate variable x_6. The function of 4 binary variables that remains to be synthesized contains 7 distinct pairs of function values with respect to variable x_5. Using transformation

T_3^{-1} : 2 : = 2,3 1 : = 0

in the table segment x_4 =0 makes some of them identical so that only 4 remain. Again transformation T_3^{-1} substitutes a function value in the segment where it does not occur, so that the inverse transformation

T_3 : 0 : = 1 2 : = 2 3 : = 2

is unique. The final step is mapping of distinct pairs into new values,

D_3^{-1} : [3 2] : = 0 [0 0] := 1 [2 0] : = 2 [3 0] : = 3

and a function of 3 variables is obtained. Note that decomposition steps are denoted D^{-1} and auxiliary transformations T^{-1}. Now by using the inverse mappings and transformations D and T in the opposite order one gets the functions of individual cells in the cascade as shown at Fig.12. The cellular cascade is redundant in the sense that some variables are used at more than one cell. Redundancy arises due to auxiliary transformations T^{-1} that are needed for reduction of a number of distinct pairs. If we could use only

decomposition steps D^{-1}, then the cascade would be nonredundant.

7. IMPLEMENTATION OF COMBINATIONAL AND SEQUENTIAL FUNCTIONS ON A BOOLEAN PROCESSOR

Equivalence of the cellular cascade in Fig.1b and the Boolean processor in Fig.3 makes it clear that the cascade synthesis explained above is also a tool for synthesis of microprograms for the Boolean processor. Relation of spatial iterative decomposition to a microprogram for a Boolean processor is such that each cell corresponds to execution of one microinstruction. We shall consider a processor consisting of at least two flip-flops and of a logic processing unit (LPU). The LPU is a combinational network that can be set to perform a specified function, Fig.13. The most convenient implementation of a LPU is in a form of ROM or PLA.

To simplify the combinational part of a Boolean processor, we must reduce the set of microoperations as much as possible. This will also reduce the μop-code width and simplify the interconnections to control unit. Note that a complete set of microoperations can never be implemented as the number of all Boolean functions grows too fast with the increasing number of variables. If k is the number of flip-flops of a Boolean processor that operates on m bits of input information simultaneously, then realization of a universal adjustable combinational network would need ROM with capacity of $k\,[\,2 \uparrow (k.2^{m+k} + m + k)]$ bits. This is too large capacity already for the lowest values of k and m (2^{20} bits for $k = 2$ and $m = 1$).

The problem of microprogram synthesis is looked at as the problem of sequential evaluation of combinational and sequential functions on a very simple Boolean processor of Fig.13. It is controlled by a microprogram stored in a control memory and processes two kinds of operands: values of input variables or operands stored in a local memory. Addresses of operands are parts of a microinstruction. The microinstruction format is shown at Fig.13. For the sake of brevity we shall consider multiple-valued signals within the circuit. It is implicitly assummed that they are represented by a number of binary signals. E.g. the internal state

variable $y \in Y$ can attain K values (i.e. we may have K states)

$$Y = \{ 0,1, \ldots, K\text{-}1 \} \qquad k = \lceil \log_2 K \rceil \tag{20}$$

where k is the number of binary flip-flops and similarly the input variables $x_i \in X$ are M-valued,

$$X = \{ 0,1, \ldots, M\text{-}1 \} \qquad m = \lceil \log_2 M \rceil . \tag{21}$$

Any change of state of the Boolean processor caused by the present value of the selected input variable $x_i \in X$ and by the present state $y \in Y$ and

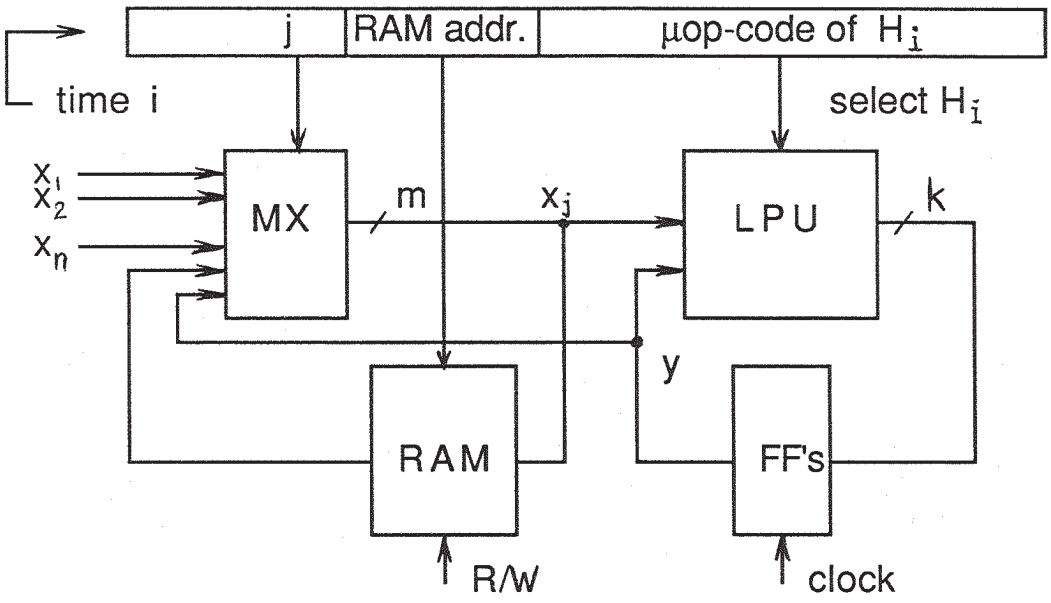


Fig. 13. Architecture of the Boolean processor

initiated by a clock pulse will be referred to as a microoperation. Thus the microoperation is defined by the mapping

$$H : X \times Y \dashrightarrow Y \tag{22}$$

The set of all K^{MK} possible microoperations will be denoted μ. In practice, for the sake of economy, the microinstruction list of a particular Boolean processor will usually cover only a limited set $\Delta \subset \mu$ of microoperations.

A sequence of microoperations with the same value of input

variable x_i is a composed microoperation. Apparently, if every microoperation from μ is either included in Δ or realizable as a composed microoperation, then the set of microoperations Δ is complete. The optimum list of microoperations does not have to be complete, though. It should be rather problem- or application-oriented. The particular application is specified by a set of operations that should be implemented on the Boolean processor. The operation is either combinational, defined by mapping

$$F: X^p \rightarrow Y \tag{23}$$

or sequential (Moore automaton) defined by mapping

$$G: X^p \times Y \rightarrow Y \tag{24}$$

The problem of microprogram synthesis can be then formulated as follows: Given the set of combinational and sequentional operations in a form (23) and (24), find the decompositions of these operations into sequences of microoperations in a form (22). In this process use the minimum possible number of microoperations.

Let us note that a minimum complete set of microoperations Ω that enables one to express the arbitrary operation as a sequence of microoperations of the same length as it is possible using microoperations from set μ has

$$|\Omega| = C(K^M, K) \tag{25}$$

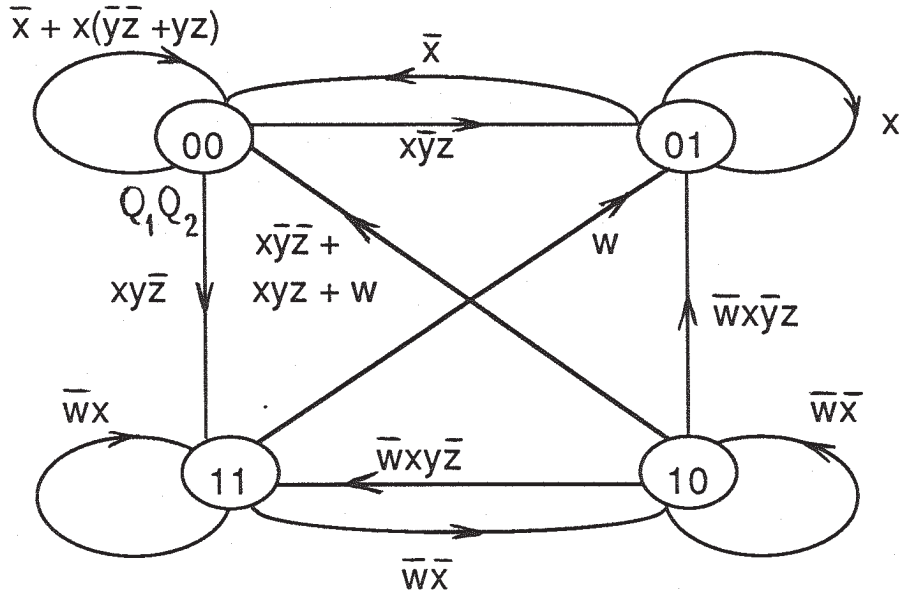
microoperations, [9]. If the timing is not critical, we can reduce the list of microoperations even further.

As the example of implementation of a combinational function on the Boolean processor we can use the Boolean function synthesized in Fig.12. Each cell in the cascade at Fig. 12 is described by one microinstruction that is executed in one microcycle. Two more microcycles are needed to read in the values of x_1 and x_2 at the beginning, but one microoperation can do that (shift). Thus to evaluate the given function we shall need 9 microcycles and 8x16 = 128 bits of ROM and 2 flip-flops in the LPU. Some of the microoperations might be used at implementation of other operations. In the extreme case we could evaluate any binary function using 4 microoperations only, but in more microcycles.

The Boolean processor of the type shown at Fig.13 can also be

made to emulate different finite state machines. One clock period of a synchronous state machine will usually take several clock periods (microcycles) of a Boolean processor. The synthesis procedure for sequential operations differs slightly from the procedure of the last sections. The difference is that internal state variables may not be used during decomposition steps, neither during transformation steps, since their values are changing in every clock cycle of the Boolean processor, whereas they should hold constant during one clock cycle of the emulated state machine. If they are really needed in decomposition, we can use the local RAM to store their values at the beginning of the state-machine cycle.

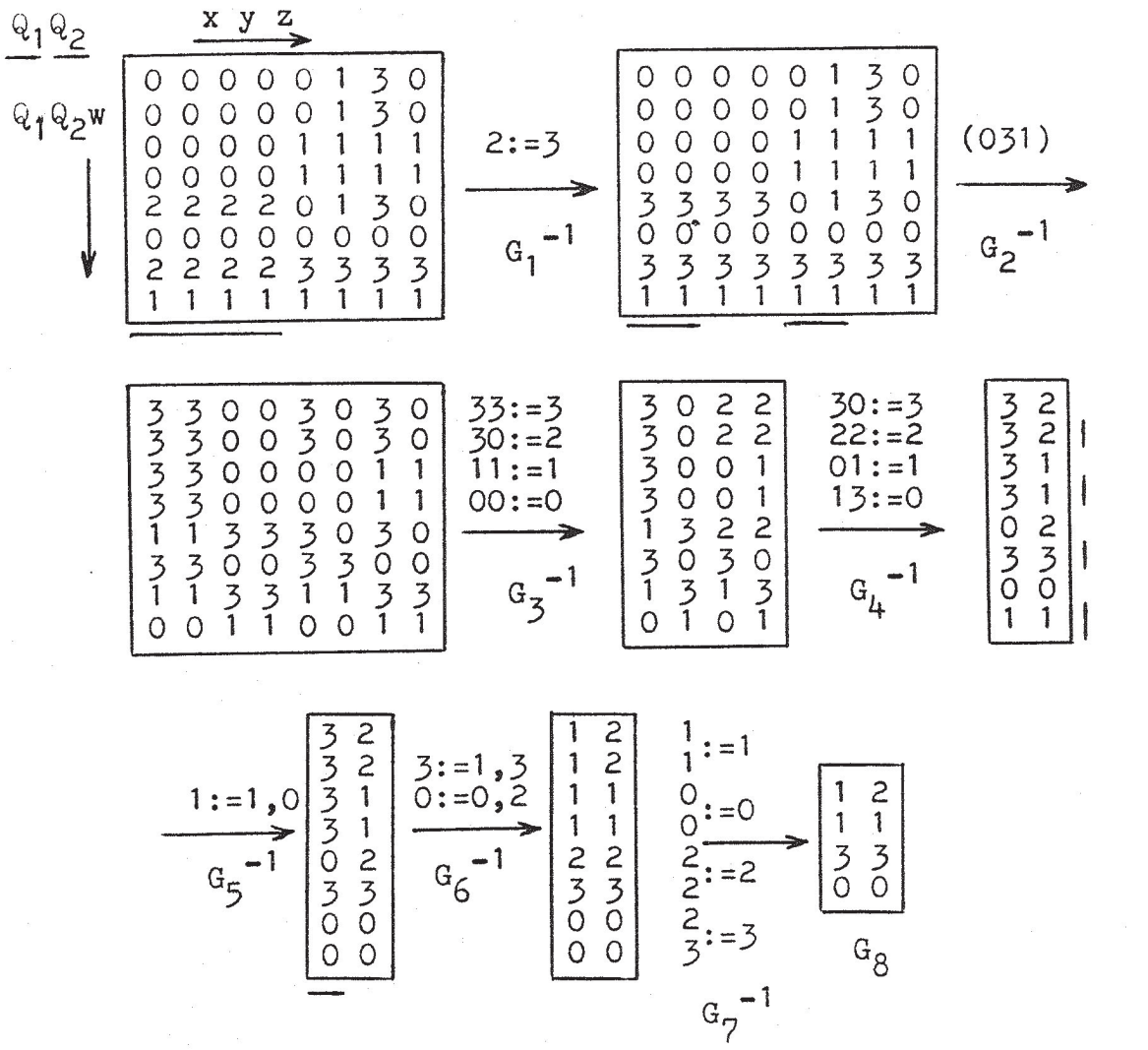
As an example, let us synthesize the microprogram to implement the finite state machine with the state diagram at Fig.14. Transformations of the state table are shown at Fig.15. Again the transformation and decomposition steps are interleaved in order to decrease the number of pairs (G_1,G_2,G_5,G_6) or to reduce the number of variables (G_3,G_4,G_7). The



$$\underline{Q_1} = \bar{w}\,(\,Q_1Q_2 + Q_1\bar{Q_2}\bar{x} + \bar{Q_2}xyz\,)$$

$$\underline{Q_2} = Q_2x + x\bar{y}z + xyz$$

Fig. 14. State diagram of the sample finite state machine



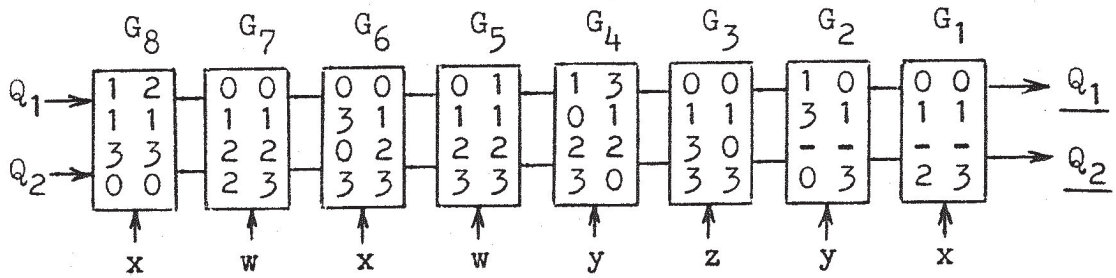


Fig.15. Microprogram synthesis for the state machine at Fig.14.

resulting microprogram is shown in a form of the cellular cascade at Fig.15. One state-machine cycle will take 8 microcycles on the Boolean processor. The required capacity of ROM in the LPU is 7 x 16 = 112 bits as $G_1 = G_7$.

The second example is a multiplier, perhaps the simplest one ever suggested, as it needs 2 flip-flops and 64 bits of ROM only. We shall assume that a multiplicand and a multiplier (of a chosen length) are stored in the bit-addressable local RAM and that the result will be saved there, too. We will synthesize the microprogram for processing of one bit of the multiplier (A) and one bit of the multiplicand (B), considering a partial product bit (Σ) and a carry to this bit (C). This microprogram will then be repeated many times over with different operands, under the control of

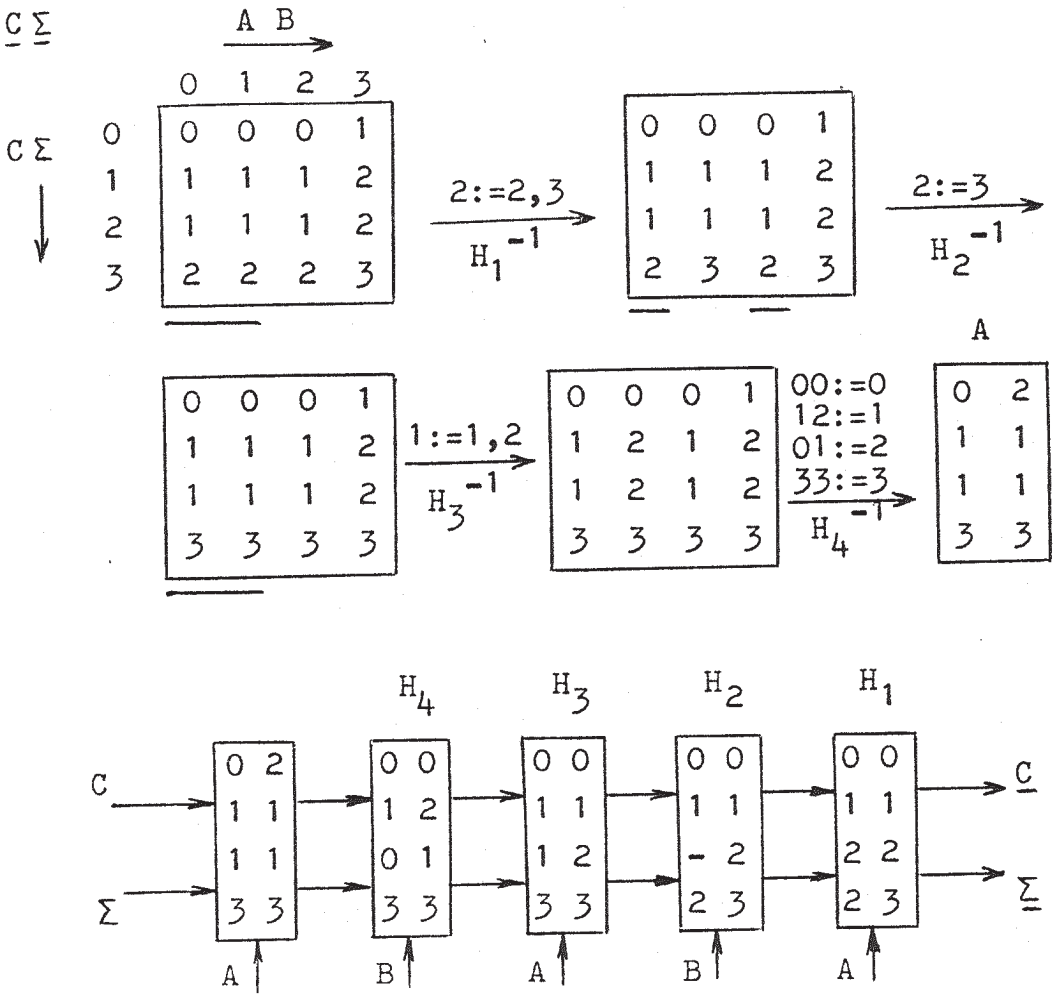


Fig.16. Microprogram segment for the serial binary multiplier

the central control memory. If Σ,C are taken as the present state, then the new state (underlined) will be

$$\underline{\Sigma} = C + \Sigma + A.B \qquad \underline{C} = C.\Sigma + A.B.C + \Sigma.A.B \tag{26}$$

The appropriate state table and its transformations are shown at Fig.16. The microprogram takes 5 microcycles, 4 distinct microoperations are needed.

8. CASCADE SYNTHESIS IN VLSI SYSTEM DESIGN

Both the spatial as well as temporal iterative decomposition of Boolean functions have applications in VLSI system design. The spatial decomposition can be used for synthesis of multi-level gate arrays and at multi-level partitioning of PLA's. Functions usually implemented by PLA's are sparse and the cascade synthesis may be attempted by making use of Theorem 2. In the design of PLA's the starting point is the list of specified vertices and function values. From this list the set of Boolean equations is obtained with a minimum number of products. This is comparable to the classical Boolean equation minimization of two-level logic, except that the size of each product is not important. Computer programs minimizing the number of products are available [7]. Sometimes large PLA's have to be broken into several smaller PLA's to meet certain design constraints such as the number of inputs, outputs or product terms. Research in partitioning of PLA's has concentrated mainly on parallel partitioning [7], where each input signal goes through one level of PLA and not on multi-level partitioning where the cascade synthesis is applicable. Multi-level partitioning may lead to more economical PLA layouts with slightly reduced speed due to the larger number of cascades.

As far as the temporal iterative decomposition is concerned, it may be used as a new microprogramming technique for Boolean processors in SIMD arrays. In the process of synthesis the microprogram itself as well as an optimum set of microoperations are derived simultaneously. Until now the approach to the selection of microoperations and to microprogram synthesis has been rather intuitive. At the beginning the set of microoperations supported by a certain combinational network of a fixed

structure is chosen and then the synthesis of all the required combinational and sequential operations is undertaken, either by exhaustive search on a computer or simply by using intuition [10]. If this approach fails to find the microprogram for every operation, new microoperations are added to the current list of microoperations, those microoperations never used are deleted, and the procedure is repeated over again. This results in changes in hardware (extra gates or changes in PLA) or in firmware (in case of ROM implementation of the LPU). At the cascade approach the list of microoperations is a by-product of the synthesis.

Microprogram synthesis can be illustrated by the following example. Let each four state-, one input- Boolean processor in a SIMD array is to carry out the following operations:

1. serial binary comparison
2. serial binary addition
3. serial binary multiplication
4. the set of symmetric combinational functions of five binary variables (1 out of 5, 2 out of 5, 3 out of 5, 4 out of 5, 5 out of 5 -AND)
5. the pair of functions of 4 variables

$f_1 = ad + \bar{c}d + bd + a\bar{b}\bar{c} + \bar{a}b\bar{c} + abc$

$f_2 = ad + \bar{c}d + b\bar{d} + \bar{b}d + a\bar{c} + \bar{a}\bar{b}c$

Find the microprograms for these operations and a list of microoperations.

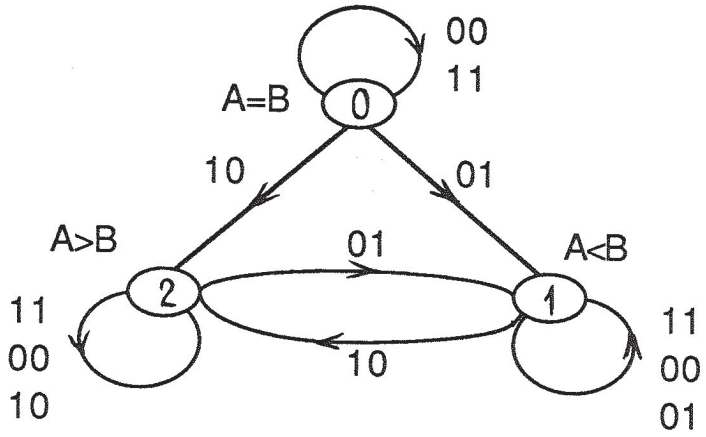
Solution.

1.

We can synthesize the sequential binary comparator with three states as shown on Fig.17. This solution would require three microcycles for each bit since the cascade is redundant. One can obtain a better solution from combinational comparator synthesized at Fig.6. If the first cell in the cascade is also decomposed according to input variables a_2, b_3, and a_3 respectively, the nonredundant cellular cascade with only two distinct cells is obtained (G_1 and G in Fig.6). These two cells alternate in the cascade of the length 2h for comparison of binary numbers h-bit long, Fig.18a.

2.

To implement the full binary adder, i.e. 2 functions (Σ_i, C_i) of 3 variables



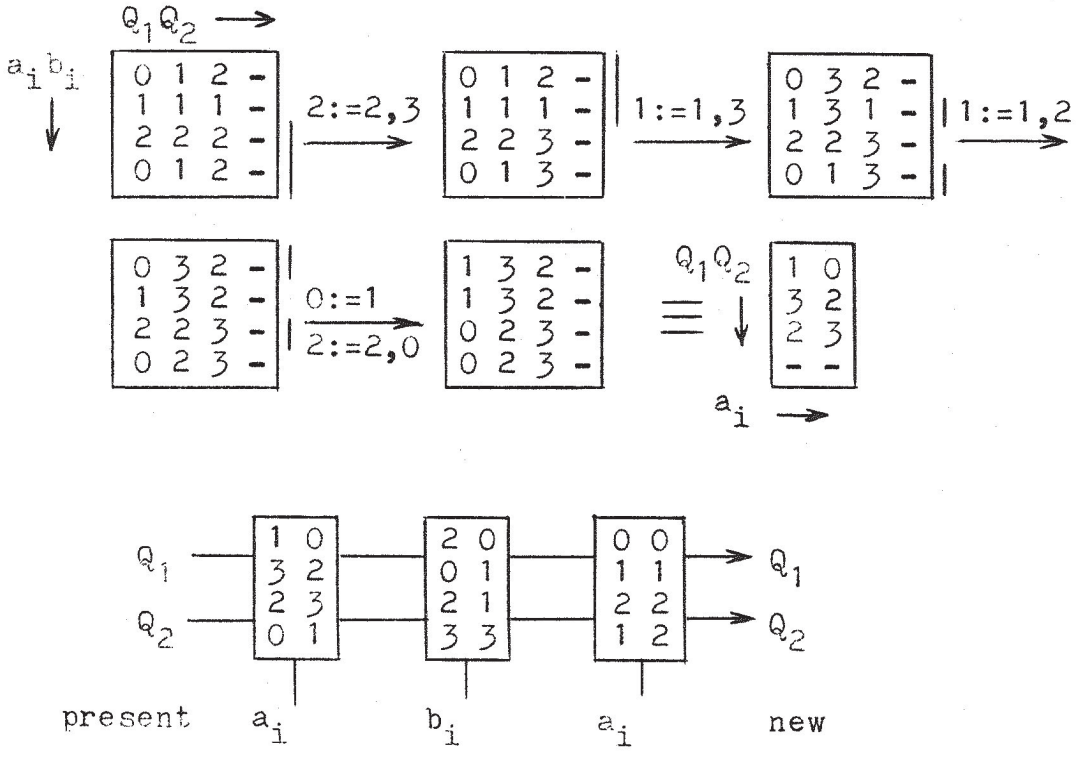


Fig.17. Cascade synthesis of a serial binary comparator

(a_i, b_i, C_{i-1}) we need just microoperation "clear" at the beginning and then two other microoperations : read a_i onto the lower rail and the full adder proper, Fig.18b.

3.

The serial binary multiplier can use just 4 distinct microoperations as it was shown in Fig.16. The rest is the matter of the microprogram in the control memory.

4.

The given set of symmetric combinational functions can be easily synthesized . The sample function 2 out of 5 is synthesized at Fig.18c, the rest is left to the reader. All the functions together require no more than 8 distinct microoperations.

5.

The pair of combinational functions can be synthesized simultaneously as shown at Fig.18d. The total number of distinct microoperations is 5 (including a shift used twice at the beginning to read input variables onto rails).

Altogether we can do with 21 microoperations, that means μop-code width 5 bits. These microoperations are strongly application -oriented and thus powerful. They enable to do specified operations the fastest possible way on the given hardware.

9. THE CELLULAR CASCADE APPROACH TO THE LOGIC CONTROLLER DESIGN

In this section we shall briefly describe two applications of cascade synthesis in the area of logic control, namely the design of microprocessor-based logic controllers and microprogrammed logic controllers.

The use of microprocessors as binary controllers is frequent not only in computer applications, but also in many different areas which have been formerly dominated by mechanical, electromechanical or pneumatic devices. Software controllers based on microprocessors and related elements have advantages in design flexibility, cost, and reliability. Their processing speed is quite satisfactory in many applications. When designing such binary controllers, one may refer to standardized hardware structure

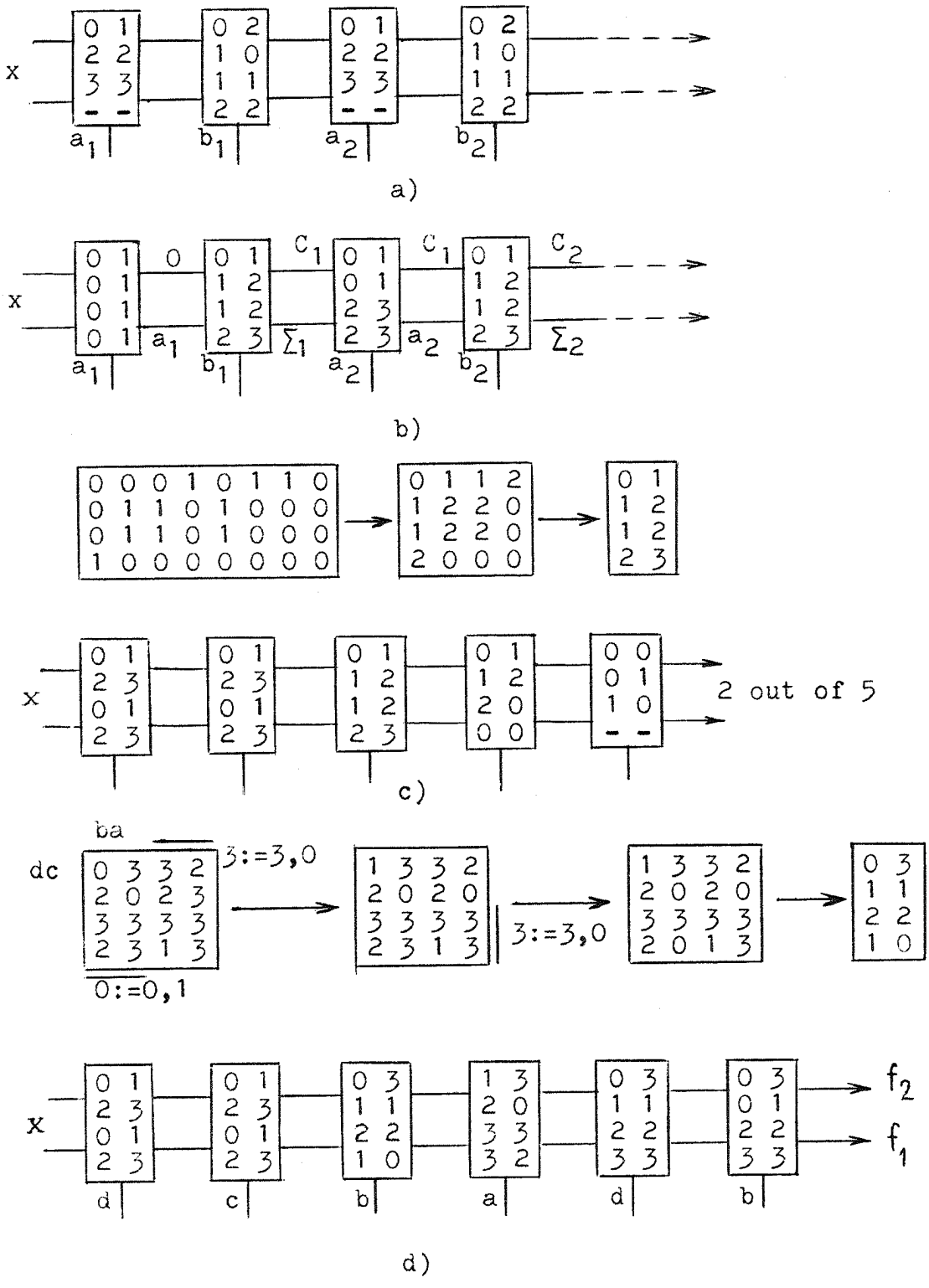


Fig.18. Some cellular cascades a) a binary comparator b) the binary adder c) 2 out of 5 d) functions f_2, f_1

of a state machine in a form of PLA with D-type flip-flops, that can be easily simulated on a microprocessor. The program that emulates a PLA controller works in the loop. One processing cycle consists of testing of input variables, generating the output and the transition to the next state. We will omit the details of synchronization and concentrate on the main task: PLA simulation on a microprocessor.

The simulation of a Boolean function is quite common on a microprocessor. To simulate a multiple output Boolean function of the PLA, two programming techniques can be used. The first technique is based on the exact emulation of PLA hardware, i.e. a logical product generator and a logical summer. For each input variable there are two masks stored in the memory that are used to modify a current vector of values of all distinct implicants. At the beginning this implicant vector is set to all 1's. Input variables are then serially tested and, depending on the value of the input variable under the test, one of two masks is selected to modify, by logical AND operation, the contents of the implicant vector. After testing all the input variables the outputs of PLA are generated serially using another set of masks determining which implicants participate in a particular output. If we consider a PLA with n inputs, r outputs and as many as p product terms, then the memory capacity to store the required masks will be in case of the 8-bit microprocessor

$$C_1 = \lceil p/8 \rceil . (2n + r) \quad \text{bytes} \tag{27}$$

since the masks are $\lceil p/8 \rceil$ bytes wide.

Until now we have evaluated all the scalar components of a certain vector $Y = F(X) = \{y_i\}$ where

$$y_i = f_i (x_1 , x_2 , ..., x_n) , \qquad i = 1, 2, ..., r \tag{28}$$

The second technique of PLA simulation starts directly from the defined input states and associated output vectors [11]. Here we evaluate the other set of functions g_j, $j = 1,2, ..., t$ such that

$$g_j (x_1, x_2, ..., x_n) = 1 \equiv F (X) = Y_j \tag{29}$$

In this case the PLA is described as a switch

$$Y = g_1(X).\ Y_1 + g_2(X).\ Y_2 + \ldots + g_t(X)\ .Y_t \qquad (30)$$

Notice that transformation from g_j to f_i is done in the process of PLA synthesis.

In the second technique the given input pattern is searched for in the table of all input patterns for which the output patterns are defined. This is done by sequence of comparisons until coincidence is reached. Then we set the corresponding output pattern at the outputs and a procedure is terminated. The table for a particular PLA will have to contain, for every specified input pattern

- a mask that determines don't care variables
- the input pattern with redundant variables set to zero for convenience and
- the output pattern corresponding to the given input pattern.

The storage capacity required to store the table will be thus

$$C_2 = s\ .\ (\ 2\ \lceil n/8 \rceil + \lceil r/8 \rceil\ ) \qquad (31)$$

where s is the number of input patterns with defined output patterns, generally $s \neq p$. The programs for this kind of PLA simulation can be found in [11].

The third technique of PLA simulation relies on multi-level decomposition of a combinational network as described in previous sections. Let us discuss the implementation of a large PLA on the 8-bit microprocessor. Let n = 16, r = 8 and the number of product terms $p \leq 64$ for convenient implementation on a 8-bit microprocessor. The structure of a program for PLA simulation can be derived directly from Fig.1b where k = 6, m = 2, B = 5 and r = 8. Function tables of cells H_i are stored in ROM so that required memory capacity is 5x256=1280 bytes , in contrast to 16 kbytes required by the single function table of the original network. The program executes 5 times memory read operation. Every time the data read out of the memory location is completed by values of two input variables - and the result serves as the next memory address. It may be seen that the response will be short and constant. According to Theorem 2

any PLA can be implemented in the suggested format that is specified at not more than 64 vertices. But many PLA's specified in more than 64 vertices will also be realizable because due to the don't cares the number of distinct M-tuples (in our case 4-tuples) will be lower than the number of specified vertices, some vertices will represent only one M-tuple. When combining vertices to distinct M-tuples , one should try to minimize the number of distinct M-tuples.

The second type of a logic controller that is to be discussed here is the most common microprogrammed controller with two-way branching evoked by a selected variable x_i . This controller accepts instructions of the following types:

1. if $x_i = 0$ then go to L_a else go to L_b
2. [out M_h ; go to L_j]

where { L_k } is a set of labels and { M_h } a set of distinct output vectors. It can be programmed to implement a general multi-way switch used in logic control, namely

if $f_j(x_1, x_2, \ldots, x_n) = 1$ then [out $M_{(j)}$; go to $L_{(j)}$] (32)

where { $f_j(x_1, x_2, \ldots, x_n)$ } is a set of orthogonal Boolean functions, $j = 1, 2, \ldots, K$ and usually $K << 2^n$.

The microprogrammed controller can perform according to one of three algorithms:

1.

Scan the disjuctive normal form (DNF) of f_j from left to right. If the term just scanned contains x_i and the value of x_i is 0, or it contains $\overline{x_i}$ and the value of x_i is 1, go to the first letter of the next term (if there is no next term, return the value 0). Else go to the next symbol. If the next symbol is "+" or there is no next symbol, return the value 1.

The processing time of this algorithm is data-dependent; its upper limit depends on the number of symbols in the DNF and on the number of branches K from the given state to next states, as functions f_j are evaluated one by one.

2.

Algorithms known as binary decision trees (optimum binary decision

trees), [2]. Here the processing time is given by the number of input variables and the length of the microprogram is up to

$$\frac{2^s}{s}\left(1 + \frac{2s}{s - \log_2 s}\right) - 1 \qquad (33)$$

microinstructions, where $s = n + \log_2 r$. The class of controllers based on the binary decision programs has been investigated in recent literature [2], [12].

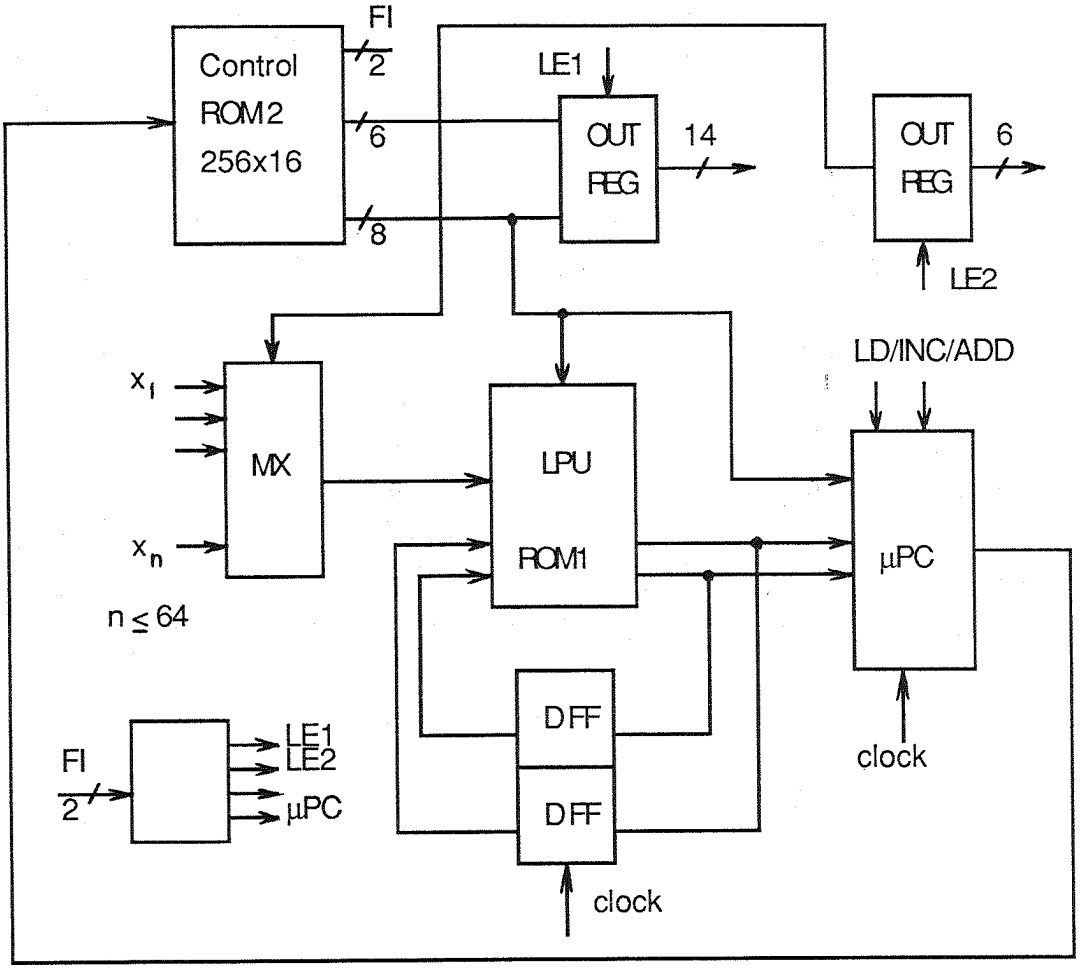
3.

Algorithm of parallel evaluation of a "branch code" on the Boolean processor based on the iterative decomposition. If there are in any given state K branches to next states, we evaluate just $k = \lceil \log_2 K \rceil$ Boolean functions. Here the execution time is determined by the number of input variables as in binary decision programs (nonredundant cascades) or slightly higher (redundant cascades). Generally the length of microprograms is much lower than in binary decision controllers. This is due to the tree-like structure of binary programs and a linear nature of microprograms obtained by iterative decomposition.

The example of the very simple logic controller driven by the Boolean processor is at Fig.19. It is a generalization of the well-known architecture of a controller with two-way branching according to the selected variable and with microprogram counter (μPC). Four microinstruction formats are defined by two format indicating bits, Fig.19b. Each EVAL microinstruction initiates one microcycle of the Boolean processor and increments the μPC. The BRANCH microinstructions does the same except that the contents of μPC is increased by 1,2,3, or 4 (incidentally by 1,3,5, or 7) depending on a branch code at the output of the LPU (0,1,2, or 3). Two control signals entering the μPC define its operation : parallel load, increment, or add branch code +1. There are two types of OUT microinstructions, one incrementing μPC and another with the unconditional jump to the specified address. In the former case outputs are the Moore-type, in the later case the Mealy-type outputs.

The typical sequence of microoperations in one state may look like this:

EVAL, . . . , EVAL, BRANCH, OUT , OUTJMP (Mealy automaton)



a)

	format	6 bits	8 bits
EVAL		addr. of x_i	µop-code LPU
BRANCH		addr. of x_i	µop-code LPU
OUT		output 2	next address
OUTJMP		output 1	output 1

b)

Fig.19. Architecture of the logic controller driven by the Boolean processor a) block diagram b) µinstruction formats

OUT, EVAL, . . . , EVAL, BRANCH, JMP (Moore automaton)
OUT, EVAL, . . . , EVAL, BRANCH, OUTJMP (hybrid automaton)

10. CONCLUSION

The suggested technique of iterative decomposition of digital systems is a tool that may be useful in different areas of logic design and digital system architecture. This preliminary report is felt to be just the introduction into the topic that should be explored more thoroughly in the near future.

The decomposition method described so far can be implemented on a computer. As the first results show, decomposition steps can be aided by computer even for large number of variables (more than 16), so that multi-level partitioning of PLA's is possible. Transformation steps are more difficult to do by a computer unless we select only a certain subclass of all possible transformations. Even then is the exhaustive search successful for small problems only (up to 8 variables). The use of pattern recognition techniques have not been tried out.

As it is, the suggested technique is also a useful tool for microprogramming large arrays of very simple Boolean processors processing serially one up to several bits at a time. Its advantage is in speed of execution of microprograms for a given task. Since the microoperations are application-oriented, they are more powerful than just AND, OR, XOR and ADD; microprograms are shorter and run faster.
The future research should investigate the question of realizability of given operations on the Boolean processor with certain parameters k and m or determining required values of these parameters.

The suggested method for PLA simulation makes it possible to implement large PLA's with a fast response on microprocessors at reasonable memory requirements. This enables the design of high-performance microprocessor-based binary controllers as well, since some of the PLA outputs may be fed back to the inputs. Conversion of PLA specification to data tables in ROM may be automated. An appropriate project is under way.

The class of microprogrammed controllers driven by the Boolean processor also deserves further research, especially how they compare to

binary decision controllers studied in [2] and [12]. As yet, the Boolean processor- driven controllers seem to require less space of the control memory ; response time is similar in both the cases.

REFERENCES

[1] R.C.Minnick, Cutpoint Cellular Logic. IEEE Trans.El.Comput., Vol.EC-13,pp.685-698, Dec.1964.

[2] M.David, J.P.Deschamps, A.Thayse, Digital Systems With Algorithm Implementation. J.Wiley, New York, 1983.

[3] V.Dvorak, A Microprogrammed Controller Based on the Logic Processing Unit. Microprogramming and Microprocessing 16, No.4 & 5, 1985, pp.341-344.

[4] K. Hwang, F. A. Briggs, Computer Architecture and Parallel Processing. McGraw Hill, Inc., 1984.

[5] H.A.Curtis, A New Approach to the Design of Switching Circuits. Van Nostrand Comp., Inc., Princeton, New Jersey, 1962.

[6] J.P.Roth, R.M.Karp, Minimization Over Boolean Graphs. IBM Journal Res. and Dev., April 1962, pp.227-238.

[7] S.Kang, Synthesis and Optimization of Programmable Logic Arrays. Ph.D. Thesis, Stanford University, 1981.

[8] V.Dvorak, Two-Rail Cascade Synthesis of Boolean Functions. IEEE Trans. Comput., Vol. C-17, June 1968, pp.592-596.

[9] M.Yoeli, A Group-Theoretical Approach to Two-Rail Cascades. IEEE Trans. El.Comput., Vol. EC-14, pp. 815-822, Dec. 1965.

[10] K.Preston, Modern Cellular Automata; Theory and Applications. Plenum, New York, 1984.

[11] V.Dvorak, I.Rukovansky, Efficient Implementation of Logic Control Algorithms on Microprocessors. Information Processing 83, ed. R.E.A.Mason. Elsevier Science Publ., North Holand, 1983.

[12] P.J.A.Zsombor - Murray et al., Binary-Decision-Based Programmable Controllers, Part I - III, IEEE Micro, Vol.3, August 1983 (pp.67-83), October 1983(pp.16-26), December 1983(pp.24-39).