2011 41st IEEE International Symposium on Multiple-Valued Logic

Index Generation Functions: Recent Developments

Tsutomu Sasao
Department of Computer Science and Electronics,
Kyushu Institute of Technology,
Iizuka 820-8502, Japan

Abstract—**This survey first introduces index generation functions, which are useful for pattern matching in communication circuits. Then, it shows various methods to realize index generation functions using memories. A linear transformation is used to reduce the number of variables and thus memory size. An extension to the multiple-valued case is also presented.**

I. Index Generation Function

This paper surveys recent results on index generation functions. Applications of index generation functions include: IP address table lookup, packet filtering, terminal access controllers, memory patch circuits, virus scan circuits, fault maps for memory, and pattern matching. In addition, this paper introduces an index generation unit that efficiently realizes an index generation function by a linear circuit and a pair of smaller memories. Due to space limitations, definitions of standard terminology used in switching circuit theory [11] are omitted.

Definition 1.1: Consider a set of k different binary vectors of n bits. These vectors are **registered vectors**. For each registered vector, assign a unique integer from 1 to k. A **registered vector table** shows the **index** of each registered vector. An **index generation function** produces the corresponding index if the input matches a registered vector, and produces 0 otherwise. k is the **weight** of the index generation function. An index generation function represents a mapping: $B^n \rightarrow \{0, 1, 2, \ldots, k\}$. An **index generator** is a circuit that realizes an index generation function.

Example 1.1: Table 1.1 shows a registered vector table with $k = 4$ vectors.

An index generation function can be directly implemented by a content addressable memory (CAM). However, a CAM dissipates much power. So, in this paper, we use conventional memories.

The rest of the paper is organized as follows: Section II discusses applications of index generation functions. Section III shows properties of incompletely specified index generation functions. Section IV shows a method to represent incompletely specified index generation functions using fewer variables by a linear transformation. Section V shows an effect of using linear transformation. Section VI explains the operation of an index generation unit. Section VII shows design examples of m-out-of-n code converters. Section VIII shows efficient methods to realize index generation functions. Section IX extends the theory to multiple-valued input functions. Section X surveys related works on linear transformations. Section XI concludes the paper.

TABLE 1.1
Registered vector table.

Vector				Index
x_1	x_2	x_3	x_4	
0	0	1	0	1
0	1	1	1	2
1	1	0	0	3
1	1	1	1	4

II. Applications

Index generators are used for address tables in the internet, terminal access controllers for local area networks, databases, memory patch circuits, electronic dictionaries, password lists, etc. [14].

A. Address Table in the Internet

IP addresses of the internet are often represented in 32 bits. An **address table** for a router stores IP addresses and corresponding indexes for a memory that stores the details of the addresses. For example, in a typical problem, the number of addresses in the table is $40,000$. Thus, the number of inputs is 32 and the number of outputs is 16, which can represent 65,536 bit patterns. Note that the address table must be updated frequently.

B. Terminal Access Controller

A **terminal access controller** (TAC) for a local area network checks whether the requested terminal has permission to access Web resources outside the local area network, E-mail, FTP, Telnet, etc.. In Fig. 2.1, eight terminals are connected to the TAC. Some can access all the resources. Others can access only limited resources because of security risks. The TAC checks whether the requested computer has permission to access the Web, E-mail, FTP, Telnet, or not. Each terminal has its unique **MAC address** represented by 48 bits. We assume that the number of terminals in the table is at most 255. To implement the TAC, we use an index generator and a memory. The memory stores the details of the terminals. The number of inputs for the index generator is 48 and the number of outputs is 8. In many cases, the table for the terminal access controller must be updated frequently.

Example 2.1: Fig. 2.2 shows an example of the terminal access controller. The first terminal has the MAC address 53:03:74:59:03:02. It is allowed to access everything, including is the Web outside the local area network, E-mail, FTP,

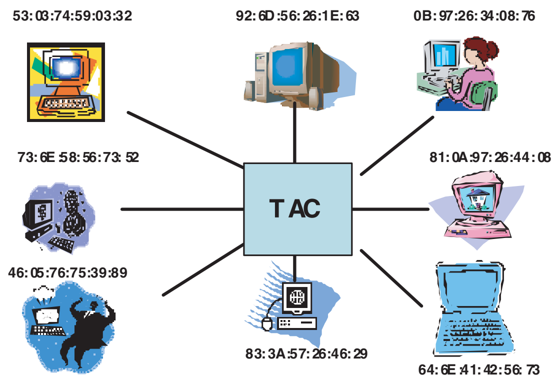


Fig. 2.1. Terminal access controller.

Index Generator	
MAC Address	Index
53:03:74:59:03:32	1
92:6D:56:26:1E:63	2
0B:97:26:34:08:76	3
73:6E:58:56:73:52	4

$\Rightarrow$

Memory				
Address	DATA			
	Web	E-mail	FTP	Telnet
1	1	1	1	1
2	1	1	0	0
3	1	0	0	0
4	0	1	0	0

48-bit 4-bit

Fig. 2.2. Index generator for terminal access controller.

and Telnet. The second one is allowed to access both the Web and E-mail. The third one is allowed to access only the Web. And, the last one is allowed to access only E-mail. The index generated by the index generator is used as an address to read the memory which stores the permissions. If we implement the TAC by a single memory, we need a memory with 256 Tera words, since the number of inputs is 48. To reduce the size of memory, we use an index generator to produce the index, and an additional memory to store the permission data for each internal address. ■

The index generators in the previous examples have common properties:

1) The values of the non-zero outputs are distinct.
2) The number of non-zero output values is much smaller than the total number of the input combinations.
3) High-speed circuits are required.
4) Data must be updated.

The third property is important in the communication networks. The last property requires that index generators must be programmable.

III. Incompletely Specified Index Generation Functions

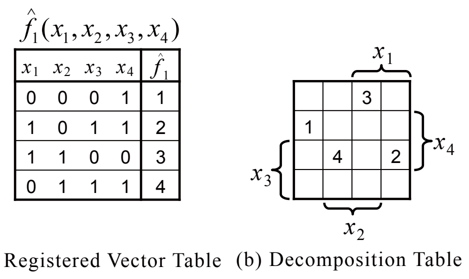
In this part, we introduce some methods to represent a given incompletely specified function with fewer variables [15], [16], [18].

Definition 3.1: Let $D = \{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k\}$ be a set of k distinct vectors in B^n, where $B = \{0, 1\}$. $\hat{f} : B^n \rightarrow \{1, 2, \ldots, k, d\}$ is an **incompletely specified index generation function with weight** k if

$$\begin{aligned} \hat{f}(\vec{a}_i) &= i, \ (when\ \vec{a}_i \in D), \text{ and} \\ \hat{f}(\vec{b}) &= d, \ (when\ \vec{b} \in B^n - D), \end{aligned}$$

where d denotes *don't care* or undefined.

The number of variables to represent incompletely specified index generation functions can be often reduced.



x_1	x_2	x_3	x_4	$\hat{f}_1$
0	0	0	1	1
1	0	1	1	2
1	1	0	0	3
0	1	1	1	4

Fig. 3.1. Reduction of variables to represent an incompletely specified index generation function.

Example 3.1: Consider the registered vector table shown in Fig. 3.1(a). It defines a 4-variable incompletely specified index generation function $\hat{f}_1(X)$. Let $X_1 = (x_1, x_2)$ and $X_2 = (x_3, x_4)$. The corresponding decomposition table for $\hat{f}_1(X)$ is shown in Fig. 3.1(b), where blank cells denote *don't cares*. In this function, for the vectors $\vec{a}_1 = (0, 0, 0, 1)$, $\vec{a}_2 = (1, 0, 1, 1)$, $\vec{a}_3 = (1, 1, 0, 0)$, and $\vec{a}_4 = (0, 1, 1, 1)$, the values of functions are $\hat{f}_1(\vec{a}_1) = 1$, $\hat{f}_1(\vec{a}_2) = 2$, $\hat{f}_1(\vec{a}_3) = 3$, and $\hat{f}_1(\vec{a}_4) = 4$, respectively. For other inputs, the values of $\hat{f}_1$ are d (don't care).

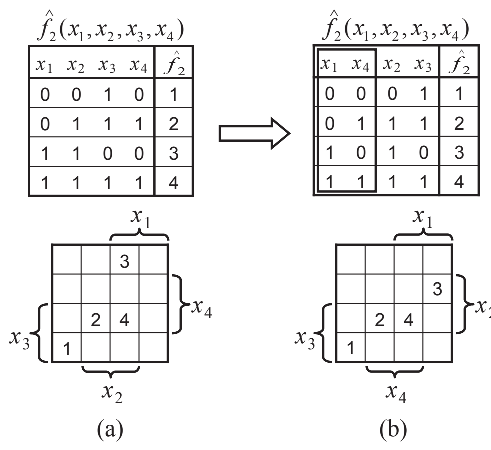
In the decomposition table, when each column has at most one specified element, then the function can be represented by column variables only, since, for each column, the values of all *don't cares* can be set to the specified value of the column. In Fig. 3.1(a), values for (x_1, x_2) are distinct, and the index can be specified by using only these two variables:

$$f_1 = 1 \cdot \bar{x}_1 \bar{x}_2 \vee 2 \cdot x_1 \bar{x}_2 \vee 3 \cdot x_1 x_2 \vee 4 \cdot \bar{x}_1 x_2.$$

■

Example 3.2: Consider the registered vector table in Fig. 3.2, and the decomposition table for an incompletely specified index generation function $\hat{f}_2$. Consider the number of variables to represent the function. In the decomposition table in Fig. 3.2(a), two non-zero elements exist in the column $(x_1, x_2) = (1, 1)$. Thus, the function $\hat{f}_2$ cannot be represented by $\{x_1, x_2\}$. Similarly, in the row $(x_3, x_4) = (1, 1)$, two non-zero elements exist, and the function $\hat{f}_2$ cannot be represented by $\{x_3, x_4\}$, either.

Next, let us change the partition of the input variables into (x_1, x_4) and (x_2, x_3) as shown in Fig. 3.2(b). In this case, each column has at most one specified element. Note that in the registered vector table in Fig. 3.2(b), values of the vectors (x_1, x_4) are all different. Thus, the function $\hat{f}_2$ can be represented by using only $\{x_1, x_4\}$. ■



x_1	x_2	x_3	x_4	$\hat{f}_2$
0	0	1	0	1
0	1	1	1	2
1	1	0	0	3
1	1	1	1	4

x_1	x_4	x_2	x_3	$\hat{f}_2$
0	0	0	1	1
0	1	1	1	2
1	0	1	0	3
1	1	1	1	4

Fig. 3.2. Reduction of variables to represent an input incompletely specified index generation function.

TABLE 3.1
AVERAGE NUMBER OF VARIABLES TO REPRESENT INCOMPLETELY SPECIFIED INDEX GENERATION FUNCTION.

k	$n = 16$	$n = 20$	$n = 24$	$2\lceil\log_2(k+1)\rceil - 3$
7	3.052	3.018	3.003	3
15	4.980	4.947	4.878	5
31	6.447	6.115	6.003	7
63	8.257	8.007	8.000	9
127	10.304	10.000	9.963	11
255	12.589	11.996	11.896	13
511	14.890	14.019	13.787	15
1023	15.991	16.293	15.874	17
2047	16.000	18.758	17.965	19
4095	16.000	19.992	20.093	21

As shown in above examples, incompletely specified index generation functions often can be represented with fewer variables. Minimization methods of input variables for single-output incompletely specified functions are considered in [2], [4], [12], [17].

Lemma 3.1: Let p be the number of variables to represent an incompletely specified index generation function with weight k. We have the following relation:

$$p \geq \lceil\log_2(k+1)\rceil.$$

(Proof) Assume that $k+1 = 2^p$. Consider the binary decision tree for the function. To distinguish 2^p different terminals, at least p variables are necessary. Note that one terminal node is used to represent non-registered vectors. □

Definition 3.2: A set of functions is **uniformly distributed**, if the probability of occurrence of any function is the same as any other function.

For example, the set of two-valued input two-valued output 4-variable incompletely specified functions with weight 1 consists of 32 members, 16 having a single 1 and 16 having a single 0. If the functions are uniformly distributed, the probability of the occurrence of any one of them is $\frac{1}{32}$. Table 3.1 shows average numbers of variables to represent incompletely specified index generation functions for different n and different weight k. From the table, we have the following:

Conjecture 3.1: Consider a set of uniformly distributed incompletely specified index generation functions of n binary input variables with weight $k \geq 7$, then the fraction of the functions represented with $p = 2\lceil\log_2(k+1)\rceil - 3$ variables approaches 1.0 as n increases.

Although there exist functions that require more than $p = 2\lceil\log_2(k+1)\rceil - 3$ variables, the fraction of such functions approaches 0.0 as n increases. When the value of k is large, the memory with $p = 2\lceil\log_2(k+1)\rceil - 3$ inputs is still too large to implement.

IV. REPRESENTATION OF INDEX GENERATION FUNCTIONS USING LINEAR TRANSFORMATIONS

In this part, we show a method to reduce the number of variables to represent an incompletely specified function by using a linear transformation of the input variables.

Definition 4.1: Consider a function $f(x_1, x_2, \ldots, x_n)$. A **compound variable** y has a form

$$y = c_1x_1 \oplus c_2x_2 \oplus \cdots \oplus c_nx_n,$$

where $c_i \in \{0, 1\}$. The **compound degree** of y is $\sum_{i=1}^{n} c_i$. A variable with the compound degree 1 is a **primitive variable**. A variable with compound degree 2 is a **bi-compound variable**, and a variable with compound degree 3 is a **tri-compound variable**.

Example 4.1: Consider the incompletely specified index generation function $\hat{f}_3$ shown in Fig. 4.1. Let us consider the number of variables to represent this function. In Fig. 4.1(a), the column $(x_1, x_2) = (1, 1)$ has two non-zero elements. So, the function cannot be represented by $\{x_1, x_2\}$. In a similar way, the row $(x_3, x_4) = (1, 1)$ has two non-zero elements. So, the function cannot be represented by $\{x_3, x_4\}$. Note that the decomposition tables with other partitions produce the same results. Thus, to represent the function $\hat{f}_3$, at least three variables are necessary. Next, consider the bi-compound variables $y_1 = x_1 \oplus x_2$ and $y_2 = x_2 \oplus x_3$. In this case, we have the function $\hat{g}_3(y_1, y_2, x_3, x_4)$ shown in Fig. 4.1(b). Note that, in the decomposition table shown in Fig. 4.1(b), each column has at most one specified element. Thus, the function $\hat{g}_3$ can be represented by using only two variables $\{y_1, y_2\}$. ■

As shown in the above example, by using linear transformation, the number of input variables for incompletely specified index generation functions can be further reduced. In the rest of the paper, both a primitive variable x_i and and a compound variables y_j are treated as input variables.

Definition 4.2: Given an incompletely specified index generation function, the linear transformation that minimizes the number of variables is **optimum**.

When the number of variables satisfies the relation $p = \lceil\log_2(k+1)\rceil$, it is an optimum linear transformation.

When only primitive variables are used, the number of variables for an incompletely specified index generation function can be minimized by solving a kind of a minimum covering problem [12], [17]. In principle, the minimization

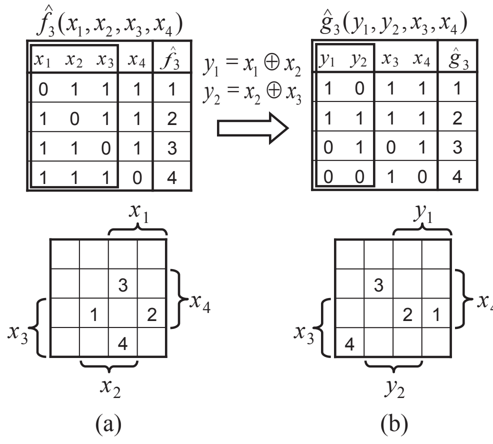


Fig. 4.1. Incompletely specified index generation function represented by compound variables.

of variables using both primitive and compound variables can be done in the same way. That is, we can perform the minimization of the variables, where not only the primitive variables $x_1, x_2, \ldots, x_n$, but also the compound variables $y_1, y_2, \ldots, y_t$ can be considered as the input variables. When both the primitive and the bi-compound variables are used, the number of the input variables to consider is

$$n + \binom{n}{2} = \frac{n(n+1)}{2}.$$

When tri-compound variables, in addition to the bi-compound and the primitive variables are used, the number of the variables to consider is

$$n + \binom{n}{2} + \binom{n}{3} = \frac{n(n^2+5)}{6}.$$

If we consider all the compound variables, the total number of (compound) variables would be $2^n - 1$. Thus, an exhaustive method would be impractical.

In [18], we developed the **information gain method**, a heuristic method to select compound variables. The selection of the compound variables can be considered as the optimization of a binary decision tree.

Definition 4.3: In an incompletely specified index generation function, the **partition difference** with respect to x_i is $|h_{i0} - h_{i1}|$, where h_{i0} is the number of registered vectors such that $x_i = 0$, and h_{i1} is the number of registered vectors such that $x_i = 1$.

Variables with a smaller partition difference tend to partition the set of registered vectors such that the bits among registered vectors tend to have nearly the same 0's and 1's. Let k be the number of registered vectors. When the given set of variables partitions the set of vectors into balanced sets, the number of variables to represent the function is reduced to $\lceil \log_2(k+1) \rceil$.

When selecting compound variables, the smaller the partition difference of a variable, the larger the information gain we obtain. Thus, a variable with a large information gain (i.e., a variable with a small partition difference) tends to reduce the number of variables needed to represent the function.

TABLE 5.1
1-OUT-OF-15 TO BINARY CONVERTER.

1-out-of-15 code															Index
x_{15}	x_{14}	x_{13}	x_{12}	x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	3
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	4
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	5
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	6
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	7
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	8
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	9
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	10
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	11
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	12
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	13
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	14
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15

V. EFFECT OF LINEAR TRANSFORMATIONS

Consider index generation functions with weight k. When the probabilities of 0's and 1's in the registered vector table are nearly the same, the function may be represented with $p = \lceil \log_2(k+1) \rceil$ variables. On the other hand, when the probabilities of 0's and 1's are quite different, many variables are necessary to represent the function. As an example of functions where the probability of 0's and 1's in the registered vector table are quite different, we consider a class of code converters.

Definition 5.1: An m**-out-of-**n **code** consists of $\binom{n}{m}$ binary code words whose weights are m.

Definition 5.2: An m**-out-of-**n **to binary converter** realizes an index generation function with $\binom{n}{m}$ non-zero elements. It has n inputs and $\lceil \log_2[\binom{n}{m}+1] \rceil$ outputs. When the number of 1's in the inputs is not m, the converter produces the all 0 code. The m-out-of-n code is produced in ascending lexicographical order. That is, the smallest number is denoted by $(0, 0, \ldots, 0, 1, 1, \ldots, 1)$, while the largest number is denoted by $(1, 1, \ldots 1, 0, 0, \ldots, 0)$.

Example 5.1: Consider the 1-out-of-15 code to binary converter $\hat{f}(x_1, x_2, \ldots, x_{15})$. It is an index generation function with weight $k = 15$, whose registered vector table is shown in Table 5.1. When only the primitive variables are used, at least 14 variables are necessary to represent the function. Next, consider the linear transformation:

$$\begin{aligned}
y_1 &= x_1 \oplus x_3 \oplus x_5 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{13} \oplus x_{15}, \\
y_2 &= x_2 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{14} \oplus x_{15}, \\
y_3 &= x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15}, \\
y_4 &= x_8 \oplus x_9 \oplus x_{10} \oplus x_{11} \oplus x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15}.
\end{aligned}$$

Then, $\hat{f}$ can be represented by $g(y_1, y_2, y_3, y_4)$ as shown in Table 5.2. In this case, we can assume that, in the inputs $(x_1, x_2, \ldots, x_{15})$, only one variable takes the value 1, while the other variables take the value 0. Note that in the original registered vector table in Table 5.1, the probability of 1's is $1/15$, while in the transformed registered vector table shown in Table 5.2, the probability of 1's is $8/15$. The linear transformation makes the height of the decision tree small,

TABLE 5.2
TRANSFORMED 1-OUT-OF-15 TO BINARY CONVERTER.

Transformed code				Index
y_4	y_3	y_2	y_1	
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

and reduces the number of variable to represent the function. Since the function is represented with $p = \lceil \log_2(k+1) \rceil = 4$ variables, it is an optimum linear transformation. ■

The next theorem shows how a linear transformation changes the probability of a variable taking the value 1.

Theorem 5.1: Let $x_1, x_2, \ldots, x_n$ be independent variables. Let x_i take the value 1 with the probability α. Then, the probability that the function

$$y = x_1 \oplus x_2 \oplus \cdots \oplus x_t$$

takes the value 1 is

$$\gamma_t(\alpha) = \sum_{i=1,3,5,\ldots,t-1} \binom{t}{i} \alpha^i \beta^{t-i} \text{ (when } t \text{ is even)},$$

$$\gamma_t(\alpha) = \sum_{i=1,3,5,\ldots,t} \binom{t}{i} \alpha^i \beta^{t-i} \text{ (when } t \text{ is odd)},$$

where $\beta = 1 - \alpha$.

Example 5.2: In the 1-out-of-15 code to binary converter, x_i takes the value 1 with the probability $1/15 = 0.066666$. After the linear transformation, y_i takes the value 1 with the probability

$$\gamma_t(\alpha) = \sum_{i=1,3,5,\ldots,7} \binom{8}{i} \alpha^i \beta^{8-i} = 0.340857.$$

The linear transformation gives a more balanced decision tree, and reduces the number of variables needed to represent the function. ■

VI. INDEX GENERATION UNIT

Fig. 6.1 shows an **index generation unit** (IGU) [15], [16], [17]. The **linear circuit** has n inputs and p outputs, where $p < n$. It produces functions:

$$\begin{aligned} y_1 &= c_{1,1}x_1 \oplus c_{1,2}x_2 \oplus c_{1,3}x_3 \oplus \cdots \oplus c_{1,n}x_n \\ y_2 &= c_{2,1}x_1 \oplus c_{2,2}x_2 \oplus c_{2,3}x_3 \oplus \cdots \oplus c_{2,n}x_n \\ y_3 &= c_{3,1}x_1 \oplus c_{3,2}x_2 \oplus c_{3,3}x_3 \oplus \cdots \oplus c_{3,n}x_n \\ \cdots &= \cdots \\ y_p &= c_{p,1}x_1 \oplus c_{p,2}x_2 \oplus c_{p,3}x_3 \oplus \cdots \oplus c_{p,n}x_n, \end{aligned}$$

where $c_{i,j} \in \{0,1\}$, and $c_{i,i} = 1$. It is used to reduce the size of the main memory. Let $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$.

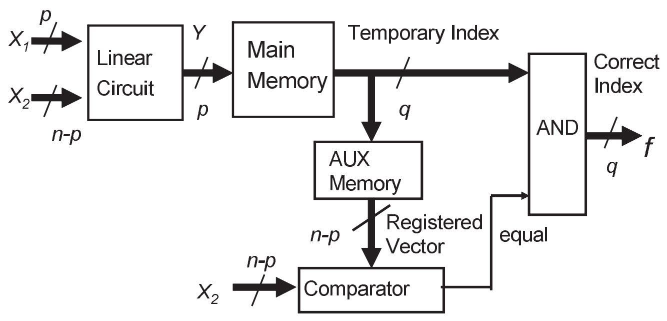


Fig. 6.1. Index Generation Unit.

The **main memory** has p inputs and $\lceil \log_2(k+1) \rceil$ outputs. The main memory produces correct indices only for registered vectors. However, it may produce incorrect indices for non-registered vectors, because the number of input variables is reduced by using *don't care* conditions. In an index generation function, if the input vector is non-registered, then it should produce 0 outputs. To check whether the main memory produces the correct index or not, we use the **AUX memory**. The AUX memory has $\lceil \log_2(k+1) \rceil$ inputs and $n - p$ outputs: It stores the X_2 part of the registered vectors for each index. The **comparator** checks if the X_2 part of the inputs are the same as the X_2 part of the registered vector. If they are the same, the main memory produces a correct index. Otherwise, the main memory produces a wrong index, and the input vector is non-registered. In this case, the **output AND gates** produce 0, showing that the input vector is non-registered. Note that the main memory produces the correct index only for the registered vectors. In this way, we can implement an incompletely specified index generation function instead of a completely specified one The size of the main memory is $p2^p$, and the size of the AUX memory is $(n-p)2^p$. Thus, the total memory size is

$$p2^p + (n-p)2^p = n2^p.$$

Example 6.1: Consider the registered vectors in Table 1.1. The number of variables is four, but only two variables x_1 and x_4 are necessary to distinguish these four registered vectors. Fig. 6.2 shows the IGU. In this case, the linear circuit produces $Y_1 = (x_1, x_4)$ from $X = (x_1, x_2, x_3, x_4)$. The main memory stores the indices for $X_1 = Y_1 = (x_1, x_4)$, and the AUX memory stores the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector. Consider two cases:

When the input vector is registered:
Suppose that a registered vector $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$ is applied to the IGU in Fig. 6.2. First, the linear circuit selects two variables, x_1 and x_4, and produces the value $X_1 = (x_1, x_4) = (1, 0)$. Second, the main memory produces the corresponding index $(0, 1, 1)$. Third, the AUX memory produces the values of $X_2 = (x_2, x_3) = (1, 0)$ corresponding registered vector $(1, 1, 0, 0)$. Fourth, the comparator confirms that the values of $X_2 = (x_2, x_3)$ of the input vector are equal

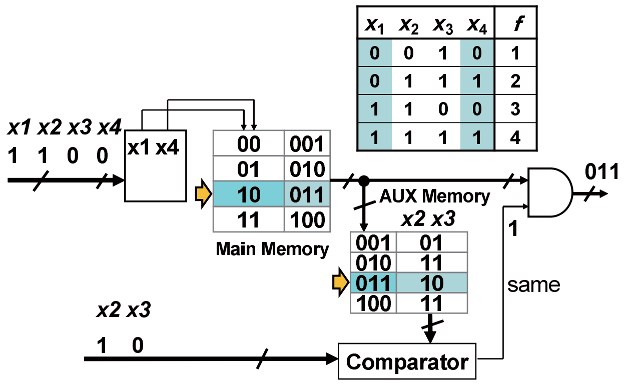


Fig. 6.2. When the input vector is registered.

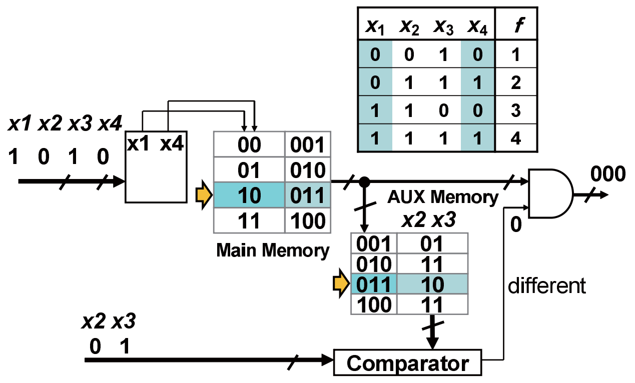


Fig. 6.3. When the input vector is not registered.

TABLE 7.1
2-OUT-OF-6 TO BINARY CONVERTER.

2-out-of-6 code						Index
x_6	x_5	x_4	x_3	x_2	x_1	
0	0	0	0	1	1	1
0	0	0	1	0	1	2
0	0	0	1	1	0	3
0	0	1	0	0	1	4
0	0	1	0	1	0	5
0	0	1	1	0	0	6
0	1	0	0	0	1	7
0	1	0	0	1	0	8
0	1	0	1	0	0	9
0	1	1	0	0	0	10
1	0	0	0	0	1	11
1	0	0	0	1	0	12
1	0	0	1	0	0	13
1	0	1	0	0	0	14
1	1	0	0	0	0	15

TABLE 7.2
TRANSFORMED 2-OUT-OF-6 TO BINARY CONVERTER.

Transformed code				Index
y_4	y_3	y_2	y_1	
0	0	0	1	1
0	0	1	1	2
0	0	1	0	3
0	1	1	0	4
0	1	1	1	5
0	1	0	1	6
1	1	0	0	7
1	1	0	1	8
1	1	1	1	9
1	0	1	0	10
1	0	0	0	11
1	0	0	1	12
1	0	1	1	13
1	1	1	0	14
0	1	0	0	15

to the output of the AUX memory. And, finally, the AND gate produces the index for the input vector.

When the input vector is not registered:

Suppose that a non-registered vector $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ is applied to the IGU in Fig. 6.3. Also in this case, the main memory produces the vector $(0, 1, 1)$, and the AUX memory produces the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector $(1, 1, 0, 0)$. However, in this case, the comparator shows that $X_2 = (x_2, x_3) = (0, 1)$ is different from the output $X_2 = (x_2, x_3)$ of the AUX memory. Thus, the AND gate produces 0, which shows that the input vector is not registered. ■

VII. DESIGN OF CODE CONVERTERS

In this part, we design m-out-of-n to binary converters.

Example 7.1: When $n = 6$ and $m = 2$, we have the function shown in Table 7.1. This is an index generation function with weight $k = \binom{n}{m} = \binom{6}{2} = 15$. When only the primitive variables are used, the number of inputs can be reduced up to five. However, when the inputs are transformed as:

$$\begin{aligned} y_4 &= x_6 \oplus x_5 \\ y_3 &= x_5 \oplus x_4 \\ y_2 &= x_4 \oplus x_3 \\ y_1 &= x_3 \oplus x_2 \end{aligned}$$

then, the code converter can be represented with only four variables: y_1, y_2, y_3, and y_4, as shown in Table 7.2. Since the function is represented with $p = \lceil \log_2(k+1) \rceil = 4$ variables, it is an optimum linear transformation. ■

In this example, the advantage of using a linear transformation is not so great. However, when n is large, a linear transformation can drastically reduce the memory size.

Example 7.2: Consider the case of $m = 2$ and $n = 20$. This is an index generation function with the weight $k = \binom{n}{m} = \binom{20}{2} = 190$. In the single-memory realization, the memory size is

$$\lceil \log_2(k+1) \rceil 2^n = 8 \times 2^{20},$$

which is too large. To obtain a decomposed realization, partition the inputs into $X_1 = (x_1, x_2, \ldots, x_{10})$ and $X_2 = (x_{11}, x_{12}, \ldots, x_{20})$. The column multiplicity with the decomposition with respect to (X_1, X_2) and (X_2, X_1) are the same and are both 57. Thus, it can be realized by the circuit shown in Fig. 7.1. In this realization, the total memory size is

$$2 \times 6 \times 2^{10} + 8 \times 2^{12} = 44 \times 2^{10}.$$

When we use an IGU to implement the function, the number of inputs to the main memory can be reduced to $p = \lceil \log(k+1) \rceil + 1 = 9$. In this case, the total memory size in the IGU is

$$n2^p = 20 \times 2^9 = 10 \times 2^{10}.$$

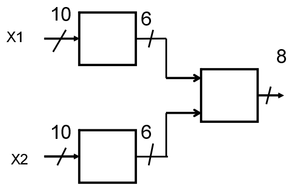


Fig. 7.1. Tree-type realization of 2-out-of-20 to binary converter.

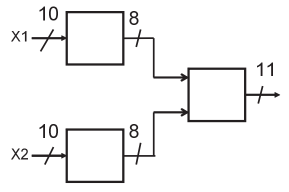


Fig. 7.2. Tree-type realization of 3-out-of-20 to binary converter.

■

Example 7.3: Consider the case of $m = 3$ and $n = 20$. This is an index generation function with weight $k = \binom{n}{m} = \binom{20}{3} = 1140$. In the single-memory realization, the memory size is

$$\lceil \log_2(k+1) \rceil 2^n = 11 \times 2^{20},$$

which is also too large. To realize a tree-type circuit, we partition the inputs into $X_1 = (x_1, x_2, \ldots, x_{10})$ and $X_2 = (x_{11}, x_{12}, \ldots, x_{20})$. The column multiplicity with the decomposition with respect to (X_1, X_2) and (X_2, X_1) are the same and are both 177. Thus, we have the circuit shown in Fig. 7.2. In this realization, the total memory size is

$$2 \times 8 \times 2^{10} + 11 \times 2^{16} = 720 \times 2^{10}.$$

When we use the IGU, the number of inputs to the main memory is reduced to $p = \lceil \log(k+1) \rceil = 11$. Thus, it is an optimum linear transformation. In this case, the total memory size in the IGU is

$$n 2^p = 20 \times 2^{11} = 40 \times 2^{10}.$$

■

VIII. Efficient Realizations of Index Generation Functions

We assume that the non-zero elements in the index generation function are uniformly distributed in the decomposition chart. In this case, we can estimate the fraction of registered vectors realized by the main memory.

Theorem 8.1: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the expected number of registered vectors of f that can be realized by the IGU is $2^p(1 - e^{-\xi})$, where $\xi = \frac{k}{2^p}$.

Corollary 8.1: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$

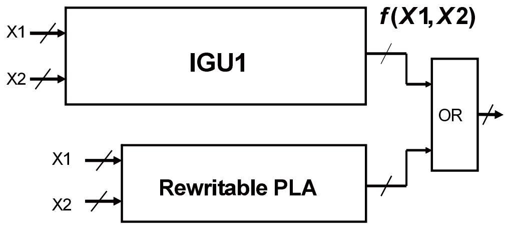


Fig. 8.1. Index generator implemented by the hybrid method.

and x_p. Then, the fraction of registered vectors of f that can be realized by the IGU is

$$\delta = \frac{1 - e^{-\xi}}{\xi},$$

where $\xi = \frac{k}{2^p}$.

Example 8.1: When $\xi = \frac{1}{4}$, we have $\delta \simeq 0.8848$, when $\xi = \frac{1}{2}$, we have $\delta \simeq 0.7869$, and when $\xi = 1$, we have $\delta \simeq 0.63212$.

We now show efficient methods to implement index generation functions using memories [16]. In an index generation function, the number of registered vectors k, is usually much smaller than 2^n, the total number of the input combinations.

Definition 8.1: The **hybrid method** is an implementation of an index generation function using the circuit consisting of IGU_1 as shown in Fig. 8.1. IGU_1 is used to realize most of the registered vectors, while a rewritable PLA is used to realize the remaining registered vectors. The OR gate in the output combines the indices to form a single output.

A **rewritable** PLA is necessary in most applications, since we often need to update the data. A rewritable PLA can be replaced by another circuit, such as an LUT cascade or a CAM.

In the hybrid method, when the main memory of IGU_1 has $p = \lceil \log_2(k+1) \rceil + 2$ inputs, we have $\xi = \frac{k}{2^p} = \frac{1}{4}$. From Example 8.1, about 88% of the registered vectors are implemented by IGU_1, and the remaining 12% of the registered vectors are implemented by the PLA.

Definition 8.2: The **super hybrid method** is an implementation of an index generation function using the circuit consisting of two IGUs as shown in Fig. 8.2. IGU_1 is used to realize most of the registered vectors, IGU_2 is used to realize the registered vectors not realized by IGU_1, and the rewritable PLA is used to realize registered vectors not realized by either IGU. The OR gate in the output combines the indices to form a single output.

The super hybrid method shown in Fig. 8.2 is more complicated than the hybrid method, but requires smaller memories. In the super hybrid method, when the main memory of the IGU_1 has $p_1 = \lceil \log_2(k+1) \rceil + 1$ inputs, and the main memory of the IGU_2 has $p_2 = \lceil \log_2(k+1) \rceil - 1$ inputs, from Example 8.1, we have $\xi_1 = \frac{k_1}{2^{p_1}} = \frac{1}{2}$, $\delta_1 = \frac{1-e^{\xi_1}}{\xi_1}$=0.7869; and $\xi_2 = \frac{k_2}{2^{p_2}} = \frac{1}{2}$, $\delta_2 = \frac{1-e^{\xi_2}}{\xi_2}$=0.7869. In this case, $k_1 = k$ and $k_2 = k_1(1 - \delta_1)$. Thus, about 80% of the registered vectors are

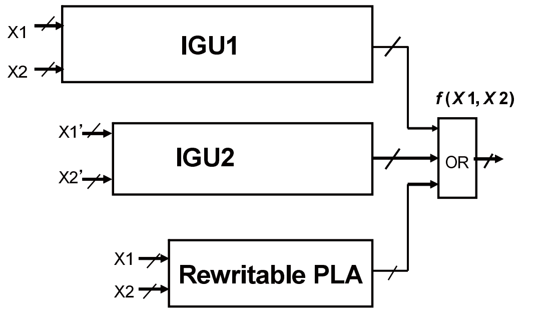


Fig. 8.2. Index generator implemented by the super hybrid method.

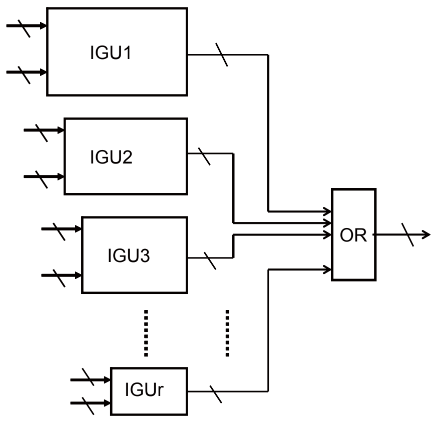


Fig. 8.3. Index generator implemented by the parallel sieve method.

implemented by IGU_1, about 16% of the registered vectors are implemented by IGU_2, and the remaining 4% of the registered vectors are implemented by the PLA.

By increasing the number of IGU's, we have the parallel sieve method, which is especially useful when the number of the registered vectors is very large [10].

Definition 8.3: The **parallel sieve method** is an implementation of an index generation function using the circuit consisting of multiple IGUs as shown in Fig. 8.3. IGU_{i+1} is used to realize a part of the registered vectors not realized by IGU_1, $IGU_2, \ldots,$ or IGU_i. The OR gate in the output combines the indices to form a single output. In the **standard parallel sieve method**, the number of inputs to the main memory is chosen as $p_i = \lceil \log_2(k_i + 1) \rceil$, where k_i denotes the number of registered vectors to be implemented by IGU_i, $IGU_{i+1}, \ldots,$ and IGU_r.

IX. Extension to Multiple-Valued Case

Index generation functions can be extended into multiple-valued input functions as follows:

Definition 9.1: A **multi-valued input index generation function** f is a mapping $\{0, 1, \ldots, r-1\}^n \rightarrow \{0, 1, \ldots, k-1\}$.

Experimental results [20] suggest the following:

Conjecture 9.1: Consider a set of uniformly distributed incompletely specified r-valued input n-variable index generation functions with weight k. If

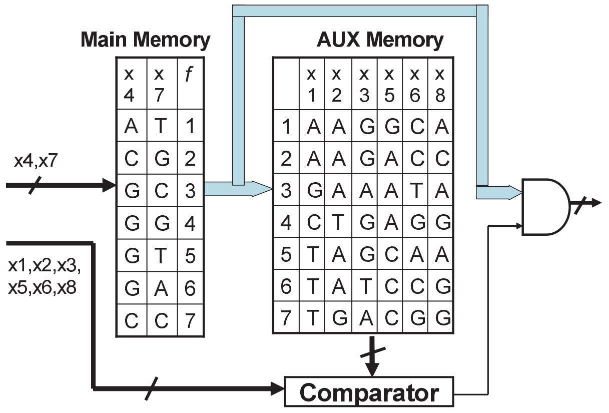
$$p \geq 2 \log_r k - \log_r 4.158,$$

then more than 95% of the functions can be represented with p variables.

Note that there exist functions that require more variables. However, the fraction of such functions approaches to 0.0 as n increases.

TABLE 9.1
4-valued input index generation function.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	f
A	A	G	A	G	C	T	A	1
A	A	G	C	A	C	G	C	2
G	A	A	G	A	T	C	A	3
C	T	G	G	A	G	G	G	4
T	A	G	G	G	A	T	A	5
T	A	T	G	C	C	A	G	6
T	G	A	C	C	G	C	G	7



x4	x7	f
A	T	1
C	G	2
G	C	3
G	G	4
G	T	5
G	A	6
C	C	7

	x1	x2	x3	x5	x6	x8
1	A	A	G	G	C	A
2	A	A	G	A	C	C
3	G	A	A	A	T	A
4	C	T	G	A	G	G
5	T	A	G	C	A	A
6	T	A	T	C	C	G
7	T	G	A	C	G	G

Fig. 9.1. Index generation unit for DNA matching.

Example 9.1: Deoxyribonucleic acid (DNA) contains the genetic instructions used in the development and functioning of all known living organisms. The four bases found in DNA are adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T). To represent DNA, we use 4-valued logic. Consider the circuit to detect DNA patterns shown in Table 9.1. Since each pattern consists of 8 characters, a single-memory realization requires a memory with $2 \times 8 = 16$ inputs. Since, it has three outputs, the memory size is $2^{16} \times 3 = 192 \times 2^{10}$ bits. However, these patterns can be distinguished by using only two characters: x_4 and x_7. Fig. 9.1 shows the circuit to detect the DNA patterns. In Fig. 9.1, the total amount of memory is only $4^2 \times 3 + 8 \times 6 \times 2 = 144$ bits. ■

X. Related Work

The use of linear transformations in the logic design was first considered by Nechiporuk in 1958 [9]. Later, Lechner [7] presented an extensive survey of the methods, and Varma and Trachtenberg [24] showed the usefulness of the linear transformation for logic synthesis benchmark functions. In these design methods, the cost measure of the circuits was the gate count. And, autocorrelation was used to estimate the cost of the function. Recently, linear transformation is considered in [6] to reduce circuit complexity. In these works, the methods seem to be effective for totally or partially symmetric functions, including adders. However, for other functions, linearization are not so effective. Reductions of the sizes of BDDs using linear transformations were considered in [8], [1], [5]. In these cases, the methods are useful for error-correcting circuits (C499, C1355, C1908) in addition to totally and partially symmetric functions including adders (C7552). In [15], the author presented a method to reduce the number of variables for incompletely specified function by linear transformation. In this case, the circuit is implemented by memories, and the cost measure is the memory size or the number of the variables for the main memory. Minimization of variables for multiple-valued index generation functions are also considered in [22].

XI. Conclusions

In this paper, we introduced index generation functions, which have wide applications in pattern matching circuits for the Internet. We also presented a method to implement an index generation function using an IGU. Reduction of the number of variables using linear transformation is shown. To implement functions with many vectors, we developed the hybrid method, the super-hybrid method and the parallel sieve method. An extension to multiple-valued input case is also shown. This survey is based on [21].

Acknowledgments

This research is partly supported by the MEXT Regional Innovation Cluster Program (Global Type, 2nd Stage). The author thanks Prof. Jon T. Butler, Dr. Hiroki Nakahara, and Mr. M. Matsuura for discussion. Prof. R. S. Stankovic provided us [9] and [23].

References

[1] W. Gunther and R. Drechsler, "Efficient minimization and manipulation of linearly transformed binary decision diagrams," *IEEE Transactions on Computers*, vol.52. No. 9, pp.1196-1209, September, 2003.

[2] C. Halatsis and N. Gaitanis, "Irredundant normal forms and minimal dependence sets of a Boolean function," *IEEE Transactions on Computers*, Vol. C-27, No. 11, pp. 1064-1068, November, 1978.

[3] Y. Iguchi, T. Sasao, and M. Matsuura, "Design methods of radix converters using arithmetic decompositions," *IEICE Trans. on Information and Systems*, Vol. E90-D, No. 6, June 2007, pp. 905-914.

[4] Y. Kambayashi, "Logic design of programmable logic arrays," *IEEE Trans. on Computers*, Vol. C-28, No. 9, pp. 609-617, September 1979.

[5] M. G. Karpovsky, R. S. Stankovic, and J. T. Astola, "Reduction of sizes of decision diagrams by autocorrelation functions," *IEEE Transactions on Computers*, Vol. 52, No.5, pp. 592-606, May, 2003.

[6] O. Keren and I. Levin, "Linearization of multi-output logic functions by ordering of the autocorrelation values," *FACTA UNIVERSITATIS (NIS)*, Vol. 20, no. 3, December 2007, pp. 479-498.

[7] R. J. Lechner, "Harmonic analysis of switching functions," in A. Mukhopadhyay (ed.), *Recent Developments in Switching Theory*, Academic Press, New York, 1971.

[8] C. Meinel, F. Somenzi, and T. Theobald, "Linear sifting of decision diagrams and its application in synthesis," *IEEE Trans. CAD*, vol. 19, no. 5, pp. 521-533, 2000.

[9] E. I. Nechiporuk, "On the synthesis of networks using linear transformations of variables," *Dokl. AN SSSR*, vol. 123, no. 4, pp. 610-612, Dec. 1958.

[10] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "A parallel sieve method for a virus scanning engine," *12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece (*DSD-2009*), Aug. 2009, pp.809-816.

[11] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[12] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," *30th International Symposium on Multiple-Valued Logic*, pp. 91-97, Portland, Oregon, U.S.A., May 23-25, 2000.

[13] T. Sasao, "Radix converters: Complexity and implementation by LUT cascades," *ISMVL-2005*, pp. 256-263.

[14] T. Sasao, "Design methods for multiple-valued input address generators,"(invited paper) *International Symposium on Multiple-Valued Logic* (ISMVL-2006), Singapore, May 2006.

[15] T. Sasao, "A Design method of address generators using hash memories," *IWLS-2006*, Vail, Colorado, U.S.A, June 7-9, 2006, pp.102-109.

[16] T. Sasao and M. Matsuura, "An implementation of an address generator using hash memories," *DSD-2007*, Aug. 27 - 31, 2007, Lubeck, Germany, pp.69-76.

[17] T. Sasao, "On the number of variables to represent sparse logic functions," *ICCAD-2008*, San Jose, California, USA, Nov.10-13, 2008, pp. 45-51.

[18] T. Sasao, T. Nakamura, and M. Matsuura, "Representation of incompletely specified index generation functions using minimal number of compound variables," *DSD-2009*, Aug. 2009, pp.765-772.

[19] T. Sasao, M. Matsuura, and H. Nakahara, "A Realization of index generation functions using modules of uniform sizes," *International Workshop on Logic and Synthesis* (IWLS-2010), Irvine, California, June 11-13, 2010, pp.201-208.

[20] T. Sasao, "On the numbers of variables to represent multi-valued incompletely specified functions," *13th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Lille, France *DSD-2010*, Sept. 2010, pp.420-423.

[21] T. Sasao, *Memory Based Logic Synthesis*, Springer, 2011 (to be published).

[22] D. A. Simovici, D. Pletea, and R. Vetro, "Information-theoretical mining of determining sets for partially defined functions," *ISMVL-2010*, May 2010, pp.294-299.

[23] R. S. Stankovic and J. Astola (eds.) E.I. Nechiporuk, "Network synthesis by using linear transformation of variables," in *Reprints from the Early Days of Information Sciences*, Tampere International Center for Signal Processing, Tampere 2007.

[24] D. Varma and E. Trachtenberg, "Design automation tools for efficient implementation of logic functions by decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.8, No.8, pp.901-916, 1989.