J. of Mult.-Valued Logic & Soft Computing, Vol. 0, pp. 1–29
Reprints available directly from the publisher
Photocopying permitted by license only

Index Generation Functions: Tutorial

TSUTOMU SASAO

Department of Computer Science, Meiji University, Kawasaki 214-8571, Japan
E-mail: sasao@cs.meiji.ac.jp

Received: September 28, 2011. Accepted: May 7, 2012.

Given a set of k distinct binary vectors of n bits, for each vector assign a unique integer from 1 to k. An incompletely specified index generation function produces an index for a given vector. This tutorial first introduces index generation functions, which are useful for pattern matching in communication circuits. Then, it shows a method to represent a given index generation function using fewer variables. A linear transformation is used to reduce the number of variables. An extension to the multiple-valued case is also presented.

1 INDEX GENERATION FUNCTION

This tutorial shows 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 [10] are omitted.

Definition 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. Let*

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

TABLE 1
Registered vector table.

the **weight** *of the index generation function be k. An index generation function represents a mapping:* $B^n \rightarrow \{0, 1, 2, \ldots, k\}$.

Example 1. *Table 1 shows a registered vector table. It shows an index generation function with weight* $k = 4$. ■

The rest of the paper is organized as follows: Section 2 discusses applications of index generation functions. Section 3 shows that an incompletely specified index generation functions can often be represented with fewer variables than the original specification. Section 4 shows an algorithm to minimize the number of variables to represent incompletely specified index generation functions. Section 5 shows that most uniformly distributed index generation functions with weight k can be represented with $2\lceil \log_2(k+1) \rceil - 3$ variables. Section 6 shows a method to reduce the number of variables by using a linear transformation. Section 7 shows the effect of linear transformations in the reduction of the number of variables. Section 8 explains the operation of an index generation unit. Section 9 shows design examples of m-out-of-n code converters. Section 10 extends the theory to multiple-valued input functions. Section 11 surveys related works on linear transformations. Section 12 concludes the paper.

2 APPLICATIONS

An **index generator** is a circuit that realizes an index generation function. 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, code converters etc. [12].

2.1 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 to a memory

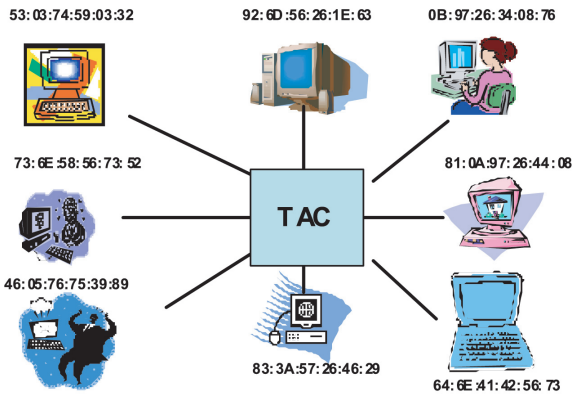


FIGURE 1
Terminal access controller.

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 addresses. Note that the address table must be updated frequently.

2.2 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 Figure 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. *Figure 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 the Web outside the local area network, E-mail, FTP, 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 remaining ones are 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.*

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
81:0A:97:26:44:08	5
46:06:76:75:39:89	6
83:3A:57:26:46:29	7
64:6E:41:42:56:73	8

48-bit

⇒

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
5	0	1	0	0
6	0	1	0	0
7	0	1	0	0
8	0	1	0	0

4-bit

FIGURE 2
Index generator for terminal access controller.

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 the 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 be programmable.

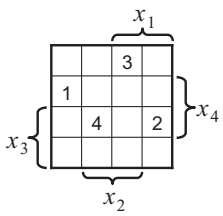
3 INCOMPLETELY SPECIFIED INDEX GENERATION FUNCTIONS

An application of index generation function often involves incompletely specified functions. In this section, we introduce some methods to represent a given incompletely specified function with fewer variables [13, 14, 16].

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

$\hat{f}_1(x_1, x_2, x_3, x_4)$

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



(a) Registered Vector Table (b) Decomposition chart

FIGURE 3
Reduction of variables to represent an incompletely specified index generation function.

index generation function with weight *k if*

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

where d denotes don't care or undefined.

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

Example 3. *Consider the registered vector table shown in Figure 3(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 chart for* $\hat{f}_1(X)$ *is shown in Figure 3(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 chart, 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 Figure 3(a), values for (x_1, x_2) *are distinct, and the index can be specified by using only these two variables:*

$$f_1 = 1 \cdot \bar{x}_1\bar{x}_2 \vee 2 \cdot x_1\bar{x}_2 \vee 3 \cdot x_1x_2 \vee 4 \cdot \bar{x}_1x_2.$$

■

Example 4. *Consider the registered vector table in Figure 4, and the decomposition chart for an incompletely specified index generation function* $\hat{f}_2$. *Consider the number of variables to represent the function. In the*

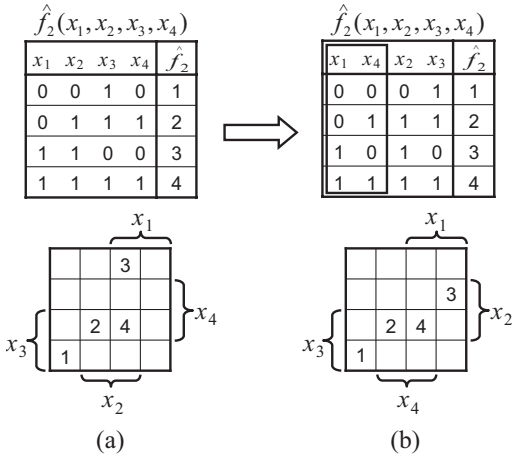


FIGURE 4
Reduction of variables to represent an input incompletely specified index generation function.

decomposition chart in Figure 4(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 Figure 4(b). In this case, each column has at most one specified element. Note that, in the registered vector table in Figure 4(b), values of the vectors* (x_1, x_4) *are all different. Thus, the function* $\hat{f}_2$ *can be represented by using only* $\{x_1, x_4\}$. ■

4 ALGORITHM TO MINIMIZE THE NUMBER OF VARIABLES

This section describes an algorithm to represent an incompletely specified index generation function $f : D \rightarrow \{1, 2, \ldots, k\}$, where $D \subset B^n$, using the minimum number of variables. Minimization methods of input variables for single-output incompletely specified functions are considered in [3,4,11,15]. To show the idea of the method, we use the following:

Example 5. *Let us minimize the number of variables to represent the index generation function shown in Figure 5.*

1. *Let the four vectors be* $\vec{a}_1 = (1, 0, 0, 1)$, $\vec{a}_2 = (1, 1, 1, 1)$, $\vec{a}_3 = (0, 1, 0, 1)$, *and* $\vec{a}_4 = (1, 1, 0, 0)$.

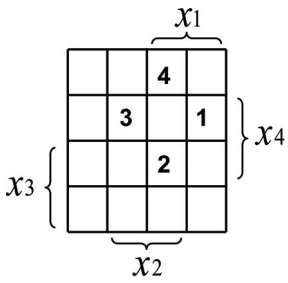


FIGURE 5
Index generation function with four-variables.

2. *To distinguish $\vec{a}_1$ and $\vec{a}_2$, either x_2 or x_3 is necessary. Thus, we have the condition $x_2 \vee x_3 = 1$, where $x_i = 1$ denotes that x_i must appear in the expression. Thus, $x_2 \vee x_3 = 1$ denotes either x_2 or x_3 must appear in the expression. In the same way, to distinguish $\vec{a}_1$ and $\vec{a}_3$, we have the condition $x_1 \vee x_2 = 1$; to distinguish $\vec{a}_1$ and $\vec{a}_4$, we have the condition $x_2 \vee x_4 = 1$; to distinguish $\vec{a}_2$ and $\vec{a}_3$, we have the condition $x_1 \vee x_3 = 1$; to distinguish $\vec{a}_2$ and $\vec{a}_4$, we have the condition $x_3 \vee x_4 = 1$; and to distinguish $\vec{a}_3$ and $\vec{a}_4$, we have the condition $x_1 \vee x_4 = 1$.*
3. *To distinguish all the vectors, all the conditions must hold at the same time. This is expressed by the condition $R = 1$, where*

$$R = (x_2 \vee x_3)(x_1 \vee x_2)(x_2 \vee x_4)(x_1 \vee x_3)(x_3 \vee x_4)(x_1 \vee x_4).$$

4. *By the distributive law, and the absorption law, we have*

$$R = x_1x_2x_4 \vee x_1x_2x_3 \vee x_2x_3x_4 \vee x_1x_3x_4.$$

5. *Since each product consists of three literals, each corresponds a minimum solution. Thus, f can be represented with three variables. Since no variable appears in all products, no variable is essential.*
6. *The expression for the original function is*

$$f = 1 \cdot x_1\bar{x}_2\bar{x}_3x_4 \vee 2 \cdot x_1x_2x_3x_4 \vee 3 \cdot \bar{x}_1x_2\bar{x}_3x_4 \vee 4 \cdot x_1x_2\bar{x}_3\bar{x}_4.$$

To obtain the expression corresponding to the first product $x_1x_2x_4$ in Step 4, remove the literals for x_3:

$$f = 1 \cdot x_1\bar{x}_2x_4 \vee 2 \cdot x_1x_2x_4 \vee 3 \cdot \bar{x}_1x_2x_4 \vee 4 \cdot x_1x_2\bar{x}_4.$$

■

Algorithm 1 (Algebraic Method)

1. Let A be the set of vectors $\vec{a}_i$, such that $f(\vec{a}_i) = i$, where $i = 1, 2, \ldots, k$
2. For each pair of vectors $\vec{a}_i = (a_1, a_2, \ldots, a_n) \in A$ and $\vec{b}_j = (b_1, b_2, \ldots, b_n) \in A$, associate a product defined by $s(i, j) = \bigvee_{r=1}^{n} y_r$, where $y_r = 0$ if $a_r = b_r$ and $y_r = x_r$ if $a_r \neq b_r$, where $r = 1, 2, \ldots, n$. Note that there are $k(k-1)/2$ pairs.
3. Define a covering function $R = \bigwedge_{i<j} s(i, j)$.
4. Represent R by the a minimum SOP.
5. The product with the fewest literals corresponds to the minimum solution.
6. Derive an expression with the minimum number of variables.

In Algorithm 1, Steps 2, 3 and 4 compute a minimum covering. By first detecting the essential variables, we can reduce the computational effort to derive the covering function. The next example illustrates this.

Example 6. *The 7-segment display shown in Figure 2 displays a decimal number by using 7 segments: a, b, c, d, e, f, and g.*

Table 2 shows the correspondence between segment data and the binary number. Consider a logic circuit that converts 7 segment data into the corresponding **Binary Coded Decimal** *(BCD) representation of a digit. The straightforward circuit requires 7 inputs. However, only five inputs are necessary to distinguish the decimal numbers. This means that only 5 segments are needed to distinguish between the 10 digits.*

7-segment							**BCD code**			
a	***b***	***c***	***d***	***e***	***f***	***g***	**8**	**4**	**2**	**1**
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
1	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	0	1	1	1	0	0	1
1	1	1	1	1	1	0	1	0	1	0

TABLE 2
7-segment to BCD converter.

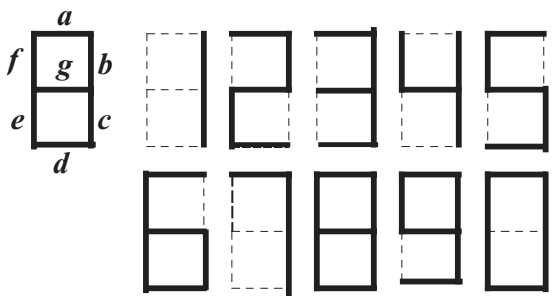


FIGURE 6
7-segment display.

1. *Let the vectors be* $\vec{a}_1 = (0, 1, 1, 0, 0, 0, 0)$, $\vec{a}_2 = (1, 1, 0, 1, 1, 0, 1)$, $\vec{a}_3 = (1, 1, 1, 1, 0, 0, 1)$, $\vec{a}_4 = (0, 1, 1, 0, 0, 1, 1)$, $\vec{a}_5 = (1, 0, 1, 1, 0, 1, 1)$, $\vec{a}_6 = (1, 0, 1, 1, 1, 1, 1)$, $\vec{a}_7 = (1, 1, 1, 0, 0, 0, 0)$, $\vec{a}_8 = (1, 1, 1, 1, 1, 1, 1)$, $\vec{a}_9 = (1, 1, 1, 1, 0, 1, 1)$, *and* $\vec{a}_{10} = (1, 1, 1, 1, 1, 1, 0)$.
2. *First, find the essential variables. From* $\vec{a}_1$ *and* $\vec{a}_7$*, we can see that* a *is essential. From* $\vec{a}_6$ *and* $\vec{a}_8$*, we can see that* b *is essential. From* $\vec{a}_8$ *and* $\vec{a}_9$*, we can see that* e *is essential. From* $\vec{a}_3$ *and* $\vec{a}_9$*, we can see that* f *is essential. From* $\vec{a}_8$ *and* $\vec{a}_{10}$*, we can see that* g *is essential.*
3. *Next, we derive* R*. Since* a*,* b*,* e*,* f*, and* g *are essential, we can ignore the pairs, where the essential variables are inconsistent. For example, from the pair* $(\vec{a}_1, \vec{a}_2)$*, we have the sum* $a \vee c \vee d \vee e \vee g$*. Note that, in this case, two vectors are inconsistent with the essential variable* a*. Since the essential variable* a *is always included in the solution, we know that* $a = 1$*. Thus, the sum is equal to one. Thus, we need not generate it. Note that there are* $\binom{10}{2} = 45$ *pairs. We can verify that all the pairs contain an essential variable. Thus, we can conclude that* $R = abefg$*.*
4. *Since the product has five literals, it corresponds to the minimum solution. Thus, the BCD numbers can be represented by five variables.*

Thus, we can eliminate segments c *and* d*, and still determine which digit is being represented. Figure 7 is Figure 6 with segments* c *and* d *eliminated. Because all 10 digits can still be identified, this illustrates why* c *and* d *can be eliminated.* ■

5 PROPERTY OF UNIFORMLY DISTRIBUTED FUNCTIONS

As shown in the previous section, incompletely specified index generation functions often can be represented with fewer variables.

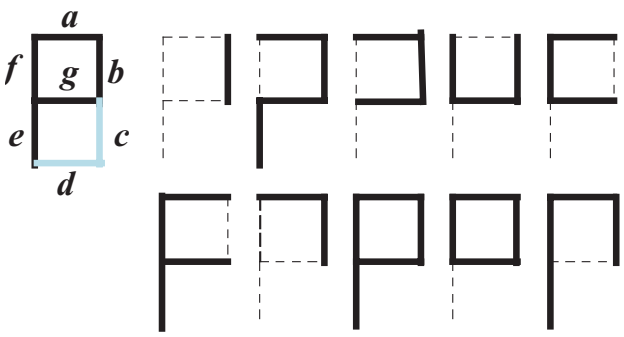


FIGURE 7
5-segment display.

Example 6 shows a 7 variable index generation function with weight 10. In this case, only five variables are sufficient to represent the function. Suppose that a 7-variable index generation function with weight 10 is given. How many variables, on the average, are necessary to represent the function? There exist $\prod_{i=0}^{9}(128-i) = 8.23 \times 10^{23}$ different 7-variable index generation functions with weight 10. It is hard to obtain the average by an exhaustive method. So, we randomly generated 1000 functions, and obtained statistical results. Figure 8 shows the numbers of functions that require 4, 5, and 6 variables. It shows that out of 1000 functions, 978 required 4 or 5 variables, while only 22 required 6 variables. Thus, Example 6 may be typical among 7-variable index generation functions with weight 10.

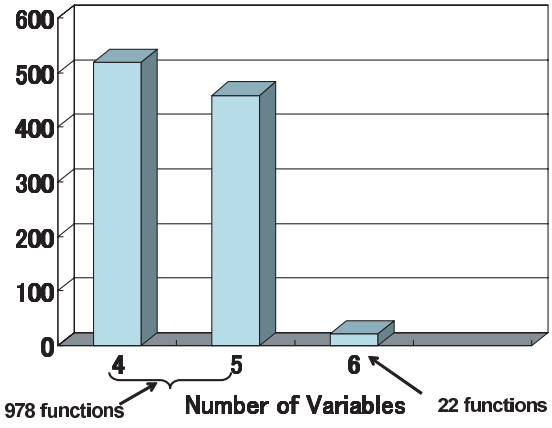


FIGURE 8
Number of variables to represent 7 variable index generation functions with weight 10.

Then, how many variables are necessary to represent an index generation function with weight k? The following shows a lower bound.

Theorem 1. *To represent any incompletely specified index generation function f with weight k, at least $q = \lceil \log_2 k \rceil$ variables are necessary.*

Proof. The number of different vectors specified with $q-1$ variables is at most $2^{q-1} < k$. Thus, at least q variables are necessary to represent an index generation function with weight k. □

To show the upper bound, we need to define a class of functions.

Definition 3. *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 shows average numbers of variables to represent incompletely specified index generation functions for different n and different weight k. This was obtained by minimizing 1000 randomly generated functions for each parameter. The variance of the distribution is quite small. For example, in the case of $n = 20$ and $k = 127$, the numbers of functions that require 9, 10, and 11 variables are shown in Figure 9. It shows that out of 1000 functions, only two required 9 variables, 997 required 10 variables, and only one required 11 variables. Thus, most functions require 10 variables.

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

TABLE 3
Average numbers of variables to represent incompletely specified index generation function.

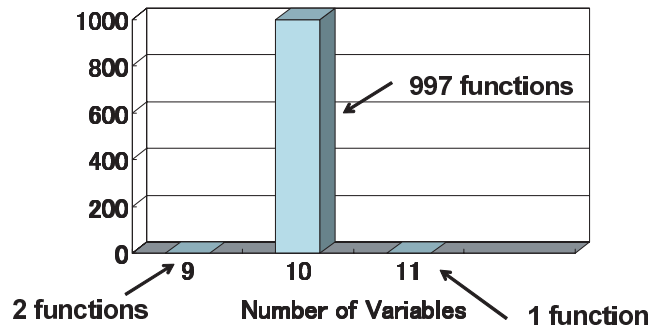


FIGURE 9
Number of variables to represent 20 variable index generation functions with weight 127.

From the experimental results and mathematical observation [18], we have the following:

Conjecture 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.

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

$$y = c_1 x_1 \oplus c_2 x_2 \oplus \cdots \oplus c_n x_n,$$

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*

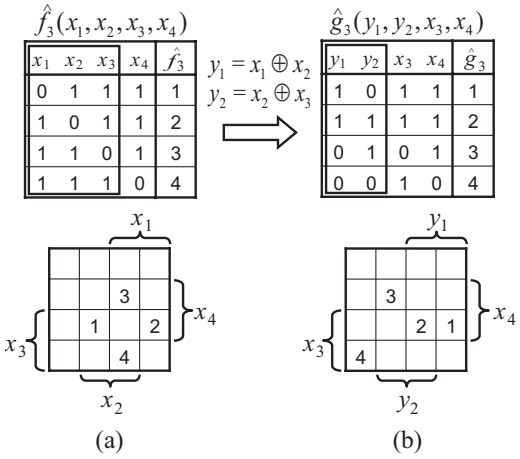


FIGURE 10
Incompletely specified index generation function represented by compound variables.

degree 2 is a **bi-compound variable**, *and a variable with compound degree 3 is a* **tri-compound variable**.

Example 7. *Consider the incompletely specified index generation function* $\hat{f}_3$ *shown in Figure 10. Let us consider the number of variables to represent this function. In Figure 10(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 chart 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 Figure 10(b). Note that, in the decomposition chart shown in Figure 10(b), each column has at most one specified element. Thus, a function* $\hat{g}_3$ *can be represented by using only two variables* $\{y_1, y_2\}$. ■

In the rest of the paper, both a primitive variable x_i and a compound variables y_j are treated as input variables.

As shown Example 7, by a linear transformation, we can often reduce the number of variables to represent the function. Why does this linear transformation reduce the number of variables? Figure 11 shows the decision tree for the original function in Figure 10. On the other hand, Figure 12 shows the decision tree for the transformed function in Figure 10. To distinguish 4 vectors, the tree in Figure 11 requires three variables, while the tree in

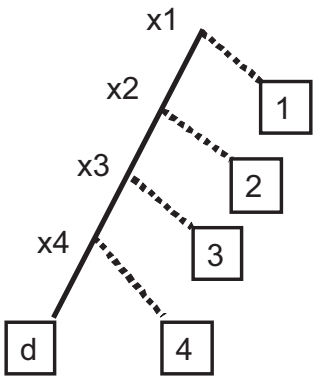


FIGURE 11
Unbalanced Decision Tree.

Figure 12 requires only two variables. In other words, if we can make a more balanced decision tree by a linear transformation, we may be able to represent the functions with fewer variables.

Definition 5. *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 $p = \lceil \log_2 k \rceil$, 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 Algorithm 1. In principle, the minimization 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

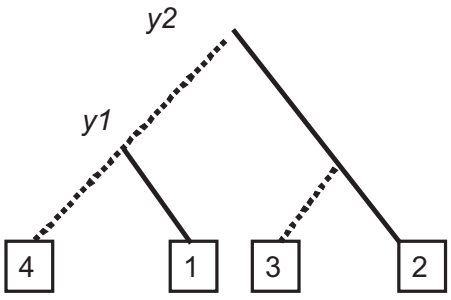


FIGURE 12
Balanced Decision Tree.

$x_1, x_2, \ldots, x_n$, 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 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 [21], we developed 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 6. *In the registered vector table, let $\nu(x_i, 0)$ be the number of vectors with $x_i = 0$, and let $\nu(x_i, 1)$ be the number of vectors with $x_i = 1$. The* **imbalance measure** *of the function with respect to x_i is defined as*

$$\omega(x_i) = \nu(x_i, 0)^2 + \nu(x_i, 1)^2.$$

In the variable x_i, when the numbers of occurrences of 0's and 1's are the same, $\omega(x_i)$ takes its minimum. The larger the difference of the occurrences of 0's and 1's, the larger the imbalance measure. Let k be the number of registered vectors. Then, $\nu(x_i, 0) + \nu(x_i, 1) = k$.

Example 8. *In Figure 10(a), since, for all x_i, $\nu(x_i, 0) = 1$ and $\nu(x_i, 1) = 3$, we have*

$$\omega(x_i) = \nu(x_i, 0)^2 + \nu(x_i, 1)^2 = 1^2 + 3^2 = 10.$$

In Figure 10(b), since $\nu(x_i, 0) = 2$ and $\nu(x_i, 1) = 2$, we have

$$\omega(x_i) = \nu(x_i, 0)^2 + \nu(x_i, 1)^2 = 2^2 + 2^2 = 8.$$

In other words, the linear transformation in Example 7 reduces the imbalance measure, and improves the balance of the decision tree. ∎

	7-Segment							After Linear Transformation			
	a	***b***	***c***	***d***	***e***	***f***	***g***	y_1	y_2	y_3	y_4
	0	1	1	0	0	0	0	0	1	0	0
	1	1	0	1	1	0	1	1	0	1	0
	1	1	1	1	0	0	1	0	0	1	1
	0	1	1	0	0	1	1	0	1	1	1
	1	0	1	1	0	1	1	0	1	0	1
	1	0	1	1	1	1	1	1	1	0	0
	1	1	1	0	0	0	0	0	1	1	0
	1	1	1	1	1	1	1	1	0	0	0
	1	1	1	1	0	1	1	0	0	0	1
	1	1	1	1	1	1	0	1	0	0	1
ω	68	68	82	58	52	52	58	52	50	52	50

TABLE 4
7-Segment to BCD Converter After Linear Transformation.

Variables with a smaller imbalance measure 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 $\lceil \log_2 k \rceil$.

When selecting compound variables, a variable with a small imbalance measure tends to produce more balanced tree, and tends to reduce the number of variables needed to represent the function.

Example 9. *For the function in Table 2, reduce the number of variables to represent the function by applying a linear transformation. By applying the linear transformation*

$$\begin{aligned} y_1 &= e, \\ y_2 &= b \oplus d, \\ y_3 &= a \oplus f, \\ y_4 &= e \oplus g, \end{aligned}$$

we have Table 4, where the last row shows the imbalance measure. The imbalance measures are reduced by the linear transformation. Also note that, after the linear transformation, the four-bit vectors (y_1, y_2, y_3, y_4) *are all distinct. This means that these four variables distinguish all the vectors. That is, these four variables can represent the original index generation function.* ■

7 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, more variables are necessary to represent the function.

7.1 Constant-Weight Code to Index Converter

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

Definition 8. *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 10. *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. When only the primitive variables are used, at least 14 variables are necessary to represent the*

1-out-of-15 code															
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	**Index**
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

TABLE 5
1-out-of-15 to binary converter.

Transformed code				
y_4	y_3	y_2	y_1	**Index**
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

TABLE 6
Transformed 1-out-of-15 to binary converter.

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 6. 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, the probability of 1's is $1/15$, while in the transformed registered vector table shown in Table 6, the probability of 1's is $8/15$. The linear transformation makes the height of the decision tree small, and reduces the number of variable to represent the function. Since the function is represented with $p = \lceil \log_2 k \rceil = 4$ variables, it is an optimum linear transformation. ■

7.2 Random Index Generation Functions

For $n = 24$ and $k = 1023$, we generated 1000 random index generation functions for different number of 1's in the registered vector table. Figure 13 shows the results. The number of variables after linear transformations is plotted along the vertical axis. The difference of occurrences of 0's and 1's is plotted along the horizontal axis:

$$s = \frac{|\nu(x_i, 0) - \nu(x_i, 1)|}{64}.$$

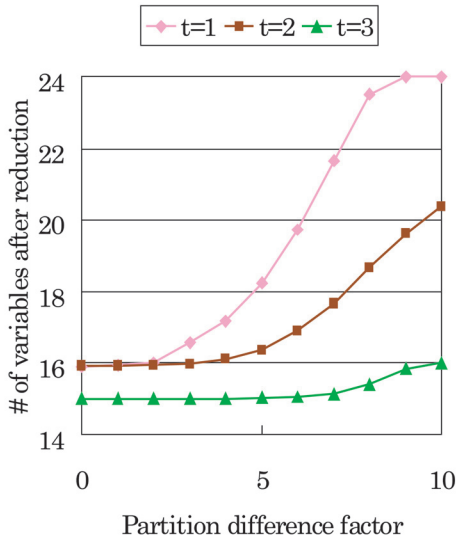


FIGURE 13
Number of Variable to Represent Random Index Generation Functions with Weight = 1023 and $n = 24$

Also, t denotes the maximum compound degree of the variables. For $s = 0$, the necessary number of variables to represent a function is reduced to 16 from 24 when $t = 1$ and $t = 2$. However, for $s = 10$, the necessary number of variables to represent a function is 24 when $t = 1$, while it is reduced to 20 when $t = 2$, and it is further reduced to 16 when $t = 3$. Thus, linear transformations with large t are especially effective when the imbalance measure is large.

8 INDEX GENERATION UNIT

Figure 14 shows an **index generation unit** (IGU) [13–15]. The **linear circuit** has n inputs and p outputs, where $p < n$. It produces functions:

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

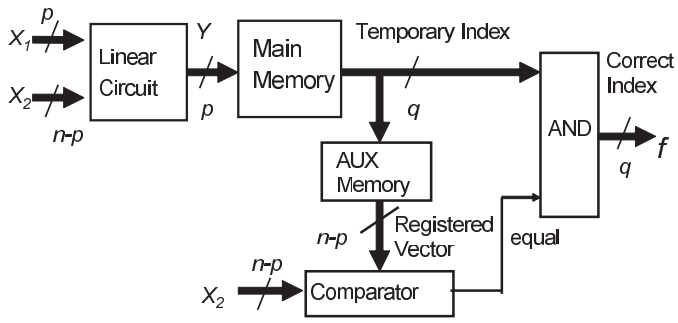


FIGURE 14
Index Generation Unit.

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 $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$.

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 $q = \lceil \log_2(k+1) \rceil$ 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 is 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 an incorrect 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 $(n - p)2^q$. Thus, the total memory size is

$$q2^p + (n - p)2^q.$$

Example 11. *Consider the registered vectors in Table 1. The number of variables is four, but only two variables x_1 and x_4 are necessary to distinguish these four registered vectors. Figure 15 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*

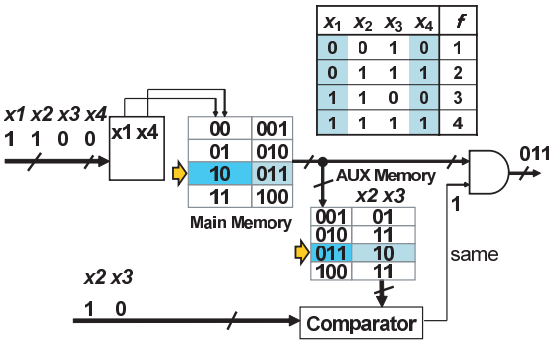


FIGURE 15
When the input vector is registered.

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 Figure 15. 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 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 Figure 16. Also in this case, the main memory produces the*

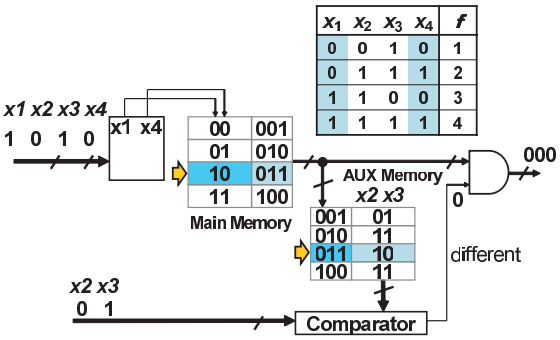


FIGURE 16
When the input vector is not registered.

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.* ■

9 DESIGN OF CONSTANT-WEIGHT CODE TO INDEX CONVERTERS

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

Example 12. *When* $n = 6$ *and* $m = 2$*, we have the function shown in Table 7. 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 to five. However, when the inputs are transformed as:*

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

2-out-of-6 code						
x_6	x_5	x_4	x_3	x_2	x_1	**Index**
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-out-of-6 to binary converter.

Transformed code				
y_4	y_3	y_2	y_1	**Index**
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

TABLE 8
Transformed 2-out-of-6 to binary converter.

then, the code converter can be represented with only four variables: y_1, y_2, y_3, *and* y_4, *as shown in Table 8. Since the function is represented with* $p = \lceil \log_2 k \rceil = 4$ *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 13. *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 Figure 17. In this realization, the total memory size is*

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

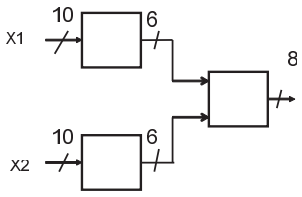


FIGURE 17
Tree-type realization of 2-out-of-20 to binary converter.

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

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

■

Example 14. *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 Figure 18. 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*

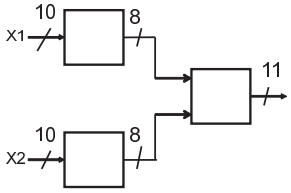


FIGURE 18
Tree-type realization of 3-out-of-20 to binary converter.

this case, the total memory size in the IGU is

$$n2^p = 20 \times 2^{11} = 40 \times 2^{10}.$$

■

Recently, an efficient method to realize constant-weight code to index converters was developed [1]. However, this method is only applicable to this type of functions.

10 EXTENSION TO MULTIPLE-VALUED CASE

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

Definition 9. *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 [17] and an observation [23] suggest the following:

Conjecture 2. *Consider a set of uniformly distributed incompletely specified* r*-valued input* n*-variable index generation functions with weight* k*, where* $r^2 \leq k \leq r^{n-2}$ *and* $n \geq 10$*. If*

$$p \geq \lceil 2 \log_r k - \log_r 5.485 \rceil,$$

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.

Example 15. *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. 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*. Figure 19 shows the circuit to detect the DNA patterns. In Figure 19, the total amount of memory is only*

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

TABLE 9
4-valued input index generation function.

$4^2 \times 3 + 8 \times 6 \times 2 = 144$ *bits, or* $1/1365$ *of the memory used in the naive method.* ■

11 RELATED WORK

The use of linear transformations in logic design was first considered by Nechiporuk in 1958 [9]. Later, Lechner [7] presented an extensive survey of the methods, and Varma and Trachtenberg [26] 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, a linear transformation is used in [6] to reduce circuit complexity. In these works, the methods apply to totally or partially symmetric functions, including

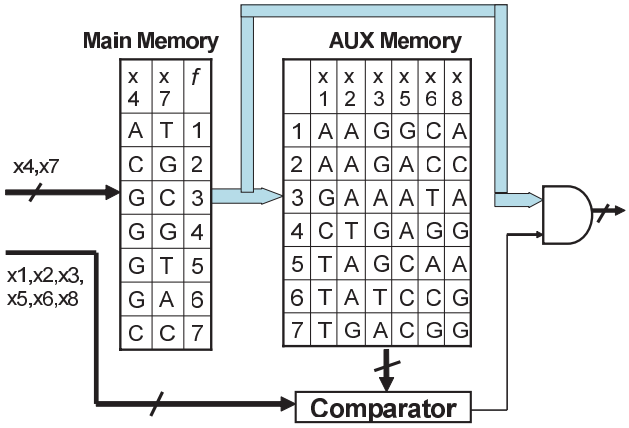


FIGURE 19
Index generation unit for DNA matching.

adders. However, for other functions, linearization are not so effective. Reductions of the sizes of BDDs using linear transformations were considered in [2, 5, 8]. 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 [13], 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 address of the memory. Minimization of variables for multiple-valued index generation functions are also considered in [24].

12 CONCLUSIONS

In this paper, we introduced index generation functions, which have wide applications in pattern matching circuits for the Internet. To represent most incompletely specified index generation functions with weight k, $2\lceil \log_2(k+1) \rceil - 3$ variables are sufficient. We also presented a method to implement an index generation function using an IGU. Reduction of the number of variables using a linear transformation is shown. An extension to multiple-valued input case is also shown. This tutorial is based on [18, 19, 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 [25].

REFERENCES

[1] J. T. Butler and T. Sasao, "Fast constant weight codeword to index converter," *The 54th IEEE International Midwest Symposium on Circuits and Systems*, Korea August 7–10, 2011.

[2] 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.

[3] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] T. Sasao, *Memory Based Logic Synthesis*, Springer, March 2011 .

[19] T. Sasao, "Index generation functions: Recent Developments," International Symposium on Multiple-Valued Logic (ISMVL-2011), Tuusula, Finland, May 23–25, 2011 (invited).

[20] T. Sasao, "Linear transformations for variable reduction," Reed Muller 2011 Workshop, Tuusula, Finland, May 25–26, 2011.

[21] T. Sasao, "Linear decomposition of index generation functions," *17th Asia and South Pacific Design Automation Conference* (ASPDAC-2012), Jan. 30–Feb. 2, 2012, Sydney, Australia, pp. 781–788.

[22] T. Sasao, "Row-shift decompositions for index generation functions," *Design, Automation & Test in Europe* (DATE-2012), March 12–16, 2012, Dresden, Germany, pp. 1585–1590.

[23] T. Sasao, "Multiple-valued input index generation functions: Optimization by linear transformation," *International Symposium on Multiple-Valued Logic* (ISMVL-2012), Victoria, Canada, May 14–16, 2012, pp. 185–190.

[24] 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.

[25] 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.

[26] 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.