IEICE TRANS. INF. & SYST., VOL.E100–D, NO.8 AUGUST 2017

PAPER *Special Section on Multiple-Valued Logic and VLSI Computing*

A Fast Updatable Implementation of Index Generation Functions Using Multiple IGUs*

Tsutomu SASAO†a), *Member*

SUMMARY This paper presents a method to realize index generation functions using multiple Index Generation Units (IGUs). The architecture implements index generation functions more efficiently than a single IGU when the number of registered vectors is very large. This paper proves that independent linear transformations are necessary in IGUs for efficient realization. Experimental results confirm this statement. Finally, it shows a fast update method to IGUs.

key words: *random function, CAM, content-addressable memory, linear decomposition, linear transformation, statistical analysis, update method*

1. Introduction

One of the important tasks in information processing is to find desired data from a large data set. For example, consider a network router, where IP addresses are represented by 32 bits. Assume that a network router stores 40,000 of the 2^{32} possible combinations of the inputs, and checks if an input pattern matches a stored pattern. A content addressable memory (CAM) [4] is a device that performs this operation directly. CAMs are also used for virus scanning and spam-mail filters.

An index generation function [10] describes the operation of a CAM. For example, an index generation function can be represented by a registered vector table such as shown in Table 1. It can also be implemented by an FPGA [7], or a combination of memories and logic. Index generation functions are used in address tables in the Internet, terminal access controllers for local area networks, databases, memory patch circuits, dictionaries, password lists, etc. [10].

An efficient method to implement an index generation function is presented in [10]. It uses a module called **IGU** (Index Generation Unit). Since an IGU uses ordinary memory and a small amount of logic, the cost and the power dissipation are much lower than typical CAM-based implementations.

In this paper, we show an efficient method to store many patterns using *multiple IGUs*. Statistical analysis is used to estimate the size of of the IGUs. The rest of the paper is organized as follows: Section 2 defines the index generation function; Section 3 shows a method to reduce the number of variables of the incompletely specified index generation functions; Section 4 introduces an IGU, the hardware to implement index generation functions; Section 5 shows a method to estimate the number of vectors realized by an IGU; Section 6 shows a method to implement an index generation function using four IGUs, which is more efficient than a single IGU realization; Section 7 shows that independent linear transformations are essential for an efficient implementation of the functions; Section 8 shows the experimental results; Section 9 shows a fast update method for IGUs; and Sect. 10 concludes the paper.

Manuscript received September 20, 2016.
Manuscript publicized May 19, 2017.

†The author is with the Department of Computer Science, Meiji University, Kawasaki-shi, 214–8571 Japan.

*A preliminary version of this paper was presented at ISMVL-2016 [14].

a) E-mail: sasao@cs.meiji.ac.jp

DOI: 10.1587/transinf.2016LOP0001

Table 1 Registered vector table.

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

2. Index Generation Function

In this part, we introduce index generation functions [10], [11], [13].

Definition 2.1: 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 **incompletely specified index generation function** is a one-to-one mapping $D \rightarrow \{1, 2, \ldots, k\}$, where $D \subseteq \{0, 1\}^n$, and $|D| = k$. Since the indices are often greater than two an index generation function is multiple-valued. It produces the corresponding index if the input matches a registered vector. k, the **weight** of the index generation function, is usually much smaller than 2^n, the total number of possible input combinations.

Example 2.1: Table 1 shows a registered vector table for a 4-variable index generation function with weight $k = 4$. ■

3. Number of Variables to Represent an Incompletely Specified Index Generation Function

An incompletely specified index generation function F can often be represented with fewer variables than the original function, when *don't care* values are properly replaced by 0 or some index [1], [2], [6], [8].

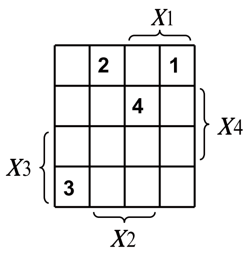


Fig. 1 4-variable index generation function.

Theorem 3.1: Assume that an incompletely specified function F is represented by a decomposition chart [5]. If each column of the decomposition chart has at most one care element, then the function can be represented by only column variables.

Example 3.2: Consider the decomposition chart in Fig. 1. x_1 and x_2 specify columns, while x_3 and x_4 specify rows. Also, blank cells denote *don't cares*. In Fig. 1, each column has at most one care element. Thus, this function can be represented with only the column variables x_1 and x_2: $F = 1 \cdot x_1\bar{x}_2 \vee 2 \cdot \bar{x}_1 x_2 \vee 3 \cdot \bar{x}_1\bar{x}_2 \vee 4 \cdot x_1 x_2$. ■

As for an upper bound on the number of variables, we have the following:

Conjecture 3.1: [10], [11], [13] When the number of the variables n is sufficiently large, most incompletely specified index generation functions with weight k (≥ 7) can be represented by $p = 2\lceil\log_2(k+1)\rceil - 3$ variables.

For an incompletely specified function F, we need to realize a circuit such that $F(x_1, x_2, \ldots, x_n) = 0$ if $(x_1, x_2, \ldots, x_n)$ is a non-registered vector.

4. Index Generation Unit (IGU)

In this section, we show an efficient method to implement an index generation function. With this method, the number of variables to the memory can be reduced. Figure 2 shows the **Index Generation Unit** (IGU). The **linear circuit** has n inputs and p outputs, where $p < n$. It is used to reduce the number of inputs to the main memory. The set of inputs to the linear circuit is partitioned into $X = (X_1, X_2)$, and the output is $Y_1 = (y_1, y_2, \ldots, y_p)$.

We consider two types of linear circuits. The first type is the **single-input linear circuit** shown in Fig. 3. It produces a function $y_j = x_{\pi(j)}$, where π denotes a permutation on n elements. It consists of p multiplexers and p registers, and selects p variables from n input variables. The multiplexers' data inputs are $x_1, x_2, \ldots, x_n$. Registers specify which variables are selected by the multiplexer.

The second type of the circuits is the **double-input linear circuit** shown in Fig. 4. It performs a **linear transformation** $y_i = x_i \oplus x_j$ or $y_i = x_{\pi(i)}$, where $x_i \in X_1$ and $x_j \in X_2$. It uses a pair of multiplexers for each variable y_i. The upper multiplexers have the inputs $x_1, x_2, \ldots, x_n$. The register with $\lceil\log_2 n\rceil$ bits specifies the variable to select by the multiplexer. The lower multiplexers have the inputs $x_1, x_2, \ldots, x_n$, except for x_i. For the i-th input, the constant input 0 is connected instead of x_i. By setting $y_i = x_i \oplus 0$, we can implement $y_i = x_i$. Note that both types of linear circuits produce a special class of linear functions. The **main memory** has p inputs and $q = \lceil\log_2(k+1)\rceil$ outputs. The main memory produces correct indices for registered vectors. However, it may produce incorrect indices for non-registered vectors, because the number of input variables is reduced to p. In an IGU, if the input vector is non-registered, then it produces 0 outputs. To check whether the main memory produces the correct index or not, we use the **AUX memory**. The AUX memory has q inputs and $(n-p)$ outputs: It stores the X_2 part of the registered vectors for each index. The **comparator** checks if the inputs are the same as the registered vector

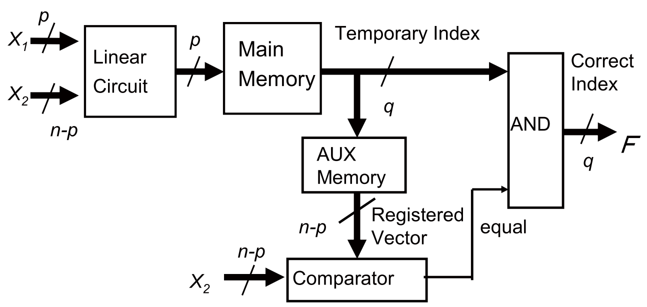


Fig. 2 Index generation unit (IGU).

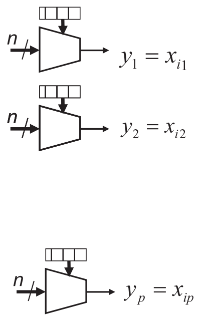
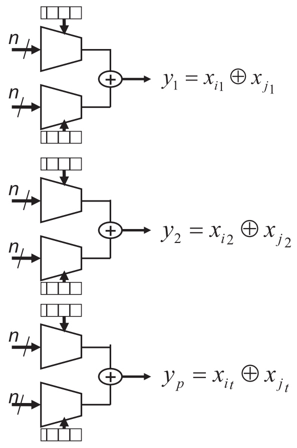


Fig. 3 Single-input linear circuit.



n
n
$y_1 = x_{i_1} \oplus x_{j_1}$
n
n
$y_2 = x_{i_2} \oplus x_{j_2}$
n
n
$y_p = x_{i_t} \oplus x_{j_t}$

Fig. 4 Double-input linear circuit.

or not. If they are the same, the main memory produces the correct index. Otherwise, the main memory produces a wrong index, and the input vector is non-registered. Thus, the **output AND gates** produce $00\ldots0$, showing that the input vector is non-registered. Note that the main memory produces the correct indices only for the registered vectors.

Theorem 4.2: Consider the IGU in Fig. 2. Assume that it realizes the index generation function $F(X_1, X_2)$, where $X_1 = (x_1, x_2, \ldots, x_p)$ and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Also, assume that $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$ for $j \in \{p+1, p+2, \ldots, n\}$, or $y_i = x_i$, are applied to the input to the main memory. Then, F can be realized by the circuit where the AUX memory stores only the values for X_2.

5. Number of Vectors Realized by an IGU

In this section, we review the expected number of registered vectors realized by an IGU [10].

Lemma 5.1: When $0 < \alpha \ll 1$, $1-\alpha$ can be approximated by $e^{-\alpha}$.

Lemma 5.2: Let $F(X)$ be a uniformly distributed random index generation function of n variables with weight k, where $k \ll 2^n$. Consider a decomposition chart [5], where p is the number of variables labelling the columns. Then, the probability that a column of the decomposition chart has all-zero elements is approximately $e^{-\xi}$, where $\xi = \frac{k}{2^p}$.

Theorem 5.3: 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 $\xi = \frac{k}{2^p}$.

6. Realization Using Four IGUs

In an IGU, the main memory has p inputs and $q = \lceil \log_2(k+1) \rceil$ outputs, while the AUX memory has q inputs and $(n-p)$ outputs. Thus, the total amount of memory for an IGU is $q2^p + (n-p)2^q$.

Conjecture 3.1 shows that to implement an index generation function with weight k by an IGU, the number of inputs to the main memory is $p \simeq 2\log_2 k - 3$. Also, note that $q \simeq \log_2 k$ and $n \ll k$. Thus, the size of the memory is $O(k^2 \log k)$.

This shows that, when k is large, a single IGU realization of an index generation function is inefficient.

Example 6.3: Let $k = 2^{20} - 1$. Then, by Conjecture 3.1, we have $p = 2\lceil \log_2(k+1) \rceil - 3 = 37$. Thus, the size of the main memory in a single IGU realization is $q2^p = 20 \times 2^{37} = 2.75 \times 10^{12}$ bits. Thus, we need a more efficient method. ■

To reduce the total amount of memory, we partition the registered vectors into m groups, and realize each group independently [3], [9]. Figure 5 shows a network using four

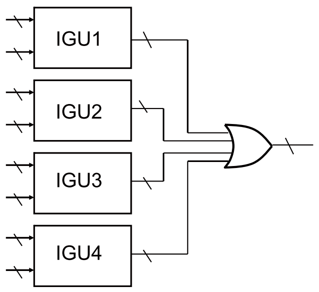


Fig. 5 Realization of an index generation function by 4IGU.

IGUs. This architecture is called a **4IGU** [9]. In this case, we should use independent linear transformations for different IGUs. The importance of the linear transformations will be discussed in Sect. 7.

Next, we show that index generation functions can be realized with a 4IGU. This is more efficient than a single IGU realization when k is large.

Theorem 6.4: Consider an index generation function with weight k. Then, more than 99.9% of the registered vectors can be realized by a 4IGU, where the number of input variables to the main memory for each IGU is $p = \lceil \log_2(k+1) \rceil$.

(Proof) Let $k_1 = k$. We assume that, for each IGU, the distribution of the vectors is uniform.

1. IGU_1: Let $\xi_1 = \frac{k_1}{2^p}$.
The number of realized vectors is $2^p(1-e^{-\xi_1})$.
The number of remaining vectors is
$$k_2 = k_1 - 2^p(1-e^{-\xi_1}) = k_1 + 2^p(e^{-\xi_1} - 1).$$
2. IGU_2: Let $\xi_2 = \frac{k_2}{2^p} = \frac{k_1}{2^p} + (e^{-\xi_1} - 1)$.
The number of realized vectors is $2^p(1-e^{-\xi_2})$.
The number of remaining vectors is
$$\begin{aligned} k_3 &= k_1 - 2^p(1-e^{-\xi_1}) - 2^p(1-e^{-\xi_2}) \\ &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} - 2). \end{aligned}$$
3. IGU_3: Let $\xi_3 = \frac{k_3}{2^p} = \frac{k_1}{2^p} + (e^{-\xi_1} + e^{-\xi_2} - 2)$.
The number of realized vectors is $2^p(1-e^{-\xi_3})$.
The number of remaining vectors is
$$\begin{aligned} k_4 &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} - 2) - 2^p(1-e^{-\xi_3}) \\ &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} - 3). \end{aligned}$$
4. IGU_4: Let $\xi_4 = \frac{k_1}{2^p} + (e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} - 3)$.
The number of realized vectors is $2^p(1-e^{-\xi_4})$.
The number of remaining vectors is
$$\begin{aligned} k_5 &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} - 3) - 2^p(1-e^{-\xi_4}) \\ &= k_1 + 2^p(e^{-\xi_1} + e^{-\xi_2} + e^{-\xi_3} + e^{-\xi_4} - 4). \end{aligned}$$

When $k_1 = 2^p$, the fraction of the original vectors that remain is about 1.6×10^{-6}. □

Note that, in the proof, we assumed that IGUs have independent linear transformations, so that the distribution of the vectors are uniform.

Example 6.4: Consider an index generation function with weight $k = 2^{20} - 1 = 1048575$. Let us realize the function by the 4IGU shown in Fig. 5. Suppose that the number of inputs to the main memory in each IGU is $p = 20$. We assume that for each IGU, the distribution of the vectors is uniform.

1. IGU_1: Let $\xi_1 = \frac{k_1}{2^p} = \frac{1{,}048{,}575}{2^{20}} = 0.9999990$. It realizes $2^p(1 - e^{-\xi_1}) = 1048576 \times 0.6321203 \simeq 662826$ registered vectors. The number of remaining vectors is $k_2 = 385749$.
2. IGU_2: Let $\xi_2 = \frac{k_2}{2^p} = \frac{385749}{2^{20}} = 0.3678789$. It realizes $2^p(1 - e^{-\xi_2}) = 1048576 \times 0.3077990 \simeq 322750$ registered vectors. The number of remaining vectors is $k_3 = 62999$.
3. IGU_3: Let $\xi_3 = \frac{k_3}{2^p} = \frac{62999}{2^{20}} = 0.0600805$. It realizes $2^p(1 - e^{-\xi_3}) = 1048576 \times 0.0583113 \simeq 61143$ registered vectors. The number of remaining vectors is $k_4 = 1856$.
4. IGU_4: Let $\xi_4 = \frac{k_4}{2^p} = \frac{1856}{2^{20}} = 0.0017700$. It realizes $2^p(1 - e^{-\xi_4}) = 1048576 \times 0.0017685 \simeq 1854$ registered vectors. The number of remaining vectors is only $k_5 = 2$.

■

Note that, in a 4IGU, the main memory of each IGU has p inputs and p outputs, while the AUX memory has p inputs and $(n-p)$ outputs. Thus, the total amount of memory for each IGU is

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

Then, the total memory for the 4IGU is $4n2^p$. Thus, when $n = 40$ and $p = 20$, the 4IGU requires $4n2^p = 4 \times 40 \times 2^{20} = 167.7 \times 10^6$ bits. This is more efficient than the single IGU realization in Example 6.3, which requires 2.75×10^{12} bits.

Definition 6.2: Let the linear circuit realize the p compound variables:

$$\begin{aligned} y_1 &= a_{1,1}x_1 \oplus a_{1,2}x_2 \oplus \cdots \oplus a_{1,n}x_n, \\ y_2 &= a_{2,1}x_1 \oplus a_{2,2}x_2 \oplus \cdots \oplus a_{2,n}x_n, \\ &\cdots\cdots\cdots \\ y_p &= a_{p,1}x_1 \oplus a_{p,2}x_2 \oplus \cdots \oplus a_{p,n}x_n. \end{aligned}$$

Then, the **transformation matrix** is

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p,1} & a_{p,2} & \dots & a_{p,n} \end{bmatrix}.$$

Definition 6.3: Let A and B be two transformation matrices of $p \times n$. The **rank** of a matrix A is the number of linearly independent row vectors, and denoted by $rank(A)$. Matrix B **depends** on A if

$$rank(A) = rank \begin{bmatrix} A \\ B \end{bmatrix}.$$

Otherwise, B is **independent** of A.

Example 6.5: Consider three matrices:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix},$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Since, $rank(A) = rank \begin{bmatrix} A \\ B \end{bmatrix} = 2 < rank \begin{bmatrix} A \\ C \end{bmatrix} = 3$, B depends on A, while C is independent of A. ■

Theorem 6.5: Consider two decomposition charts for an index generation function. Assume that in the first chart, the column variables are $Y = (y_1, y_2, \dots, y_p)$, while in the second chart, the column variables are $Z = (z_1, z_2, \dots, z_p)$. Also assume that the row variables are the same. If two transformation matrices for Y and Z are dependent each other, then one decomposition chart is obtained from the other by permuting the columns of the other chart. Thus, the numbers of variables to represent two functions that corresponds to these two decomposition charts are the same.

Example 6.6: Consider the function $f_1(x_1, x_2, x_3, x_4)$ in Fig. 1, where $X_1 = (x_1, x_2)$ are the column variables and $X_2 = (x_3, x_4)$ are the row variables. Let $Y_1 = (y_1, y_2)$, where $y_1 = x_1$ and $y_2 = x_1 \oplus x_2$. Consider the decomposition chart, where $Y_1 = (y_1, y_2)$ are column variables. Figure 6 (left) is the corresponding chart, and let $f_2(y_1, y_2, x_3, x_4)$ be the function. Note that columns for $y_1 = 1$ are permuted. Thus, the numbers of variables to represent two functions $f_1(x_1, x_2, x_3, x_4)$ and $f_2(y_1, y_2, x_3, x_4)$ are the same, and both are two.

Next, consider the decomposition chart, where $Z_1 = (z_1, z_2)$, $z_1 = x_1$ and $z_2 = x_2 \oplus x_3$, are column variables. Figure 6 (right) is the corresponding chart, and let $f_3(z_1, z_2, x_3, x_4)$ be the function. Compared with Fig. 1, the element 3 is moved to the right in Fig. 6 (right). The number of variables to represent $f_3(z_1, z_2, x_3, x_4)$ is different from that of $f_1(x_1, x_2, x_3, x_4)$. Note that $f_1(x_1, x_2, x_3, x_4)$ corresponds to the matrix A, $f_2(y_1, y_2, x_3, x_4)$ corresponds to the matrix B, and $f_3(z_1, z_2, x_3, x_4)$ corresponds to the matrix C, in Example 6.5. ■

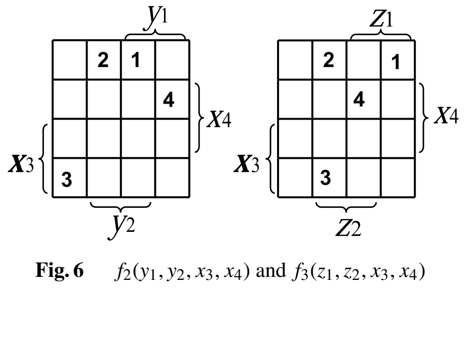


Fig. 6 $f_2(y_1, y_2, x_3, x_4)$ and $f_3(z_1, z_2, x_3, x_4)$

			0	0	0	0	1	1	1	1	x_3
			0	0	1	1	0	0	1	1	x_2
			0	1	0	1	0	1	0	1	x_1
0	0	0					**7**	**10**	**1**		
0	0	1				**20**	*14*		*6*		
0	1	0	**1**			*15*		*13*			
0	1	1		**18**							
1	0	0	*16*							**17**	
1	0	1							$\underline{19}$	*3*	
1	1	0					$\underline{12}$			$\underline{9}$	
1	1	1	$\underline{4}$	*11*	**2**		5			8	
x_6	x_5	x_4									

Fig. 7 Decomposition chart for $F(X_1, X_2)$.

7. Selection of Linear Transformations

In the previous sections, we assume that IGUs have independent linear transformations. However, when the linear transformations are the same for all the IGUs, the number of registered vectors realized by IGUs will be decreased. In this part, we will prove this using statistical analysis. First, we illustrate the design method for a 4IGU.

Example 7.7: Consider a realization of an index generation function shown in Fig. 7 by a 4IGU. It is a random function of 6 variables. Blank entries denote 0's. Note that the column variables are $X_1 = (x_1, x_2, x_3)$, while the row variables are $X_2 = (x_4, x_5, x_6)$. Assume that the column variables are used for the main memories. The number of registered vectors is $k = 20$. The registered vectors are divided into four subsets, and realized separately as follows:

1. IGU1 realizes the mapping of vectors to index values **1, 18, 2, 20, 7, 10, 1**, and **17** (topmost registered vectors in boldface numbers).
2. IGU2 realizes the mapping of vectors to index values *16, 11, 15, 14, 13, 6*, and *3* (vectors in italic numbers).
3. IGU3 realizes the mapping of vectors to index values $\underline{4}$, $\underline{12}$, $\underline{19}$, and $\underline{9}$ (vectors in underlined numbers).
4. IGU4 realizes the mapping of vectors to index values 5 and 8.

When $X_1 = (x_1, x_2, x_3)$ are used for the main memories, four IGUs are necessary to implement the function. ■

Theorem 7.6: Let k be the number of registered vectors, and p be the number of inputs to the main memory. Then, the expected number of vectors realized by a 4IGU using the same linear transformations is

$$2^p[4 - e^{-\beta}(4 + 3\beta + \beta^2 + \frac{1}{6}\beta^3)],$$

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

(Proof) Consider the decomposition chart of a random index generation function. Let p be the number of inputs to the main memory. Note that the number of non-zero elements in the decomposition chart correspond to that of the distinct balls in distinct 2^p bins. Assume that k balls are randomly thrown into $N_1 = 2^p$ bins. Also assume that k and N_1 are large. Let $\alpha = \frac{1}{N_1}$. Then, $\alpha k = \beta$.

No Ball: The probability that a certain bin has no ball after one throw is

$$\frac{N_1 - 1}{N_1} = 1 - \alpha.$$

The probability that a certain bin has no ball after k throws:

$$P_0 = (1 - \alpha)^k \simeq e^{-\alpha k} = e^{-\beta},$$

because each throw is an independent event.

One Ball: The probability that a certain bin has one ball after one throw is α. The probability that a certain bin has exactly one ball after k throws:

$$\begin{aligned} P_1 &= \binom{k}{1}\alpha(1-\alpha)^{k-1} \\ &= k\alpha(1-\alpha)^{k-1} \\ &\simeq \beta e^{-\alpha(k-1)} \simeq \beta e^{-\beta}. \end{aligned}$$

Two Balls: The probability that a certain bin has two balls after two throws is α^2. The probability that a certain bin has exactly two balls after k throws:

$$\begin{aligned} P_2 &= \binom{k}{2}\alpha^2(1-\alpha)^{k-2} \\ &\simeq \frac{1}{2}\beta^2 e^{-\alpha(k-2)} \simeq \frac{1}{2}\beta^2 e^{-\beta}. \end{aligned}$$

Three Balls: The probability that a certain bin has three balls after three throws is α^3. The probability that a certain bin has just three balls after k throws:

$$\begin{aligned} P_3 &= \binom{k}{3}\alpha^3(1-\alpha)^{k-3} \\ &= \frac{k(k-1)(k-2)}{3!}\alpha^3(1-\alpha)^{k-3} \\ &\simeq \frac{1}{3!}\beta^3(1-\alpha)^{k-3} \simeq \frac{1}{3!}\beta^3 e^{-\beta}. \end{aligned}$$

In this case, most of the vectors can be realized by a 4IGU as follows:

1. IGU1 stores one element from each of the columns that have at least one element. It stores $2^p(1 - P_0)$ vectors, on the average.
2. IGU2 stores one element from each of the columns that have two or more elements. It stores $2^p[1 - (P_0 + P_1)]$ vectors, on the average.
3. IGU3 stores one element from each of the columns that have three or more elements. It stores $2^p[1 - (P_0 + P_1 + P_2)]$ vectors, on the average.
4. IGU4 stores one element from each of the columns that have four or more elements. It stores $2^p[1 - (P_0 + P_1 + P_2 + P_3)]$ vectors, on the average.

Thus, in total, the 4IGU stores $2^p[4 - (4P_0 + 3P_1 + 2P_2 + P_3)]$ vectors, on the average. □

Example 7.8: Let $k = 2^{20} - 1$ and $p = 20$. In this case, we have $\beta \simeq 1.0$. When the same linear transformations are used, on the average, IGU1 stores $2^{20}(1 - P_0) = 662826$ vectors; IGU2 stores $2^{20}[1 - (P_0 + P_1)] = 277076$ vectors; IGU3 stores $2^{20}[1 - (P_0 + P_1 + P_2)] = 84201$ vectors; and IGU4 stores $2^{20}[1 - (P_0 + P_1 + P_2 + P_3)] = 19910$ vectors. The number of remaining vectors is 4562, on the average. ■

Example 7.9: When independent linear transformations are used, the function in Fig. 7 can be realized with only three IGUs. In this case, IGU1 and IGU3 use $X_1 = (x_1, x_2, x_3)$ as inputs to the main memory, while IGU2 uses $X_2 = (x_4, x_5, x_6)$ as inputs to the main memory. The registered vectors are divided into three parts, and realized separately as follows:

1. IGU1 stores one element for each non-empty column. It realizes the mapping of vectors to index values 4, 11, 2, 15, 5, 13, 19, and 9.
2. IGU2 stores one element for each row. It realizes the mapping of vectors to index values 7, 14, 1, 18, 16, 3, 12, and 8.
3. IGU3 stores the remaining elements for four columns. It realizes the mapping of vectors to index values 20, 10, 6, and 17.

In this case, all the vectors can be realized by three IGUs. ■

8. Experimental Results

8.1 Realization with 4IGUs

To show the validity of the analysis, we generated 100 random index generation functions with $n = 40$ and $k = 2^{20} - 1$, and realized them by 4IGUs, where $p = 20$.

In the experiment, we used the following linear transformations: Let $(x_1, x_2, \ldots, x_n)$ be the input variables. For the i-th IGU, $(y_1, y_2, \ldots, y_p)$ were used as the inputs to the main memory, where $y_j = x_j \oplus x_{p-i+j}$, $(1 \leq j \leq p)$. Table 2 compares the estimated values and experimental results. The column labeled *Estimated* denotes the results that were obtained in Example 6.4. The column labeled *Experimental* shows the average of 100 sample functions.

In the estimation, the remaining vectors not realized by the 4IGU is only two, that is $k_5 = 2$. On the other hand, in the experiment, the number of the remaining vectors is 1.82, on the average.

The reasons for the disparity may be

- The approximations in the estimation made an error.
- The registered vectors in the experiment were not truly random.
- The number of sample functions were not sufficient.

In practice, we can easily find a good linear transformation using a minimization tool [12] for the last IGU. Thus, each function can be realized by a 4IGU. The total amount of memory is $mn2^p = 4 \times 40 \times 2^{20} = 160 \times 2^{20} \simeq 167.8 \times 10^6$.

Table 2 Numbers of Vectors Realized by 4IGU ($k = 2^{20} - 1$)

IGU	Estimated		Experiment	
	k_i	Realized Vectors	k_i	Realized Vectors
1	1048575	662826	1048575.00	656230.28
2	385749	322750	381858.97	319474.10
3	62999	61143	62384.87	60550.56
4	1856	1854	1834.31	1832.49
Remain	2		1.82	

Table 3 Average Numbers of Vectors Realized by 4IGU ($k = 2^{20} - 1$)

IGU	Same Transformations		Independent Transformations	
	k_i	Realized Vectors	k_i	Realized Vectors
1	1048575.00	656209.60	1048575.00	656230.28
2	381879.65	274321.17	381858.97	319474.10
3	107558.48	83343.99	62384.87	60550.56
4	24214.49	19709.61	1834.31	1832.49
Remain	4504.88		1.82	

8.2 Effect of Independent Linear Transformations

In Sect. 7, we showed that independent linear transformations should be used for IGUs. To demonstrate this, we used the previous 100 random index generation functions with $n = 40$ and $k = 2^{20} - 1$, and realized them by 4IGUs, where $p = 20$. Table 3 compares the two 4IGU realizations. In the column labeled *Same*, the same linear transformations are used for four IGUs. In the column labeled *Independent*, independent linear transformations were used for the different IGUs. The sample functions are the same as that of Table 2.

The effect is very clear. When the same linear transformations are used for the 4IGU, on the average, 4504.88 vectors remain, which is not far from the estimated value 4562 in Example 7.8. On the other hand, when the independent linear transformations are used for the 4IGU, on the average, only 1.82 vectors remained which is near to the estimated value 2.0 in Example 6.4.

9. Fast Update of IGUs

Index generation functions often require quick update. For example, in the routers for the internet, registered vectors are often updated in every milli second. In this part, we show a fast method to update registered vectors.

To update the registered vectors quickly, we have to modify IGUs. Figure 8 shows a fast updatable IGU. Two additional outputs are appended to the original IGU shown in Fig. 2. The first one is the collision detection signal (CD). It shows that the main memory produces non-zero output. CD can be generated by the OR of the outputs of the main memory. The second one is the match signal (MT). MT shows that the input vector matches to a registered vector in the IGU. With these two signals, an update of the registered vectors becomes quite easy. An update of a registered vector can be done by two steps:

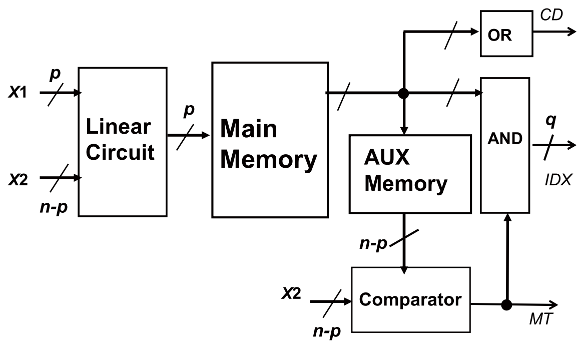


Fig. 8 A fast updatable IGU.

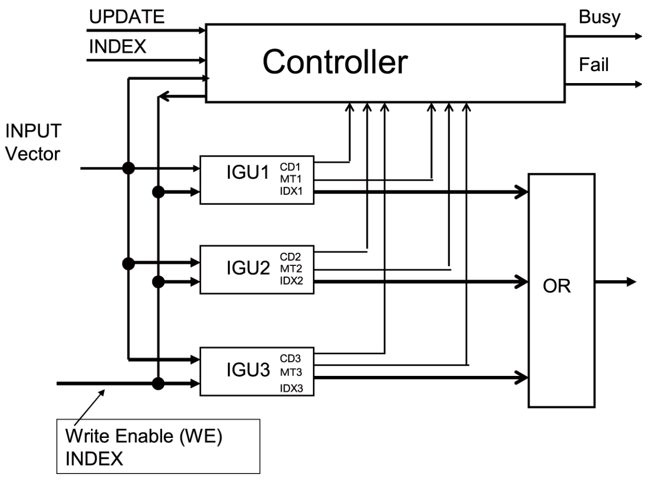


Fig. 9 Circuit for update.

1. Delete the registered vector.
2. Append a new registered vector.

Figure 9 illustrates an update circuit using three IGUs.

When the UPDATE signal is 1, this circuit performs update. During the update, the Busy signal is 1, and the search operation is forbidden. When the update is successful (i.e., the new vector can be stored in one of the IGUs in Fig. 9) the fail signal is 0, while when the update is unsuccessful the fail signal is 1.

Let $X = (X_1, X_2)$ be the vector to be updated, and let $INDEX(X)$ be its index. According to the values of the collision detection signal (CD) and the match signal (MT), the following operations are done.

1. When $(CD, MT) = (1, 1)$. Since the registered vector exists in this IGU, rewrite the index.
2. When $(CD, MT) = (0, 0)$. Since the registered vector does not exit, append the vector, and add the corresponding index.
3. When $(CD, MT) = (1, 0)$. Since other registered vector exist in this IGU, try to update other IGU.
4. When $(CD, MT) = (0, 1)$. This state never happens.

The controller works as follows: Then the UPDATE signal is 1, the state is the update mode. When the UPDATE is 1 and INDEX is non-zero, it appends a new registered vector. When the UPDATE is 1 and INDEX is zero, it deletes the corresponding registered vector.

With CD and MT signals, the controller can quickly locate the IGU that stores the registered vectors to delete, or the IGU to append the new registered vector.

Table 4 Registered vector table.

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

Table 5 Registered vectors stored in IGU1.

Vector						Index
x_1	x_2	x_3	x_4	x_5	x_6	
0	0	0	1	1	1	4
0	0	1	1	1	1	5
0	1	0	1	1	1	2
0	1	1	0	0	0	1
1	0	0	1	1	1	11
1	0	1	0	0	0	10
1	1	0	0	1	0	15
1	1	1	1	0	1	3

Table 6 Registered vectors stored in IGU2.

Vector						Index
x_4	x_5	x_6	x_1	x_2	x_3	
0	0	0	0	0	1	7
0	0	1	–	–	–	0
0	1	0	1	0	1	13
0	1	1	0	0	1	12
1	0	0	0	1	1	6
1	0	1	–	–	–	0
1	1	0	–	–	–	0
1	1	1	1	1	1	8

Example 9.10: Implementation of the original function Consider the registered vector table shown in Table 4. It has $n = 6$ variables, and its weight is $k = 15$. Lets us implement this function by three IGUs each of which has a main memory with three inputs. IGU1 implements the function shown in Table 5. The inputs are (x_1, x_2, x_3), and its weight is 8. Note that the main memory and the AUX are combined together.

IGU2 implements the function shown in Table 6. The inputs are (x_4, x_5, x_6), and its weight is 5. The index 0 denotes that no vector is stored in the memory.

IGU3 implements the function shown in Table 7. The inputs are (y_1, y_2, y_3), where

$$\begin{aligned} y_1 &= x_1 \oplus x_4, \\ y_2 &= x_2 \oplus x_5, \\ y_3 &= x_3 \oplus x_6, \end{aligned}$$

Table 7 Registered vectors stored in IGU3.

Vector						Index
y_1	y_2	y_3	x_4	x_5	x_6	
0	0	0	–	–	–	0
0	0	1	–	–	–	0
0	1	0	–	–	–	0
0	1	1	–	–	–	0
1	0	0	0	1	1	9
1	0	1	1	0	0	14
1	1	0	–	–	–	0
1	1	1	–	–	–	0

and its weight is 2. Note that these three IGUs implement all the registered vectors shown in Table 4.

Deletion of a vector
To delete a vector, the UPDATE signal to the controller in Fig. 9 is set to 1. Then, the vector to be deleted is set to the inputs of three IGUs. Each IGU checks if the vector is stored or not. If the vector is stored, the match signal (MT) will be one. Next, the controller identifies the IGU that stores the input vector using ($MT1, MT2, MT3$) and ($CD1, CD2, CD3$). If an IGU store the input vector, both the match signal (MT) and collision signal (CD) will be one.

If there is an IGU that store the input vector, the controller send the WE signal to the corresponding IGU, and rewrite the index to zero.

For example, to delete the first vector in Table 4, the controller rewrite the the index of the vector $(x_1, x_2, x_3) = (0, 1, 1)$ in IGU1, to zero.

Addition of a vector
To add a new vector, UPDATE signal is set to 1. Also the index to be add is set to the INDEX of the controller. Then, the vector to be add is set to the inputs of three IGUs. Each IGU checks if a collision exists or not. If the collision exists, the collision signal (CD) will be one.

The controller find the IGU that has no collision with the input vector using the values of $(CD1, CD2, CD3)$. If a vacant location is available in an IGU, the controller write the index to the IGU by setting WE to 1.

Assume that a new vector $(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 1, 0, 0, 0, 0)$ and index = 1 is appended. IGU1 and IGU2 have collisions, but IGU3 has no collision. Thus, the controller writes the index = 1 to the address $(y_1, y_2, y_3) = (1, 1, 0)$. ■

10. Conclusion and Comments

In this paper, we presented a method to implement index generation functions using multiple IGUs. Important results are

- An index generation function with many registered vectors should be realized by an mIGU rather than a single IGU.
- Most index generation function with weight k can be realized by a 4IGU, where $p = \lceil \log_2(k + 1) \rceil$.
- In an mIGU, the liner transformations should be independent.

With the result of this paper, we can estimate the size of of the IGUs necessary to implement a given number of vectors.

In the application to the internet, the registered vectors must be updated frequently, but only a short time is available for reconfiguration. With updatable IGUs, we can quickly update the vectors.

In this paper, to insert a new vector, we used the IGU with the smallest index. Although this strategy works well in most cases, it can fail to store a few vectors in the given IGUs. In such a case, we can use one of the following methods:

- Increase the number of IGUs.
- Use an additional small CAM to store the remaining vectors.
- Optimize the linear transformation [12] of the last IGU to store all the remaining vector into the last IGU. Since the number of registered vectors to store in the last IGU is much smaller than its capacity, all the remaining vectors can be stored in the last IGU. Conjecture 3.1 shows a sufficient number of variables to represent a given index generation function.

Acknowledgments

This research is partly supported by the Japan Society for the Promotion of Science (JSPS) Grant in Aid for Scientific Research. Discussions with Drs. Hiroki Nakahara, Hisashi Iwamoto, Kazunari Inoue, and Jon T. Butler were quite useful. The author thanks the reviewers who found some misprints.

References

[1] C. Halatsis and N. Gaitanis, "Irredundant normal forms and minimal dependence sets of a Boolean functions," IEEE Trans. Comput., vol.C-27, no.11, pp.1064–1068, Nov. 1978.
[2] Y. Kambayashi, "Logic design of programmable logic arrays," IEEE Trans. Comput., vol.C-28, no.9, pp.609–617, Sept. 1979.
[3] Y. Matsunaga, "Synthesis algorithm for parallel index generator," IEICE Trans. Fund. Electronics, Communications and Computer Sciences, vol.E97-A, no.12, pp.2451–2458, Dec. 2014.
[4] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," IEEE J. Solid-State Circuits, vol.41, no.3, pp.712–727, March 2006.
[5] T. Sasao, Switching Theory for Logic Synthesis, Kluwer Academic Publishers, 1999.
[6] T. Sasao, "On the number of dependent variables for incompletely specified multiple-valued functions," International Symposium on Multiple-Valued Logic (ISMVL-2000), pp.91–97, May 2000.
[7] T. Sasao and H. Nakahara, "Implementations of reconfigurable logic arrays on FPGAs," International Conference on Field-Programmable Technology 2007 (ICFPT'07), Kitakyushu, Japan, pp.217–223, Dec. 12–14, 2007.
[8] T. Sasao, "On the numbers of variables to represent sparse logic functions," International Conference on Computer Aided Design (ICCAD-2008), pp.45–51, Nov. 2008.
[9] T. Sasao, M. Matsuura, and H. Nakahara, "A realization of index generation functions using modules of uniform sizes," 19th International Workshop on Logic and Synthesis (IWLS-2010), pp.201–208, June 18–20, 2010.

[10] T. Sasao, Memory-Based Logic Synthesis, Springer, 2011.
[11] T. Sasao, "Index generation functions: Recent developments," (invited paper) International Symposium on Multiple-Valued Logic (ISMVL-2011), Tuusula, Finland, May 23–25, 2011.
[12] T. Sasao, "Linear decomposition of index generation functions," 17th Asia and South Pacific Design Automation Conference (ASPDAC-2012), Sydney, Australia, pp.781–788, Jan. 30–Feb. 2, 2012.
[13] T. Sasao, "Index generation functions: Tutorial," Journal of Multiple-Valued Logic and Soft Computing, vol.23, no.3-4, pp.235–263, 2014.
[14] T. Sasao, "A realization of index generation functions using multiple IGUs," International Symposium on Multiple-Valued Logic, (ISMVL-2016), Sapporo, Japan, pp.113–118, May 17–19, 2016.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in Electronics Engineering from Osaka University, Osaka Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan; IBM T. J. Watson Research Center, Yorktown Height, NY; the Naval Postgraduate School, Monterey, CA; and Kyushu Institute of Technology, Iizuka, Japan. Now, he is a Professor of Department of Computer Science, Meiji University, Kawasaki, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than 9 books on logic design including, Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis, Logic Synthesis and Verification, and Memory-Based Logic Synthesis, in 1993, 1996, 1999, 2001, and 2011, respectively. He has served Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan in 1998. He received the NIWA Memorial Award in 1979, Takeda Techno-Entrepreneurship Award in 2001, and Distinctive Contribution Awards from IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003, 2004 and 2012. He has served an associate editor of the IEEE Transactions on Computers. He is a Life Fellow of the IEEE.