A Design Method of Address Generators Using Hash Memories

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

Abstract

An address generator produces a unique address from 1 to k when the input that matches one of k registered vectors, and produces 0 for other inputs. This paper presents a method to design an address generator using a hash memory and an LUT cascade. The hash memory realizes about 90% of the registered vectors, while the LUT cascade realizes the remaining 10% of the registered vectors. This method uses a non-disjoint functional decomposition to reduce the size of memory. The experimental results using lists of English words show that the usefulness of the approach. The total amount of memory is only 20% to 25% of the memory necessary to implement the function by using an LUT cascade alone. Theoretical analysis supports the experimental results.

1 Introduction

Consider a set of k distinct binary vectors of n bits. An address generation function produces a unique address from 1 to k for the input that matches a vector in the set, and produces 0 for vectors outside the set [12]. Address generation functions are used in hardware for the internet [1], memory patching circuits [7], etc. In this paper, we assume that the number of vectors k in the set is much smaller than that of the maximal possible input combinations 2^n.

For example, consider an address generation function with $n = 32$ and $k = 40,000$. The straightforward way to implement this address generation function is to store the truth table into a memory. However, this method require a memory with unrealistic size, since the size of the memory is proportional to 2^n. Another method to implement the function is a two-level logic circuit or a Programmable Logic Array (PLA). Unfortunately, this method still requires large chip area.

The third method is a content addressable memory (CAM)[6]. It requires a special circuit that cannot be implemented by ordinary logic gates or memory. Recently, we developed a method that uses an LUT cascade [13]. The LUT cascade is realized by a series connection of ordinary memories. It realizes address generation functions efficiently when k is up to $k = 1000$. However, when k is larger, an LUT cascade requires cells with $\lceil \log_2(k+1) \rceil + 1$ inputs. So, it can be too large for the available resource in an embedded system.

In this paper, we present an efficient method to implement an address generation function by a hybrid method that uses a hash memory and an LUT cascade. The hash memory implements about 90% of the vectors, while the LUT cascade realizes remaining 10% of the vectors. The hybrid method requires only 20 to 25% of the memory necessary to implement the function by an LUT cascade alone. Theoretical analysis is also done, which supports the experimental results.

Besides address generation functions, this design method can implement an n-variable function, where the number of non-zero outputs k is much smaller than 2^n.

2 Address Generation Function

Definition 2.1 *Consider a set of k binary vectors of n bits. Denote these vectors as* **registered vectors***. For each registered vector, assign unique integer from* 1 *to k. A* **registered vector table** *shows the relation of registered vectors and corresponding integers. An* **address generation function** *produces the corresponding integer if the input matches to a registered vector, and produces* 0 *otherwise. k is the* **weight** *of the address generation function.*

In this paper, we assume that k is much smaller than 2^n, the total number of input combinations.

Example 2.1 *Table 2.1 shows a registered vector table consisting of 7 vectors. The corresponding address generation function is shown in Table 2.2, where only non-zero outputs are shown. It produces a 3-bit number (e.g., 001) corresponding to the integer of the matched vector. When no entry matches to the input vector, the function produces* 000. *(End of Example)*

Table 2.1: Registered vector table

Address	Vector
1	000010
2	010010
3	001010
4	001110
5	000001
6	111011
7	010111

Table 2.2: Address generation function

x_1	x_2	x_3	x_4	x_5	x_6	f_2	f_1	f_0
0	0	0	0	1	0	0	0	1
0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	0	1	1
0	0	1	1	1	0	1	0	0
0	0	0	0	0	1	1	0	1
1	1	1	0	1	1	1	1	0
0	1	0	1	1	1	1	1	1

3 Realization Using LUT Cascade

An address generation function can be directly implemented by an ordinary memory. For example, the address generation function shown in Table 2.2 can be directly implemented by a 64-word memory, where each word consists of 3 bits. In the case of an address generation function of n variables, the size of the memory is proportional to 2^n even if the registered vector table contains only a few elements. For such a case, the LUT cascade realization greatly reduces the necessary amount of memory.

Definition 3.1 *Given a function $f(X) : B^n \rightarrow \{0,1,\ldots,k\}$, where $B = \{0,1\}$ and $X = (x_1,x_2,\ldots,x_n)$. Let (X_1,X_2) be a partition of X. Let n_1 be the number of variables in X_1, and let n_2 be the number of variables in X_2 ($n_1+n_2 = n$). The* **decomposition chart** *of f is a two-dimensional matrix with 2^{n_1} columns and 2^{n_2} rows. The column labels correspond to all possible binary numbers of n_1 bits, and the row labels correspond to all possible binary numbers of n_2 bits. And the corresponding matrix value is equal to $f(X_1,X_2)$. Among the decomposition charts for the function f, one with $X_1 = (x_1,x_2,\ldots,x_{n_1})$ and $X_2 = (x_{n_1+1},x_{n_1+2},\ldots,x_n)$ is the* **standard decomposition chart**. *The number of different column patterns in a decomposition chart is the* **column multiplicity**. *As a special case of a decomposition chart, we also consider the case where $X_1 = X$.*

Definition 3.2 *Let $(x_1,x_2,\ldots,x_n)$ be the ordering of the input variables. The* **C-measure** *of f is the maximum column multiplicity over all the standard decomposition charts for the function f.*

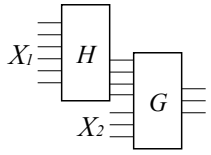

Figure 3.1: Realization of logic function by decomposition.

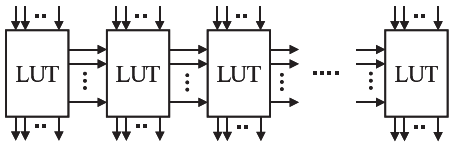

Figure 3.2: LUT cascade with intermediate outputs

Example 3.1 *The C-measure of $f_1 = x_1x_2 \vee x_3x_4 \vee x_5x_6$ is 3, while the C-measure of $f_2 = x_1x_5 \vee x_2x_6 \vee x_3x_4$ is 8. (End of Example)*

Lemma 3.1 *The C-measure of a logic function f with weight k is at most $k+1$.*

Theorem 3.1 *[2] For a given function f, let X_1 be the variables corresponding to the columns, and let X_2 be the variables corresponding to the rows of the decomposition chart. Let μ be the column multiplicity of the decomposition chart. Then, the function f is realized by the circuit shown in Fig. 3.1. In this case, the number of connections that connect two blocks H and G is $\lceil \log_2 \mu \rceil$.*

When the number of connections that connect two blocks is smaller than the number of variables in X_1, we have a chance to reduce the amount of memory to realize the function. The decomposition shown in Fig. 3.1 is a **disjoint decomposition**, since X_1 and X_2 do not have a common element. By decomposing the given function iteratively, we have an **LUT cascade**[9] shown in Fig. 3.2. An LUT cascade consists of **cells** connected to each other by **rails**. A function with a small C-measure can be realized with a compact LUT cascade.

Theorem 3.2 *[11] A logic function with C-measure μ can be realized by an LUT cascade consisting of cells with at most $q+1$ inputs and at most q outputs, where $q = \lceil \log_2 \mu \rceil$.*

A function with a small C-measure can be efficiently realized by an LUT cascade. Thus, the C-measure can be used to predict the **complexity** of the LUT cascade.

Theorem 3.3 *[11] Consider an LUT cascade that realizes a function f. Let n be the number of primary inputs; let s be the number of cells; let q be the maximum number of rails (i.e., the number of signal line between adjacent cells); let p be the maximum number of inputs to cells; and let μ be the C-measure of the function f. When $p \geq \lceil \log_2 \mu \rceil + 1$,*

there exists an LUT cascade for f that satisfies the following relation:

$$s \leq \left\lceil \frac{n-q}{p-q} \right\rceil$$

An address generation function with weight k is realized by an LUT cascade with $\lceil \log_2(k+1) \rceil$ rails. However, when the value of k is large, the single LUT cascade requires cells with many inputs, and this results in a large circuit. For example, when $k = 40{,}000$, a cascade requires cells with 17 inputs and 16 outputs. In such a case, we can partition the set of vectors into several groups, and realize each group by a separate LUT cascade to reduce the total amount of memory. Such a method requires a special encoder to combine the outputs of cascades[8].

4 Hash-Based Design

4.1 Basic Idea

For an address generation function $f(X_1, X_2)$ with weight k, we transform the input variable (X_1, X_2) into (Y_1, X_2) to hash the address space. Let $\hat{f}(Y_1, X_2)$ be the function after hashing, and consider its decomposition chart (Fig.4.1). Let p be the number of variables in Y_1. If the non-zero elements are uniformly distributed in the decomposition chart, then each column of the decomposition chart has at most one non-zero element when $2^p > k$. For simplicity, let us assume that each column of the decomposition chart has at most one non-zero element. Next, let

$$\hat{h}(Y_1) = \max_{\vec{b} \in B^{n_2}} \hat{f}(Y_1, \vec{b})$$

and realize $\hat{h}(Y_1)$ by a hash memory.

Note that the function $\hat{f}$ depends on (Y_1, X_2). On the other hand, the function $\hat{h}(Y_1)$ depends only on Y_1. Thus, the output of $\hat{h}(Y_1)$ may not be equal to $\hat{f}(Y_1, X_2)$. We will check the equality of the functions by the auxiliary (AUX) memory shown in Fig. 4.2. Note that the auxiliary memory has $q = \lceil \log_2(k+1) \rceil$ inputs and $r = n - p$ outputs. The auxiliary memory stores the X_2 part of the registered vector in the corresponding address. When the values of X_2 are equal to the values produced by the AUX memory, the output of the hash memory $\hat{f}(Y_1, X_2)$ is correct, and it is is sent to the OR gates. Otherwise, the hash memory produces an incorrect output, and the zero vector is sent to the OR gates. When a column has more than one non-zero element, we decompose $f(X_1, X_2)$ into two: $f(X_1, X_2) = \hat{f}_1(Y_1, X_2) \vee f_2(X_1, X_2)$, where $\hat{f}_1(Y_1, X_2)$ has at most one non-zero element in each column of the decomposition chart. On the other hand $f_2(X_1, X_2)$ is the remaining address generation function with smaller weights. Thus, $f_2(X_1, X_2)$ can be realized by a smaller LUT cascade.

The features of the realization method are as follows: The hash memory is efficient, but it realizes a limited class of address generation functions. On the other hand, the LUT cascade realizes any address generation function, but it is not so efficient when k is large. The hybrid method shown in Fig.4.2 combines these two methods to implement an address generation function efficiently.

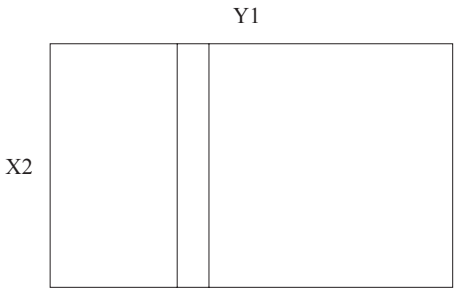

Figure 4.1: Decomposition chart for the hashed function $\hat{f}(Y_1, X_2)$.

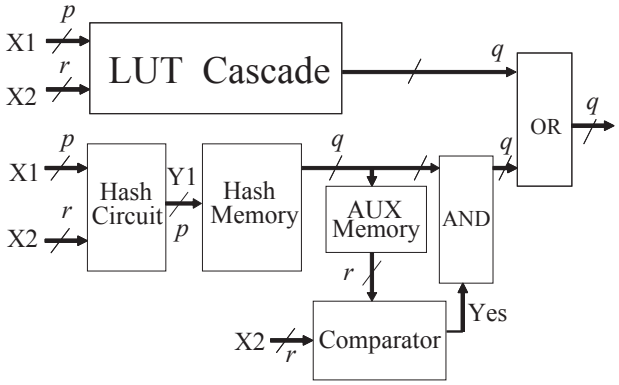

Figure 4.2: Realization of address generation function by the hybrid method.

The hybrid method is effective for the address generation function, where $k < 2^p$. In this method, the hash memory and the auxiliary memory realize about 90% of the non-zero elements, while the LUT cascade realizes the remaining 10% of non-zero elements. This fact is confirmed by the the theoretical analysis in Section 5, and the experimental results in Section 6. Note that in Fig. 4.2, the upper part (i.e., the LUT cascade) uses a disjoint decomposition, while the lower part, uses a **non-disjoint decomposition**.

4.2 A Method to Generate A Hash Function

A hash function is used to scatter the non-zero elements of the address generation function uniformly in the decomposition chart. In this paper, we use the following function $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$ and $x_j \in \{X_2\}$.

Generation of the hash function

In an address generation function $f(X_1, X_2)$, let $X_1 = (x_1, x_2, \ldots, x_p)$ be the bound variables, and let $X_2 =$

$(x_{p+1}, x_{p+2}, \ldots, x_n)$ be the free variable. Let $\hat{f}(Y_1, X_2)$ be the function which is obtained by replacing the bound variables X_1 with $Y_1 = (y_1, y_2, \ldots, y_p)$. Let w the number of columns that have at least one non-zero element. To store more registered vectors in the hash memory, we choose a Y_1 that maximizes w. We use the following heuristic:

Algorithm 4.1 *For each element x_i $(i = 1, 2, \ldots, p)$ in X_1, let $y_i = x_i \oplus x_j$. Select $x_j \in \{X_2\}$ that makes w maximum. Continue this operation while w increases.*

The hash function is obtained by using Y_1.

4.3 Design of Address Generator

For an address generation function $f(X_1, X_2)$ with weight k, let $\hat{f}(Y_1, X_2)$ be the function that is obtained by replacing the the bound variables $X_1 = (x_1, x_2, \ldots, x_p)$ with $(y_1 \oplus x_{j_1}, y_2 \oplus x_{j_2}, \ldots, y_p \oplus x_{j_p})$, where, $p > \lceil \log_2(k+1) \rceil$. For each $\vec{a} \in B^p$, when $\hat{f}(\vec{a}, X_2)$ has more than one non-zero output, replace the non-zero elements except for the minimum value by 0, to obtain the function $\hat{f}_1(Y_1, X_2)$. Next, let $\hat{f}_2(Y_1, X_2) = \hat{f}(Y_1, X_2) \oplus \hat{f}_1(Y_1, X_2)$. Since $\hat{f}_1(Y_1, X_2) \cdot \hat{f}_2(Y_1, X_2) = 0$, we have the relation: $\hat{f}(Y_1, X_2) = \hat{f}_1(Y_1, X_2) \vee \hat{f}_2(Y_1, X_2)$. Note that in the decomposition chart for $\hat{f}_1(Y_1, X_2)$, each column has at most one non-zero element. Next, let

$$\hat{h}(Y_1) = \max_{\vec{b} \in B^{n_2}} \hat{f}_1(Y_1, \vec{b}),$$

and realize $\hat{h}(Y_1)$ by the hash memory. Since the value of $\hat{h}(Y_1)$ can be different from the value of $\hat{f}_1(Y_1, X_2)$, we check if it is correct or not by using the auxiliary memory. Also, by transforming $x_i = y_i \oplus x_j$, we generate the function $f_2(X_1, X_2)$ from $\hat{f}_2(Y_1, X_2)$. Finally, realize $f_2(X_1, X_2)$ by an LUT cascade.

Example 4.1 *Table 4.1 is a decomposition chart of the 6 variable function $f(X_1, X_2)$ with weight $k = 7$ that appeared in Table 2.2. In this function, transform the bound variables $X_1 = (x_1, x_2, x_3)$ into $Y_1 = (y_1, y_2, y_3) = (x_1 \oplus x_6, x_2 \oplus x_5, x_3 \oplus x_4)$. The decomposition chart of the hashed function $\hat{f}(Y_1, X_2)$ is shown in Table 4.2. In the hashed function, the columns of the original decomposition tables are permutated. Also, each row has a different permutation. In the original table, three columns for $(x_1, x_2, x_3) = (0,0,0), (0,1,0), (0,0,1)$ have two non-zero elements. On the other hand, in the decomposition table in Table 4.2 for the hashed function $\hat{h}(Y_1, X_2)$, only one column $(y_1, y_2, y_3) = (0,1,0)$ has two non-zero elements. Let $\hat{f}_1(Y_1, X_2)$ be the function where the non-zero element 4 is replaced by 0. The decomposition chart is shown in Table 4.3. Table 4.4 shows the decomposition chart of the function $\hat{f}_2(Y_1, X_2)$ that is realized by the cascade. In this case, the function has only one non-zero element. $\hat{f}_1(Y_1, X_2)$ is implemented by the hash memory shown in Table 4.5 and the auxiliary memory shown in Table 4.6. The output of the hash memory $\hat{h}(Y_1)$ shows the non-zero value of the function $\hat{f}_1$ for the column $Y_1 = (y_1, y_2, y_3)$. The auxiliary memory shown in Table 4.6 decides if the output is zero or not. The function that is implemented by the LUT cascade has non-zero output 4. The corresponding input values are $(x_1, x_2, x_3, x_4, x_5, x_6) = (0,0,1,1,1,0)$. Fig. 4.3 shows the whole network for function f. The AUX memory and comparator check if (x_4, x_5, x_6) is the input that produces the non-zero output. The LUT cascade consists of AND gates. The non-zero output is 4, and its binary representation is $(1,0,0)$. This is implemented by ORing the most significant bit of the AND gates and the output of the cascade.*

(End of Example)

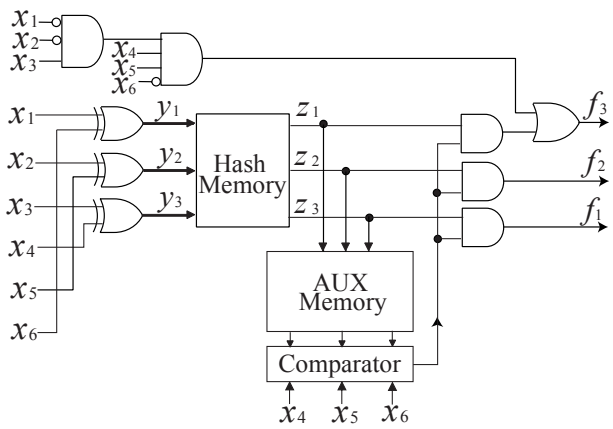

Figure 4.3: Realization of a 6-variable function by the hybrid method.

Table 4.1: Decomposition chart for $f(X_1, X_2)$.

	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
000	0	0	0	0	0	0	0	0	
001	0	0	0	0	0	0	0	0	
010	1	0	2	0	3	0	0	0	
011	0	0	0	0	4	0	0	0	
100	5	0	0	0	0	0	0	0	
101	0	0	0	0	0	0	0	0	
110	0	0	0	0	0	0	0	6	
111	0	0	7	0	0	0	0	0	
$x_6x_5x_4$									

Theorem 4.1 *When $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$, for $x_j \in \{X_2\}$, is used as the hash function, only the outputs for X_2 are necessary in the auxiliary memory and the comparator in Fig.4.2.*

Table 4.2: Decomposition chart for $\hat{f}(Y_1, X_2)$ (hashed function).

	0	0	0	0	1	1	1	1	y_3
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_1
000	0	0	0	0	0	0	0	0	
001	0	0	0	0	0	0	0	0	
010	2	0	1	0	0	0	3	0	
011	0	0	4	0	0	0	0	0	
100	0	5	0	0	0	0	0	0	
101	0	0	0	0	0	0	0	0	
110	0	0	0	0	6	0	0	0	
111	0	0	0	0	0	7	0	0	
$x_6x_5x_4$									

Table 4.3: Decomposition chart for $\hat{f}_1(Y_1, X_2)$.

	0	0	0	0	1	1	1	1	y_3
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_1
000	0	0	0	0	0	0	0	0	
001	0	0	0	0	0	0	0	0	
010	2	0	1	0	0	0	3	0	
011	0	0	0	0	0	0	0	0	
100	0	5	0	0	0	0	0	0	
101	0	0	0	0	0	0	0	0	
110	0	0	0	0	6	0	0	0	
111	0	0	0	0	0	7	0	0	
$x_6x_5x_4$									

(Proof) The hash memory realizes the function $\hat{h}(Y_1)$. The output value is derived using the outputs of the hash memory and the output of the auxiliary memory. In the decomposition chart of the hashed function, the row elements of the original decomposition chart are permuted. To verify the correctness of the outputs of the hash memory, we need

Table 4.4: Decomposition chart for $\hat{f}_2(Y_1, X_2)$

	0	0	0	0	1	1	1	1	y_3
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_1
000	0	0	0	0	0	0	0	0	
001	0	0	0	0	0	0	0	0	
010	0	0	0	0	0	0	0	0	
011	0	0	4	0	0	0	0	0	
100	0	0	0	0	0	0	0	0	
101	0	0	0	0	0	0	0	0	
110	0	0	0	0	0	0	0	0	
111	0	0	0	0	0	0	0	0	
$x_6x_5x_4$									

Table 4.5: Function $\hat{h}(Y_1)$ realized by the hash memory.

y_3	0	0	0	0	1	1	1	1
y_2	0	0	1	1	0	0	1	1
y_1	0	1	0	1	0	1	0	1
$\hat{h}(Y_1)$	2	5	1	0	6	7	3	0

Table 4.6: Contents of the auxiliary memory

$z_1z_2z_3$	x_4	x_5	x_6
000	0	0	0
001	0	1	0
010	0	1	0
011	0	1	0
100	0	0	0
101	0	0	1
110	0	1	1
111	1	1	1

only to check the value of X_2.

That is, if the output values of the auxiliary memory and the values of X_2 are the same, the values for X_1 are also the same, which is shown to be true by the relation $y_i = x_i \oplus x_j$. Thus, the outputs of the hash memory are equal to the outputs. If the output values of the auxiliary memory and the value of X_2 are different, the output value is 0. (Q.E.D.)

Example 4.2 *For the network shown in Fig. 4.3, we need to check only $X_2 = (x_4, x_5, x_6)$ in the output of the auxiliary memory to see if the input produces the non-zero output.*

(End of Example)

5 Number of Registered Vectors Realized by Hash Memory

In this part, we assume that the non-zero elements are uniformly distributed in the decomposition chart, and obtain the fraction of registered vectors realized by the hash memory.

Theorem 5.1 *Let f be an n-variable address generation function with weight k, and the non-zero elements be uniformly distributed in the decomposition chart. Then, the fraction of registered vectors realized by the hash memory shown in Fig. 4.2 is given by*

$$\delta \simeq 1 - \frac{1}{2}\left(\frac{k}{2^p}\right) + \frac{1}{6}\left(\frac{k}{2^p}\right)^2,$$

where $q = \lceil \log_2(k+1) \rceil$, and $p = |Y_1|$ denotes the number of bound variables in the decomposition chart for $f(Y_1, X_2)$, and $k < 2^p$.

(Proof) Let k be the total number of non-zero elements in the decomposition chart. Then, $\alpha = \frac{k}{2^n}$ denotes the fraction of the non-zero elements in the decomposition chart. Also, $\beta = 1 - \alpha$ denotes the fraction of the zero elements in the decomposition chart. And, $r = n - p = |X_2|$ denotes the number of free variables in the decomposition chart. In this case, we have the following:

1. The probability that a column has only 0 elements is β^{2^r}.
2. The probability that a column has at least one non-zero element is $1 - \beta^{2^r}$.

In total, there are 2^p columns, and the total number of non-zero elements is k. For the column with more than one non-zero element, the hash memory realizes only one non-zero element. In this case, the fraction of registered vectors realized by the hash memory is given by

$$\begin{aligned}\delta &= (1-\beta^{2^r})\cdot\frac{2^p}{k}\\ &= (1-(1-\alpha)^{2^r})\cdot\frac{2^p}{k}\\ &= (1-(\sum_{i=0}^{2^r}(-1)^i\binom{2^r}{i}\alpha^i))\cdot\frac{2^p}{k}\\ &= (\sum_{i=1}^{2^r}(-1)^{i+1}\binom{2^r}{i}\alpha^i)\cdot\frac{2^p}{k}.\end{aligned}$$

Note that when $2^p > k$, the absolute value of the each term of the above expression decreases with the increase of i. Next, by approximating δ by using first three terms of the above expression, we have

$$\begin{aligned}\delta &\simeq [2^r\alpha - \frac{2^{2r}\alpha^2}{2} + \frac{2^{3r}\alpha^3}{6}]\cdot\frac{2^p}{k}\\ &\simeq [2^r(\frac{k}{2^n}) - \frac{1}{2}2^{2r}(\frac{k}{2^n})^2 + \frac{1}{6}2^{3r}(\frac{k}{2^n})^3]\cdot\frac{2^p}{k}\\ &\simeq 1 - \frac{1}{2}(\frac{k}{2^p}) + \frac{1}{6}(\frac{k}{2^p})^2\end{aligned}$$

(Q.E.D.)

For example, when $\frac{k}{2^p} = \frac{1}{4}$, we have $\delta \simeq 0.8854$.

Example 5.1 *Consider the case of $n = 40$ and $k = 1730$ in Theorem 3.2. Since $q = \lceil \log_2(k+1) \rceil = \lceil \log_2(1730+1) \rceil = 11$, the number of bound variables is $p = 13$.*

1. **When the function is realized by an LUT cascade alone.**
   Let $p = 13$ be the number of inputs for cells. Then, from Theorem 3.3, the number of levels of the cascade is given by

$$\lceil \frac{n-q}{p-q} \rceil = \lceil \frac{40-11}{13-11} \rceil = \lceil \frac{29}{2} \rceil = 15.$$

   For each cell, the size of the memory is $2^p \times q = 2^{13} \times 11$ (bits). Thus, the total amount of memory is $2^{13} \times 11 \times 15 = 1,351,680$ (bits).

2. **When the function is realized by the hybrid method.**
   Since $k < 2^p$, the assumption for the approximation in Theorem 5.1 is valid. From Theorem 5.1, we have

$$\begin{aligned}\delta &\simeq 1 - \frac{1}{2}(\frac{k}{2^p}) + \frac{1}{6}(\frac{k}{2^p})^2\\ &= 1 - \frac{1}{2}(\frac{1730}{2^{13}}) + \frac{1}{6}(\frac{1730}{2^{13}})^2 \simeq 0.901.\end{aligned}$$

   The hash memory has $p = 13$ inputs and $q = 11$ outputs. The auxiliary memory has $q = 11$ inputs and $r = n - p = 27$ outputs. The LUT cascade realizes the address generation function with weight $1730 \times (1 - 0.901) = 170$. In this case, each cell in the cascade has $\lceil \log_2(170+1) \rceil = 8$ outputs. Let the number of inputs of cells be 10, then the number of levels in the LUT cascade is

$$\lceil \frac{n-q}{p-q} \rceil = \lceil \frac{40-8}{10-8} \rceil = \lceil \frac{32}{2} \rceil = 16$$

   Note that the size of a cell except for the last stage is $2^{10} \times 8$ (bits). The size of the cell in the last stage is $2^{10} \times 11$ (bits). Thus, the total amount of memory for the cascade is $2^{10} \times 8 \times 15 + 2^{10} \times 11 = 134,144$ (bits). The size of the hash memory is $2^{13} \times 11 = 90,112$ (bits). The size of the auxiliary memory is $2^{11} \times 27 = 55,296$ (bits). Thus, the total amount of memory is $279,552$ (bits), which is 20.7% of the total memory for the LUT cascade-only realization.

In this example, the hybrid method requires smaller amount of memory than the LUT cascade alone. (End of Example)

6 Experimental Results

6.1 Realization of English Word Lists

As for examples of address generators, we realized lists of frequently used English words. Here, we use three kinds of English word lists: List 1, List 2, and List 3. The number of letters in the each word is at most 13, but we only consider the first 8 letters. For the English words consisting of fewer than 8 letters, we append blanks to the end of words to make them 8-letter words. Each English alphabet letter is represented by 5 bits. Thus, each English word is represented by 40 bits. The number of words in the lists are 1730, 3366, and 4705, respectively. In each word list, each English word has a unique index, or an integer from 1 to k, where $k = 1730$ or 3360 or 4705. The numbers of bits for these indices are 11, 12, and 13, respectively.

Table 6.1: Realization of English word Lists.

	List 1	List 2	List 3
# of words: k	1730	3366	4705
# of inputs: n	40	40	40
# of outputs: q	11	12	13
# of inputs for the hash function: p	13	14	15
# of columns with only one non-zero element	1389	2752	4000
# of columns with two or more non-zero elements	165	293	342
# of registered vectors not realized by hash memory	176	321	363

Table 6.2: Memory sizes for English word lists.

Realization by an LUT Cascade alone				
		List 1	List 2	List 3
# of inputs	n	40	40	40
# of outputs	q	11	12	13
# of inputs of cells	p	13	14	15
# of levels	s	15	14	14
Total amount of memory (bits)	$sq2^p$	1,351,680	2,752,512	5,963,776
Realization by the hybrid method				
		List 1	List 2	List 3
Size of hash memory	$q2^p$	90,112	196,608	425,984
Size of auxiliary memory	$r2^q$	55,296	110,592	221,189
Size of cascade		134,144	301,056	304,104
Total amount of memory (bits)		279,552	608,256	951,277

Next, generate the function realized by the hash memory. The number of inputs for the hash function is $\lceil \log_2(k+1) \rceil + 2$.

List 1 consists of $k = 1730$ words. The number of bits for the index is $q = \lceil \log_2(1+k) \rceil = \lceil \log_2(1+1730) \rceil = 11$. The number of bound variables is $p = q+2 = 13$. The number of columns in the decomposition chart is $2^p = 2^{13} = 8192$. The number of columns that has only one non-zero element is 1389. The number of columns that has two or more non-zero elements is 165. The number of registered vectors that are not realized by the hash table is 176. In other words, about 90% of the registered vectors are realized by the hash memory, and the remaining 10% of the registered vectors are realized by the LUT cascade. Table 6.1 shows the experimental results for three English word lists.

Table 6.2 compares memory sizes. It shows that the hybrid method requires much smaller amount of memory than the LUT cascade alone.

Table 6.3: Realization of address generation functions produced by pseudo-random numbers (average of 100 functions).

	Function 1	Function 2	Function 3
# of words: k	1730	3366	4705
# of inputs: n	40	40	40
# of outputs: q	11	12	13
# of inputs for the hash function: p	13	14	15
# of columns with only one non-zero element	1398.4	2737.7	4075.1
# of columns with two or more non-zero elements	160.0	302.7	307.0
# of registered vectors not realized by hash memory	171.6	325.6	322.9
# of registered vectors not realized by hash memory (obtained by Theorem 5.1)	169.8	322.1	321.6

6.2 Realization of Randomly Generated Functions

Next, we generated address generations functions with the same sizes by pseudo-random numbers. Table 6.3 shows the average of 100 randomly generated functions. In this case, the average number of columns with only one non-zero element is 1398.4, the average number of columns with two or more non-zero elements is 160.0, and the average number of registered vectors not realized by the hash memory is 171.6. We did similar experiments for List 2 and List 3.

These results show that the experimental results using real word lists and the randomly generated functions do not have much difference with the theoretical results obtained in Section 5. This shows that the hash function generated by Algorithm 4.1 effectively scatters the non-zero elements in the decomposition charts.

7 Conclusions and Comments

In this paper, we presented a method to realize an address generation function using a hash memory and an LUT cascade. It uses both disjoint and non-disjoint functional decompositions. The basic idea is as follows: First decompose the address generation function f into two non-overlapping address generation functions: $f(X_1,X_2) = \hat{f}_1(Y_1,X_2) \vee f_2(X_1,X_2)$. In this case, $\hat{f}_1(Y_1,X_2)$ is decomposed as $\hat{f}_1(Y_1,X_2) = g(\hat{h}(Y_1),X_2)$, and each column of the decomposition chart has at most one non-zero element. The function g is realized by the comparator and the auxiliary memory. Since f_2 is also an address generation function having smaller weight than f_1, it can be implemented by

an LUT cascade, or further decomposed by using the same technique.

This method is only useful for the functions with $k << 2^n$. Unfortunately, for most of the MCNC benchmark functions, this method is not useful. In a typical MCNC benchmark function, the number of non-zero outputs k is not so small compared with the total number of the input combinations 2^n.

Recent FPGAs contain embedded memories in addition to the LUTs with 4 or 5 inputs. If we implement the address generation function by using only LUTs with 4 or 5 inputs, the resulting circuit will be too large to fit into the FPGA. On the other hand, the method presented in this paper uses the embedded memories in FPGAs, so we have a compact and fast circuit.

Acknowledgments

This research is supported in part by the Grants in Aid for Scientific Research of JSPS, and the grant of Kitakyushu Innovative Cluster Project. Discussion with Prof. Jon T. Butler improved English presentation. Mr. M. Matsuura did experiments.

References

[1] ALTERA, "Designing switches and routers with APPEX CAM," *White Paper*, Oct. 2000, Altera Corporation.

[2] H. A. Curtis, *A New Approach to the Design of Switching Circuits,* Van Nostrand, Princeton, N.J., 1962.

[3] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," ACM SIGCOMM'03, August 25-29, 2003, Karlsruhe, Germany.

[4] T. Kohonen, *Content-Addressable Memories*, Springer Series in Information Sciences, Vol. 1, Springer Berlin Heidelberg 1987.

[5] V. N. Kravets and K. A. Sakallah. "Constructive library-aware synthesis using symmetries", Proc. DATE '00, pp. 208-216, 2000.

[6] P-F. Lin and J. B. Kuo, "A 1-V 128-kb four-way set-associative CMOS cache memory using wordline-oriented tag-compare (WLOTC) structure with the content-addressable-memory (CAM) 10-transistor tag cell," *IEEE Journal of Solid-State Circuits*, Vol. 36, pp. 666 - 675, April 2001.

[7] J. C. Moran, "Memory patching circuit," US Patent 4028678.

[8] H. Qin and T. Sasao, "Design of address generators using multiple LUT cascade on FPGA," SASIMI 2006, April 2006, pp.146-152.

[9] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *International Workshop on Logic and Synthesis (IWLS01),* Lake Tahoe, CA, June 12-15, 2001, pp. 225-230.

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

[11] T. Sasao, Y. Iguchi, M. Matsuura, "LUT cascades and emulators for realizations of logic functions," RM2005, Tokyo, Japan, Sept. 5 - Sept. 6, 2005, pp.63-70.

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

[13] T. Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by LUT cascades," International Symposium on Multiple-Valued Logic, Singapore, May 17-20, 2006 (accepted).

[14] H. Vandierendonck and K. D. Bosschere, "XOR-Based hash functions," *IEEE Transactions on Computers*, Vol. 54, No. 7, July 2005, pp.800-812.