An Implementation of an Address Generator Using Hash Memories

Tsutomu Sasao and Munehiro Matsuura
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 *for the input that matches to one of* k *registered vectors, and produces 0 for other inputs. This paper presents the super hybrid method to design an address generator. The hash memories realize about* 96% *of the registered vectors, while the reconfigurable PLA realizes the remaining* 4% *of the registered vectors. With the super hybrid method, we can implement up to 20 times more registered vectors than the conventional method that uses only logic elements of an FPGA. Experimental results using lists of English words show that the usefulness of the approach.*

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. Address generation functions are used in the IP filtering in the internet, pattern matching, memory patching circuits, etc. Address generators often need to be reconfigured dynamically. Also, the functions are often random. Thus, conventional design methods are unsuitable for the design of address generators.

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 programmable logic array (PLA). Unfortunately, this method often requires excessive number of logic elements when it is implemented by an FPGA.

In this paper, we present, the **super hybrid method**, an efficient method to implement an address generation function using hash memories and a reconfigurable PLA.

This method is particularly suitable for FPGAs where both logic elements and embedded memories are available. In this method, hash memories implement about 96% of the vectors, while the reconfigurable PLA implements the remaining 4% of the vectors. Theoretical analysis 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. These vectors are* **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 produces a 3-bit number (e.g., 001) to the integer of the matched vector. When no entry matches to the input vector, the function produces* 000. *(End of Example)*

3. Reconfigurable PLA

An address generation function can be directly implemented by a PLA. In a *reconfigurable* PLA, we can change the logic data during the operation. The reconfigurable PLA is quite similar to a reconfigurable content addressable

Table 2.1. Registered vector table

Address	Vector
1	0010
2	0111
3	1101
4	0101
5	0011
6	1011
7	0001

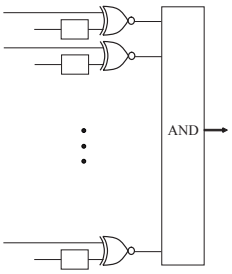

Figure 3.1. Match circuit by register and gates.

memory (CAM)[2, 6]. Various method exist to implement reconfigurable PLAs or CAMs. The **register and gates** approach uses a register to store the value of each bit. Fig. 3.1 shows a match circuit. A PLA or a CAM can be implemented by adding an encoder consisting of OR gates. With this approach, words of any width can be configured, and a fast reconfiguration is possible. Suppose that the reconfigurable PLA is implemented by Altera Cyclone II FPGAs. When the output part is fixed, to implement an n-input q-output and k-vector PLA, we need

$$(\lceil\frac{n}{2}\rceil + \lceil\frac{2n-1}{3}\rceil)k + \lceil\frac{\frac{k}{2}-1}{3}\rceil q \qquad (3.1)$$

LEs [1].

This formula was obtained by designing many reconfigurable PLAs of various sizes on a Cyclone II FPGA. Note that the first term is related to the registers, the second term is related to the comparators and AND gates, and the last term is related to the encoder. This implies that we need approximately $\frac{7}{6}nk$ LEs. [2] For example, when $n = 40$ and $k = 1730$, we have $q = \lceil\log_2(1730+1)\rceil = 11$. Thus, the number of LEs is

$$(20 + \lceil\frac{80-1}{3}\rceil) \cdot 1730 + 288 \times 11q = 84,478.$$

Table 3.1 shows the number os LE's and M4Ks of Altera Cyclone II FPGA. This shows that the reconfigurable PLA approach requires more LEs than available, since the FPGAs contain at most 68,416 LEs. However, the FPGAs contains many embedded memories (M4Ks) in addition to the LEs. In the next section, we will show the method to utilize these embedded memories.

[1] In Altera device [1], the LE (logic element) denotes the basic building block consisting of a 4-input look-up table (LUT), a register and an additional carry and cascade logic. In Xilinx device, it corresponds to the LB (logic block), which consists of a 4-input LUT, a register, and a carry logic.

[2] If we use the LUTs of a Xilinx FPGA, we can implement the address generators more efficiently. This requires SRL16E macro [15].

Table 3.1. Altera Cyclone II FPGA.

	LEs	M4Ks
EP2C5	4,608	26
EP2C8	8,256	36
EP2C20	18,752	52
EP2C35	33,216	105
EP2C50	50,528	129
EP2C70	68,416	250

4. Hash-Based Design

From here, we are going to study a method to implement an address generation function using memories.

Before explaining the super hybrid method, we introduce the hybrid method, a simpler version of the super hybrid method.

4.1. Hybrid Method

In the address generation function, the number of registered vectors k, is much smaller than 2^n, the total number of the input combinations. Consider the set of linear hash functions that maps 2^n elements into 2^p elements, where $2^p \geq k+1$. By using linear hash functions $y_i = x_i \oplus g_i(X_2)$, $(i = 1, 2, \ldots, p)$, we can reduce the 2^n-element space into a 2^p-element space. With this, we can implement the address generation function by using a p-input memory instead of an n-input memory.

Unfortunately, *collisions* of data occur. That is, two or more registered vectors are mapped into the same element. In such cases, we implement only one registered vector by the hash memory, and other registered vectors are implemented by other circuit.

Let $f(X_1, X_2)$ be the given address generation function. We can decompose it into $f(X_1, X_2) = \hat{f}_1(Y_1, X_2) \vee f_2(X_1, X_2)$. As shown in Fig. 4.1, $\hat{f}_1(Y_1, X_2)$ is implemented by the hash network, the p-input hash memory,

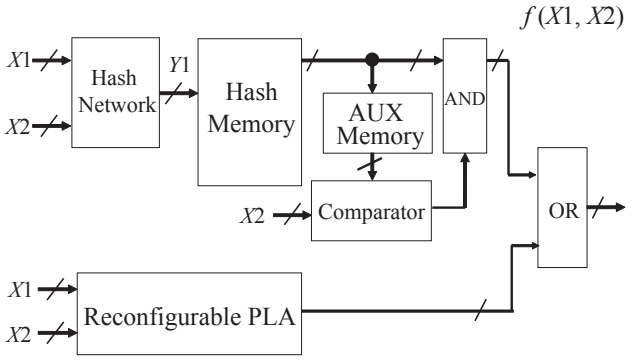

Figure 4.1. Address generator using hybrid method.

the AUX memory, and the comparator, while $f_2(X_1, X_2)$ is implemented by the reconfigurable PLA. In the hybrid method, we implement about 90% of the registered vectors by the hash memory. Since the 2^n-element space is reduced into the 2^p-element space by a set of linear functions, each output combination of the hash memory corresponds to 2^{n-p} input combinations.

1. When all the 2^{n-p} input combinations are non-registered, the hash memory stores zero for that input.
2. When only one combination is registered, and other $2^{n-p} - 1$ combinations are non-registered, the hash memory stores the index of the registered vector.
3. If two or more input combinations are registered, the hash memory stores an index of only one registered vector.

Thus, when the output of the hash memory is non-zero, the input vector can be registered or no-registered. To decide whether it is registered or not, we use the AUX memory. The AUX memory has q inputs and $(n - p)$ outputs. It stores the values of X_2 for each registered vector. If the input X_2 is equal to the output of the AUX memory, then the hash memory produces the correct output. Otherwise, the output of the hash memory is wrong, so 0 is sent to the output. In this way, $\hat{f}_1(Y_1, X_2)$ is implemented by the hash memory, the AUX memory, and the comparator.

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 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 $x_j \in \{X_2\}$.

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 $X_1 = (x_1, x_2, \ldots, x_p)$ with $(y_1 \oplus x_{j_1}, y_2 \oplus x_{j_2}, \ldots, y_p \oplus x_{j_p})$, where, $p \geq \lceil \log_2(k+1) \rceil$. For each $\vec{a} \in B^p$, where $B = \{0, 1\}$, 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 $\hat{f}_2(Y_1, X_2)$ be the function that shows the remaining non-zero elements. 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 the function $\hat{f}_1(Y_1, X_2)$ takes non-zero value for at most one non-zero element for each Y_1. 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, where n_2 denotes the number of variables in X_2. 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 AUX 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 a reconfigurable PLA.

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 AUX memory and the comparator in Fig.4.1.*

Example 4.1 *Table 4.1 is a decomposition chart of a 6 variable function $f(X_1, X_2)$ with weight $k = 7$. In this function, transform the 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 table 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 truth 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 reconfigurable PLA. 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 AUX memory shown in Table 4.6. The output of the hash memory $\hat{f}(Y_1)$ shows the non-zero value of the function $\hat{f}_1$ for the column $Y_1 = (y_1, y_2, y_3)$. The AUX memory*

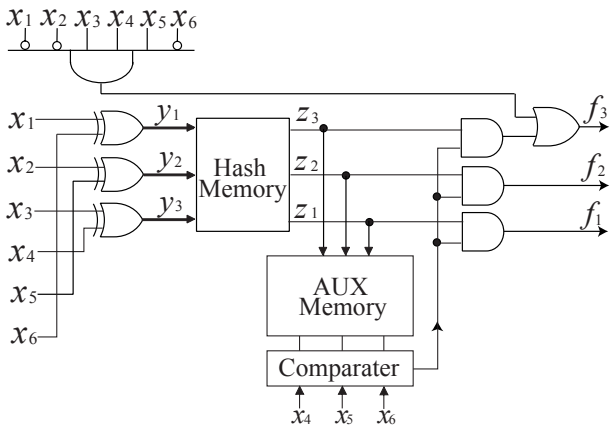

Figure 4.2. Realization of a 6-variable function by a hybrid method.

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	
$x_6x_5x_4$									

shown in Table 4.6 decides if the output is zero or not. The function that is implemented by the reconfigurable PLA 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. 4.2 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 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. (End of Example)*

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	
$x_6x_5x_4$									

5. Numbers of Registered Vectors Realized by Hash Memory

In this part, we assume that the non-zero elements in the address generation function are uniformly distributed in the decomposition chart. In this case, we can estimate 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. 4.1 is given by*

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

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

where $p = |Y_1|$ denotes the number of bound variables in the decomposition chart for $f(Y_1, X_2)$, and $k < 2^p$.

For example, when $\frac{k}{2^p} = \frac{1}{4}$, we have $\delta \simeq 0.8854$, and when $\frac{k}{2^p} = \frac{1}{2}$, we have $\delta \simeq 0.792$.

6. Super Hybrid Method

6.1. Principle

In the hybrid method, about 90% of the registered vectors are implemented by the hash memory and the remaining 10% of the registered vectors are implemented by the PLA. When we use two hash memories, we can implement about 96% of the registered vectors, and the remaining 4% of the registered vectors are implemented by the PLA. Such implementation is called **super hybrid method**. The super hybrid method shown in Fig. 6.1 is more complicated than the hybrid method, but requires smaller memories.

Table 4.6. Contents of the AUX memory

$z_3z_2z_1$	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

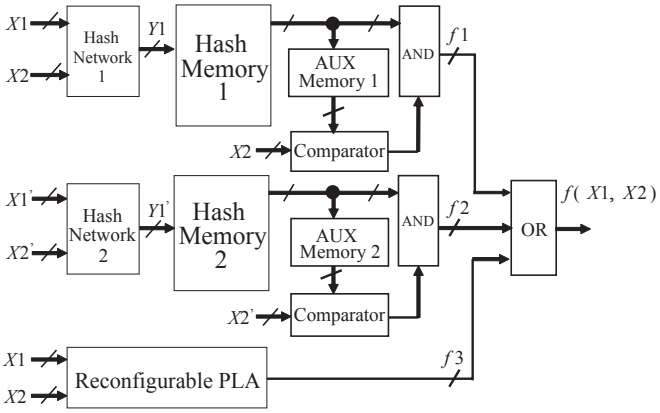

Figure 6.1. Address generator using super hybrid method.

Hybrid Method

The hash memory has $(q + 2)$ inputs and q outputs. The AUX memory has q inputs and $(n - q - 2)$ outputs. Therefore, the total amount of memory is $q \cdot 2^{q+2} + (n - q - 2) \cdot 2^q = (4n + 12q - 8) \cdot 2^{q-2}$.

Super Hybrid Method

The first hash memory has $(q+1)$ inputs and q outputs. The first AUX memory has q inputs and $(n-q-1)$ outputs. The second hash memory has $(q-1)$ inputs and $(q-2)$ outputs. The second AUX memory has $(q-2)$ inputs and $(n-q+2)$ outputs.

Therefore, the total amount of memory is $q \cdot 2^{q+1} + (n - q - 1) \cdot 2^q + (q - 2) \cdot 2^{q-1} + (n - q + 2) \cdot 2^{q-2} = (5n + 5q - 6) \cdot 2^{q-2}$.

This implies that when $n \leq 7 \log_2(k + 1) - 2$, the super hybrid method requires smaller amount of memory.

Theorem 6.1 *By using the hybrid method shown in Fig. 4.1, we can implement about 90% of the registered vectors by the hash and AUX memories. By using the super hybrid method shown in Fig. 6.1, we can implement about 96% of the registered vectors by the hash and AUX memories.*

6.2. Example

Example 6.1 *Consider the case of $n = 40$ and $k = 1730$. In this case, $q = \lceil \log_2(k + 1) \rceil = \lceil \log_2(1730 + 1) \rceil = 11$.*

Reconfigurable PLA.

The number of vectors realized by the reconfigurable PLA is

1730. From equation (3.1), the number of LEs to implement the reconfigurable PLA is 84,478.

Hybrid Method.

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 AUX memory has $q = 11$ inputs and $r = n - p = 27$ outputs. The size of the hash memory is $2^{13} \times 11 = 90,112$ (bits). The size of the AUX memory is $2^{11} \times 27 = 55,296$ (bits). Thus, the total amount of memory is $145,408$ (bits). The number of vectors realized by the reconfigurable PLA is 173.

Super Hybrid Method.

From Theorem 5.1, we have

$$\begin{aligned}\delta &\simeq 1-\frac{1}{2}(\frac{k}{2^p})+\frac{1}{6}(\frac{k}{2^p})^2 \\ &= 1-\frac{1}{2}(\frac{1730}{2^{12}})+\frac{1}{6}(\frac{1730}{2^{12}})^2 \simeq 0.8185\end{aligned}$$

The first hash memory has $p_1 = 12$ inputs and $q_1 = 11$ outputs. The first AUX memory has $q_1 = 11$ inputs and $r_1 = n - p_1 = 27$ outputs. The second hash memory has $p_2 = 10$ inputs and $q_2 = 9$ outputs. The second AUX memory has $q_2 = 9$ inputs and $r_2 = n - p_2 = 30$ outputs. The size of the first hash memory is $2^{12} \times 11 = 45,056$ (bits). The size of the first AUX memory is $2^{11} \times 28 = 57,344$ (bits). The size of the second hash memory is $2^{10} \times 9 = 9,216$ (bits). The size of the second AUX memory is $2^9 \times 30 = 15,360$ (bits). Thus, the total amount of memory is $126,976$ (bits). The number of vectors realized by the reconfigurable PLA is 43.

Thus, for this problem, the super hybrid method requires smaller amount of hardware. (End of Example)

A problem in the super hybrid method is that the second hash memory has only $q - 2$ outputs. Thus, the indices of the registered vectors in the second hash memory should be smaller than or equal to $2^{q-2} - 1$. The first hash memory stores registered vectors whose indices are greater than 2^{q-2}.

7. Experimental Results

7.1. List of English Words

To demonstrate the usefulness of the design method, first 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 numbers of letters in the word lists are 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, an integer from 1 to k, where $k = 1730$ or 3360 or 4705. The numbers of bits for the indices are 11, 12, and 13, respectively.

Table 7.1. Realization of English word Lists by hybrid method.

	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	3980
# of columns with two or more non-zero elements	165	293	351
# of registered vectors not realized by hash memory	176	321	374

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 reconfigurable PLA. Table 7.1 shows the design results for three English word lists by the hybrid method.

Table 7.2 compares the amount of hardware for reconfigurable PLA, the hybrid method, and the super hybrid method. It shows that the super hybrid method efficiently uses both LEs and M4Ks of the FPGA. In the super hybrid method, the number of vectors realized by the reconfigurable PLA is smaller than 4% of the registered vectors. This is because we optimized hash functions.

7.2. Randomly Generated Functions

Next, we generated address generation functions with the same sizes by pseudo-random numbers. We did the similar experiments for List 2 and List 3. The experimen-

tal results using randomly generated functions and English word lists do not have much difference with the theoretical results obtained in Chapter 5. This shows that the hash function generated by the hash network effectively scatters the non-zero elements in the decomposition charts.

7.3. IP Address Table

To verify the effectiveness of the method, we also used IP addresses of computers that accessed our web cite in a certain period. Table 1 contains 1730 addresses, Table 2 contains 3366 addresses, and Table 3 contains 4588 addresses. The number of inputs are all 32, but the number of outputs for Table 1, Table 2, and Table 3 are 11,12, and 13, respectively. Also, in this case, there were not much differences among the experimental data using real address tables, the data obtained from the random address tables, and the data obtained by analytical results in Chapter 5. (Experimental reuslts are omitted.)

8. Conclusions and Comments

In this paper, we presented the super hybrid method to realize an address generation function. In the super hybrid method, the address generation function f is decomposed into three non-overlapping address generation functions: $f(X_1, X_2) = \hat{f}_1(Y_1, X_2) \vee \hat{f}_2(Y_1', X_2) \vee f_3(X_1, X_2)$. In this case, $\hat{f}_1(Y_1, X_2)$, $\hat{f}_2(Y_1', X_2)$, and $f_3(X_1, X_2)$ represent about, 80%, 16 %, and 4% of the registered vectors, respectively. The functions $\hat{f}_1(Y_1, X_2)$ and $\hat{f}_2(Y_1, X_2)$ are implemented by hash memories, AUX memories and comparators, while f_3 is implemented by a reconfigurable PLA. With the super hybrid method, we can implement up to 20 times more vectors than the conventional method that uses only LEs of an FPGA.

Implementations of logic functions with LUTs and embedded memories have been also developed [3, 7, 14]. However, these methods produce dedicated circuits for given functions. Thus, even a slight change of a logic function will produce a circuit with different structure. However, the presented method produces a circuit where a slight change of a logic function can be allowed by the modification of the contents of the memories.

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.

Table 7.2. Amount of hardware for English word lists.

Size of Lists				
		List 1	List 2	List 3
# of inputs	n	40	40	40
# of outputs	q	11	12	13
# of vectors	k	1730	3366	4705
Reconfigurable PLA				
		List 1	List 2	List 3
# of LEs		84,478	164,934	231,340
Hybrid Method				
		List 1	List 2	List 3
# of inputs for hash memory	p	13	14	15
Size of hash memory	$q2^p$	90,112	196,608	425,984
Size of AUX memory	$r2^q$	55,296	106,496	204,800
Total amount of memory (bits)		145,408	303,104	630,784
# of vectors realized by reconfigurable PLA		176	321	374
# of M4Ks		36	74	154
# of LEs		13,278	24,888	29,892
Super Hybrid Method				
		List 1	List 2	List 3
# of inputs for hash memory 1	p_1	12	13	14
# of inputs for hash memory 2	p_2	10	11	12
Size of hash memory 1	$q_1 2^{p_1}$	45,056	98,304	212,992
Size of AUX memory 1	$r_1 2^{q_1}$	57,344	110,592	212,992
Size of hash memory 2	$q_2 2^{p_2}$	9,216	20,480	40,960
Size of AUX memory 2	$r_2 2^{q_2}$	15,360	2,969	28,672
Total amount of memory (bits)		126,976	232,345	495,616
# of vectors realized by reconfigurable PLA		30	61	42
# of M4Ks		32	64	121
# of LEs		2,426	4,889	3,560

References

[1] http://www.altera.com

[2] ALTERA, "Implementing high-speed search applications with Altera CAM," *Application Note 119*, Altera Corporation, July 2001.

[3] J. Cong and K. Yan, "Synthesis for FPGAs with embedded memory blocks", In *Proc. of the 2000 ACM/SIGDA 8th International Symposium on Field Programmable Gate Arrays*, pp. 75-82, ACM Press NY, 2000, Monterey, California.

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

[5] J. Ditmar, K. Torkelsson, and A. Jantsch, "A reconfigurable FPGA-based content addressable memory for internet protocol characterization," *Proc. FPL2000*, LNCS 1896, Springer, 2000, pp. 19-28.

[6] S. Guccione, D. Levi, and D. Downs,"A reconfigurable content addressable memory," *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Volume 1800,May 2000. Parallel and Distributed Processing, p.882.

[7] S. Krishnamoorthy and R. Tessier, "Technology mapping algorithms for hybrid FPGAs containing lookup tables and PLAs", In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22, No. 5, 2003, pp. 545-559.

[8] K. McLaughlin, N. O'Connor, and S. Sezer, "Exploring CAM design for network processing using FPGA technology," *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services* (AICT-ICIW'06), p. 84, 19-25, Feb. 2006.

[9] G. Nilsen, J. Torresen and O. Sorasen, "A variable word-width content addressable memory for fast string matching," *NorChip 2004*, pp. 214- 217.

[10] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712-727, March 2006.

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

[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.

[14] S. J. E. Wilton, "SMAP: Heterogeneous technology mapping for FPGAs with embedded memory arrays," In *Proc. o f the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 171-178, 1998.

[15] Xilinx,"Designing flexible, fast CAMs with Virtex family FPGAs," *Application Note*, XAPP203, Sept. 23, 1999.