Index Generation Functions: Theory and Applications

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

***Abstract*—This survey first introduces index generation functions, which are useful for pattern matching in the communication circuit. Then, it shows various methods to realize index generation functions by using LUTs and memories. These methods are useful to design FPGAs with embedded memories.**

I. Index Generation Function

This paper surveys new methods to design memory-based pattern matching circuits [1], [3], [4], [5], [6]. Due to the page limitation, all the proofs are omitted.

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

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

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

II. Applications

Index generators are used in address table in the internet, terminal access controller for local area networks, databases, memory patch circuits, dictionaries, password lists, etc. [3].

A. Address Table in the Internet

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

TABLE 1.1
Registered vector table.

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

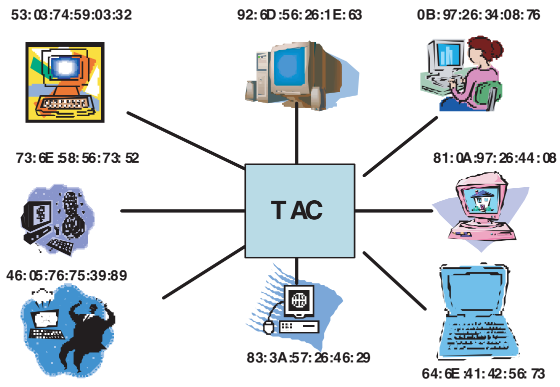


Fig. 2.1. Terminal access controller.

B. Terminal Access Controller

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

Example 2.1: Fig. 2.2 shows an example of the terminal access controller. The first terminal has the MAC address 53:03:74:59:03:02. It is allowed to access everything, including is the Web outside the local area network, e-mail, FTP, and Telnet. The second one is allowed to access both the Web and e-mail. The third one is allowed to access only the Web. And, the last one is allowed to access only e-mail. The index

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

$\Rightarrow$

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

48-bit 4-bit

Fig. 2.2. Index generator for terminal access controller.

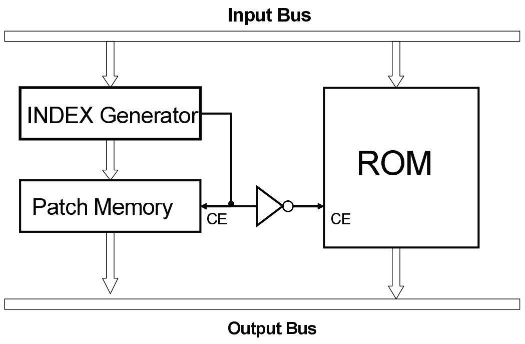


Fig. 2.3. Memory patch circuit.

generated by the index generator is used as an address to read the memory which stores the permissions. If we implement the TAC by a single memory, we need a memory with 256 Tera words, since the number of inputs is 48. To reduce the size of memory, we use an index generator to produce the index, and an additional memory to store the permission data for each internal address. ■

C. Memory Patch Circuit

The firmware of an embedded system is usually implemented by Read-Only Memories (ROMs). After shipping the product, it is often necessary to modify a part of the ROM, for example to upgrade to a later version. To convert the address of the ROM to the address of the patch memory, we use the index generator shown in Fig. 2.3.

The index generator stores addresses (vectors) of the ROM to be updated, and their corresponding indexes. A **patch memory** stores the updated data of the ROM. When the address does not match any elements in the index generator, the output of the ROM is sent to the output bus. In this case, the output the patch memory is disabled. When the address matches to an element in the index generator, the index generator produces the corresponding index, and the corresponding data of the patch memory is sent to the output bus. In this case, the output of the ROM is disabled. This method can be also used to improve the yield of large-scale memory, which can "patched" instead of discarded.

III. Properties of Index Generation Functions

The index generators in Section II have common properties:

1) The values of the non-zero outputs are distinct.

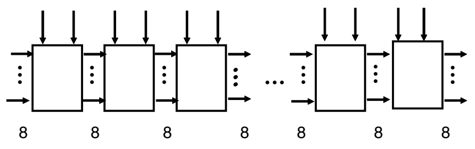


Fig. 4.1. LUT cascade realization of index generator.

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

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

IV. Realization using (p, q)-elements

Lemma 4.1: Let F be an index generation function with weight k. Then, there exists a functional decomposition [2]

$$F(X_1, X_2) = G(H(X_1), X_2),$$

where G and H are index generation functions, and the weight of G is k, and the weight of H is at most k.

Definition 4.1: A $(\boldsymbol{p}, \boldsymbol{q})$**-element** realizes an arbitrary p-input q-output logic function. Its **memory size is** $q2^p$.

Theorem 4.1: An arbitrary two-valued input n-variable index generation function with weight k can be realized as a multi-level network of (p, q)-elements. The number of such elements is at most $\lceil \frac{n-q}{p-q} \rceil$, where $p > q$ and $q = \lceil \log_2(k+1) \rceil$. We can generate various multi-level logic networks, including cascades.

Example 4.1: Let us design index generator where $n = 48$ and $k = 255$. Let $p = q + 2$ and $q = \lceil \log_2(255 + 1) \rceil = 8$. For each (p, q)-element, we can reduce the number of input lines by two. So, by using 20 (p, q)-elements, we can reduce the number of inputs into 8. For example, we have the LUT cascade as shown in Fig. 4.1. Or, we have the multi-level logic network shown in Fig. 4.2, where the number of levels is 10. In this case, the variables are permutated during functional decompositions. Note that both structures require the same amount of memory: 160k bits. ■

When the weight k of the function satisfies the relation $\lceil \log_2(k + 1) \rceil < K$, an index generation function can be realized by a cascade of K-input cells as well as a multi-level network of (K, q)-elements, or by a multi-level network with K-LUTs. However, when $k > 63$, such methods are inapplicable in most FPGAs, in the current FPGAs, the number of inputs to an LUT is at most 6. That is $K \leq 6$. In the next section, we show a method to use embedded memories of FPGAs.

V. Index Generation Unit (IGU)

Fig. 5.1 shows the **Index Generation Unit** (IGU). The **programmable hash circuit** has n inputs and p outputs. It is used to permute the non-zero elements. We consider two

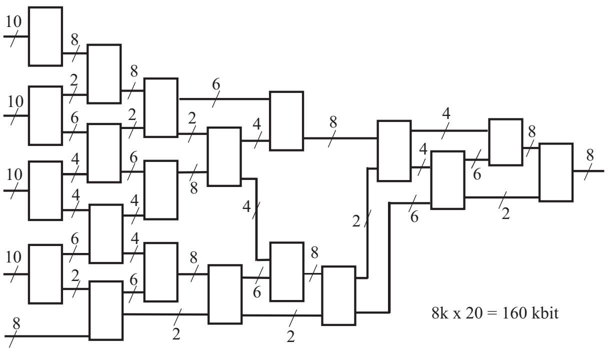


Fig. 4.2. Index generator ($p = 10$).

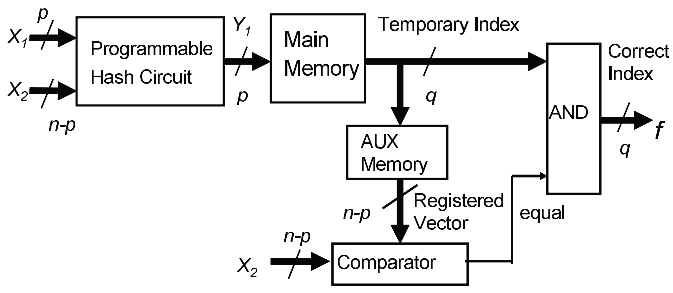
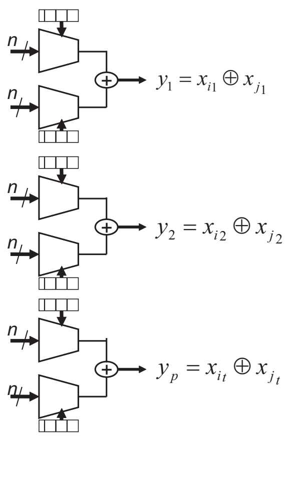


Fig. 5.1. Index generation unit (IGU).

types of programmable hash circuits. The first type is the **double-input hash circuit** shown in Fig. 5.2. It performs a **linear transformation** $y_i = x_i \oplus x_j$ or $y_i = x_i$, where $i \neq j$. 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 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 $y_i = x_i$. The second type of a programmable hash circuit is the **single-input hash circuit** shown in Fig. 5.3. It consists of only p multiplexers, and selects p variables from n input variables. Note that both types of hash circuits produce only specific kinds of hash functions. We have found that these functions are suitable for our application. The **main memory** has p inputs and $\lceil \log_2(k+1) \rceil$ outputs. The main memory produces correct outputs only for registered vectors. However, it may produce incorrect outputs for non-registered vectors, because the number of inputs to the main memory is reduced. In an index generation function, if the input vector is non-registered, then it should produce 0 outputs. To check whether the main memory produces the correct output or not, we use the **AUX memory**. The AUX memory has $\lceil \log_2(k+1) \rceil$ inputs and $n-p$ outputs: It stores the X_2 part of the registered vectors for each index. The **comparator** checks if the inputs are the same as the registered vector or not. If they are the same, the main memory produces a correct output. Otherwise, the main memory produces a wrong output, and the input vector is non-registered. Thus, the **output AND gates** produce 0 outputs, showing that the input vector is non-registered. Note that the main memory produces the correct outputs only for the registered vectors. A method to reduce the number of inputs to the main memory is considered in [5].



$y_1 = x_{i1} \oplus x_{j1}$
$y_2 = x_{i2} \oplus x_{j2}$
$y_p = x_{it} \oplus x_{jt}$

Fig. 5.2. Double-input hash circuit.

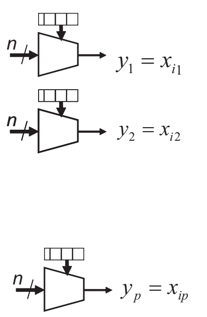


Fig. 5.3. Single-input hash circuit.

Example 5.1: Consider the registered vectors in Table 1.1. The number of variables is four, but only two variables x_1 and x_4 are necessary to distinguish these four registered vectors. Fig. 5.4 shows the IGU. In this case, the programmable hash circuit produces $Y_1 = (x_1, x_4)$ from $X = (x_1, x_2, x_3, x_4)$. The main memory stores the indices for $X_1 = Y_1 = (x_1, x_4)$, and the AUX memory stores the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector.

When the input vector is registered:

Suppose that a registered vector $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$ is applied to the IGU in Fig. 5.4. First, the programmable hash circuit selects two variables, x_1 and x_4, and produces the value $X_1 = (x_1, x_4) = (1, 0)$. Second, the main memory produces the corresponding index $(0, 1, 1)$. Third, the AUX memory produces the values of $X_2 = (x_2, x_3) = (1, 0)$ corresponding registered vector $(1, 1, 0, 0)$. Fourth, the comparator confirms that the values of $X_2 = (x_2, x_3)$ of the input vector are equal to the output of the AUX memory. And, finally, the AND gate produces the index for the input vector.

When the input vector is not registered:

Suppose that a non-registered vector $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$ is applied to the IGU in Fig. 5.5. Also in this case, the main memory produces the vector $(0, 1, 1)$, and the AUX memory produces the values of $X_2 = (x_2, x_3)$ for the corresponding registered vector $(1, 1, 0, 0)$. However, in this case, the comparator shows that $X_2 = (x_2, x_3) = (0, 1)$ is different from the output $X_2 = (x_2, x_3)$ of the AUX memory. Thus, the AND gate produces zero output, which shows that the input vector is not registered. ■

Consider the incompletely specified index generation function f^*, where the zero values of f are replaced by *don't cares*. The above example shows that in the main memory we can implement f^* instead of f. From the experimental results, we have the following:

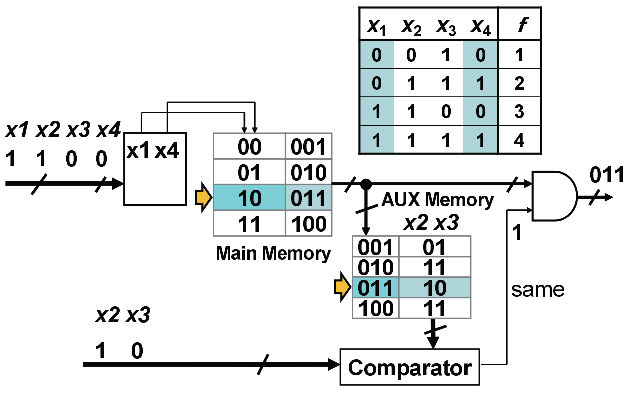


Fig. 5.4. When the input vector is registered.

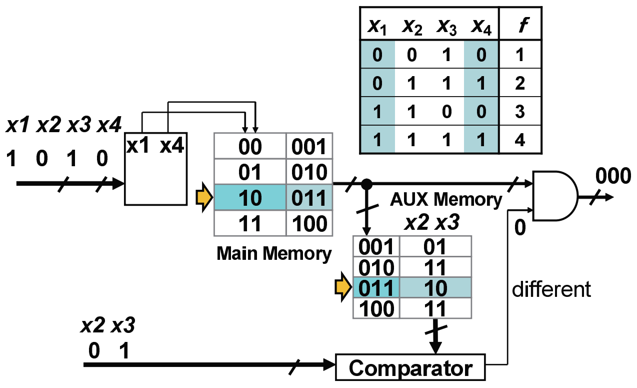


Fig. 5.5. When the input vector is not registered.

Conjecture 5.1: Consider a set of uniformly distributed incompletely specified index generation functions of n binary input variables with weight k. Then, the fraction of the functions represented with $p = 2\lceil \log_2(k+1) \rceil - 1$ variables approaches 1.0 as n increases.

Although there exist functions that require more than $p = 2\lceil \log_2(k+1) \rceil - 1$ variables, the fraction of such functions approaches 0.0 as n increase. When the value of k is large, the main memory with $p = 2\lceil \log_2(k+1) \rceil - 1$ inputs is too large to implement. We need more efficient methods. From the next section, we will consider such methods.

VI. Reduction by a Linear Transformation

As shown in Conjecture 5.1, most incompletely specified index generation functions with weight k can be represented by at most $p = 2\lceil \log_2(k+1) \rceil - 1$ variables. However, there exist functions that require more variables. In such a case, we can often reduce the number of variables by a linear transformation of the input variables. We have developed a heuristic algorithm [6] to find a linear transformation that reduces the number of variables, when the double-input hash circuit is used. To find a linear transformation, we use the following:

Theorem 6.1: Let $f(x_1, x_2, \ldots, x_n)$ be an index generation function. Let $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$ and $j \in \{p+1, p+2, \ldots, n\}$, and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$. Consider the transformed function $g(Y_1, X_2) = f(X_1, X_2)$. Then, f can be represented by using only Y_1, if each column of the decomposition chart (Y_1, X_2) has at most one non-zero element.

VII. Registered Vectors Realized by Main Memory

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

Theorem 7.1: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the expected number of registered vectors of f that can be realized by the IGU is $2^p(1-e^{-\xi})$, where $\xi = \frac{k}{2^p}$.

Corollary 7.1: Consider a set of uniformly distributed index generation functions $f(x_1, x_2, \ldots, x_n)$ with weight k. Consider an IGU whose inputs to the main memory are $x_1, x_2, \ldots,$ and x_p. Then, the fraction of registered vectors of f that can be realized by the IGU is

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

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

For example, when $\xi = \frac{1}{4}$, we have $\delta \simeq 0.8848$, when $\xi = \frac{1}{2}$, we have $\delta \simeq 0.7869$, and when $\xi = 1$, we have $\delta \simeq 0.63212$.

VIII. Efficient Methods

From here, we are going to show efficient methods to implement index generation functions using memories [4]. In an index generation function, the number of registered vectors k, is usually much smaller than 2^n, the total number of the input combinations.

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

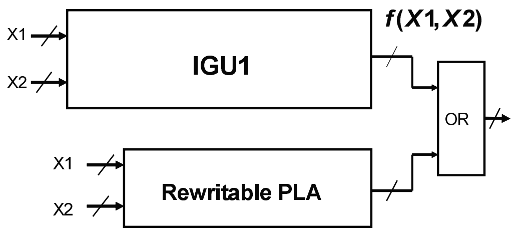


Fig. 8.1. Index generator implemented by hybrid method.

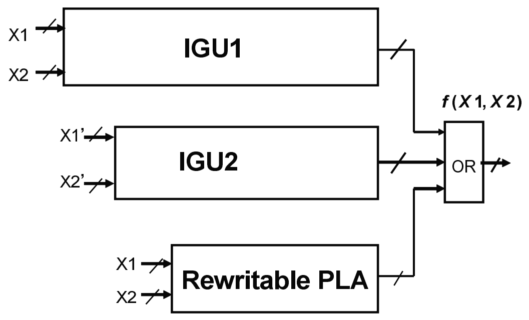


Fig. 8.2. Index generator implemented by super hybrid method.

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

Definition 8.2: The **super hybrid method** is an implementation of an index generation function using the circuit consisting of two IGUs as shown in Fig. 8.2. IGU_1 is used to realize most of the registered vectors, IGU_2 is used to realize the registered vectors not realized by IGU_1, and the rewritable PLA is used to realize registered vectors not realized by neither IGUs. The OR gate in the output combines the indices to form a single output. The rewritable PLA can be replaced by another circuit, such as an LUT cascade, a CAM or an IGU.

The super hybrid method shown in Fig. 8.2 is more complicated than the hybrid method, but requires smaller memories. In the super hybrid method, when the main memory of the IGU_1 has $p_1 = \lceil \log_2(k+1) \rceil + 1$ inputs, and the main memory of the IGU_2 has $p_2 = \lceil \log_2(k+1) \rceil - 1$ inputs, about 80% of the registered vectors are implemented by IGU_1, about 16% of the registered vectors are implemented by IGU_2, and and the remaining 4% of the registered vectors are implemented by the PLA. By increasing the number of IGU's, we have the parallel sieve method, which is especially useful when the number of the registered vectors is very large [1].

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

IX. Design Examples

This part shows various designs of index generation functions:

1) **Single LUT**.
When $n = 10$ and $k = 500$. The number of inputs for the memory is $n = 10$, and the number of outputs is $q = \lceil \log_2(500+1) \rceil = 9$. Thus, the size of the memory is $2^{10} \times 9 = 9$ kilobits.

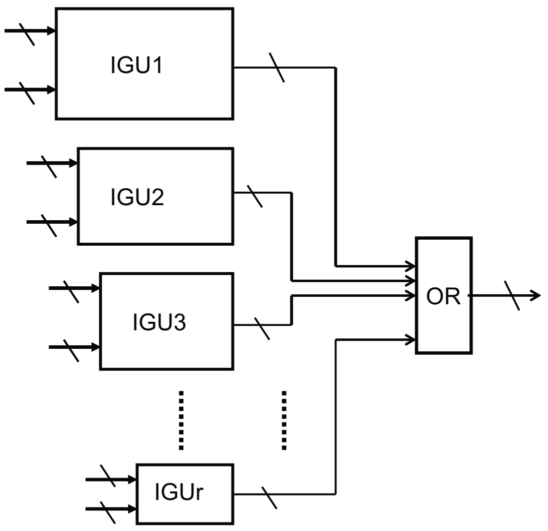


Fig. 8.3. Index generator implemented by parallel sieve method.

2) **LUT Cascade**.
When $n = 10$ and $k = 15$. Consider the LUT cascade with $(K = 6)$-input LUTs. The number of rails is $w = \lceil \log_2(15+1) \rceil = 4$, and the number of outputs is $m = w = 4$. The number of cells is

$$s = \lceil \frac{n-w}{K-w} \rceil = \lceil \frac{10-4}{6-4} \rceil = \frac{6}{2} = 3.$$

The total amount of memory is

$$2^6 \times 4 \times 3 = 3 \times 2^8 = 0.75 \times 2^{10}.$$

Thus, 0.75 kilobits.

3) **Hybrid Method**.
When $n = 48$, $k_1 = 100$. $q = \lceil \log_2(100+1) \rceil = 7$. In this case, the LUT cascade would be too large. So, we use the hybrid method. Let the number of inputs to the main memory be $p = q + 2 = 9$. In this case, by Corollary 7.1, the fraction of remaining registered vectors is

$$\gamma_1 = 1 - \delta_1 = \frac{\xi - 1 + e^{-\xi}}{\xi}$$

Since $p = 9$ and $k_1 = 100$, we have $\xi = 0.1953$, and

$$\gamma_1 = 1 - 0.9084 = 0.0916.$$

Thus, the number of remaining vectors is $\gamma_1 k_1 \simeq 9$, which can be implemented by an LUT cascade or a rewritable PLA.
The sizes of memories are as follows:
Main memory: 9-input, 7-outputs: 3.5 kilobits.
AUX memory: 7-input, 41-outputs: 5.1 kilobits.
Thus, the total memory size is 8.6 kilobits.

4) **Super Hybrid Method**.
When $n = 48$, $k_1 = 1000$. $q_1 = \lceil \log_2(1000+1) \rceil = 10$.

In the hybrid method, the remaining vector is 10% of the original vectors. That is, 100, which is fairly large. Thus, we use the super hybrid method. In the super hybrid method, we use the first main memory with $p_1 = q_1 + 1 = 11$ inputs, and $q_1 = 10$ outputs. The fraction of vectors not realized by the 1st IGU is

$$\gamma_1 = 1 - \delta_1 = \frac{\xi_1 - 1 + e^{-\xi_1}}{\xi_1}.$$

When, $k_1 = 1000$ and $p_1 = 11$, we have $\xi_1 = 0.48828$ and $\gamma_1 = 0.2088$. The number of remaining vectors is $k_2 = k_1\gamma_1 \simeq 209$. $q_2 = \lceil \log_2(209 + 1)\rceil = 8$.
The second main memory has $p_2 = q_2 + 1 = 8 + 1 = 9$ inputs and $q_2 = 8$ outputs. The fraction of vectors not realized by the 2nd IGU is

$$\gamma_2 = 1 - \delta_2 = \frac{\xi_2 - 1 + e^{-\xi_2}}{\xi_2}.$$

When, $k_2 = 209$ and $p_2 = 9$, we have $\xi_2 = 0.398437$, and $\gamma_2 \simeq 0.1752$. Thus, the number of remaining vectors is $k_3 = k_2\gamma_2 \simeq 36$, which can be implemented by an LUT cascade or a rewritable PLA.
The sizes of memories are as follows:
1st main memory: 11-input, 10-outputs: 20 kilobits.
1st AUX memory: 10-input, 37-outputs: 37 kilobits.
2nd main memory: 9-input, 8-outputs: 4 kilobits.
2nd AUX memory: 8-input, 39-outputs: 9.75 kilobits.
Thus, the total memory size is 70.75 kilobits.

5) **Standard Parallel Sieve Method**.
When $k_1 = 500,000$, we have $q_1 = \lceil \log_2(k_1 + 1)\rceil = 19$. In the super hybrid method, the remaining vector is 4% of the original vectors. That is, 20000, which is very large. Thus, we use the standard parallel sieve method. In this method, the first main memory has $p_1 = q_1 = 19$ inputs and $q_1 = 19$ outputs. The fraction of vectors not realized by the 1st IGU is

$$\gamma_1 = 1 - \delta_1 = \frac{\xi_1 - 1 + e^{-\xi_1}}{\xi_1}$$

When $k_1 = 500000$ and $p_1 = 19$, we have $\xi_1 = 0.953674$ and $\gamma_1 = 0.35546$. Thus, the number of remaining vectors is $k_2 = k_1\gamma_1 \simeq 177733$. and $q_2 = \lceil \log_2(177733 + 1)\rceil = 18$.
The second main memory has $p_2 = q_2 = 18$ inputs and $q_2 = 18$ outputs. The fraction of vectors not realized by the 2nd IGU is

$$\gamma_2 = 1 - \delta_2 = \frac{\xi_2 - 1 + e^{-\xi_2}}{\xi_2}$$

When, $k_2 = 177733$ and $p_2 = 18$, we have $\xi_2 = 0.6779976$ and the number of remaining vectors is $k_3 = k_2\gamma_2 \simeq 48662$.

In the similar way, we have

$$\begin{aligned} p_3 &= 16,\ k_4 \simeq 14316. \\ p_4 &= 14,\ k_5 \simeq 4771. \\ p_5 &= 13,\ k_6 \simeq 1155. \\ p_6 &= 11,\ k_7 \simeq 273. \\ p_7 &= 9,\quad k_8 \simeq 62. \end{aligned}$$

Thus, the number of remaining vectors is $k_8 \simeq 62$, which can be implemented by an IGU, an LUT cascade or a rewritable PLA.
For each IGU_i, the main memory has p_i inputs and p_i outputs, while the AUX memory has p_i inputs and $(n - p_i)$ outputs. Thus, the total amount of memory is

$$p_i 2^{p_i} + (n - p_i)2^{p_i} = n2^{p_i}.$$

So, the total amount of memory for the parallel sieve method is

$$\sum_{i=1}^{r} n2^{p_i} = 32 \cdot (2^{19} + 2^{18} + 2^{16} + 2^{14} + 2^{13} + 2^{11} + 2^{9}).$$

It is about 24 megabits.

X. Conclusions

In this paper, we introduced index generation functions, which have wide applications in pattern matching circuits for the Internet. We also presented various methods to implement index generation functions: Method using (p, q)-elements, method using an IGU, the hybrid method, the super-hybrid method and the parallel sieve method.

Acknowledgments

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

References

[1] H. Nakahara, T. Sasao, M. Matsuura, and Y. Kawamura, "A parallel sieve method for a virus scanning engine," *12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece (*DSD-2009*), Aug. 2009, pp.809-816.
[2] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.
[3] T. Sasao, "Design methods for multiple-valued input address generators,"(invited paper) *International Symposium on Multiple-Valued Logic* (ISMVL-2006), Singapore, May 2006.
[4] T. Sasao and M. Matsuura, "An implementation of an address generator using hash memories," *DSD-2007*, Aug. 27 - 31, 2007, Lubeck, Germany, pp.69-76.
[5] T. Sasao, "On the number of variables to represent sparse logic functions," *ICCAD-2008*, San Jose, California, USA, Nov.10-13, 2008, pp. 45-51.
[6] T. Sasao, T. Nakamura, and M. Matsuura, "Representation of incompletely specified index generation functions using minimal number of compound variables," *DSD-2009*, Aug. 2009, pp.765-772.