An Architecture for IPv6 Lookup Using Parallel Index Generation Units

Hiroki Nakahara[1], Tsutomu Sasao[2], and Munehiro Matsuura[2]

[1] Kagoshima University, Japan
[2] Kyushu Institute of Technology, Japan

Abstract. This paper shows an area-efficient and high-speed architecture for IPv6 lookup using a parallel index generation unit (IGU). To reduce the size of memory in the IGU, we use a liner transformation and a row-shift decomposition. Also, this paper shows a design method for the parallel IGU. A single memory realization requires $O(2^n)$ memory size, where n denotes the length of prefix, while the IGU requires $O(nk)$ memory size, where k denotes the number of prefixes. In IPv6 prefix lookup, since n is at most 64 and k is about 340 K, the IGU drastically reduces the memory size. Since the parallel IGU has a simple architecture compared with existing ones, it performs lookup by using complete pipelines. We loaded more than 340 K IPv6 pseudo prefixes on the Xilinx Virtex 6 FPGA. Its lookup speed is higher than one giga lookups per second (GLPS). As for the normalized area and lookup speed, our implementation outperforms existing FPGA implementations.

1 Introduction

1.1 Demands for Lookup Architecture in IPv6 era

The core routers forward packets by IP-lookup using **longest prefix matching (LPM)**. With the rapid growth of the Internet, LPM has become the bottleneck in the network traffic management. The following conditions must be satisfied to solve the problems:

High speed lookup: When a core router works at more than 40 Gbps link throughput (OC-768), it requires more than 125 million lookups per second (MLPS) for a minimum packet size (40 bytes). Now, a 100 Gbps link requires more than 320 MLPS, and the next generation router requires 400 Gbps link.

Low-power consumption: R. Tucker predicted that, with the rapid increase of traffic, core routers would dissipate the major part the total network power dissipation [14]. Thus, we cannot use power-hungry ternary content addressable memories (TCAMs). Le *et al.* proposed the memory-based IP lookup architecture on the FPGA, which dissipate lower power than the TCAM [5]. This paper also considers a method that uses a memory-based architecture.

Reconfigurability: On Feb. 3, 2011, IPv4 addresses maintained by Internet Assigned Numbers Authority (IANA) are depleted. Since transition from IPv4 addresses

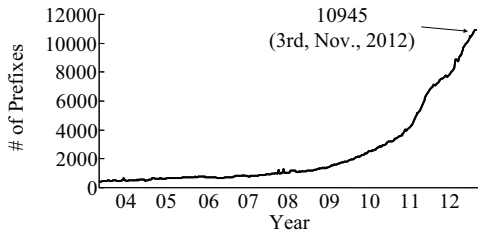

Fig. 1. Numbers of IPv6 prefixes in the routing table for border gateway protocol (BGP).

to IPv6 addresses are encouraged, IPv6 addresses are widely used in core routers. However, since it is a transition period, specifications for IPv6 address are changed frequently[3]. Thus, reconfigurable architecture is necessary to accomodate the changes of specifications.

Large-capacity: As shown in Fig.1, on Nov. 3, 2012, the number of raw IPv6 address in the border gateway protocol (BGP) was about 10 K. The number of IPv4 addresses increased by 25-50 K prefixes per year [2]. Also, the number of IPv6 addresses increases with the rapid growth. Thus, large-capacity routers are necessary for the future IPv6.

1.2 Proposed Architecture and Contributions of the Paper

This paper proposes a memory-based architecture satisfying the four conditions. When IPv6 prefixes with length n are loaded in a single memory, the amount of memory would be $O(2^n)$, which is too large to implement. In this paper, we use a parallel index generation unit (IGU) that reduces the total amount of memory to $O(kn)$, where k denotes the number of prefixes [9]. Also, since the parallel IGU has a simpler architecture than existing ones, it performs a fast lookup by using pipelines. Our contributions are as follows:

1. We loaded more than 340 K pseudo IPv6 prefixes on the parallel IGU implemented on a single FPGA. Its performance is more than 1 GLPS (Giga lookups per second) lookup. As far as we know, this is the first implementation of 1 GLPS engine on a single FPGA.
2. We reduced the total amount of memory for IGUs by using both a linear transformation and a row-shift decomposition. This paper reports of the first implementation of LPM architecture using the parallel IGU.
3. We compared the parallel IGU with existing implementations on FPGA, and showed that the parallel IGU outperforms others.

The rest of the paper is organized as follows: Chapter 2 introduces an architecture for LPM; Chapter 3 shows the IGU and its memory reduction method; Chapter 4 shows the design method for the parallel IGU; Chapter 5 shows the experimental results; and Chapter 6 concludes the paper.

[3] IPv4-compatible IPv6 addresses are abolished. Also, site-local addresses would be abolished.

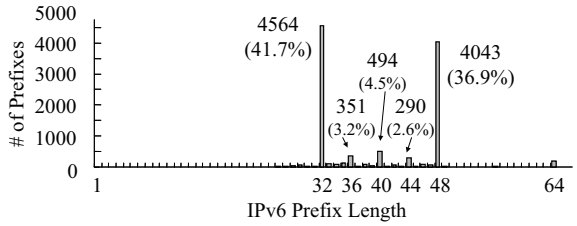

Fig. 2. Distribution of raw IPv6 prefixes (Nov. 3, 2012) [6].

2 Architecture for IPv6 Prefix Lookup

2.1 IPv6 Prefix

The IPv6 address (128 bits) is an extension of the IPv4 address (32 bits). This extension accommodates much larger number of addresses than IPv4. An IPv6 address consists of 64 bits **network prefix (prefix)** and 64 bits interface ID. Since only prefixes are used by the core routers to make forwarding decisions, this paper considers an architecture for the IPv6 prefix lookup. Similar to IPv4 prefix, an IPv6 prefix follows the variable-length subnet masking (VLSM) rule. It consists of n bits network ID and $(64 - n)$ bits sub-network ID. The prefix consists of the following information with bit length:

- FP (Fixed Prefix): 3 bits represented by "001". It means a global unicast accepting route aggregation.
- TLA (Top-Level Aggregation) ID: 13 bits
- sub-TLA: 13 bits
- RES (Reserved for future use): 6 bits
- NLA (Next-Level Aggregation) ID: 13 bits
- SLA (Site-Level Aggregation) ID: 16 bits

Fig. 2 shows the distribution of raw IPv6 prefixes (Nov. 3, 2012) [6]. We observe that variance of the numbers of prefixes with different lengths are quite large. In this paper, we utilize this property to reduce the amount of hardware.

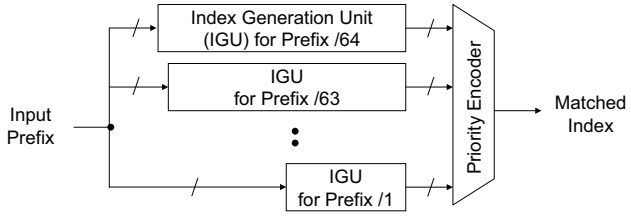

Fig. 3. Architecture for an LPM function.

2.2 Longest Prefix Matching (LPM) Function [12]

Definition 2.1 *The* **LPM table** *stores ternary vectors of the form $VEC_1 \cdot VEC_2$, where VEC_1 consists of $0's$ and $1's$, and VEC_2 consists of $*'s$ (don't cares). The* **length** *of prefix is the number of bits in VEC_1. To assure that the longest prefix address is produced, entries are stored in descending prefix length. The* **LPM function** *is the logic function $\boldsymbol{f} : B^n \rightarrow B^m$, where $\boldsymbol{f}(x)$ is a minimum address whose VEC_1 corresponding to $\boldsymbol{x}$. Otherwise, $\boldsymbol{f}(\boldsymbol{x}) = 0^m$.*

Let P_l be the subset of the prefixes with length l, and $\mathcal{P} = \{P_1, P_2, \ldots P_s\}$ be a set of subsets of the prefixes. Each P_l is represented by an **index generation function** [10].

Definition 2.2 *[10] A mapping $F(\boldsymbol{X}) : B^n \rightarrow \{0, 1, \ldots, k\}$, is an* **index generation function with weight k**, *where $F(\boldsymbol{a}_i) = i$ $(i = 1, 2, \ldots, k)$ for k different* **registered vectors**, *and $F = 0$ for other $(2^n - k)$ non-registered vectors, and $\boldsymbol{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$. In other words, an index generation function produces unique indices ranging from 1 to k for k different registered vectors, and produces 0 for other vectors.*

Example 2.1 *Table 1 shows an index generation function with weight seven.* ■

An LPM function can be decomposed into a set of index generation functions. Thus, this paper focuses on a compact realization of an index generation function.

2.3 Architecture for an LPM function

Fig. 3 shows an architecture for an LPM function realized by index functions with weight k for P_l and a priority encoder, where k equals to the number of prefixes in P_l. When we realized an index generation function for P_l by a single memory, the memory size becomes $O(2^l)$, which is too large for large l. This paper uses an **index generation unit (IGU)** with $O(k)$ memory size.

Table 1. Example of an index generation function f.

x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

Table 2. Decomposition chart for $f(X_1, X_2)$.

	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	x_5
	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1	x_4
	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1	x_3
	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	x_2
00	0 0 0 0 0 0 0 0 1 2 3 0 0 0 4 0	
01	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	
10	5 0 0 0 0 0 0 0 0 0 0 0 0 7 0 0	
11	0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0	
x_6, x_1		

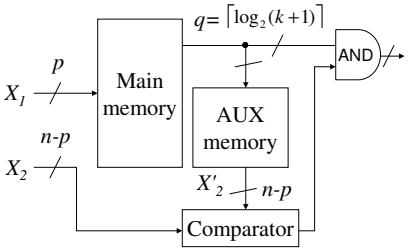

Fig. 4. Index Generation Unit (IGU).

3 Index Generation Unit (IGU)

Table 2 is a **decomposition chart** for the index generation function f shown in Table 1. The columns labeled by $X_1 = (x_2, x_3, x_4, x_5)$ denotes the **bound variables**, and rows labeled by $X_2 = (x_1, x_6)$ denotes the **free variables**. The entry denotes the function

value. We can represent the non-zero elements of f by the **main memory** $\hat{f}$ whose input is X_1. The main memory realizes a mapping from a set of 2^p elements to a set of $k+1$ elements, where $p = |X_1|$. The output for the main memory does not always represent f, since $\hat{f}$ ignores X_2. Thus, we must check whether $\hat{f}$ is equal to f or not by using an **auxiliary (AUX) memory**. To do this, we compare the input X_2 with the output for the AUX memory by a **comparator**. The AUX memory stores the values of X_2 when the value of $\hat{f}(X_1, X_2)$ is non-zero. Fig. 4 shows the index generation unit (IGU). First, the main memory finds the possible index corresponding to X_1. Second, the AUX memory produces the corresponding inputs X_2' ($n-p$ bits). Third, the comparator checks whether X_2' is equal to X_2 or not. Finally, the AND gates produce the correct value of f.

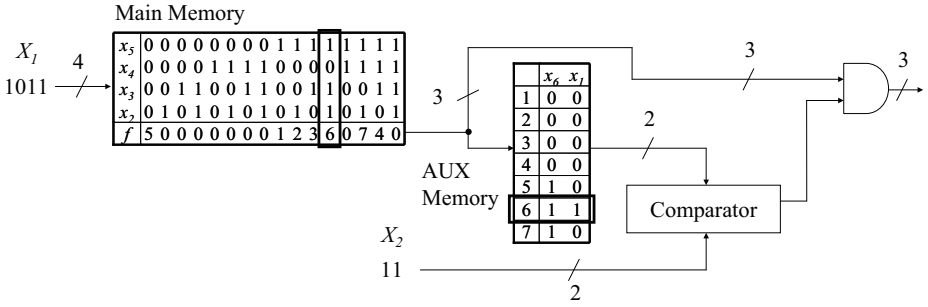

Fig. 5. IGU for Table 1.

Example 3.2 *Fig. 5 shows an example of the IGU realizing the index generation function shown in Table 1. When the input vector is* $X(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 1, 1, 0, 1, 1)$*, the corresponding index is "6". First, the main memory produces the index. Second, the AUX memory produces the corresponding value of* X_2'*. Third, the comparator checks whether* X_2 *and* X_2' *are equal. Since the corresponding input* X_2 *is equal to* X_2'*, the AND gates produces the index. In this case,* $n = 6$*,* $p = 4$*, and* $q = 3$*.* ■

Example 3.3 *To realize the index generation function* f *shown in Table 1, a single memory realization requires* $2^6 \times 3 = 192$ *bits. On the other hand, in the IGU shown in Fig. 5, the main memory requires* $2^4 \times 3 = 48$ *bits, and the AUX memory requires* $2^3 \times 2 = 16$ *bits. Thus, the IGU requires* 64 *bits in total. In this way, we can reduce the total amount of memory by using the IGU.* ■

Example 3.2 is an ideal case. Actually, a column may have two or more than non zero-elements. In such a case, the column has a **collision**. When a collision occurs, a main memory cannot realize a function.

Example 3.4 *Table 4 shows a decomposition chart for an index function* f' *shown in Table 3. In Table 4, the first column has a collision for elements "5" and "6".* ■

3.1 Linear Transformation [8]

Let $\hat{f}(Y_1, X_2)$ be the function whose variables $X_1 = (x_1, x_2, \ldots, x_p)$ are replaced by $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$, $x_i \in \{X_1\}$, $x_j \in \{X_2\}$, and $p \geq$

Table 3. An index generation function f' causing a collision.

x_1	x_2	x_3	x_4	x_5	x_6	f'
0	0	0	0	0	1	1
0	0	0	0	1	0	2
0	0	0	1	0	0	3
0	0	1	0	0	0	4
0	1	0	0	0	0	5
1	0	0	0	0	0	6

Table 4. Decomposition chart for $f'(X_1, X_2)$.

	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	x_3
	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1	x_4
	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1	x_5
	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	x_6
00	1 2 3 4	
01	5	
10	6	
11		
x_1, x_2		

Table 5. Decomposition chart for $\hat{f}'(Y_1, X_2)$.

	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1	$y_1 = x_3 \oplus x_1$
	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1	$y_2 = x_4 \oplus x_1$
	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1	$y_3 = x_5$
	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1	$y_4 = x_6$
00	1 2 3 4	
01	5	
10	6	
11		
x_1, x_2		

$\lceil \log_2(k+1) \rceil$. This replacement is called a **linear transformation**, which can avoid a collision.

Example 3.5 *Let f' be an index generation function shown in Table 3. Table 5 shows the decomposition chart for $\hat{f}'(Y_1, X_2)$, where $Y_1 = (x_3 \oplus x_1, x_4 \oplus x_1, x_5, x_6)$, and the column labels denote Y_1, and the row labels denote X_2. In Table 5, since no collision occurs, it can be realized by the IGU shown in Fig. 4.* ■

The linear transformation for p variables is realized by p copies of two-input EX-ORs. In an FPGA, since these can be realized by p LUTs, their amount of hardware is negligible small.

As shown in Example 3.5, index generation functions often can be represented with fewer variables than original functions. By increasing the number of inputs p for the main memory, we can store virtually all vectors.

Conjecture 3.1 *[9] Consider a set of uniformly distributed index generation functions with weight k (≥ 7). If $p \geq \lceil log_2(k+1) \rceil - 3$, then, more than 95% of the functions can be represented by an IGU with the main memory having p inputs.*

Thus, for the IPv6 prefix lookup problem, a linear transformation of p variables can reduce the amount of memory $O(2^n)$ into $O(2^p)$.

Table 6. Decomposition chart for $\hat{f}'(Y_1)$.

	0 0 0 0 1 1 1 1	y_1
	0 0 1 1 0 0 1 1	y_2
	0 1 0 1 0 1 0 1	y_3
0	5 2 3 4 6	
1	1	
y_4		

Table 7. decomposition chart for $\hat{f}'$ after row-shift.

	0 0 0 0 1 1 1 1	y_1
	0 0 1 1 0 0 1 1	y_2
	0 1 0 1 0 1 0 1	y_3
0	5 2 3 4 6	
1	→ → → 1	
y_4		

3.2 Row-shift Decomposition [7]

In this part, we introduce a **row-shift decomposition** to further reduce of memory size for the IGU. Table 6 shows a decomposition chart for the index generation function

$\hat{f}'(Y_1, Y_2)$, where $Y_1 = (x_3 \oplus x_1, x_4 \oplus x_1, x_5)$ and $Y_2 = (x_6)$. In Table 6, the first column has a collision for the entries "1" and "5". Consider the decomposition chart shown in Table 7 that is obtained from Table 6 by shifting the rows for $y_4 = 1$ by three bit to the right. Table 7 has at most one non-zero element in each column. Thus, the modified function can be realized by a main memory with inputs Y_1.

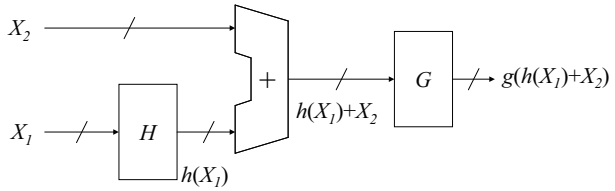

Fig. 6. Row-shift decomposition.

Let X_1 be the row variables, and X_2 be the column variables. In Fig. 6, assume that the memory for H stores the number of bits to shift ($h(X_1)$) for each row specified by X_1, while the memory for G stores the non-zero element of the column after the shift operation: $h(X_1) + X_2$, where "+" denotes an unsigned integer addition. We call this a **row-shift decomposition**.

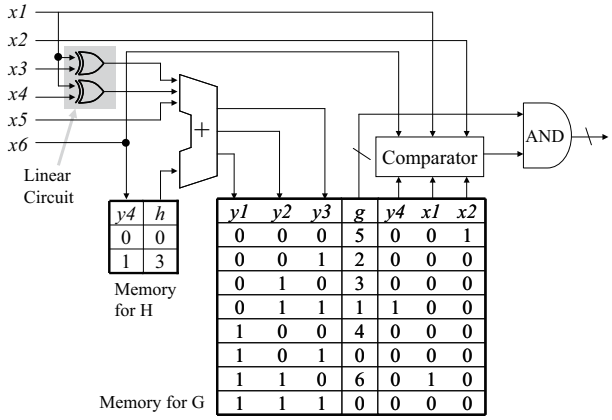

Fig. 7. IGU using a linear transformation and a row-shift decomposition.

Example 3.6 *Fig. 7 shows the IGU using a linear transformation and a row-shift decomposition realizing f' shown in Table 3. Let $\mathcal{Y} = (Y_1, Y_2)$, where $Y_1 = (x_3 \oplus x_1, x_4 \oplus x_1, x_5)$, and $Y_2 = (x_6)$. First, EXOR gates generates $\mathcal{Y}$. Second, the first memory for h produces the shift value $h(Y_1)$. Third, the adder produces $h(Y_1) + Y_2$. In this implementation, since we realize both the main memory and the AUX memory by a single memory, the second memory g produces the index and the corresponding (y_4, x_1, x_2) simultaneously. Next, the comparator checks if they are equal or not. Finally, the AND gates produces the correcting index.* ■

Example 3.7 *To realize f' shown in Table 3, a single memory realization requires $2^6 \times 3 = 192$ bits. On the other hand, in the IGU shown in Fig. 7, the first memory for H requires $2^1 \times 3 = 6$ bits, and the second memory for G requires $2^3 \times (3+3) = 48$ bits.*

Thus, the IGU requires 54 *bits in total. In this way, we can reduce the total amount of memory by using a linear transformation and a row-shift decomposition.* ■

4 Design of Parallel IGU

4.1 Method to Find Linear Transformation

From here, we present a method to find a linear transformation. We assume that the prefix lookup architecture updates its prefix patterns. In this case, it is impractical to find an optimum solution by spending much computation time. To find a reasonably good setting of the EXOR gates, we use the following heuristic algorithm [10], which is simple and efficient.

Algorithm 4.1 *Let* $f(X_1, X_2)$ *be the index generation function of* n *variables with weight* k*, and let* $p = \lceil log_2((k+1)/3) \rceil + 1$ *be the number of the bound variables in the decomposition chart.*

1. *Let* $\{X_1\} = (x_1, x_2, \ldots, x_p)$ *be the bound variables, and* $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$ *be the free variables.*
2. *While* $|X_1| \leq p$*, find variables* $x_i \in \{X_2\}$ *that makes the following value minimum.*

$$|(\text{\# of vectors with} x_i = 0) - (\text{\# of vectors with} x_i = 1)|.$$

   Let $X_1 \leftarrow X_1 \cup \{x_i\}$.
3. *For each pair of variables* (x_i, x_j)*, where* x_i *is a bound variables, and* x_j *is a free variables, if the exchange of* x_i *with* x_j *decreases the number of collision, then do it, otherwise discard it.*
4. *For each pair of variables* (x_i, x_j)*, if the replacement of* x_i *with* $y_i = x_i \oplus x_j$ *decreases the number of collisions, then do it, otherwise discard it.*
5. *Terminate.*

4.2 Design of IGU Using Row-Shift Decomposition [7]

For Table 7, we could represent the function without increasing the columns. However, in general, we must increase the columns to represent the function. Since each column has at most one non-zero element after the row-shift operations, at least k columns are necessary to represent a function with weight k. We use the **first-fit method** [13], which is simple and efficient.

Algorithm 4.2 *(Find row-shifts)*

1. *Sort the rows in decreasing order by the number of non-zero elements.*
2. *Compute the row-shift value for each row at a time, where the row displacement* $r(i)$ *for row* i *has the smallest value such that no non-zero element in row* i *is in the same position as any non-zero element in the previous rows.*

3. *Terminate.*

When the distribution of non-zero elements among the rows is uniform, Algorithm 4.2 reduces the memory size effectively. To reduce the total amount of memories, we use the following:

Algorithm 4.3 *(Row-shift decomposition)*

1. *Reduce the number of variables by the method [9]. If necessary, use a linear transformation [8] to further reduce the number of the variables. Let n be the number of variables after reduction.*
2. *Let $q_1 \leftarrow \lceil \frac{n}{2} \rceil$. From $t = -2$ to $t = 2$, perform Steps 3 through 6.*
3. *Partition the inputs X into (X_1, X_2),where $X_1 = (x_p, x_{p-1}, \ldots, x_1)$ denotes the rows, and $X_2 = (x_n, x_{n-1}, \ldots, x_{p+1})$ denotes the columns.*
4. *$p \leftarrow q_1 + t$.*
5. *Obtain the row-shift value by Algorithm 4.2.*
6. *Obtain the maximum of the shift value, and compute the total amount of memories.*
7. *Find t that minimizes the total amount of memories.*
8. *Terminate.*

x1	*x2*	*x3*	*x4*	*f*
0	0	0	1	1
0	0	1	*	2
0	0	*	*	3

Prefix Expansion

x1	*x2*	*x3*	*x4*	*f*
0	0	0	1	1
0	0	1	0	2
0	0	1	1	2
0	0	0	0	3

Fig. 8. Example of prefix expansion.

4.3 Prefix Expansion

Let P_l be the set of the prefixes with length l, and let $\mathcal{P} = \{P_1, P_2, \ldots P_s\}$ be the set of the set of the prefixes. As shown in Fig. 3, the parallel IGU consists of s IGUs and a priority encoder whose size is proportional to s. Thus, the straightforward implementation requires large amount priority encoder and many IGUs.

To reduce the number of required IGUs, we merge multiple P_l into a group. By expanding the prefixes in P_l to ones with length $l + 1$, we can make a group including P_{l+1} and P_l. We call this **prefix expansion**. The next example shows it.

Example 4.8 *The left-hand side Table in Fig. 8 stores $\{P_2, P_3, P_4\}$, where $P_2 = \{00 * *\}$, $P_3 = \{001*\}$, $P_4 = \{0001\}$. By performing prefix expansion to P_2, we have $P_3' = \{000*, 001*\}$. By the longest prefix matching (LPM) rule, the prefix $\{001*\}$ that is equal to $\{001*\}$ in P_3 is ignored. Also, by performing prefix expansion to P_3', we have P_4' shown in the right-hand side Table in Fig. 8.* ■

Fig. 2 shows that the variance of the numbers of prefixes with different lengths P_l is quite large. When the prefix expansions to P_l consisting of a small number of prefixes is applied, they can be stored into a single BRAM [4]. On the other hand, when the prefix expansion to P_l consisting of a large number of prefixes is applied, in the worst case, the size would exceed that of the available BRAMs. Thus, we make a **non-uniform grouping** G_i $\mathcal{G} = \{G_1, G_2, \ldots, G_r\}$, where G_i is generated from the different number of P_l. To find an optimal grouping without increasing BRAMs, we use the following:

[4] For Xilinx Virtex 6 FPGA, the BRAM stores 36Kbits.

Table 8. The number BRAMs to realize IGUs with non-uniform grouping (BRAMs marked with "*" were realized by distributed RAMs in the actual implementation).

Group	#prefixes	Memory H		Memory G		# of 36Kb
	in a group	#In	#Out	#In	#Out	BRAMs
(15,16,17,18)	102	4	6	7	18	2 *
(19,20,21,22)	225	7	7	8	22	2 *
(23,24,25,26)	1,571	10	11	11	26	3 *
(27,28)	806	6	11	11	28	3 *
(29,30)	1,240	6	12	12	30	5 *
(31)	2,824	9	12	12	31	5 *
(32)	8,474	9	14	14	32	16
(33)	1,469	8	11	11	33	3 *
(34)	4,408	10	12	12	34	5 *
(35)	2,318	10	11	13	35	9
(36)	6,957	11	13	13	36	9
(37)	4,079	13	12	12	37	8
(38)	12,237	14	14	14	38	24
(39)	6,592	12	12	13	39	11
(40)	19,776	13	14	14	40	22
(41)	6,874	13	13	13	41	12
(42)	20,623	14	15	15	42	42
(43)	9,451	14	14	14	43	27
(44)	28,354	13	15	15	44	47
(45)	8,522	13	14	14	45	24
(46)	25,569	14	15	15	46	48
(47)	42,768	14	16	16	47	92
(48)	128,305	14	17	17	48	179
(49,50)	929	10	10	10	50	3*
(51,52)	1,048	11	11	11	52	4*
(53,54)	594	9	10	10	54	3*
(55,56)	421	8	9	9	9	2*
(57,58)	530	9	9	9	58	2*
(59,60,61,62)	289	7	8	9	62	2*
(63,64)	386	8	9	9	64	2*
Total	347,749					616

Table 9. Comparison of non-uniform grouping with uniform ones.

Grouping	#prefixes	#groups	#BRAMs	#Slices
Non-uniform (Table 8)	347,749	30	616	2,299
without grouping (direct realization of $\mathcal{P}$)	348,877	50	655	3,979
Uniform for two subsets	382,132	25	785	1,899
Uniform for four subsets	1,250,695	13	2,512	939

Algorithm 4.4 *(Non-uniform grouping) Let P_l be the set of the prefixes with length l, $\mathcal{P} = \{P_1, P_2, \ldots, P_s\}$ be the set of the prefixes, G_j consists of single or several P_ls, r be the number of groups, and $\mathcal{G} = \{G_1, G_2, \ldots, G_r\}$, where $r \leq s$.*

1. *Apply Algorithms 4.1 and 4.3 to P_l $(1 \leq l \leq s)$ to generate the IGU. Let B_l be the number of BRAMs to realize the IGU.*
2. *$r \leftarrow 1$, and $i \leftarrow 1$.*
3. *$G_r \leftarrow P_i$, and $B \leftarrow B_i$.*
4. *$i \leftarrow i + 1$. If $i > s$, then go to Step.7.*
5. *Perform a prefix expansion to $G_r \cup P_i$, then apply Algorithms 4.1 and 4.3 to them to generate the IGU. Let B_{temp} be the number of BRAMs to realize the IGU.*

6.1. *If $B + B_i \geq B_{temp}$, then $G_r \leftarrow G_r \cup P_i$, $B \leftarrow B_{temp}$, and go to Step.4.*

6.2. *$r \leftarrow r + 1$, and go to Step.3.*

7. *Terminate.*

Table 10. Comparison with existing FPGA implementations.

Architecture	#prefixes	#Slices	# of 36Kb BRAMs	Off-chip SRAM [Mb]	Normalized area #Slices	Normalized area #BRAMs	Throughput [MLPS]
Baboescu et al. (ISCA2005) [1]	80 K	1,405	530	—	17.5	6.6	125
Fadishei et al. (ANCS2005) [3]	80 K	14,274	254	—	178.4	3.1	263
Le et al. (FCCM2009) [5]	249 K	16,617	473	—	66.7	1.8	340
2-3-tree-IPv6 (IEEE Trans.2012) [4]	330 K	15,358	580	32.5	46.5	4.5	373
BST-IPv6 (IEEE Trans.2012) [4]	330 K	14,096	1,025	3.2	42.7	3.3	390
Parallel IGU	340 K	5,577	575	—	**16.4**	**1.7**	**1,002**

5 Experimental Results

5.1 Implementation of the Parallel IGU

We designed the parallel IGU using Xilinx PlanAhead 14.2, and implemented on the Roach2 board (FPGA: Xilinx Virtex 6 (XC6VSX475T), 74,400 Slices, 1,064 BRAMs (36Kb)). Pseudo IPv6 prefixes were generated from the present raw 340 K IPv4 prefixes (Nov. 3, 2012) using a method [15]. Since the present IPv6 uses prefixes with length 15 or more, we generated such prefixes only.

Table 8 shows the number of BRAMs in IGUs to load 340 K pseudo IPv6 prefixes generated by Algorithm 4.4. Table 9 compares non-uniform grouping with uniform one. In Table 9, *#Slices* includes the number of slices for both IGUs and the priority encoder. Table 9 shows that, the number of BRAMs for the non-uniform grouping is smaller than that for the uniform grouping. Although the non-uniform grouping requires more slices than the uniform one, it consumes less than 10% of the available resources in the FPGA.

Since we implemented small memory part (marked with "*" in Table 8) by distributed RAMs instead of BRAMs, they consumed 3,288 slices. Thus, the parallel IGU used 5,577 slices and 575 BRAMs. Since we implemented a complete pipeline architecture, the maximum clock frequency was 501.4 MHz. By using a dual port BRAM, the lookup speed for the parallel IGU was 1,002 MLPS (mega lookups per second).

5.2 Comparison with Existing Implementations

Table 10 compares the parallel IGU with existing implementations. Since existing implementations store different numbers of prefixes to compare the efficiency, we used

the normalized area, which shows the number of primitives (# of slices or BRAMs) per a prefix. As for the off-chip SRAMs, we converted them to the equivalent of 36Kb BRAM numbers. Table 10 shows that, as for the lookup speed, the parallel IGU is 2.56-8.01 times faster than existing implementations. As shown in Fig. 7, the parallel IGU has a simple architecture which is suitable for pipelined implementation to increase the throughput. Also, as for the normalized area, the parallel IGU has the smallest implementation. Therefore, the parallel IGU outperforms existing FPGA realizations.

6 Conclusion

This paper showed the parallel IGU for IPv6 Lookup. To reduce the memory size of the IGU, we used linear transformation and row-shift decompositions. Also, this paper showed a design method for the parallel IGU. We implemented the parallel IGU on the Xilinx Virtex 6 FPGA, it loaded more than 340 K IPv6 prefixes, and its lookup speed is 1,002 MLPS. Experimental results showed that our implementation outperforms existing FPGA realizations.

7 Acknowledgments

This research is supported in part by Strategic Information and Communications R&D Promotion Program (SCOPE), and the Grants in Aid for Scientistic Research of JSPS.

References

1. F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," *ISCA2005*, 2005, pp.123.
2. H. J. Chao and B. Liu, *High performance switches and routers,* JohnWiley& Sons, Inc.,Hoboken, NJ, USA, 2007.
3. H. Fadishei, M.S. Zamani, and M.Sabaei, "A novel reconfigurable hardware architecture for IP address lookup," *ANCS2005*, 2005, pp.81-90.
4. H. Le and V. K. Prasanna, "Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning," *IEEE Trans. on Compt*, Vol. 61, No. 7, 2012, pp.1026-1039.
5. H. Le and V. K. Prasanna, "Scalable high throughput and power efficient IP-lookup on FPGA," *FCCM2009*, April, 2009.
6. University of Oregon route views project, http://http://www.routeviews.org/
7. T. Sasao, "Row-shift decompositions for index generation functions," *DATE2012*, 2012, pp.1585-1590.
8. T. Sasao, "Linear decomposition of index generation functions," *ASPDAC2012*, 2012, pp.781-788.
9. T. Sasao, *Memory-based logic synthesis, Springer.*, 2011.
10. T. Sasao, M. Matsuura and H. Nakahara, "A realization of index generation functions using modules of uniform sizes," *IWLS'10*, June 18-20, 2010, pp.201-208.
11. 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.
12. T. Sasao and J. T. Butler, "Implementation of multiple-valued CAM functions by LUT cascades," *ISMVL-2006*, Singapore, May 17-20, 2006.
13. R. E. Tarjan and A. C-C. Yao, "Storing a sparse table," *Communications of the ACM*, Vol. 22, No. 11, 1979, pp.606-611.
14. R. Tucker, "Optical packet-switched WDM networks: a cost and energy perspective," *OFC/NFOEC2008*, 2008.
15. M. Wang, S. Deering, T. Hain, and L.Dunn, "Non-random generator for IPv6 tables," *HOTI2004*, 2004, pp.35-40.