

# Scalable High Throughput and Power Efficient IP-Lookup on FPGA \*

Hoang Le and Viktor K. Prasanna  
Ming Hsieh Department of Electrical Engineering  
University of Southern California  
Los Angeles, USA  
{hoangle, prasanna}@usc.edu

## Abstract

Most high-speed Internet Protocol (IP) lookup implementations use tree traversal and pipelining. Due to the available on-chip memory and the number of I/O pins of Field Programmable Gate Arrays (FPGAs), state-of-the-art designs cannot support the current largest routing table (consisting of 257K prefixes in backbone routers). We propose a novel scalable high-throughput, low-power SRAM-based linear pipeline architecture for IP lookup. Using a single FPGA, the proposed architecture can support the current largest routing table, or even larger tables of up to 400K prefixes. Our architecture can also be easily partitioned, so as to use external SRAM to handle even larger routing tables (up to 1.7M prefixes). Our implementation shows a high throughput (340 mega lookups per second or 109 Gbps), even when external SRAM is used. The use of SRAM (instead of TCAM) leads to an order of magnitude reduction in power dissipation. Additionally, the architecture supports power saving by allowing only a portion of the memory to be active on each memory access. Our design also maintains packet input order and supports in-place non-blocking route updates.

## 1 Introduction

### 1.1 Internet Protocol Packet Forwarding

With the rapid growth of the Internet, IP packet forwarding, or simply IP lookup, becomes the bottle-neck in network traffic management. Therefore, the design of high speed IP routers has been a major area of research. Advances in optical networking technology are pushing link rates in high speed IP routers beyond OC-768 (40 Gbps). Such high rates demand that packet forwarding in IP routers must be performed in hardware. For instance, a 40 Gbps link requires a throughput of 125 million packets per sec-

**Table 1:** Comparison of TCAM and SRAM

|                                  | TCAM(18Mb) | SRAM(18Mb) |
|----------------------------------|------------|------------|
| Maximum clock rate (MHz)         | 266        | 400        |
| Cell size (# of transistors/bit) | 16         | 6          |
| Power consumption (Watts)        | 12 ~ 15    | ≈ 1        |

ond (MPPS), for a minimum size (40-byte) packet. Such throughput is impossible to achieve using existing software-based solutions [1].

IP lookup is a classic problem. Most hardware-based solutions in network routers fall into two main categories: TCAM-based and dynamic/static random access memory (DRAM/SRAM)-based solutions. Although TCAM-based engines can retrieve results in just one clock cycle, their throughput is limited by the relatively *low speed* of TCAMs. They are expensive, *power-hungry*, and offer little adaptability to new addressing and routing protocols [7]. As shown in Table 1, SRAMs outperform TCAMs with respect to speed, density, and power consumption [2, 3, 4, 5, 6].

SRAM-based solutions, on the other hand, require multiple cycles to process a packet. Therefore, pipelining techniques are commonly used to improve the throughput. These SRAM-based approaches, however, result in an inefficient memory utilization. This inefficiency limits the size of the supported routing tables. In addition, it is not feasible to use external SRAM in these architectures, due to the constraint on the number of I/O pins. This constraint restricts the number of external stages, while the amount of on-chip memory confines the size of memory for each pipeline stage. Due to these two constraints, state-of-the-art SRAM-based solutions do not scale to support larger routing tables. This scalability has been a dominant issue for any implementations on FPGAs. Furthermore, pipelined architectures increase the total number of memory accesses per clock cycle, and thus, increase the dynamic power consumption. The power dissipation in the memory dominates that in the logic [8, 9, 10]. Therefore, reducing memory power dissipation contributes to a large reduction in the to-

\*Supported by the U.S. National Science Foundation under grant No. CCF-0702784. Equipment grant from Xilinx is gratefully acknowledged.

tal power consumption.

## 1.2 Challenges and Contributions

The key issues to be addressed in designing an architecture for IP packet forwarding engine are: (1) size of supported routing table, (2) throughput, (3) scalability (with multiple chips or external storage), (4) in-order packet output, (5) incremental update, and (6) power consumption. To address these challenges, we propose and implement a scalable, high-throughput SRAM-based multiple-linear-pipeline architecture for IP lookup on FPGAs. This architecture eliminates the need to store the addresses of the child nodes by using a binary search tree (BST) structure. In a complete binary tree, the amount of memory to store the nodes in a next level doubles as we go from a level to the next. As a result, we can move the largest levels onto external SRAMs. In addition, the linear architecture also preserves the packet order of incoming network traffic. The number of levels (or pipeline stages) is determined by the size of the supported routing table, as discussed in detail in Section 5.

This paper makes the following **contributions**:

- To the best of our knowledge, the proposed architecture is the first binary-tree-based design to use on-chip FPGA resources only to support the *current largest routing table* of over 260K prefixes, and up to **400K prefixes** (Section 5.1).
- The architecture can easily utilize external SRAM to handle up to 1.7M prefixes (Section 5.2).
- The proposed architecture scales well as the number of prefixes grows, due to a linear storage complexity and low resource utilization (Section 4.4).
- The implementation results show a *sustained* throughput of **340 MLPS**, even when external SRAM is used (Section 7.1).
- It supports power efficiency by minimizing the amount of active on-chip memory (Section 6).

The rest of the paper is organized as follows. Section 2 covers the background and related work. Section 3 introduces the IP lookup algorithm. Section 4 and 5 describe the proposed architecture and its implementation. Section 6 discusses the power optimization. Section 7 presents implementation results. Section 8 concludes the paper.

## 2 Background and Related Work

The following definitions and notations are used throughout this paper:

1. Packet’s IP address, or simply IP address, is the network address where the packet is destined to. It is extracted from the header of the incoming packet.
2. Next hop (routing) index is a parameter that determines where the packet is forwarded.
3.  $|P|$  is the length or the number of bits of prefix  $P$ .
4. The “\_” symbol represents concatenation operation.

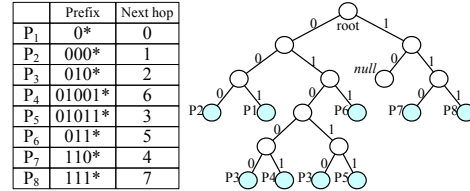


Figure 1: A sample routing table and its leaf-pushed trie.

### 2.1 Background

In computer networking, a routing table is an electronic table or database that is stored in a router or a networked computer. The routing table stores the routes and metrics associated with those routes, such as next hop routing indices, to particular network destinations. The IP lookup problem is longest prefix matching (LPM). LPM refers to an algorithm used by routers in IP networking to select an entry from a routing table. To determine the outgoing port for a given address, the longest matching prefix among all the prefixes needs to be determined. Routing tables often contain a default route, which has the shortest possible prefix match, in case matches with all other entries fail. In a sample routing table shown in Figure 1, binary prefix  $P_4$  (01001\*) matches all destination addresses that begin with 01001. Similarly, prefix  $P_5$  matches all destination addresses that begin with 01011. The destination address  $IP = 01011100$  is matched by the prefixes  $P_1$ ,  $P_3$ , and  $P_5$ . Since,  $|P_1| = 1$ ,  $|P_3| = 3$ , and  $|P_5| = 5$ ,  $P_5$  is the longest prefix that matches  $IP$ . In longest-prefix routing, the next hop index for a packet is given by the table entry corresponding to the longest prefix that matches its destination IP address.

Current solutions can be classified into two main groups: TCAM-based and SRAM-based. In TCAM-based solutions, each prefix is stored in a word. An incoming IP address is searched in parallel on all words in TCAM in one clock cycle. TCAM-based solutions are simple, and therefore, are de-facto solutions for today’s routers. In SRAM-based solutions, the common data structure in algorithmic solutions for performing LPM is some form of tree, such as a trie [1]. A trie is a binary-tree-like data structure for LPM. Each prefix is represented by a node in the trie, and the value of the prefix corresponds to the path from the root of the tree to the node. IP lookup is performed by traversing the trie according to the bits in the IP address. When a leaf is reached, the last seen prefix along the path to the leaf is the longest matching prefix for the IP address. A sample trie is shown in Figure 1.

The common problem of these trie-based IP lookup architectures is that each node must store the addresses of its child nodes and the next hop index. Even with an optimization called leaf-pushing [11], each node still needs to store one field: either the next hop index or the pointer to the child nodes. This address book-keeping overhead leads to an inefficient storage as the address length increases when

the number of nodes (or prefixes) increases.

## 2.2 Related Work

Despite a large amount of work in IP lookup ([11, 12, 13, 14]), most of them do not target FPGA platform. Since the proposed work addresses FPGA implementation, we summarize the related work in this area. TCAM is widely used to simplify the complexity of the designs. However, TCAM results in lower overall clock speed and higher power consumption of the entire system.

The fastest IP lookup implementation on FPGAs to date is reported in [15], which can achieve a lookup rate of 325 MLPS. This is a bidirectional optimized linear pipeline architecture, named BiOLP, which takes advantage of the dual-ported SRAM to map the prefix trie in both directions. By doing this, BiOLP achieves a perfectly balanced memory allocation over all pipeline stages. BiOLP can support a Mae-West routing table (rrc08, 84K prefixes). BiOLP is based on trie, whereas in the proposed work, binary search tree algorithm is used.

Another fast IP lookup implementation on FPGAs thus far is described in [16], which can achieve 263 MLPS. Their architecture takes advantage of the benefit of both a traditional hashing scheme and reconfigurable hardware. They implement only the colliding prefixes (prefixes with the same hashing value) on reconfigurable hardware, and the remaining prefixes in a main table on memory. This architecture supports a Mae-West routing table (rrc08, 84K prefixes), and can be updated using partial reconfiguration when adding or removing prefixes. The update time is lower bounded by the reconfiguration time. It is also not clear how to scale this design to support larger routing tables.

Baboescu et al. [17] propose a Ring pipeline architecture for tree-based search engines. The pipeline stages are configured in a circular, multi-point access pipeline so that the search can be initiated at any stage. The implementation in [18] achieves a throughput of 125 MLPS. Sangireddy et al. [19] propose two algorithms, Elevator-Stairs and log W-Elevators, which are scalable and memory efficient. Yet, their designs can achieve only up to 21.41 MLPS. Meribout et al. [20] present another architecture, with a lookup speed of 66 MLPS. In this design, Random Access Memory is required, and the achieved lookup rate is reasonably low.

As summarized above, all of the prior work on FPGA can not support a routing table with 100K or more prefixes. Moreover, none of the prior work offers any techniques to reduce power consumption of the memory.

## 3 IP Lookup Algorithm

### 3.1 Prefix Properties

Let  $W$  be the length of the IP address,  $W = 32$  for IPv4. We first introduce two prefix properties that are used as the foundation of our work.

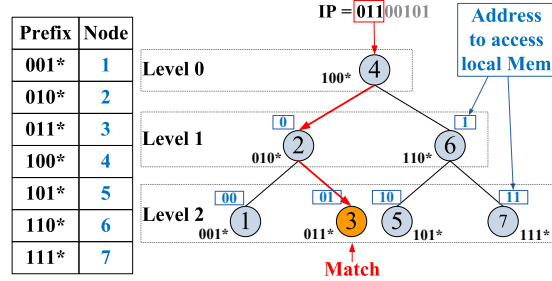


Figure 2: A sample routing table with prefix length of 3 and its corresponding binary search tree.

**Property 1:** Given two prefixes,  $P_A$  and  $P_B$ , if  $|P_A| = |P_B|$  then  $P_A$  and  $P_B$  do not overlap.

**Proof:** In the IP address space, each prefix represents an address range. Without loss of generality, assume that  $P_A < P_B$ . The range of  $P_A$  and  $P_B$  are  $[P_A.0..0, P_A.1..1]$  and  $[P_B.0..0, P_B.1..1]$ , respectively. The number of appended 0s and 1s is  $W - |P_A|$  for both  $P_A$  and  $P_B$ , as they are of the same length. Since  $P_A < P_B$ , we have  $[P_A.1..1] < [P_B.0..0]$ , and hence the two ranges do not overlap.

**Property 2:** Given a prefix  $P_A$  with length  $n$ ,  $P_A$  can be represented as the union of 2 prefixes  $P_B$  and  $P_C$  of length  $(n + 1)$  by appending 0 and 1 to  $P_A$ .

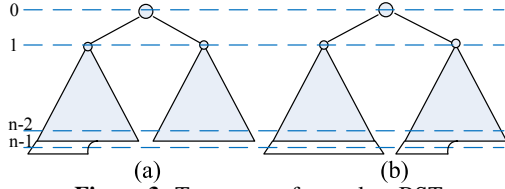
**Proof:** Let  $P_B = P_A.0$  and  $P_C = P_A.1$ . The ranges of  $P_B$  and  $P_C$  are  $[P_A.0.0..0, P_A.0.1..1]$  and  $[P_A.1.0..0, P_A.1.1..1]$ , respectively. Since  $P_A.0.1..1 + 1 = P_A.1.0..0$ , the union of these two ranges is  $[P_A.0.0..0, P_A.1.1..1]$ , or  $[P_A.0..0, P_A.11..11]$ , which is exactly the range of  $P_A$ .

### 3.2 Binary-Search-Tree-based IP Lookup

We propose a memory efficient data structure based on a binary search tree (BST). BST is a special binary tree data structure with the following properties: (1) each node has a value, (2) the left subtree of a node contains only values less than the node's value, and (3) the right subtree of a node contains only values greater than the node's value. The binary search algorithm is a technique to find a specific element in a sorted list. In a balanced binary search tree, an element, if it exists, can be found in at most  $(1 + \lfloor \log_2 N \rfloor)$  operations, where  $N$  is the total number of nodes in the tree.

A sample routing table with prefixes of length 3 and its corresponding BST are illustrated in Figure 2. For simplicity, 8-bit IP addresses are considered. Each node of the binary tree contains a *prefix* and its corresponding *next hop index*. Only the  $k$  most significant bits of the IP address enter the tree from its root, where  $k$  is the length of the prefix (3 in this case). At each node, the  $k$  most significant bits of the IP address and node's prefix are compared to determine the matching status and also the traversal direction.

For each set of prefixes of the same length, we build a binary search tree. Given such a binary search tree, IP lookup is performed by traversing left or right, depending on the comparison result at each node. If the entering IP address is



**Figure 3:** Two cases of complete BST

smaller or equal to a node’s prefix, it is forwarded to the left branch of the node, and to the right branch otherwise. For example, assume that a packet with destination address of 01100101 arrives. At the root, the prefix 100 is compared with 011, which is smaller. Thus, the packet traverses to the left. The comparison with the prefix in node #2 yields a greater outcome, hence the packet traverses to the right. At node #3, the packet header matches the node’s prefix, which is the final result.

We must ensure that the proposed algorithm actually finds the longest matching prefix. Based on *Property 1*, prefixes with the same length do not overlap. If a given IP address has the longest prefix match of length  $k$ , its  $k$  most significant bits must be a prefix, which is a node in the binary search tree of prefixes of length  $k$ . The property of binary search tree guarantees that that prefix is found. Other binary search trees of prefixes of shorter lengths may give some matches as well. However, only the match from the binary search tree of prefixes of the longest length is returned by using a priority selector.

### 3.3 BST Construction

---

**Algorithm 1** COMPLETEBST(SORTED\_ARRAY)

---

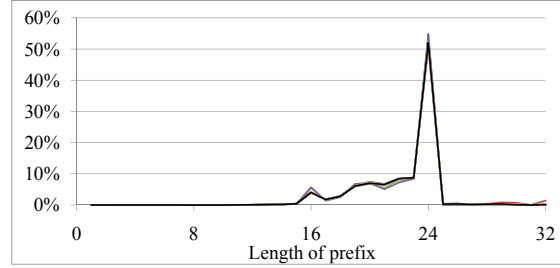
**Input:** Array  $A[N]$  of  $N$  prefixes sorted in ascending order

**Output:** Complete BST

- 1:  $n = \lceil \log_2(N + 1) \rceil, \Delta = N - (2^{n-1} - 1)$
  - 2: **if**  $(\Delta \leq 2^{n-1}/2)$  **then**
  - 3:    $x = 2^{n-2} - 1 + \Delta$
  - 4: **else**
  - 5:    $x = 2^{n-1} - 1$
  - 6: **end if**
  - 7: Pick element  $A[x]$  as root
  - 8: Left-branch = COMPLETEBST(left-of- $x$  sub-array)
  - 9: Right-branch = COMPLETEBST(right-of- $x$  sub-array)
- 

We use a *complete* BST for efficient memory utilization. In such a tree, all levels are fully occupied, except possibly the last one. If the last level is not full, all nodes are as far left as possible. A group of prefixes of the same length is sorted into an array in ascending order. The corresponding BST can easily be built by picking the right prefix (pivot) as the root, and recursively building the left and right subtrees. Two cases of complete BST, in which the last level is not complete, are illustrated in Figure 3.

Let  $N$  be the number of prefixes in one group,  $n$  be the number of levels, and  $\Delta$  be the number of prefixes in the last level. The total number of nodes in all stages, excluding the



**Figure 4:** Prefix histogram of sample routing tables.

last stage, is  $2^{n-1} - 1$ . Therefore, the number of nodes in the last stage is  $\Delta = N - (2^{n-1} - 1)$ . There are  $2^{n-1}$  nodes in the last stage if it is full. If  $\Delta \leq 2^{n-1}/2$ , we have a complete BST, as in Figure 3 (a), or (b) otherwise. Let  $x$  be the index of the desired root,  $x$  can be calculated as  $x = 2^{n-2} - 1 + \Delta$  for case (a), or  $x = 2^{n-1} - 1$  for case (b). The complete BST can be built recursively, as described in Algorithm 1.

### 3.4 Prefix Partitioning

16 real-life routing tables are shown in Table 2. These routing tables were collected from [21] on 2007/11/30. The prefix histograms of different routing tables are illustrated in Figure 4. An interesting observation is that all 16 sample routing tables have a similar distribution, which suggests that a common prefix partitioning can be used.

The number of binary search trees can be reduced by applying *Property 2*. By grouping 2 consecutive prefix lengths together, we reduce the number of BSTs by half. There may be some conflicts when converting prefixes with length  $n$  to length  $n + 1$ . These conflicts occur if prefix  $P_A$  of length  $n + 1$  is a longer prefix of  $P_B$  of length  $n$ . In this case, the converted prefix of  $P_B$  that conflicts with  $P_A$  is ignored, as  $P_A$  is longer.

Groups of more than 2 consecutive prefix lengths can also be considered. However, while the number of BSTs decreases, the total number of combined prefixes may increase. A dynamic programming algorithm can be utilized to determine the optimal size of each group. Only group of size 2 is addressed in this paper.

We use the largest sample routing table rrc7 (248856 prefixes) for our analysis. The number of prefixes of length 1 to 14 is very small compared to others. Hence, we convert the first 13 prefix lengths to length 14. The number of original and converted prefixes are shown in Table 3. Since the total number of prefixes up to prefix of length 14 after conversion is 3592, it is reasonable to perform direct memory access on them. This search requires only 14-bit address, which is extracted from the higher-order bits of the incoming IP address.

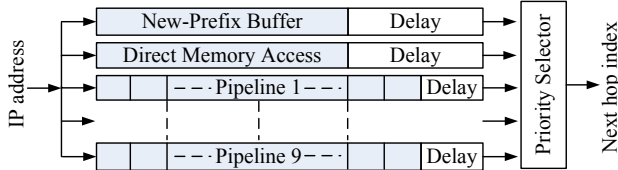
The rest of the prefixes are organized into 9 groups, each with 2 consecutive prefix lengths: (15,16), (17,18),..., (31,32). The notation  $(m, n)$  represents a group of all prefixes of length  $m$  to  $n$ . The distribution of the orig-

**Table 2:** Collected sample routing tables

| Table      | rrc0 | rrc1 | rrc2 | rrc3 | rrc4 | rrc5 | rrc6 | rrc7 | rrc8 | rrc9 | rrc10 | rrc11 | rrc12 | rrc13 | rrc14 | rrc15 |
|------------|------|------|------|------|------|------|------|------|------|------|-------|-------|-------|-------|-------|-------|
| # prefixes | 244K | 241K | 238K | 247K | 240K | 242K | 240K | 249K | 84K  | 133K | 237K  | 239K  | 244K  | 239K  | 244K  | 243K  |

**Table 3:** Converted prefixes from length (1 to 13) to 14

| Prefix's length      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8    | 9   | 10  | 11  | 12  | 13  | 14  | TOTAL       |
|----------------------|---|---|---|---|---|---|---|------|-----|-----|-----|-----|-----|-----|-------------|
| # prefixes           | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18   | 9   | 16  | 40  | 134 | 275 | 490 | 982         |
| # converted prefixes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1152 | 288 | 256 | 320 | 536 | 550 | 490 | <b>3592</b> |

**Figure 5:** Overall Architecture.

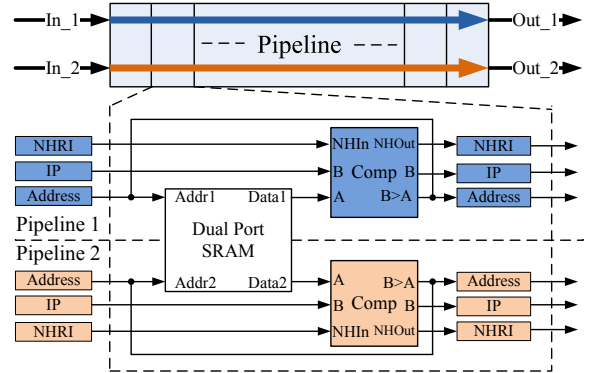
inal prefixes and the combined prefixes of each group can be seen in Table 4. The worst-case combined prefix occurs when there is no prefix that is longer than any other prefixes in a group. The actual total number of combined prefixes in each group is often less. Even with the largest table, the amount of required memory is only 8878 Kb, which is less than the 10 Mb on-chip memory of a state-of-the-art FPGA device. Analysis of other tables show similar memory distribution and require a smaller amount of memory. Due to space limitation, these results are not shown.

## 4 Architecture

### 4.1 Overall Architecture

Binary tree structure is utilized in our design. To ensure that every IP lookup results in the same number of operations or cycles, the IP address continues with all the comparisons even though a match may have already been found. Pipelining is used to increase the throughput. The number of pipeline stages is determined by the height of the BST. Each level of the tree is mapped onto a pipeline stage, which has its own memory (or table). The size of table doubles from one stage to the next stage. The maximum number of prefixes in a stage is determined by  $2^n$ , where  $n$  is the stage index.

Figure 5 describes the overall architecture of the proposed IP lookup engine. There are nine pipelines (one for each group of prefixes), one direct memory access block, and one new-prefix buffer block. The IP address is extracted from the incoming packet and routed to all branches. The searches are performed in parallel in all the pipelines, the memory block, and the new-prefix buffer. The results are fed through a priority selector to select the next hop index of the longest matched prefix. The variation in the number of stages in these pipelines results in latency mismatch. The delay block is appended to each shorter pipeline to match with the latency of the longest pipeline. A similar delay approach is used in the direct memory access and the update

**Figure 6:** Block diagram of a single pipeline (Addr - Address; NHRI - Next hop routing index; IP - IP Address).

block to match the latencies.

### 4.2 Single-pipeline Architecture

The block diagram of the basic pipeline and a single stage are shown in Figure 6. The on-chip memory in FPGAs comes with a dual-ported feature. To take advantage of it, the architecture is configured as dual linear pipelines to double the lookup rate. At each stage, the memory has dual Read/Write ports so that two packets can be input every clock cycle. The content of each entry in the memory includes the prefix and its next hop routing index. In each pipeline stage, there are 3 data forwarded from the previous stage: (1) the IP address, (2) the memory access address, and (3) the next hop index. The forwarded memory address is used to retrieve the node prefix, which is compared with the IP address to determine the matching status. In case of a match, the next hop index of the new match replaces the old result. The comparison result (1 if the IP address is greater than node prefix, 0 otherwise) is appended to the current memory address and forwarded to the next stage.

### 4.3 Routing Table Update

Routing table update includes three operations: (1) existing prefix modification, (2) prefix deletion, and (3) new prefix insertion. The first update requires changing the next hop indices of the existing prefixes in the routing table. This type of update can easily be done by inserting write bubbles [22], as shown in Figure 7. The new content of the memory is computed off-line. When a prefix update is initiated, a write bubble is inserted into the pipeline. Each write bubble is assigned an ID. There is one write bubble

**Table 4:** Converted prefixes of length 15 to 32

|                                |       |      |       |      |       |       |       |       |                |        |
|--------------------------------|-------|------|-------|------|-------|-------|-------|-------|----------------|--------|
| Prefix length                  | 15    | 16   | 17    | 18   | 19    | 20    | 21    | 22    | 23             | 24     |
| # prefixes                     | 961   | 9763 | 4240  | 6824 | 14826 | 16949 | 15948 | 20213 | 21346          | 125592 |
| # worst case combined prefixes | 11685 |      | 15304 |      | 46601 |       | 52109 |       | 168284         |        |
| Required amount of memory (Kb) | 258   |      | 368   |      | 1212  |       | 1460  |       | 5049           |        |
| Prefix's length                | 25    | 26   | 27    | 28   | 29    | 30    | 31    | 32    | TOTAL          |        |
| # prefixes                     | 1084  | 1090 | 568   | 905  | 2083  | 1899  | 0     | 3583  | <b>248846</b>  |        |
| # worst case combined prefixes | 3258  |      | 2041  |      | 6065  |       | 3583  |       | <b>312522</b>  |        |
| Required amount of memory (Kb) | 107   |      | 70    |      | 219   |       | 137   |       | <b>8878 Kb</b> |        |

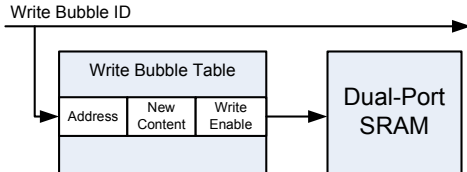
**Figure 7:** Route update using write-bubbles.

table (WBT) in each stage. The table stores the update information associated with the write bubble ID. When it arrives at the stage prior to the stage to be updated, the write bubble uses its ID to look up the WBT. Then the bubble retrieves (1) the memory address to be updated in the next stage, (2) the new content for that memory location, and (3) a write enable bit. If the write enable bit is set, the write bubble will use the new content to update the memory location in the next stage. This updating mechanism supports non-blocking prefix updates at system speed.

The second type of update requires deleting existing prefix. There are two methods: lazy deletion and complete deletion. The lazy approach assigns the default next hop index to the to-be-deleted prefixes. It can be done using the same updating mechanism described above. In the complete deletion, if the structure of the tree changes, the BST must be rebuilt, and the entire memory content of each pipeline stage must be reloaded.

In the third type of update new prefixes are inserted into the existing routing table. A TCAM-like buffer is used to store a small number of new prefixes, as shown in Figure 5. If the new prefix changes the structure of the BST, it is added into the buffer, or into the BST otherwise. When the buffer is full, the BST must be rebuilt and repopulated.

#### 4.4 Scalability

The use of the complete BST structure leads to a linear storage complexity in our design, as each node contains exactly one prefix. The height of a complete BST is  $(1 + \lfloor \log_2 N \rfloor)$ , where  $N$  is the number of nodes. Since each level of the BST is mapped to a pipeline stage, the height of the BST determines the number of stages. Our proposed architecture is simple, and is expected to utilize a small amount of logic resource. Hence, the major constraint that dictates the number of pipeline stages, and in turn the size of supported routing table, is the amount of on-chip memory. As mentioned before, the memory size doubles as

we go from one level to the next of a complete BST. Therefore, we can move the largest stages onto external SRAM. Consequently, for each additional stage on external SRAM, the size of the supported routing table is doubled. However, due to the limit on the number of I/O pins of FPGA devices, we can fit only a certain number of stages on SRAM, as described in detail in Section 5.2.

The scalability of our architecture relies on the close relationship between the size of the routing tables and the number of required pipeline stages. As the number of prefixes increases, extra pipeline stages are needed. To avoid reprogramming the FPGA, we can allocate the maximum possible number of pipeline stages and use only what we need. The only draw back is that this approach introduces more latency (by the extra number of pipeline stages).

## 5 Implementation

### 5.1 Without External SRAM

As stated earlier, the memory size in each stage doubles that of the previous stage if they are full. Therefore, Stage 0 has one entry, Stage 1 has two entries,  $\dots$ , Stage  $i$  has  $2^i$  entries. Each entry includes: a prefix (length of  $n$ ), and its next hop index (6 bits). That makes a total of  $(n + 6)$  bits per entry. On a Xilinx FPGA, BRAM comes in blocks of 18 Kb. The distribution of the number of prefixes, number of pipeline stages, and the required BRAMs in each group of prefixes are shown in Table 5. Our target chip is a large state-of-the-art device. The Virtex-4 FX140, has 9936 Kb of BRAM on chip, or 552 blocks. Our design utilizes 497 memory blocks, or about 90% of the available memory, to support routing table `rrc7` (248856 prefixes). With the maximum utilization of on-chip memory, the design can support a routing table of up to 260K prefixes. Using the Virtex-5 FX200T (with 16 Mb or 912 BRAMs) as our target chip, we can scale up the design to support over 400K prefixes.

### 5.2 With External SRAM

In our design, external SRAMs can be used to handle even larger routing tables, by moving the last stages of the pipelines onto external SRAMs. Currently, SRAM is available in 2 – 32 Mb chips [4], with data widths of 18, 32, or 36 bits, and a maximum access frequency of over 400MHz. Each stage uses dual port memory, which requires two address and two data ports. In general, stage  $i$  of pipeline

**Table 5:** Number of required BRAMs for each group of prefixes

| Group of prefixes | (1,14) | (15,16) | (17,18) | (19,20) | (21,22) | (23,24) | (25,26) | (27,28) | (29,30) | (31,32) |
|-------------------|--------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| # prefixes        | 3592   | 11685   | 15304   | 46601   | 52109   | 168284  | 3258    | 2041    | 6065    | 3583    |
| # pipeline stages |        | 14      | 14      | 16      | 16      | 18      | 12      | 11      | 13      | 12      |
| # BRAM            | 8      | 15      | 21      | 67      | 80      | 275     | 7       | 4       | 12      | 8       |

**Table 6:** Number of required BRAMs and SRAMs for each group of prefixes

| Group of prefixes          | (1,14) | (15,16) | (17,18) | (19,20) | (21,22) | (23,24) | (25,26) | (27,28) | (29,30) | (31,32) |
|----------------------------|--------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| # internal/external stages | 0/0    | 16/0    | 17/0    | 17/2    | 15/4    | 17/4    | 14/0    | 13/0    | 15/0    | 14/0    |
| # BRAM                     | 8      | 79      | 171     | 185     | 50      | 214     | 29      | 31      | 64      | 34      |
| Amount of SRAM (Mb)        | 0      | 0       | 0       | 13.6    | 14.7    | 62.9    | 0       | 0       | 0       | 0       |

$(n, n + 1)$  requires  $2 \times i$  (address) +  $2 \times (n + 7)$  (data) pins going into the FPGA chip. For instance, stage 15 of pipeline (15,16) requires 76 pins. Similarly, stage 15 of pipeline (17,18) needs 80 pins. The largest Virtex package, which has 1517 I/O pins, can interface easily with 10 banks of dual port SRAMs. Using this package, we can add extra stage to each pipeline, and hence, double the size of the supported routing table. Moreover, since the access frequency of SRAM is twice that of our targeting frequency (200 MHz), the use of external SRAM should not adversely affect the performance of our design. Table 6 describes the relationship between the groups of prefixes, the amount of on-chip BRAM (using Virtex-5 FX200T), and external SRAM needed. This configuration can support a routing table consisting of over 1.7M prefixes.

## 6 Power Optimization

### 6.1 Memory Activation

As mentioned before, in our design, the power dissipation in the memory dominates that in the logic. Therefore, reducing power consumption in memory contributes to a large reduction in the total power consumption. Note that a BRAM consumes power as soon as it is enabled and clocked, even if no input signals are changing. In each stage of the pipeline, more than one BRAM is used. However, only one BRAM is accessed per clock cycle. This observation suggests a way to reduce memory power consumption by turning off the BRAMs that are not being accessed. We can easily achieve this by using a decoder to manually control each individual BRAM. We use the upper bits of the memory address to control the decoder, and the lower bits to access the individual BRAM. The number of lower bits depends on the data width of the memory. Let  $N$  be the number of nodes and  $N_B$  be the number of nodes per BRAM, the power saving factor can be calculated as  $N/(\log_2 N \times N_B)$ . The additional logic used to build these decoders is justified when considering the amount of power that can be saved.

### 6.2 Cross-Pipeline Optimization

We can further reduce power consumption across the pipelines using the longest prefix matching property. If a

**Table 7:** Implementation results

| Min. clock period | Max. frequency | # BRAM | Logic   |
|-------------------|----------------|--------|---------|
| 5.875 ns          | 170 MHz        | 497    | 26 %    |
| Power consumption |                | 1.59 W | 0.573 W |

match with length  $n$  has been found, there is no need to find matches with lengths less than  $n$ . Hence, in Figure 5, if pipeline  $i$  has matched a prefix at stage  $j$ , the subsequent stage traversals in pipelines 1 to  $i$  are not necessary, and therefore, can be turned off to save power.

In the worst case scenario, all the pipelines must traverse to the last stages to find a match. In this case, we have 126 active stages, each requiring one active BRAM, for the total of 126 active BRAMs. Using Xilinx XPower with the running clock frequency of 200 MHz, the power consumption of a 18 Kb-BRAM is recorded as 14.83 mW/200 MHz. With this result, the total power consumption of all BRAMs in the design is only 1.87 W.

The same technique can also be applied to reduce the power consumption of external SRAM. The memory controller for SRAM can either be built using on-chip logic resource or using external hardware. However, to take the burden off the internal resource, and to utilize the widely available memory controller in the market, an external approach is preferable.

## 7 Implementation Results

### 7.1 Throughput and Power Consumption

We implemented the proposed architecture in Verilog, using Synplify Pro 9.6.2 and Xilinx ISE 10.1.3, with Virtex-4 FX140 as the target. The results are shown in Table 7. The implementation showed a minimum clock period of 5.875 ns, or a maximum frequency of 170 MHz. With a dual pipeline architecture, this design can achieve a throughput of 340 MLPS, or 109 Gbps. Throughout this paper, throughput in Gpps is calculated based on a minimum packet size of 40 bytes (or 320 bits). We also performed a detailed analysis on power consumption using Xilinx XPower. The BRAM power consumption reported is lower than our analysis of 1.87 W due to a slower clock frequency (170 MHz vs. 200 MHz).

**Table 8:** Performance comparison

| Architecture | # slices     | # BRAMs | # prefixes | Throughput |
|--------------|--------------|---------|------------|------------|
| 1 ([17, 18]) | 1405(2.3%)   | 530     | 80K        | 125 MLPS   |
| 2 ([16])     | 14274(22.7%) | 254     | 80K        | 263 MLPS   |
| 3 (USC)      | 16617(26%)   | 473     | 249K       | 340 MLPS   |

## 7.2 Performance Comparison

Two key comparisons were performed with respect to the size of supported routing table and throughput. The two candidates were (1) the Ring architecture [17, 18] and (2) the state of the art architecture on FPGA [16]. These architectures can support the largest routing tables to date and have the highest throughput. All the resource data were normalized to Virtex-4 FX140, as shown in Table 8. Note that power consumption comparison is not possible as it was not reported for these designs.

With a lookup rate of 340 MLPS, our design had higher throughput than Architecture 1 (125 MLPS) and Architecture 2 (263 MLPS). Using only BRAM, the proposed architecture outperformed the two architectures with respect to the size of the supported routing table (260K vs. 80K). Using Virtex-5 FX200T with 16 Mb or 912 BRAMs, our design supported up to 400K prefixes. Our architecture also supports fast, non-blocking incremental update at run time without any design tool involvement (as does Architecture 1). This update operation is done by inserting the write bubble into the traffic stream whenever there is an update. In contrast, Architecture 2 relies on partial reconfiguration, which requires modifying, synthesizing, and implementing the code to regenerate the partial bitstream. Additionally, our design allows in-order output packets. With regards to scalability, it can be partitioned to use BRAM+SRAM, as discussed in Section 5.2, to support larger routing tables of up to 1.7M prefixes. This can be done without sacrificing the sustained throughput.

## 8 Concluding Remarks

This paper proposed and implemented a scalable high throughput, low power SRAM-based pipeline architecture for IP lookup, that does not TCAM. By using a binary search tree algorithm, the address of the child node can be eliminated, resulting in an efficient memory utilization. Consequently, our architecture can support large routing tables of up to 400K prefixes, using only on-chip BRAM. This is 1.5 times the size of the current largest routing table (260K prefixes). Using external SRAM, this architecture can handle even larger routing tables consisting of over 1.7M prefixes. Our design sustained a lookup rate of 340 MLPS even when external SRAM is used. This throughput translates to 109 Gbps, which is  $2.7\times$  the speed of OC-768. This is the highest throughput reported for FPGA implementation. In addition, we introduced a power saving technique to reduce the memory power consumption.

The design also maintains the packet input order and supports nonblocking route update.

## References

- [1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, 2001.
- [2] RENESAS CAM [Online]. [http://www.renesas.com].
- [3] CYPRESS SRAMs [Online]. [http://www.cypress.com].
- [4] SAMSUNG SRAMs [Online]. [http://www.samsung.com].
- [5] M. J. Akhbarizadeh, M. Nourani, D. S. Vijayasarathi, and T. Balsara, "A non-redundant ternary CAM circuit for network search engines," *IEEE Trans. VLSI Syst.*, vol. 14, no. 3, pp. 268–278, 2006.
- [6] K. Zheng, C. Hu, H. Lu, and B. Liu, "A tcam-based distributed parallel IP lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 863–875, 2006.
- [7] F. Baboescu, S. Rajgopal, L. Huang, and N. Richardson, "Hardware implementation of a tree based IP lookup algorithm for OC-768 and beyond," in *Proc. DesignCon '05*, 2005, pp. 290–294.
- [8] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: fast and efficient IP lookup architecture," in *Proc. ANCS '06*, 2006, pp. 51–60.
- [9] L. Peng, W. Lu, and L. Duan, "Power efficient IP lookup with supernode caching," *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pp. 215–219, Nov. 2007.
- [10] Z. L. A. Kennedy, X. Wang and B. Liu, "Low power architecture for high speed packet classification," in *Proc. ANCS*, 2008.
- [11] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Comput. Syst.*, vol. 17, pp. 1–40, 1999.
- [12] S. Sahni and K. S. Kim, "An  $O(\log n)$  dynamic router-table design," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 351–363, 2004.
- [13] H. Lu and S. Sahni, " $O(\log n)$  dynamic router-tables for prefixes and ranges," *IEEE Transactions on Computers*, vol. 53, no. 10, pp. 1217–1230, 2004.
- [14] K. S. Kim and S. Sahni, "Efficient construction of pipelined multibit-trie router-tables," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 32–43, 2007.
- [15] H. Le, W. Jiang, and V. K. Prasanna, "A SRAM-based architecture for trie-based IP lookup using FPGA," in *Proc. FCCM '08*, 2008.
- [16] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for IP address lookup," in *Proc. ANCS '05*, 2005, pp. 81–90.
- [17] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Proc. ISCA '05*, 2005, pp. 123–133.
- [18] W. Jiang and V. K. Prasanna, "A memory-balanced linear pipeline architecture for trie-based IP lookup," in *Proc. HOTI '07*, 2007, pp. 83–90.
- [19] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed IP lookup algorithms," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 802–812, 2005.
- [20] M. Meribout and M. Motomura, "A new hardware algorithm for fast IP routing targeting programmable routers," in *Network control and engineering for Qos, security and mobility II*. Kluwer Academic Publishers, 2003, pp. 164–179.
- [21] RIS RAW DATA [Online]. [http://data.ris.ripe.net].
- [22] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," in *Proc. INFOCOM '03*, 2003, pp. 64–74.