Technical University of Crete

Department of Electronic & Computer Engineering

Microprocessor & Hardware Laboratory (MHL)

Diploma Thesis

Hardware for IPv6 Longest Prefix Matching

By Marios A. Eliofotou

Committee: Assoc. Professor D. Pnevmatikatos (Thesis Advisor)
Assoc. Professor A. Liavas
Assist. Professor M. Bucher

July 2005

Chania

Necessity, who is the mother of invention.
Plato, The Republic

Dedicated to my family

Acknowledgements

I would like to express my sincere gratitude and appreciation to my advisor Prof. D. Pnevmatikatos for his guidance and patience throughout the development of my thesis.

Table of Content

1. Introduction

Due to the rapid growth of traffic in the Internet, backbone links of several gigabits per second are commonly deployed. To handle gigabit-per-second traffic rates, the backbone routers must be able to forward millions of packets per second on each of their ports. Fast IP address look-up in the routers, which uses the packet's destination address to determine for each packet the next hop, is therefore crucial to achieve the packet forwarding rates required.

After the introduction of Classless Inter-Domain Routing (CIDR) in 1993, IP address look-up has become a much more difficult task. That is because now, a router must not only find an exact prefix match but it also has to choose the Longest Prefix Matching (LPM) the IP destination address. Moreover, nowadays the shortage of IP addresses is becoming an even more pressing issue. This has brought growing interest in the next generation internet protocol, known as IPv6. With 128-bit addresses, IPv6 provides 3.4×10^{38} addresses theoretically and it has been gaining wider acceptance to replace its predecessor, IPv4 [3]. However, the 128-bit address length has intensified the problem of routing millions of communication packets every second, based on longest prefix matching.

During our research, we have investigated a plethora of solutions proposed in order to solve the longest prefix matching predicament. For this thesis, our task was to choose and implement one of these solutions, which we believe is suitable for addressing the IPv6 address look-up problem. In the rest of this section, we will first introduce the reader with the concepts of IP addressing and routing, then we will present the next generation IP protocol (IPv6), discuss the evaluation metrics for look-up schemes and, finally, we conclude with a brief description of the solution we have chosen to implement.

1.1 IP Addressing and Routing

We begin by tracing the evolution of the IP addressing architecture. The addressing architecture is of fundamental importance to the routing architecture, and reviewing it will help us to understand the address look-up problem.

The Classful Addressing Scheme

One of the fundamental objectives of the Internet Protocol is to interconnect networks, so routing on a network basis was a natural choice (rather than routing on a host basis). Thus, the IP address scheme initially used a simple two-level hierarchy, with networks at the top level and hosts at the bottom level. This hierarchy is reflected in the fact that an IP address consists of two parts, a network part and a host part. The network part identifies the network to which a host is attached, and thus all hosts attached to the same network agree in the network part of their IP addresses. Since the network part corresponds to the first bits of the IP address, it is called the **address prefix**. We will write prefixes as bit strings of up to a specific length (e.g. 32 for IPv4 or 128 for IPv6) followed by a *. For example, the prefix 1000001001010110* represents all the 2^{16} addresses that begin with the bit pattern 1000001001010110 (for the 32-bit IPv4). Alternatively, prefixes can be indicated using the dotted-decimal notation, so the same prefix can be written as 130.86/16 (for IPv4), where the number after the slash indicates the length of the prefix.

Destination address prefix	Next hop	Output interface
24.40.32/20	192.41.177.148	2
130.86/16	192.41.177.181	6
208.12.16/20	192.41.177.241	4
208.12.21/24	192.41.177.196	1
167.24.103/24	192.41.177.3	4

Table 1.1: A forwarding table for IPv4

With a two-level hierarchy, IP routers forwarded packets based only on the network part, until packets reached the destination network. As a result, a forwarding table only needed to store a single entry to forward packets to all the hosts attached to the same network. This technique is called **address aggregation** and allows using prefixes to represent a group of addresses. Each entry in a forwarding table contains a prefix, as can be seen in Table 1.1 [2]. Thus, finding the forwarding information requires searching for the prefix in the forwarding table that matches the corresponding bits of the destination address.

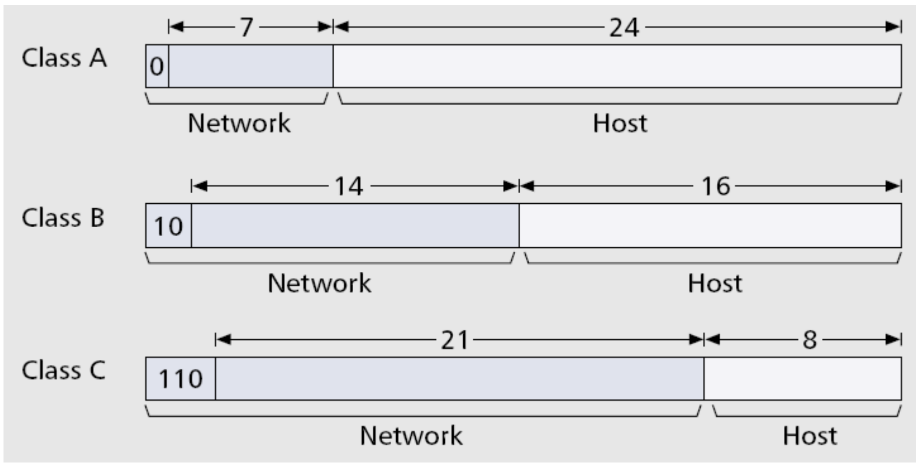


Figure 1.1: IPv4 classful addresses

The addressing architecture specifies how the allocation of addresses is performed; that is, it defines how to partition the total IP address space — specifically, how many network addresses will be allowed and what size each of them should be. When Internet addressing was initially designed, a rather simple address allocation scheme was defined, which is known today as the **classful addressing scheme**.

Basically, three different sizes of networks were defined in this scheme, identified by a class name: A, B, or C (Figure 1.1). Network size was determined by the number of bits used to represent the network and host parts. Thus, networks of class A, B, or C consisted of an 8, 16, or 24-bit network part and a corresponding 24, 16, or 8-bit host part.

With this scheme there were very few class A networks, and their addressing space represented 50 percent of the total IPv4 address space (2^{31} addresses out of a total of 2^{32}). There were 16,384 (2^{14}) class B networks with a maximum of 65,534 hosts/network, and 2,097,152 (2^{21}) class C networks with up to 256 hosts. This allocation scheme worked well in the early days of the Internet. However, the continuous growth of the number of hosts and networks has made apparent two problems with the classful addressing architecture. First, with only three different network sizes from which to choose, the address space was not used efficiently and the IP address space was getting exhausted very rapidly, even though only a small fraction of the addresses allocated were actually in use. Second, although the state information stored in the forwarding tables did not grow in proportion to the number of hosts, it still grew in proportion to the number of networks. This was especially important in the backbone routers, which must maintain an entry in the forwarding table for every allocated network address. As a result, the forwarding tables in the backbone routers grew very rapidly. The growth of the forwarding tables resulted in higher lookup times and higher memory requirements in the routers, and threatened to impact their forwarding capacity.

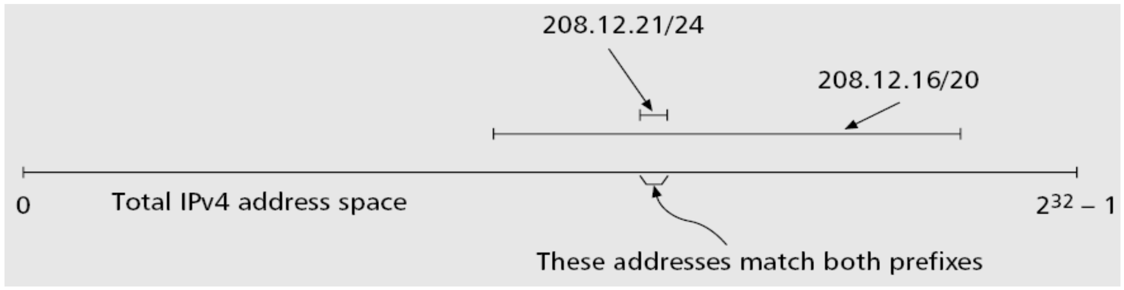


Figure 1.2: Prefix ranges for IPv4

Classless Inter-Domain Routing (CIDR)

In order to allow more efficient use of the IP address space as well as slow down the growth of the backbone forwarding tables, a new scheme called **classless inter-domain routing** (CIDR) was introduced in 1993 [2].

To address the problem of forwarding table explosion, CIDR allows address aggregation at several levels. In this way, we can recursively aggregate addresses at various points within the hierarchy of the Internet’s topology. As a result, backbone routers maintain forwarding information not at the network level, but at the level of arbitrary aggregates of networks. Thus, recursive address aggregation reduces the number of entries in the forwarding table of backbone routers [2]. To understand how this works, consider the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24 (Figure 1.2). Suppose that in a router all these network addresses are reachable through the same service provider. From the binary representation we can see that the leftmost 20 bits of all the addresses in this range are the same (11010000 00001100 0001). Thus, we can aggregate these 16 networks into one "super-network" represented by the 20-bit prefix, which in decimal notation gives 208.12.16/20. Note that indicating the prefix length is necessary in decimal notation, because the same value may be associated with prefixes of different lengths; for instance, 208.12.16/20 (11010000 00001100 0001*) is different from 208.12.16/22 (11010000 00001100 000100*).

While a great deal of aggregation can be achieved if addresses are carefully assigned, in some situations a few networks can interfere with the process of aggregation. For example, suppose now that a customer owning the network 208.12.21/24 changes its service provider and does not want to renumber its network. Now, all the networks from 208.12.16/24 through 208.12.31/24 can be reached through the same service provider, except for the network 208.12.21/24 (Figure 1.2). We cannot perform aggregation as before, and instead of only one entry, 16 entries need to be stored in the forwarding table. One solution that can be used in this situation is aggregating in spite of the exception networks and additionally storing entries for the exception networks. In our example, this will result in only two entries in the forwarding table: 208.12.16/20 and 208.12.21/24 (Figure 1.2 and Table 1.1). Note, however, that now some addresses will match both entries because prefixes overlap. In order to always make the correct forwarding decision, routers need to do more than to search for a prefix that matches. Since exceptions in the aggregations may exist, a router must find the most specific match, which is the **longest prefix matching (LPM)** the IP destination address of the datagram. In summary, the address lookup problem in routers requires searching the forwarding table for the longest prefix that matches the destination address of a packet.

The Complexity of Longest Prefix Matching Searches

In the classful addressing architecture, the length of the prefixes was coded in the most significant bits of an IP address (Figure 1.1), and the address lookup was a relatively simple operation: Prefixes in the forwarding table were organized in three separate tables, one for each of the three allowed lengths. The lookup operation amounted to finding an exact prefix match in the appropriate table. The search for an exact match could be performed using standard algorithms based on hashing or binary search. While CIDR allows the size of the forwarding tables to be reduced, the address lookup problem now becomes more complex. With CIDR, the destination prefixes in the forwarding tables have arbitrary lengths and no longer correspond to the network part since they are the result of an arbitrary number of network aggregations. Therefore, when using CIDR, the search in a forwarding table can no longer be performed by exact matching because the length of the prefix cannot be derived from the address itself. As a result, determining the longest matching prefix involves not only comparing the bit pattern itself, but also finding the appropriate length. Therefore, we talk about searching in two dimensions: value and length.

1.2 The Next Generation Internet Protocol (IPv6)

Due to the fast growth of the Internet, the shortage of IP addresses is becoming a more pressing issue. This has brought growing interest in the next generation internet protocol, known as IPv6. With 128-bit address, IPv6 provides 3.4 x 10^{38} addresses theoretically and it has been gaining wider acceptance to replace its predecessor, IPv4. IPv6 has already emerged out of the testing phase and is seeing early deployment in Europe, Asia, and North America [3].

IPv6 Address Allocation

Because of the large size of the available address space in IPv6, a hierarchical structure of the allocation space is necessary to permit the aggregation of routing information and also to limit the expansion of Internet routing tables. This goal is particularly important in IPv6 addressing, where the size of the total address pool creates significant implications for both internal and external routing. IPv6 address policies should seek to avoid fragmentation of address ranges. Further, RIRs (Regional Internet Registries) should apply practices that maximize the potential for subsequent allocations to be made contiguous with past allocations currently held. However, there can be no guarantee of contiguous allocation. It must be noted that IP addresses will be allocated and assigned on a license basis, with licenses subject to renewal on a periodic basis. The granting of a license is subject to specific conditions applied at the start or renewal of the license. RIRs will generally renew licenses automatically, provided requesting organizations are making a good-faith effort at meeting the criteria under which they qualified for or were granted an allocation or assignment. However, in those cases where a requesting organization is not using the address space as intended, or is showing bad faith in following through on the associated obligation, RIRs reserve the right to not renew the license. In IPv6 address policy, the goal of aggregation is considered to be the most important [4].

Under the current IPv6 Address Allocation and Assignment Policy, the Regional Internet Registries (RIR), receive /23's from the Internet Assigned Numbers Authority (IANA). In turn, the RIRs assign addresses to Local Internet Registries (LIR) or ISPs with a minimum allocation size of /32. In Asia, an extra layer of National Internet Registry (NIR) exists between the RIR and the LIRs. In the general case, end users, such as different organizations and small ISPs, are assigned /48's from the LIR/ISPs. /64 is assigned when it is known that one and only one subnet is needed and /128 is assigned when it is absolutely known that one and only one device is connecting. When a LIR/ISP achieves sufficient address utilization, subsequent allocation for additional address space will be provided. As shown in Figure 1.3, the lower half of the 128-bit IPv6 address is assigned the interface ID. The same allocation policy specifies using the MAC address for this field [3, 4]. Thus, for most cases, only the first 64 bits of IPv6 address are used for routing in the network. In this thesis, we focus on global unicast addresses all of which start with the first three bits 001. These addresses are aggregatable with contiguous bit-wise mask similar to IPv4 under CIDR [5].

Figure 1.3 shows the format for the IPv6 global unicast address [5]. This address format was designed to facilitate scalable Internet routing, by providing an address hierarchy flow aggregation. The address format has a fixed structure as shown in Figure 1.3 and is organized into a three level hierarchy: public topology; site topology; and interface identifier. The public topology consists of a two level hierarchy of service providers with a top-level aggregation identifier (TLA ID) and a next-level aggregation identifier (NLA ID). The TLA ID is initially to be restricted to 13 bits which translates to 8192 routers in the core IPv6 network. This was done to constrain core routing table sizes. The NLA ID is 24–bits long and allows for a flat or hierarchical allocation of the NLA address space. The site-level aggregation identifier (SLA ID) is 16-bits long. It is used by an individual organization to define its local address hierarchy and subnets [6].

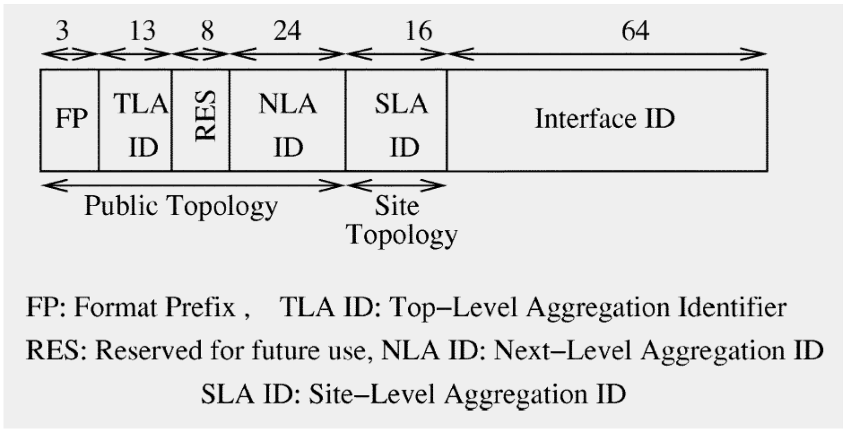

3
13
8
24
16
64
FP
TLA ID
RES
NLA ID
SLA ID
Interface ID
Public Topology
Site Topology
FP: Format Prefix , TLA ID: Top–Level Aggregation Identifier
RES: Reserved for future use, NLA ID: Next–Level Aggregation ID
SLA ID: Site–Level Aggregation ID

Figure 1.3: Format of the aggregatable global unicast address for IPv6

IPv6 Routing Tables

In order to design and evaluate any IPv6 look-up scheme, IPv6 tables are needed. Wang et al. [3], in their work, stated that the analysis of existing IPv4 tables reveals that there is an underlying structure which differs greatly from random distributions. Moreover, since there are few users on IPv6 at present, current IPv6 table sizes are small and unlikely to reflect future IPv6 network growth. Thus, neither randomly generated tables nor current IPv6 table are good benchmarks for analysis. More representative IPv6 lookup tables are needed for the development of IPv6 routers. Therefore, Wang et al. from the analysis of current IPv4 tables, proposed some algorithms for generating IPv6 look-up tables. Tables generated by the methods they have suggested, exhibited certain features characteristic of real look-up tables, reflecting not only new IPv6 address allocation schemes but also patterns common to IPv4 tables. We assert that these tables provide useful research tools by a better representation of future look-up tables as IPv6 becomes more widely deployed.

Wang et al. work was based on the assertion that although IPv6 allows better addressing structure and provides enhancements over IPv4, many of the features observed in IPv4 routing tables are expected to emerge in IPv6 routing tables. This is due to three main reasons: allocation policies, routing practices, and the evolution of the Internet. These are the three key areas that affect the formation of the IP table structure. First of all, the allocation policies for IPv6 follow the similar fundamentals of IPv4 allocation, despite the differences in detailed rules. Secondly, the overall network topological distribution, which affects routing, is expected to be intact through IPv4 to IPv6 migration. Thirdly, the natural evolution of the Internet will continue. The business relationship among IP prefix providers and the customers remain the same: the same companies that offer IPv4 services will provide IPv6 services. Thus, we expect that similar structures will be developed according to IPv6 allocation schemes for the IPv6 Internet. Figure 1.4 shows the prefix length distribution of a synthesized IPv6 table. All of our design selection as well as simulations, where made using this distribution.

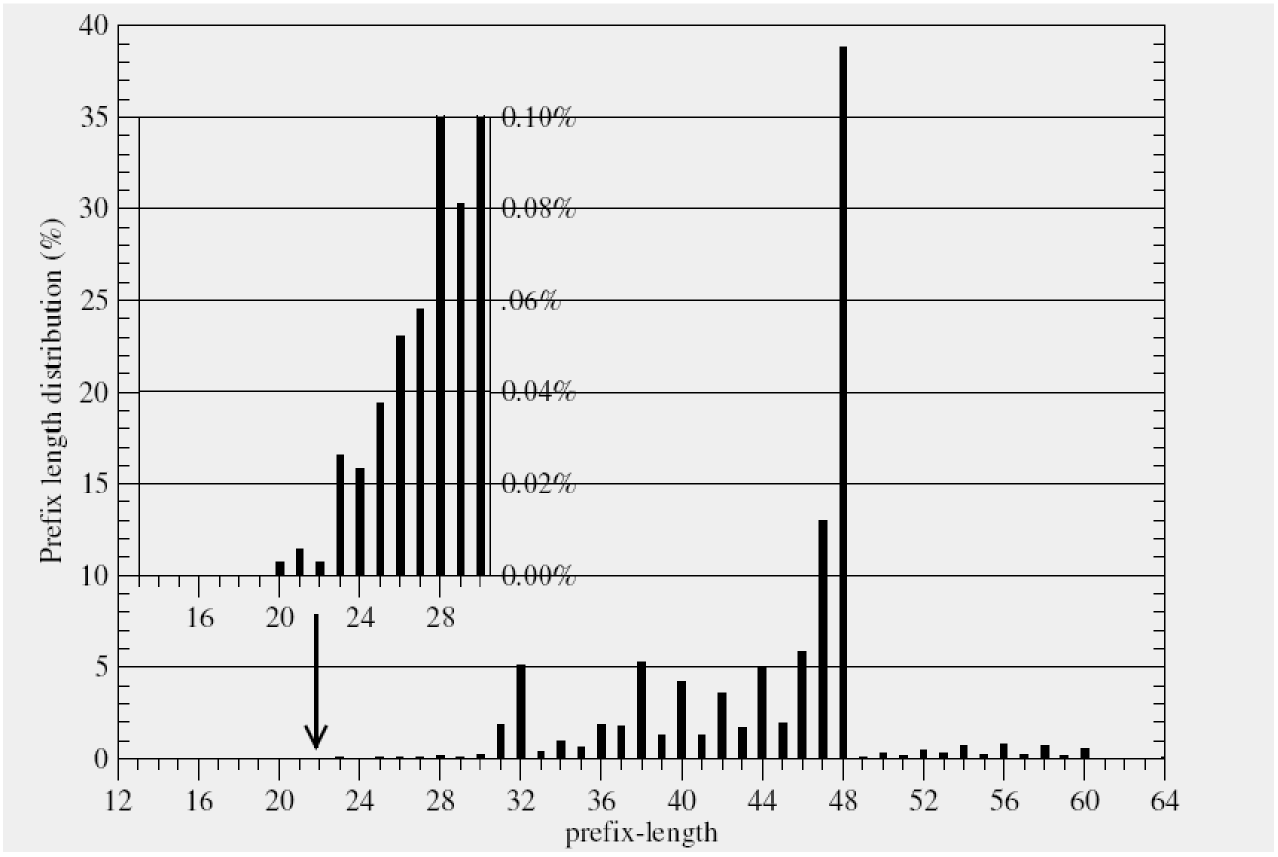


Figure 1.4: Prefix length distribution for a synthesized IPv6 table [3]

Concerning the size of an IPv6 table, we believe that is not clear how it will evolve compared with the current IPv4 table size. On one hand, IPv6 is designed to provide more aggregation through proper allocation schemes to prevent the explosion of routing table size if not reducing it. On the other hand, multi-homing is contributing 20-30% of the prefixes in current IPv4 routing tables. Multi-homing is expected to increase in IPv6, which makes controlling the routing table size difficult. It is a challenging task to design a lookup algorithm with good performance that scales with both longer prefix lengths and larger table sizes. It will take time for IPv6 table size to catch up with the current IPv4 table size. For studies within the near future, it is reasonable to use a table size for IPv6 that lies between 1x and 2x of today's IPv4 table size. Figure 1.5 shows the growth of the IPv4 tables' sizes form 1994 to present (July 2005) [7]. Given that the largest existing IPv4 table has around 200,000 entries, we have designed our look-up scheme targeting a routing table with 200,000 up to 500,000 entries (prefixes). In Figure 1.5 we show the growth of the largest existing IPv6 table (Telstra) with 759 entries; 107 entries larger than 6Net and 652 entries larger than 6Bone [7].

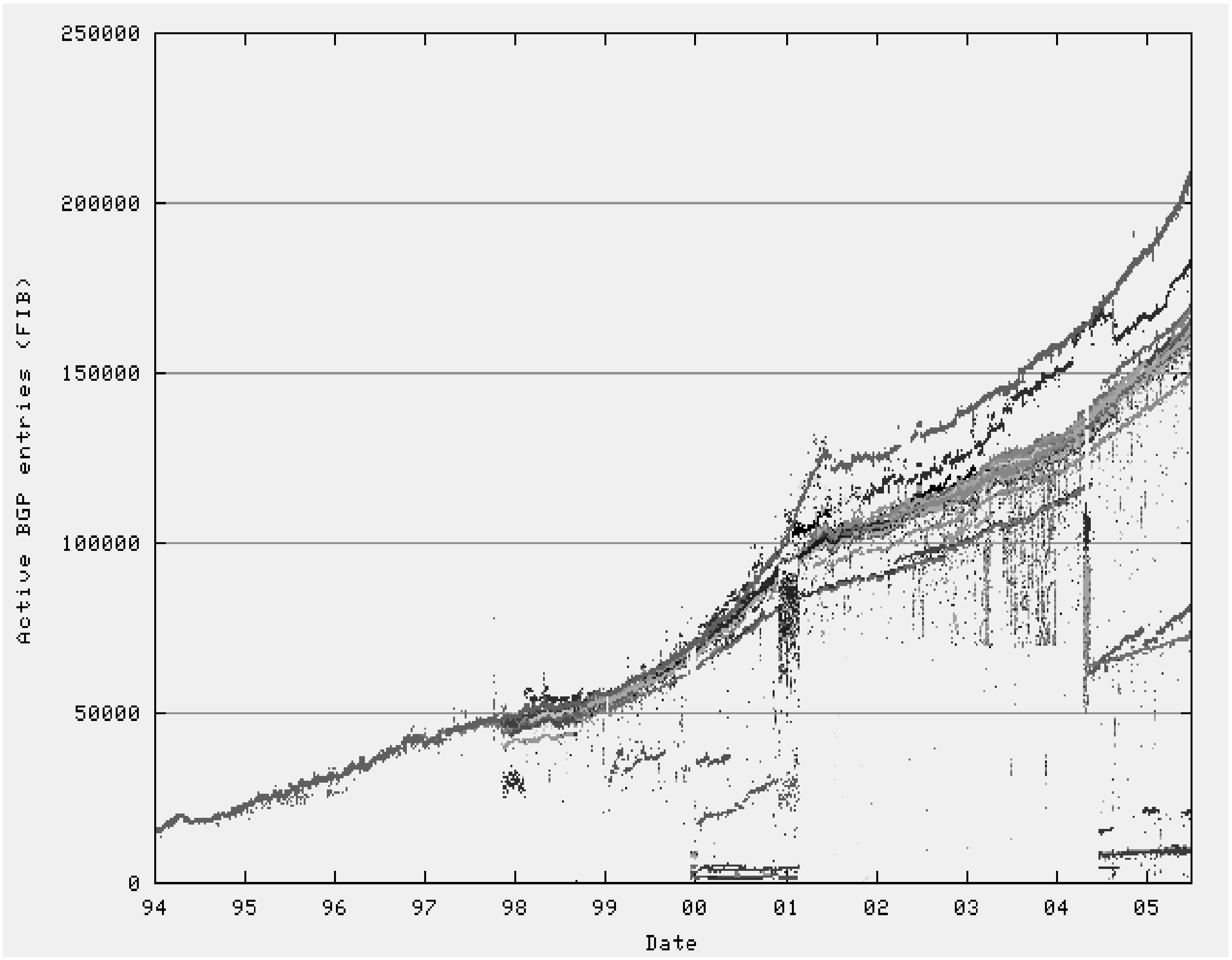


Figure 1.5 Growth of the IPv4 tables' sizes form 1994 to present (July 2005) [7]

1.3 Evaluation Metrics for IP Look-Up Schemes

The metrics taken into consideration, in general, while designing a IP look-up scheme are *look-up rate*, *update time* and *storage requirements*.

Look-up Rate

Lookup rate is the most significant parameter that needs to be addressed. With the latest advancements in the network technology, the communication speed is leaping from Ethernet of 10 Mb/s to fiber distributed-data interface (FDDI) of 100 Mb/s to gigabit Ethernet. With the Optical Carrier OC-192c Line (Line-rate 10 Gb/s), 25 million packets (average size of 50 Bytes for IPv6) have to be processed each second, while for the OC-768c (Line-rate 40 Gb/s), the processing rate required is 100 million packets per second. The data throughput rates of various transmission links and the corresponding time budget for packet processing in a network processor are shown in Table 1.2. It is significant to note that, apart from the look-up and forwarding operation the packet processing in a network processor includes various other functions like *protocol recognition and classification*, *segmentation assembly and reassembly* (SAR), *queuing and access control*, and *quality-of-service* (QoS) [6]. The time budget shown for packet processing includes the execution of all these functions, and the look-up operation is required to consume a portion of that budget. Hence, the significance of designing a mechanism for high-speed IP address lookups cannot be overemphasized.

Media	Link Rate	Packets / second (Million)	Time allotted for each packet (ns)
OC-3	~150 Mbps	~0.375	~ 2,666.7
OC-12	625 Mbps	~1.563	640
OC-48	2.5 Gbps	~6.25	160
OC-192	10 Gbps	25	40
OC-768	40 Gbps	100	10

Table 1.2: Data Throughput and datagram (IPv6) processing time budgets for ATM over SONET

Update Time

Another characteristic of the routing environment is that a forwarding table needs to be updated dynamically to reflect route changes. In fact, instabilities in the backbone routing protocols can fairly frequently change the entries in a forwarding table. Labovitz [8] found that backbone routers may receive bursts of route changes at rates exceeding several hundred prefix updates per second. He also found that, on average, route changes occur 100 times/s. Thus, update operations must be performed in 10 ms or less. Table 1.3 shows the types of updates to be supported and their frequency of occurrence [7].

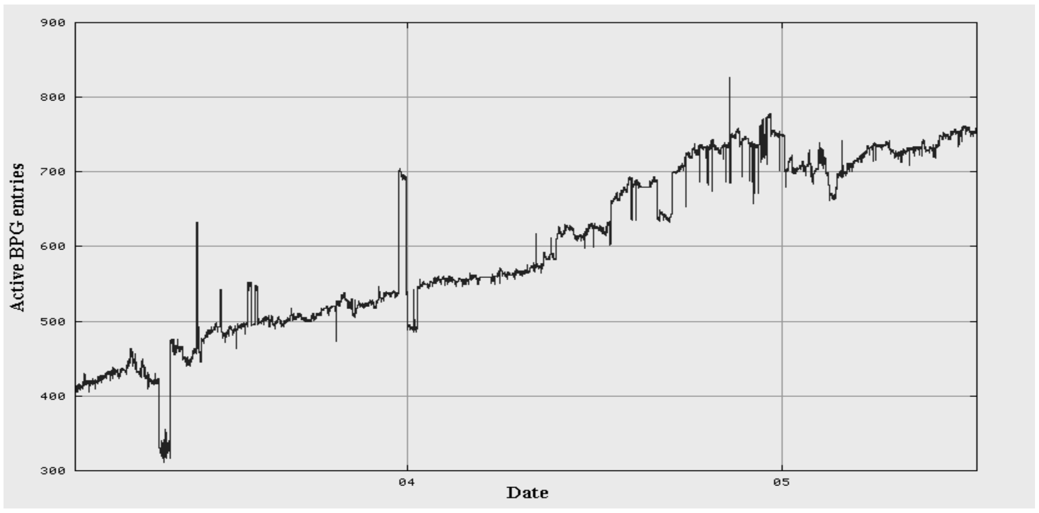


Figure 1.6: Growth of the IPv6 Telstra router from 2003 to present (July 2005) [7]

However, due to the bursty nature of route updates, new estimations state that a router must support up to 5,000 updates per second. In addition, it is noted that most routing updates are route flaps – that is, the same prefix is added and then removed repeatedly in quick successions. It is thus straightforward to reduce the number of actual updates by keeping a buffer of recent route update announcements and updating only the end result. This eliminates most of the route flaps [9].

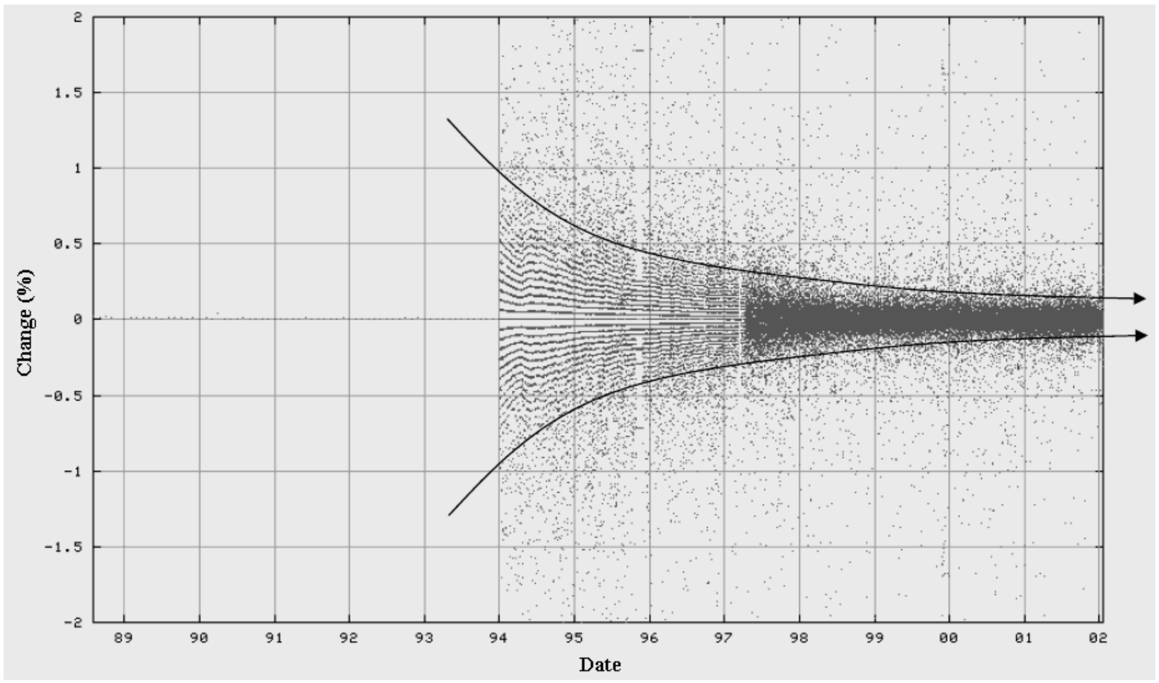


Figure 1.7: Hourly BGP (Border Gateway Protocol) Update Rates (%) [7]

Figure 1.7 illustrates the hourly BGP (Border Gateway Protocol, is the de-facto inter-AS routing protocol in the internet) update rates. As we can see, the portion of BGP table entries updated each hour is decreasing over time. The BGP tables are becoming more stable mostly due to (a) protocol implementation maturity (b) widespread deployment of flap damping (c) greater levels of circuit reliability [7]. However, there are insufficient data to determine if this is a short term growth correction that will be followed by a resumption of exponential growth. This is due to the introduction of Multi-homing, mobility, and various dynamic path controllers which all contribute to a continuing pressure for non-aggregated atomic entries to be externally advertised. Even though these alternations are not to be taken lightly, until present the number of updates per second is remaining constant.

Update Type	(%)
Announce	16
Withdraw	72
Path Change	12

Table 1.3: Types of Route Updates [7]

Storage Requirement

Storage requirement is becoming even more important for modern routers. As we can see from Figure 1.5, the growth of the IPv4 routing tables is increasing exponentially. This is something we expect to happen and with IPv6 tables. Therefore, look-up schemes must support scalability to both the number and length of prefixes. Moreover, from Table 1.2, we can see that the time allotted for packet processing is reducing. It is hence apparent that memory resources must be both amble and with low latencies. Table 1.4 summarizes the costs, power dissipations and latencies for the three basic storage elements used in packet forwarding schemes [10]. (Note: Price, speed and power are manufacturer and market dependent).

Technology	**Single chip density**	**$/chip ($/MB)**	**Access delay**	**Watts/chip**
Networking DRAM	128 MB	$60-$100 ($0.50-$0.80)	40-80ns	0.5-2W
SRAM	18 MB	$90-$150 ($5-$8)	3-8ns	1-3W
TCAM	1 MB	$200-$250 ($200-$250)	10-15ns	15-30W

Table 1.4: Memory Technology (2005) [10]

1.4 Implemented Look-up Scheme

In this thesis, we have chosen to implement an algorithm proposed by Dharmapurikar et al. [1]. In their work they have employed Bloom filters for IPv4 Longest Prefix Matching. A Bloom filter is an efficient data structure for membership queries with tunable false positive errors [11]. The probability of a false positive is dependent upon the number of entries stored in a filter, the size of the filter, and the number of hash functions used to probe the filter. Background on Bloom filter theory is presented in Section 3. Their algorithm begins by sorting the forwarding table entries by prefix length, associating a Bloom filter with each unique prefix length, and "programming" each Bloom filter with prefixes of its associated length. A search begins by performing parallel membership queries to the Bloom filters by using the appropriate segments of the input IP address. The result of this step is a vector of matching prefix lengths, some of which may be false matches. Hash tables corresponding to each prefix length are probed in the order of longest match in the vector to shortest match in the vector, terminating when a match is found or all of the lengths represented in the vector are searched. The key feature of this technique is that the performance, as determined by the number of dependent memory accesses per look-up, can be held constant for longer address lengths (IPv6) as well as for more prefixes given that memory resources scale linearly with the number of prefixes in the forwarding table. This is due to a property of Bloom filters where the bits used to store a key, with a predefined false positive probability, is independent of the number of bits of the key.

The rest of the thesis is organized as follows. In Section 2, we present related work for solving the LPM problem. An overview of our technique as well as a performance analysis is provided in Section 3. In Section4, we show how we have implemented this look-up scheme using FPGA technology. In Section 5 we report simulation results and the FPGA's resources consumed. In addition, in Section 5, we show that implementation with current technology is capable of average performance of over 250M look-ups per second using a commodity SRAM device operating at 300 MHz. We assert that this approach offers better performance, scalability, and lower cost than TCAMs, given that commodity SRAM devices are denser, cheaper, and operate more than three times faster than TCAMs (Table 1.4). Finally, our conclusions as well as future work are given in Section 6.

2. Related Work

Due to its importance, IP Address Look-up is a well studied field of Internet's technology. Hence, during our research, we have encountered a plethora of proposed ideas for solving this problem. In general, we can divide these methodologies into two categories. The first address the problem using an algorithmic approach, whereas the second uses massive parallel searches with hardware. In this section we will introduce the reader with some of the methods we believe are the most important and that they will make him/her understand why we have chosen to implement the particular IP look-up scheme.

2.1 Algorithmic Approaches

Binary Tries

The Binary Trie is the classical solution to the longest prefix matching (LPM) problem. A trie is a tree-based data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching [2]. Figure 2.1 shows a binary trie (each node has at most two children) representing a set of prefixes of a forwarding table. In a trie, a node on level *l* represents the set of all addresses that begin with the sequence of *l* bits consisting of the string of bits labeling the path from the root to that node. For example, node c of Figure 2.1 is at level 3 and represents all addresses beginning with the sequence 011. The nodes that correspond to prefixes are shown in a darker shade; these nodes will contain the forwarding information or a pointer to it. Note also that prefixes are not only located at leaves but also at some internal nodes. This situation arises because of exceptions in the aggregation process. For example, in Figure 2.1 the prefixes b and c represent exceptions to prefix a.

Tries allows finding, in a straightforward way, the longest prefix that matches a given destination address. The search in a trie is guided by the bits of the destination address. At each node, the search proceeds to the left or right according to the sequential inspection of the address bits. While traversing the trie, every time we visit a node marked as prefix (i.e., a dark node) we remember this prefix as the best match found so far. The search ends when there are no more branches to take, and the longest or best matching prefix will be the last prefix remembered. For instance, if we search the longest prefix match (LPM) for an address beginning with the bit pattern 10110 we start at the root in Figure 2.1. Since the first bit of the address is 1 we move to the right, to the node marked as prefix d, and we remember d as the LPM found so far. Then we move to the left since the second address bit is 0; this time the node is not marked as a prefix, so d is still the LPM found so far. Next, the third address bit is 1, but at this point there is no branch labeled 1, so the search ends and the last remembered LPM (prefix d) is the longest matching prefix.

The update operations are also straightforward to implement in binary tries. Inserting a prefix begins by doing a search. When arriving at a node with no branch to take, we can insert the necessary nodes.

Deleting a prefix starts again by a search, unmarking the node as prefix and, if necessary deleting unused node (i.e., leaves nodes not marked as prefixes). Note finally that since the bit strings of prefixes are represented by the structure of the trie, the nodes marked as prefixes do not need to store the bit strings themselves.

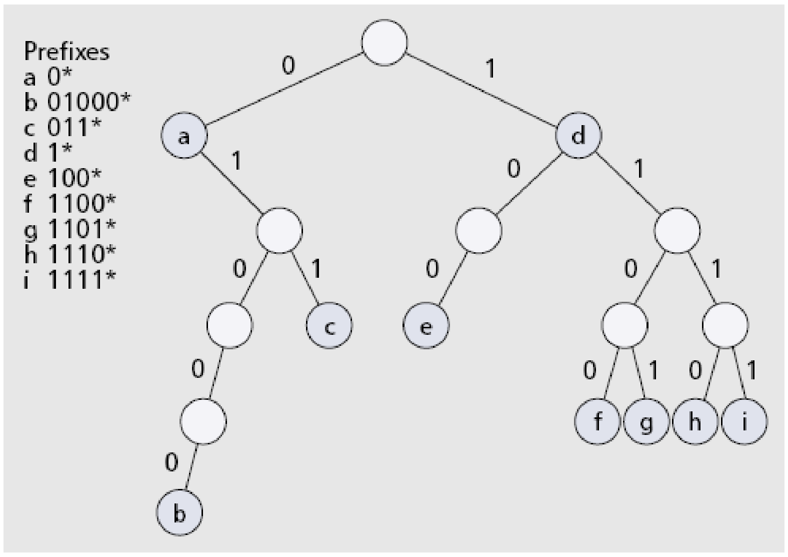


Figure 2.1: A binary trie for a given set of prefixes

Path-Compressed Tries

Even though binary tries allow the representation of arbitrary-length prefixes, they have the characteristic that long sequences of one-child nodes may exist. Since these bits need to be inspected, even though no actual branching decision is made, search time can be longer than necessary in some cases. Also, one-child nodes consume additional memory. In an attempt to improve time and space performance, a technique called path compression can be used.

Figure 2.2 illustrates a compressed version of the binary trie of Figure 2.1. Note that the two nodes preceding b now have been removed. Note also that since prefix a was located at a one-child node, it has been moved to the nearest descendant that is not a one-child node. However, since in a path to be compressed several one-child nodes may contain prefixes, in general, a list of prefixes must be maintained in some of the nodes. Because one-way branch nodes are now removed, we can jump directly to the bit where a significant decision is to be made, bypassing the bit inspection of some bits. As a result, a bit number field must be kept now to indicate which bit is the next bit to inspect. In Figure 2.2 these bit numbers are shown next to the nodes. Moreover, the bit strings of prefixes must be explicitly stored. A search in this kind of path-compressed trie is as follows. The algorithm performs, as usual, a descent in the trie under the guidance of the address bits, but this time only inspecting bit positions indicated by the bit-number field in the nodes traversed. When a node marked as a prefix is encountered, a comparison with the actual prefix value is performed. This is necessary since during the descent in the trie we may skip some bits. If a match is found, we proceed traversing the trie and keep the prefix as the LPM so far. The search ends when a leaf is encountered or a mismatch found. As usual, the LPM will be the last matching prefix encountered. For instance, if we look for the LPM of an address beginning with the bit pattern 010110 in the path-compressed trie shown in Figure 2.2, we proceed as follows. We start at the root node and, since its bit number is 1, we inspect the first bit of the address. The first bit is 0, so we go to the left. Since the node is marked as a prefix, we compare prefix a with the corresponding part of the address (0). Since they match, we proceed and keep a as the LPM so far. Since the node's bit number is 3, we skip the second bit of the address and inspect the third one. This bit is 0, so we go to the left. Again, we check whether the prefix b matches the corresponding part of the address (01011). Since they do not match, the search stops, and the last remembered LPM (prefix a) is the correct LPM.

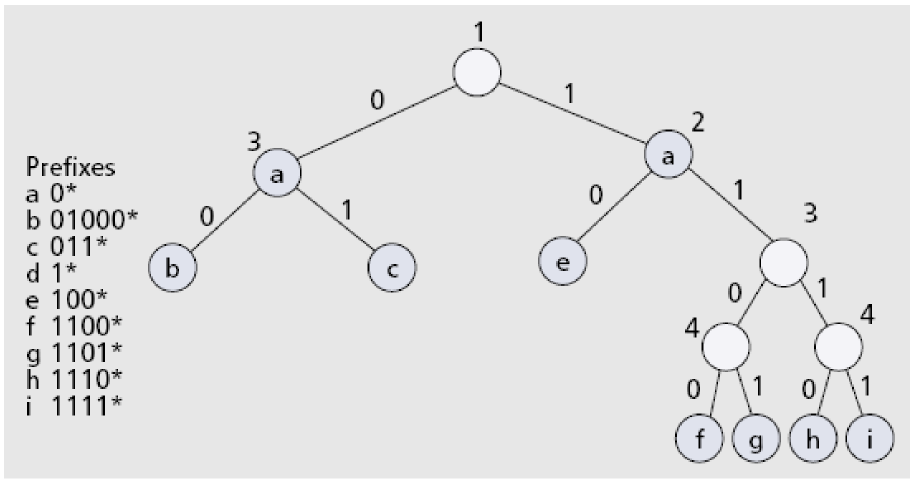


Figure 2.2: A Path-Compressed Trie

Until some years ago, the longest matching prefix problem was addressed by using data structures based on path-compressed tries. Path compression makes a lot of sense when the binary trie is sparsely populated. On the other hand, when the number of prefixes increases and the trie gets denser, using path compression has little benefit. Moreover, the principal disadvantage of path-compressed tries, as well as binary tries in general, is that a search needs to do many memory accesses, in the worst case 64 for IPv6 addresses. Therefore, Path-Compressed Tries, even though reduce the average number of required memory accesses compared to conventional Binary Tries are not suitable for today's forwarding demands.

Multi-bit Tries

In order to speed up the lookup process, multi-bit trie schemes were developed which perform a search using multiple bits of the address at a time [2]. For instance, if we inspect 8 bits at a time we would need 8 memory accesses in the worst case for an IPv6 address. The number of bits to be inspected per step is called **stride** and can be constant or variable. A trie structure that allows the inspection of bits in strides of several bits is called a multi-bit trie. Thus, a multi-bit trie is a trie where each node has 2^k children, where *k* is the stride.

Since multi-bit tries allow the data structure to be traversed in strides of several bits at a time, they cannot support arbitrary prefix lengths. To use a given multi-bit trie, the prefix set must be transformed into an equivalent set with the prefix lengths allowed by the new structure. This can be achieved by using a technique known as **Control Prefix Expansion** (CPE) [2]. For instance, a multi-bit trie corresponding to our example from Figure 2.1 is shown in Figure 2.3. Note that the height of the trie has decreased, and so has the number of memory accesses when doing a search. We can see that prefixes a and d have been expanded to length 3. However, two of the prefixes produced by expansion already exist (prefixes c and e). We must preserve the forwarding information of prefixes c and e since their forwarding information is more specific than that of the expanded prefix. Thus, expansion of prefixes a and d finally results in six prefixes, not eight. In general, when an expanded prefix collides with an existing longer prefix, forwarding information of the existing prefix must be preserved to respect the longest matching rule.

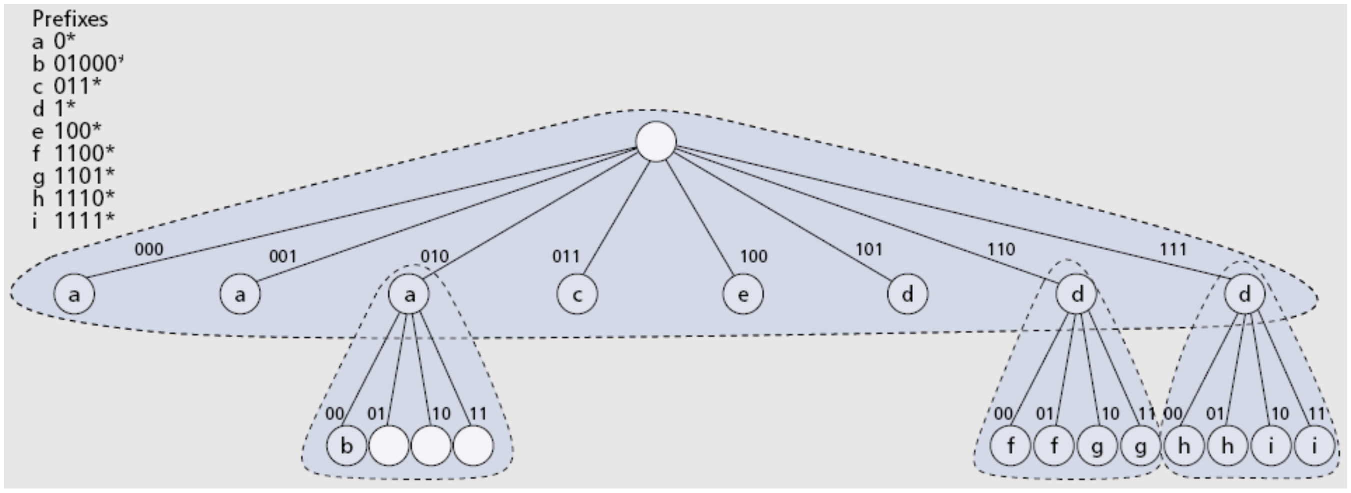


Figure 2.3: A Multi-bit Trie

When it comes to updating, in a multi-bit trie inserting and deleting operations are slightly more complicated than with binary tries because of CPE; especially when we choose to use larger strides. Choosing larger strides will make faster searches, but updates will require more entries to be modified. Inserting one prefix means finding the appropriate sub-trie, doing an expansion, and inserting each of the resulting prefixes. Deleting is still more complicated because it means deleting the expanded prefixes and, more important, updating the entries with the next LPM. The problem is that original prefixes are not actually stored in the trie. To see this better, suppose we insert prefixes 101*, 110* and 111* in the multi-bit trie in Figure 2.3. Clearly, prefix d will disappear; and if later we delete prefix 101*, for instance, there will be no way to find the new LPM (d) for node 101. Thus, update operations need an additional structure for managing original prefixes (e.g. a conventional Binary Trie).

A good example of an application that employs multi-bit tries in order to address the LPM for IPv4 is found in Gupta et al. [12]. They have developed a special case of CPE specifically targeted to hardware implementation. Arguing that DRAM is a plentiful and inexpensive resource, their technique sacrifices large amounts of memory in order to bound the number of off-chip memory accesses to two or three. Figure 2.4 shows their basic scheme. As we can see, is a two level multi-bit trie with an initial stride length of 24 and second level tables of stride length eight. Given that random accesses to DRAM may require up to eight clock cycles and current DRAMs operate at less than half the speed of SRAMs, this technique fails to out-perform techniques utilizing SRAM and requiring less than 10 memory accesses.

Other techniques such as Lulea [13] and Eatherton and Dittia's Tree Bitmap [14] employ multi-bit tries with compressed nodes. The Lulea scheme essentially compresses an expanded, leaf-pushed trie with stride lengths 16, 8, and 8. The technique of Leaf Pushing reduces the amount of information stored in each table entry by "pushing" best match information to leaf nodes such that a table entry contains either a pointer or information [1]. In the worst case, the scheme requires 12 memory accesses; however, the data structure only requires a few bytes per entry. While extremely compact, the Lulea scheme's update performance suffers from its implicit use of leaf pushing. The Tree Bitmap technique avoids leaf pushing by maintaining compressed representations of the prefixes stored in each multi-bit node. It also employs a clever indexing scheme to reduce pointer storage to two pointers per multi-bit node. Storage requirements for Tree Bitmap are on the order of 10 bytes per entry, worst-case memory accesses can be held to less than eight with optimizations, and updates require modifications to a few memory words resulting in excellent incremental update performance.

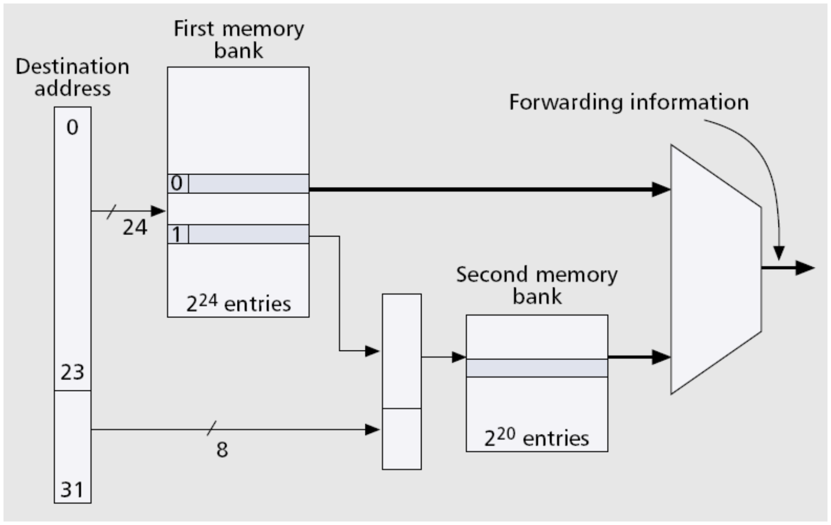


Figure 2.4 Hardware implementation of a fix-stride Multi-bit Trie for IPv4 LPM [12]

The fundamental issue with all trie-based techniques is that performance and scalability are fundamentally tied to address length. As many in the Internet community are pushing to widely adopt IPv6, it is unlikely that trie-based solutions will be capable of meeting performance demands [1].

Binary Search on Prefix Lengths

The problem with arbitrary prefix lengths is that we do not know how many bits of the destination address should be taken into account when compared with the prefix values. Tries allows a sequential search on the length dimension: first we look in the set of prefixes of length 1, then in the set of length 2 prefixes, and so on. Moreover, at each step the search space is reduced because of the prefix organization in the trie. Another approach to sequential search on lengths without using a trie is organizing the prefixes in different tables according to their lengths. In this case, a hashing technique can be used to search in each of these tables. We here note that this approach is most closely related to the algorithm we have implemented in this thesis. Since we look for the longest match, we begin the search in the table holding the longest prefixes; the search ends as soon as a match is found in one of these tables. Nevertheless, the number of tables equals the number of different prefix lengths. If *W* is the address length, the time complexity of the search operation is *O*(*W*) assuming a perfect hash function, which is the same as for a trie.

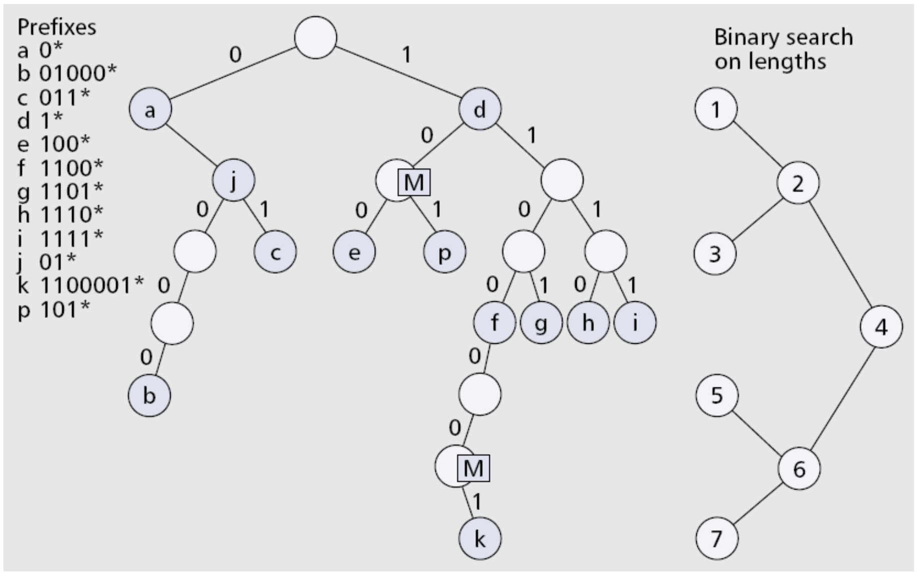


Figure 2.5: Binary search on prefix lengths

In order to reduce the search time, a binary search on lengths was proposed by Waldvogel *et al.* [14]. In a binary search, we reduce the search space in each step by half. On which half to continue the search depends on the result of a comparison. However, an ordering relation needs to be established before being able to make comparisons and proceed to search in a direction according to the result. Comparisons are usually done using key values, but our problem is different since we do binary search on lengths. We are restricted to checking whether a match exists at a given length. Using a match to decide what to do next is possible: if a match is found, we can reduce the search space to only longer lengths. Unfortunately, if no match is found, we cannot be sure that the search should proceed in the direction of shorter lengths, because the LPM could be of longer length as well. Waldvogel et al. insert extra prefixes of adequate length, called *markers*, in order to be sure that, when no match is found, the search must proceed necessarily in the direction of shorter prefixes.

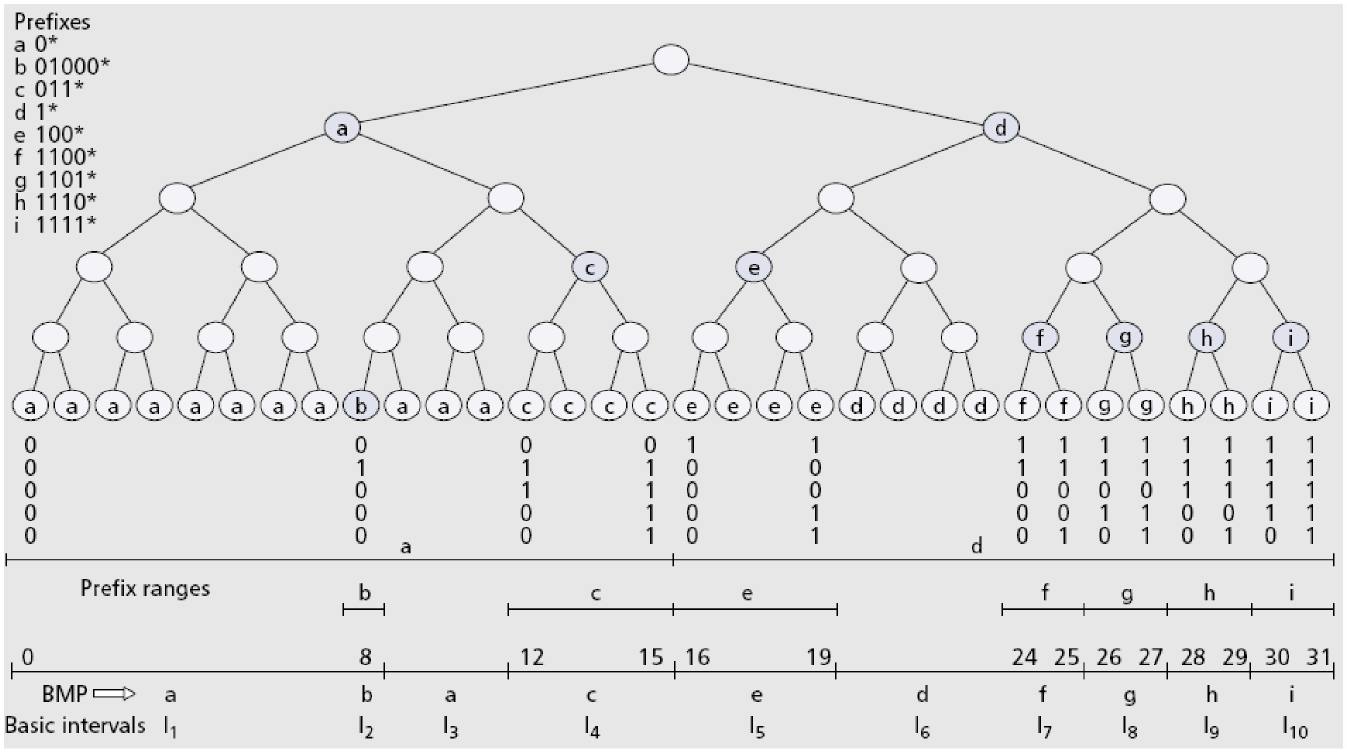


Figure 2.6: Binary range search

To illustrate this approach consider the prefixes shown in Figure 2.5. In the trie we can observe the levels at which the prefixes are located. At the right, a binary search tree shows the levels or lengths that are searched at each step of the binary search on lengths algorithm. Note that the trie is only shown to understand the relationship between markers and prefixes, but the algorithm does not use a trie data structure. Instead, for each level in the trie, a hash table is used to store the prefixes. For example, if we search for the LPM of the address 11000010, we begin by searching the table corresponding to length 4; a match will be found because of prefix f, and the search proceeds in the half of longer prefixes. Then we search at length 6, where the marker 110000* has been placed. Since a match is found, the search proceeds to length 7 and finds prefix k as the longest matching prefix. Note that without the marker at level 6, the search procedure would fail to find prefix k as the LPM. In general, for each prefix entry a series of markers are needed to guide the search. Since a binary search only checks a maximum of $\log_2 W$ levels, each entry will generate a maximum of $\log_2 W$ markers. In fact, the number of markers required will be much smaller for two reasons: no marker will be inserted if the corresponding prefix entry already exists (prefix f in Figure 2.5), and a single marker can be used to guide the search for several prefixes (e.g., prefixes e and p, which use the same marker at level 2). However, for the very same reasons, the search may be directed toward longer prefixes, although no longer prefix will match. For example, suppose we search for the LPM for address 11000001. We begin at level 4 and find a match with prefix f, so we proceed to length 6, where we find again a match with the marker, so we proceed to level 7. However, at level 7 no match will be found because the marker has guided us in a bad direction. While markers provide valid hints in some cases, they can mislead in others. To avoid backtracking when being misled, Waldvogel et al. uses pre-computation of the LPM for each marker. In our example, the marker at level 6 will have f as the pre-computed LPM. Thus, as we search, we keep track of the pre-computed LPM so far, and then in case of failure we always have the last LPM. The markers and pre-computed LPM values increase the memory required. Empirical measurements using an IPv4 database resulted in memory requirements of about 42 bytes per entry. Additionally, the update operations become difficult because of the several different values that must be updated.

Prefix Range Search

A prefix represents an aggregation of contiguous addresses; in other words, a prefix determines a well-defined range of addresses. For example, supposing 5-bit-length addresses, prefix a = 0* defines the range of addresses {0…15}. So why not simple store the range endpoints instead of every single address? The LPM of the endpoints is, in theory, the same for all the addresses in the interval; and search of the LPM for a given address would be reduced to finding any of the endpoints of the corresponding interval (e.g., the predecessor, which is the greatest endpoint smaller than or equal to a given address). The LPM problem would be readily solved, because finding the predecessor of a given address can be performed with a classical binary search method. Unfortunately, this approach may not work because prefix ranges may overlap. For example, Figure 2.6 shows the full expansion of prefixes assuming 5-bit-length addresses. The same figure shows the endpoints of the different prefix ranges, in binary as well as decimal form. There we can see that the predecessor of address value 9, for instance, is endpoint value 8; nevertheless, the LPM of address 9 is not associated with endpoint 8 (b), but with endpoint 0 (a) instead. Clearly, the fact that a range may be contained in another range does not allow this approach to work. One solution is to avoid interval overlap. In fact, by observing the endpoints we can see that these values divide the total address space into disjoint basic intervals.

In a basic interval, every address actually has the same LPM. Figure 2.6 shows the LPM for each basic interval of our example. Note that for each basic interval, its LPM is the LPM of the shortest prefix range enclosing the basic interval. The LPM of a given address can now be found by using the endpoints of the basic intervals. Nevertheless, we can observe in Figure 2.6 that some basic intervals do not have explicit endpoints (e.g., I3 and I6). In these cases, we can associate the basic interval with the closer endpoint to its left. As a result, some endpoints need to be associated to two basic intervals, and thus endpoints must maintain in general two LPMs, one for the interval they belong to and one for the potential next basic interval. For instance, endpoint value 8 will be associated with basic intervals I2 and I3, and must maintain LPMs b and a.

Figure 2.7 shows the search tree indicating the steps of the binary search algorithm. The leaves correspond to the endpoints, which store the two LPMs (= and >). For example, if we search the LPM for address 10110 (22), we begin comparing the address with key 26; since 22 is smaller than 26, we take the left branch in the search tree. Then we compare 22 with key 16 and go to the right; then at node 24 we go to the left arriving at node 19; and finally, we go to the right and arrive at the leaf with key 19. Because the address (22) is greater than 19, the LPM is the value associated with > (i.e., d). Concluding, as for traditional binary search, the implementation of this scheme can be made by explicitly building the binary search tree.

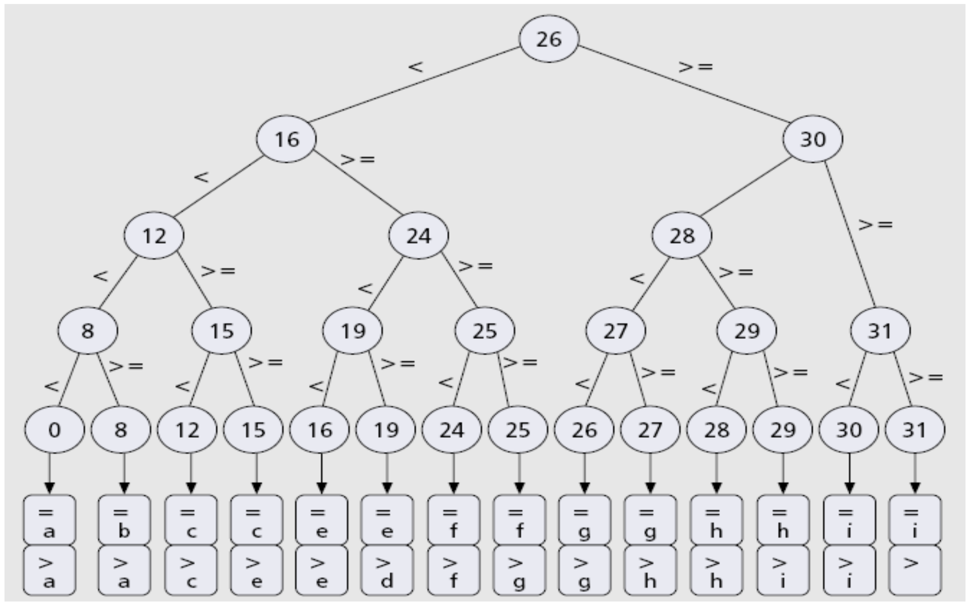


Figure 2.7: A basic range search tree

Comparison of Algorithmic Schemes

The ideal scheme would be one with fast searching, fast dynamic updates, and a small memory requirement. The schemes presented make different tradeoffs between these aspects. The most important metric is obviously lookup time, but update time must also be taken into account, as well as memory requirements. Scalability is also another important issue, with respect to both the number and length of prefixes. The complexity of the different schemes is compared in Table 2.1 [2]. A detailed extraction of the complexities can be found in [2]. For Table 2.1, *W* denotes the length of a prefix, *N* is the number of prefixes in the database and – indicates a “can’t do” operation.

Algorithmic Scheme	**Worst case lookup**	**Update**	**Memory**
Binary Trie	O(W)	O(W)	O(NW)
Path-compressed tries	O(W)	O(W)	O(N)
k-stride multibit tries	O(W/k)	$O(W/k + 2^k)$	$O(2^k NW/k)$
Lulea Trie	O(W/k)	-	$O(2^k NW/k)$
Binary search on prefix length	$O(\log_2 W)$	$O(N\log_2 W)$	$O(\log_2 W)$
Binary range search	$O(\log_2 N)$	O(N)	O(N)

Table 2.1: Complexity comparison [2]

An important issue in the Internet is scalability. Two aspects are important: the number of entries and the prefix length. The last aspect is especially important because of the next generation of IP (IPv6), which uses 128-bit addresses. Multi-bit tries improve lookup speed with respect to binary tries, but only by a constant factor on the length dimension. Hence, multi-bit tries scale badly to longer addresses. Binary search on lengths has a logarithmic complexity with respect to the prefix length, and its scalability property is very good. The range search approaches have logarithmic lookup complexity with respect to the number of entries but independent, in principle, of prefix length. Thus, if the number of entries does not grow excessively, the range search approach is scalable for IPv6. This is something that, as we have seen in Section 1, is unlikely to happen.

Like the “Binary Search on Prefiix Lengths" technique, our approach begins by sorting the database into sets based on prefix length. We assert that our scheme exhibits several advantages over the previously mentioned techniques First and foremost, we will show that the number of dependent memory accesses required for a lookup can be held constant given that memory resources scale linearly with the size of the forwarding table. We also show that our approach remains memory efficient for large databases and provide evidence for its applicability to IPv6. Second, by avoiding significant pre-computation like “markers” and “leaf pushing" our approach retains good incremental update performance [1].

2.2 Hardware Based Approaches

Most memory devices store and retrieve data by addressing specific memory locations. As a result, this path often becomes the limiting factor for systems, such as IP address look-up, that rely on fast memory accesses. The time required to find an item stored in memory can be reduced considerably if the item can be identified for access by its content rather than by its address. A memory that is accessed in this way is called content addressable memory or CAM.

Content Addressable Memories (CAMs)

Taking a cue from fully-associative cache memories, Content Addressable Memory (CAM) devices perform massive parallel search over all prefixes of the routing table. A CAM has three basic operations (a) write into the next free location (b) search for a word match (c) read matching entries. Data may be transferred to or from CAM without knowing the memory address of the word. Binary data is automatically written to the next free word. To read a word the user must first do a search operation. Then, if there are multiple matches, the CAM decides (based on some internal state) which matched word to read next. Reading is useful because a CAM word has two parts. The most important part is the **search-field**, which is the part of the word that is matched with the search pattern. This typically contains the addresses of the known destinations. The CAM word also contains a **return-field**, which is the information returned during a read. For the case of IP look-up, the return-filed contains the port number for the next hop.

All the CAM look-up schemes are similar to the parallel search used in a RAM; however, the size of a CAM needed for direct access is determined primarily by the number of words that require storage. The size of the search-field only affects the number of bits a word requires. For example, with a 10-bit search field, 10 bits per word are required. Thus, if there were only 256 possible inputs, each with a 16-bit search field, the CAM must have 256 words with each word being at least 16 bits. The size and cost of a CAM grows linearly with the size of the search field as well as the number of entries.

Ternary Content Addressable Memories (TCAMs)

While binary CAMs perform well for exact match operations and can be used for route look-ups in strictly hierarchical addressing schemes [15], the wide use of address aggregation techniques like CIDR requires storing and searching entries with arbitrary prefix lengths. In response, Ternary Content Addressable Memories (TCAMs) were developed with the ability to store an additional “Don't Care” state thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. The longest prefix matching operation is performed in a TCAM by storing entries in decreasing order of prefix lengths. The TCAM searches the destination address of an incoming packet with all the prefixes in parallel. Several prefixes (up to 32 in IPv4 and up to 128 in IPv6) may match the destination address. A priority encoder logic then selects the first matching entry – that is, the entry with the matching prefix at the lowest memory address. Figure 2.8 shows an example [16].

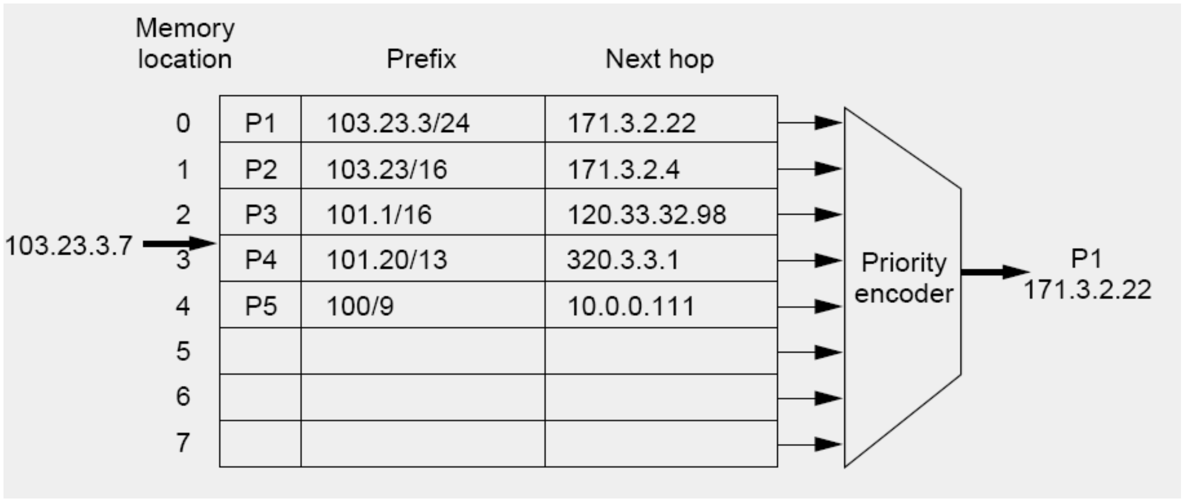
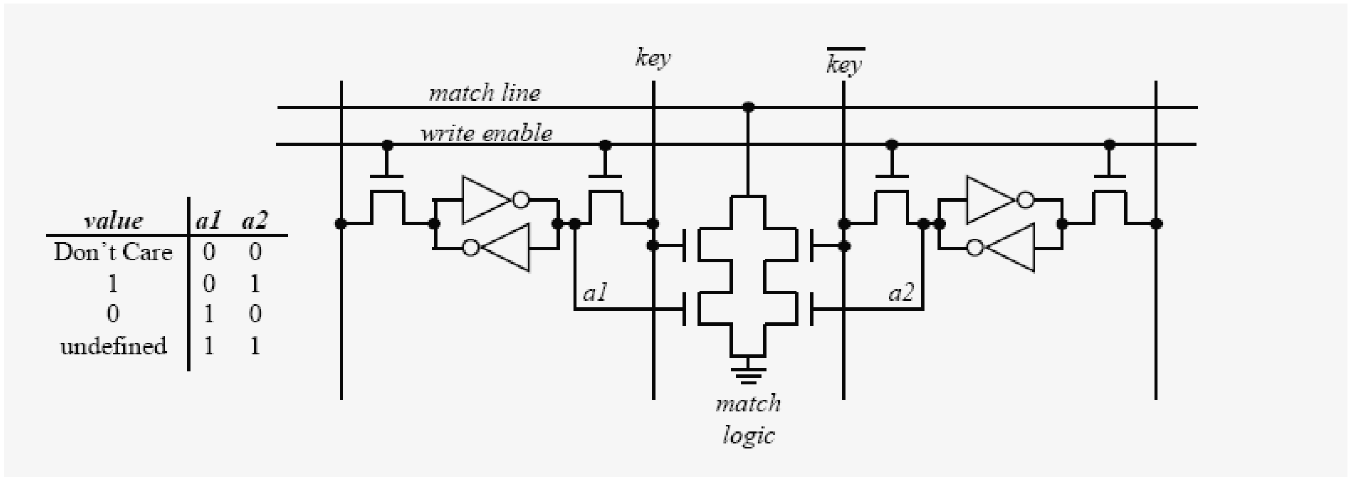


Figure 2.8: IPv4 longest prefix matching using a TCAM

The downside of CAMs

The CAM’s high degree of parallelism comes at the cost of storage density, access time, and power consumption. Since the input key is compared against every memory word, each bit of storage requires match logic to drive a match word line which signals a match for the given key. The extra logic and capacitive loading due to the massive parallelism lengthens access time and increases power consumption. A circuit diagram of a standard TCAM cell is shown in Figure 2.9. We can see, that in addition to the six transistors required for binary digit storage, a typical TCAM cell requires an additional six transistors to store the mask bit and four transistors for the match logic, resulting in a total of 16 transistors and a cell 2.7 times larger than a standard SRAM cell [17]. Some proprietary architectures allow TCAM cells to require as few as 14 transistors [15].

We can make a first-order comparison between Static Random Access Memory (SRAM) and TCAM technology by observing that SRAM cells typically require six transistors to store a binary bit of information, consume between 20 to 30 nano-Watts per bit of storage, and operate with 3ns clock periods (333 MHz). Current generation Dual Data Rate (DDR) SRAMs are capable of retrieving two words per clock cycle; however, these devices often have a minimum burst length of two words [1]. Hence, the DDR feature effectively doubles the I/O bandwidth but retains the same random access time. We have already seen that a typical TCAM cell requires additional six transistors to store the mask bit and four transistors for the match logic (Figure 2.9). In addition to being less dense than SRAM, current generation TCAMs achieve 100 million lookups per second, resulting in access times over 3.3 times longer than SRAM due to the capacitive loading induced by the parallelism [18]. Additionally, power consumption per bit of storage is on the order of 3 micro-Watts per bit [19]. In summary, TCAMs consume 150 times more power per bit than SRAM and currently cost about 30 times more per bit of storage. Therefore, a look-up technique that employs standard SRAM, requires less than four memory accesses per lookup, and utilizes less than 44 bytes per entry, for IPv6, not only matches TCAM performance and resource utilization, but also provides a significant advantage in terms of cost and power consumption.



value	*a1*	*a2*
Don't Care	0	0
1	0	1
0	1	0
undefined	1	1

Figure 2.9: Circuit diagram of a standard TCAM cell; the stored value (0, 1, Don't Care) is encoded using two registers *a1* and *a2* [15]

3. Our Approach

The look-up scheme we have developed is primarily based on the use of Bloom filters and conventional hash coding methods. Therefore, in the beginning of this section we will explain the theory behind Bloom filters and present the hardware hash functions we have selected to use. In the rest of the section, we will show the basic configuration of our approach and conclude by defending our selection of implementing the system using FPGA technology.

3.1 Bloom Filters

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. A filter is first "programmed" with each messages in the set and then queried to determine the membership of a particular message. It was first introduced by Burton H. Bloom in 1970 as a structure providing space/time trade-offs in hash coding methods with allowable error. B. H. Bloom stated that the reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications, in particular, applications in which a large amount of data is involved and a core resident hash area is consequently not feasible using conventional hash coding schemes [11].

Bloom Filter Theory

A Bloom filter is essentially a bit-vector of length $\boldsymbol{m}$ used to efficiently represent a set of messages. Given a set of messages $\boldsymbol{X=\{x_1, x_2, \ldots, x_n\}}$ with $\boldsymbol{n}$ members, the Bloom filter is "programmed" as follows. For each message $\boldsymbol{x_i}$, in $\boldsymbol{X}$, $\boldsymbol{k}$ hash functions are computed on $\boldsymbol{x_i}$ producing $\boldsymbol{k}$ values each ranging from 1 to $\boldsymbol{m}$. Each of these values, in general, address a single bit in the m-bit vector, thus each message $\boldsymbol{x_i}$ causes $\boldsymbol{k}$ bits in the vector to be set to 1. We here note that if one of the $\boldsymbol{k}$ hash functions addresses a bit in the vector that is already set (i.e. by some other member of the set) then that bit stays unchanged. Querying the filter for a set membership, of a given message $\boldsymbol{x}$, is similar to the programming process. Given message $\boldsymbol{x}$, $\boldsymbol{k}$ hash values are generated using the same hash functions used to program the filter. The bits in the m-bit long vector at the locations corresponding to the $\boldsymbol{k}$ hash values are checked. If at least one of the $\boldsymbol{k}$ bits is 0, then the message is declared to be a non-member of the set. If all the bits are found to be 1, then the message is said to belong to the set $\boldsymbol{X}$ with a certain probability.

If all the $\boldsymbol{k}$ bits are found to be 1 and $\boldsymbol{x}$ is not a member of $\boldsymbol{X}$, then it is said to be a false positive match. This ambiguity in membership comes from the fact that the $\boldsymbol{k}$ bits in the m-bit vector can be set by any of the $\boldsymbol{n}$ members of $\boldsymbol{X}$. Thus, finding a bit set to 1 does not necessarily imply that it was set by the particular message being queried. Figure 3.1 shows a Bloom filter, with m=12, programmed using members $\mathbf{x_1}$ and $\mathbf{x_2}$. However, when we queried it using the nonmember $\mathbf{y_3}$ the filter indicates that this is a member of the set **X={x₁, x₂}**. This is a typical example of a false positive match. On the other hand, finding a 0 bit certainly implies that the message does not belong to the set, since if it were a member then all **k**-bits would have been set to 1 when the Bloom filter was programmed.

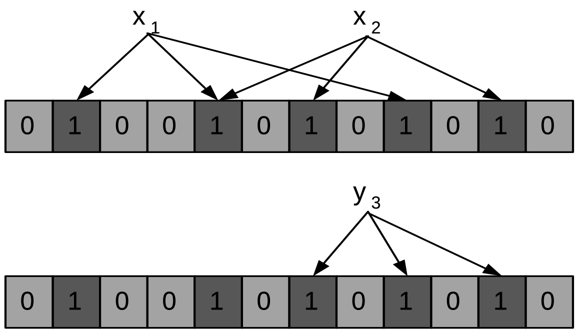


Figure 3.1: A false positive match

Now we take a look at the step-by-step derivation of the false positive probability (i.e., for a message that is not programmed, we find that all ***k*** bits that it hashes to be 1). The probability that a random bit of the m-bit vector is set to 1 by a hash function is simply

$$P(set) = \frac{1}{m}$$

The probability that it is not set is

$$P(not\ set) = 1 - \frac{1}{m}$$

Thus, the probability that it is not set by any of ***n*** members of X is

$$P(not\ set\ after\ n\ members\ are\ programmed) = \left(1 - \frac{1}{m}\right)^{n}$$

And since each of the messages sets ***k*** bit in the bit vector we have

$$P(randomly\ selected\ bit\ is\ not\ set) = \left(1 - \frac{1}{m}\right)^{n \cdot k}$$

So, the probability that a randomly selected bit is found to be 1 is

$$P(randomly\ selected\ bit\ is\ set) = 1 - \left(1 - \frac{1}{m}\right)^{n \cdot k}$$

For a message, not programmed into the filter, to be detected as a member all ***k*** hash address must be found to be addressing at a bit that is set. Hence, the probability that this happens, ***fp***, is:

$$fp = \left(1 - \left(1 - \frac{1}{m}\right)^{n \cdot k}\right)^{k} \quad \text{(The false positive probability)}$$

For large values of ***m*** the asymptotic approximation is $\left(1-\frac{1}{m}\right)^{k\cdot n} \approx e^{-\frac{n}{m}\cdot k}$ and thus,

$$fp = \left(1-e^{-\frac{n}{m}\cdot k}\right)^{k}$$

Note that we use the asymptotic approximation from now on for convenience with the mathematical analysis.

Since this probability is independent of the input message, it is termed the **false positive probability**. The false positive probability can be reduced by choosing appropriate values for ***m*** and ***k*** for a given size of the member set, ***n***. It is clear that the size of the bit-vector, ***m***, needs to be quite large compared to the size of the message set, ***n***. For a given ratio of ***m/n***, the false positive probability can be reduced by increasing the number of hash functions, ***k***. In the optimal case, when false positive probability is minimized with respect to ***k***, we get the following relationship:

$$k_{optimal} = \frac{m}{n}\ln 2$$

Proof:

$f = \left(1-e^{-\frac{n}{m}\cdot k}\right)^{k}$ the $\frac{n}{m}$ ratio is considered to be constant ,so $\frac{n}{m} = c$

$$f(k) = \left(1-e^{-c\cdot k}\right)^{k} \Rightarrow \ln\left(f(k)\right) = k\cdot\ln\left(1-e^{-c\cdot k}\right) \xrightarrow{d/dk} \frac{d}{dk}\left(\ln\left(f(k)\right)\right) = \frac{d}{dk}\left(k\cdot\ln\left(1-e^{-c\cdot k}\right)\right)$$

$$\Rightarrow \frac{1}{f(k)}\cdot\frac{d}{dk}\left(f(k)\right) = \ln\left(1-e^{-c\cdot k}\right) + ck\cdot e^{-c\cdot k} \Rightarrow \frac{d}{dk}\left(f(k)\right) = \left(1-e^{-c\cdot k}\right)^{k}\left(\ln\left(1-e^{-c\cdot k}\right) + \frac{ck\cdot e^{-c\cdot k}}{1-e^{-c\cdot k}}\right)$$

$$\overset{\frac{d}{dk}f(k)=0}{\Rightarrow} \left(1-e^{-c\cdot k}\right)^{k} \neq 0, \forall k\in\Re \Rightarrow \ln\left(1-e^{-c\cdot k}\right) + \frac{ck\cdot e^{-c\cdot k}}{1-e^{-c\cdot k}} = 0\,, 1-e^{-c\cdot k} \neq 0, \forall k\in\Re$$

$$\Rightarrow \ln\left(1-e^{-c\cdot k}\right)\left(1-e^{-c\cdot k}\right) + ck\cdot e^{-c\cdot k} = 0\,, \left[e^{-c\cdot k} = x \Leftrightarrow -ck\ln e = \ln x \Leftrightarrow k = -\frac{\ln x}{c}\right]$$

$$\Rightarrow (1-x)\ln(1-x) - \frac{\ln x}{c}c\cdot x = 0 \quad \Rightarrow \quad (1-x)\ln(1-x) = x\ln(x)\,, thus\ x = 1-x \Leftrightarrow 2x = 1 \Leftrightarrow x = \frac{1}{2}$$

$$\Rightarrow k = \frac{1}{c}\cdot\left(-\ln\frac{1}{2}\right) \Rightarrow k = \frac{1}{c}\ln 2 \;\; where\ c = \frac{n}{m} \;\; thus\;\; k = \frac{m}{n}\cdot\ln 2$$

We have to calculate the second derivative of ***f(k)*** in order to proof that this point is indeed the minimum of the function.

$$(1-x)\ln(1-x) - x\ln x = 0 \xrightarrow{d/dx}$$
$$\Rightarrow -\frac{1}{1-x}(1-x) + (-1)\ln(1-x) - \left(\ln x + \frac{1}{x}x\right) = 0$$
$$\Rightarrow -1 - \ln(1-x) - \ln x - 1 = 0 \Rightarrow -\left(\ln(1-x) + \ln(x)\right) = 2$$
$$\Rightarrow \ln\left(x(1-x)\right)^{-1} = 2 \Rightarrow \ln\frac{1}{x(1-x)} = 2 \Rightarrow \frac{1}{x(1-x)} = e^2$$
$$\Rightarrow \frac{1}{e^2} = x - x^2 \Rightarrow x^2 - x + \frac{1}{e^2} = 0 \ @\, x = \frac{1}{2} \Rightarrow \frac{1}{4} - \frac{1}{2} + \frac{1}{e^2} < 0$$

Thus at x=1/2 we have the minimum of the false positive probability. The false positive probability at this optimal point is given by

$$f = \left(\frac{1}{2}\right)^k$$

It should be noted that if the false positive probability is to be fixed, then the size of the filter, ***m***, needs to scale linearly with the size of the message set, ***n***.

The intuitive explanation why we have to minimize for ***k*** is that there are two competing forces in the false positive equation. On the one hand, by using more hash functions gives as more chances of finding a zero bit for an element that is not a member of ***X***. But on the other hand, using fewer hash functions increases the fraction of zero bits in the array. These two are the trade-offs of choosing the number of hash functions (k) for a given n/m ratio and they can be shown graphically in Figure 3.2.

In Figure 3.2 we can observe that, with a fixed ***n/m*** ratio, as we increase the number of hash functions up to a point (that is $\mathbf{k}_{\mathbf{optimal}}$) the false positive probability is decreasing. However, if we continue increasing the number of hash functions beyond that optimal point we can see that the false positive probability is getting higher.

A slightly different configuration for Bloom filters

In this configuration instead of having one array of size ***m*** shared by all the hash functions, each hash function has a range of **m/k** consecutive bit locations disjoint from all others (*Figure 3.3*). The total number of bits is still ***m***, but the bits are divided equally among the ***k*** hash functions ($h_1 \rightarrow h_k$).

The false positive probability, in this scheme, is slightly worst than that of the conventional configuration being analyzed in the previous section. In order to derive the new probability we follow the same steps as before. We know that each smaller bit vector is **m/k** bits long.

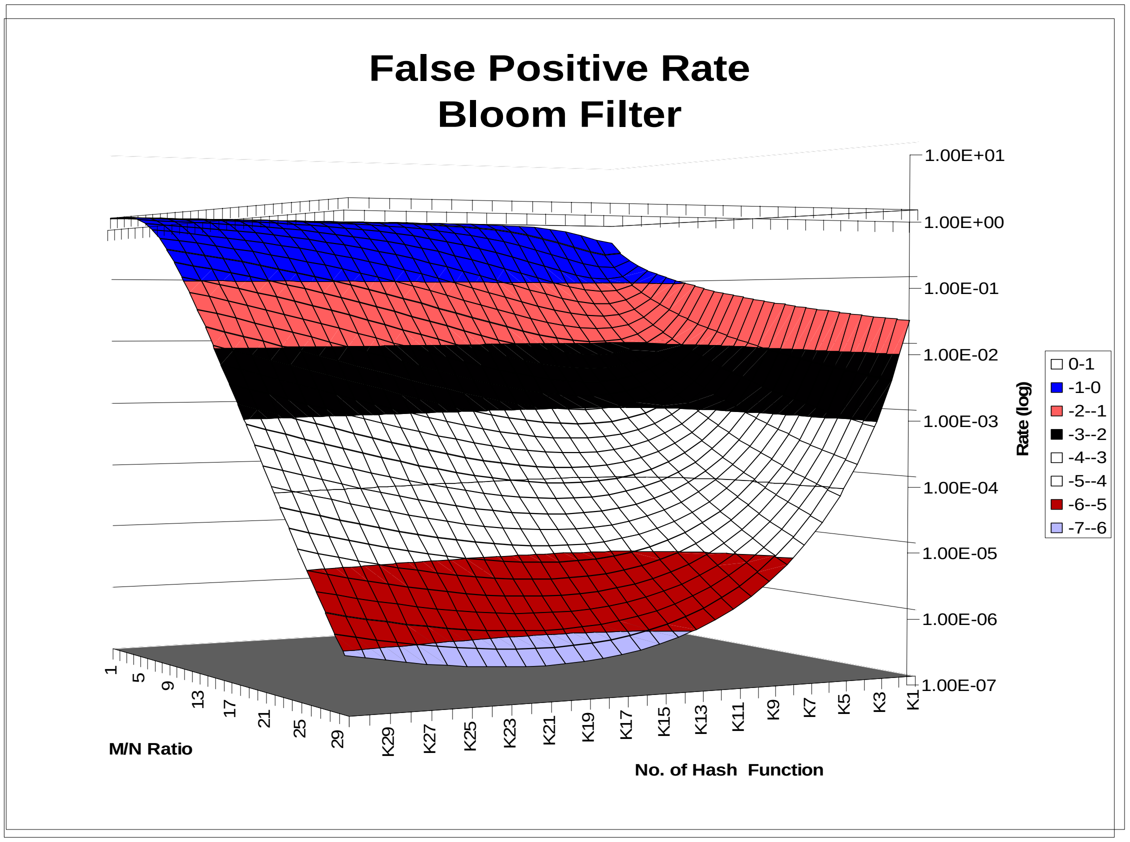


Figure 3.2: The false positive probability for various numbers of n/m ratios and number of hash functions

Hence, the probability that a randomly selected bit is set by a single hash function is:

$$P(set) = \frac{1}{m/k} = \frac{k}{m} \quad \text{and} \quad P(not\ set) = 1 - \frac{k}{m}$$

So the probability that is not set by any of the ***n*** members of ***X*** is:

$$P(not\ set\ by\ any\ member) = \left(1 - \frac{k}{m}\right)^n \approx e^{-\frac{k}{m}n}$$

Asymptotically, then, the performance is the same as the original scheme. However, since

$$\left(1 - \frac{k}{m}\right)^n \le \left(1 - \frac{1}{m}\right)^{kn},$$

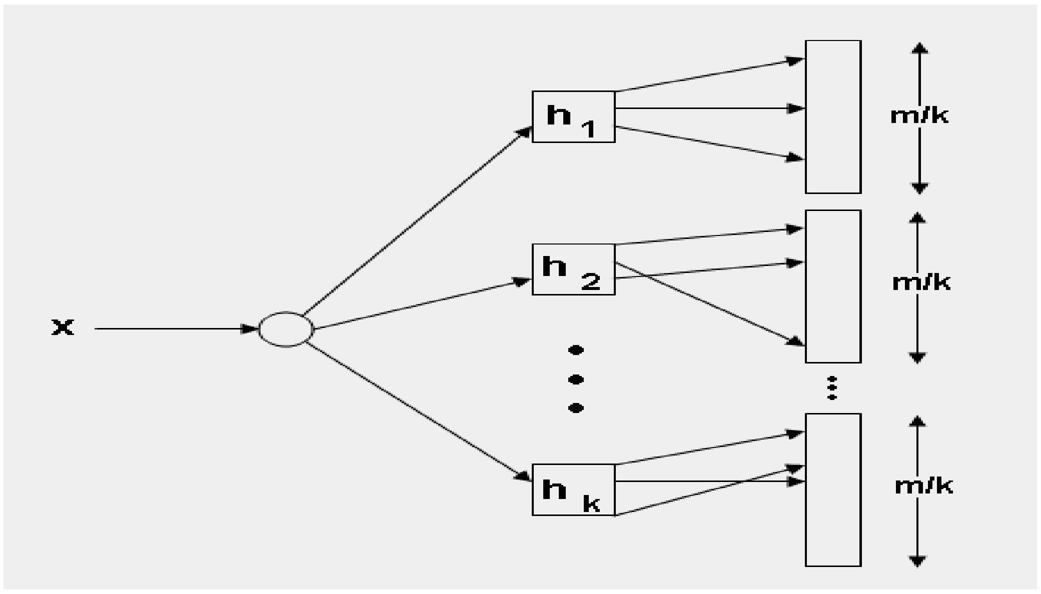


Figure 3.3: The C^th^ hash function produces a position at the C^th^ smaller bit vector

the probability of a false positive is actually slightly higher with this division. Since the difference is small, this approach may be still useful for implementation reasons; for example, dividing the bits among the hash functions may make parallelization of array accesses easier.

In general, the number of hash functions ***k***, is essentially the lookup capacity of the actual memory storing the Bloom filter. Thus, k=6 implies that 6 random locations must be accesses in the time allotted for a Bloom filter query. In the case of single cycle Bloom filter queries, on-chip memories need to support at least ***k*** reading ports. For designs with values of ***k*** higher than what can be realized by technology, a single memory with the desired lookups is realized by employing multiple smaller memories, with fewer ports. For instance, if the technology limits the number of ports on a single memory to 4, then 2 such smaller memories are required to achieve a lookup capacity of 8 as shown in *Figure 3.4b*. The basic Bloom filter allows any hash function to map to any bit in the vector. It is possible that for some member, more than 4 hash functions map to the same memory segment, thereby exceeding the lookup capacity of the memory. This problem can be solved by restricting the range of each hash function to a given memory. This avoids collision among hash functions across different memory segments.

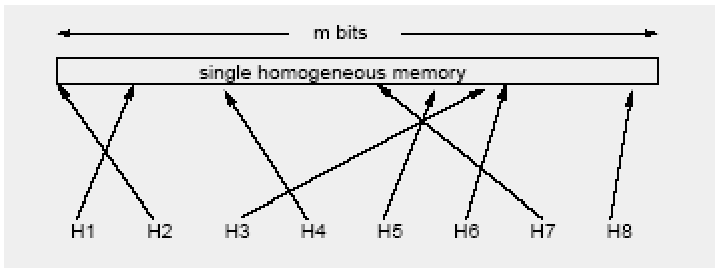


Figure 3.4a: A Bloom filter with single memory vector of length m and k=8

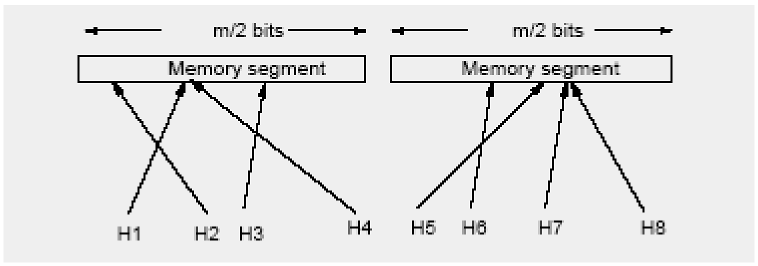


Figure 3.4b: Two Bloom filters of length m/2 and k=4, together they realize an m bits long Bloom filter with k=8

Therefore, if ***h*** is the maximum lookup capacity of a RAM as limited by the technology, then ***k/h*** such memories of size ***m/(k/h)*** can be combined to realize the desired capacity of ***m*** bits and ***k*** hash functions. When only ***h*** hash functions are allowed to map to a single memory, the false positive probability can be expressed as

$$f' = \left[1 - \left(1 - \frac{1}{\frac{m}{\left(k/h\right)}}\right)^{hn}\right]^{\left(k/h\right)h} \approx \left[1 - e^{-nk/m}\right]^{k}$$

Properties of Bloom filters

Here we will present some standard Bloom filter tricks. The simple structure of Bloom filters makes certain operations very easy to implement. For example, suppose one has two Bloom filters representing sets S_1 and S_2 with the same number of bits (m) and using the same number of hash functions (k). Then a Bloom filter that represents the union ($S_1 \cup S_2$) of two sets can be obtained by taking the **OR** of the two bit vectors of the original Bloom filters, as it can be shown in Figure 3.5.

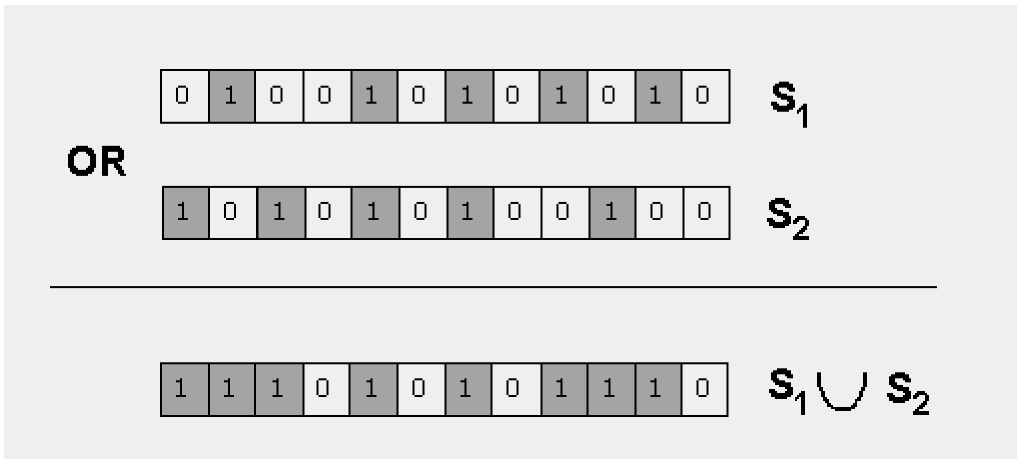


Figure 3.5: Merging two Bloom filters

Another interesting feature is that the Bloom filters can easily be halved in size. Suppose the size of the filter is a power of 2. If one wants to half the size of the filter, then just **OR** the first and second halves together (Figure 3.6). When hashing, the high order bits can be masked in order for the new hash address to map directly to the new hash table address space.

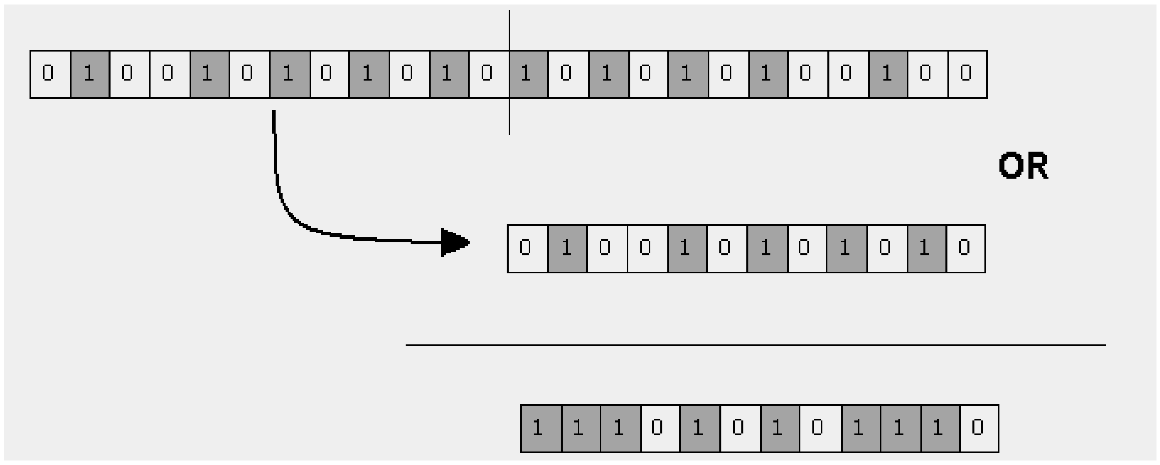


Figure 3.6: Decreasing the size of a Bloom filter by 50%

Counting Bloom Filters

In this system we have a set that is changing over time, with elements being inserted and deleted. Inserting elements into the Bloom filter is relatively easy; hash the elements ***k*** times and set the bits to 1. Unfortunately, one cannot perform a deletion by reversing the same process. If we hash the elements to be deleted and set the corresponding bits to 0, we may be setting a location to 0 that hashed to by some other element in the set. In this case, the Bloom filer will no longer correctly reflects all elements in the set and *false negatives* will occur.

In order to address this problem, P. Cao et al. [20], introduced the idea of a *Counting Bloom Filter*. In a counting Bloom filter, each entry is not a single bit but instead a small counter. When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. To avoid counter overflow, we choose sufficiently large counters. Analysis from P. Cao reveals that 4 bits per counter should suffice for most applications. In Section 4 (Implementation Details), we performed our own simulations and came to the same conclusion as [20].

3.2 Hardware Hash Functions

The performance of our IP look-up scheme is strongly depended on the efficiency of the hash functions that it uses. From our study we have seen that **bit extraction** and **exclusive ORing** "methods" are two commonly used hashing functions for hardware applications. However, from our research, we came to the conclusion that there is no study of the performance of these functions and no mention anywhere of the practical performance of the hash functions in comparison with the theoretical performance prediction of hashing schemes. The only serious study of hardware hashing schemes is from M.V Ramakrishna et al. [21] Most of the content of this subsection is based on their work. For easy reference, we summarize below all the notations about hashing used throughout the thesis.

A	the key space
B	the address space
i	number of bits in the keys
j	number of bits in the address
I	given key set
$x_1, \dots x_2$	given keys
b	bucket size (number of keys per address)
n	number of keys in I
m	number of buckets in the hash table
llps	length of the longest probe sequence
Q	set of all i x j Boolean matrices
q	a member of Q
H_3	a class of universal hashing functions
$h_q()$	a hashing function from the class H_3

Performance of Hash Functions

Let A = {0, 1,…, α -1} be the key space and B = {0, 1, …, m-1} the address space. For hardware applications, it is more appropriate to speak in terms of the number of bits. Let i denote the number of bits in the key and, hence, the size of the key space α is 2^i. Let j denote the number of bits in the address space and, thus, $m = 2^j$. Let I be the given key set, $I = \{x_1, x_2, \ldots, x_n\}$, $I \subset A$. There are a total of m^n possible functions from I to B.

There are hundreds of papers analyzing the theoretical performance of hashing schemes. Most of these analyses are under the assumption that each one of these m^n mappings is equally likely. This assumption is the same as the statement that the probability of a hash function mapping a given key to a particular location is *1/m*, independent of the other outcomes. This is the theoretical model of hashing on which analytical results reported in the literatures are based.

In addition, we are interested at a particular class of hashing functions which are called *universal₂* [21]. A class H of hashing functions is said to be *universal₂* if no pair of keys collide under more that $|H|/m$ of the functions in the class. Here $|H|$ is the number of hashing functions in H and m is the size of address space. Therefore, is of essential importance to select a hardware hash function that complies with the specifications mentioned above.

Due to the importance of this selection, we will present a portion of the experimental result from [21] in order to support our choice. The experiments were conducted as follows: The hash table was organized as a number of 1,024 buckets, corresponding to j = 10 bit hash address. The keys are 32 bits long (i = 32). For each experiment, the number of keys (n) corresponding to the chosen bucket size (b) and load factor is determined. The corresponding numbers of keys from the key set are hashed using the chosen hashing function according to the hashing scheme and the count of the total number of proves required is maintained.

bkt	load=0.6			load=0.7			load=0.8			load=0.9		
size	anal.	regu.	rand.	anal.	regu.	rand.	anal.	regu.	rand.	anal.	Regu.	rand.
1	4,3	12	19	4,6	13	23	4,9	14	27	5,2	14	31
2	5,0	15	35	5,5	16	37	6,0	16	39	6,5	16	45
3	5,5	15	44	6,1	16	37	6,8	19	56	7,4	24	57
4	5,8	18	55	6,6	23	59	7,4	23	60	8,1	23	69
5	6,0	22	58	7,0	22	68	7,9	23	73	8,8	27	81
10	6,4	28	83	8,0	35	95	9,6	40	106	11,2	44	121

Table 3.1: Length of Longest Probe Sequence (llp) for 'bit extraction functions', Separate Changing Scheme with 1,024 Buckets

The Bit Extraction Hash Function

These hashing functions consists of selecting *j* bits out of the *i* bits of the key. In his work, Ramakrishna [21], experimented with two different functions. For the first function (regu), the bits chosen were positions 1 +3*k*, where *k*=0, …,9, which give the *j* = 10 bit hash addresses. For the second function (rand), the bit positions were chosen randomly. From the results obtained [21], we observed that the experimental values of the search lengths are higher that the analytical results in most cases. For the ease of the reader these result are illustrated in Table 3.1.

The main reason **bit extraction** does not produce results close to the theoretical performance of uniform hashing, is due to the lack of prior knowledge about the entropy of a particular bit of the key. If this prior knowledge is practicable, then, **bit extraction** can be an excellent hash function scheme with extremely low computational cost and with absolute uniform distribution of hash addresses. On the other hand, if bit entropy is unknown to the designer, **bit extraction**, suffers from poor random distribution of produced hash addresses that can, hence, result to long deviations from the predicted theoretical performance of hashing.

Hash Functions using XOR operation

In this "method", the *i* bit key is partitioned into *j* bit segments. The segments are exclusived-ORed to give the hash address. Assume, we have eight bit key $x = x_1x_2...x_8$, then the four bit hash address is computed as $h(x)=(x_1 \oplus x_5)\ (x_2 \oplus x_6)\ (x_3 \oplus x_7)\ (x_4 \oplus x_8)$. Any other bit combination is also valid and, thus, can be used in order to generate more hash functions. In addition, we note that it is not necessary for all the *i* bits to be used in XORing. The experimental results are shown in Table 3.2.

bkt	**load=0.6**		**load=0.7**		**load=0.8**		**load=0.9**	
size	**anal.**	**rand.**	**anal.**	**Rand.**	**anal.**	**rand.**	**anal.**	**rand.**
1	7	4,3	7	4,6	8	4,9	9	5,2
2	9	5,0	10	5,5	11	6,0	13	6,5
3	12	5,5	14	6,1	15	6,8	15	7,4
4	14	5,8	14	6,6	14	7,4	15	8,1
5	13	6,0	14	7,0	15	7,9	15	8,8
10	13	6,4	14	8,0	15	9,6	15	11,2

Table 3.2: Length of the Longest Probe Sequence (lls) with 'XOR Hash Functions', Separate Changing Scheme with 1,024 Buckets

Hash Functions from the H_3 Class

Ramakrishna et al. experimented with the *H_3* Class of Universal Hash Functions. They support that, this class of hash functions, is suitable for hardware applications because of its computational simplicity and its high level of parallelism. In order to present these hash functions, let as first denote Q to be the set of all possible *i* x *j* binary matrices. For a given *q ε Q,* let *q(k)* be the bit string which is the *k*th row of the matrix *q*, and let *x(k)* denote the *k*th bit of *x*.

The hashing function $h_q(x) : A \to B$ is defined as,

$$h_q(x) = x(1) \cdot q(1) \oplus x(2) \cdot q(2) \oplus \ldots \oplus x(i) \cdot q(i) ,$$

where $\cdot$ denotes the bitwise AND operation and $\oplus$ the bitwise exclusive OR (XOR) operation.

The class H_3 is the set $\{ h_q \mid q \in Q \}$. We can observe that the hash functions are calculated cumulatively and hence the results calculated over the first *i* bits can be used for calculating the hash functions over the first $i + 1$ bits. This property, of the hash functions, results in a regular and less resource consuming hash function matrix. The hardware which stores the *i* x *j* binary matrix can be organized e.g. in a bank of registers. With this configuration, the same hardware can realize any desired hashing function for this class and the hash function can change dynamically by loading new data into the register bank. For more implementation details refer to Section 4 ("Implementation Details").

The hash functions from the class *H_3* are essentially linear transformations from A to B. To realize the analytical performance with real-life data, in practice, we need hash functions that distribute each key randomly into the address range. The requirement we are posing is similar to the requirement of random number generation. Since the hash functions from the *H_3* class are linear transformation, we infer that the functions will distribute the keys randomly [21].

In order to test this inference Ramakrishna et al. repeated their experimental procedure with 500 hashing function chosen randomly from the H_3 class. Table 3.3 gives a comparison of the experimental values of expected *llps* with the analytical values obtained from [21]. The experimental *llps* averages presented for each set of table parameters are averages over 500 hashing functions. We observe from the tables that all the experimental averages are within one standard deviation from the corresponding analytical expected values. In this table, the maximum of the 500 *llps* is no more than twice the expected values of *llps*. We can also see the range of *llps* values among the 500 values within which 95 percent of the *llps* values fell. This range is shown as "95% conf limits" in Table 3.3. We assert that these values provide experimental evidence for the theoretical conclusions found in the literature.

load factor	bucket size	1	2	3	4	5	10
0,6	**experimental**	4,326	5,048	5,498	5,788	6,014	6,33
	Analytical	4,333	5,04	5,48	5,777	5,988	6,366
	std. deviation	0,642	0,786	0,845	0,857	0,963	1,169
	95% conf. limits	3 to 6	4 to 7	4 to 7	5 to 8	5 to 8	5 to 9
	maximum llps	7	8	9	10	12	13
0,7	**experimental**	4,656	5,56	6,164	6,614	6,94	8,066
	Analytical	4,636	5,542	6,139	5,592	6,951	8,013
	std. deviation	0,686	0,812	0,877	0,937	1,016	1,824
	95% conf. limits	4 to 6	4 to 7	5 to 8	5 to 9	5 to 9	6 to 11
	maximum llps	7	9	11	12	12	14
0,8	**experimental**	4,964	6,032	6,788	7,392	7,896	9,6
	Analytical	4,947	6,016	6,777	7,377	7,88	9,613
	std. deviation	0,718	0,867	0,857	1,001	1,038	1,368
	95% conf. limits	4 to 6	5 to 8	6 to 9	6 to 9	6 to 10	8 to 13
	maximum llps	7	10	11	13	13	16
0,9	**experimental**	5,27	6,498	7,388	8,126	8,784	11,31
	Analytical	5,242	6,48	7,391	8,139	8,782	11,18
	std. deviation	0,708	0,873	0,906	1,005	1,087	1,351
	95% conf. limits	4 to 7	5 to 8	6 to 9	7 to 10	7 to 11	9 to 15
	maximum llps	8	10	12	13	15	17

Table 3.3: Expected Length of the Longest Probe Sequence (lls) for hashing functions from H_3 class, Separate Changing Scheme with 1,024 Buckets

We find it useful to introduce the reader with an example illustrating how these hash functions work. Let *i* be 8 and *j* be 3. Then the address space is *A={0,…,255}* and the key space is *B={0,…,7}*. We randomly choose an *8 x 3* matrix *q*:

$$q^T = [110 \quad 001 \quad 111 \quad 101 \quad 011 \quad 100 \quad 001 \quad 000]$$

Then the hash addresses for keys 53 and 100 are:

$$h_q(53) = h_q(00110101) = q(3) \oplus q(4) \oplus q(6) \oplus q(8)$$
$$= 001 \oplus 101 \oplus 100 \oplus 000 = 110 = 6\ (decimal)$$

$$h_q(53) = h_q(01100100) = q(2) \oplus q(3) \oplus q(6)$$
$$= 001 \oplus 111 \oplus 100 \ = 010 = 2\ (decimal)$$

Figure 3.7 shows a circuit implementation of the above example. Both, the examples, as well as the circuit of Figure 3.7 are self explanatory and, thus, we will not elaborate further.

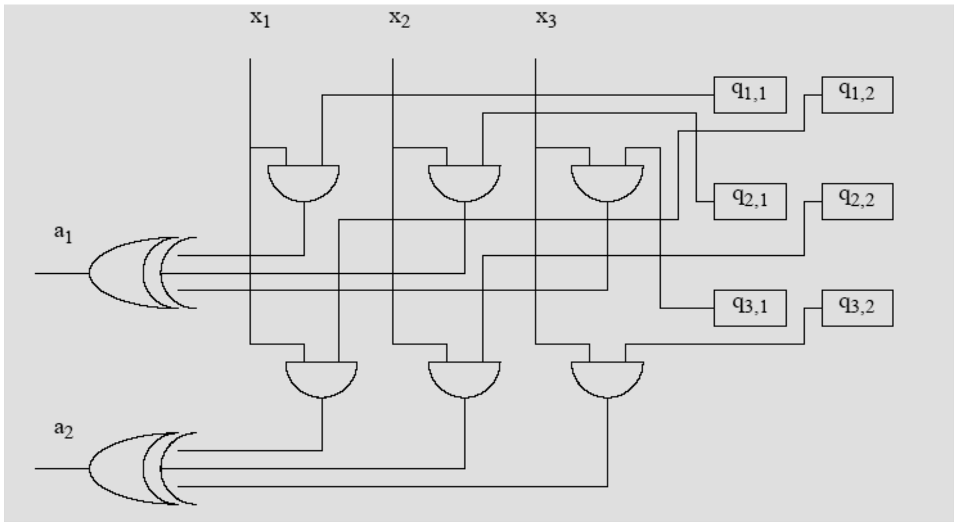


Figure 3.7: Hash address generator, key = $x_1x_2x_3$, hash matrix elements $=q_{i,j}$ and hash address = a_1a_2

3.3 Look-up Processor Configuration

A basic configuration of our approach is shown in Figure 3.8. We begin by grouping the database of prefixes into sets according to prefix length. The system employs a set of **W** counting Bloom filters where **W** is the length of input addresses, and associates one filter with each unique prefix length. Each filter is "programmed" with the associated set of prefixes according to the procedure described in subsection 3.1. It is important to note that while the bit-vectors associated with each Bloom filter are stored in embedded memory, the counters associated with each filter are stored in an external memory unit. This unit could be an SRAM/DRAM chip or the counters could be maintained by a separate control processor responsible for managing route updates. Separate control processors with ample memory are common features of high-performance routers **[**1**]**. A hash table is also constructed for each distinct prefix length. Each hash table is initialized with the set of corresponding prefixes, where each hash entry is a {prefix, next hop} pair. While we assume the result of a match is the next hop for the packet, more elaborate information may be associated with each prefix if so desired. The set of hash tables is stored off-chip in a separate memory device; for the rest of this section we will assume it is a large SRAM or DRAM chip. More about the hash table's implementation will be elaborated in Section 4 ("Implementation Details").

In order to perform a look-up operation, the input IP address is used to probe the set of **W** Bloom filters in parallel. One-bit prefix of the address is used to probe the filter associated with length one prefixes, two-bit prefix of the address is used to probe the filter associated with length two prefixes and so on. Each filter simply indicates match or no match. By examining the outputs of all filters, we compose a vector of potentially matching prefix lengths for the given address, which we will refer to as the hit vector. Consider an IPv6 example where the input address produces matches in the Bloom filters associated with prefix lengths 8, 17, 23, and 48; the resulting match vector would be {8, 17, 23, 48}. Remember that Bloom filters may produce false positives, but never produce false negatives; therefore, if a matching prefix exists in the routing table, it will be represented in the match vector. Note that the number of unique prefix lengths in the prefix database, $\mathbf{W_{dist}}$, may be less than **W (**Figure 1.4**)**. In this case, the Bloom filters representing empty sets will never contribute a match to the hit vector, valid or false positive. The search proceeds by probing the hash tables associated with the prefix lengths represented in the hit vector in order of longest prefix to shortest. The search continues until a match is found or the vector is exhausted.

Theoretical Performance

We now show the relationship between false positive probability and its effect on the throughput of the system. Note that for the performance analysis, hash probes denote the entire hashing sequence and not the actual number of memory access. We measure the throughput of the system as a function of the average number of hash probes per lookups, E_{exp}. The worst-case performance for the basic configuration of our system is W_{dist} hash probes per lookup. The number of hash probes required to determine the correct prefix length for an IP address is determined by the number of matching Bloom filters [1]. For an address which matches a prefix of length *l*, we first examine any prefix lengths greater than *l* represented in the hit vector. For the basic configuration of the system, we assume that all Bloom filters share the same false positive probability, *f*. Now, let B_l represent the number of Bloom filters for the prefixes of length greater than *l*. From [1] we know that probability that exactly *i* filter associated with prefix lengths greater than *l* will generate false positives is given by:

$$P_l = \binom{B_l}{i} \cdot f^i \cdot (1-f)^{B_l - i} \qquad (1)$$

For each value of *i*, we would require *i* additional hash probes. Hence, the expected number of additional hash probes required when matching a length *l* prefix is:

$$E_l = \sum_{i=1}^{B_l} i \cdot \binom{B_i}{i} \cdot f^i \cdot (1-f)^{B_l - i} \qquad (2)$$

which is the mean for a binomial distribution with Bl elements and a probability of success f. Therefore,

$$E_l = B_l \cdot f \qquad (3)$$

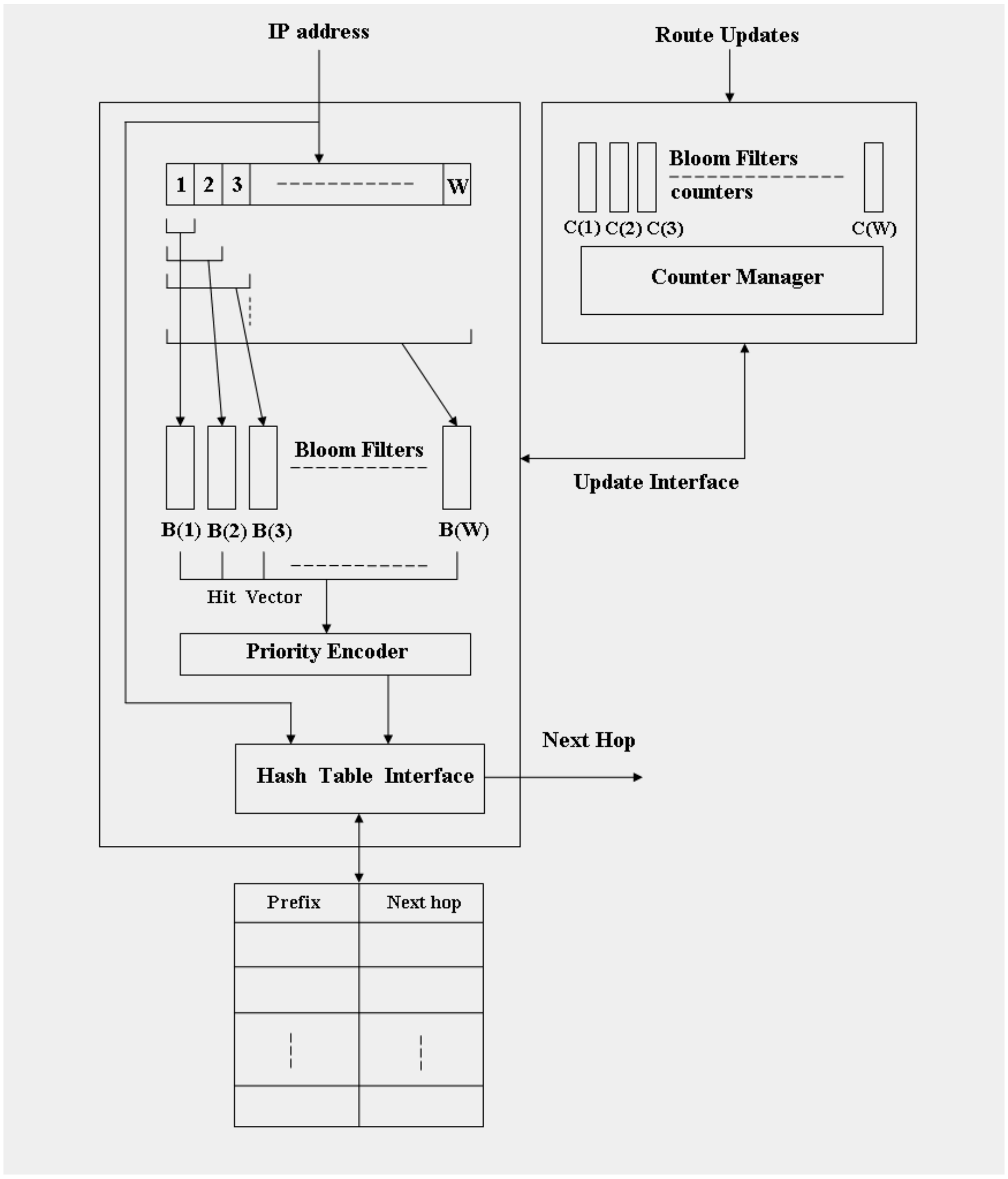


Figure 3.8: Basic configuration of Longest Prefix Matching using Bloom filters

Equation 3 shows that the expected number of additional hash probes for the prefixes of a particular length is equal to the number of Bloom filters for the longer prefixes times the false positive probability (which we assume is the same for all the filters). Let B be the total number of Bloom filters in the system for a given configuration. The worst case value of E_l, which we denote as E_{add}, can be expressed as:

$$E_{add} = B \cdot f \qquad (4)$$

This is the maximum number of additional hash probes per lookup, **independent of input address**. Since these are the expected additional probes due to the false positives, the total number of expected hash probes per lookup for any input address is:

$$E_{\exp} = E_{add} + 1 = B \cdot f + 1 \qquad (5)$$

where the additional one probe accounts for the probe at the matching prefix length. Note that there is the possibility that an IP address creates false positive matches in all the filters in the system. In this case, the number of required hash probes is:

$$E_{worst} = B + 1 \qquad (6)$$

Thus, while Equation 5 gives the expected number of hash probes for a longest prefix match, Equation 6 provides the maximum number of hash probes for a worst case lookup. Since both values depend on B, the number of filters in the system, reducing B is important for limiting the worst case. Note that for the basic configuration the value of B is simply W_{dist}. We will next present the methodology we used in order to reduce the number of required Bloom filters and hence reduce the expected and worst number of hash probes.

Reducing the Number of Bloom Filters

We have seen that the main problem with the configuration of Figure 3.8 is the large number of used Bloom filters. This is a problem because more Bloom filters results to more false positives. In order to address this problem, we reduce the number of distinct prefix lengths by employing the Control Prefix Expansion (CPE) technique described in Section 2 ("Related Work"). Our goal is to limit the worst case hash probes (due to false positives) to as few as possible without prohibitively large embedded memory requirements. However, even though CPE reduces the number of required Bloom filters, it has two disadvantages. First, the total number of prefixes increases and hence the average number of on-chip memory bits per prefix decreases. In addition, more prefixes require a larger hash table and thus a larger external RAM. Second, more updates must be executed in order for an expanded prefix to be updated. For example, if we expand prefix 1* so as to be stored in a Bloom filter supporting prefixes of length two, it will become prefixes 10* and 11*. Therefore, even though we are actually inserting/deleting only one prefix, two insertion/deletion operations must be performed. This is of grater importance for more specific matches (that is, for longer prefixes) where the update frequency is higher [7].

Clearly, the appropriate choice of CPE strides depends on the prefix distribution. As illustrated in the distribution of IPv6 prefixes shown in Figure 1.4, there is a significant concentration of prefixes from prefix length 32 to 48. In particular, 91.9% of the prefixes fell in the range 32 to 48. On the other hand, prefixes with lengths in the range 49 to 64 consist of only 5.7% of the database and in range 1 to 31 are just the 2.4% of the overall routing table. Therefore, the majority of Bloom filters will store distinct prefix length in the range 32 to 48 while for ranges 1 to 31and 49 to 64, filters will be employed more sparsely. From Figure 1.4 we can also observe that the number of prefixes which feel in the range 1 to 16 is so small that we can expand them all to length 16 without increasing the overall routing table size. Note that all global unicast addresses start with the bit sequence 001 hence we could store all the prefixes with lengths 1 to 16 in on-chip RAM, consuming only 64Kb of embedded memory while allowing the router to support 256 ports. All other prefixes are stored in Bloom filters. Figure 3.9 shows the final distinct prefix lengths (after CPE for a routing table with 400,000 entries) and the resulting prefix length distribution. The new routing table is 10.34% larger than before and it now has a total of 441,376 prefixes.

Performance Estimation for a Given Internet Traffic

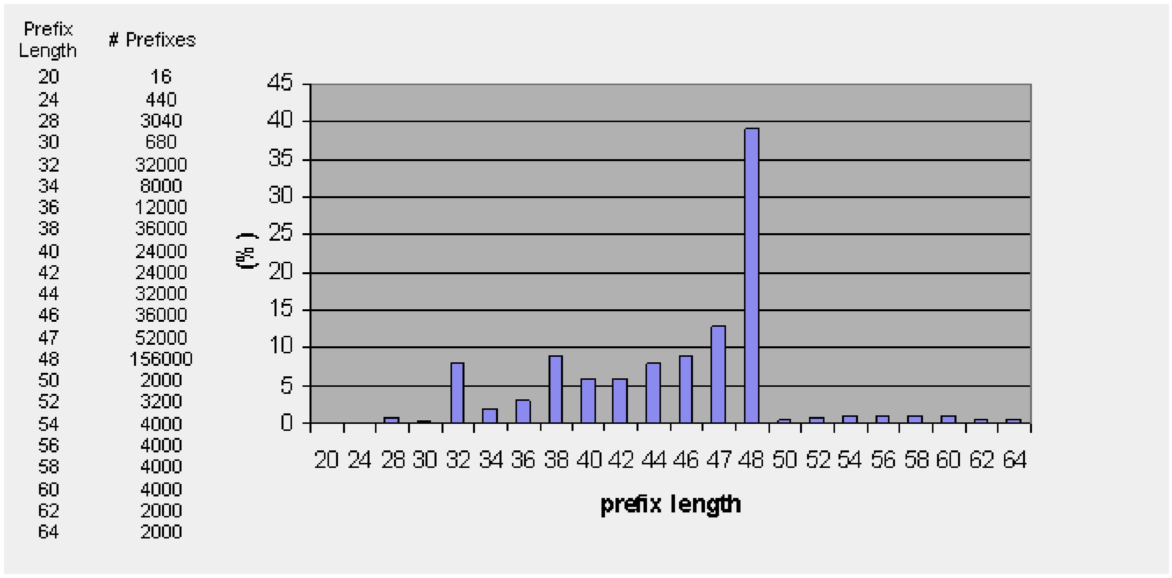
Someone can intuitively predict that the performance (given in memory accesses per IP look-up) is strongly depended on the LPM of each incoming IP. That is, if we have prior knowledge that all of the incoming prefixes will have a longest prefix match at length 64, then the overall false positive probability is zero. We can now illustrate how we can extract a mathematical equation that will predict the performance of the system when (a) that the traffic's distribution is known, (b) the false positive probability is different for each Bloom filter and (c) the load factors are not the same.
Let,

C_x	: The average cost in memory access when the matching prefix is at length x
H_x	: denotes the probability that an IP address has a longest matching prefix at length x
λ_x	: denote the load factor of Hashing Table Element storing prefixes with length x
$P_u(\lambda)$	: The average unsuccessful probe length for a hashing table with load factor λ
$P_s(\lambda)$	: The average successful probe length for a hashing table with load factor λ
f_x	: The false positive probability of the Bloom filter storing prefixes with length x

Therefore, the expected number of memory accesses is:

$$E_{\exp} = H_{20} \cdot C_{20} + H_{24} \cdot C_{24} + H_{28} \cdot C_{28} + \ldots + H_{64} \cdot C_{64}$$

This holds when $H_{20}+H_{24}+H_{28}+...+H_{64}=1$. That is, when the probabilities to have an IP address with longest prefix match at lengths 1 to 16 is zero. This assumption holds because, as we can see in the Routing table's distribution, the number of prefix lengths smaller than 20 is extremely small compare to



Prefix Length	# Prefixes
20	16
24	440
28	3040
30	680
32	32000
34	8000
36	12000
38	36000
40	24000
42	24000
44	32000
46	36000
47	52000
48	156000
50	2000
52	3200
54	4000
56	4000
58	4000
60	4000
62	2000
64	2000

Figure 3.9 Prefix length distribution after employing Control Prefix Expansion

prefix lengths larger than 20 bits. Now, given that we have a match at prefix length 64 the average memory accesses are:

$$C_{64}=P_s(\lambda_{64})$$

likewise for prefix length 63:

$$C_{62}=P_s(\lambda_{62})+P_u(\lambda_{64})\cdot f_{64}$$

for prefix length 60:

$$C_{60}=P_s(\lambda_{60})+P_u(\lambda_{62})\cdot f_{62}\cdot(1-f_{64})+P_u(\lambda_{64})\cdot f_{64}\cdot(1-f_{62})+f_{64}\cdot f_{62}\cdot\left(P_u(\lambda_{64})+P_u(\lambda_{62})\right)$$

Given that we have a match at prefix length 58 the average cost in memory access is,

$$\begin{aligned}C_{58}&=P_s(\lambda_{58})+P_u(\lambda_{62})\cdot f_{62}\cdot(1-f_{64})\cdot(1-f_{60})+P_u(\lambda_{64})\cdot f_{64}\cdot(1-f_{62})\cdot(1-f_{60})\\&+P_u(\lambda_{60})\cdot f_{60}\cdot(1-f_{62})\cdot(1-f_{64})\ +\ f_{64}\cdot f_{62}\cdot(1-f_{60})\cdot\left(P_u(\lambda_{64})+P_u(\lambda_{62})\right)\\&+f_{62}\cdot f_{60}\cdot(1-f_{64})\cdot\left(P_u(\lambda_{62})+P_u(\lambda_{60})\right)+f_{64}\cdot f_{60}\cdot(1-f_{62})\cdot\left(P_u(\lambda_{64})+P_u(\lambda_{60})\right)\\&+f_{64}\cdot f_{62}\cdot(1-f_{60})\cdot\left(P_u(\lambda_{64})+P_u(\lambda_{62})\right)+f_{64}\cdot f_{62}\cdot f_{60}\cdot\left(P_u(\lambda_{64})+P_u(\lambda_{62})+P_u(\lambda_{60})\right)\end{aligned}$$

Because the false positive probability, for each Bloom Filter, as well as the load factor for each hashing table element is different, the extraction of a compact mathematical equation describing the performance is impractical. By making the assumption that all Bloom filters and hash tables share the same false positive probability and load factors respectively, we can modify Equation 2 so as to reflect a scenario with a given traffic distribution Now the new costs can be described by the equation:

$$C_l = P_s(\lambda) \ + \ P_u(\lambda)\cdot\sum_{i=1}^{B_l} i\cdot\binom{B_l}{i}\cdot f^i\left(1-f\right)^{B_l-i} \quad , \quad \binom{B_l}{i} = \frac{B_l\,!}{i!(B_l-i)!}, 0!=1 \qquad (7)$$

Prefixes per Bloom Filter	Simulation Results (Mem. accesses/lookup)	Theoretical Results (Mem. accesses/lookup)	Deviation (%)
1000	1,008636	1,0102041	0,155226058
2000	1,021432	1,0208359	0,058393323
3000	1,03303	1,0319972	0,100077791
4000	1,044136	1,0443264	0,018231848
5000	1,062191	1,0602069	0,187142717
6000	1,086227	1,0853276	0,082868988
7000	1,130097	1,1297682	0,029103315
8000	1,208506	1,2077504	0,062562596
9000	1,335773	1,3362604	0,036474927
10000	1,527695	1,5329833	0,344967881
11000	1,80605	1,8142556	0,452284673
12000	2,180905	2,1935289	0,575506436
13000	2,658336	2,6805679	0,829372761
14000	3,263549	3,2813687	0,543056926
15000	3,964855	3,9986402	0,84491723

Table 3.4: Simulation Vs Theoretical Results

In order to evaluate the correctness of our analysis, we compare the theoretical results with results taken from simulating the simplified model presented above. The simulation results are the average of 20 simulations with different IP sequences first programmed and then queried.

Simulation Configuration:

- All the Bloom filters are the same and they store equal number of prefixes. That is, the false positive probability is the same for all Bloom filters.
- The priory probability that an IP address will have a match at a specific prefix length is the same for all prefix lengths ($H_{20} = H_{24} = ... = H_{64} = 1/22$).

The results are summarized in Table 3.4 and are graphically illustrated in Figure 3.10. In Figure 3.10, the solid line indicates the theoretical performance given by the performance Equation 7 and the dots indicate the results from the simulations. As we can observe, the theoretical model we have derived can be used to predict the performance of this IP look-up scheme under a particular traffic condition. Moreover, we can see that the number of memory accesses is increasing exponentially with the number of prefixes been stored in the routing table. This is something that we expect to happen because of the exponential relation between the false positive probability and the number of prefixes stored in the Bloom filter and, in addition, because of the exponential relation between the load factor and the average probe length for hashing.

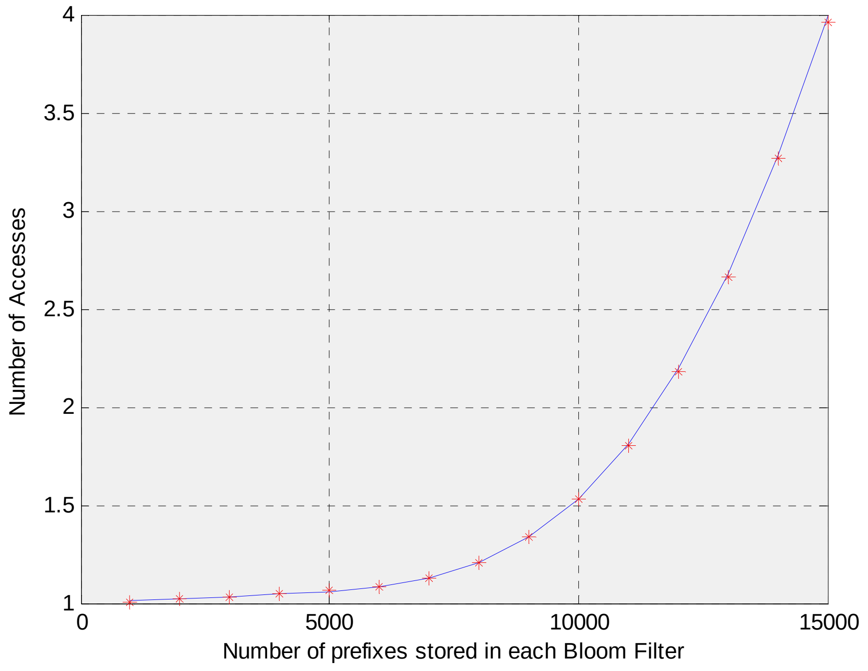


Figure 3.10: Theoretical Vs Simulation results of performance when every prefix length is equally populated

3.4 Selecting the Hardware Implementation Technology

When it comes to hardware implementation, the major issue to address is the support of variable prefix length distributions. From previous discussions, it is clear that Bloom filters which are designed to suit a particular prefix length distribution tend to perform better. However, an ASIC (Application Specific Integrated Circuit) design optimized for a particular prefix length distribution will have sub-optimal performance if the distribution varies drastically. Note that this can happen even if the new distribution requires the same aggregate embedded memory resources as before. Thus in spite of the available embedded memory resources, this inflexibility in allocating resources to different Bloom filters can lead to poor system performance **[**1**]**. The ability to support a lookup table of certain size, irrespective of the prefix length distribution is a desirable feature of this system.

Therefore, we have selected to implement our system using Field Programmable Gate Array (FPGA) technology. Modern FPGAs have on-chip RAMs with more than one port (typically dual port) that can be exploited to create Bloom filters [22]. The main advantage of embedded RAMs in FPGAs is that they are not permanent hardwired. Thus, in the case where the original prefix length distribution changes, the FPGA can be reconfigured so as to adjust the assignment of embedded RAM to Bloom Filters accordingly.

Moreover, we can take advantage of FPGAs' reconfigurable nature in order to support integrated memory controllers. Such controllers could be employed for managing the external RAMs we use for the counting Bloom filters and for the hash tables. If we use ASIC instead of FPGA, these controllers could be either integrated into the chip or purchased as an individual chip. The first choice limits the user's flexibility to select which type of external RAM to use e.g. SRAM or DRAM. While the second, consumes extra board area and increases the overall system's cost. On the other hand, when we use FPGA, the designer can write in VHDL any memory controller according to the user's demands for speed and/or money. In this way, we have different versions of our system without producing expensive and labor intensive ASIC masks [23].

For the purpose of this work, we have selected to use a Xlinx FPGA from the Virtex 4 family because they provide the largest number of on-chip Block RAMs. In particular, Virtex 4 FX140 supports 552 Block RAMs 18 Kb each, enables the use of over 1 Mb of distributed RAM and provides 142,148 4-input LUTs (Look-Up Tables) [24].

4. Implementation Details

In the previous section we have presented the architecture of our look-up scheme and developed a framework for its internal structure. Taking all these under consideration, our major implementation target is to create a system with the ability of realizing a 100% external RAM utilization. If we succeed on this, our goal of developing a look-up processor with throughput close to one look-up per memory access will be met. Our second objective is to keep the required recourse well within the limits of a modern FPGA thus making the development of our look-up processor feasible. Throughout this section we will present the look-up processor's inputs and outputs, its internal organization and show the implementation details of all its subsystems.

4.1 Look-up Processor's Inputs and Outputs

Figure 4.1 shows the inputs and outputs of the look-up processor. When we issue a look-up command, we provide the look-up processor with the destination address of the IPv6 datagram we want to route and we expect from the processor to give us the output port that corresponds to the most specific destination (that is, of the longest matching prefix). On the other hand, when we issue an update command we must provide the look-up processor with the prefix to be update, the length of that prefix, the corresponding output port (only for the insertion operation), and the type of update to be perform. As we can see, there are three different types of instructions that can be issued to the look-up processor. These instructions are (a) Look-up (b) Insertion of a new destination (c) Deletion of an existing destination. However, sometimes the processor has no commands to execute hence a forth type of command, indicating absents of instruction, must be used. The Status output (Figure 4.1) is used in order for the processor to communicate with the other components of the line-card and with the Control Processor. Table 4.1 summarizes all the inputs and outputs showing the number of bits, the name of each field and a brief description.

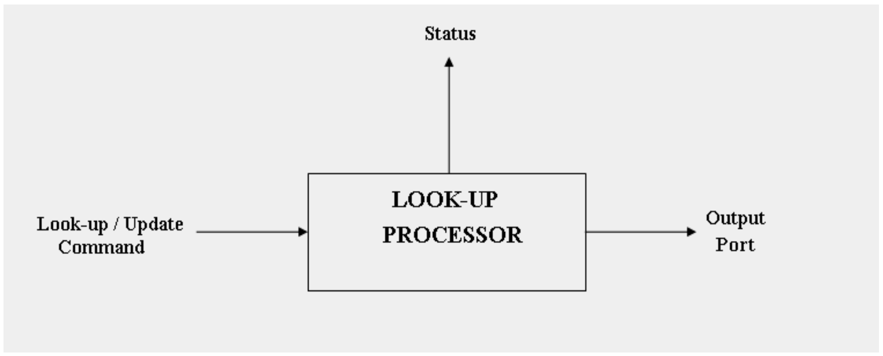


Figure 4.1: Look-up Processor's Inputs and Outputs

INPUT

Field	Opcode	Address	Prefix Length	Update Port
Number of bits	2	128	5	8
Description	"00": No operation "01": Look-up "10": Insertion "11": Deletion	The Dest. Address of the IPv6 datagram	The length of the prefix to be inserted or deleted	The output port to be selected when we have a LPM at the new prefix we are inserting

OUTPUTS

Name	Status	Output Port
Number of bits	2	8
Description	"00": No reports "01": Output port is valid "10": Update completed "11": System is busy	The port to which the IPv6 datagram must be forwarded

Table 4.1: The look-up processor's basic inputs and outputs

4.2 Look-up Processor's Organization

Figure 4.2 shows how the look-up processor is internally organized .When an instruction is issued, the processor first calculates the hashing function for all prefix lengths supported. This operation is performed by the *Hash Function Module* (HFM). In addition to calculating the hash addresses, the HFM integrates all the pipeline registers used to propagate the instruction's fields to the rest of the processor. These fields are the IPv6 Address, the opcode, the prefix length and the update port.

Look-up Operation

After all the hash addresses are calculated, the processor decides whether an update or a look-up operation is to be performed. This decision as well as the counters update process (that is, the counters used to implement the Counting Bloom filters), are handled by the *Counting Bloom Filters Control Unit* (CBFCU). The decision is performed by selecting which hash addresses are to be supplied to the Bloom filters. This is done with the use of the *Update* signal. When we have a look-up instruction, the CBFCU selects the *Hash Addresses* generated by the HFM. In addition to the selection of the hash addresses, during this stage, the *Hash bits* and *Spreader* signals are generated by the XOR gates shown in Figure 4.2. The Hash bits are used for probing the external *Hash Table*. The use of the *Spreader* signal will be better explained in subsection 4.3.

Now that the hash addresses used for probing the Bloom filters are selected, the processor performs the actual Bloom filter queering operation. After the Bloom filters' access is completed the responses from all Bloom filters (hit vector) are supplied to the *Engine Scheduler* (ES). The ES is then trying to find an available *Hash Engine* so as to be assigned the look-up operation. If no Hash Engine is free, the system is stall until one of the engines becomes available.

When a look-up operation has been assigned to it, a Hash Engine begins execution by first inspecting the hit vector. If the hit vector indicates that only one Bloom filter has reported a possible hit, the engine calculates the hash address for that particular prefix. On the other hand, if multiple Bloom filters indicate a match, the Hash Engine starts the probing sequence by first examining the longest of the matching prefixes. If the longest of the matching prefixes turns out to be a false positive, the Hash Engine selects the second longest match and restarts the probing sequence. This goes on until either the Hash Engine finds a match or until all indicated possible matches turn out to be false positives.

Whenever a Hash Engine wants to access the external RAM (Hash Table) it generates a *Request for Access* to the *Memory Scheduler* (MS). If during a particular clock cycle only one engine request to access the Hash Table, the Memory Scheduler allows that engine to access the Memory Bus. However, when multiple engines request to access the memory, then the MS decides which one is to be permitted to perform its probing operation. Finally, when a Hash Engine finds a match is then ready to report the port to which the datagram must be forwarded. This is done by notifying the *Results* unit.

Update Operation

When an update instruction is issue, the HFM generates the *Addresses for Update* signal (Figure 2.4). This signal contains all k hash addresses that will be used in order to either program a Bloom filter with a new member (prefix) or remove an existing member from the filter. We here note that when an update operation is issue, the length of the prefix, which is going to be inserted or deleted, is provided by the Control Processor. Hence, even though the HFM generates hash addresses for all supported prefix lengths, we are only interested with the hash address of the update prefix length. The *Addresses for Update* signal consists of these hash addresses.

As soon as the HFM generates the *Addresses for Update* signal, CBFCU begins updating the counters which comprise the Counting Bloom Filters. This is done in parallel with the rest of the look-ups. That is, the CBFCU updates all the counters while the rest of systems continue with the next look-up instruction. We here note that the next instructions could only be look-ups because, with only one CBFCU unit, only one update can be computed at a time. More about how the CBFCU performs the update of counters will be discuss in subsection 4.5. When CBFCU completes the counters' modification it then (a) stalls the HFM (b) selects the *Update Hash Addresses* for probing the Bloom Filters (this is done using the *Update* signal) (c) uses the *Control Interface* so as to inform the filter which operation it must perform and (d) notifies the *Report* unit that the update is completed. The *Report* unit then sets the *Status* signal to "10" (Table 4.1) and, in this way, lets the Control Processor know that a new update instruction can now be issue.

In parallel with the counters been updated (by the CBFCU), the look-up processor continues with the modifications that have to be performed to the external *Hash Table*. That is, if we have a deletion or an insertion operation an entry must be removed form or inserted to the Hash Table respectively. This is done by the *Hash Engines*, just like in the case where we have a look-up. However, the actual processes performed by the engines are different than those of a look-up. Now, the hit vector is not examined because the prefix length of the prefix to be updated is already known. Hence, only one probe sequence must be performed. The external memory is accessed the same why as if we had a look-up and thus will not be elaborated further.

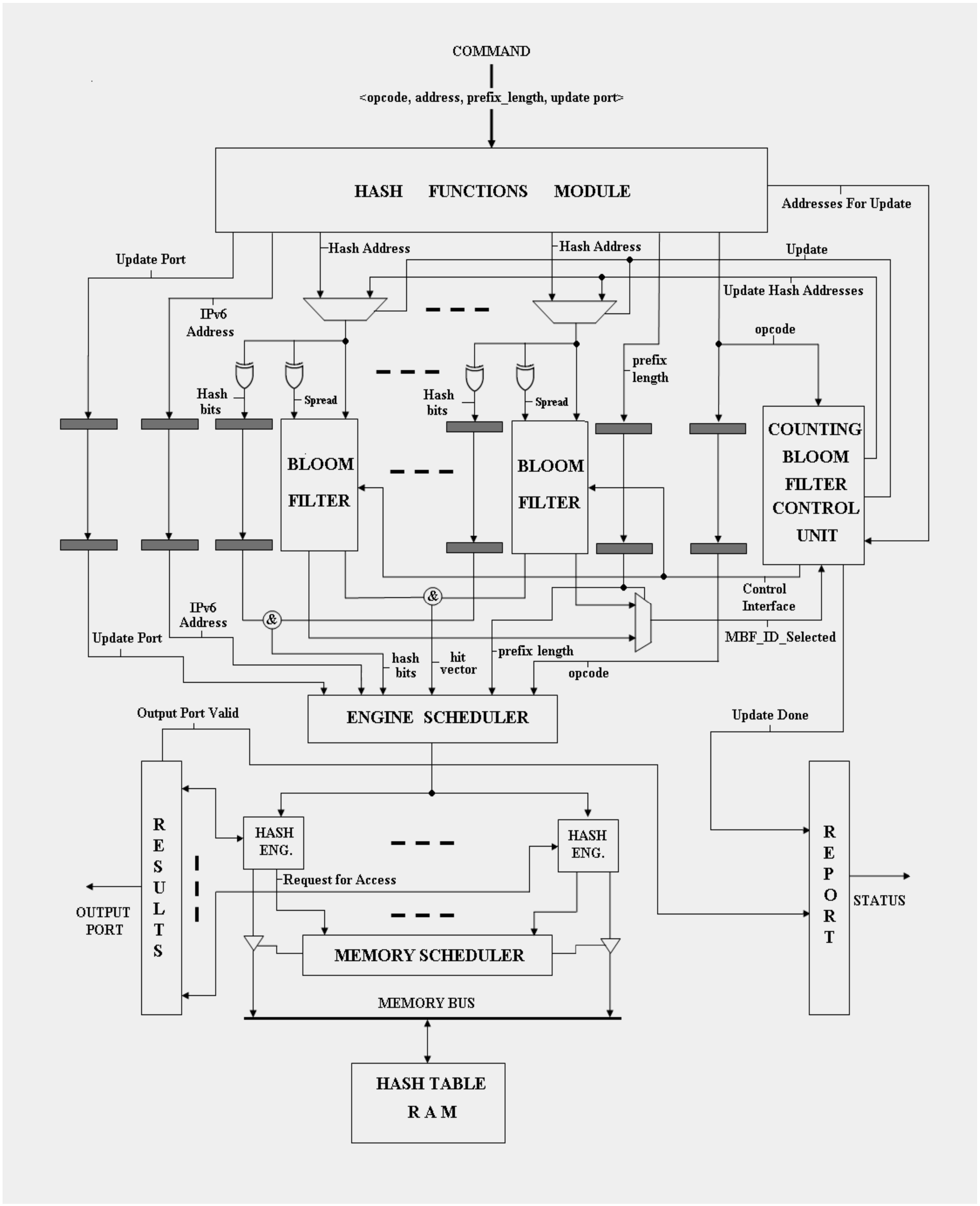


Figure 4.2: Look-up Processor’s Organization

4.3 Implementing the Bloom Filters

In order to reduce the false positive probability considerably, a long bit vector is required. Even for supporting the programmability for hundreds of prefixes, many thousand bits are required. The cost of using the on-chip flip-flops for this purpose is too high. However, modern FPGAs have on-chip RAMs with more than one port (typically dual port) that can be exploited to create the Bloom filter vector [22]. For instance, the Xilinx Virtex 4 VFX140 chip has 552 on-chip RAMs [24] each with two ports and with the ability to operate at 333 Mhz. Each Block RAM can be configured as a single bit wide and 16,384 bits long vector requiring two 14-bit addresses and giving a bit look-up throughput of 2 bits per clock cycle. We now describe how these on-chip RAMs can be used for building the basic Bloom Filter.

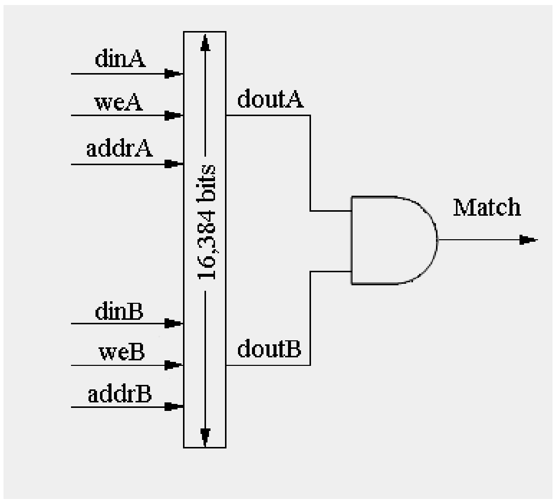


Figure 4.3: A Bloom filter using one Dual Port RAM

Figure 4.3 shows how a Block RAM can be used to construct a Bloom filter. The Block RAM is configured as a single bit wide and 16,384 bits long bit array. It has two read/write ports, both of which can be used to look-up the bit values corresponding to two distinct hash functions. Thus, this Bloom filter can support k=2 hash function, m=16,384 in the array and hence can support $n = (m/k) \cdot \ln 2 = 5,678$ prefixes with P(false positive) = $(1/2)^2$. From now on we will refer to this Bloom filter as Bloom Filter Element (BFE) since multiple such Bloom filters are needed to create a Bloom filter with smaller false positive probability [22].

Creating the mini-Bloom filter (MBF)

By using multiple BFE to store the same set of strings, the false positive probability can be reduced. For instance, if two BFEs are used then the false positive probability will be $\left((1/2)^2\right)^2 = (1/2)^4$, if three

BFEs are used then the false positive probability will be $\left((1/2)^2\right)^3 = (1/2)^6$ and so on. For this work, five BFEs are used to achieve a false positive probability of $\left((1/2)^2\right)^5 = (1/2)^{10} = 0.0000977$ and a storage capacity of 5,678 prefixes. This set of five BFEs is called a *mini-Bloom filter (**MBF**)* [22].

From Section 3 we have seen that if we want to create a Bloom filter with the ability to support a deletion operation we must use the counting Bloom filters configuration. As we will see later on, the actual counters are stored in an external RAM in order to preserve the embedded RAM solely for implementing the Bloom filter vector. In addition, another unit called Counting Bloom Filters Control Unit (CBFCU) is employed in order to handle the counters and to manage the update operations. However, this modification complicates the implementation of the MBFs. Therefore, in order for a MBF to support update operations an additional mechanism must be added.

During the design of the system each MBF is assigned a 9-bit MBF identifier, which is unique for every MBF of the system. When the CBFCU initiates an update it sends to all the MBFs a control MBF identifier (MBF_ID_control) and a 20 bit control signal. Each MBF compares its MBF_ID with the one been sent by the CBFCU and if the two identifiers are equal the MBF knows that the update is designated for it. It then, automatically uses the 20 bit control signal in order to extract the type of update to be performed. The structure of the control signal is quite simple. The first bit (LSB) is the write enable of the first port (weA, Figure 4.3) of the first BFE, the second bit is the data in of the first port (dinA, Figure 4.3), the third bit is the weB of the first BFE, the forth is the dinB of the first BFE, the fifth is the waA of the second BFE and so on. We here note that if the MBF_ID been sent by the CBFCU does not match the MBF_ID of a MBF then all the bits of the control signal are cleared, hence no change is to occur for that MBF. The complete version of the mini-Bloom Filter with the MBF Control Unit (MBF_CU) is illustrated in Figure 4.4.

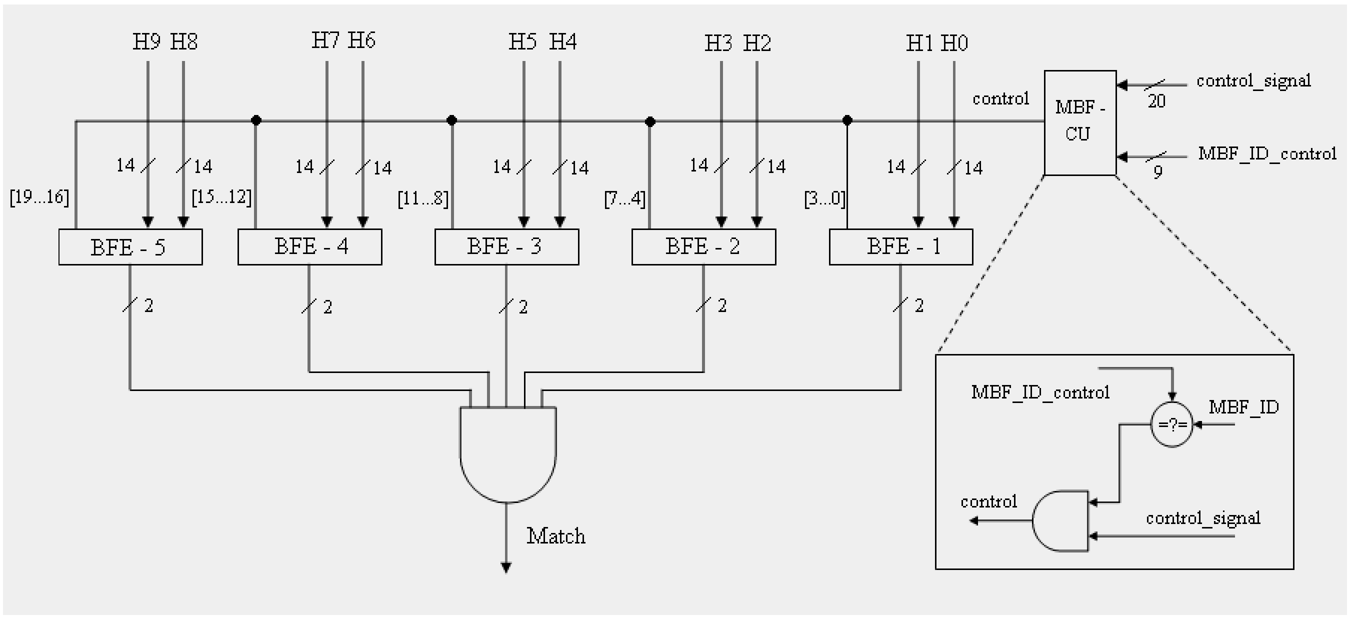


Figure 4.4: The mini-Bloom Filter (MBF)

Creating the Large Bloom Filter (LBF)

A MBF can support only a small number of prefixes. To create a Bloom filter with a bigger capacity, multiple MBFs are used in parallel. This set of MBFs is called *Large Bloom Filter (**LBF**)*. Figure 4.5 shows the schematic of the LBF constructed using *r* MBFs. This LBF has a capacity of *rx5678* prefixes with the same false positive probability as if 5678 prefixes are stored in a single MBF. A prefix is stored in only one of the MBFs. The MBF for storing a particular prefix is chosen randomly by calculating a hash value over the ten hashing addresses generated by the Hash Matrices Module (HMM). During the query process, the same hash value will be calculated and hence the same MBF will be probed for the presence of the prefix. For probing a particular LBF, *k* hash values are calculated and they are routed through a decoder to the MBFs chosen for probing. The output of the same MBF is then routed out.

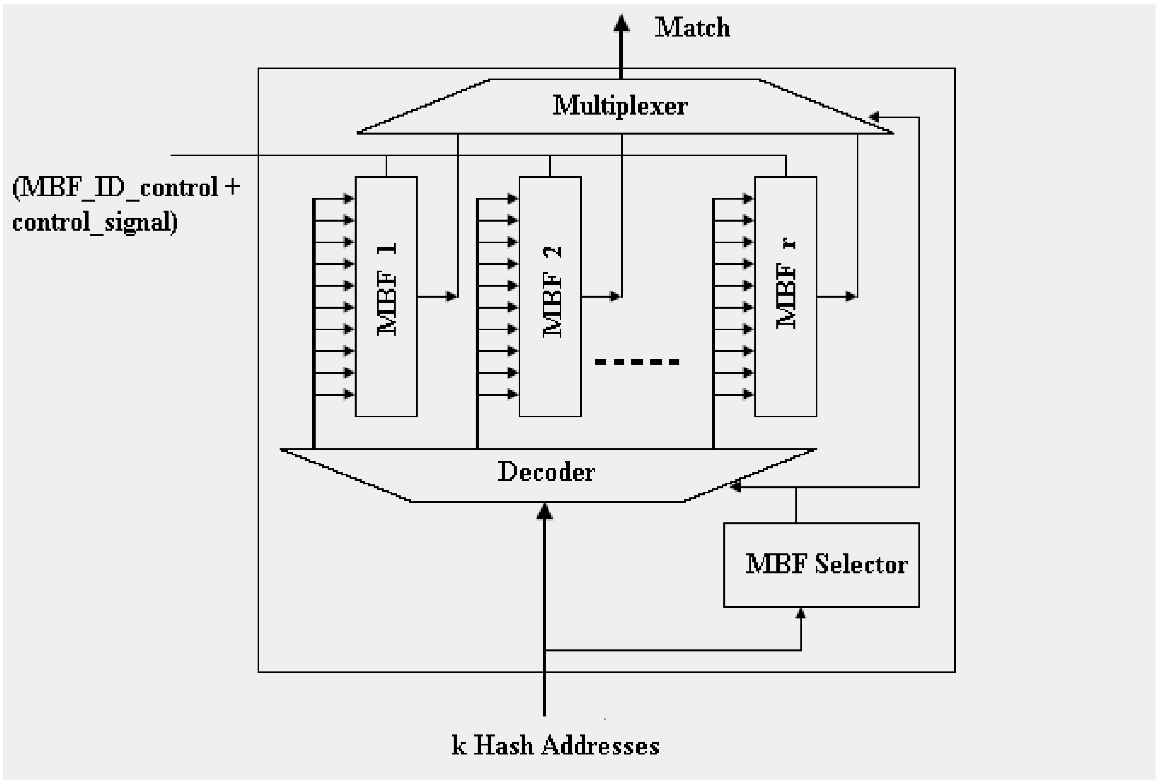


Figure 4.5: The Large Bloom Filter

The LBF of Figure 4.5, even though illustrates the basic concept of how we can construct a Bloom filter by using multiple MBFs, it has some important drawbacks. First of all, the **MBF Selector** must produce a hashing address that spreads the prefixes uniformly in all the MBFs. If r is a power of 2, any hash function examined in Section 3 could be used. However if r is not a power of 2, in order to uniformly spread the prefixes, MBF Selector must perform a modulo operation. Even though implementing modulo in hardware is feasible, the delay of such a unit will be to long hence degrading the performance of the overall system. Moreover, the MBF selector is also used in order to choose the MBF_ID so as to be used for the update operation. The MBF_ID is 9-bits long and given that a LBF could use up to 64 MBFs the delay of selecting a particular identifier is prohibitive. Therefore, in order to address this problem we have exploited the unused embedded Block RAMs. For implementing 100 MBFs we use 500 out of 552 on-chip RAMs and 4 are used for the direct table, so we have 48 of these RAMs that we could use to create the MBF Spreader and the MBF_ID. The second problem with the LBF of Figure 4.5 is the MUX used for selecting the response of the MBF selected by the MBF Spreader. The delay of this circuit comes to be added with the access time of the on-chip RAMs hence increasing the minimum period of the system. We address this problem by storing the responses from all the MBFs to a register and thus adding another pipeline stage to our system. The schematic of a complete LBF, with 3 MBFs, along with all the modifications mentioned above is shown in Figure 4.6.

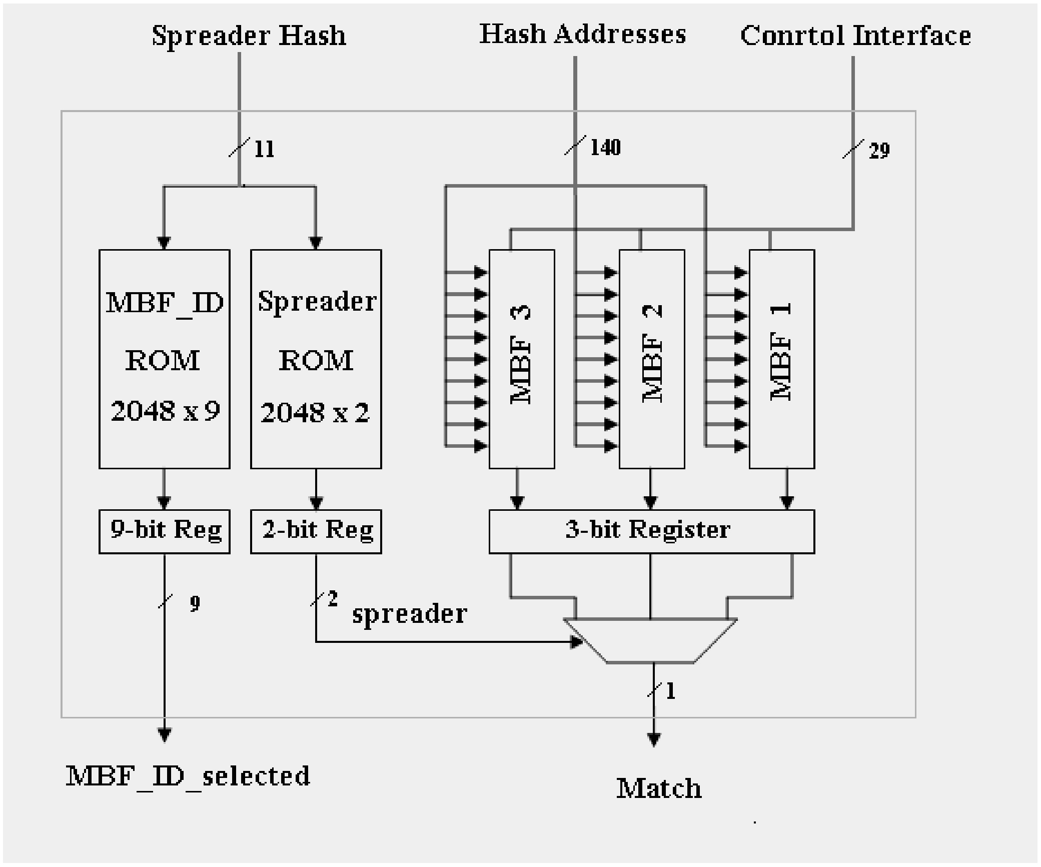


Figure 4.6: Schematic of a complete LBF constructed using 3 MBFs

Now that we have illustrated how we can implement large Bloom filter in FPGAs, we present the amount of MBFs used for storing each supported prefix length, in Table 4.2. In addition, Table 4.2

shows the amount of entries consumed, by each prefix length, in the hash table and their corresponding load factors.

A/A	Prefix Length	%	# Prefixes	MBF#	Entries / Hash Table	Load factor	False Positive Prob.
1	20	0,00004	16	1	128	0,125000000	$8{,}002x10^{-28}$
2	24	0,0011	440	1	4096	0,107421875	$1{,}53x10^{-13}$
3	28	0,0076	3040	1	32768	0,092773438	$8{,}9x10^{-5}$
4	30	0,0017	680	1	8192	0,083007813	$1{,}29x10^{-11}$
5	32	0,08	32000	6	262144	0,122070313	$6{,}.29x10^{-4}$
6	34	0,02	8000	2	65536	0,122070313	$7{,}.41x10^{-5}$
7	36	0,03	12000	3	131072	0,091552734	$7{,}\ 41x10^{-5}$
8	38	0,09	36000	8	524288	0,068664551	$1{,}82x10^{-4}$
9	40	0,06	24000	5	262144	0,091552734	$2{,}\ 94x10^{-4}$
10	42	0,06	24000	5	262144	0,091552734	$2{,}94x10^{-4}$
11	44	0,08	32000	6	262144	0,122070313	$6{,}29x10^{-4}$
12	46	0,09	36000	8	524288	0,068664551	$1{,}82x10^{-4}$
13	47	0,13	52000	12	524288	0,099182129	$1{,}37x10^{-4}$
14	48	0,39	156000	33	2097152	0,074386597	$2{,}63x10^{-4}$
15	50	0,005	2000	1	32768	0,061035156	$2{,}28x10^{-7}$
16	52	0,008	3200	1	32768	0,097656250	$1.25x10^{-5}$
17	54	0,01	4000	1	65536	0,061035156	$7.4x10^{-5}$
18	56	0,01	4000	1	65536	0,061035156	$7.4x10^{-5}$
19	58	0,01	4000	1	65536	0,061035156	$7.4x10^{-5}$
20	60	0,01	4000	1	65536	0,061035156	$7.4x10^{-5}$
21	62	0,005	2000	1	32768	0,061035156	$2.28x10^{-7}$
22	64	0,005	2000	1	32768	0,061035156	$2.28x10^{-7}$

Table 4.2: System basic configuration's parameters

From Table 4.2 we can see that the load factors and the false positive probabilities for longer prefix lengths are kept low. In Section3 we have shown that the false positives when generated from Bloom filters holding more specific routes (that is, longer prefix lengths) are more expensive than from less specific routes (that is, smaller prefix lengths). Therefore, we have done this in order to reduce the penalty of false positive as well as their frequency of occurrence. The penalty is lessen by reducing the load factors, while the false positives are reduced by reducing the false positive probability.

The total number of entries in the hash table is 5,353,600. Hence, a 23-bit hash address is required in order to address all its entries. More about how have implemented hashing, why we have selected using different hash tables for each prefix and why the number of entries for each prefix is a power of two will be discussed in subsection 4.6.

4.4 Implementing the Hash Functions

We have seen in Section 3 that the hash functions used belong to a Class called Universal H_3 and that hash function $h_q(x): A \to B$ (A: key space, B: address space) is defined as,

$$h_q(x) = x(1)\cdot q(1) \oplus x(2)\cdot q(2) \oplus \ldots \oplus x(i)\cdot q(i)\,,$$

where $\cdot$ denotes the bitwise AND operation and $\oplus$ the bitwise exclusive OR (XOR) operation. A circuit implementing a hash function that produces a 2-bit hash value for a 3-bit Key is shown in Figure 4.7. The criteria used for the dimensions of the hash matrix are summarized in Table 4.3.

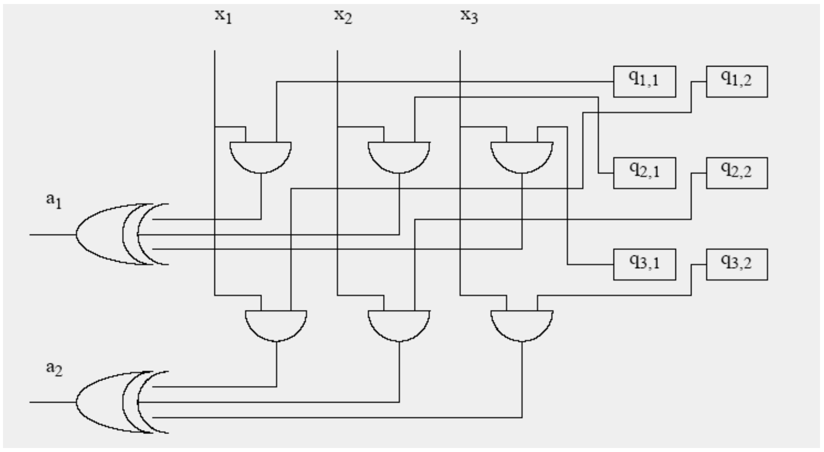


Figure 4.7: A hash function producing a 2-bit hash value for a 3-bit Key

Therefore, in order to create ten hash functions, for this work, we have created ten randomly generated hash matrices with dimensions 64 x 14 (rows x columns). In the rest of this subsection we will describe how we have implemented the ten hash functions so as to operate at 300 MHz.

Number of rows	**Number of columns**
Number of bits of the Key	Number of bits of the hash value

Table 4.3: Dimensioning the Hash Matrix

Pipelining the Hash Functions

We have already seen that the hash functions can be calculated cumulatively. That is the results calculated over the first *i* bits can be used for calculating the hash functions over the first *i* + 1 bits [21]. This property can be used in order to divide the required logic into separate pipe stages. Figure 4.8 shows the processes of dividing the logic of a hash function which produces a k-bit hash value, for a j-bit key, into n pipe stages. At every pipe stage j/n – bits of the key are used for calculating a partial hash value. The partial hash value is then XORed with the partial hash value of the previous pipe stage. Finally, at the end of the last stage, the final hash value is calculated.

The same methodology was used for creating the hash functions of this system. However, in contrast to the example shown in Figure 4.8, our system requires the extraction of partial hash functions at different bit intervals of the key. For example, no intermediate hash value is required for the first 16 bits because prefixes with lengths from 1 to 16 are stored in a direct table and thus are not programmed in any Bloom filter. On the other hand, for bits within the interval 46 to 48 a partial hash function is needed for every bit because a Bloom filter is used for prefix length 46 and for prefix length 47 and for 48. We here note that a partial hash value is extracted only if it corresponds to a prefix length for which a Bloom filter is used. Table 4.4 summarizes the partial hash values calculated at each pipe stage.

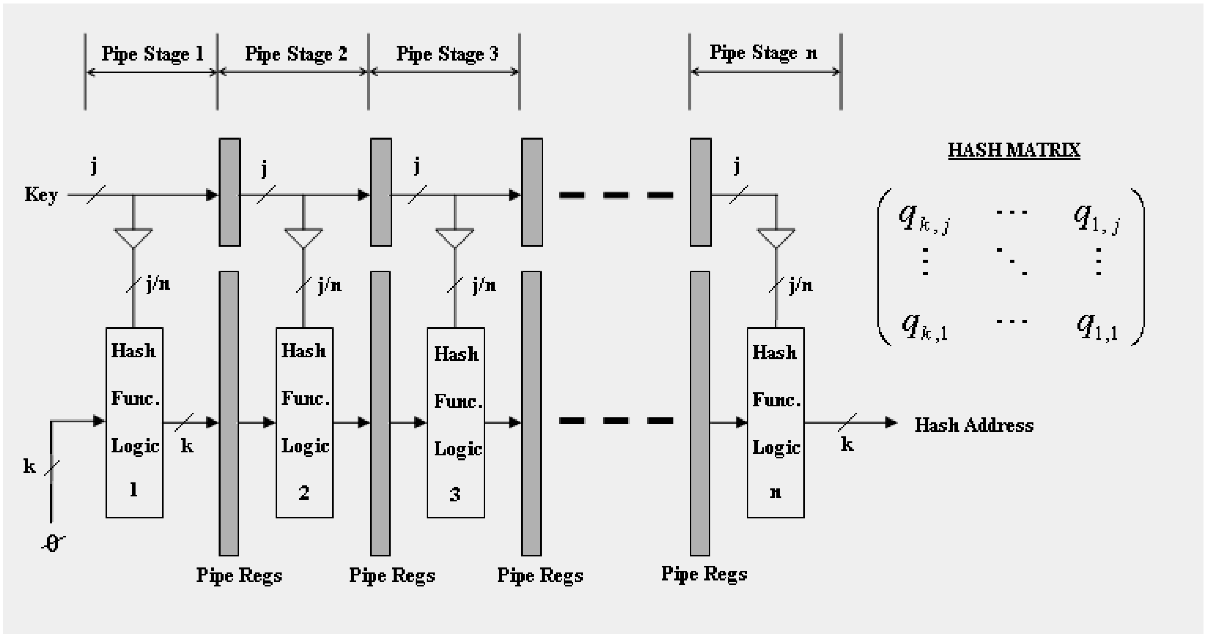


Figure 4.8: Pipelining a Hash Function that produces a k-bit hash value for a j-bit key, into n pipe stages

The criterion we used, for selecting how many bits of the IPv6 address are to be used at each pipe stage, was to hold the period below 3.3 ns. The imbalance usage of bit intervals, at each pipe stage, is due to the way a FPGA implements a logical function. For example, if a multiple input logical function is to be implemented and we don't need any intermediate result, a CLB (Complex Logic Block) is fully utilized. The internal connections of such a block as well as its internal logic (LUTs, MUXs, XORs, carry logic, flip-flops, etc) are much faster than if we have used only a part of CLB in order to implement a simple logical function and then route the result to a neighboring unit for further process. A CLB and a Slice of a Xilinx FPGA are shown in Figure 4.9.

Pipe Stage	Hash Addresses for Prefix Lengths	Number of Bits of the IPv6 Address Used
1	20,24	24
2	28,30,32,34	8
3	36,28,40,42	8
4	44,46,47,48	6
5	50,52,54,56	8
6	58,60,62,64	8

Table 4.4: Partial hash values calculated at each pipe stage

Selecting the Update Hash Addresses

When we have an update operation we are only interested in calculating the hash addresses of a particular prefix length. From now on we will refer to the hash addresses, of the prefix length to be updated, as Update Hash Addresses (UHA). This is in contrast to the IP look-up operation, where all supported prefix lengths' hash addresses must be calculated in order to select the longest matching prefix. A straightforward solution is to implement a 32x1 140-bit MUX, use the prefix length field as select and then route the UHA to the CBFCU. Unfortunately, such a circuit has a delay much larger than the 3.3 ns limit. Therefore, in order to address this problem we have integrated the selection of the UHA into the first 6 pipe-stages. That is, we select the UHA in parallel with the calculation of the hash addresses.

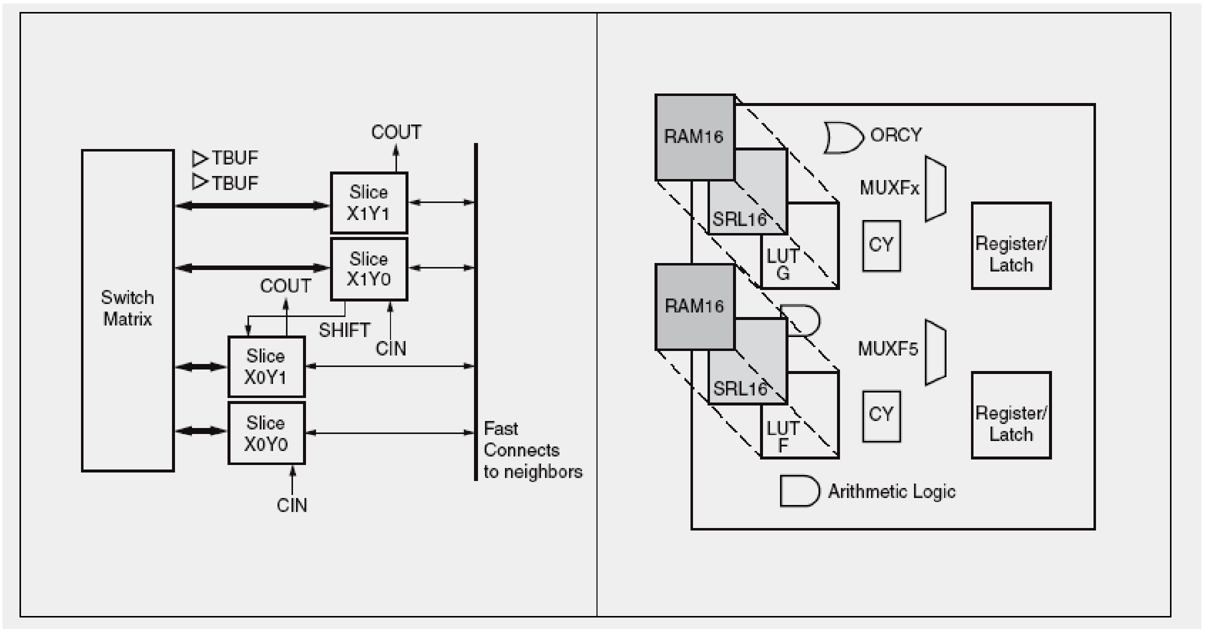


Figure 4.9: Block diagram of a CLB (left) and a Slice (right) [25]

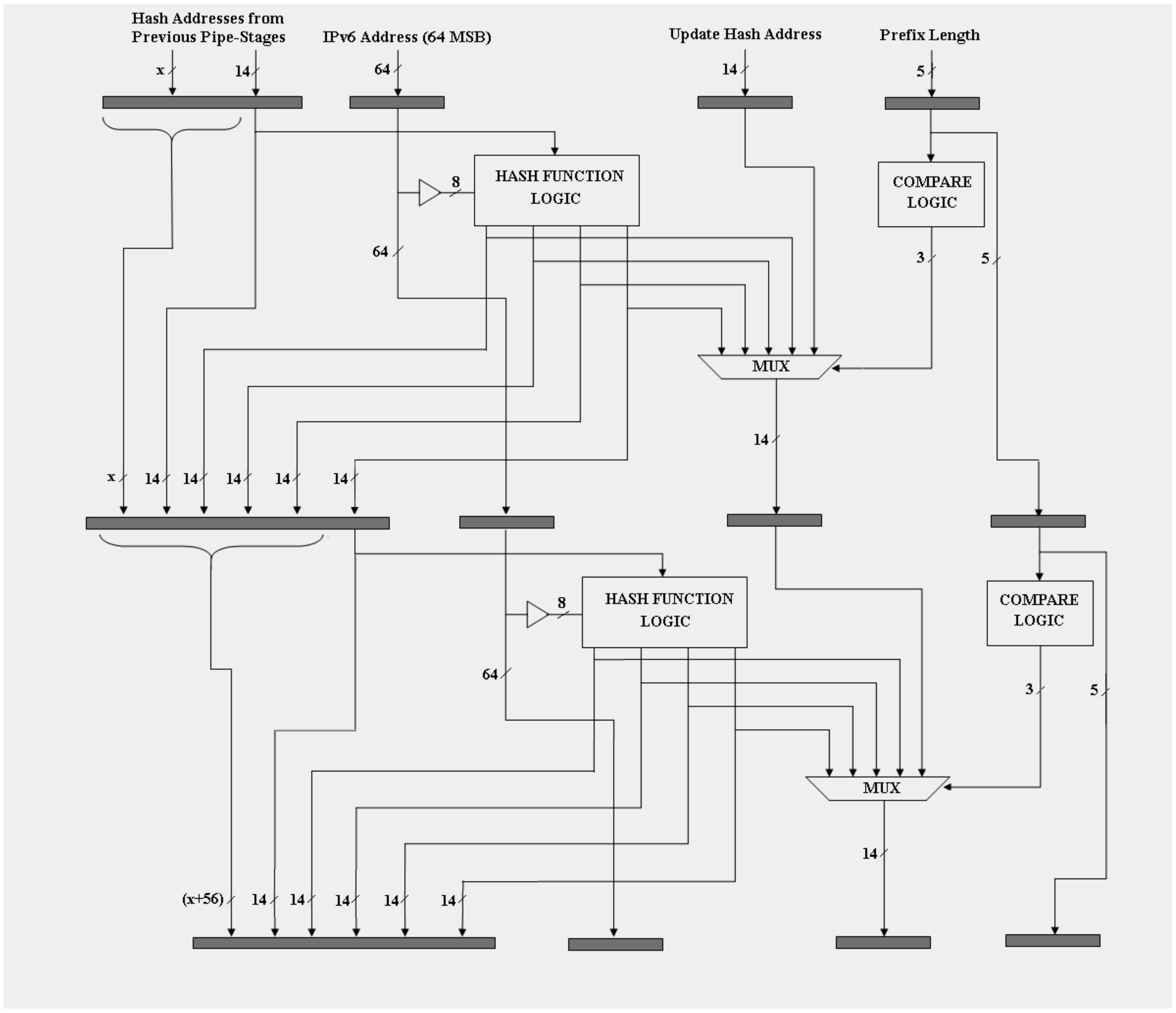


Figure 4.10: Two typical Hash Function’s pipe-stages

Figure 4.10 shows two successive pipe-stages, with integrated UHA selection logic. At this example, each pipe-stage uses 8 bits of the IPv6 address and generates hash addresses for each second bit. That is, if these two successive stages are stages 5 and 6 of ours system then the first one will generate hash addresses for prefix lengths 50, 52, 54 and 56 while the second one for prefix lengths 58, 60, 62 and 64. The Compare Logic Module (CLM) compares the length of each prefix length supported by that particular pipe-stage (e.g. 50, 52, 54, 56), with the Prefix Length field of the input command. If there is a match, the CLM indicates to the multiplexer (MUX) which one of the supported prefix lengths hash address must be selected as the new UHA. If there is no match, the CLM indicates to the multiplexer that it must select the UHA from the previous pipe-stage. Hence, at the end of the final pipe-stage, the UHA register will have the actual hash address of the prefix to be updated.

4.5 Implementing the Counting Bloom Filter Control Unit (CBFCU)

We have seen in Section 3 that the main problem with Bloom filters is that when they use only one bit to represent an entry, we can only query them or insert new data to them but we can not delete data from them. Therefore, we have used the Counting Bloom Filters methodology in order to support all update operations (insertion and deletion). However, the available embedded bits (within the Block RAMs) are not enough to support today's routing tables and, at the same time, provide a counter for each Bloom filter's entry. In order to address this problem we store the counters in an external SRAM or DRAM chip. The type of RAM to be used affects only the time required for an update and does not affect, in any way, the system's look-up throughput.

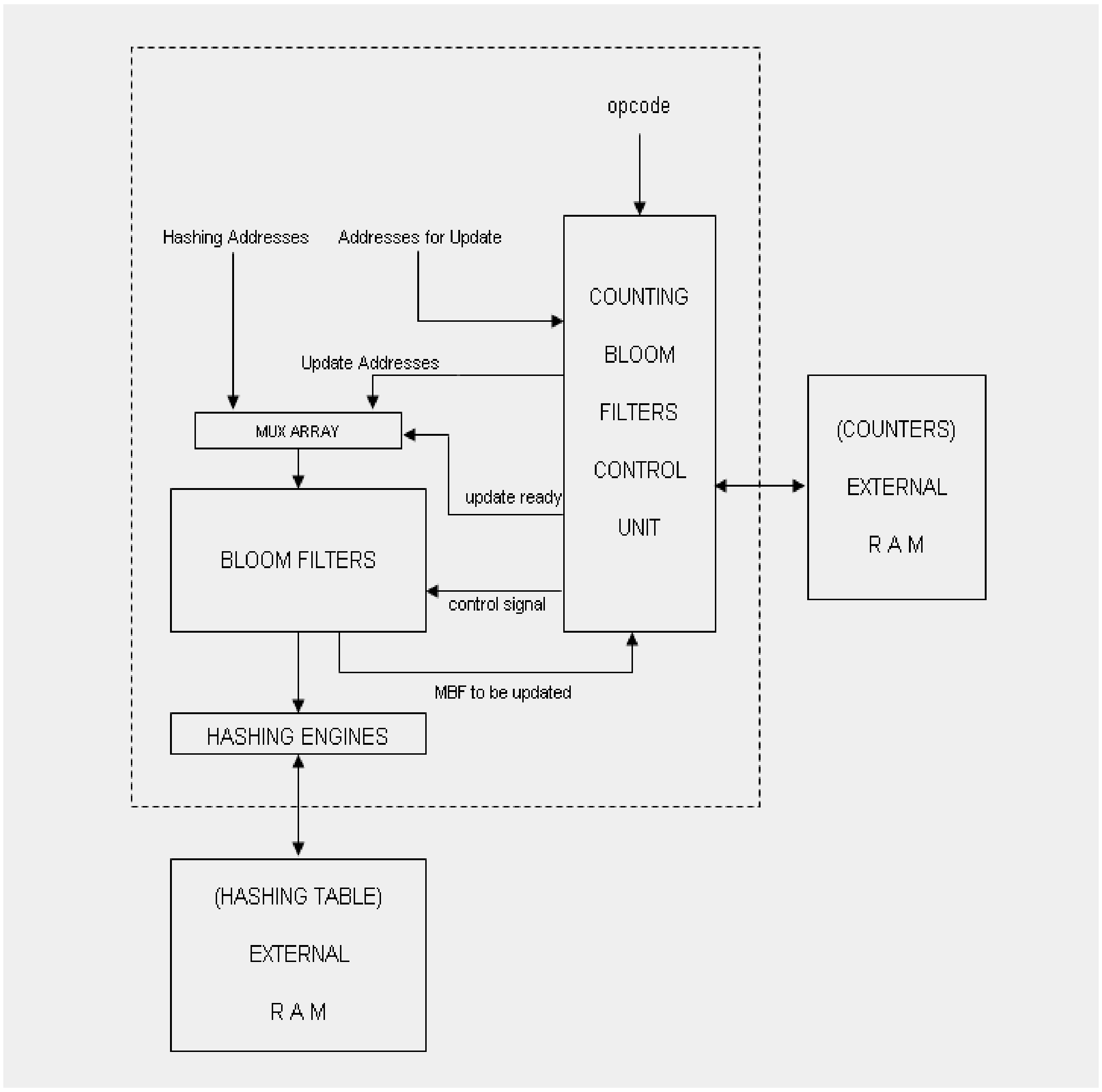


Figure 4.11: CBFCU interaction with the rest of the system and the counters RAM

Processes Performed by the Counting Bloom Filters Control Unit (CBFCU)

In Figure 4.11 we can see the interaction between the CBFCU and the rest of the system. We know that the first six stages of the pipeline are used for calculating the hashing addresses as well as for selecting the addresses of the prefix length to be updated. In addition, we know that the opcode is used as an indicator when we have an update operation. So, when we have an insertion or deletion operation the **addresses for update** (see Figure 4.11) are stored into the CBFCU and the FSM that controls the unit begins the update process. If the update is issued at cc X, the update addresses will be provided to the CBFCU at cc X+6 and the MBF ID, of the MBF that holds the prefix to be deleted or will hold the prefix to be inserted, is calculated during cc X+8 (this is the signal named **MBF to be updated** in Figure 4.1). Thus, the actual update process begins nine cc after the update operation is issued. During the update process, the CBFCU retrieves a counter from the external RAM and based on the type of update (deletion or insertion) increases or decreases the value of the counter and then stores it back to the external RAM. Based on the change of the value of the counter, the CBFCU generates two control bits that intricate the operation to be performed at the Bloom Filter. The first of the two bits is the write enable signal for the Bloom Filter Element (BFE), while the second is the actual value to be written. Therefore, when we have a deletion operation and the counter's values changes from one to zero the bit sequence will be "10" and, likewise, when we have an insertion operation and the counter's value changes from zero to one, the bit sequence will be "11". When no change is needed, the control bits are both set to zero. In addition, due to the fact that the Block RAMs (used to implement the Bloom filters) do not support dual write operation at the same memory location, a mechanism is implemented by the CBFCU in order to prevent this from happening. This is done by allowing only one port of the Block RAM to be enabled at the same time.

The method with the control bits allows as generating a sequence of twenty control bits (called **control signal**) that are to be used by the five BFE (four bits for each BFE) during the update operation. When all ten of the counters are read and modified the CBFCU notifies the Stall Unit (SU) that an update operation is ready by setting the **update ready** bit to one. When this happens, the SU stalls the Hashing Functions Module (HFM) and permits the Bloom filters to be updated by issuing the control signal, to the appropriate MBF, and selecting the **update addresses** (see Figure 4.11) to pass through the MUX array.

From the process described above, it is obvious that when the control processor initiates an update operation it must wait until the update finishes before another can be issued. For the whole update operation 50 cc are needed, thus, allowing as performing six million updates per second when an SRAM operating at 300 MHz is used. If we choose to use DRAM instead of SRAM, the number of update per second will be less than the one mentioned above, but even in this case we asset that the number of updates we be well above the number of updates per second required for a modern router to perform.

Estimating the Number of Bits Required for Storing the Counters

Given that the basic configuration was designed under the constrain that each MBF should not store more than 5678 prefixes, by using a 3 bit counter the probability that we will encounter a counter overflow is extremely low. After simulating a Bloom filter's programming (insertion) operation with 5678 prefixes, 1,000 times, we have never encountered any overflows and in only 4 of these simulations we had a counter with the value of 7. We have averaged all the results of the simulations and plotted the distribution in Figure 4.12.

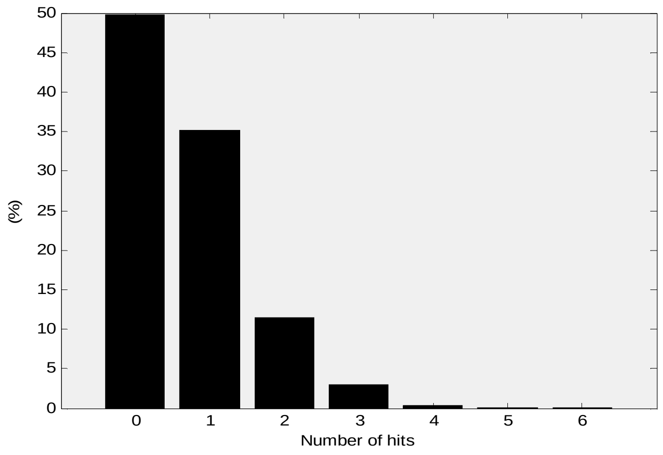


Figure 4.12: Distribution of counter's values in a Bloom Filter programmed with 5678 prefixes

In our designed we have selected the number of bits, for each counter, so as to support the worst case scenario. That is, we allow each Bloom filter to store 5x times more prefixes than the nominal number. That means we support a routing table with 2,839,000 prefixes. We have simulated this scenario 1,000 times and the resulting distribution is illustrated in Figure 4.13.

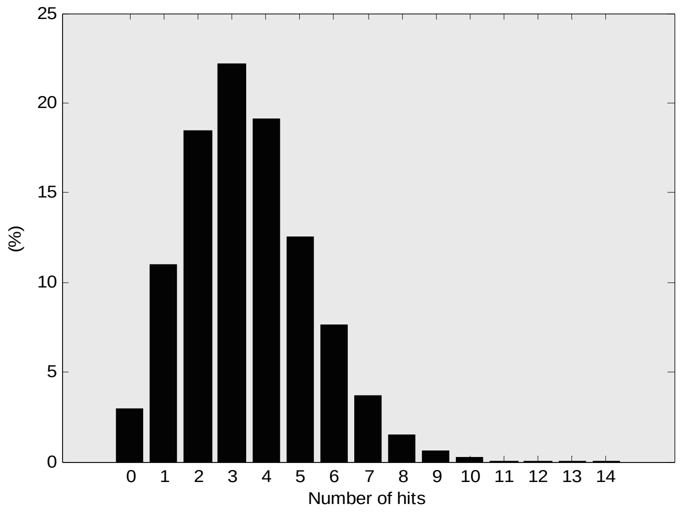


Figure 4.13: Distribution of counter's values in a Bloom Filter programmed with 28,390 prefixes

With the worst case scenario 99% of the simulation resulted in a maximum number of hits between 13 and 14, twenty two times we had a counter with 15 hits and only two times we had an overflow. Therefore, we have selected the number of bits for each counter to be four. However, even though the probability that we have an overflow is extremely low, an overflow is not impossible to happen. The problem with a counter been overflow is the false negatives. That is, a Bloom filter, even though it stores a prefix indicates a “Not Found”. This is something we never want to happen. Therefore, in order to prevent this from happening, an exception handling mechanism must be used. A simple solution is to use a small portion of the distributed bits, of the FPGA, in order to create a small memory with 128 entries, 39 bit each. Figure 4.14 shows the usage of each bit of the exception entry.

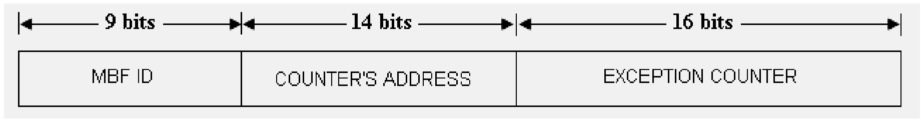


Figure 4.14: The exception entry

Now that we have chosen the number of bits, for each counter, we can calculate the total number of bits required in order to store all the counters in the external RAM. For 100 MBF we need 500 Block RAMs with 2^{14} bits each. Hence, the total number of counters is 8,192,000. When we use 4 bits for each counter we need a total of 32,768,000 bits.

Implementing the CBFCU

Figure 4.15 shows the schematic of the CBFCU. The reregisters shown on the top of the schematic are used in order to store the ten hash addresses of the Bloom filter, for which the update is indented. This is done on the first clock cycle of the update operation. As we have already seen, the MBF identifier (MBF_ID), of the MBF to be updated, is computed three clock cycles after the hash addresses are stored. Therefore, the CBFCU dose not begins the actual update process until the MBF_ID is provided to it. We remind you from subsection 4.3 that the MBF_ID is a 9-bit value. These bits are used in order to give the 9 most significant bits (MSBs) of the actual addresses at which the counters associated with the particular MBF are stored. The 14 less significant bits of the address are taken from the ten hash addresses. Thus, we now have the 23 bits used to address all counters.

We remind you from subsection 4.3 that the Block RAMs, we have used for implementing the Bloom Filters, do not support a dual write operation at the same memory location. Even though this is not possible to occur, we do not have a mechanism inside the Hash Function so as to prevent it from happening. Therefore, we have integrated this functionality inside the CBFCU. We have done this by comparing each pair of two successive hash addresses. That is, the first with second, the third with the forth and so on (all such pair are used in a single Dual Port RAM). If we ever find that such two addresses are equal and that they address a port in which a write operation is to be performed, then only one port is allowed to execute a write.

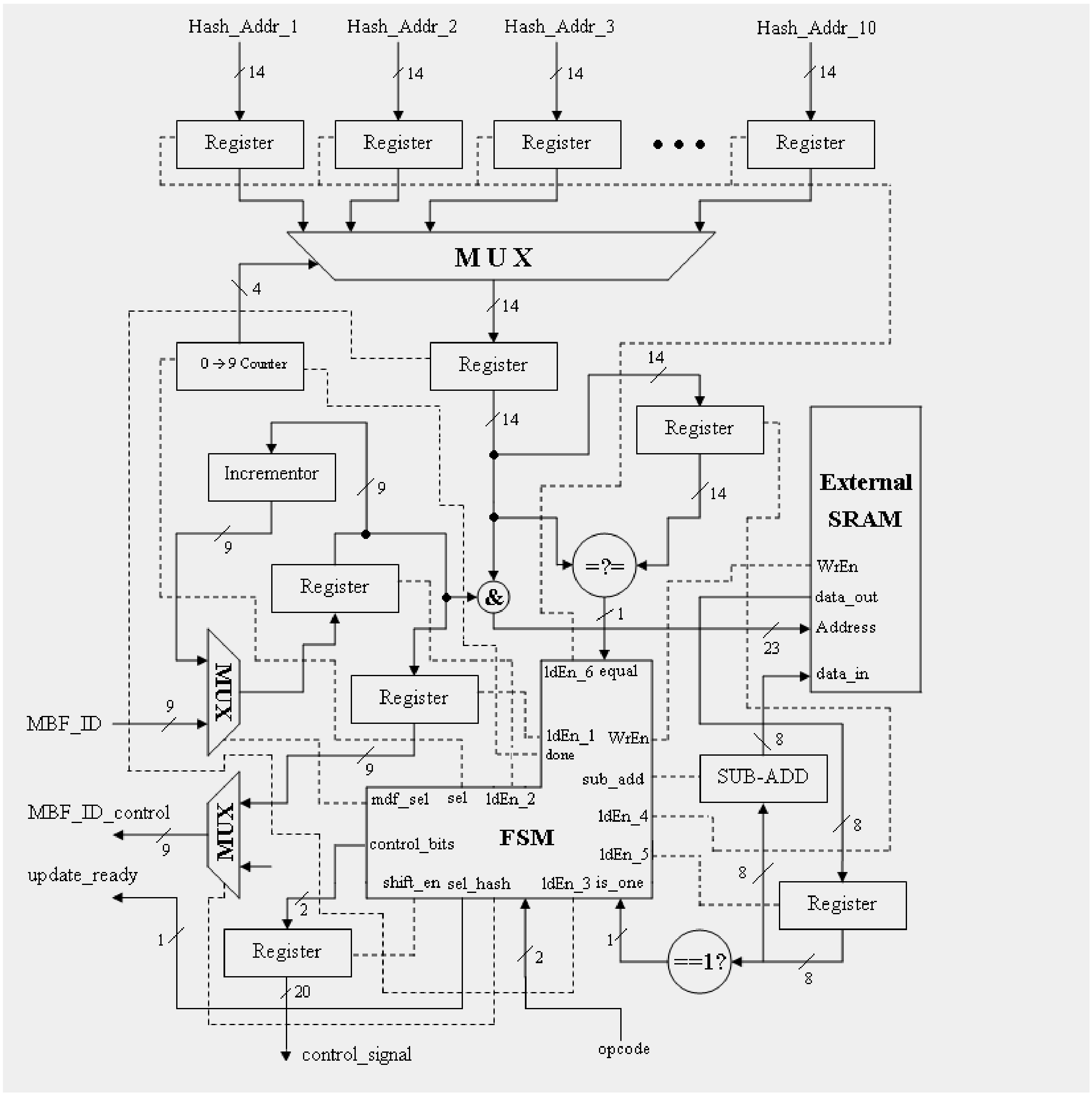


Figure 4.15: The CBFCU

All the processes performed by the CBFCU are controlled by its FSM unit (Figure 4.15). The state transitions diagram for this FSM is shown in Figure 4.16. The transitions diagram can be divided into two parts. At the left hand-side we can see the transitions that take place when we have an insertion operation and which are denoted with prefix *In*. Likewise, at the write hand-side we have the transitions for the deletion operation and which, denoted with prefix *De*. The signals activated at each state are illustrated in Table 4.5.

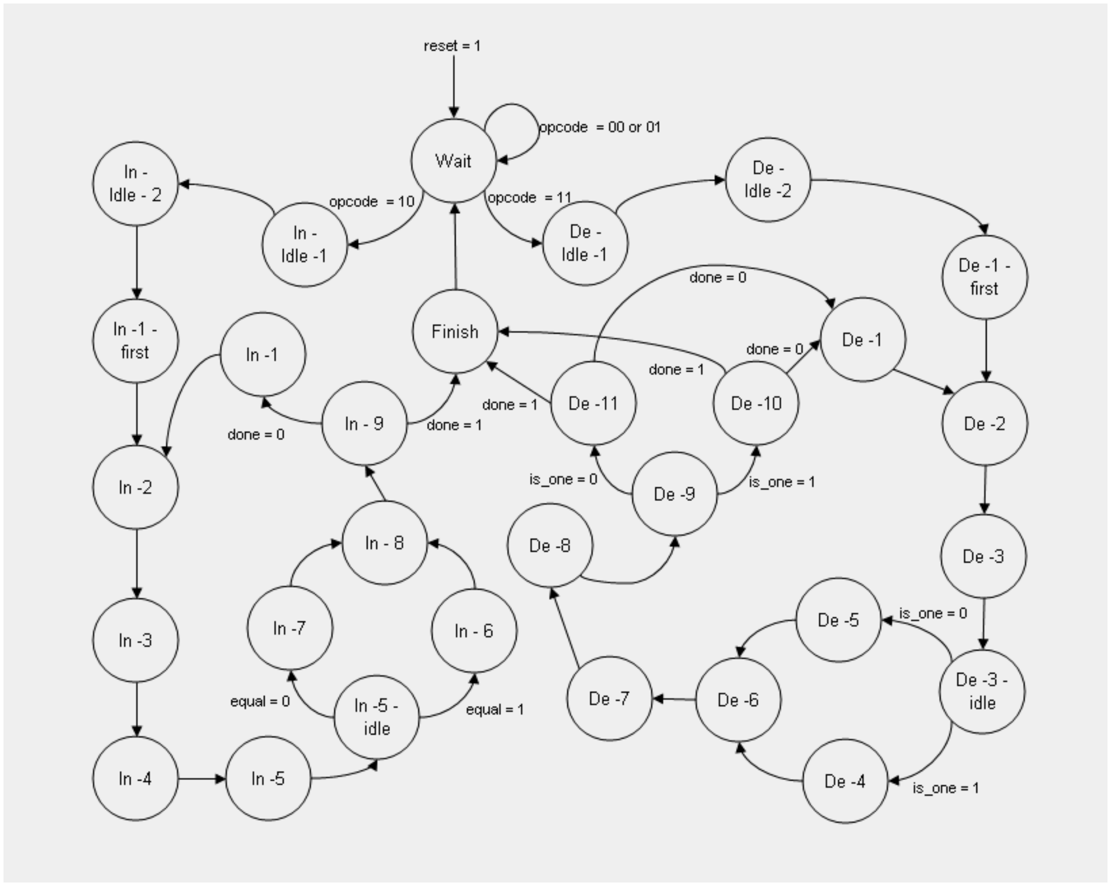


Figure 4.16: FSM Transitions Diagram

FSM State	Signal Activated	FSM State	Signal Activated
Wait	ldEn_6	De - idle - 1	None
In - idle -1	none	Delete	ldEn_2, MBF_ID_sel
Insert	ldEn_2, MBF_ID_sel	De - 1 - first	ldEn_1, ldEn_3
In - 1 - first	ldEn_1, ldEn_3, Shift_en, Control_bits=11, sel	De - 1	ldEn_2, ldEn_3
In - 1	ldEn_2, ldEn_3, Shift_en, Control_bits=11, sel	De - 2	ldEn_4, sel
In - 2	ldEn_4	De - 3	ldEn_5, sub_add
In - 3	ldEn_5	De - 3 - idle	sub_add
In - 4	WrEn	De - 4	Shift_en, Control_bits=01, sub_add, WrEn
In - 5	ldEn_3	De - 5	Shift_en, sub_add, WrEn
In - 5 - idle	none	De - 6	ldEn_3
In - 6	Shift_en, sel	De - 7	ldEn_4, sel
In - 7	Shift_en, sel, Control_bits=11	De - 8	ldEn_5, sub_add
In - 8	ldEn_5	De - 9	sub_add
In - 9	WrEn	De - 10	Shift_en, Control_bits=01,sub_add, WrEn
Finish	update_ready	De - 11	Shift_en, sub_add, WrEn

Table 4.5: Signals Activated at each State (if not listed then the signal is considered to be zero)

4.6 Implementing Hashing

Up to this point we have illustrated how we implemented the Bloom filters, the hash functions used for querying the filters and the CBFCU used for managing the Bloom filter's counters. In this subsection we will show how we have implemented the second part of our look-up scheme, hashing. We use hashing in order to (a) verify that the indicated matches, from the Bloom filters, are not false positives and (b) to find the port at which we must forward the IPv6 datagram.

When it comes to hashing, the main issue that concerns us is collisions. In bibliography, there are many collision resolution schemes. These schemes can be deviated into two categories: Open Addressing and Separate Chaining. Even though Separate Changing bounds the worst case probe sequence length, it is more complicated that Open Addressing especially when it comes to hardware implementation. In Open Addressing, there are three main hashing resolution schemes: linear probing, quadratic probing and double hashing. The performances of quadratic probing and double hashing are almost the same, and both have better performance than linear probing for larger load factors. However, we have selected to impalement our hashing unit using linear probing because of its simplicity and its performance (Figure 4.17) for small load factors. In particular, for load factors between 0.1 and 0.2 linear probing has better performance than both quadratic probing and double hashing.

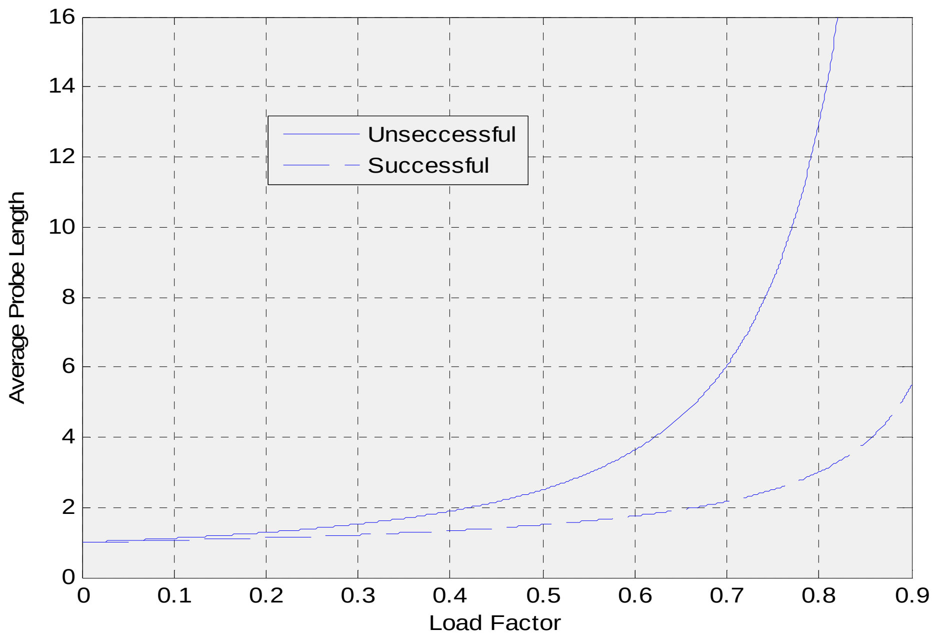


Figure 4.17: Linear probing performance

The average lengths for a successful (P_s) and unsuccessful (P_u) search sequences are given by:

$$P_u(\lambda) = \frac{1+\frac{1}{(1-\lambda)^2}}{2} \quad \text{and} \quad P_s(\lambda) = \frac{1+\frac{1}{1-\lambda}}{2} \quad [26]$$

Implementing Linear Probing

The second choice we have to make, when use hashing, is which hash function to employ. In Figure 4.2 we can see that a bitwise XOR gate used in order to compute a bit vector named *Hash bits*. We have computed those hash bit in order to use them as hash addresses for the external Hash Tables. Therefore, our hash function is (a) the hash functions used for generating the hash addresses for the Bloom filters (b) a bitwise XOR gate producing a hash value to be used for probing the external hash table. In other words, we take advantage of the already computed hash addresses over the key (that is, the prefix) so as to realize the hash function for the main hash table.

For our implementation we have selected using disjoined address spaces for each prefix length. That is, an address of the hash table can accommodate only prefixes of a particular prefix length. For the rest of this thesis, we will refer to these disjoin address spaces as Hash Table Elements (HTE). We have selected using this configuration for two main reasons. First end foremost, in this way we give the flexibility for someone to assign different entry widths for different prefix lengths. For example, an entry storing prefixes of length 20 can have a 20-bit width whereas an entry storing prefixes of length 64 can have a 64-bit width. This is of grate importance when we implement the hash tables using SRAM in which the density is low and the price per bit is high. The second reason we have chosen to use disjoined address spaces is because we can adjust the load factors for a HTE storing a particular prefix. For example, we can make the HTE for prefixes with lengths from 50 to 64 to have a very small load factors (e.g. 0.05) so as to reduce the penalty of a false positive (see subsection 3.3).

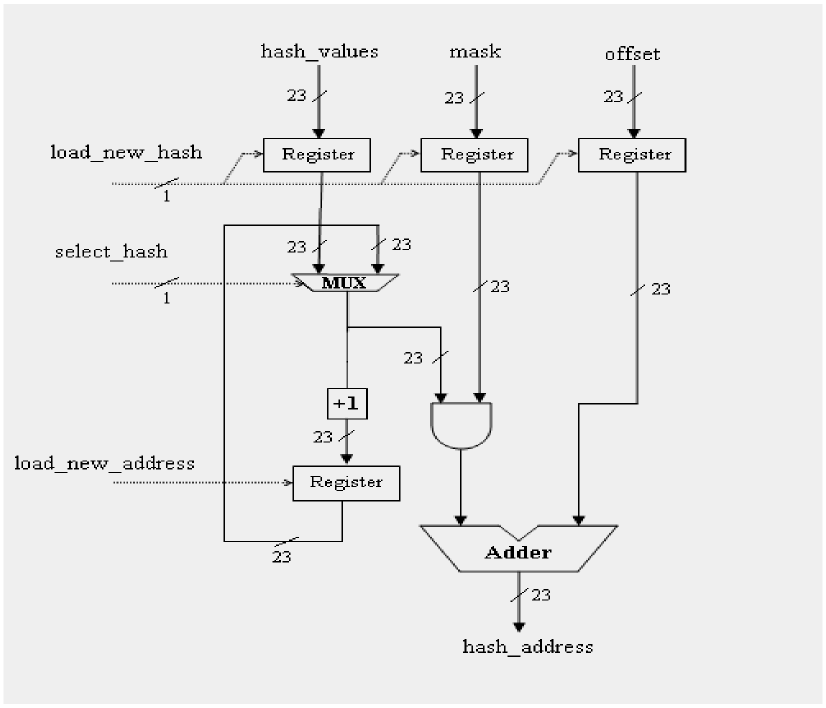


Figure 4.18: Implementing Linear Probing With Disjoined Address Spaces

Figure 4.18 shows the circuit we used for realizing linear probing. The mask and offset are used in order to impalement the disjoined address spaces. The mask confines the size of the HTE and the offset indicates at which memory location the HTE begins. For example, a 7-bit mask for confining the size of a HTE to 16 entries is 0001111.

In order to make possible the usage of linear probing we need to add two more bits in the hash table's entry. An *occupied bit* is maintained to signify that an entry exists at a specific location. A *deleted bit* is maintained to specify that an entry existed at one time on this location, but it has since been deleted. This allows the search to continue in case an entry has been deleted. When a prefix is added, we use the hash bits for generating a hash address as shown in Figure 4.18. This address is then checked for occupancy. If an entry dose not exist at that location (that is, the *occupied bit* is clear) or *deleted bit* has been set, the new prefix is added to the address. If an entry exists at that location (the *occupied bit* is set), the next location will be checked. The next address is calculated by adding one to the address that caused the collision. This is continued until an empty slot is found.

When a prefix is queried, the same process is repeated, stopping when a prefix match occurs or the *occupied bit* and *deleted bit* are found to both be cleared (indicating that the prefix was not in the hash table). Deleting a prefix consists of performing a query, followed by setting the *deleted bit* if the signature was found.

The Hash Engines

The problem with FPGAs is that they are slower relatively to modern ASIC technologies. We have already seen that we employed pipelining in order to overcome this problem. However, after the Bloom filters are accessed we can not use pipelining because of false positives and/or collisions. That is, the steps required for a look-up are not the same for every IP address. In order to address this problem, we assign each look-up or update operations to a device specialized in performing false positive and collision resolution. We will refer to these devices as Hash Engines.

Figure 4.19 shows how a Hash Engine is internally organized. The opcode is used in order to indicate the engines which type of operation is to be performed. When the operation is known, the *Prefix Selection Module* (PSM) is used for inspecting the hit vector so as to decide which prefix length is to be probed next. This is done with the use of a priority encode, as shown in Figure 4.20. For example, if the hit vector is {24, 48, 58} (prefix lengths indicated by the Bloom filters as possible matches), the PSM will first provide the prefix with the highest priority, that is prefix length 58. This prefix length will be used to select which hash bits, mask and offset are to be used by the *Hash Address Generator* (Figure 4.18). If prefix length 58 turns out to be a false positive, the PSM will proved the next longest prefix length, 48, and so on.

We can now track the basic steps followed by a Hash Engine for a look-up when there are no false positives or collisions. First, the PSM selects the prefix length for which we have the match. We will refer to this prefix length as *X*. Next, *X* is used by the MUXs in order to select the corresponding hash bits, mask and offset. After that, the *Hash Address Generator* generates the address which is going to be

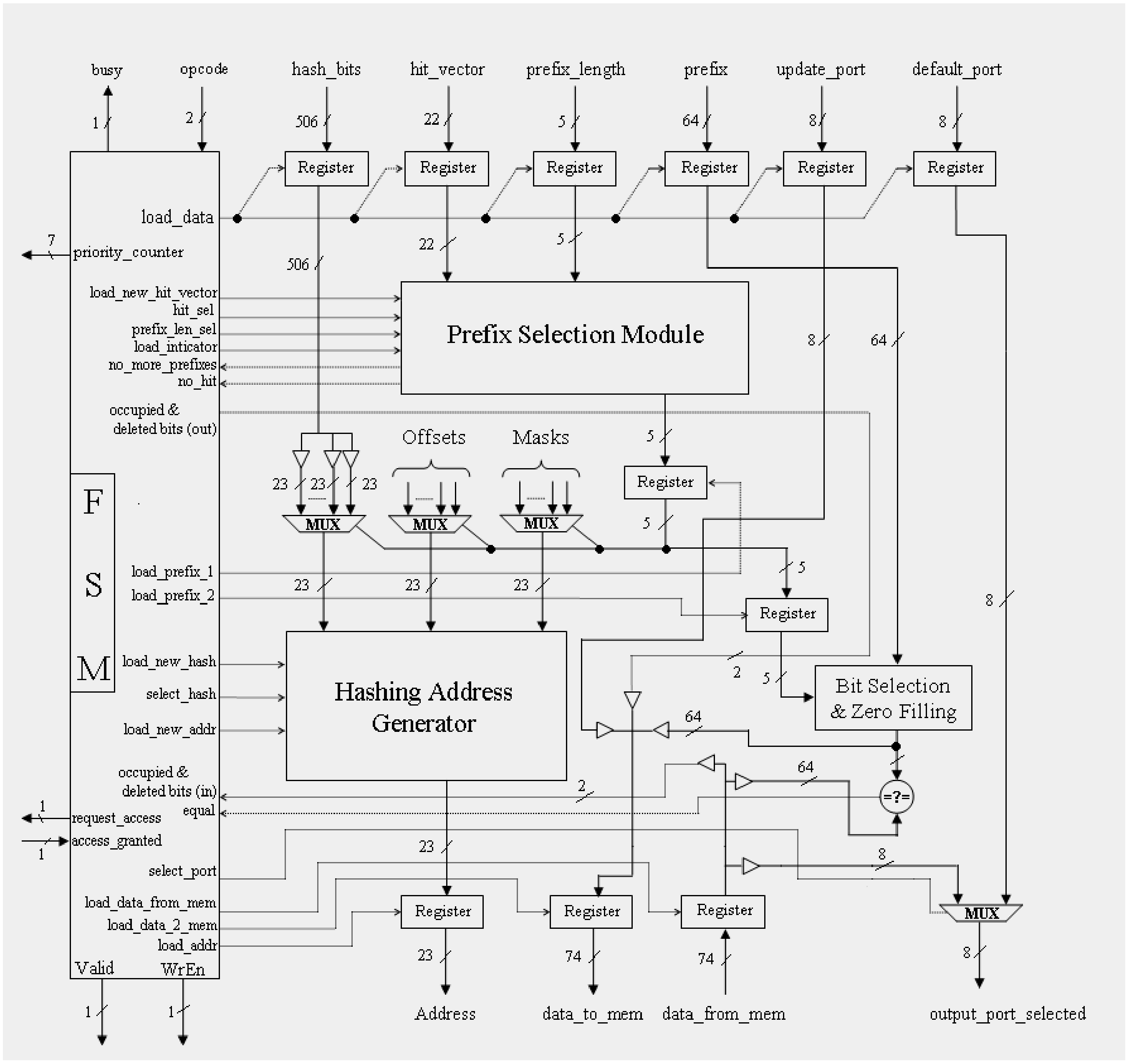


Figure 4.19: The Hash Engine

used for probing the Hash Table. At the same clock cycle, the *Bit Selection & Zero Filling* module modifies the IP address so as to reflect a prefix with length X. For example, if X is 24 the *Bit Selection & Zero Filling* module will clear the last 40 bits of the prefix leaving unchanged only the 24 most significant bits. After the hash address is generated, the Hash Engine generates a request for access to the Memory Controller. When the access is granted, the engine waits one clock cycle for the memory to be accessed. Next, the occupied and deleted bits are inspected, as described above, and at the same time the retrieved prefix is compared with the prefix computed by the *Bit Selection & Zero Filling* module. If there is a match, the valid bit is set and the *output_port_selected* signal now contains the port at which the datagram must be forwarded. We can see that this process takes seven clock cycles to complete and only in one of these the memory is accessed. Therefore, if we want to achieve higher memory utilization we must employ at least seven hash engines.

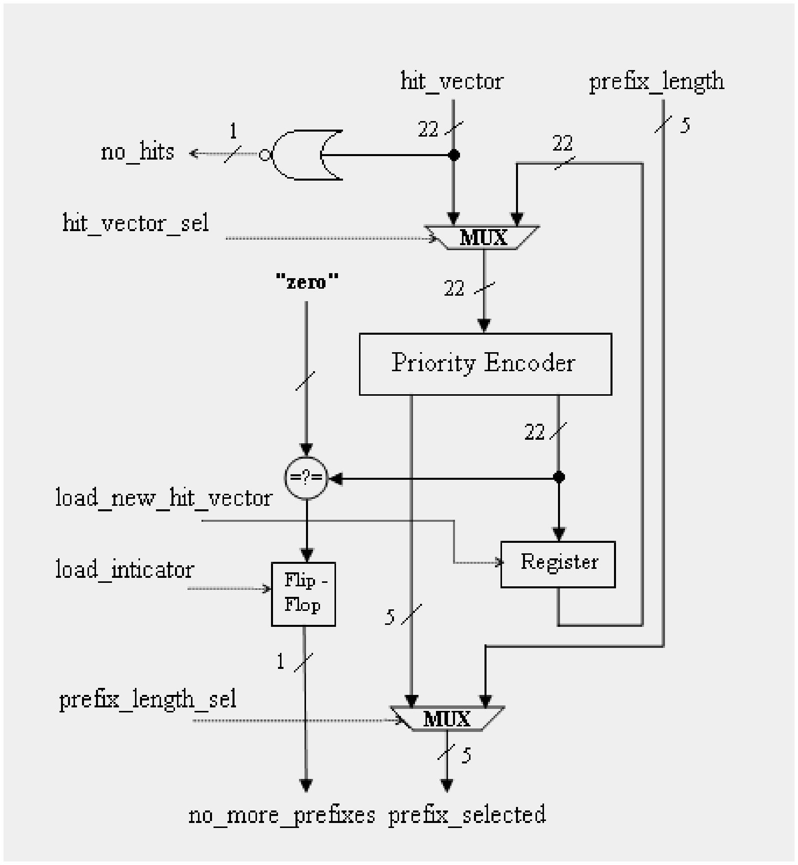


Figure 4.20: The Prefix Selection Module

4.7 Implementing the Schedulers

This system has three schedulers (a) the Hash Engines Scheduler (HES) (b) the Memory Scheduler (MS) and (c) the Report Scheduler (RS). The Hash Engine Scheduler is responsible for assigning look-ups or updates to a particular Hash Engine based on engine's availability. The Memory Scheduler is used in order to control which engine is going to be allowed to access the external Hash Table. And, finally, the Report Scheduler has the responsibility of managing engine reporting.

The Hash Engines Scheduler

After the Bloom filters are accessed, a look-up or update operation must be assign to one of the Hash Engines (Figure 4.2). This functionality is provided by the Hash Engines Scheduler. From Figure 4.19 we can see that each Hash Engine has an output signal named *busy*. This signal is routed to the HES so as to inform it about the status of each Hash Engine. So, when the HES receive an operation, it first inspects the *busy* signals, from all Hash Engines, and then assigns the operation to one of the free engines. If all engines are found to be busy, the HES stall the system until one of the engines becomes available.

The assignment of an operation to an available engine is realized with the use of a priority encoder and a decoder. First all the *busy* signals are routed to the priority encoder which then generates a selection signal. This selection signal is used by the decoder in order to notify a particular engine that an operation has been assigned to it. The decoded has *N* outputs, where *N* is the number of hash engines. Each one of these outputs is the *opcode* signal used by a hash engines. Hence, when the HES wants to assign an operation, to a particular hash engine, it routes the *opcode* signal through the decoder to the *opcode* input of that engine. On the other hand, when an operation is not assigned to a hash engine, the *opcode* routed form the decoder is “00”, hence no actions are to be taken by that engine.

The Memory Scheduler

The Memory Scheduler (MS) has the responsibility of coordinating the accesses to the external hash table. We have already seen that when a Hash Engines wants to access the hash table it generated a request to the MS. Then, the MS based on FCFS (First Came First Serve) priority allows the “oldest” of the operations to access the memory. Thus, the MS must be able to distinguish which one of the engines is serving the oldest operation. This is done using a 7-bit *priority counter* integrated into the hash engines (Figure 4.19). When an engine begins serving a look-up or an update it starts an internal counter. This counter is increasing every clock cycle. When a request for access is issued by multiple engines simultaneously, the MS first serves the hash engine having the largest values in its *priority counter*. In this way, the oldest operation always has the highest priority.

However, with a 7-bit counter we can only measure duration of 127 clock cycle. Hence, when a counter reaches this value we prevent any more assignment of operations to hash engines and, in addition, all the other counters are stopped. This goes on until the engine that had its counter reach the 127 clock cycle limit, completes its operation.

The Report Scheduler

When a Hash Engine completes the execution of a look-up it has to report, to the outside world, at which port the IPv6 datagram must forwarded to. The RS is responsible for managing this kind of reports. The problem with our architecture is that look-ups might complete out-of-order. This is because the number of memory accesses required for a look-up is not the same for every IP. If we allow IP look-ups to complete out-of-order, we have to implement an external circuit that can distinguish for which datagram the reported port corresponds to. However, this process can be simplified if we recall that at a particular clock cycle only *N* look-ups (where *N* is the number of Hash Engines used) can complete and that look-ups are assigned to Hash Engines in-order. Therefore, we can implement *N* distinct datagram buffers each storing the datagram which is served by the corresponding hash engine. In this way, when an engine wants to report its results, the system knows that the datagram, to be forwarded to the indicated port, is the datagram stored in the buffer corresponding to the particular hash engine.

Nevertheless, the Report Scheduler can be programmed so as to report results in order. Thereby reduce the system's complexity. Unfortunately, this comes at a cost of lower throughput. In order for the RS to evaluate which operation is "older" the same *priority counter* used for accessing the external memory can be used.

5. Simulations and Resource Consumption

So far, we have studied the architecture of this look-up processor and all the proposed solutions that aim to maximize the external SRAM's utilization. What we ideally want is a system with the ability to perform one probe, to the external hash table, every clock cycle. If we achieve this, then in the case where there are no false positives or collisions this will result to a look-up processor with the ability to perform one IP look-up per memory access. Using an SRAM running at 300 MHz this look-up scheme can perform 300M look-ups per second. This is an important enhancement compared to previously propose algorithmic solutions, which require at least four to six memory accesses for a single lookup operation, [27] and to TCAM based solutions that have a constant performance of 100M look-ups per second [1]. However, the problem is that whereas, in the simple configuration we have examined in Section 3 (Longest Prefix Matching using Bloom filters) the extraction of performance estimation was straightforward, this is not the case for the architecture we have designed in the previous sections. This is due to the different false positive probability for each LBF as well as the different load factors for each hashing table element. Moreover, all these come to combine with the parallel access to the external SRAM (by the hash engines) making it even more difficult for someone to intuitively evaluate the performance of the implemented look-up scheme. Therefore, this section primarily aims at illustrating that this look-up processor has the ability to reach a throughput close to one IP look-up per memory access and, secondly, to prove that this can be achieved at a reasonable cost by utilizing less than 20% of the logic recourses of a modern FPGA.

5.1 Simulations

For all simulations we use the prefix distribution generated by Wang et al. [3] (Figure 5.1) and we followed the same methodology for generating IP traces as in [1], [2], [28]. The problem is that while prefix databases in backbone routers are publicly available, this is not the case for traffic traces. Indeed, traffic statistics depend on the location of the router. Therefore, in order to address this problem, we consider that every prefix has the same probability of being accessed. In other words, the traffic per prefix is supposed to be the same for all prefixes. Although knowledge of the access probabilities, of the forwarding table entries, would allow better evaluation of the average look-up time, assuming constant traffic per prefix still allows us to measure important characteristics, such as the worst-case look-up time. Even thought this methodology is used in most other previous LPM studies, in the second configuration (of our simulation model) we change the distributions of incoming prefixes and examine how this can affect the overall performance of the system. Based on the distribution of Figure 5.1 the actual prefixes stored in the forwarding table, after control prefix expansion (CPE), are shown in Table 4.2. In the same table we show the number of MBF used by each LBF, the number of entries for each hashing table element and their corresponding load factors. By choosing to use distinct prefixes at the lengths shown in Table 4.2 the forwarding table's size increases about 10.34%. Therefore, by using a routing table with 400,000 unique prefixes the actual forwarding table size (that is, the number of elements stored in the Bloom filters) is 441,376 entries. We here underline that all simulations are made without any alternation to the number of MBFs and to the number of hash table entries.

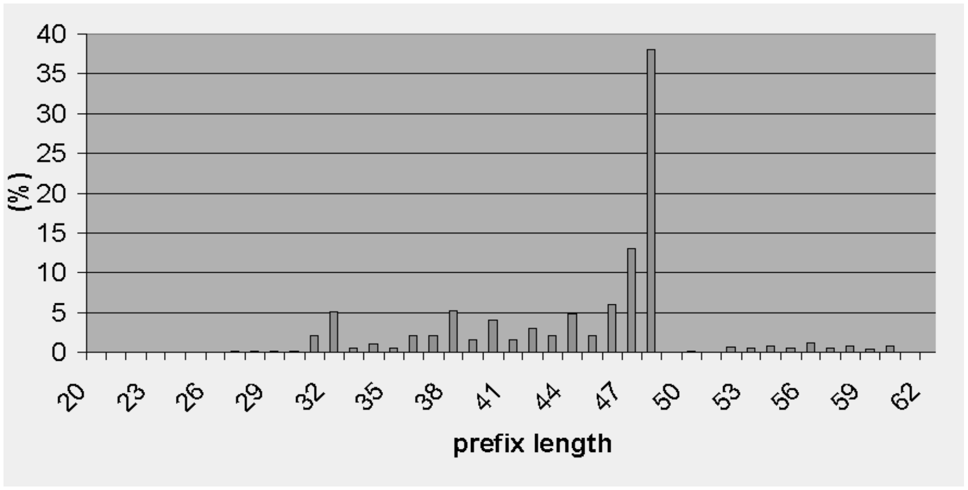


Figure 5.1: The default prefix length distribution used in the simulations

Configuration 1: Calculating average memory accesses per look-up operation

In this configuration, we are only interested on the relation between the numbers of prefixes stored in the forwarding table and the corresponding average number of memory accesses required for a look-up. The simulation results are illustrated in Table 5.1. Note, that the throughput (given in Gbps) is calculated considering a 400-bits minimum size package. This is reasonable for IPv6 datagram given that this protocol uses a minimum header size of 40-bytes (40x8=320 bits).

In Figure 5.2 we plot the throughput with respect to the routing table size. We can see that the throughput is decreasing exponentially with respect to the number of prefixes stored in the routing table. This is something that we expect to happen, firstly, because of the exponential relation between the false positive probability and the number of prefixes stored in the Bloom filter (Figure 3.2) and, secondly, due to the exponential relation between the load factor and the average hash probe length (Figure 4.17). We remind you from Section 1, that for studies within the near future, it is reasonable to use a table size for IPv6 that lies between 1x and 2x of today's IPv4 table size [3]. If, for example, we use the largest IPv4 routing table existing today (208,905: Telstra, July 2005) the number of entries a IPv6 look-up processor should support is between 200,000 and 500,000. Within this range we can observe that (a) the throughput is well over 110 Gbps corresponding to less than 1.07 average memory accesses per look-up operation and (b) that we achieve 270 million look-ups per second. We here note that, all throughput measurements are taken under the assumption that we have 100% utilization of an SRAM running at 300 MHz.

Number of entries after expansion	Number of memory accesses	# Accesses / Lookup	False Positives / Lookup	Max. False Positives	Max. Mem. Accesses	Average Throughput (Gbps)	Worst Throughput (Gbps)
110.344	111.593	1,011319147	0	0	4	118,656905	30
165.516	168.344	1,017085961	0	0	4	117,984128	30
220.688	225.754	1,022955485	1,35939E-05	1	5	117,307157	24
275.860	284.013	1,029554847	3,26252E-05	1	5	116,555228	24
331.032	343.069	1,036362044	7,25005E-05	1	6	115,789652	20
386.204	402.765	1,042881482	0,000339199	1	7	115,065808	17,1428571
441.376	463.533	1,05019983	0,001067117	1	8	114,263968	15
496.548	525.493	1,058292451	0,002243489	2	8	113,390207	15
551.720	589.626	1,06870514	0,004971725	2	10	112,285415	12
662.064	726.491	1,097312344	0,016383612	3	11	109,358106	10,9090909
772.408	885.810	1,14681619	0,042667865	3	12	104,637518	10
882.752	1.086.196	1,230465635	0,092502764	5	16	97,5240564	7,5
993.096	1.353.469	1,362878312	0,173582413	5	16	88,0489468	7,5
1.103.440	1.724.233	1,562597876	0,292794352	6	23	76,7951895	5,2173913
1.158.612	1.963.735	1,694903039	0,368060231	7	24	70,8005103	5
1.213.784	2.239.463	1,845025968	0,453666385	7	24	65,039735	5
1.268.956	2.569.460	2,024861382	0,549462708	7	38	59,263316	3,15789474

Table 5.1: Memory Accesses Simulation

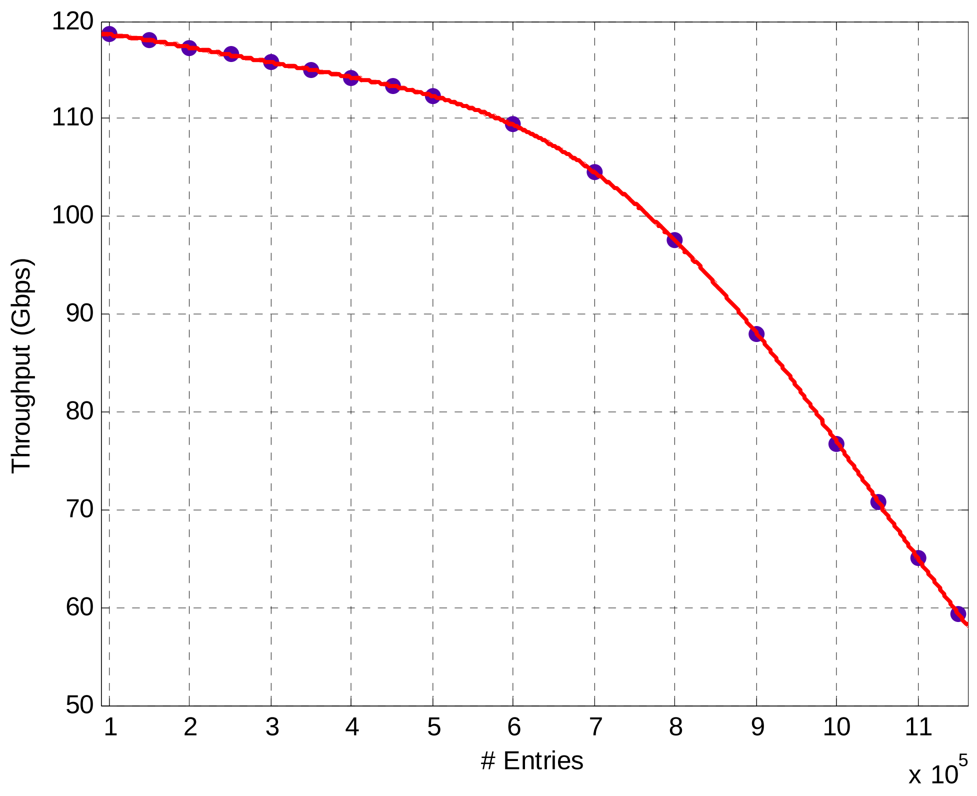


Figure 5.2: Throughput Vs number Routing Table Entries

In Table 5.1, the false positives per look-up can be used as the overall false probability of our system. As we can observe, this probability is below 0.5% for all simulations with forwarding table's size within the nominal range of 200,000 to 500,000 entries. Another important parameter is the maximum number of false positives. We here not that this number is not the maximum number of false positives observed during a look-up, but are actually the maximum number of false positives that forced the system to access the external memory. For example, if the hit vector is {20, 32, 48, 50, 56, 58, 64} and the longest prefix matching the IP destination address is of length 48, then the maximum number of false positives is four and not six. This parameter is useful because it gives as an indication about the maximum number of hash probes required, for a look-up, when we assume that there are no collisions. For our range of interest, the maximum number of false positives is less than three, showing that the worst-case look-up hash probes, even though theoretically are not bounded, are practically very small. The difference between the maximum number of false positives and maximum number of memory access is due to collisions. Therefore, if we increase the number of hash tables entries we expect to have as many memory access as the number of false positives. Finally, the worst throughput is the throughput we expect our system to have when all look-ups require the maximum number of memory accesses for completion. We can see that even for our range of interest, the worst throughput is relatively very low. However, from Table 1.2, we can see that even if our system constantly operates at its worst throughput, is still suitable for all optical carrier (OC) lines except for OC-768 (Table 1.2).

In order to give an insight view of how many memory accesses are required per look-up, we illustrate their distribution, for two routing tables with 400,000 and 500,000 entries respectively, in Figure 5.3. For the 400,000 entries table, more than 95% of the look-ups required only one memory access, 4% required two accesses and only 0.0002% required eight accesses to complete. As for the second table, 94% required one memory access, 5.3% two and just 0.0003% required eight accesses.

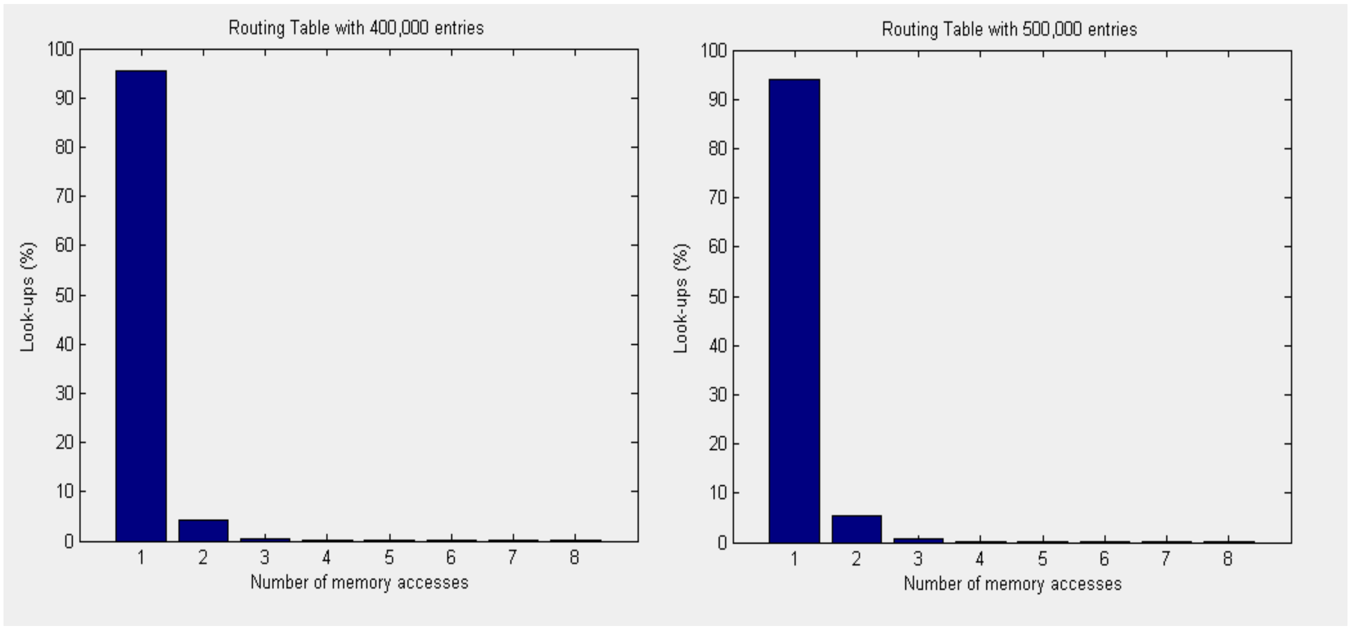


Figure 5.3: Distribution of memory access for completion

Configuration 2: Variation of the false positive probability with respect to the distribution of incoming IP addresses

In Section 3 we have proved that the overall false positive probability depends on the distribution of matching longest prefix. In this simulation, we try to find out at what degree the variation of incoming IP addresses distribution, can affect the overall system performance. The results of this simulation are presented in Table 5.2. For each measurement, we randomly generated five routing table and for each table we generated ten different incoming IP sequences; one for each distribution. The measurements of Table 5.2 are the average of these simulations. Graphical representations of all the distributions we have used are shown in Figure 5.4.

Routing Table Size (after expansion)	Incoming Queries Distribution	Number of False Positives	Mem access / lookup	Max Number of Memory Access
441376	Initial	440	1,0502	6,8
441376	Step HL	852	1,0542	7,9
441376	Step LH	284	1,0453	6,5
441376	Pyramid	529	1,0513	7,5
441376	Uniform	562	1,0516	7.3
551.720	Initial	2654	1,0688	9,1
551.720	Step HL	5053	1,0775	9,5
551.720	Step LH	1805	1,0628	8,8
551.720	Pyramid	3160	1,0702	9,1
551.720	Uniform	3402	1,0699	9,1
662.064	Initial	10840	1,0973	10
662.064	Step HL	20493	1,1202	10,2
662.064	Step LH	7379	1,0892	9,6
662.064	Pyramid	12801	1,1013	9,4
662.064	Uniform	13902	1,1058	9,5

Table 5.2: Incoming distribution variation

We here note that while in Table 5.1 we illustrate the maximum number of false positives and memory accesses, observed in all of our simulation, in Table 5.2 these parameters are average values. From Table 5.2 we can see that even though the change in the number of false positives is affecting the average memory access per look-up, the maximum number of memory accesses remains almost constant. Therefore, we can assert that this look-up scheme will present similar performance characteristics even if the probability of an IP address matching a particular prefix is not the same for all the prefixes of the routing table. However, if the distribution is known prior to the configuration of our look-up processor, we can adjust the false positive probabilities as well as the load factors so as to achieve optimal performance; for that particular distribution. This flexibility is another advantage of our scheme compared to TCAM-based as well as others algorithmic solutions.

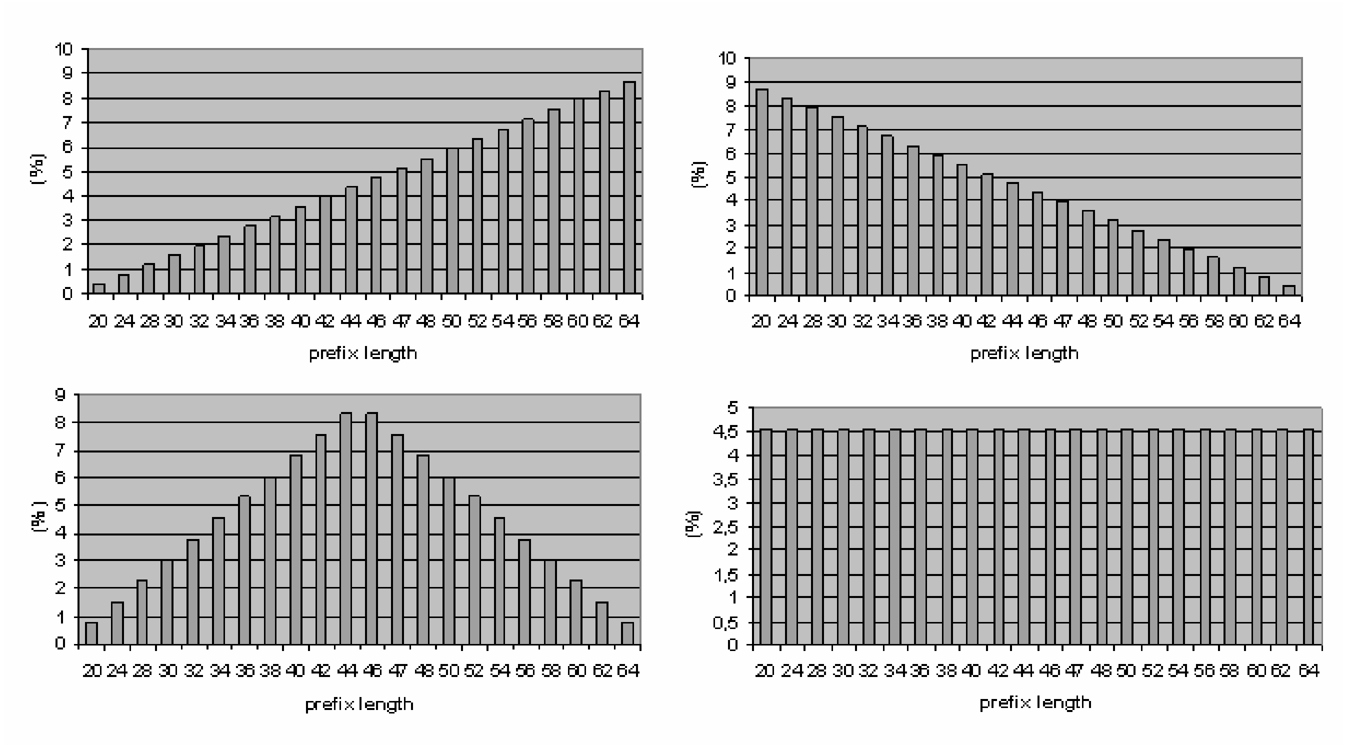


Figure 5.4: Step LH -lower to higher- Distribution (Upper left), Step HL -higher to lower- Distribution (Upper right), Pyramid Distribution (Lower left) and Uniform Distribution (Lower right)

Configuration 3: Changing the number of Hash Engines

Given that we have one look-up issued per clock cycle, when there are no false positives and collision is obvious that with seven Hash Engines we can achieve 100% external SRAM. Unfortunately, due to collisions and/or false positives things tend to become more complicated. Even though we know that only one hash engine can produce an answer during a single clock cycle (an engine completes three clock cycles after its last memory access and only one engine can access the memory at each cycle) we can not expect the engines to provide results in-order. We have shown in Section 4 how we can achieve synchronization between the engines so as to give results in-order. With this simulation we are trying to find out with how many hash engines we can achieve, for the in-order scheme, performance close to the one realized by the out-of-order completion scheme. The simulation results are shown in Table 5.3.

As we can seen, in Table 5.3, if we want to achieve performance close to the out-of-order configuration we have to use sixteen or more hash engines. However, with the use of more than sixteen hash engines the throughput is not improving significantly while with ten the performance is very close to the out-of-order configuration. Therefore, we assert that with the use of ten hash engines we can achieve the required throughput. The main problem when using more hash engines is that the number of comparators required is increasing. For example, with seven engines we require twenty one comparators (combinations of two for seven items), with eight engines we require twenty nine and so on. In addition, the bits per counter must also increase. This can be shown in Table 5.3 where the required clock cycles for completion are increasing with respect to the number of hash engines used.

Number of unique prefixes:	400.000		**Number of Queries:**	441.376
Number of Hash Engines used	**Total Clock Cycles**	**CPL (Clock Cycles per Lookup)**	**Max delay (cc)**	**Lookups with Max delay**
1	3157054	7,152754114	37	1
2	1592110	3,607151272	37	1
3	1072229	2,429287048	37	2
4	813494	1,843086167	37	2
5	662966	1,502043609	37	3
6	563851	1,277484503	38	2
7	495217	1,121984431	38	4
8	479896	1,08727253	38	5
9	467262	1,058648409	38	6
10	465360	1,054339158	39	7
16	463558	1,050256471	45	13
32	463550	1,050238346	62	16
64	463550	1,050238346	100	3

Number of unique prefixes:	500.000		**Number of Queries**:	551.720
Number of Hash Engines used	**Total Clock Cycles**	**CPL (Clock Cycles per Lookup)**	**Max delay (cc)**	**Lookups with Max delay**
1	3981253	7,216075183	37	2
2	2015749	3,653572464	37	3
3	1363676	2,471681288	37	3
4	1039097	1,883377438	37	5
5	851310	1,543010948	38	1
6	728026	1,319557022	38	2
7	642687	1,164878924	38	3
8	618953	1,121860726	39	1
9	599802	1,087149279	39	2
10	594757	1,078005148	40	5
16	589687	1,068815704	46	18
32	589642	1,068734141	67	1
64	589642	1,068734141	104	3

Table 5.3: Changing the number of Hash Engines for the in-order look-up completion

In order to get a better view of what is happening to the clock cycles for completion, we illustrate its distribution for various numbers of hash engines in Figure 5.4. All the distributions are the results of simulating a routing table with 600.000 unique prefixes (662.064 prefixes after prefix expansion). From Figure 5.5 we can observe that even thought with more hash engines we achieve better utilization of the external SRAM, the average delay as well as the maximum delay (that is, the maximum number of clock cycles when a look-up operation begins until the look-up processor finish processing it) are both increasing. Hence, the processing delay, in the router, for each IPv6 datagram will increase resulting to larger end-to-end transfer delay. This is of grate importance for end-to-end time sensitive applications like Voice over IP (VoIP) . This is the main disadvantage of the in-order scheme.

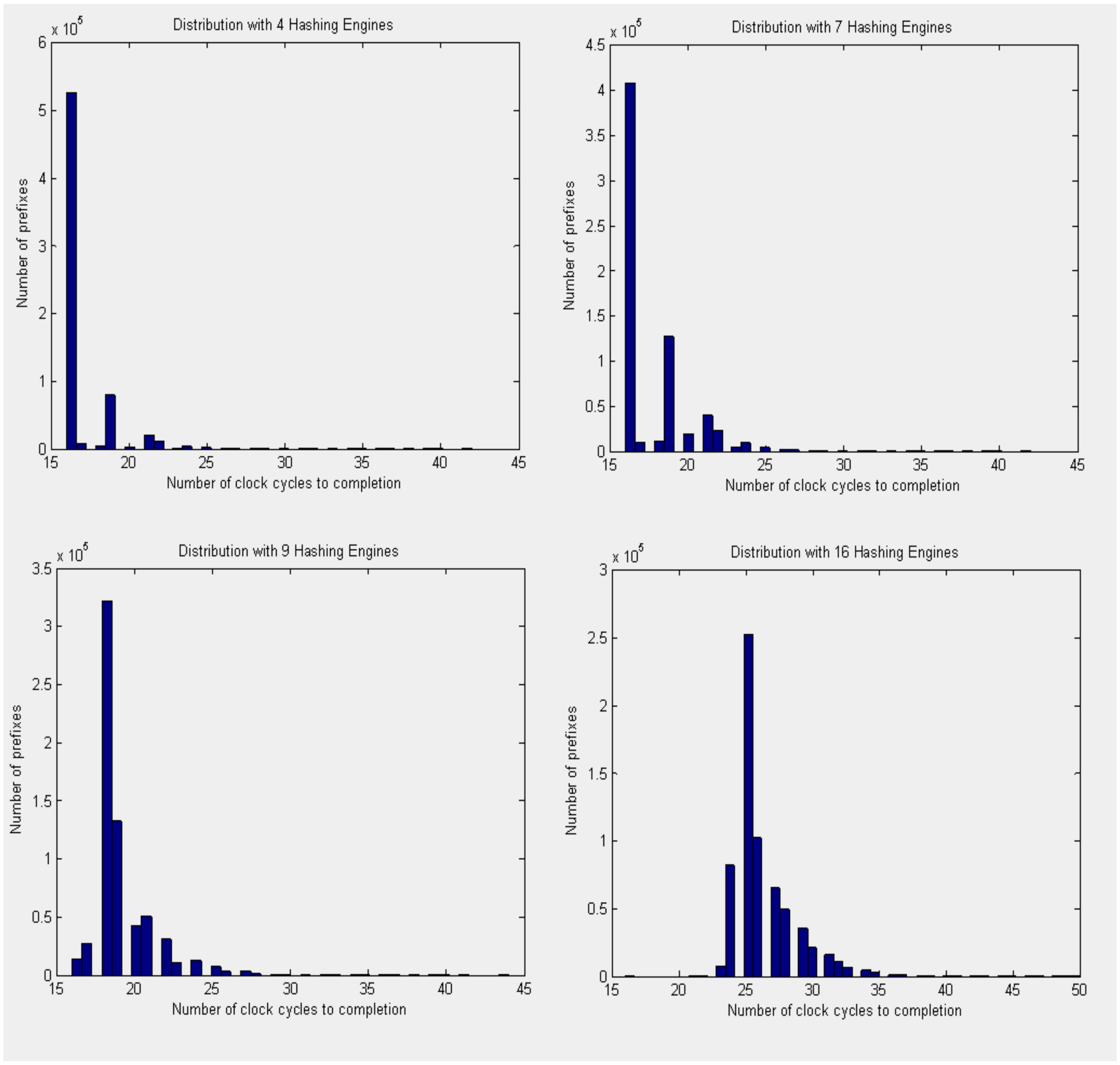


Figure 5.5: Distributions of required clock cycles to completion

Configuration 4: Expected stall durations for the in-order and out-of-order configurations

In Section 3 we have seen that when all the engines are found to be busy the system is stalled and hence no more look-ups can be issued to the look-up processor. When we have a stall, all the received datagrams are stored into the router’s input buffer. When the number of accumulated bit exceeds the capacity of the buffer datagrams will be dropped. Even thought this is tolerable for the best effort service offered by the Internet Protocol, it is not something that we want to happen very often. Therefore, we assert that the number of stalls and especially the duration of a stall is an evaluation metric that must be taken under consideration when designing a look-up processor.

In Table 5.4 we show the total number of stalls, the average stall duration and max stall duration for the in-order and out-of-order configuration. For each configuration we used seven, nine and sixteen hash engines and simulated them using a routing table with 400,000 and 600,000 entries.

Number of Queries	400.000		**Number of unique prefixes**	441.376				
Order of completion	**Number of Hashing Engines used**	**Total Clock Cycles**	**CPL (Clock Cycles per Lookup)**	**Max delay (cc)**	**Lookups with Max delay**	**Max stall duration (cc)**	**Avg stall duration (cc)**	**Number of Stalls**
In-order	7	495217	1,12198443	38	4	19	3,02	17829
	9	467262	1,05864841	38	6	19	1,48	17473
	16	463558	1,05025647	45	13	19	1,51	14740
out-of-order	7	463550	1,05023835	38	1	6	1,07	20659
	9	463550	1,05023835	40	1	6	1,07	20657
	16	463550	1,05023835	47	1	6	1,07	20649

Number of Queries	600.000		**Number of unique prefixes**	662.064				
Order of completion	**Number of Hashing Engines used**	**Total Clock Cycles**	**CPL (Clock Cycles per Lookup)**	**Max delay (cc)**	**Lookups with Max delay**	**Max stall duration (cc)**	**Avg stall duration (cc)**	**Number of Stalls**
In-order	7	816509	1,23327805	42	9	26	3,52	43879
	9	751842	1,1356032	44	9	26	2,15	41733
	16	726675	1,09759026	50	26	25	2,19	29377
out-of-order	7	726506	1,097335	42	2	6	1,13	57087
	9	726506	1,097335	44	2	6	1,13	57087
	16	726506	1,097335	54	2	6	1,13	57080

Table 5.4: Stall duration simulations

As we can see, the average as well as the maximum stall durations are much higher for the in-order configuration. This is reasonable because with the in-order scheme an engine remains occupied even if the operation which it serves has completed. The reason we have less stalls in the in-order configuration is because during a stall interval a large number of hash engines will be ready to report results. Therefore, when the look-up that caused the stall is completed, during the next clock cycles a lot of hash engines will report their results, thereby reducing the overall number stalls.

Concluding, it is obvious that the input buffer will overflow much more frequently with the in-order scheme compared to the out-of-order. Therefore, we assert that the use of extra hardware so as to support out-of-order completion is tolerable due to the enhancement they will provide to our system.

5.2 Resource Consumption

Table 5.5 summarizes the resources consumed by each major subsystem of our look-up scheme. As we can see, the most recourse intensive subsystem is the Hash Functions Module. This is something that we have expected it to happen due to the large amount of logical gates required for implementing the H_3 Hash Functions (Figure 4.7).

Subsystem: Hash Functions Module		
Resource	**Amount**	**(%) of Device Resources**
Slices	6462	10.23
Slice Flip Flops	9649	7.64
4-input LUTs	6027	4.77
		Max. Frequency : 307 MHz
Subsystem: Counting Bloom Filter Control Unit		
Resource	**Amount**	**(%) of Device Resources**
Slices	268	0.43
Slice Flip Flops	394	0.31
4-input LUTs	205	0.16
		Max. Frequency : 348 MHz
Subsystem: Bloom Filters with Control Logic		
Resource	**Amount**	**(%) of Device Resources**
Slices	1438	2.28
Slice Flip Flops	265	0.21
4-input LUTs	2528	2
RAMB16s	522	94
		Max. Frequency : 301 MHz

Subsystem: Hash Engines		
Resource	**Amount**	**(%) of Device Resources**
Slices	1,526	2.45
Slice Flip Flops	2,114	1.68
4-input LUTs	1,617	1.33
		Max. Frequency : 303 MHz

Table 5.5: A summary of the resources consumed by each subsystem

The final resources consumed by the complete IP look-up processor are shown in Table 5.6. The total amounts of resources of the FPGA we have used are summarized in Table 5.7. As we can see, our implementation consumed only 16.4 % of the logic resources of a modern FPGA.

Resource	**Amount**	**(%) of Device Resources**
Slices	10,356	16.4
Slice Flip Flops	15,623	12.37
4-input LUTs	13,256	10.5
RAMB16s	526	95.29
Max. Frequency : 301 MHz		

Table 5.6: Resources consumed by the complete IP look-up processor

Xilinx Virtex-4 Series FPGA Device: vfx140ff1517-12	
	Amount
CLB Resources	
CLB Array (Row x Column)	192 x 84
Slices	63,168
Logic Cells	142,128
CLB Flip Flops	126,336
Memory Resources	
Max. Distributed RAM Bits	1,010,688
Block RAM1 w/ECC (18 kbits each)	552
Total Block RAM (kbits)	9,936

Table 5.7: Xilinx Virtex-4 Series FPGA VFX140 Device resource summary [24]

6. Summary and Future Work

6.1 Future Work

There are several directions for future work that we can pursue. First, we can use *unbalance MBFs* that require fewer Block RAMs for storing prefix lengths with a small number of prefixes. Thereby, preserve some Block RAMs which could be exploited in order to create MBFs with larger capacities or with smaller false positive probabilities. For example, the MBF that stores prefixes with length 20 could be implemented using only one Block RAM and we could then use the other four BFE (Bloom Filter Element) to reinforce Bloom Filters which store more prefixes. If we increase the number of BFEs from five to six we decrease four-fold the false positive probability. However, this will require the use additional hash function.

We can achieve a second enhancement by exploiting the available distributed RAM in order to create a number of Dual Port RAM which could then be used for improving the false positive probabilities. Of course, these RAMs will be much smaller than the Block RAMs because of their relatively higher access time. However, this is a very good enhancement considering that more than 87% of the device flip flops are unexploited.

In order to reduce the number of false positives, we could use a cache that temporarily stores the most resent prefixes which caused a false positive. This *False Positive Cache* could be implemented using either the available distributed memory or the unexploited Block RAMs. The cache's associativity must be full due to lack of special locality from false positives. We here note that, when we use as a tag the IP address (that caused the false positive) we will have a problem when updating a prefix. For example, if the cache entry is a <IP address, longest matching prefix length> pair, then in the case where we delete the prefix (which used to be the longest prefix matching that address) then this will result to a false negative. A solution is to have a cache for each Bloom Filter. In this case, the cache's entry will be a <prefix, valid> pair. If during a cache query we have a hit, the indication from the corresponding Bloom filter will be ignored because we know that the match is a false positive. When we keep a different cache for each Bloom filter, we can reduce the number of memory bits required. That is because, the tag for the cache storing prefix with length 20 will be smaller than the tag of the cache storing prefixes of length 64. Moreover, with this solution, we can use caches with more entries for the more dense filters while keep fewer entries for the sparser ones.

Besides using the available resources of the FPGA, we could employ some external circuits for improving this system. For example, we could reduce the overall false positive probability by using a TCAM to store prefix lengths with few prefixes. We have seen that prefixes with lengths ≥ 49 consist of only the 6% of the forwarding table. So, we could use a TCAM for storing these prefixes. This will result to less distinct Bloom filters and hence less false positives. Moreover, the Block RAMs used for those filters could be distributed to the remained Bloom filter and thus further reducing the overall false positive probability.

A TCAM could also be used in order to create a cache for storing prefixes that are causing a large number of collisions when probe the external hash table. If we only store prefixes which cause more than i.e. 3 collisions, we assert that this cache will have a relatively small number of entries. In addition, we can force or block some prefixes from being added, to the cache, based on their type of flow. For example, we can choose to hold, in the cache, prefixes that correspond to a Voice over IP (VoIP) flow and not to keep prefixes that correspond to a flow transferring e.g. FTP datagrams. Thereby further reducing the number of required entries.

6.2 Summary

We have shown an implementation of a Longest Prefix Matching (LPM) algorithm for IPv6 that employs Bloom filters to efficiently narrow the scope of the search. In order to optimize average performance (reduce the number of distinct Bloom Filters), we have used of a direct look-up array and Controlled Prefix Expansion (CPE). Performance analysis and simulations show that average performance approaches one hash probe per look-up with modest embedded memory resources, less than 18 bits per prefix. Moreover, we have shown that when our system implemented in current FPGA semiconductor technology it consumes less than 17% of its logic resources. . Finally, we have proposed a variety of enhancements that could further improve the performance of our system and bound its worst case.

In addition, we support that if our system is coupled with a commodity SRAM device operating at 300 MHz, it could achieve average performance of over 250 million lookups per second. In comparison, state-of-the-art TCAM-based solutions for LPM provide 100 million lookups per second, consume 150 times more power per bit of storage than SRAM, and cost approximately 30 times as much per bit of storage than SRAM. While the cost of TCAMs may decrease and power-saving features may emerge, we assert that SRAM technologies will always provide faster access times and lower power consumption due to the massive parallelism inherent in TCAM architectures. As a result, algorithms such as this that employ commodity RAM devices and achieve comparable or better performance will continue to provide an attractive alternative to TCAM-based solutions.

References

[1] Sarang Dharmapurikar, Praveen Krishnamurthy, David E. Taylor, *"Longest Prefix Matching using Bloom Filters"*. ACM SIGCOMM, August 2003.

[2] Miguel A. Ruiz-Sanchez, Ernest W. Biersack, Walid Dabbus, *"Survey and Taxonomy of IP Address Look-up Algorithms"*. IEEE Network Vol.15 No.2, April 2001.

[3] Mei Wang, Stephen Deering, Tony Hain, Larry Dunn, *"Non-Random Generation for IPv6 Tables"* High Performance Interconnects. Proceedings. 12th Annual IEEE Symposium, August 2004.

[4] Thomas Narten, *"IPv6 Address Allocation and Assignment Policy"*. APNIC, June 2006.

[5] *"IP Version 6 Addressing Architecture"*. Internet RFC 2373, July 1998.

[6] Rama Sangireddy, Arun K. Somani, *"High-Speed IP Routing with Binary Decision Diagrams Based Hardware Address Look-up Engine"*. IEEE Journal on Selected Areas in Communications Vol. 21 No. 4, May 2003.

[7] BGP Reports. http://bgp.potaroo.net/, July 2005.

[8] C. Labovitz, *"Scalability of the Internet Backbone Routing Infrastructure"*. Ph.D. thesis, University of Michigan, 1999.

[9] Huan Liu, *"Routing Table Compaction in Ternary CAM"*. IEEE MICRO Vol. 22. No.1, January 2002.

[10] Micron Technology Inc. http//www.micron.com, July 2005.

[11] B. H. Bloom, *"Space/Time Trade-offs in Hash Coding with Allowable Errors"*. Communications of the ACM Vol. 13 No. 7, July 1970.

[12] P. Gupta, S. Lin, and N. McKeown, *"Routing Look-ups in Hardware at Memory Access Speeds"*. IEEE INFOCOM, April 1998.

[13] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. *"Small Forwarding Tables for Fast Routing Look-ups"*. ACM SIGCOMM, September 1997.

[14] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, *"Scalable High Speed IP Routing Look-ups"*. ACM SIGCOMM , September 1997.

[15] David E. Taylor, *"Survey & Taxonomy of Packet Classification Techniques"*. Technical Report, Washington University in Saint Louis, May 2004.

[16] Devavrat Shah and Pankaj Gupta, *"Fast Updating Algorithms for TCAMs"*. IEEE MICRO January 2001.

[17] R. K. Montoye, *"Apparatus for Storing 'Don't Care' in a Content Addressable Memory Cell."* United States Patent 5,319,590. HaL Computer Systems, Inc. June 1994.

[18] SiberCore Technologies Inc. SiberCAM Ultra-18M SCT1842. Product Brief, January 2002.

[19] Micron Technology Inc. Harmony TCAM 1Mb and 2Mb. Datasheet, January 2004.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, *"Summary cache: A scalable wide-area web cache sharing protocol"*. IEEE/ACM Transactions on Networking Vol. 8 No. 3, June 2000.

[21] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, *"Efficient Hardware Hashing Functions for High Performance Computers"*. IEEE Transactions on Computers Vol. 46 No. 12, December 1997.

[22] Sarang Dharmapurikar, Michael Attig and John Lockwood, *"Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters"*. Technical Report, Washington University in Saint Louis, March 2004.

[23] Reuse Methodology Manual, *"Xilinx Design Reuse Methodology for ASIC and FPGA Designers"*, Xilinx, May 2004.

[24] Datasheets Xilinx Virtex-4 Series FPGAs. http://www.xilinx.com/virtex4. Winter 2005.

[25] Virtex-4 Product Specifications. http://www.xilinx.com/virtex4. Winter 2005.

[26] Donald E. Knuth, *"The Art of Computer Programming"*. Volum 3: Sorting and Searching, Addison Wesley, 1997.

[27] P. Gupta, *“Algorithms for Routing Lookups and Packet Classification”*. Doctoral dissertation. Dept. Computer Science, Stanford University, 2000 .

[28] Mikael Sundstrom and Lars-Ake Larzon, *“High-Performance Longest Prefix Matching supporting High-Speed Incremental Updates and Guaranteed Compression”*, IEEE INFOCOM, March 2005.