FAST IP ADDRESS LOOKUP ENGINE FOR SOC INTEGRATION

Tomas Henriksson
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping
tomhe@isy.liu.se

Ingrid Verbauwhede
UCLA EE Dept
7440B Boelter Hall
Los Angeles CA 90075
ingrid@ee.ucla.edu

Abstract. *IP packet forwarding is a key operation in internet routers and the limiting factor is the IP address lookup. Since almost a whole router is integrated on a chip it is important to design the functional blocks with SoC integration in mind. Here a hardware architecture for IP address lookup based on the k-multibit trie search algorithm is presented. The designed and simulated architecture is based on one SRAM access in series with two multiplexers and is claimed to be optimally fast. Based on the performance of embedded SRAMs an approximate throughput of 500 MPackets/s is achieved when implemented in 0.18* µ*m technology. Memory requirements, multiplexing, scaling, and updating issues of the architecture as well as 2-dimensional search are described.*

1 Introduction

To make the continued growth of internet applications possible the routers must keep improving in performance much faster than Moore's law predicts. Due to space, power consumption and performance requirements it is desired to integrate as much as possible of a router on one single chip. One way of doing that is to design the router as several heterogeneos domain specific processors. An example of such a processor can be seen in [1].

The most critical component of today's internet routers is the IP address lookup, which is needed for forwarding packets to the right output port of the router.

During the last few years some algorithms have been presented for IP address lookup. A good survey is provided in [2]. However, most algorithms are for software implementations and therefore inherently not optimal from a performance point of view and not suitable when a domain specific processor can be used. New software algorithms, that make use of the access pattern to improve the average performance have been suggested more recently [3] and [4], but they suffer from similar limitations. Another approach is to use hash functions [5], but the memory requirement seems to be too high to allow an efficient SoC hardware implementation.

Some approaches have been introduced for hardware schemes [8], [9], [10], and [11]. These are more interesting, when looking for a fast solution and we discuss them more thoroughly in section 7.

In this paper we present a novel, regular, and scalable hardware architecture for performing IP address lookup optimally fast. We start with IPv4 addresses, for which we provide an extensive performance analysis of our architecture. Then, we enhance the architecture with three multiplexing schemes, which add significant flexibility to the architecture and thus we achieve a domain specific processor for IP address lookup, which is suitable for SoC router integration. We also extend the analysis to 128 bit long IPv6 addresses and forwarding based both on destination and source addresses.

The ability to update a forwarding table is essential to an implementation and we describe two different approaches, which both support instantaneous updates for our architecture.

In section 2 the general problems with longest prefix matching are briefly explained, then a discussion on embedded memories, which are essential and performance limiting components for IP address lookup, is provided in section 3. Thereafter, in section 4, our novel approach is outlined and in section 5 our simulation results are presented. Updating issues and further extensions are discussed in section 6. Our results are compared to related work in section 7 and finally conclusions are drawn and future work is outlined in section 8.

2 IP Address Lookup with Longest Prefix Match

The IP address lookup with longest prefix match problem has been described in many papers (e.g. [2]). The forwarding of packets is based on matching the destination address of the packet to prefixes in a routing table. Each prefix consist of a tuple <routing prefix, prefix length>. The number of valid bits in the prefix is determined by the prefix length. In the Internet of today the classless interdomain routing (CIDR) is used, which means routing prefixes of any length can be used. Still, however, many routing prefixes are of length 16 or 24, because of the original internet class-based routing scheme.

The task is conceptually simple, the routing prefix with the longest length, that match the destination address is the best match. To each prefix an action pointer is associated. The action pointer associated with the best match points to the action that should be taken for the packet. For simplicity we can think of this action pointer as the output port identifier, but more information is normally provided, such as next hop IP address.

Although the task is simple to understand, it has proven hard to implement in a good way, especially with ever growing routing tables [7]. There are examples of more than 100k rules in some core routers. To avoid a complicated search algorithm, direct memory lookup can be applied. One way is to let the destination address be the address to the memory and access the result directly. Already with IP addresses of 32 bits 4 billion entries would be needed and for IPv6 addresses of 128 bits the approach is totally infeasible. Instead a combination of search algorithm and memory lookup can be used.

3 Memories for IP Address Lookup

It can be assumed that the memory and the processing element will be implemented on the same silicon die. Although it may be possible to use embedded DRAM with a somewhat more complicated manufacturing process, it is not desirable since the speed-up techniques used for cache-based memory hierarchies cannot be used in the IP address lookup algorithms due to the non sequential access scheme. Therefore embedded SRAM with true random access is better suited. On the other hand, the problem with SRAM is the limited storage capacity and bigger silicon die area requirement. As will be shown later, the performance of the lookup is highly dependent on the memory access time. Also it must be noticed, that the memory access time is dependent on the size of the SRAM, both on the number of words and the word length [12].

So it is crucial to keep the size of the memory as small as possible and definitely below the limit for embedded SRAMs which is some hundred kilobytes. If the memory can be split into several, preferably equally sized, independent components the constraints on the memory are considerably relaxed. Although, the total size must still remain less than some hundred kilobytes.

4 Direct Implementation of k-multibit Tries

The previous observations have led to the design of an architecture based on the k-multibit trie search algorithm. In the original configuration the architecture consists of W/k stages, each with a processing element and an associated memory, see Fig. 1. W denotes the number

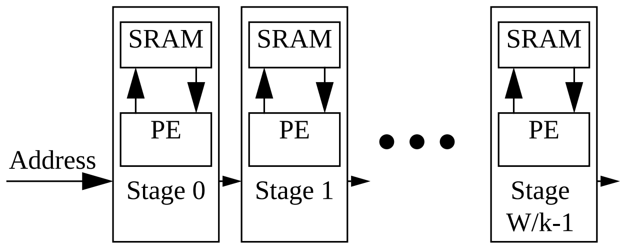


Figure 1: General architecture

of bits in the address and k the number of bits processed at each stage. In the first stage the k most significant bits (msbs) will be used to address the memory, thus the first memory will always need 2k entries. The output of a stage is a tuple <is_pointer, result/pointer>, where the is_pointer = 0 means that the second field is the result, and the is_pointer = 1 means that the second field is a pointer to the next stage. The internal structure of a stage can be seen in Fig. 2. Each stage has a pipeline register in order to make full utilization of the processing

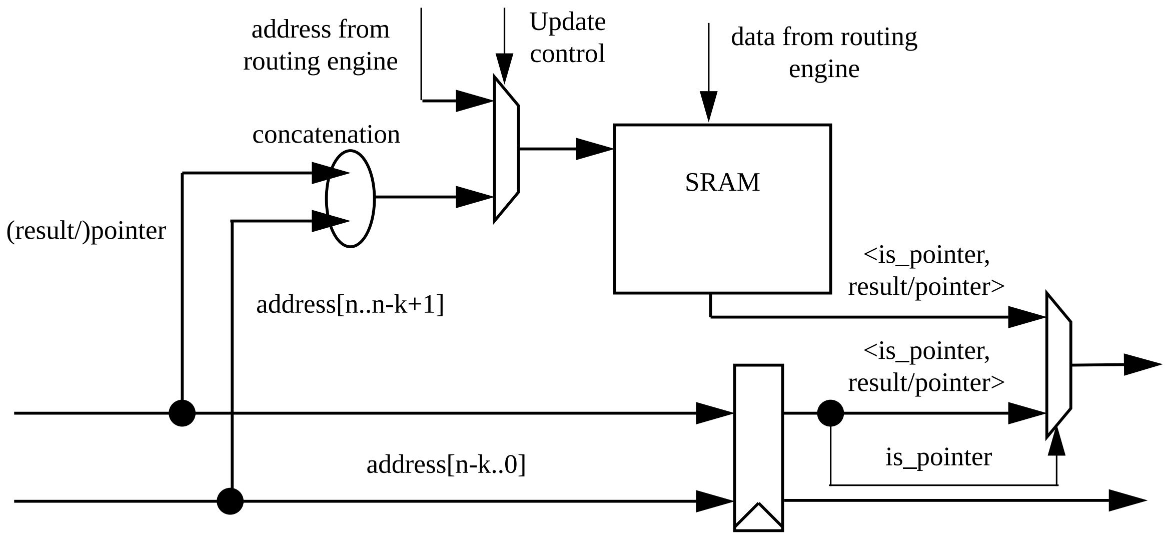


Figure 2: The architecture of one pipeline stage

elements and the memories possible. The output from the memory is also synchronous, so the pipelining takes also place inside the memory. To have a deterministic behavior, a result is always forwarded to the last stage. This is done by letting the msb of the output of the previous stage (the is_pointer bit) control the multiplexer to the result/pointer output. If the previous output was a result, it is forwarded, otherwise it was a pointer and the <is_pointer, result/pointer> output from the memory is chosen to be forwarded to the following stage. To the first stage the only input will be the address. After the last stage the output will always be a result. The address that is passed on to the next stage does not contain the k msbs, since they will not be used by later stages. After the last stage there is no remaining part of the address left.

Since the architecture is pipelined, one search can be started every clock cycle and also one will be finished every clock cycle at full utilization. The delay from initiated search request until the result is produced is W/k clock cycles. The length of a clock cycle is as mentioned earlier mostly dependent of the memory access time. To be precise it is the memory access time + 2 multiplexer delays. We claim that this is optimal, in the sense that we will always need at least a memory access to be able to perform a search operation within a big number of prefixes. If we also want to avoid the full lookup of 2W entries in the memory, we need some evaluation mechanism. There is no faster evaluation mechanism than a 2-input multiplexer. Updating of the routing table is necessary, therefore we also need a multiplexer to support that. Hence, because we only use fundamentally required operations, we have reached optimality.

5 Results

5.1 Memory requirements

To find out how big the memories need to be, the available routing tables from IPMA of May 23rd 2001 [7] where used. Five different routing tables are available, in table 1 the results for the smallest table, mae-east, with 16416 prefixes are presented. The unit of the values are number of entries. An entry is a tuple <is_pointer, result/pointer>. As expected the memory requirements increase when k increases, since the memory is used in chunks of 2k. It can also be seen that when k=2x the memory requirements are relatively smaller than otherwise. This is because the bits in the address exactly matches the number of stages. For all other values of k the last stage will have less than k bits to process. This increases the amount of required memory. The most promising values of k are 2 and 4, since they give good trade-offs between number of stages and memory requirements. Similar analysis were made for the other tables, but are left out due to space constraints.

Unfortunately the number of entries at the different stages are varying with several orders of magnitude. It will be the stage with the biggest memory requirement that will decide the minimum clock cycle period. The size of the biggest memory, emax, will also decide how many bits, m, are necessary for the result/pointer field in each entry, assuming that the result can be expressed with less bits than the pointer. The memory address consists of m+k bits and thus 2m+k >= emax. For example, if k=4, emax=298320 for mae-west, and m>=15. Setting m=15 would allow for a emax of up to 215+4=524288 and therefore m=15 could handle routing tables with more entries than mae-west. Exactly how big routing tables can be handled is dependent of the exact distribution of prefixes in the routing table.

Having m=15 is certainly reasonable, since each entry in the memory would consist of 16 bits, 1 bit for the is_pointer field and 15 bits for the result/pointer field. With 215+4 entries, the biggest memory requires 1 mega byte (MB). It should also be noticed, that the memory with the most entries normally does not have to have that many bits for the pointer since the next stage has a lot fewer entries. However, for the sake of regularity it is reasonable to make each stage identical. According to [12], a 2 MB 4-transistor SRAM can operate at 500 MHz. In [13] a 4K word (16 bit) 6-transistor SRAM has an access time of 1.21 ns, a larger memory will have slightly longer access time. It is also known that 2 input multiplexers have a propagation delay of about 0.15 ns. All these numbers are for the 0.18 mm generation of technology. It is probable that our architecture has a cycle delay of about 2 ns and since we handle one routing lookup request every clock cycle the maximum throughput is approximately 500MPackets/s.

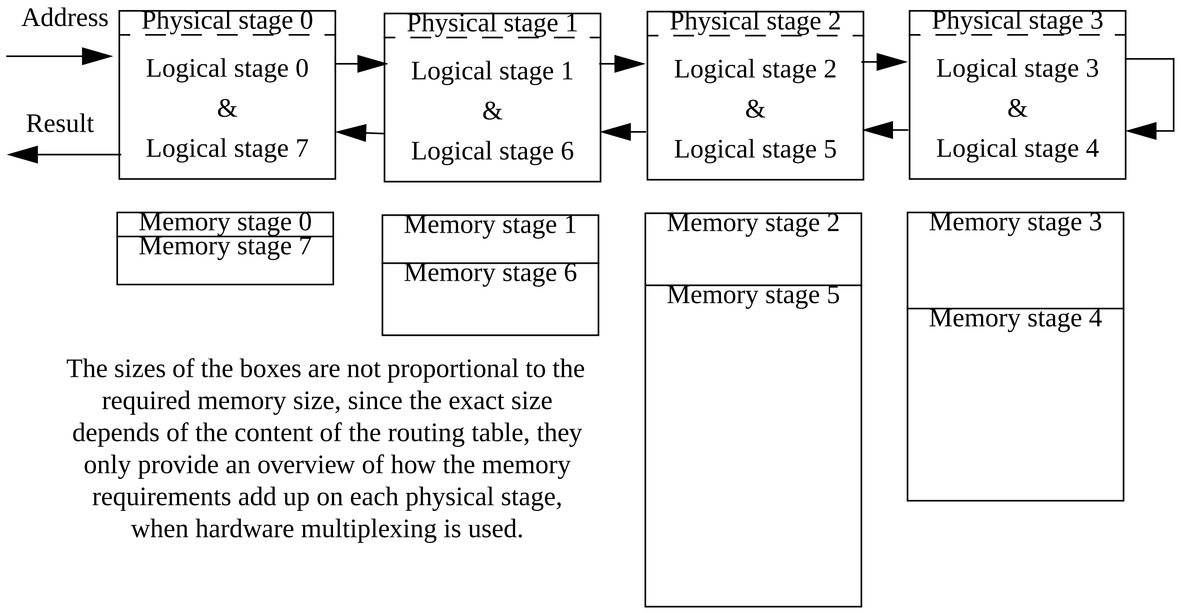
Table 1: Memory Requirements for Mae-East, 16416 Prefixes.

Stage	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10
0	2	4	8	16	32	64	128	256	512	1024
1	4	16	48	160	608	1728	6272	21248	76288	274432
2	8	40	216	1328	8576	52736	258048	713728	2384896	5517312
3	12	108	1192	13184	91264	298112	794496	4608	5632	1024
4	20	332	6592	44608	172416	1152	640			
5	38	1072	22816	86208	320	64				
6	54	3296	37264	288	32					
7	98	8064	49656	80						
8	166	11152	144							
9	298	18632	88							
10	536	21552	8							
11	978	26788								
12	1648	72								
13	2676	36								
14	4032	20								
15	5704	4								
16	5576									
17	7440									
18	9316									
19	9478									
20	10776									
21	12414									
22	13394									
23	14100									
24	36									
25	20									
26	18									
27	22									
28	10									
29	4									
30	2									
31	2									
Total	98882	91188	118032	145872	273248	353856	1059584	739840	2467328	5793792

5.2 Hardware Multiplexing

In order to decrease the differences between the memory requirements of the stages a trade-off with the throughput can be made. The idea of hardware multiplexing is to use each physical stage for two or more logical stages of the k-multibit trie search algorithm. This can be done in different ways, e.g. mirroring, serial stage reuse, and full loops. For the example of mirroring, see Fig. 3. Each physical stage will need an extra multiplexer in order to chose the correct input. There is also a need for a control part, which supports the select signals to all multiplexers according to the multiplexing scheme selected. It is also roughly shown how the memory requirements add up on each physical stage.

To describe the three previously mentioned multiplexing schemes we define pi as the i:th physical stage and li as the i:th logical stage. The tuple <pi, lj> represents that logical stage j



The sizes of the boxes are not proportional to the required memory size, since the exact size depends of the content of the routing table, they only provide an overview of how the memory requirements add up on each physical stage, when hardware multiplexing is used.

Figure 3: Hardware multiplexing by mirroring for k=4

is executed on physical stage i. In tables 2-7 it can be seen how the scheduling for the three schemes work for k=4 with 2 and 4 logical stages being executed on each physical stage

Table 2: Scheduling for Mirroring, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1, l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6	p_0,l_7							
2			p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6	p_0,l_7					
3					p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6	p_0,l_7			
4							p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l0	p_1,l_6	p_0,l_7	
5									p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4	p_2,l_5	p_1,l_6
6											p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_3,l_4
7													p_0,l_0	p_1,l_1	p_2,l_2
8															p_0,l_0

Table 3: Scheduling for Double Mirroring, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1, l_1	p_1,l_2	p_0,l_3	p_0,l_4	p_1,l_5	p_1,l_6	p_0,l_7							
2			p_0,l_0	p_1,l_1	p_1,l_2	p_0,l_3	p_0,l_4	p_1,l_5	p_1,l_6	p_0,l_7					
3									p_0,l_0	p_1,l_1	p_1,l_2	p_0,l_3	p_0,l_4	p_1,l_5	p_1,l_6
4											p_0,l_0	p_1,l_1	p_1,l_2	p_0,l_3	p_0,l_4

respectively. Each row is representing a task and each column represents a clock cycle.

It can easily be seen that all three schemes provide full utilization of the provided physical stages, both for double and quadruple multiplexing. The exact scheduling determines when results will be ready. This differs dependent on which scheme that is used and the results can be seen in the tables 2-7. Although necessary for the exact implementation it is of less general interest.

The fact of how the memory requirements add up, on the other hand, is most interesting, since now we have to consider the total memory on one physical stage as the performance limiting factor. As mentioned above, an extra multiplexer for the input selection and some control logic for selecting the correct input is also needed, so now the critical path will con-

Table 4: Scheduling for 2 Serial Reuses, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7							
2			p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7					
3					p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7			
4							p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6	p_3,l_7	
5									p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4	p_2,l_5	p_3,l_6
6											p_0,l_0	p_0,l_1	p_1,l_2	p_1,l_3	p_2,l_4
7													p_0,l_0	p_0,l_1	p_1,l_2
8															p_0,l_0

Table 5: Scheduling for 4 Serial Reuses, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_0,l_1	p_0,l_2	p_0,l_3	p_1,l_4	p_1,l_5	p_1,l_6	p_1,l_7							
2					p_0,l_0	p_0,l_1	p_0,l_2	p_0,l_3	p_1,l_4	p_1,l_5	p_1,l_6	p_1,l_7			
3									p_0,l_0	p_0,l_1	p_0,l_2	p_0,l_3	p_1,l_4	p_1,l_5	p_1,l_6
4													p_0,l_0	p_0,l_1	p_0,l_2

Table 6: Scheduling for 2 Full Loops, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1, l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7							
2		p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7						
3			p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7					
4				p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6	p_3,l_7				
5									p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5	p_2,l_6
6										p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4	p_1,l_5
7											p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3	p_0,l_4
8												p_0,l_0	p_1,l_1	p_2,l_2	p_3,l_3

Table 7: Scheduling for 4 Full Loops, k=4

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	p_0,l_0	p_1, l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5	p_0,l_6	p_1,l_7							
2		p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5	p_0,l_6	p_1,l_7						
3									p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5	p_0,l_6
4										p_0,l_0	p_1,l_1	p_0,l_2	p_1,l_3	p_0,l_4	p_1,l_5

sist of 3 multiplexer delays + the memory access time. Table 8 shows the memory requirements per physical stage when the various multiplexing schemes are used on the mae-east routing table. For this routing table full loops are best when each stage is used twice, but double mirroring is better when using each stage 4 times. The purpose is obviously to minimize the maximum memory requirement per physical stage, the emax when hardware multiplexing is used.

Using the right multiplexing scheme makes it possible to only slightly increase the maximum memory requirement per stage, e.g. for double mirroring, k=2 it increases from 26788 to 27300. We can approximate that the memory access time will remain the same and the addition of an extra multiplexer will only minorly influence the minimum clock cycle period. There will also be extra wiring needed and the output of some stages will have a higher fan-out. Totally, however, the multiplexed solution should be able to run almost as fast as

Table 8: Memory requirements when multiplexing is used for mae-east, 16416 prefixes

Stage	Mirroring		Double mirror		2 serially		4 serially		2 full loops		4 full loops	
	k = 2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4	k=2	k=4
0	8	96	19224	57888	20	176	168	14688	11156	44624	11560	46240
1	36	448	21964	87984	148	14512	12764	131184	18648	86368	19756	99632
2	76	87536	22700		1404	130816	78124		21592	1616	24908	
3	180	57792	27300		11360	368	132		26896	13264	34964	
4	27120				29784				404			
5	22624				48340				1108			
6	21928				108				3316			
7	19216				24				8068			
Total	91188	145872	91188	145872	91188	145872	91188	145872	91188	145872	91188	145872

the original configuration. It must be noticed that since each physical stage is used 2 or 4 times for each lookup request, the throughput will be decreased with the same factor.

Interestingly one can use the same configuration of, for example, 8 physical stages to implement a scheme with k=2 with multiplexing (mirroring and 2 full loops are almost equally good) and also a scheme with k=4 with no multiplexing. Since the memory sizes must be determined at the time of manufacturing this observation increases the flexibility of the architecture. A small routing table will fit in the memory with k=4 and the throughput can be kept maximal. When the routing table grows too big, k can be changed to 2 and multiplexing can be used so the same hardware can support increasingly bigger routing tables, with a throughput decrease. Even bigger routing tables will fit in with k=1 and double mirroring.

Building the memory in blocks can also allow post-manufacturing memory allocation to the stages by using configurable interconnects. This however introduces more hardware overhead and increases the access time.

5.3 Scaling

When talking about the scalability of an IP address lookup implementation, two different views can be considered. First it is the scaling to more prefixes in the routing table and second the scaling to longer addresses, e.g. 128 bit IPv6 addresses.

To investigate the scalability of the routing table, all five routing tables from IPMA were used. The simulations were performed for k=1, 2, and 4. As already noticed k=2x gives good values and k should be kept fairly small in order to limit the size of the memories. The results can be seen in Fig. 4. The interesting fact is that for growing routing tables the emax grows less than linearly for k=4. For k=2 the growth is approximately linear for the used routing tables. This is because many routing prefixes share the same entry in the k-mulitbit trie. With an increasing number of prefixes, this aggregation helps keeping the emax limited. For bigger k, emax grows, but one must keep in mind that the number of stages decreases with a factor of k, so the total memory requirement does not grow as rapidly as emax.

The second scalability issue is harder to investigate, since there is a lack of accessible IPv6 routing tables of sufficient size. IPMA provides us with one, which was used as of December 31st, 2000 23:14. It only contains 219 distinct prefixes and the simulation gives the results of table 9. It is hard to tell how well our architecture will handle big IPv6 tables. What is clear, however, is that the same principle is applicable to IPv6 and that the delay until the result is available will increase since more stages are needed. How the throughput is affected is dependent of how much emax grows.

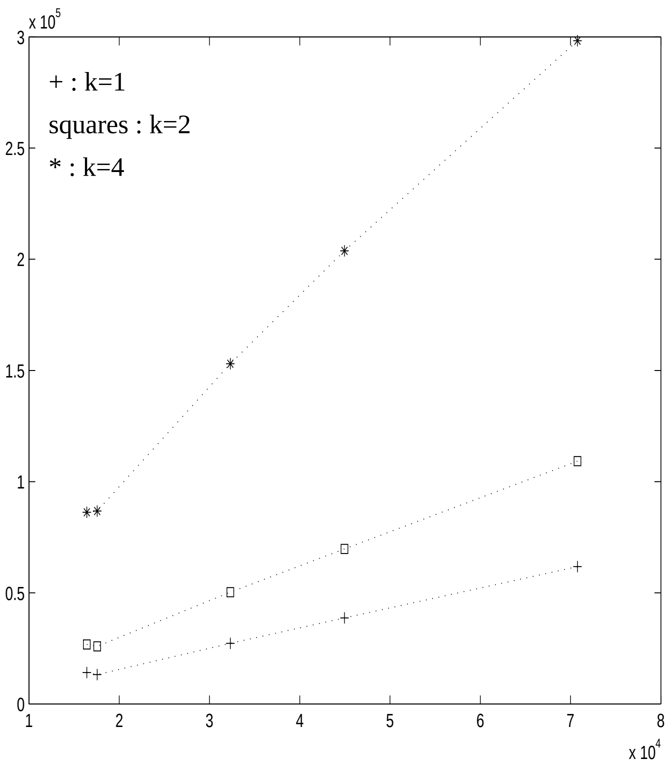


Figure 4: (Maximum memory entries per stage, e_{max}, as a function of number of prefixes, n

Table 9: Memory requirements for IPv6 routing table

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=16
e_{max}	4	16	40	128	384	1280	4480	14336	7864320
Total	510	812	1424	3168	7904	20160	47488	100352	20316160

6 Updating and Further Extensions

6.1 Updating Issues

A routing table is constantly changing its content. The implementation must allow instant incremental changes to the routing table. The k-multibit trie implementation in software has an updating complexity of O(W/k + 2k) [2]. Our hardware implementation has the same complexity and more precisely, an update requires at most 2k writes in the memory at each stage. By scheduling these writes so that they are pipelined in the same fashion as the lookup requests, only 2k clock cycles will be lost for an update since W/k stages can be accessed in parallel. This all requires that the routing update engine has a copy of the memory structure, but since it is normally implemented on a general purpose CPU this is not a problem. The C-program used for simulation is capable of generating the necessary memory structures for configuration and updating of the forwarding engine.

Another tempting way of managing the updating is to make use of dual-port memories. These memories are almost as fast as single port, but require more area and totally smaller storage capacity is supported [13]. However, the updating could be performed at the same time as a lookup request is serviced, which would allow zero time overhead for updates. An issue that has to be addressed is that of the consistency. When a new rule is added this can require a maximum of 2k writes in each stage as previously mentioned, therefore an update may be partial at the time of a lookup, which could lead to an incorrect forwarding decision.

6.2 2-Dimensional Search

IP packet forwarding based only on the destination address limits the router performance to best effort, since no differentiation of packets can be done. To improve this more fields from the packet header, such as source address, transport layer protocol, TCP/UDP source port and destination port are used. Each field can be seen as a search dimension. Normally the different dimensions have different search criteria, only IP destination address and source address can be assumed to have longest prefix match search. Transport layer protocol typically requires exact match and TCP/UDP ports often require range matching.

Thanks to the work done by Srinivasan et al. [6], it is known that the k-multibit trie search algorithm can be used also for 2-dimensional longest prefix match. Since our architecture is a direct implementation of this algorithm, it can also handle the search on source address as well as destination address. Further investigations, on how this affects the number of stages and the emax have not been performed, since there is also a lack of such routing tables.

7 Related Work

Through the last years some papers have been presented on IP address lookup. Here, we compare our solution to these.

In 1997 Zitterbart et al. [10] presented an IP lookup scheme for IPv6. The architecture is based on a binary search algorithm and thus has 128 stages. They consider an FPGA-based implementation with discrete SRAM components since also pipelining is not considered they can only support 62.5 kPackets/s.

Gupta et al. [8] presented a routing lookup scheme, based on DRAMs. They compared their solution to software algorithms and had a big improvement. The lookup needs only 2 memory accesses and can be pipelined, reaching a performance of approximately 20MPackets/s using DRAMs with 50 ns access time.

Huang and Zhao presented a scheme based on SRAM lookup [9], which needs maximally three memory accesses. This work was later improved by Wang et al. [11] to handle IP address lookup with only 2 SRAM accesses. There is a need for some logic, including adders and multiplexors in their solution. They have used other routing tables to evaluate their architecture and therefore it is hard to compare the memory requirements, but our solution seems to have smaller memory requirements. In their work a routing table of 44075 prefixes needs 456 kilo byte (kB) for three memory accesses and 565 kB for two memory accesses respectively. We used one routing table with 44959 prefixes and need only 331 kB when k=2 is used and minimal word lengths for each memory are used.

We have smaller memories and less logic between the memories, which gives us higher throughput. Their latency is however smaller than ours, since we need in total more memory accesses.

The two last discussed architectures do not consider IPv6 addresses and when examining them it is obvious that they are very specific for 32 bit addresses. Our architecture is general and can be used for arbitrary long addresses.

In [14], Kim extends the architecture by Huang et al. to handle IPv6 addresses by limiting the search to 48 bits in the 128 bits address space. Thereby a performance of 4 MPackets/s is achieved in an FPGA implementation. It is mentioned that it is very hard to extend the search to the full 128 bit length of IPv6 addresses.

8 Conclusions and Future Work

We have designed and simulated an IP address lookup architecture based on one SRAM access and two multiplexers in each pipeline stage. The performance reached is about 500MPackets/s, based on SRAM performance. Hardware multiplexing has been described as a way of making a trade-off between throughput and hardware reuse, double mirroring is a very efficient scheduling technique in limiting the increased memory need per stage. Hardware multiplexing also increases the flexibility of the architecture. The architecture scales better than linearly for growing routing tables, but it is hard to conclude the behavior for IPv6 addresses.

By using only SRAM and multiplexers, the lookup engine is made suitable for SoC integration.

The architecture is capable of 2-dimensional search, but in the near future even more dimensions need to be searched. It is intended to extend the architecture in order to make it capable of handling search in more dimensions.

Acknowledgments

The authors would like to thank SwitchCore for interesting discussions.

References

[1] Attia, M., Verbauwhede, I.: Programmable Gigabit Ethernet Packet Processor Design Methodology, ECCTD 2001, Vol. III, pp. 177-180

[2] Ruis-Sanchez, M. A., Biersack, E. W., Dabbous, W.: Survey and Taxonomy of IP Address Lookup Algorithms, IEEE Network, March/April 2001, pp. 8-23

[3] Gupta, P., Prabhakar, B., Boyd, S.: Near-Optimal Routing Lookups with Bounded Worst Case Performance, IEEE Infocom 2000, pp. 1184-1192

[4] Ergun, F., Hittra, S., Sahinalp, S. C., Sharp, J., Sinha, R. K.: A Dynamic Lookup Scheme for Bursty Access Patterns, IEEE Infocom 2001, pp. 1444-1453

[5] Broder, A., Mitzenmacher, M.: Using Multiple Hash Functions to Improve IP Lookups, IEEE Infocom 2001, pp. 1454-1463

[6] Srinivasan, V., Varghese, G., Suri, S., Waldvogel, M.: Fast and Scalable Layer Four Switching, Proceedings of ACM SIGCOMM 98, pp. 191-202

[7] Internet Routing Table Statistics, http://www.merit.edu/ipma/routing_table/

[8] Gupta, P., Lin, S., McKeown, N.: Routing Lookups in Hardware at Memory Access Speeds, IEEE Infocom'98, San Francisco, CA, April 1998, pp. 1240-1247

[9] Huang, N.F., Zhao, S.M.: A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers, IEEE Sel. Areas in Comm., Vol. 17, No. 6, June 1999, pp. 1093-1104

[10] Zitterbart, M., Harbaum, T., Meier, D., Brokelmann, D.: Efficient Routing Table Lookup for IPv6, IEEE Workshop on High-Performance Communication Systems, 1997, pp. 1-9

[11] Wang, P.-C., Chan, C.-T., Chen, Y.-C.: A Fast IP Routing Lookup Scheme, IEEE Communications letters, Vol. 5, No. 3, March 2001, pp. 125-127

[12] Noda, K., Takeda, K., Matsui, K., Ito, S., Masuoka, S., Kawamoto, H., Ikezawa, N., Aimoto, Y., Nakamura, N., Iwasaki, T., Toyoshima, H., Horiuchi, T.: An Ultrahigh-Density High-Speed Loadless Four-Transistor SRAM Macro with Twisted Bitline Architecture and Triple-Well Shield, IEEE JSSC, Vol. 36, No. 3, March 2001, pp. 510-515

[13] Embedded Memories, http://www.umc.com/english/design/e.asp

[14] Kim, D.: Design and Implementation of a Gigabit Wireless Router, master thesis at Electrical Engineering Department at UCLA, pp. 17-26