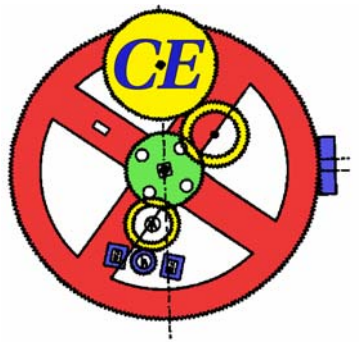


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Design and Implementation of a Range Trie for Address Lookup

Georgios Stefanakis

CE-MS-2009-07

Abstract

The rapid growth of internet traffic and the eminent shift from IPv4 to IPv6 addresses indicated the need for an efficient address lookup method that can keep pace with the ever-increasing throughput demands. As current address lookup solutions tend to become the bottleneck in internet routing, the efficient Range Trie address lookup method was designed for a hardware implementation in this thesis. A complete Range Trie hardware design was derived that is parameterizable in many of its aspects. The proposed design offers the required properties of low latency, high throughput and low memory requirements, while these properties scale well with the increase of the address width and/or the lookup table size. After implementing the Range Trie using 90nm ASIC process, an operating frequency of 540MHz up to 694MHz was achieved for IPv4 addresses, depending on the lookup table size (256-512K entries). For IPv6, the throughput was sustained between 442MHz up to 571MHz. For both IPv4 and IPv6, an OC-3072 (160Gbps) wire speed is supported for the worst-case of 512K ranges. The memory requirements ranged from a few KBytes up to 24 MBytes, while the occupied area ranged from 0.014 cm^2 up to 4.64 cm^2 and the power consumption ranged from 0.2 W up to 31 W depending on the address width (32, 64 or 128-bits) and the lookup table size (256-512K entries).

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Design and Implementation of a Range Trie for Address Lookup

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Georgios Stefanakis
born in Athens, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design and Implementation of a Range Trie for Address Lookup

by Georgios Stefanakis

Abstract

The rapid growth of internet traffic and the eminent shift from IPv4 to IPv6 addresses indicated the need for an efficient address lookup method that can keep pace with the ever-increasing throughput demands. As current address lookup solutions tend to become the bottleneck in internet routing, the efficient Range Trie address lookup method was designed for a hardware implementation in this thesis. A complete Range Trie hardware design was derived that is parameterizable in many of its aspects. The proposed design offers the required properties of low latency, high throughput and low memory requirements, while these properties scale well with the increase of the address width and/or the lookup table size. After implementing the Range Trie using 90nm ASIC process, an operating frequency of 540MHz up to 694MHz was achieved for IPv4 addresses, depending on the lookup table size (256-512K entries). For IPv6, the throughput was sustained between 442MHz up to 571MHz. For both IPv4 and IPv6, an OC-3072 (160Gbps) wire speed is supported for the worst-case of 512K ranges. The memory requirements ranged from a few KBytes up to 24 MBytes,, while the occupied area ranged from 0.014 cm^2 up to 4.64 cm^2 and the power consumption ranged from 0.2 W up to 31 W depending on the address width (32, 64 or 128-bits) and the lookup table size (256-512K entries).

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-07

Committee Members :

Advisor:	Dr.ir. Georgi N. Gaydadjiev, Assistant Prof., CE, TU Delft
Advisor:	Dr. Ioannis Sourdis, CE, TU Delft
Chairperson:	Dr. Koen Bertels, Associate Prof., CE, TU Delft
Member:	Dr. Cristian Doerr, Assistant Prof., NAS, TU Delft

To those who inspire me...

Acknowledgements

First of all, I am thankful to Ioannis Sourdis and Georgi Gaydadjiev for offering me this thesis topic and, most importantly, for their guidance and support during the progress of my research and also during my MSc studies. I would also like to thank all of my (current and previous) professors that brought me to where I stand now.

There are also a lot of people whose help was vital to the completion of this thesis. First of all, Ruben de Smet whom I collaborated with, as we both worked on the Range Trie method. Also, Rene van Leuken, Arjan van Genderen, Daniele Ludovici and Radu Stefan that assisted me with the issues regarding the successful usage of the Design Compiler tool. I am certain that there are also a lot more people that helped me with their insights and experience. I am thankful to you too.

I am grateful to my parents and my brother for their support and encouragement. I am also glad about the great friends I made during my stay in Delft. My life would definitely be less interesting without you. Many thanks to Angelos, Stavros, Maria, the Georges, Nikos and all the others that we spent time together and had fun.

Finally, I would like to thank the CE administrator, Erik de Vries, and the CE secretary, Lidwina Tromp, for their valuable technical and administrative assistance.

Georgios Stefanakis
Delft, The Netherlands
June 25, 2009

Contents

List of Figures

List of Tables

List of Acronyms

ASIC	Application-Specific Integrated Circuit
BU-SLC	Bottom-Up with Single Length Comparisons
BU-VLC	Bottom-Up with Variable Length Comparisons
CAMP	Circular, Adaptive and Monotonic Pipeline
CIDR	Classless InterDomain Routing
FPGA	Field-Programmable Gate Array
HiCuts	Hierarchical Intelligent Cuttings
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LC-Tries	Level-Compressed Tries
MPPS	Mega Packets Per Second
PATRICIA	Practical Algorithm To Retrieve Information Coded In Alphanumeric
SRAM	Static Random Access Memory
TCAM	Ternary Content Addressable Memory
TD-SLC	Top-Down with Single Length Comparisons
TD-VLC	Top-Down with Variable Length Comparisons

1 Introduction

"You 'll find your way, John. You always do."
-Benjamin Linus to John Locke

Modern societies rely heavily on information. If information was to stay in one place, then there would be no much use of it. This is what led to the vast development and employment of information networks, ranging from simple computer networks to advanced telecommunication networks. In any of these cases, information travels throughout the network and it is imperative to find its way from its source to its destination.

In the case of computer networks, information (represented as data) is divided into packets that are routed from the source towards the destination through intermediate nodes. Each packet consists of a header (that contains information about the source and the destination) and the payload (that contains the actual data). In each intermediate node it must be decided where to forward the packet based on a routing table and the destination. The destination address is looked up in the routing table to decide the action to be taken (see Figure 1.1). This process of searching in the routing table is called *address lookup*.

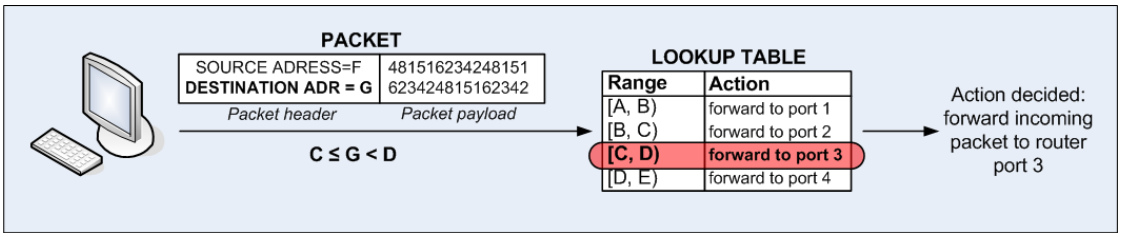


Figure 1.1: A simple depiction of address lookup for routing purposes. Destination address G is looked up and found to belong in range $[C, D)$.

A lot of research is available regarding routing and its issues. This thesis focuses specifically on the address lookup problem in the context of internet routing and presents an efficient and scalable hardware design for address lookup based on the novel Range Trie algorithm introduced by I. Sourdis in [26].

The rest of this introductory chapter is organized as follows: Section 1.1 explains in more details the address lookup and gives an overview of its use in a variety of research fields. Section 1.2 focuses on the specifics of the address lookup current problems that motivated this thesis. Section 1.3 states the goals of this thesis and its contributions. Finally, section 1.4 concludes this chapter with an overview of the thesis.

1.1 Address Lookup

It was previously stated that our focus is on address lookup. Address lookup is an elementary operation in computer networks (and information exchange networks in general). Every network routing element must support it and its significance triggered the development of the address lookup research field.

In computer networks, each network element is characterized by its Internet Protocol (IP) address. In the case of Internet Protocol version 4 (IPv4), each IP address is represented as a 32-bits wide binary number. Address lookup for computer networks may be defined as the procedure of determining the range that an IP address belongs to, out of a given set of IP address ranges (routing table). Given an address space $[0, 2^n)$ and k unique IP addresses/bounds A_i, where $0 < A_i < 2^n - 1$ and $i = 1, 2, ..., k$, that define $k + 1$ address ranges R_j $(j = 1, 2, ..., k + 1)$, then an address lookup is to determine the range R_j an incoming address A_{IN} belongs to. In the case of IPv4, $n = 32$. Figure 1.2 depicts graphically an example address lookup, as specified above, for an address space consisting of 6 ranges $(k = 5)$ and an incoming address A_{IN} belonging in range R_3 $(A_2 \leq A_{IN} < A_3)$.

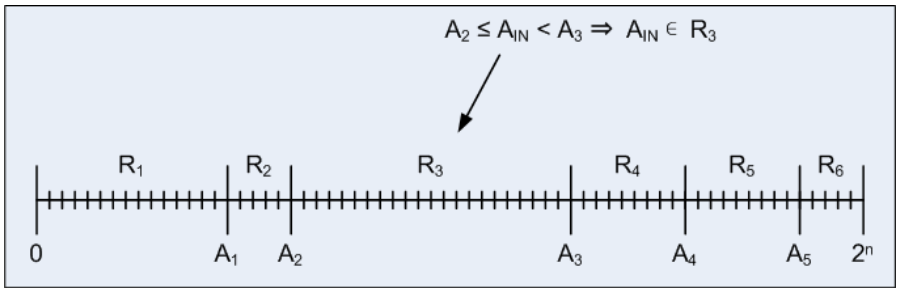


Figure 1.2: Address lookup problem specification

Address lookup is not limited just to computer networks. It could also be applied to other research fields that may or may not relate to network routing. In general, this lookup process narrows down to determining in which range an address belongs to out of a given set of ranges.

A variety of research domains may need this function and thus a lot of research effort has been spent on address lookup. A list of some of these research fields follows:

- In *internet routing* [22], as mentioned, routers need to forward packets based on the destination IP address of a packet. The decision is made by searching the destination IP address in the router's lookup table.
- In *packet classification* [31], [13], [24], [9] a lookup must be performed for one or more of the packet header fields to classify the packet based on a set of rules and then perform a defined action.
- In *interprocessor communication*, the newly proposed progressive address translation of virtual addresses to physical ones [16] needs to perform a translation lookup as the interprocessor communication shares a common address space.

The rest of this thesis will focus more on the application of address lookup of IP addresses for routing purposes. In any case, the findings of this thesis could be applied to all the mentioned research fields that have a need for address lookup.

1.2 Problem Statement

In the previous section, it was stated that a lot of research has been performed on address lookup for a variety of research fields. The question now is why deal with address lookup, since it is already a mature research domain, as old as network routing. In this section the issues that motivated this thesis are presented.

The rapid growth of internet traffic (see Figure 1.3), the increase in the number of network devices and the subsequent growing size of routing tables make more difficult for address lookup to keep pace with the increasing need for faster processing rates posed by the technological advancements in communication speed and bandwidth. Furthermore, the transition from the 32-bits wide IPv4 addresses to 128-bits wide IPv6 addresses demands for address lookup solutions that may scale efficiently in terms of the address width.

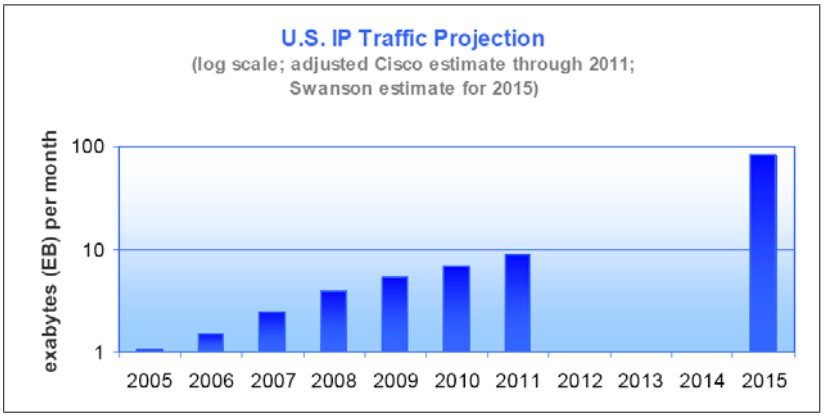


Figure 1.3: Projected U.S. Internet traffic growth until 2011 and for 2015. An exabyte equals to 2^{60} *bytes. (Source: [30])*

The currently available lookup solutions have started lagging behind; they may have worked well for past routing conditions but need to be improved in order to handle the multi-gigabits-per-second traffic rates. In particular, a mechanism is desired for address lookup that has *low latency*, *high throughput* and *low memory requirements*. At the same time, it should *scale efficiently* when the address width and/or the number of ranges increases. By scaling efficiently, we mean that an increase in address width and/or the number of ranges should affect minimally the latency, throughput and memory requirements. Out of these requirements, extra effort must be placed on the memory requirements because the size/bandwidth of the memories tends to dictate the efficiency and cost of the implementations.

Although these requirements are targeted for dealing with upcoming internet routing problems, there are present also to the other research fields that use address lookup

and they could benefit from solving these problems. In packet classification, a high throughput is still needed for looking up the multiple fields of the incoming packets, although the number of ranges is significantly smaller than those of internet routing. Similarly, in interprocessor communication [16], where the number of ranges is also small, there are higher constraints for lookup latency and throughput in order to sustain the performance of multicore systems.

In general, it could be said that the problem is that address lookup has the trend of becoming the bottleneck in the systems that they use it. An address lookup must be performed at wire speeds. At the same time the other requirements for a good lookup scheme (memory, scalability) should be considered [33]. The problem of designing such a scheme is the main motivation behind this thesis.

There is already a variety of algorithms and methods for address lookup that have started lagging behind, as already mentioned. This thesis focuses on the Range Trie algorithm introduced by I. Sourdis in [26] that promises to solve these problems.

The Range Trie algorithm posed an extra motivation behind this thesis because it is addressed here from the aspect of hardware design and implementation. Although the work in [25] and [6] proved the benefits of the Range Trie algorithm, this thesis focuses on an efficient hardware design and implementation of the Range Trie in order to sustain its benefits in a real-world design.

1.3 Thesis Goals and Contributions

In the previous section, the problems that arise in current solutions for address lookup were described. This thesis tries to solve the problems mentioned before by an efficient hardware design and implementation of the Range Trie algorithm.

The goal of this thesis is a design and implementation of the Range Trie algorithm that exploits optimally the inherit characteristics of the Range Trie structure: (a) low lookup latency, (b) high throughput, (c) low memory requirements, (d) acceptable scalability of (a)-(c) in terms of the lookup address width and number of address ranges.

The contributions of this thesis are:

- **A hardware design of the Range Trie algorithm:** For the first time, the Range Trie algorithm is designed for a hardware implementation. During the design, extra effort was used to ensure that the hardware design exploits the characteristics of the Range Trie algorithm, as mentioned in (a)-(d) above. The resulting design is parameterizable in terms of address width, memory bandwidth and number of processing stages in order to accommodate the address lookup needs of the application under consideration.

- **A complete design flow for hardware implementation and validation:** A complete design flow was created for generating Range Trie instances. Starting from the required design parameters and a Range Trie structure, generated by Ruben de Smet in [6], a synthesizable design of a Range Trie is generated in a hardware description language, along with the means to (a) configure it according to the given structure and (b) validate its correct operation.

- **Evaluation of the Range Trie design:** A variety of Range Trie instances was generated and synthesized for 2 ASIC technologies (90nm and 130nm) in order to evaluate each design point in terms of operating frequency, area, power consumption and memory requirements. This design space exploration proved the Range Trie scalability and offers the designer a chance to choose the suitable parameters for its case. A comparison was performed with other existing solutions for address lookup. The findings may be used accordingly to other research fields that need address lookup.

The means to achieve the goals and the contributions of this thesis are presented in the rest of this text's chapters, as described in the following overview of this thesis.

1.4 Thesis Overview

This section concludes the introductory chapter of this thesis and presents the overview of the following chapters.

In Chapter 2, the related work that exists in the literature regarding address lookup is presented. The presented solutions are both algorithmic approaches and hardware-targeting schemes and they origin mainly from the network routing and packet classification research domains. This chapter also introduces and details the Range Trie algorithm of [26] that is the main focus of this thesis.

Afterwards, the hardware design of the Range Trie algorithm is explained in detail in Chapter 3. All the design steps are discussed for obtaining a complete, synthesizable design. Chapter 4 presents and evaluates the results of synthesizing the Range Trie design for ASIC.

Finally, Chapter 5 concludes this thesis by summarizing its contributions and conclusions and by presenting some ground for future works.

2 Background

Address lookup is the operation of looking up in a table to find a range that an incoming address belongs to. As mentioned in Section 1.2, a good address lookup scheme should have low lookup latency, high throughput, low memory requirements and good scalability in terms of the lookup table size and the address width. There is already a variety of address lookup schemes but they have started lagging behind due to the recent internet traffic/speed growth. Thus, it is imperative to improve on the current schemes. The Range Trie algorithm, introduced in [26] promises a lookup scheme that satisfies the posed requirements. This Range Trie approach is the main focus point of this thesis.

In this chapter, the background material that is needed for the rest of this thesis will be presented. After discussing on a variety of related designs and algorithms for address lookup (Section 2.1), the Range Trie algorithm will be described (Section 2.2).

In order to understand the related methods for address lookup, it is important to start from the first attempts that tried to deal with the causes of the lookup problem.

Initially, the approach that was followed was to reduce the lookup tables growth rates by using different addressing schemes. The first addressing scheme that was used was a simple address allocation scheme that divided addresses into three classes. This is known as the *classful addressing scheme*. In this scheme the address was split into two parts: the network part, followed by the host part. There were three different classes of addresses with different network part widths. Addresses of class A, B, or C consisted of an 8, 16, or 24-bits network part and a corresponding 24, 16, or 8-bits host part. Doing an address lookup in this scheme was a relatively simple operation; a lookup was narrowed down to make an exact prefix match using standard algorithms based on hashing or binary search. This scheme worked well initially, but the continuous growth in the number of hosts and networks lead on the exhaustion of the IP address space rather quickly. Furthermore, the increase in the lookup table size was still prominent.

To deal with the problems of the clasful addressing scheme, the *classless interdomain routing* (CIDR) addressing scheme was introduced [11]. With CIDR the prefixes may be of variable length, instead of 8, 16 or 24-bits wide. This scheme allowed for a more efficient use of the IP addresses, along with the aggregation of addresses. So, the lookup tables needed to keep less prefixes. The trade-off was that the lookup process got more complicated as it was now required to make a longest prefix match instead of an exact prefix match.

Despite the use of CIDR, the routing tables kept growing. This resulted in a shift in the research effort from trying to reduce the inherit growth of the lookup table sizes into finding more efficient lookup methods that are scalable with respect to the number of lookup table entries. At the same time, the lookup methods should have low memory

requirements, low latency and fast lookup times. A new requirement that appeared with the introduction of IPv6 was to find lookup algorithms that are also scalable with respect to the address width.

As already mentioned, there is a plethora of currently available solutions for address lookup that try to achieve these requirements but have started to lag behind due to the internet traffic/speed growth. Some of these will be discussed in Section 2.1 of this chapter. In Section 2.2, the Range Trie algorithm, that is an improvement on the current solutions, will be presented in detail. The improvements of the Range Trie will be proven later in Chapter 4. Finally, this chapter concludes with its summary in Section 2.3.

2.1 Related Designs and Algorithms for Address Lookup

In this section, a representative set of address lookup solutions which exist in the related literature will be presented.

Before proceeding, it must be noted that these solutions approach the address lookup problem from two equivalent sides; others consider it as a longest prefix matching problem, others as a range lookup problem. The actual difference is in the way that they represent the lookup table. The former approaches consider the lookup table as a set of *address prefixes*. Each address prefix is in the form of a binary number followed by a star, which actually represents the range of addresses that start with this binary number. The latter approaches consider the lookup table as a set of *range bounds* that define the address ranges. Both representations of the lookup tables are equivalent (see Table 2.1) and suitable for address lookup. Although the research problem is the same, the difference is in the means to solve it. Former approaches research on performing a longest prefix matching, while the latter try to find a matching range.

Table 2.1: The address ranges represented as equivalent address prefixes. These address ranges/prefixes will be used as an example lookup table in the methods discussed in the rest of this section.

	Address Range	**Address Prefix**
a	[000000, 111111)	0*
b	[010000, 010010)	01000*
c	[011000, 100000)	011*
d	[100000, 111111]	1*
e	[100000, 101000)	100*
f	[110000, 110100)	1100*
g	[110100, 111000)	1101*
h	[111000, 111100)	1110*
i	[111100, 111111]	1111*

According to the taxonomy of Ruiz-Sanchez et. al. in [22] the existing address lookup solutions may be categorized into *"search on length"* or *"search on values"* approaches according to the dimension the search is based on (see Figure 2.1). Waldvogel et al. in

[32] added an extra classification of the methods depending on the type of the search traversal (sequential or binary search on length or values).

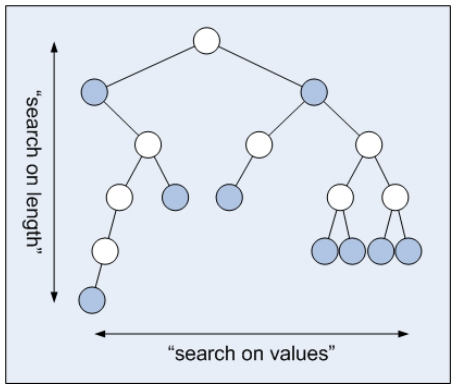


Figure 2.1: Address lookup search dimensions. The search space that is being traversed by a lookup method may be either on the length of the addresses ("search on length") or on their values ('search on values").

In the rest of this section, a variety of existing designs and algorithms for address lookup will be presented. These address lookup schemes may not come directly from the internet routing domain, but from other research domains that also need a form of address lookup, such as the packet classification research field. The presented related work is a mixture of algorithmic approaches to address lookup and hardware-targeting address lookup designs.

Specifically, in Sections 2.1.1 through 2.1.5 the sequential "search on length" approaches are considered starting from the elementary binary trie structure (in Section 2.1.1) and proceeding to more advanced trie structures. Then a binary "search on length" approach will be discussed in Section 2.1.6. Afterwards, the basics of sequential "search on values" will be presented in Section 2.1.7, followed by the binary "search on values" approach of range trees in Section 2.1.8. This section concludes with a set of hardware-targeting approaches in Sections 2.1.9 through 2.1.13.

2.1.1 Binary Trie

The most natural way to perform longest prefix matching for the purposes of address lookup is to represent the address prefixes using a trie. A trie performs a sequential "search on length". As shown on Figure 2.2, a trie is a tree-based data structure allowing the organization of prefixes on a binary basis by using the bits of prefixes to direct the branching [22].

Each node of the binary trie has at most two children, each one corresponding to the next bit of the address prefixes. A search in a trie is guided by the bits of the destination address. At each node, the search branches either left or right, depending on the next bit of the incoming address. This means that at the level l of the range trie, the l most significant bits of the incoming address have been inspected and address prefixes of length l may be matched. The search ends when there are no more branches to take and the search result is the last address prefix node encountered.

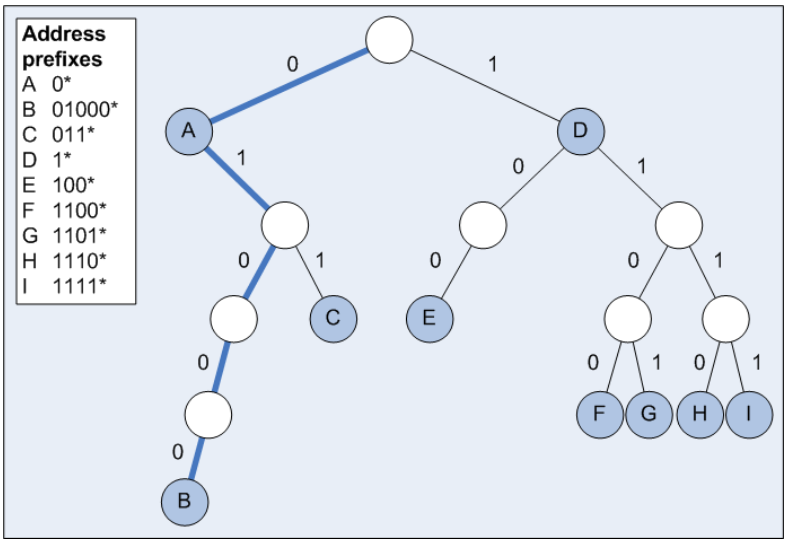


Figure 2.2: A binary trie for a set of address prefixes. Prefix nodes are shown in a darker color. As an example, the path followed to match the incoming address 010000 to prefix B is shown.

As mentioned, a binary trie search is a "search on length" sequential approach. At every step through the trie, the search space is reduced and a better match is trying to be found. A binary trie lookup for incoming addresses of length W takes $O(W)$ steps. This means that the binary tries doesn't scale well in terms of the address width. Furthermore, they produce an unbalanced decision tree with large depth and relatively high memory requirements.

2.1.2 Path-Compressed Tries

An improvement on the binary tries, presented in Section 2.1.1, are the *path-compressed tries* which are also a sequential "search on length" approach. Binary tries often suffer from long search paths that are a long sequence of one-child nodes (i.e. address prefix B in Figure 2.2). It may be avoided to traverse such paths and also reduce the needed memory by invoking a technique called *path compression*. Path compression removes one-way branch nodes, so that each non-leaf node has two child nodes, resulting in a more balanced trie with less levels. That way, during the traversal of the decision tree, some bits of the incoming address may not be inspected.

An example of applying path-compression on the trie of Figure 2.2 may be seen in Figure 2.3. Note that additional information is stored on the trie to ensure the correct matching operation during the traversal of the trie: (a) each node is annotated with the bit position of the incoming address to be inspected next and (b) prefixes may need to be stored in the nodes where path-compression was applied in order to perform a check that the prefix match is valid (due to the non-inspection of some bits on the one-child nodes paths, the depth of a node is no longer equal to the length of the address prefix stored).

Traversing the path-compressed trie is similar to the one of binary tries, except for

the following differences: (a) the bit of the incoming address to be inspected is the one indicated by the bit-number field in the visited node, instead of just the next bit, and (b) when a prefix node is encountered, a comparison to the actual address prefix is performed. If a match occurs, the prefix is stored as a match and the traversal continues. The procedure stops when reaching a leaf node or when a mismatch occurs. The result of the lookup is the last matched prefix.

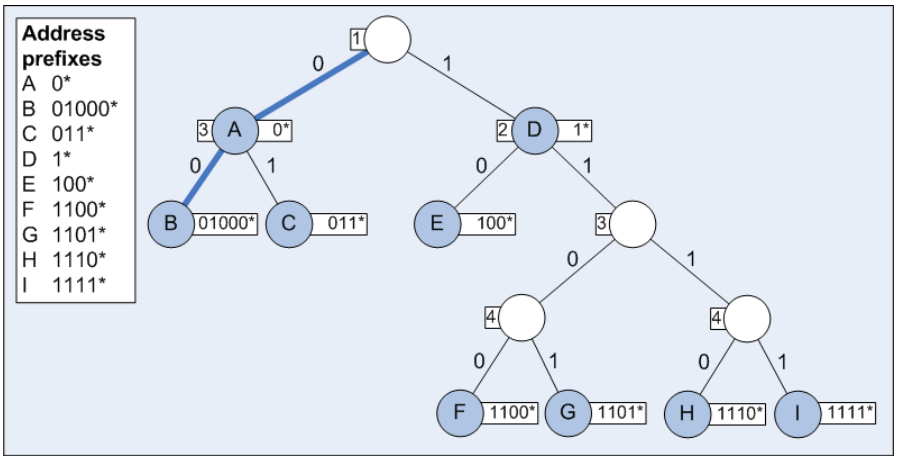


Figure 2.3: A path-compressed trie for a set of address prefixes. Prefix nodes are shown in a darker color. As an example, the path followed to match the incoming address 010000 to prefix B is shown. The inspection starts from the 1st bit of the incoming address (0) and thus we go to the left. This is a prefix node, so a successful prefix matching is performed for the corresponding part of the incoming address (0) and prefix A is stored as a match. Then we skip the 2nd bit of the incoming address and its 3rd bit (0) will be inspected as indicated by the bit-number field and thus we go to the left. This is again a prefix node, so a successful prefix matching is performed for the corresponding part of the incoming address (01000) and prefix B is stored as a match. Since this is a leaf node, the process completes and it reports a match to prefix B.

Path-compression was first introduced in the *PATRICIA* scheme by Morrison in [19]. This was modified later by Sklower, in [23], to also support longest prefix matching and non-contiguous masks. The most commonly available implementation of a path-compressed trie is the one found in the BSD Unix kernel, also known as *BSD trie*, where during the lookup process a backtrack occurs in order to retrieve the longest prefix match. This means that the worst-case search time is $O(2 * W)$, where W is the address width. Even for a efficient implementation of the BSD trie, the worst-case search time would be at best $O(W)$. This indicates that the path-compressed trie does not scale well in terms of the address width. This is the reason why the rest sequential "search on length" approaches to be presented focus on improving the worst-case search time complexity by reducing the trie depth furthermore.

2.1.3 Multibit Tries

Multibit tries are an improvement of the binary tries. They are a sequential "search on length" approach that provides a constant factor improvement. Instead of inspecting

one bit at a time, they inspect several bits simultaneously. The number of bits that are inspected per step is called *stride*. As a result, the trie depth decreases and the branching factor of a node increases, as each node has now 2^k child nodes, where k is the stride. A multibit trie may have a fixed stride (all nodes of a level have the same stride) or a variable stride. The process of searching in a multibit trie is almost identical to the binary trie, except that more bits may be inspected per step. An example variable-stride multibit trie based on the binary trie of Figure 2.2 is depicted in Figure 2.4.

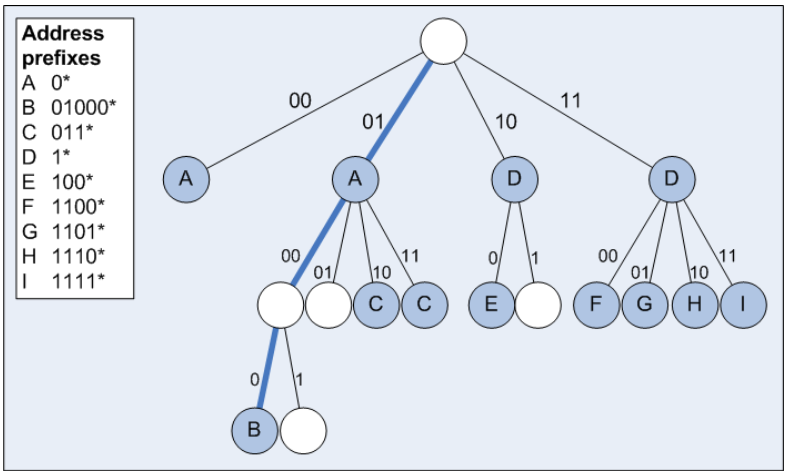


Figure 2.4: A variable-stride multibit trie for a set of address prefixes. Prefix nodes are shown in a darker color. The stride of each node is the number of bits that need to be inspected in the next search step. Note the duplication and prefix expansion of nodes A, D and C. As an example, the path followed to match the incoming address 010000 to prefix B is shown.

Choosing the optimal strides is a trade-off between search speed (trie depth) and memory requirements (number of nodes). Because multibit tries cannot support arbitrary prefix lengths, it is needed to transform the prefix addresses into a compatible set where some prefixes need to be expanded. For example, in Figure 2.4, prefixes A and A needed to be expanded into 2 bits and prefix C into 4 bits. Prefix expansion leads to more memory use. Also, using a fixed stride tends to waste more memory due to unnecessary node duplications.

In [28], Srinivasan et al. present a method to select the optimal strides for fixed-stride tries and variable-stride tries, based on the given address prefixes. They use dynamic programming in order to minimize memory requirements and guarantee a worst-case search time.

Another practical approach for choosing the strides was followed from Gupta et al. in [12]. They noticed that in a typical backbone router of the time most of the address prefixes had a length of 24 bits or less. So, they opted for a first-level stride of 24 bits and a second-level stride of 8 bits. In this way, only two steps are needed for address lookup in the cost of extra memory resources (i.e. just the first-level memory size is 32 MBytes).

2.1.4 Level-Compressed Tries

Nilsson et al. in [20] combined the multibit trie (see Section 2.1.3) with the path compression technique (see Section 2.1.2) to gain in search time. The new scheme they introduced is also a sequential "search on length" approach and is called *Level-Compressed Tries* (LC-Tries). In level-compression, k-level full binary subtries are recursively replaced with a corresponding one-level k-stride multibit trie. That way the initial k levels are "compressed" into one and the search time is reduced. An example LC-Trie is shown on Figure 2.5, based on the path-compressed trie of Figure 2.3.

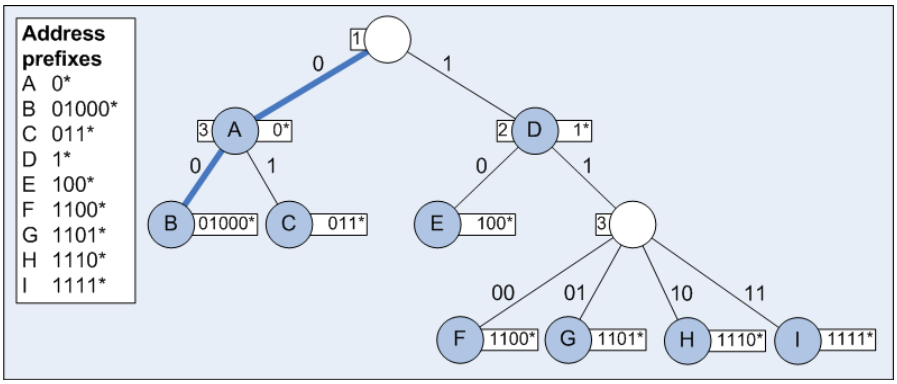


Figure 2.5: A Level-Compressed trie for a set of address prefixes. It is the result of transforming the path-compressed trie of Figure 2.3 into a multibit trie. As an example, the path followed to match the incoming address 010000 to prefix B is shown.

An optimization to the level-compression technique is to loose the criterion for subtrie replacement. Instead of demanding a full binary subtrie, we may demand only a fraction of branches to be present. The required fraction is represented by the *fill factor* x ($0 < x \leq 1$) that may be used as the deciding value. The fill factor offers the possibility of a trade-off between time (trie levels) and memory requirements (number of nodes); using low fill factors decreases the trie depth, by increasing the branching factor, but may introduce unnecessary leaf nodes.

This approach of combining multibit tries with compression yields fast search times and has been used by various schemes, like the *full expansion/compression scheme* by Crescenzi et al. in [5] and the *Lulea algorithm* by Degermark et al. in [7].

2.1.5 Hierarchical Intelligent Cuttings (HiCuts)

The *Hierarchical Intelligent Cuttings* (HiCuts) algorithm introduced by Gupta and McKeown in [14] is targeted for packet classification, where the search space is multi-dimensional.

In HiCuts, heuristic methods are used to partition the multi-dimensional space and to create a tree to search the partitioned space. Each leaf node of a HiCuts tree represents a small set (bucket) of matches that must be searched sequentially in order to obtain the best match. The characteristics of the HiCuts tree are decided based on the characteristics of the search space. The parameters of the HiCuts heuristics may be

tuned in for a trade-off between search time and memory requirements.

In our problem domain, where we deal with address lookup, the search space is single-dimensional. An example HiCuts tree for address lookup is depicted in Figure 2.6. Note that HiCuts actually constructs a variable-stride multibit trie with leaf nodes containing more than one address ranges.

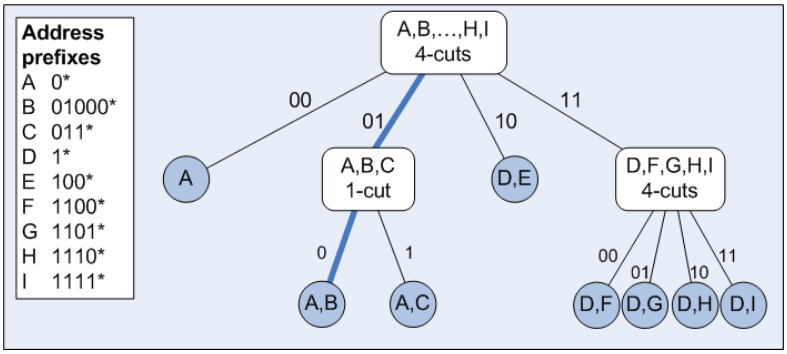


Figure 2.6: A HiCuts tree for address lookup for a set of address prefixes. The HiCuts method has been applied for a single dimension in order to support the single-dimensional search space of address lookup. The maximum bucket size was set to 2 and the maximum number of cuts was set to 4. Bucket nodes are shown in a darker color. An example traversal for matching the incoming address 010000 is shown. Traversing the HiCuts tree directs us to bucket $\{A, B\}$, where a sequential longest matching procedure matches the incoming address to the address prefix B.

A similar approach to HiCuts was introduced by Woo in [34], called *modular packet classification*. This is a solution for the packet classification problem based on a multi-stage search over ternary strings representing the classification rules. There are three steps of searching performed to classify a packet: search an index jump table that points to a search tree, searching the corresponding tree to reach a bucket, search into the bucket to retrieve a match. The second and third search resemble the HiCuts procedure. The interesting part is the beginning of the search where selected bits of the input packet are addressing the index jump table. An example of modular packet classification modified for address lookup is depicted in Figure 2.7.

2.1.6 Binary Search on Length

The trie "search on length" approaches presented so far were sequential approaches; in each step one or more bits of the incoming address were inspected leading to the reduction of the search space. Another sequential "search on length" approach would be to organize prefixes into different hash-tables according to their lengths and start the search from the hash-table holding the longest prefixes using hash techniques [32]. The latter technique does not result in a reduction of search time.

Waldvogel et al. in [32] suggested a binary "search on length" approach where the search space is reduced by half after every step. In each step the corresponding length hash-table is searched to check if a match exists in the given length and to decide how to proceed; in which half to continue the search. To ensure choosing the correct next step (checking for shorter or longer length) extra prefixes (called markers) are added to

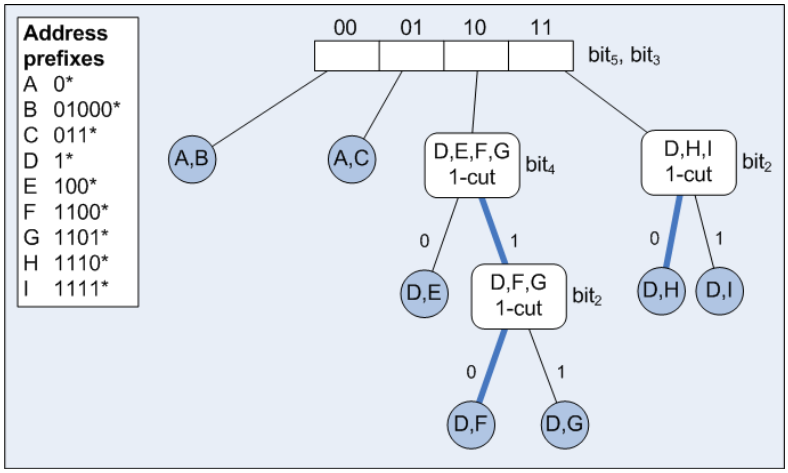


Figure 2.7: The modular packet classification approach for address lookup for a set of address prefixes. The approach has been applied for a single dimension in order to support the single-dimensional search space of address lookup. The maximum bucket size was set to 2 an the maximum number of cuts was set to 4. Bucket nodes are shown in a darker color. Bit_i denotes the bit to be inspected. An example traversal for matching the incoming address 110000 is shown. Traversing the structure is the same with HiCuts, except for the first step were bits Bit_5 and Bit_3 (10) are searched in the index jump table and guide the search to the corresponding search tree.

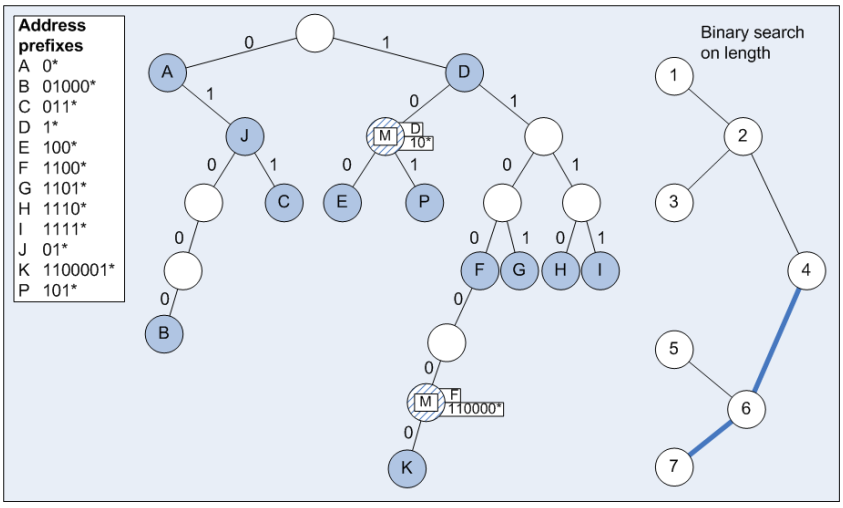


Figure 2.8: A binary "search on length" approach for a set of address prefixes. Prefix nodes are shown in a darker color. Markers are notated with 'M'. The trie is not used during the search; it is present only for clarification. As an example, the path followed to match the incoming address 11000010 to prefix K is shown. We begin by searching in the table for length 4 and a match occurs with prefix f, so we move to search the table for length 6. A match occurs with the marker 110000, so we move to search the table for length 7, where the final match with prefix K is found. (If the marker 110000* was not there, then a mismatch would have happened and we could not finally get the correct prefix match. If the incoming address was not an exact match with prefix K, then we would not be able to get the correct prefix match (i.e. prefix F). This is why marker nodes also hold the best matching prefix (i.e. prefix F) so far.)*

the tables. So, when looking for a match in a specific prefix length table, if a match is found, the search proceeds to longer lengths; otherwise it proceeds to shorter lengths.

To perform the binary "search on length" a binary tree and the corresponding length prefix hash-tables are needed. Figure 2.8 is an example of such a structure. The trie in the figure is not needed but it is shown for clarification purposes. The search is performed by traversing the binary tree on the right-side. Depending on the visited node of the binary tree, the corresponding hash-table is checked for a match. Each hash-table for prefix length l stores the prefixes and markers of the level l of the trie.

2.1.7 Sequential Search on Values

The simplest method for address lookup is a *sequential "search on values"*. In this an exhaustive linear search is performed; the incoming address is attempted to be matched with each one of the address prefixes one by one. Every time that there is a match, the longest match is kept and the process continues until all the address prefixes have been checked. Although this scheme is straightforward and independent of the address width, the required time scales linearly with the number of address prefixes, so it is prohibited to be used in the current internet traffic conditions. Non-sequential "search on value" approaches are preferable and are presented in the following sections.

2.1.8 Range Tree

The *range tree* is a typical binary "search on values" approach. Instead of moving on the length dimension, the search is performed by moving on the values dimension. This means that the range tree performs value comparisons to traverse the search space.

The search in a range tree is done in a similar way as the known binary search. In every node, one value is compared against the incoming address. Depending on the outcome of the comparison, we move to the next corresponding node, until a leaf node is reached and get a range match. The range tree divides the search space in two parts after every comparison. So, it is important to choose the optimal values to compare in every step. It must be noted that we no longer try to match address prefixes, but rather find the address range that an incoming address belongs to. This means that the values to be compared at each step must have the same width as the incoming address and the address prefixes have been expanded to this common length.

An example range tree may be seen in Figure 2.9. Note that the range table covers the entire address space. Range trees lead to more balances trees, with smaller depth, but the cost of memory accesses is higher, since more data need to be retrieved per step.

A descendant method of the range tree is the *multiway range tree* [33], where instead of a single comparison per step, more comparisons are performed. In a multiway range tree, internal nodes may have k branches by perfomring k-1 comparisons. An example multi-way range tree is shown in Figure 2.10 and is the equivalent of the single-way range tree of Figure 2.9. It can be seen that the tree depth was further reduced at the expense of higher needed memory bandwidth and resources.

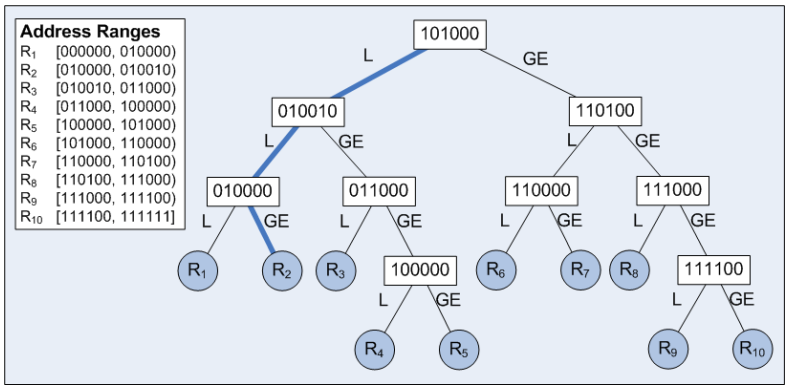


Figure 2.9: A binary range tree for a set of address prefixes translated into range bounds. Range nodes are shown in a darker color. As an example, the path followed to match the incoming address 010000 to range R_2 (prefix B) is shown. The search starts from the root node, where incoming address 010000 is compared to 101000 and is found to be smaller (L). Thus, we visit next its left child, where 010000 is found to be less (L) than 010010. The process continues until the range node R_2 is reached.

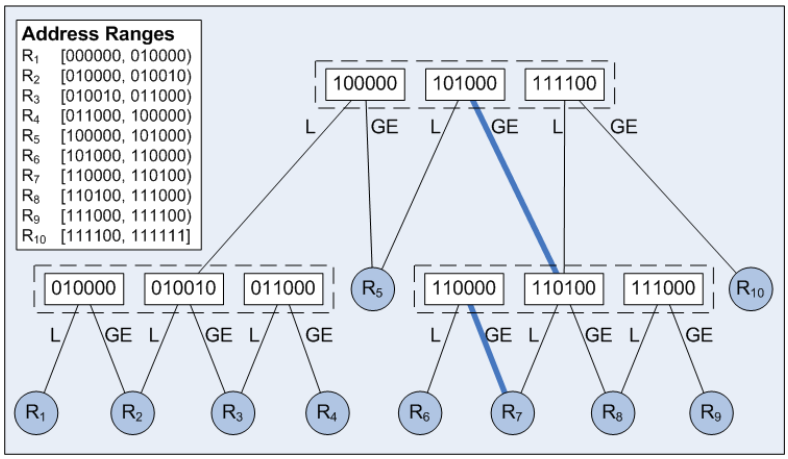


Figure 2.10: A multiway range tree for a set of address prefixes translated into range bounds. Range nodes are shown in a darker color. As an example, the path followed to match the incoming address 110000 to range R_7 (prefix F) is shown. The search starts from the root node, where incoming address 110000 is compared against 100000, 101000 and 111100 and is found to be larger (GE) than 101000 and less than 111100. Thus, we visit next the right node of the second level, where 110000 is found to be egual (GE) to 110000 and less (L) than 110100. So, we visit next node R_7 which happens to be a range node and the search is completed.

2.1.9 TCAMs

Using the hardware element of *Ternary Content Addressable Memories* (TCAMs) is an attractive hardware-based solution for constant-time address lookup. A TCAM, unlike regular memories, is addressable by data. The user supplies a query data and the memory tries to find it and returns the address that the data are stored (if they exist). Furthermore, TCAMs allow a third matching state "X" (don't care), instead of matching

just zeros and ones. This means that a memory entry in TCAMs may hold the binary value of the address prefix to be matched, along with a mask specifying which bits of the memory entry should be compared to the query data.

Such a structure is all that is needed to perform address lookup. The TCAM is set up with the address prefixes/masks, so that a longest prefix match may be done in every clock cycle ($O(1)$ search time). Using TCAMs is like performing an exhaustive sequential "search on values" but in a completely parallel fashion; all the comparisons happen at the same time due to the TCAM structure (see Figure 2.11).

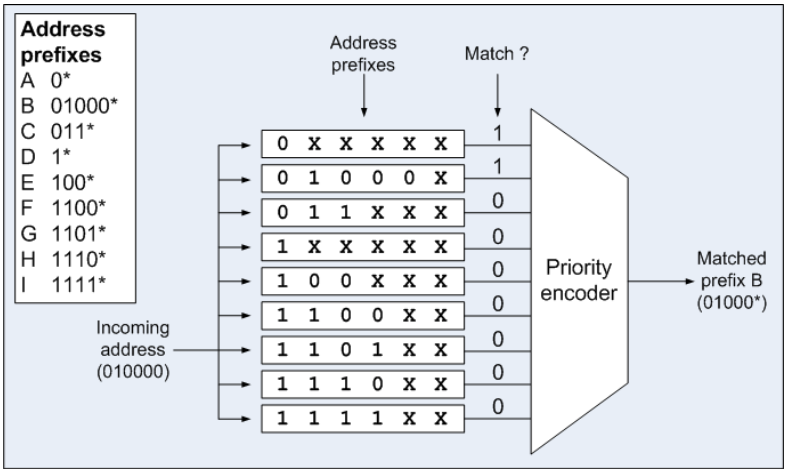


Figure 2.11: The TCAM structure for address lookup for a set of address prefixes. Note the existence of a priority encoder because multiple memory entries may match for a single search key. The TCAM has been properly set up with the address prefixes values and masks. An example lookup for matching the incoming address 010000 is shown.

Although TCAMs offer the best available lookup time, their use is reasonable only for small lookup tables. TCAMs suffer from four deficiencies [31]: (1) high cost per bit compared to other memory technologies, (2) storage inefficiency, (3) high power consumption, (4) limited scalability to long input keys.

In [27], Spitznagel et al. introduced the *extended TCAM* to address the power consumption and storage inefficiency problems of TCAMs by limiting the active regions of the device during a search and by employing a multi-level memory hierarchy. Despite the improvements, using TCAMs is still not a suitable solution for the future large-scale lookup problems.

2.1.10 IPstash

In [17], Kaxiras and Keramidas proposed a memory architecture (the *IPStash*) for address lookup that acts as a TCAM replacement. Their approach offers higher performance and significant power savings compared to the TCAM approach, while allowing for high update rates.

Their proposed architecture is similar to the set-associative caches but is designed to facilitate address lookup (in particular longest prefix matching). IPStash holds a

complete lookup table instead of just a small part of the data set (as caches do). It is based on the observation that address lookup only requires associativity depending on the routing table characteristics. To perform the address lookup in IPStash, first they define the index and tag parts of the incoming addresses. The index is set to a static length (i.e. the 8 most significant bits) and the tag is set to be a variable-number of N following bits. The IPStash lookup process consists of, first, storing the prefixes into a set-associative structure according to their indices and then determine within a set which is the longest prefix match based on the tags (see Figure 2.12).

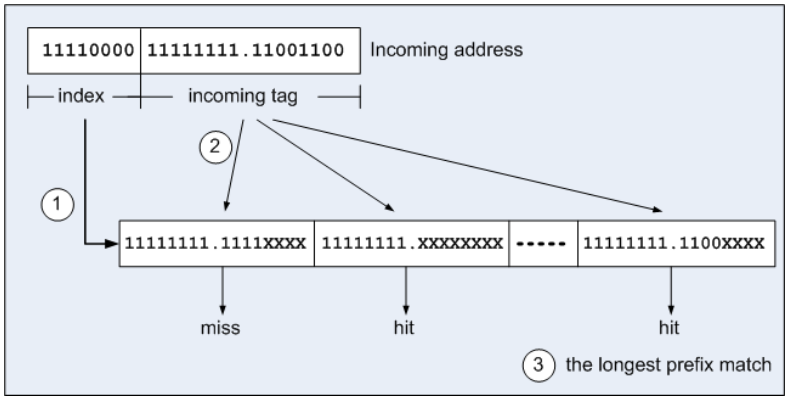


Figure 2.12: The IPStash architecture for address lookup (longest prefix matching) for a set of address prefixes. The index (8 most significant bits) of the incoming address is used to "retrieve" (using an indexing function) a set of candidate matching prefixes out of a set-associative structure. Then each prefix in the set is attempted to be matched to the incoming address. What needs to be matched is the tag of the incoming address (the N bits following the index). The tag may be of variable length and this length is stored in the structure. Since there may be many candidate matches, the longest prefix match is chosen by determining which matched tag is wider.

To benefit from this approach they investigated on the lookup table characteristics in order to increase the index length (to address a larger number of sets) and partition the lookup table into classes (each with its own index). The resulting increase in index numbers and, thus, the requirement of extra hash functions led to the application of skew associativity where different indexing functions are used for each of the set-associative ways.

2.1.11 Bloom Filters

The Bloom filter, conceived by Bloom in 1970 [3], is a probabilistic data structure that is used to test if an element is a member of a set of elements or not. A Bloom filter is actually a bit-vector of m bits, initially all set to 0. Alongside, there are k hash functions defined, each one of them mapping a set element to one of the m bits of the Bloom filter. Before using the Bloom filter, it must be programmed for the given set of elements. To add a set element A, the element A is fed to the k hash functions in order to get k bit-vector positions and set them to 1. To search if an element B is part of the set of elements, the element B is fed to the k hash functions to get k bit-vector positions. If any of the bits in these k positions is 0, then B is not part of the set. Otherwise, if all

bits are 1, then B belongs to the set with a certain probability; B may either be a part of the set or not (false positive). This ambiguity comes from the fact that the bits of the Bloom filter may be set by any of the set elements. The probability of a false positive may be reduced (but not eliminated) by changing (a) the number of entries stored in a filter, (b) the size of the filter, (c) the number of hash functions used to probe the filter. An example of a Bloom filter is shown on Figure 2.13.

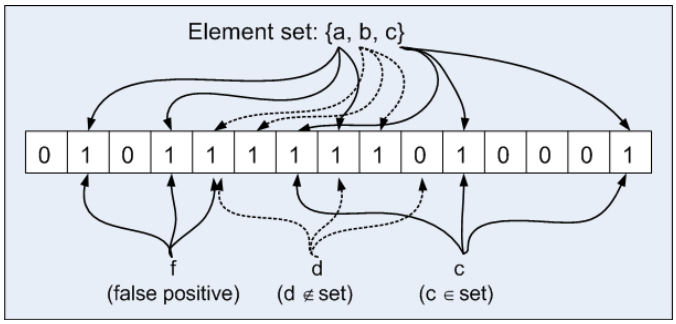


Figure 2.13: An example of the Bloom filter data structure. In this case the Bloom filter is 15-bits wide and there are three hash functions. The original set element that programs the Bloom filter is a, b, c. The 3 hash functions are applied for each element, resulting in the filling with ones of the corresponding positions in the bit-vector. Searching if element d belongs in the set fails, because one of the bits retrieved from the hash functions is 0. Examples of a successful search (for element c) and a false positive search (for element f) are also shown.

Another approach to Bloom filters are the *counting Bloom filters* [4], where m counters are used instead of an m bit-vector. The use of counters allows the deletion of elements from the original element set by decrementing the respective counters. This dynamically changing of the element set was not possible on the original Bloom filters. The trade-off is that a counting Bloom filter will occupy more space compared to the original Bloom filters.

Dharmapurikar et al. in [8] used Bloom filters for address lookup. Their approach consists of sorting the address prefixes by prefix length. For each prefix length they associated a Bloom filter and programmed it. Also, hash-tables were constructed for each prefix length. The search on this approach starts by performing parallel queries to the Bloom filters by querying the respective part of the incoming address. The result is a bit-vector stating the matches for each prefix length (false positives may occur in it). Afterwards, the corresponding length hash-tables are searched, starting from the longest prefix length ones. The search stops when a match is found or when all hash-tables indicated by the bit-vector have been searched. Searching in the hash-tables eliminates the effect of false positives.

Their motivation for using Bloom filters was to avoid the inefficient TCAMs and use modest amounts of SRAMs. The proposed architecture managed to achieve better performance and scalability than TCAM approaches. They researched on system configurations (amount and allocation of memories) to minimize the number of hash-table searches per address lookup. To succeed they introduced the use of (a) *asymmetric Bloom filters* (scaled width Bloom filters depending on the prefix distribution), (b) *direct*

lookup arrays (for smaller hash-tables) and (c) *controlled prefix expansion*[1] (to reduce the number of required filters). Performance may now stay constant for longer address widths or larger lookup tables, as long as memory requirements scale linearly with the number of address prefixes. Using Bloom filters for address lookup achieved on average-case one hash search per lookup or, at worst-case, two hash searches and one array access per lookup.

2.1.12 Tree Bitmap

Eatherton et al. in [10] introduced the *tree bitmap* data structure that resembles a multibit trie and the compression techniques of Lulea [7]. Their main effort was into compressing the address prefixes as much as possible in order to reduce the memory access width of the lookup process.

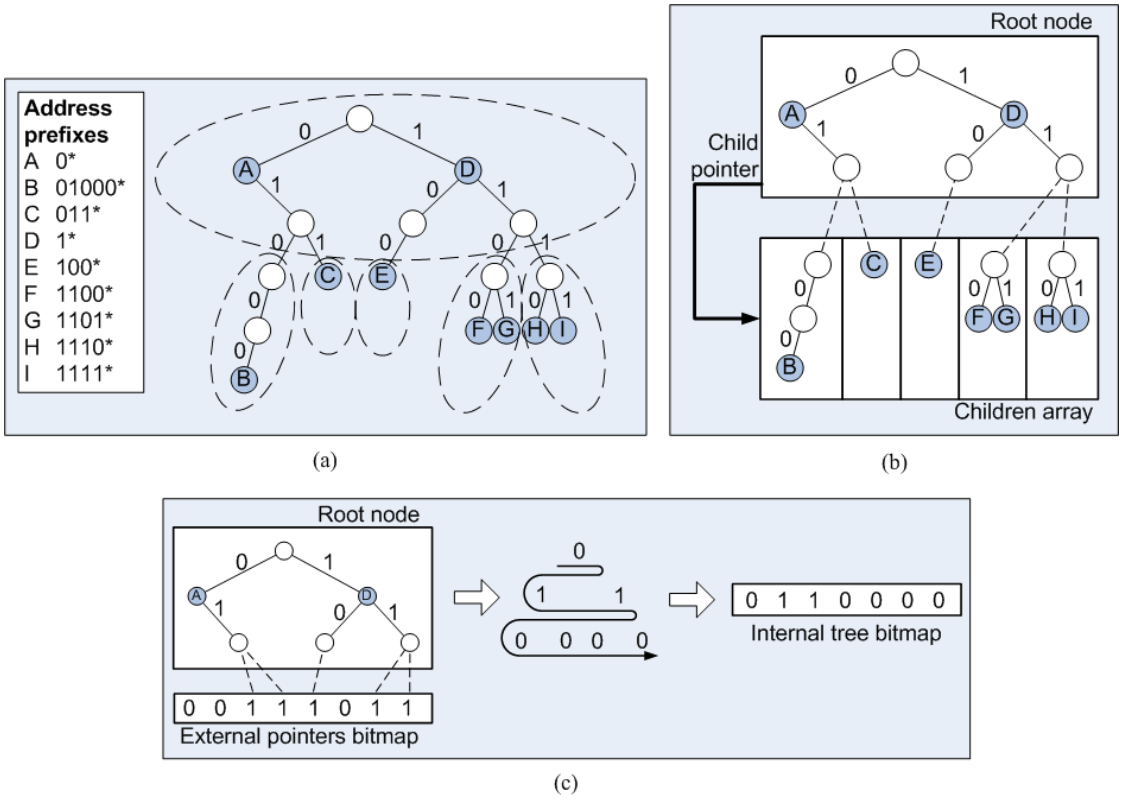


Figure 2.14: The tree bitmap data structure. In (a), the original trie is shown partitioned into 3-stride multibit nodes. Each multibit node holds a position to an array as shown in (b). Note the single pointer to the beginning of the child array. The result array and result pointers are not shown (assume that the dark shaded prefix nodes point to it). The way to retrieve the two bitmaps of a multibit node is shown in (c).

[1]Controlled prefix expansion [28] selects a small number of prefix lengths to be searched. The address prefixes with different length, than the selected ones, are expanded into multiple entries of the next higher selected length.

They used a data structure where they first partition the original trie into multibit nodes. Each multibit node is a k-level tree of the trie. For each multibit node, the necessary search information are encoded as: (a) a pointer to the children multibit nodes, (b) a bitmap for the internal stored prefixes of the multibit node and (c) a bitmap for the external pointers of the multibit node. All child nodes of a multibit node are stored contiguously in the memory, thus a single child pointer is required per node. At every step of the search, a multibit node produces the pointer to the next multibit node to be processed at the next step. When the search terminates, a result pointer points to the result array to determine the action to be taken by the lookup process. An example of a tree bitmap data structure is shown in Figure 2.14.

The tree bitmap data structure may be used with any structure of modern memories by tuning the multibit node strides for the specific memory characteristics. Changing the stride affects the number of bits that represent each multibit node data structure. Eatherton et al. applied a set of optimizations to further reduce the size of multibit nodes, such as: (a) initial array optimization, where an initial array is used like in modular packet classification (see Section 2.1.5), (b) end node optimization, where multibit nodes containing just a prefix node are eliminated and the necessary data are integrated to the parent multibit node, (c) split tree bitmaps, where the two bitmaps of a multibit node are retrieved through separated memory accesses, (d) segmented bitmaps, where a bitmap is split in two and (e) using CAM nodes that occupy the same space with a multibit node that has few internal prefixes.

2.1.13 Pipelines

Assuming that the lookup decision trees are stored in a memory, then performing the decision tree traversal requires multiple memory accesses to match the incoming address. To make the process faster a multiple-stage *pipeline* may be used, where the decision tree is organized in a multitude of memory units. The main issue with pipeline approaches is how to organize the decision tree in the pipeline stages in a balanced way to keep the memory utilization high, while retaining the ability for fast updates.

A lot of research exists on mapping a decision tree to pipeline stages. A simple approach is to match a tree level to a pipeline stage. In [15], Hasan and Vijaykumar describe a more elaborate scheme where a trie is mapped to pipeline stages based on the height of its nodes. They managed to achieve a scalable design that guarantees worst-case performance bounds.

Another approach to pipeline is the use of circular pipelines, where a lookup may be initiated at any stage. Using a circular pipeline decouples the number of pipeline stages from the number of decision tree levels. In [18], Kumar et al. introduced the Circular, Adaptive and Monotonic Pipeline (CAMP) architecture, which is an extension of the circular pipelines. In CAMP, a trie is split into a root sub-trie and multiple leaf sub-tries (see Figure 2.15). The root sub-trie is implemented as a table that directs which leaf sub-trie to visit. Each leaf sub-trie is mapped to start at a different pipeline stage. They developed a mapping algorithm that results into a large number of small memory stages that facilitates a high troughput.

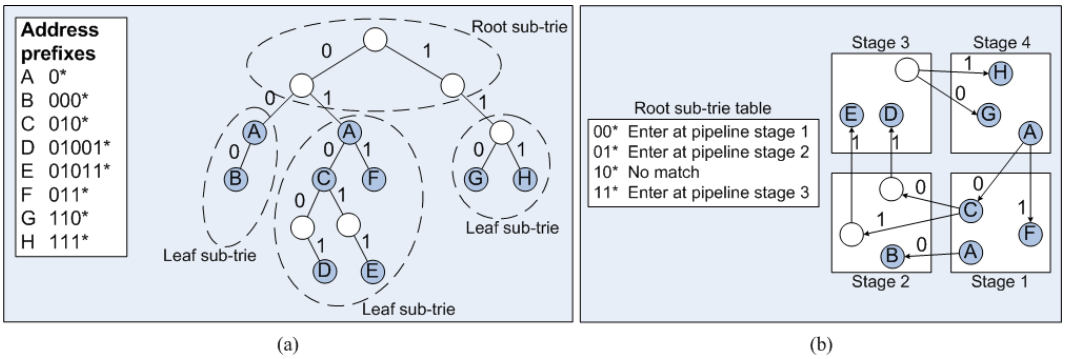


Figure 2.15: The CAMP pipeline architecture for address lookup (longest prefix matching) for a set of address prefixes. In (a), the original trie is depicted along with its partition into the root sub-trie and the leaf sub-tries. There has been a prefix expansion to ensure that all prefixes are longer than 2-bits. In (b), the leaf sub-tries are mapped to the pipeline stages and the pipeline stage to enter is determined by looking in the root sub-trie table.

2.2 The Range Trie

In the previous section, a multitude of current address lookup schemes was presented. These schemes have not managed to keep up with the pace of the internet traffic/speed growth. This pointed out the need for an address lookup method that has (a) low latency, (b) high throughput, (c) low memory requirements and (d) maintain these properties even if the lookup table size and address width are increasing. Furthermore, it would be useful if the lookup structure is fast to construct and update. A method promising to solve these problems, is the *Range Trie* algorithm introduced by Sourdis in [26] and it is presented in this section.

The Range Trie is a new approach for address lookup. It is considered to be between the range tree approaches and the trie approaches; hence the name Range Trie. While tries perform an *exact match in parts of addresses* and range trees perform *comparisons of full addresses*, the Range Trie lies between them by performing *comparisons of parts of addresses*. It may not be categorized neither as a "search on length" approach, nor as a "search on values" approach, but rather as a combination, as it tries to combine the benefits of each concept.

The Range Trie is a tree data structure that is traversed to perform the search, according to the algorithm specifications. The key effort during the construction of a Range Trie is to perform as many comparisons (on parts of addresses) as possible per traversal step and thus utilizing optimally the given memory bandwidth. Also, that way the branching factor of each node increases, resulting into a shorter tree structure.

In the rest of this section, all the details regarding the Range Trie algorithm are presented. Specifically, in Section 2.2.1 the Range Trie is introduced in more details, followed by Section 2.2.2, where the rules that guide the construction of a Range Trie are defined. In Section 2.2.3 the complete Range Trie structure is described, along with the search method to traverse the Range Trie to retrieve the matching range. An automatic way to construct a Range Trie structure using heuristic approaches is outlined in Section

2.2.4. Section 2.2.5 outlines the required modifications of the Range Trie algorithm in order to support also longest prefix matching. Finally, Section 2.2.6 concludes the Range Trie description by summarizing its achievements.

2.2.1 Range Trie description

The Range Trie tackles efficiently the address lookup problem in the context of internet routing, but it may be used in other research fields requiring a form of lookup. The address lookup problem is defined as follows: Given an address space $[0, 2^n)$ and r unique IP addresses/bounds A_i, where $0 < A_i < 2^n - 1$ and $i = 1, 2, ..., r$, that define $r+1$ address ranges R_j $(j = 1, 2, ..., r + 1)$, then an address lookup is to determine the range R_j an incoming address A_{IN} belongs to.

The Range Trie method consists of both a structure and an algorithm to search the structure. In particular, the Range Trie is a tree-like structure, resembling the multiway range tree (see Section 2.1.8). At every step of the multiway range tree, k full comparisons were performed of the incoming address against k values in order to decide which node to visit on the next step. This approach is straightforward at the expense of performing full width comparisons and under-utilizing the available memory bandwidth. On the contrary, the Range Trie exploits the characteristics of the values to be compared per node in order to perform as many comparisons as possible and avoid unnecessary comparisons. This is done by performing comparisons only on selected parts of values, instead of full-width values. Actually, a Range Trie reduces the number of bits to be compared and, given a memory bandwidth, increases the number of comparisons per step. As a result, the depth of the Range Trie gets smaller compared to a range tree with the same available memory bandwidth.

As mentioned, the Range Trie is a tree-like structure and, in every step, one of its nodes is visited to decide which node to visit next until reaching a result (a matching range). A Range Trie node N maps to an address range $[N_a, N_b)$ and divides it into $k+1$ subranges $R_1, ..., R_{k+1}$, where $R_1 = [N_a, A_1)$, ..., $R_i = [A_{i-1}, A_i)$, ..., $R_{k+1} = [A_k, N_b)$ and $A_i \in [N_a, N_b)$, $i \leq k$. The length of the range that node N maps to is defined as $D = N_b - N_a$. A Range Trie node is depicted in Figure 2.16. One final remark is that the union of the nodes ranges in a single tree level equals to the entire address space and the union of the children nodes ranges equals to the parent's address range.

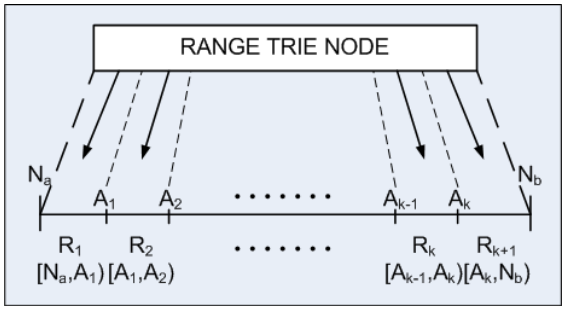


Figure 2.16: A Range Trie node mapping to an address range $[N_a, N_b)$ and dividing it into $k+1$ subranges $R_1, ..., R_{k+1}$.

Before presenting an example of a Range Trie and the way to traverse it, it is impor-

tant to understand the fundamental concepts and rules (described in the next section) that differentiate the Range Trie from the existing address lookup solutions.

2.2.2 Range Trie fundamental concepts and rules

The main idea behind the Range Trie algorithm is that in every step a plurality of comparisons is performed on *selected parts of comparison values*. The effort is on minimizing the comparison widths and, thus, performing as many comparisons as possible per node, for a given memory bandwidth. That way the branching factor increases and the Range Trie depth decreases. In this section, the Range Trie fundamental concepts and rules will be discussed, that help into achieving the goal of minimizing the comparison widths.

The development of the Range Trie was based on the following general observations:

- Nodes closer to the root compare addresses that are sparser in the address space. Thus, there is no need to compare their suffixes.
- Nodes closer to the leafs compare addresses that are denser in the address space. Thus, their prefixes may be shared or omitted.
- Other nodes that may need to compare addresses that are sparser or denser in the address space can compare the respective part of the addresses that results in the best tree balance.

Performing as many comparisons as possible is achieved by exploiting the characteristics of the values to be compared per step. Thus, the following 5 rules were established to *minimize the comparison widths*:

1. **Rule 1**: Omit the common prefix of the node borders.
2. **Rule 2**: Share addresses' common prefix.
3. **Rule 3**: Share addresses' common suffix.
4. **Rule 4**: Omit address suffix of value '0'.
5. **Rule 5**: Align addresses.

These rules guide us to compare *only parts of addresses* that can have *variable width*. Unnecessary or common information is omitted or reduced. Rules 1-4 may be applied independently as they do not interfere with each other. Special care must be taken for combining Rule 5 with others. Applying the rules may be seen as an encoding process of the comparison values. At the same time, the correctness of the lookup procedure must be ensured.

In the rest of this section, each one of the Range Trie rules will be explained. For proof of their correctness you may see [26].

2.2.2.1 Rule 1: Omit the common prefix of the node borders

The first rule suggests that if there is a common prefix of length L at the node borders N_a and N_b of a node N that maps to range $[N_a, N_b)$, then this common prefix (the L most significant bits) can be omitted from all the comparisons included in node N.

An example application of Rule 1 may be seen in Figure 2.17. In the top of this figure, there is a node mapping to address range [0xFFFF0000, 0xFFFFFFFF). If an incoming address A_{IN} reaches this node, then we know for sure that it belongs in the address range of the node. Even more, the node borders share a common prefix of 16 bits (0xFFFF) that will be also common to the incoming address and the rest node bounds. So, this common prefix may be omitted and instead compare the rest bits of the incoming address to the rest bits of the node bounds. The result of applying Rule 1 in the initial node is seen in the bottom of the figure. It may be seen that the ranges are identical, despite the omission of the node common prefix.

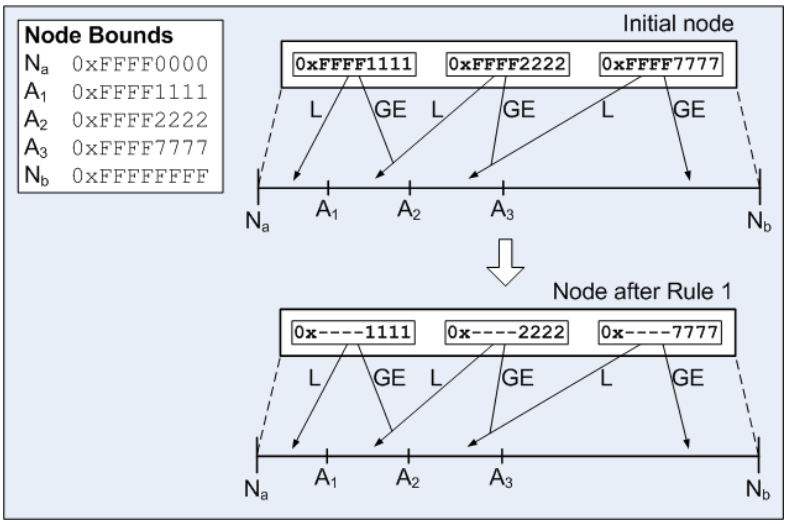


Figure 2.17: An example application of Rule 1. The top node is converted into the equivalent bottom node after removing the comparison of the node borders' common prefix. The dashes represent the non-compared bits.

2.2.2.2 Rule 2: Share addresses' common prefix

The second rule suggests that if there is a common prefix (CP) of length L at all the node bounds A_i of a node N that maps to address range $[N_a, N_b)$, then this common prefix (the L most significant bits) can be shared among the multiple node comparisons and get compared separately. In particular, if the incoming address A_{IN} prefix of length L is less than CP, then $A_{IN} \in [N_a, A_1)$. If it is greater than CP, then $A_{IN} \in [A_k, N_b)$. If it is equal, then the rest bits of A_{IN} are compared against the rest bits of the A_i to determine the matching range.

An example application of Rule 2 may be seen in Figure 2.18. In the top of this figure, there is a node that compares three addresses (A_1, A_2, A_3) to match the 4 ranges in

[0xDDDD0000, 0xFFFF1111). These three addresses share a 16-bit wide common prefix (0xEEEE). Instead of performing 3 full-width comparisons, it is possible to perform one separate 16-bit comparison on the common prefix and share its result with the 3 comparisons on the rest bits of the 3 addresses, according to Rule 2. The resulting node may be seen on the bottom of the figure, along with which range to match depending on the outcome of the comparisons.

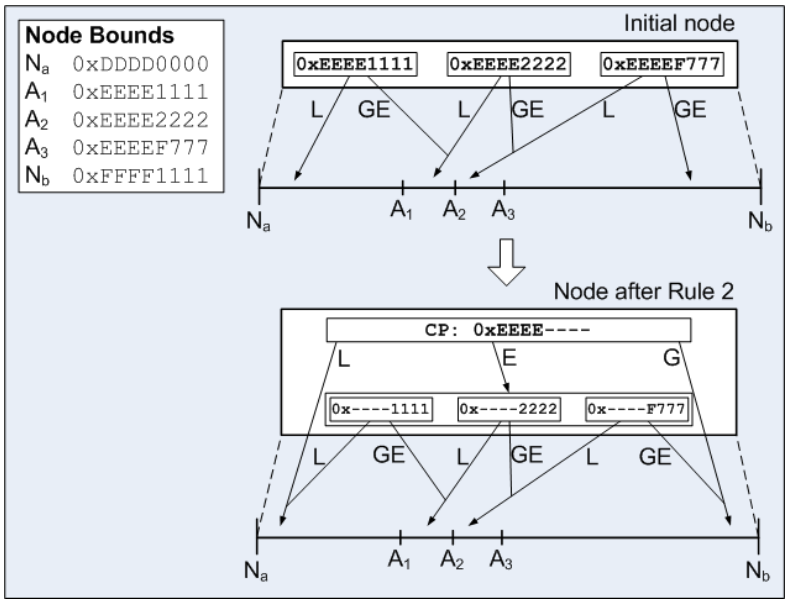


Figure 2.18: An example application of Rule 2. The top node is converted into the equivalent bottom node after sharing the 16-bit common prefix. The dashes represent the non-compared bits.

2.2.2.3 Rule 3: Share addresses' common suffix

The third rule suggests that if there is a common suffix (CS) of length L at all the node bounds A_i of a node N that maps to address range $[N_a, N_b)$, then this common suffix (the L least significant bits) can be shared among the multiple node comparisons and get compared separately. In particular, if the incoming address A_{IN} prefix of length $W - L$ is less than the prefix of length $W - L$ of A_i, then $A_{IN} \in [A_{i-1}, A_i)$. If it is greater, then $A_{IN} \in [A_i, A_{i+1})$. If it is equal, then then the match result depends on the comparison of the suffix of length L of A_{IN} against the CS. If it is less than the CS, then $A_{IN} \in [A_{i-1}, A_i)$. If it is greater or equal, then$A_{IN} \in [A_i, A_{i+1})$. Note that $N_a \equiv A_0$ and $N_b \equiv A_{k+1}$.

An example application of Rule 3 may be seen in Figure 2.19. In the top of this figure, there is a node that compares two addresses (A_1, A_2) to match the 3 ranges in [0xAAAA2222, 0xDDDD3333). These two addresses share a 16-bit wide common suffix (0x1111). Instead of performing 2 full-width comparisons, it is possible to perform one separate 16-bit comparison on the common suffix and share its result with the 2 comparisons on the rest bits of the 2 addresses, according to Rule 3. The resulting node

may be seen on the bottom of the figure, along with which range to match depending on the outcome of the comparisons.

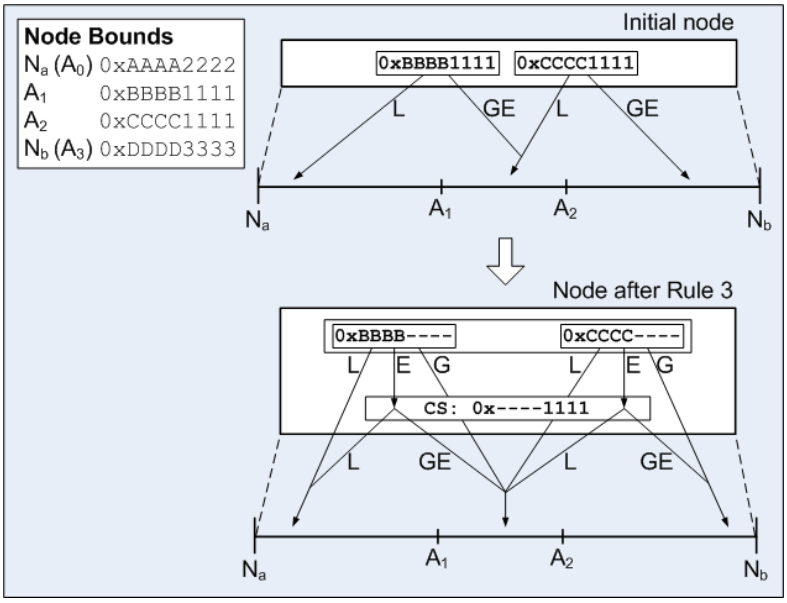


Figure 2.19: An example application of Rule 3. The top node is converted into the equivalent bottom node after sharing the 16-bit common suffix. The dashes represent the non-compared bits.

2.2.2.4 Rule 4: Omit address suffix of value '0'

The fourth rule states that if a node bound A_i of a node N that maps to address range $[N_a, N_b)$ has a suffix of length L that is zero-valued, then this zero suffix may be omitted from the respective comparison in node N. The match result is then obtained by performing a comparison of the rest bits of the truncated A_i with the respective bits of the incoming address A_{IN}.

What this rule claims is that a zero-value suffix does not need to be compared, since we always know the outcome of the comparison (comparing to zero will always yield greater or equal). An example application of Rule 4 may be seen in Figure 2.20. In the top of this figure, there is a node mapping to address range [0xAAAA1111, 0xDDDD2222). Both addresses A_1 and A_2 have a zero suffix of 16-bits and 24-bits respectively. Since the outcome of comparing the zero suffix to the respective bits of the incoming address is always GE, then there is no reason into comparing these bits, resulting into the node seen in the bottom of the figure. It may be seen that the ranges are identical, despite the omission of the zero suffix.

2.2.2.5 Rule 5: Align addresses

This rule states that the lookup of an incoming address A_{IN} in a node N that maps to address range $[N_a, N_b)$ and compares node bounds A_i is equivalent to the lookup of the

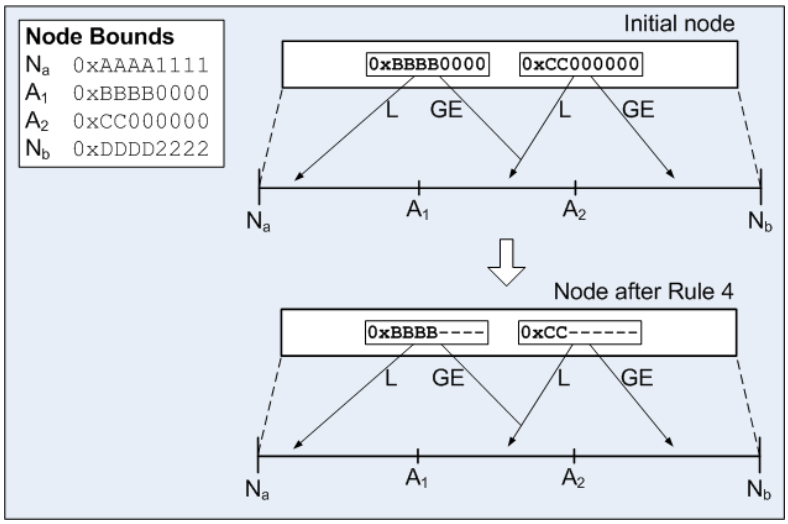


Figure 2.20: An example application of Rule 4. The top node is converted into the equivalent bottom node after omitting the suffixes of value '0'. The dashes represent the non-compared bits.

address $A_{IN} - N_a$ in a node N' that maps to address range $[0, N_b - N_a)$ and compares bounds $A'_i \equiv A_i - N_a$.

The purpose of this rule is to align the addresses in a node in such a way that a wide zero-valued common prefix appears and then perform narrower comparisons. It can be calculated that the width of the comparisons will be $L = log_2(N_a - N_b)$ in the worst case. Thus the useful bits of the incoming address that we need to subtract and then compare will also be L. This means that the subtraction $A_{IN} - N_a$ suffices to be calculated for the L least significant bits.

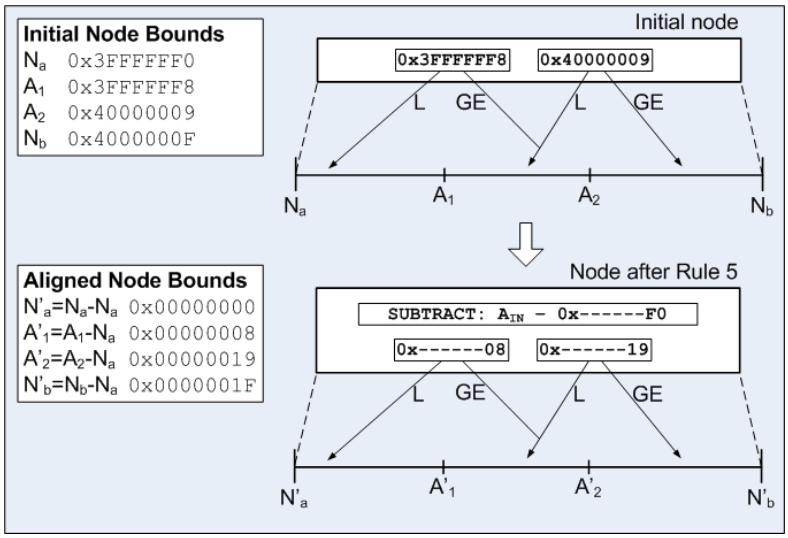


Figure 2.21: An example application of Rule 5. The top node is converted into the equivalent bottom node after address alignment. The dashes represent the non-compared bits.

Although Rules 1-4 can be applied independently to one another, Rule 5 is a special case. It can be combined with Rule 1, as it maximizes the common node prefix, but it must be applied before Rule 2. Regarding Rules 3-4, Rule 5 may be combined independently with them.

An example application of Rule 5 may be seen in Figure 2.21. In the top of this figure, there is a node mapping to address range [0x3FFFFFF0, 0x4000000F). These node borders have just one bit common prefix. By aligning the addresses of this node by subtracting 0x3FFFFFF0 we get the node on the bottom side of the figure. This aligned node now has 24-bits node borders common prefix and only 2 8-bit comparisons need to be performed. Note that only an 8-bit subtraction on the incoming address suffices, since the 24 most significant bits of the subtraction result are known to be zero.

2.2.3 Range Trie structure and algorithm

In the previous section it was described how to apply the five Range Trie rules to minimize the comparison widths and thus optimally utilize the given memory bandwidth. In this section, the method to search in a Range Trie structure will be presented.

The Range Trie, as already mentioned, is a tree structure that spans in a number of levels and is used to match an incoming address A_{IN} to a range R_i. At each level there is a number of nodes, except the top level where is just one node (the root node). This structure may be used to perform address lookup under the guidance of an algorithm that directs the decisions made during the lookup process. An example Range Trie is shown in Figure 2.22.

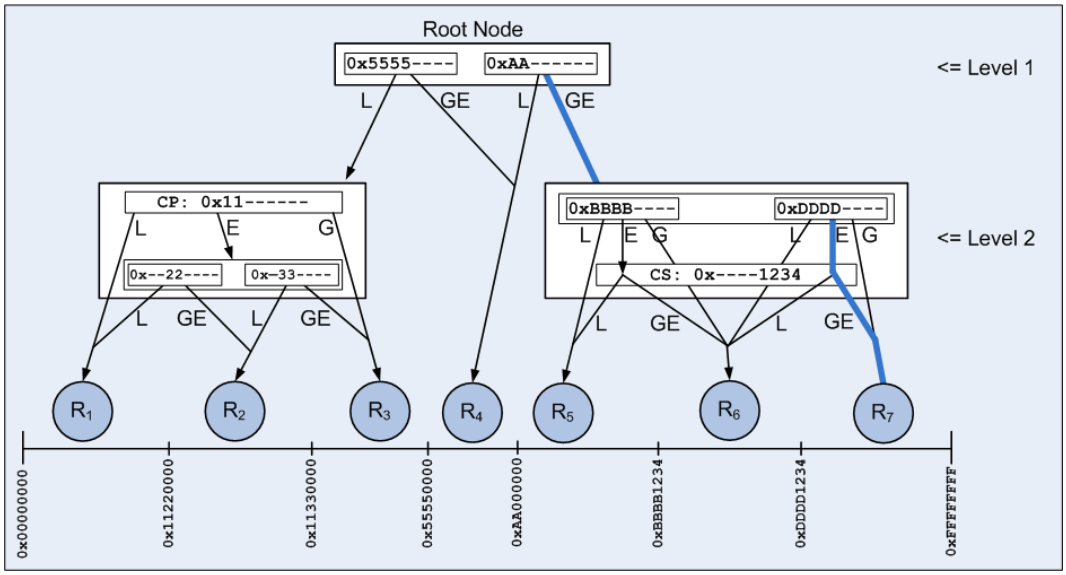


Figure 2.22: An example of a Range Trie structure for a given set of 7 address ranges R_i. There are two levels in total. Nodes contain the necessary information for choosing the next node to visit, until reaching a leaf node. Leaf nodes are drawn in a darker shade. An example traversal for matching incoming address 0xDDDD4444 to address range R_7 is shown.

There are two types of nodes in a Range Trie: the leaf nodes (indicating a range

match) and the internal nodes. Internal nodes contain information about (a) the number of comparisons to be performed, (b) which bits of the incoming address to compare, (c) the values to be compared with and (d) the branches to the nodes of the next level. If it is needed, each node might also suggest (a) to share a common prefix comparison, (b) to share a common suffix comparison and (c) perform an address alignment. In any case, the node also offers the necessary information, like the common prefix position and value, the common suffix position and value, the alignment value to subtract from A_{IN}. It must be noted that the nodes are a result of following the 5 rules defined in Section 2.2.2.

Assume that we need to lookup the incoming address A_{IN}. The search starts by first visiting the root node. The information in the root node are used to perform the necessary comparisons. If needed there could also be a prefix comparison, a suffix comparison and a subtraction on the incoming address. The outcome of the comparisons dictates which will be the next node (of the second level) to visit. This decision process is then repeated until a leaf node is reached, which suggest that A_{IN} has been matched to the range R_i that it belongs to.

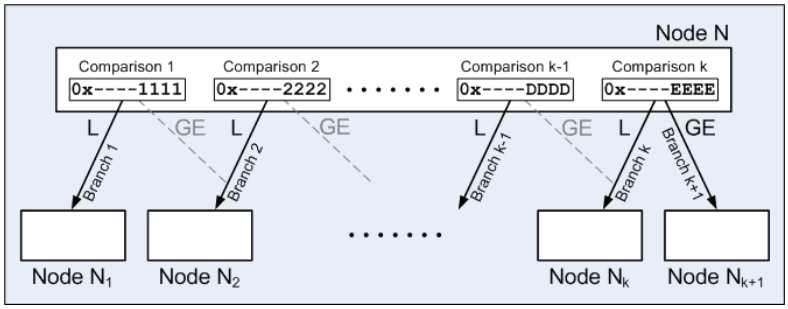


Figure 2.23: A Range Trie node N and its children. Node N stores the necessary information for choosing the next node to visit. The depicted node does not share common prefix/suffix or align addresses by subtraction. The children nodes N_i may be either leaf nodes or internal nodes. Note that all the GE branches, except the last one are not used for branching to a next node N_i, since their result is identical to the L branch of the next comparison. The dashes represent the non-compared bits.

As already mentioned, choosing the next node to visit is based on the outcome of the performed comparisons in the current node. Assume that we are currently in node N (see Figure 2.23) that performs k comparisons (not counting shared prefix/suffix comparisons). This means that there would be $k + 1$ outgoing branches to next level nodes N_i, where $1 \leq i \leq k + 1$. To decide which N_i node to visit next, the following steps must be performed (based on the node information):

1. If an address alignment must be performed, then the subtraction value is subtracted from the indicated bits of A_{IN}.

2. Then the k comparisons are performed on the indicated bits of A_{IN} against the k comparison values. Each comparison may result in greater or equal (GE) or less (L).

- If the result of comparison 1 is L, we set $N_i = N_1$ as the node to visit next.
- If the result of comparison k is GE, we set $N_i = N_{k+1}$ as the node to visit next.
- Otherwise, there is a comparison $j \neq k$ that results in GE and a comparison $j + 1$ that results in L. Thus, we set $N_i = N_{j+1}$ as the node to visit next.

3. If there is a common prefix sharing, then the corresponding prefix bits of A_{IN} are compared against the common prefix.
   - If the result of the comparison is L, we set N_1 as the node to visit next.
   - If the result of the comparison is G, we set N_{k+1} as the node to visit next.
   - If the result of the comparison is E, we set as the next node to visit the node N_i that was set in step 2.
4. If there is a common suffix sharing, then the corresponding suffix bits of A_{IN} are compared against the common suffix.
   - If the result of the comparison is GE, we set as the next node to visit the node N_i that was set in step 2.
   - If the result of the comparison is L, we set as the next node to visit the previous node N_{i-1} than the one (N_i) that was set in step 2. If in step 2, N_1 was set as the next node, then N_1 is set as the next node to visit.

Although the steps were presented sequentially, the respective comparisons may be performed in parallel and afterwards decide the next node to visit, based on the outcome of the comparisons. This process repeats in every visited node of the Range Trie, until a leaf node is reached that reports the range that A_{IN} belongs to.

2.2.4 Construction of a Range Trie

In the previous section, the Range Trie structure was detailed, along with the process to traverse it. In this section, an automatic way to generate the Range Trie structure for a given set of address ranges is outlined.

Given a set of k address bounds A_i that define $k+1$ address ranges R_j, a Range Trie must be constructed following the 5 Range Trie rules (see Section 2.2.2) and having the defined structure (see Section 2.2.3). The construction process should take advantage of the inherit characteristics of the Range Trie and maximally exploit the 5 Range Trie rules. It is also needed that the generation of a Range Trie to be fast enough. There are two targets during the Range Trie generation: (a) minimize the number of bits to compare per comparison (in order to fully utilize the given memory bandwidth, maximize the numbers of outgoing branches per node and create low depth Range Tries) and (b) create a balanced Range Trie (where nodes branch to subtrees of equal or similar depth).

Sourdis et al. in [26] and [25] proposed the use of heuristic methods for generating Range Tries. Heuristic methods are preferred because they are faster to complete and give near-optimal results, rather than trying to find the optimal solution. Two different recursive approaches were followed: *top-down* (the root node is created first, then the

children nodes, until reaching the leaf nodes) and *bottom-up* (the leaf nodes are created first, then their parent nodes, until reaching the root node). For each heuristic approach, two variations were created: one allowing comparisons of only a *single length* per node and one allowing comparisons of *variable lengths*. The resulting four heuristic methods are outlined below. A more detailed description of these may be found in [26], [25] and [6].

- *Top-Down with Single Length Comparisons (TD-SLC)*:
  1. Apply Range Trie rules 5, 1, 4; address alignment, omit node borders' common prefix, omit zero-valued address suffix.
  2. Determine the comparison width that maximizes the number of branches.
  3. Assume that all addresses in the set are to be compared according to the determined comparison width. Omit address suffixes that exceed the comparison width by assuming that they are equal to zero.
  4. Create the defined groups.
  5. Merge adjacent groups until the number of needed comparisons is reduced to the number of available comparison resources. Apply Range Trie rules 2 and 3 to share common address prefixes and suffixes. The resulting groups are the node branches and their borders are the comparisons to perform.
  6. Repeat Steps 1-5 recursively for the created children nodes.
  7. Terminate when each group contains only a basic address range R_i.
- *Top-Down with Variable Length Comparisons (TD-VLC)*:
  - This is same as TC-SLC, except Step 5 that is modified as follows: Merge adjacent small groups and split large groups. Splitting is achieved by adding an extra comparison of longer length. The number and widths of the available comparison resources should be considered.
- *Bottom-Up with Single Length Comparisons (BU-SLC)*:
  1. Apply the Range Trie rules and select the first b addresses $A_i > G_d$ (i.e., A_i, A_{i+1}, ..., A_b) that can be compared at one node based on the single-length available comparison resources. G_d is initially 0.
  2. Set as the selected groups' upper bound (G_u) any point in the address space, where $G_u \in (A_t, A_b]$ and $t/b = C$ (C is a user defined constant), such that G_u has the longest zero-valued suffix. The resulting group that maps to the node is the $[G_d, G_u)$.
  3. Repeat the above, starting from the upper bound of the previous group ($G_d^{new} = G_u^{prev}$), until all addresses A_i are grouped.
  4. Repeat steps 1-3 recursively using as a new set of A_i addresses the bounds G_i of the previous level.
  5. Terminate when all addresses are processed in a single iteration (root node reached).

- *Bottom-Up with Variable Length Comparisons (BU-VLC)*:
  - This is same as BU-SLC, except Step 1 where the available comparison resources may now be of variable lengths.

Out of these four heuristic methods, the bottom up approaches were found to provide better results. The constructed Range Trie is balanced with a small depth. The drawback of using bottom-up heuristics is that the execution time increases compared to the top-down ones.

2.2.5 A Range Trie supporting longest prefix matching

As it have been mentioned, the address lookup problem may be approached in two equivalent ways; as a range lookup problem or as a longest prefix matching problem. Up until now, the presented Range Trie solution approached it as a range lookup problem. In this section, the necessary modifications to perform longest prefix matching using a Range Trie will be presented.

To support longest prefix matching in a Range Trie means that more information need to be stored in the Range Trie nodes. Since a Range Trie has the same tree structure as a range tree of unlimited memory bandwidth and number of branches per node, the range tree approach of [29] and [22] may be followed for supporting longest prefix matching in a Range Trie.

The idea is that the Range Trie should also store the prefixes in the internal Range Trie nodes, rather than only the leaf nodes (where a range is matched). These prefixes will be actually stored in an external array, rather than on the nodes, to reduce the memory requirements. Modifying the technique of [29] the following must be stored in the Range Trie nodes:

- A pointer to a prefix, along with the prefix length, must be stored at every Range Trie node that maps to a range that is part of the prefix, while the parent node's range is not part of the prefix.
- For every address compared in a Range Trie node, a counter must be stored holding the number of prefixes having an endpoint on that address.

Supporting longest prefix matching in a Range Trie offers also the fundamentals for an efficient Range Trie updating scheme. By manipulating the counters, the pointers and the addresses compared in a Range Trie node, it is possible to insert or delete addresses that define new ranges. This will not be explained further, since the incremental updating of a Range Trie was out of scope of this thesis.

2.2.6 Range Trie achievements

Through Sections 2.2.1-2.2.4 all the details of the Range Trie method were presented, starting from the basic concepts behind the Range Trie, until a full description of the Range Trie structure, lookup process and construction process.

All of these pointed out that the Range Trie satisfies the properties that motivated its development:

- *Low latency*: The latency of a Range Trie depends on the depth of the tree structure. The Range Trie requires a low depth tree structure due to the following reasons:
  - The number of bits to compare per comparison is minimized by sharing common prefix/suffix comparison results, by omitting unnecessary comparisons and by aligning addresses.
  - The number of comparisons per node is maximized resulting in a high branching factor per node.
- *High throughput*: The operations to perform per search step are simplified and may be done in a fast manner.
- *Low memory requirements*: A limited amount of memory is required to store a Range Trie. Also, the given memory bandwidth is fully utilized due to the minimization of the number of bits to compare per comparison and due to the good balance of the Range Trie.
- *Better scalability*: The latency and memory requirements are scaling well, both in terms of the address width and the number of address ranges, due to the following reasons:
  - The property of comparing parts of addresses allows the good scalability in terms of the address width.
  - The property of the good balance of a Range Trie allows the good scalability in terms of the number of address ranges.

The characteristics of the Range Trie that resulted in these properties will be exploited during the hardware design and implementation of the Range Trie. In the chapter to follow, the hardware design of the Range Trie is explained in detail and all the design steps are discussed for obtaining a synthesizable and validated design.

2.3 Summary

In this chapter, the background material that is needed for the rest of this thesis was presented. After discussing on a variety of related designs and algorithms for address lookup in Section 2.1, the Range Trie address lookup method, which is the main focus of this thesis, was presented in Section 2.2.

In Section 2.1, a representative set of related designs and algorithms for address lookup that exists in the literature was presented. The presented methods came from the internet routing and the packet classification research domains. They were a mixture of algorithmic approaches to address lookup and hardware-targeting address lookup designs. The algorithmic approaches were classified based on the dimension of the performed search ("search on length" or "search on values") and on the type of the search traversal (sequential or binary). First the sequential "search on length" approaches were

presented that are based on a trie structure (Binary Trie, Path Compressed Tries, Multi-bit Tries, Level-Compressed Tries and HiCuts), followed by a binary "search on length" approach. Afterwards, the basics of the sequential "search on values" approach were presented, followed by the binary "search on values" approaches that are based on a range tree structure (Range Tree, Multiway Range Tree). Finally, a set of hardware-targeting solutions was presented (TCAMs, IPStash, Bloom Filters, Tree Bitmaps, pipelines).

Because of the internet traffic/speed growth, the previously presented methods have started lagging behind. In Section 2.2, the novel address lookup approach of Range Tries (introduced by Sourdis in [26]) was presented in detail. The Range Trie is between the "search on length" and "search on values" approaches and delivers a method with (a) low latency, (b) high throughput, (c) low memory requirements and (d) good scalability in terms of address width and lookup table size. The Range Trie is a specific tree structure with a multitude of nodes per level, along with a specific algorithm to traverse the tree structure. The presented Range Trie structure performs range matching and it can also be extended to support longest prefix matching. In each node, comparisons are performed on parts of addresses based on the 5 Range Trie rules that formulate the fundamental concepts behind the Range Trie development. Based on the 5 rules, the parts of addresses to compare are minimized by sharing common prefix/suffix comparisons, by omitting unnecessary comparisons and by aligning the addresses to be compared. Minimizing the parts of addresses to compare, results in a higher utilization of the given memory bandwidth, in an increase of the branches per node and in a decrease of the Range Trie depth. Generating a Range Trie structure is done in an automated way based on heuristic methods that exploit the 5 Range Trie rules.

3 Design and Implementation

In this chapter, the hardware design of the Range Trie method will be presented. Based on the description of the Range Trie (see Section 2.2), all the design steps will be discussed for obtaining a complete Range Trie design. The design effort was to exploit the Range Trie method inherit characteristics into building a fast, efficient and scalable design.

The Range Trie method (see Figure 3.1) is an iterative method to traverse the Range Trie structure and match the range that the incoming address A_{IN} belongs to. According to the method, in every iteration a node is visited, the necessary comparisons on parts of addresses are performed and the next node to visit is decided. This process ends when a leaf node is reached and the matching range is reported. Since the Range Trie method is based on bit-level manipulation of addresses (such as selecting parts of addresses, performing comparisons, selecting a branch to take, etc.), it is efficient to design it in hardware without excluding a potential software-based Range Trie implementation that could still benefit from the Range Trie improvements.

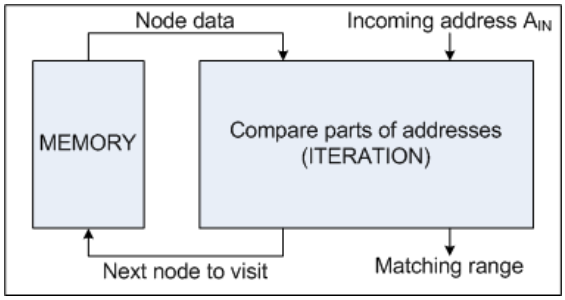


Figure 3.1: An abstract block diagram of the Range Trie method. The details to design the Range Trie hardware are the focus of this chapter.

Transforming the Range Trie method into a hardware design means that each of the method's steps must be carried out by an equivalent hardware design. It is evident that the hardware design will consist of a *memory structure* (where the Range Trie node data are stored that direct the performed comparisons) and a number of *iteration functional units* (where the comparisons are performed and the next node to visit is decided) organized in a pipeline fashion.

Specifications of the design: In the rest of this chapter, the complete Range Trie design will be presented that is parameterizable in the following attributes:

- Memory bandwidth: The allowed memory bandwidth (BW) may be 256, 512 or

1024 bits wide.

- Address width: The incoming address width (W) may be 32 (IPv4), 64 or 128 (IPv6) bits.
- Number of Range Trie iterations (levels): This number indicates how many iterations are needed in the worst-case for a Range Trie to match a range, according to the given Range Trie structure.
- Memory depth per Range Trie iteration (level): This set of numbers indicates the memory entries that shall be stored in the memory structure per Range Trie iteration, according to the given Range Trie structure.
- Output width: The output of the Range Trie will be the matching range and not the action to be performed. In other words, the Range Trie design will output the address that the matching range resides in an *action array*. For generalization purposes, we opted not to include the action array in the Range Trie design, since its size requirements are unknown and depend on the address lookup requirements of the system that the Range Trie will be used in. So, it was safe to require that the Range Trie just outputs the matching range number. The output width depends on the size of the lookup table (number of matching ranges).

All the details that lead to a complete Range Trie design will be presented in this chapter. Section 3.1 elaborates on the design of the Range Trie iteration unit. After presenting the limitations that were posed on the general Range Trie method for an efficient hardware design, the necessary components to build all possible instances of Range Trie iteration units are presented. At the same time, the control signals (according to the node information) that drive the correct operation of an iteration will be defined. Afterwards, in Section 3.2, the chosen memory structure will be presented. The way that the node information must be stored in the memories and the way that the memory is organized/addressed will be presented, along with the details on migrating a Range Trie structure into the memory structure. Section 3.3 combines the memory structure with the iteration units to form the complete Range Trie design in a pipeline fashion. Alongside, the top-level Range Trie module and its usage will be presented. Finally, this chapter concludes with its summary in Section 3.4.

3.1 Range Trie iteration

In Section 2.2, the Range Trie method was presented. In general, it may be defined as a three step iterative method, where given a Range Trie structure, (a) we visit a node, (b) perform the necessary comparisons and (c) decide which node to visit next. The basic block of the Range Trie design is the *iteration functional unit*, where all the necessary computations and decisions are made (steps b and c) given a current node N. In this section, all the details for the Range Trie iteration unit hardware design will be discussed. It must be noted that the design effort was to minimize the required time to perform the computation. This will be evident throughout all the design choices.

When a Range Trie node N is visited, the following actions are performed during the iteration (see Figure 3.2), according to the information stored in node N. The node N information are retrieved from the adjacent memory structure, as described later in Sections 3.2 and 3.3.

1. The incoming address A_{IN} is prepared for comparison: The parts of A_{IN} to be compared are selected, according to the node information. If needed, a subtraction is performed for alignment purposes.

2. The comparison of the parts of A_{IN} is performed against the values defined by the node N information.

3. The results of the comparisons are interpreted.

4. Based on the comparison results and (possible) shared prefix/suffix comparison results, the branch to be taken is decided in order to visit the correct node in the next iteration.

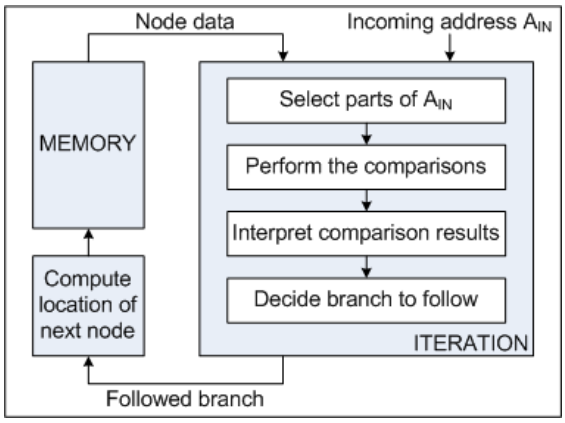


Figure 3.2: An abstract block diagram of the Range Trie method focusing on the iteration steps.

Limitations on the Range Trie iteration for an efficient hardware design: The Range Trie algorithm for address lookup, presented in Section 2.2, is a general algorithm to be used in any possible configuration. I.e. any part of an incoming address may be compared, or any comparison width may be used, or any subtraction width for alignment may be used, etc. The hardware design could be as general as the Range Trie method in the cost of extra needed hardware, higher memory requirements and a higher clock cycle. If this was the case, then designing the Range Trie on hardware would not yield any significant performance benefits. This led to posing some valid limitations on the Range Trie method to ensure the efficiency of the hardware design. Actually, the Range Trie method was developed by having the hardware design in mind, so it is not a stretch to limit the Range Trie in such a way. The work in [25] and [6] assumed the Range Trie

limitations for hardware purposes, while constructing a Range Trie, and it was found out that they were not hampering the Range Trie benefits.

For the purposes of an efficient hardware design the following attributes and limitations of the Range Trie were set as follows:

- Memory bandwidth (BW): The Range Trie is designed for available memory bandwidths of 256, 512 and 1024 bits.
- Address width (W): The Range Trie is designed for possible address widths of 32 (IPv4), 64 and 128 (IPv6) bits.
- Number of available comparators: In every iteration of the Range Trie method a number of comparisons must be performed using a number of comparators. Assuming that the used comparators are W-bits wide, we limited the number of comparators used per iteration to $k = BW/W$.
- Comparison widths: For address width W, the possible comparison widths that may be used are 8, $2*8$, $4*8$, ..., W. As it was mentioned, the Range Trie design uses $k = BW/W$ comparators per step. Each one of these is W bits wide and may be configured to perform s comparisons of variable width $[w_1, w_2, ..., w_s]$. The sum of the comparison widths w_i to be performed should be at-most W. The comparison widths w_i may be 8, $2*8$, $4*8$, ..., W. The allowed combinations of comparison widths are those that the sum of two consecutive comparison widths $w_i + w_{i+1}$, where i is odd, is one of the available comparison widths (8, $2*8$, $4*8$, ..., W). Comparator k is always configured to perform W bits wide comparisons.
- Position of address parts to compare: The parts of addresses to compare should be equal to the possible comparison widths (8, $2*8$, $4*8$, ..., W) and they must be accessible by shifting the incoming address by a number of 2-bits shifts.
- Subtraction width for address alignment: The subtractor for performing the address alignment was limited to be $W/4$ bits wide.

The heuristic construction methods in [25] and [6] were tailored to these hardware limitations and provided with a Range Trie structure that is compatible with the hardware design.

Brief overview of the Range Trie iteration hardware design: Before explaining the iteration design it is useful to present a brief overview of the designed hardware to have a better idea of the issues to be discussed throughout Sections 3.1.1-3.1.4. Figure 3.3 presents such an iteration hardware overview where the various iteration steps are indicated. The iteration hardware follows the previously set limitations and implements the decision process as defined by the Range Trie method (see Section 2.2.3). The depicted Range Trie iteration hardware is for an incoming address width (W) of 32 bits and an available memory bandwidth (BW).

Generally speaking, the four iteration steps are designed as follows and operate according to the visited node information:

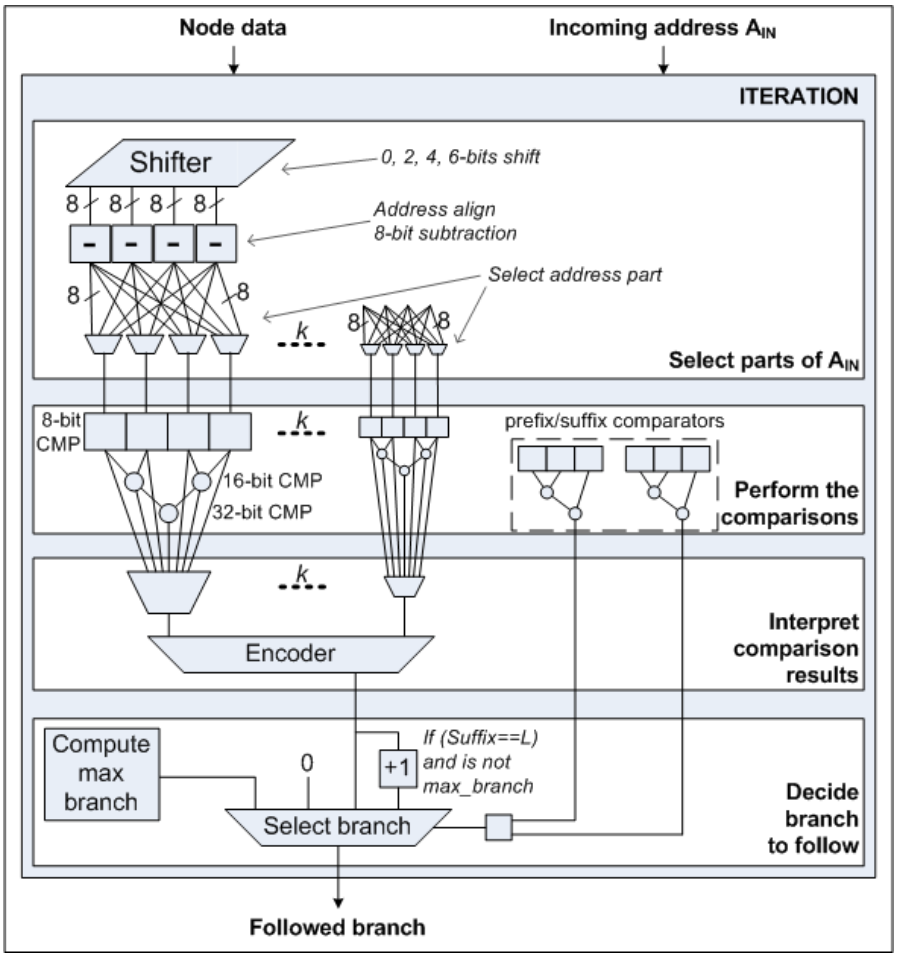


Figure 3.3: An abstract block diagram depicting the hardware design of the Range Trie iteration. The exact details of the required hardware to perform each iteration step will be discussed in the coming sections.

1. The incoming address A_{IN} is byte-aligned (using the shifter). If needed, the byte-aligned address is subtracted to perform the address bounds' alignment. Then the byte-selection occurs using one level of 4-to-1 multiplexors to select the parts of A_{IN} to be compared in every comparator.

2. The comparisons are performed between the predetermined values and the parts of A_{IN} using $k = BW/W$ comparators. Each comparator comprises of 8-bit comparators, whose results may be combined to form 16-bit comparisons, 32-bit comparisons, etc. In parallel with the k comparators, the common prefix/suffix comparisons are performed using two $W - 8$ bit wide comparators.

3. The comparison results (of the k comparators) are interpreted into a single value (*range*), after filtering out the invalid comparison results.

4. According to the computed *range* and the common prefix/suffix comparison re-

sults, the correct branch to follow is chosen (out of 0, *max_branch*, *range*, $range + 1$), while following the decision criteria of the Range Trie method.

The two parameters that mainly affect the Range Trie iteration hardware design are the *available memory bandwidth* (BW) and the *incoming address width* (W), as these two values indicate the number of employed comparators, the widths of the internal signals and the used variation of the units. As already mentioned, in this thesis, the Range Trie was designed for a specific combination of available memory bandwidths (256, 512 or 1024 bits wide) and incoming address widths (32, 64 or 128 bits), as these values result into representative designs which can be evaluated in terms of scalability.

In the rest of this section, all the details for the hardware design of a Range Trie iteration unit will be presented. The required components will be described for all possible combinations of the memory bandwidth and address width parameters in order to support the generation of every possible Range Trie instance. Alongside, the control signals for performing the required computations will be defined in order to formulate the way to represent the node information.

In particular, in the next four sections the hardware design of the units needed for performing each of the iteration actions will be explained, while targeting a fast and scalable design. Afterwards, Section 3.1.5 will conclude the description of the Range Trie iteration hardware by combining all required units to form the complete iteration functional unit, along with an overview of the designed iteration unit.

3.1.1 Selecting parts of addresses

The Range Trie method is based on performing comparisons *on parts of addresses*. In this section, it will be explained how we designed the hardware for selecting parts of addresses, along with the possible alignment of addresses by subtraction.

The first action to be performed during an iteration is to prepare the values to be compared, based on the incoming address A_{IN} and the current visited node N information. Assume that we are currently visiting an internal Range Trie node N, that the given memory bandwidth (BW) is 256, 512 or 1024 bits and that the incoming address A_{IN} width (W) is 32, 64 or 128 bits. It has been mentioned that the hardware design uses $k = BW/W$ comparators. The necessary node N information to prepare the values to be compared per comparator are the following:

- The common prefix length (CP) and the common suffix length (CS): These range between 0 and $W - 8$, where W is the incoming address width, and must be a multiplicand of 2. Also, the following must hold: $CS + CP \leq W - 8$.

- The subtraction value (SUB) for alignment purposes: If $SUB = 0$, then no alignment is performed. Otherwise, it must be an $W/4$ bits wide value.

- The comparison widths of the comparisons to be performed per comparator: There are $k = BW/W$ comparators in total. Each one is W bits wide and performs a set $[w_1, w_2, ..., w_s]$ of s variable-length comparisons. The allowed comparison lengths may be 8, $2*8$, $4*8$, ..., W. The sum of the comparison widths w_i to be performed

should be W. The allowed combinations of comparison widths are those that the sum of two consecutive comparison widths $w_i + w_{i+1}$, where i is odd, is one of the available comparison widths ($2 * 8$, $4 * 8$, ..., W). The k-th comparator always performs a full-length (W bits) comparison. A given comparator may be disabled completely. If a comparison within a comparator must be disabled, the comparison width must still be set and the disabling is achieved by storing zeros in the respective bits of the comparison value. In case an alignment must occur, then the only allowed comparison width is $W/4$. I.e. the allowed comparison widths for a 32-bit wide comparator are [32], [16 16], [16 8 8], [8 8 16], [8 8 8 8].

All of these information are enough to determine which parts of A_{IN} to select for comparison. At the same time, the allowed values of these information show the hardware limitations posed on the Range Trie method for an efficient hardware design.

What needs to be done is to select the parts of the incoming address and possibly perform a subtraction for address alignment purposes (see Figure 3.4). The selection means stripping the CP most significant bits and the CS least significant bits of A_{IN} and select 8, $2 * 8$, $4 * 8$, ..., or W bits from the remaining A_{IN} bits, according to the comparison widths per comparator. Also, a subtraction might need to be performed in the selected part of A_{IN}.

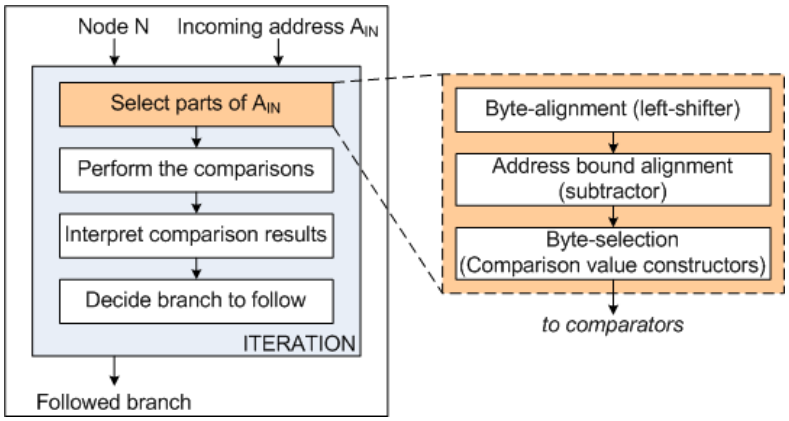


Figure 3.4: An abstract block diagram of the Range Trie iteration focusing on iteration step 1.

There are numerous ways to perform these tasks. In the rest of this section, the chosen design for preparing the values for comparison is presented. Since a part of A_{IN} must be selected, A_{IN} must be aligned properly and then feed the comparators with the selected part. Instead of performing the alignment using a shifter that might need to shift 0, 2, 4, ..., or $W - 8$ positions, we used two levels of alignments. First, a byte-alignment is performed by shifting A_{IN} by 0, 2, 4 or 6 bits left. Then the corresponding parts are selected by performing byte-selection on the shifted A_{IN}. This is advantageous, since the byte-alignment is performed by simple 4-to-1 multiplexors and the byte-selection using $(W/8)$-to-1 multiplexors. The benefit of limiting the address parts to be obtained by 2-bits shifts may already be seen. If any part was to be chosen, then the byte-alignment would require a slower 8-to-1 multiplexor instead of a 4-to-1 multiplexor.

Using the two levels of alignments is beneficial for other reasons as well. The byte-alignment is common for all the k comparators, while the byte-selection is different for each comparator depending on the comparison widths performed by each comparator. If just one level of alignment was to be used, then the byte-alignment would have to be replicated for each comparator, resulting into redundant logic replication. Furthermore, the design of each alignment level was simplified, since the byte-alignment is performed in the same way for every possible incoming address width and the byte-selection for address width W is performed by using the byte-selection hardware for address width $W/2$.

Another factor that led to the use of two levels of alignment was the need for (possible) subtraction on the selected part of addresses. Instead of using a subtractor for each comparator after an address part has been chosen, we may now employ the subtractor after byte-alignment. Since the subtraction width is know to be always $W/4$, that meant that after byte-alignment it is possible to subtract on the correct part of address and then byte-select the correct part of the address.

Up to this point we mentioned the existence of k comparators and the need to select parts of addresses for each one of them. The special case of the k-th comparator must be noted, which always performs a full-width W-bits comparison. This means that no address part selection is needed for it. The reason for using such a comparator is to take even more advantage of the given memory bandwidth. In most of the iterations, when there is a shared common prefix/suffix, we occupy $W-8$ bits of the memory bandwidth to hold the common prefix/suffix value to be compared. In case that there is not a shared common prefix/suffix and the rest $k-1$ perform full width comparisons, then an extra comparison may be performed by using the k-th comparator and by storing one extra value to be compared in place of the now non-existent common prefix/suffix value. Using the extra k-th comparator further utilizes the memory bandwidth, does not affect the critical path and the only configuration signal it needs is a simple enable/disable.

Finally, there is the prefix/suffix value of A_{IN} that needs to be compared (if needed) in the prefix and suffix comparators. Selecting the prefix/suffix of A_{IN} is trivial and is explained in a later section, when the prefix/suffix comparators are introduced.

To summarize, first a 0, 2, 4 or 6-bits left shift is performed on A_{IN}, then the byte to be selected is subtracted by SUB (if needed) and finally the bytes are selected according to the comparison widths per comparator. This process is not needed for the k-th comparator.

To perform these operations three hardware units were designed. The *left-shifter with 0-filling*, the *subtractor unit* and the *comparison value constructor*. These will be detailed afterwards for all possible incoming address widths. In order to drive the correct operation of the units, we used two parameters: (a) $shift\_ctrl$ (2-bits wide) that determines the number of left shifts for byte-alignment and (b) $start\_byte$ ($log_2(W/8)$ bits wide) that determines the most significant byte for byte-selection. These values are set as follows:

- If there is no alignment ($SUB = 0$), then the part to be selected starts after the common prefix. Thus, $shift\_ctrl = CP\%8$ and $start\_byte = (W/8 - 1) - CP/8$.

- If alignment must be performed, then the part to be selected is $W/4$ bits long and ends before the common suffix. Thus, $start\_byte = CS/8 + (CS\%8 \neq 0)$ and $shift\_ctrl = 0$ (if $CS\%8 = 0$) or $shift\_ctrl = 8 - CS\%8$ (if $CS\%8 \neq 0$).

The three required hardware units will be described in the rest of this section, for all possible incoming address widths (32, 64, 128 bits). The given memory bandwidth does not affect these components, apart from the times that they are going to be used in the complete design. After presenting each unit, the way to integrate all of them will be shown, along with the means to configure all the necessary control signals for a correct operation.

3.1.1.1 Left-shifter with 0-filling

The *left-shifter with 0-filling* is responsible for shifting A_{IN} left by 0, 2, 4 or 6 bits, while filling the least significant bits with zeros. As mentioned, this shifting is needed to "byte-align" the incoming address and, thus, simplifying the selection of address parts to be compared.

The hardware design of the left-shifter is depicted in Figure 3.5. The input is the W-bit wide A_{IN} and the number of bit-shifts is controlled by the 2-bit $shift\_ctrl$ signal. The possible values of $shift\_ctrl$ are "00", "01", "10" and "11" representing 0, 2, 4 or 6-bits shift respectively. The output is the shifted A_{IN} by 0, 2, 4 or 6-bits left with 0-filling. As seen in Figure 3.5, the shifter is implemented using an array of $W/2$ 4-to-1 multiplexors. Although the figure depicts the shifter for incoming address width of 32-bits, the design is similar for 64 and 128-bits.

The shifted A_{IN} output of the left-shifter will then be passed to the following subtractor unit to perform a subtraction, if needed.

3.1.1.2 Subtractor unit

The *subtractor unit* performs a subtraction (if needed) on a part of the shifted incoming address. This subtraction is done according to the Rule 5 of the Range Trie method (see Section 2.2.2) in order to align the address bounds in a given Range Trie node N.

The subtraction speed on hardware depends highly on the length of the operands. For that reason, we limited the subtraction width to $W/4$, as mentioned before, without affecting the Range Trie method benefits. Furthermore, to eliminate the cost of a slow subtraction, we employed advanced subtraction techniques.

The input to the subtractor unit is the W-bit wide shifted A_{IN} that the left-shifter outputs. Then the $W/4$ bits wide SUB value is subtracted from $W/4$ bits of A_{IN}. The part of A_{IN} to subtract is defined by the $start\_byte$ parameter. The output of the subtractor unit is identical to the input, except the subtracted part.

In the hardware design of the subtractor unit we used an adder instead of a subtractor, which is an equivalent solution as long as SUB is the 2's-complement of the value to subtract. This is not done in hardware due to the high cost of 2's-complementing a binary number. Instead, the 2's-complement of the value to subtract should be provided

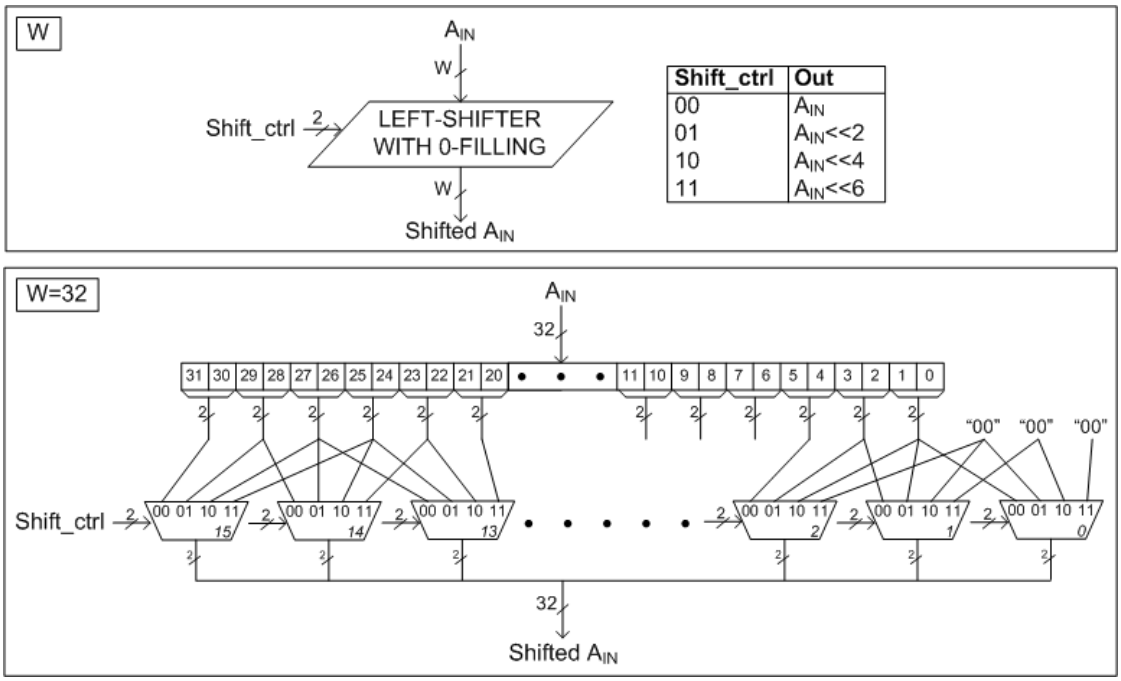


Figure 3.5: The left-shifter with 0-filling. In the top of the figure, the block diagram of the unit is depicted, along with its usage. In the bottom, the specific hardware design is shown for the case of 32-bits wide incoming address.

directly to the subtractor unit. In case that a subtraction is not needed, then SUB should be set to 0.

In the rest of this section, the designs for the subtractor units for incoming address widths (W) of 32, 64 and 128-bits will be presented in detail.

Subtractor unit for W=32: The subtractor unit for incoming address width of 32-bits must perform an 8-bit wide subtraction to one of the 4 bytes of the shifted A_{IN}. The bytes of the shifted A_{IN} are counted from right-to-left and the 2-bit value *start_byte* determines which byte of the shifted A_{IN} to subtract. The rest bytes remain unaltered.

Since the target is a fast design, we used 4 parallel adders (each 8-bits wide) to perform the subtraction. The alternative would be to use just one adder. This alternative was dismissed since it would require a 4-to-1 multiplexor before the subtraction to choose the correct byte of the shifted A_{IN} and an array of multiplexors after the subtraction to place the subtracted byte in the correct position in the output.

The design that was chosen is shown in Figure 3.6. It may be seen that just one byte of the shifted A_{IN} is subtracted by SUB, depending on the *start_byte*. To achieve this, an array of 4 8-bit adders is used to subtract SUB out of each byte. Then an array of 2-to-1 multiplexors chooses if a addition result or the original value will be output, depending on the *start_byte*.

In order to design a fast subtraction unit the effort was placed on using advanced fast adders. We opted to design the 8-bit adders as a two-level carry select adder, each

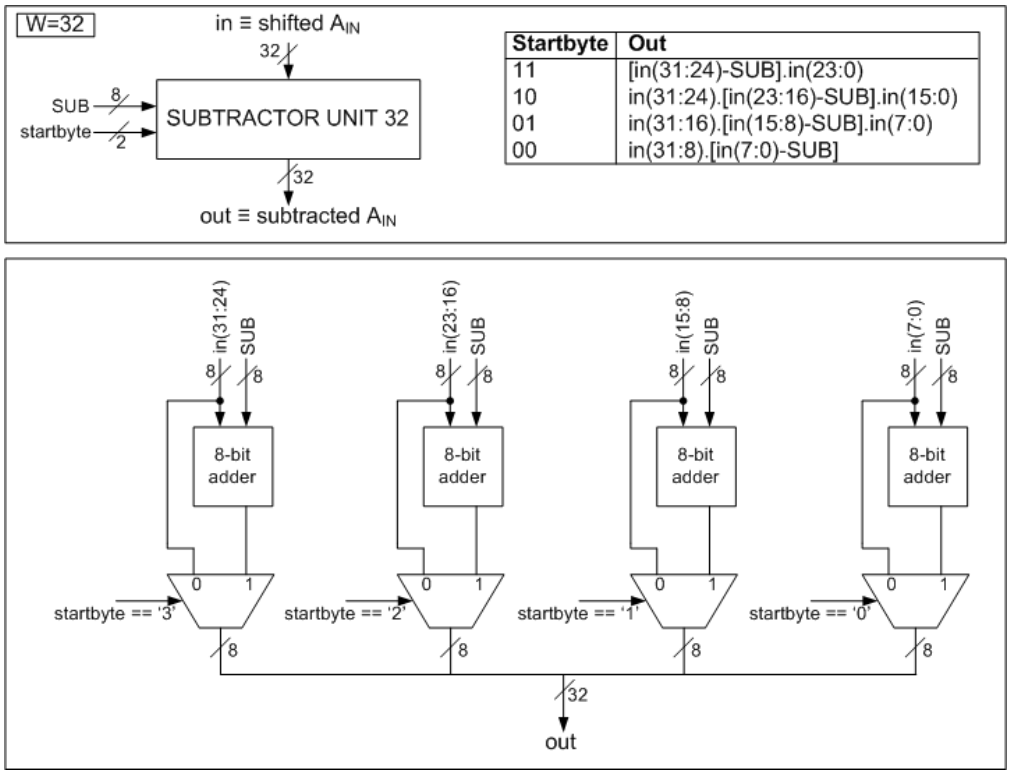


Figure 3.6: The subtractor unit for incoming address width of 32-bits. In the top of the figure, the block diagram of the unit is depicted, along with its usage. In the bottom, the specific hardware design is shown for the case of 32-bits wide incoming address. The design details of the used 8-bit adders are shown in Figure 3.7 assuming a carry-in of 0-value. The multiplexor control signals that state if start_byte is "11", "10", "01" or "00" are computed through simple logic functions.

level using a fast 4-bit carry lookahead adder [21]. The design of the 8-bit carry select adder may be seen in Figure 3.7. It must be noted that carry lookahead adders and carry select adders are a fast solution for addition in the expense of extra logic. The same adder design concept will be used later on to build wider adders out of narrower ones.

Subtractor unit for W=64 and W=128: Since the subtractor unit design depends on the length of the incoming address (W), it is necessary to design it for W=64 and W=128.

As mentioned, the subtract value (SUB) width is limited to $W/4$ bits, while the part of the shifted A_{IN} to subtract may start in any byte of the shifted A_{IN}. This means that the additions to be performed must be $W/4$ bits wide and that the *start_byte* now is $log_2(W/8)$ bits wide.

A similar design is used as the subtractor unit for W=32. The differences are (a) in the additions width (16-bits adders for W=64 and 32-bits adders for W=128) and (b) in the way that the output is formed based on the addition results (as the part to be subtracted is still chosen on a byte basis).

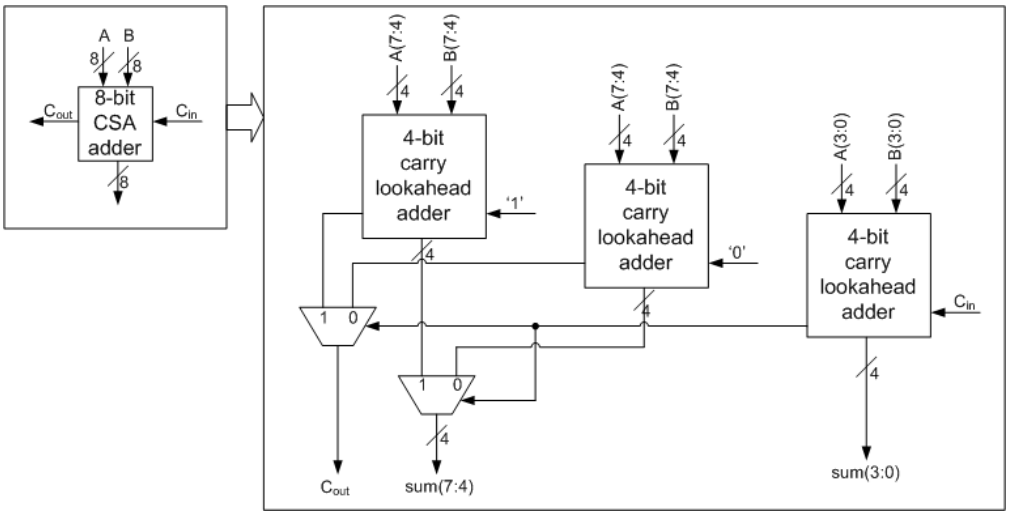


Figure 3.7: The 8-bit carry select adder design. It uses two levels of 4-bit carry lookahead adders. The first level (see right adder) adds the 4 least significant bits of the operands. The second level (see two left adders) adds the 4 most significant bits of the operands for an assumed carry-in of 0 and 1 and then chooses the correct result according to the carry-out of the first level. The 4-bit carry lookahead adders are designed based on standard fast logic functions (see [21]).

Since the adders width is larger for W=64 and W=128, it is even more imperative to use a fast addition scheme. The new adders that are needed (16 and 32-bits wide) are designed in the same fashion as the 8-bit adders for the subtractor unit for W=32 (see Figure 3.7). To build a 16-bit carry select adder, the 4-bit carry lookahead adders of Figure 3.7 were replaced by the whole 8-bit carry select adder. In the same way, the 32-bit carry select adders were designed. Using this adder design, we achieved to design a wide adder (i.e. 32-bits wide) that has almost the delay of a 4-bit carry lookahead adder, plus the delays for selecting the correct results based on the correct carry-ins. This was achieved in the cost of extra hardware logic.

The subtractor units for W=64 and W=128 may perform a subtraction in every byte of the incoming shifted A_{IN} based on the *start_byte* value. The *start_byte* is 3-bits wide for W=64 and 4-bits wide for W=128. Assuming that *start_byte* is i, then a 16-bits addition must be done on bytes $i+1$ and i, for W=64, and a 32-bits addition on bytes $i+3$, $i+2$, $i+1$ and i, for W=128. Since the bytes to be added must be compatible with the number of bytes of the shifted A_{IN} (total bytes are 8 for W=64 and 16 for W=128), then the allowed values of *start_byte* are $[0, 6]$ for W=64 ($i+1$ must be less than 8) and $[0, 12]$ for W=128 ($i+3$ must be less than 16).

It is evident that there are 7 possible 16-bit additions for W=64 and 13 possible 32-bit additions for W=128. Although there are overlaps between the needed additions, separate parallel adders are used for each address part to ensure the minimum delay. Figures 3.8 and 3.9 depict the designs of the subtractor units for W=64 and W=128 bits respectively. As in the subtractor unit for W=32, an array of $W/4$ bit adders is used followed by an array of multiplexors that choose either the addition result or the initial address part, according to a part of the *start_byte* value. Furthermore, there is

an extra needed final multiplexor to choose the correct output, according to the rest of the *start_byte* value.

Variations on the subtractor unit - A variable-width subtractor unit: Up until now, the performed subtractions had a constant-width of $W/4$ bits in order to maintain a static subtraction delay. This was posed as a limitation on the hardware Range Trie design, as mentioned in the beginning of Section 3.1. A variation of the subtractor unit that allows variable-width subtractions will be presented in this paragraph.

According to the Range Trie method rules (see Section 2.2.2), the subtraction is performed (if needed) to align the bounds of a given Range Trie node N and increasing the subtraction width might increase the efficiency of a Range Trie structure. At the same time, widening the subtraction, increases the delay of the subtractor unit. For that reason, we decided to explore the possibility of a subtractor unit with a variable subtraction width. The subtraction width is limited to be either $W/4$ (as before) or $W/2$. Not every possible subtraction width is allowed. These subtraction widths were chosen since the subtraction may be performed using two existing consecutive $W/4$ wide subtractors and do not add to the complexity of the design. The effects of using this variable-width subtractor unit will be explored later in Chapter 4. In any case, in the rest of the Range Trie hardware design the use of the constant-width subtractor will be assumed.

The node information for controlling such a variable-width subtraction are different than before. First, a wider SUB value ($W/2$ bits long) must be used to hold either the $W/4$ bits long subtract value or the $W/2$ bits long subtract value. Also, a signal is needed to indicate the length of the subtraction ($W/4$ or $W/2$ bits).

The *start_byte* value width remains $log_2(W/8)$ and its semantics are the same as before. Furthermore, as in the constant-width subtractor units, there are also limitations on the possible values of *start_byte*. Specifically:

- For W=32: The allowed subtraction widths are 8 and 16 bits. The total number of bytes are 4. When performing 8-bit subtractions, the allowed values of *start_byte* are $[0, 3]$. When performing 16-bit subtractions, then they are $[0, 2]$.
- For W=64: The allowed subtraction widths are 16 and 32 bits. The total number of bytes are 8. When performing 16-bit subtractions, the allowed values of *start_byte* are $[0, 6]$. When performing 32-bit subtractions, then they are $[0, 4]$.
- For W=128: The allowed subtraction widths are 32 and 64 bits. The total number of bytes are 16. When performing 32-bit subtractions, the allowed values of *start_byte* are $[0, 12]$. When performing 64-bit subtractions, then they are $[0, 8]$.

The hardware design of the variable-width subtractor units shares a great deal of similarity with the constant-width subtractor units presented before. The way that the array of adders is organized is identical. The same number and types of adders are used and the same parts of the shifted A_{IN} are subtracted per adder. The main difference is the extra logic for deciding which value to subtract per part.

The design of the variable-width subtractor unit for W=32 is depicted in Figure 3.10 and it will be described against its constant-width counterpart of Figure 3.6. Since the

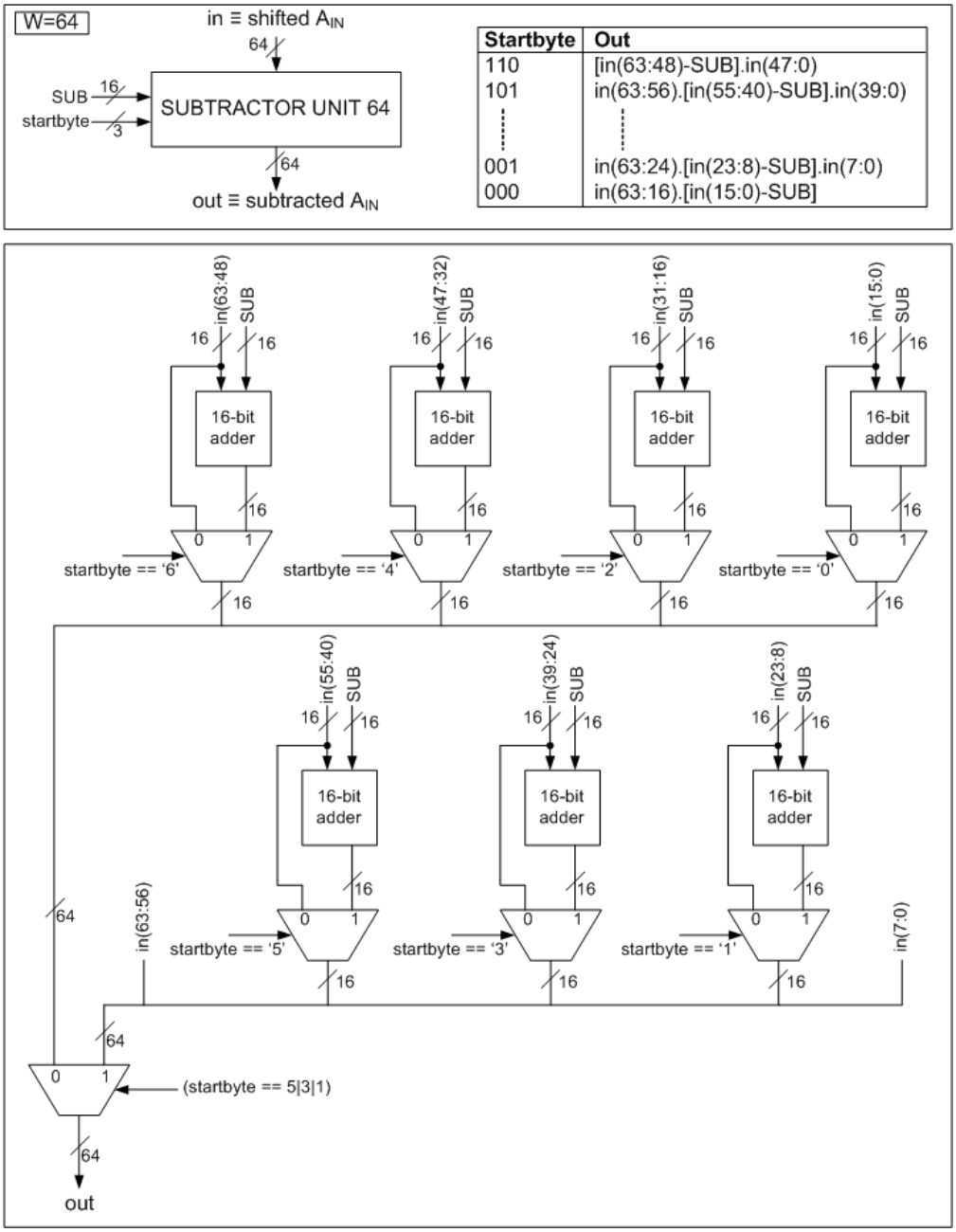


Figure 3.8: The subtractor unit for incoming address width of 64-bits. In the top of the figure, the block diagram of the unit is depicted, along with its usage. In the bottom, the specific hardware design is shown for the case of 64-bits wide incoming address. The 16-bit adders are designed in a similar way as the 8-bit adders of Figure 3.7. The allowed values of the 3-bit start_byte are [0, 6]. The control signals for the multiplexors that are after the adders are computed through simple logic functions that inspect the two most significant bits of start_byte. The control signal for the final multiplexor is identical to the least significant bit of start_byte.

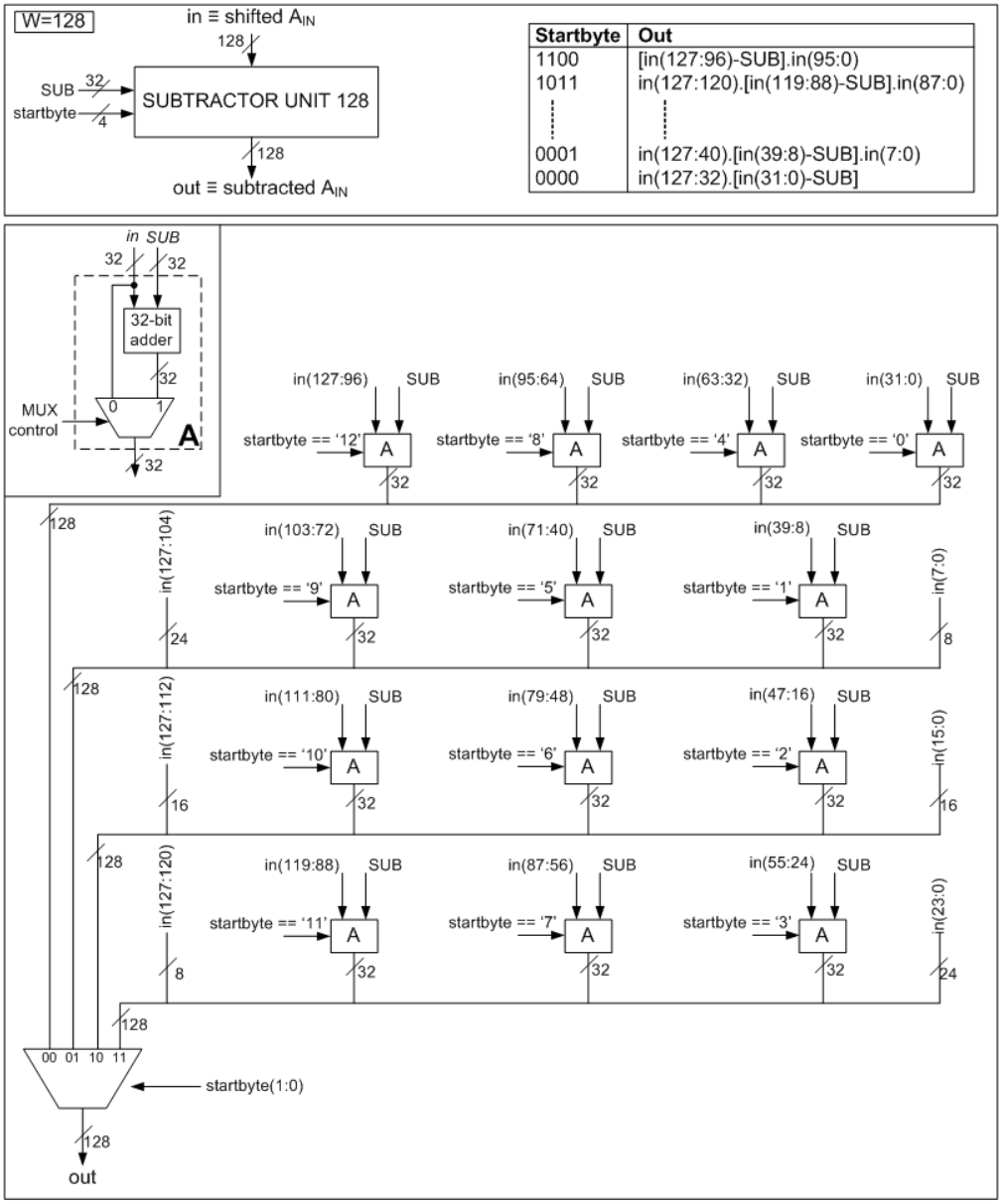


Figure 3.9: The subtractor unit for incoming address width of 128-bits. In the top of the figure, the block diagram of the unit is depicted, along with its usage. In the bottom, the specific hardware design is shown for the case of 128-bits wide incoming address. The 32-bit adders are designed in a similar way as the 8-bit adders of Figure 3.7. The allowed values of the 4-bit start_byte are [0, 12]. The control signals for the multiplexors that are after the adders are computed through simple logic functions that inspect the two most significant bits of start_byte. The control signal for the final multiplexor is identical to the two least significant bits of start_byte.

variable-width subtractor units for W=64 and W=128 are obtained in the same manner, they will not be detailed in this paragraph.

The variable-width subtractor unit for W=32 of Figure 3.10 may perform either 8 or

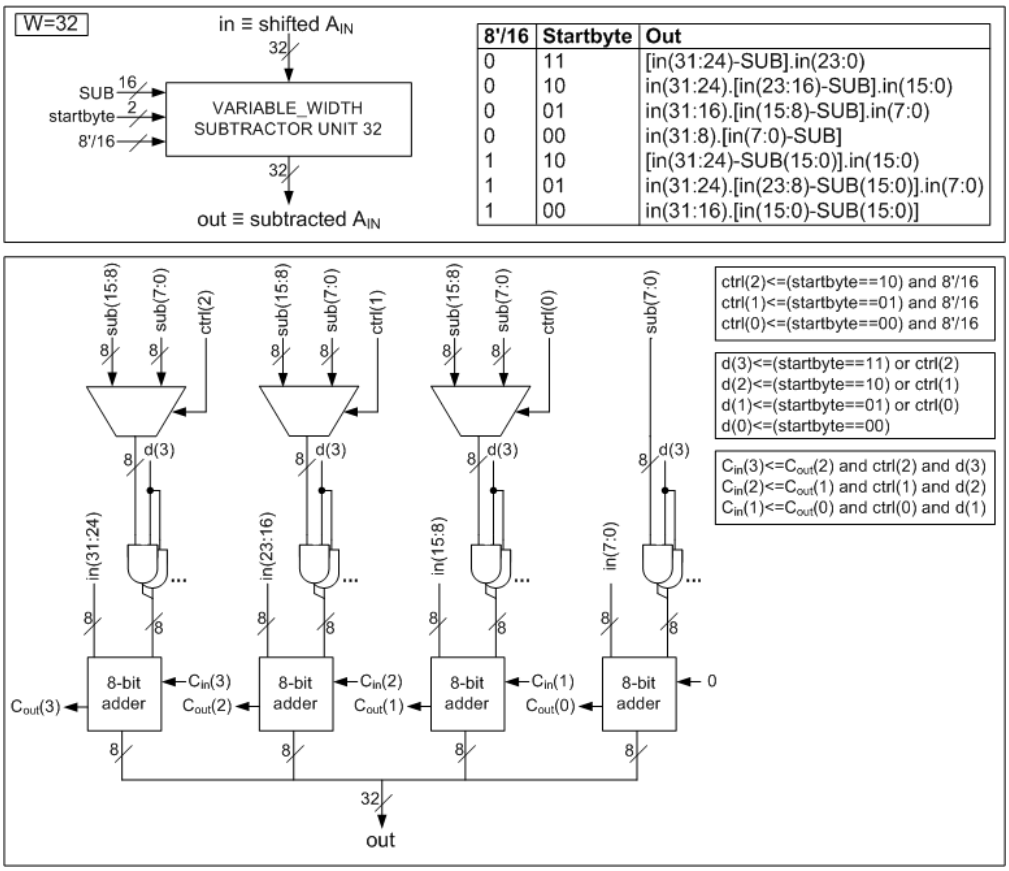


Figure 3.10: The variable-width subtractor unit for incoming address width of 32-bits. In the top of the figure, the block diagram of the unit is depicted, along with its usage. In the bottom, the specific hardware design is shown for the case of 32-bits wide incoming address, along with the logic functions that drive its operation. This design is a modification of the constant-width subtractor unit of Figure 3.6. The variations for W=64 and W=128 bits may be designed in a similar way based on their constant-width counterparts. The 8-bit adders are the carry select adders of Figure 3.7. The allowed values of the 2-bit start_byte are [0, 3] (when subtraction width is 8-bits) and [0, 2] (when subtraction width is 16-bits).

16-bits subtraction (depending on the signal `8'/16`). The SUB signal holds the value to subtract and it is 16-bits wide. In case of an 8-bit subtraction the actual value is stored in the 8 least significant bits of SUB. Once again, SUB must be the 2's-complement of the actual value to be subtracted, since the subtraction is done using adders.

The design for the variable-width subtractor unit for W=32 also uses 4 8-bit carry select adders (as depicted in 3.7) and adds the corresponding byte of the shifted A_{IN} to the correct part of the SUB value (if needed). Assuming that *start_byte* is i, then the 8 least significant bits of SUB are added to byte i of the shifted A_{IN}. In case that the subtraction width is set to 16-bits, then an extra addition must occur, where the 8 most significant bits of SUB are added to byte $i + 1$ of the shifted A_{IN}. In case of a 16-bits subtraction, the two 8-bit adders must be connected for a correct carry propagation. In

the rest bytes of the shifted A_{IN} no addition is performed (by actually adding a 0-value to them).

The procedure that was just described is what differentiates the variable-width subtractor unit from its constant-width counterpart. To achieve this, the two following modifications were made. These two modifications may be seen in Figure 3.10, along with the logic functions that ensure the correct operation of the variable-width subtractor unit.

- Each adder z is connected to its right adder $z-1$ through the carry-out of the right adder $z-1$. In particular, the carry-out of $z-1$ must be considered by z, whenever there is a 16-bit subtraction that starts in byte $i = z-1$.
- The 8-bit value to be added in adder z is chosen out of the 0-value or the 8 least significant bits of SUB or the 8 most significant bits of SUB, based on the *start_byte* and the subtraction width (`8'/16`). The hardware that was used for this selection is a 2-to-1 multiplexor followed by an array of AND-gates.

To conclude, the details of the hardware design of the subtractor unit were presented for the cases of incoming address widths of 32, 64 and 128 bits. The usage and semantics of each case were presented, along with a variation of the subtractor unit that performs variable-width subtractions. The presented subtraction units operate on the incoming address only when an address alignment is needed. If there is no need for alignment the incoming address remains unaltered. In any case, the output of the subtractor unit is passed to the comparison value constructor to byte-select the parts of the incoming address to be compared later on at the comparators.

3.1.1.3 Comparison value constructor

The *comparison value constructor* is the unit responsible for the second level of the selection of parts of the incoming address A_{IN}. There is one such unit per employed comparator, since the comparison value constructor creates the value to be compared in the corresponding comparator. The input of the comparison value constructor is the subtracted A_{IN} coming out of the subtractor unit.

The necessary values needed to select the parts of the subtracted A_{IN} to be compared in comparator i are: (a) the comparison widths for comparator i and (b) the *start_byte* (that is common to all comparators per iteration). These values are obtained out of the currently visited node information.

Assuming that *start_byte* is $i \in [0, W/8]$ (bytes are counted from right-to-left) and the comparison widths for a comparator are $[w_1, ..., w_s]$, then:

- For each w_j, the w_j-bit long part of the subtracted A_{IN} is selected that starts in bit $(i+1)*8-1$ of the subtracted A_{IN} (bits are counted from right-to-left starting from 0). In case the combination of i and w_j is such that the available part to select is less than w_j bits, then the rest bits are set to zeros. Since the allowed comparison widths may be 8, $2*8$, $4*8$, ..., W, then this process is narrowed down to a byte-selection, instead of a bit-selection.

- The comparison value to be compared in the comparator is a concatenation of all the selected parts in order.

The hardware that is needed for the comparison value construction is just an array of $(W/8)$-to-1 multiplexors. Each multiplexor is connected to all bytes of the subtracted A_{IN} and there are control logic functions that dictate which byte to select per multiplexor, according to the *start_byte* and the widths to be selected.

It is evident that the comparison value constructor design depends on the the incoming address width (W). For that reason, the respective designs for W=32, W=64 and W=128 bits will be presented in the rest of this section. It is also shown that the design is modular, since the designs for W=64 and W=128 use the logic controls of the design for $W/2$.

Comparison value constructor for W=32: The comparison value constructor for incoming address width of 32 bits performs the byte-selection of the subtracted A_{IN} according to a given *start_byte* and a set of comparison widths. The *start_byte* may be any of $[0, 3]$ (the bytes are counted from right-to-left). The allowed sets of comparison widths are [32], [16 16], [8 8 16], [16 8 8] and [8 8 8 8], according to the limitations posed in the beginning of Section 3.1. A comparator may be disabled and this case is not dealt during the comparison value construction, but during the interpretation of the comparison results.

To better understand what the comparison value constructor does and its hardware design, Table 3.1 shows what is the desired output in terms of the *start_byte* and the possible comparison widths.

Out of the expected output of Table 3.1, the following set of functions may be derived that describe the operation of the comparison value constructor. Note that the required subtraction on *start_byte* are assumed to allow overflow (i.e. if $start\_byte = 1$, then $start\_byte - 3 = 2$).

- $OutByte_3 = Byte_{start\_byte}$
- $OutByte_2 = Byte_{start\_byte-1}$ (if [32] or [16 16] or [16 8 8])
  $OutByte_2 = Byte_{start\_byte}$ (if [8 8 16] or [8 8 8 8])
- $OutByte_1 = Byte_{start\_byte-2}$ (if [32])
  $OutByte_1 = Byte_{start\_byte}$ (if [16 16] or [16 8 8] or [8 8 16] or [8 8 8 8])
- $OutByte_0 = Byte_{start\_byte-3}$ (if [32])
  $OutByte_0 = Byte_{start\_byte-1}$ (if [16 16] or [8 8 16])
  $OutByte_0 = Byte_{start\_byte}$ (if [16 8 8] or [8 8 8 8])
- $Byte_3$ must be dealt as a 0-value, when $start\_byte < 3$
  $Byte_2$ must be dealt as a 0-value, when $start\_byte < 2$
  $Byte_1$ must be dealt as a 0-value, when $start\_byte < 1$

These previous functions point that the byte selection process may be easily designed based on the values of *start_byte* and the comparison widths. Before presenting the actual hardware design the representation of the comparison widths in binary format

Table 3.1: The desired operation of the comparison value constructor for incoming address width of 32 bits. All the possible combinations of start_byte and comparison widths are present. $Byte_i$ denotes the respective byte of the subtracted A_{IN} (bytes are counted from right-to-left starting from 0). The starred entries are for the cases where the comparison width exceeds the subtracted A_{IN} width and will be set to 0. These were chosen in a way to follow the pattern of the non-starred entries and thus leading to simplified selection logic.

Comparator widths	**Start byte**	**Output**			
		$OutByte_3$	$OutByte_2$	$OutByte_1$	$OutByte_0$
[32]	3	$Byte_3$	$Byte_2$	$Byte_1$	$Byte_0$
	2	$Byte_2$	$Byte_1$	$Byte_0$	$Byte_3*$
	1	$Byte_1$	$Byte_0$	$Byte_3*$	$Byte_2*$
	0	$Byte_0$	$Byte_3*$	$Byte_2*$	$Byte_1*$
[16 16]	3	$Byte_3$	$Byte_2$	$Byte_3$	$Byte_2$
	2	$Byte_2$	$Byte_1$	$Byte_2$	$Byte_1$
	1	$Byte_1$	$Byte_0$	$Byte_1$	$Byte_0$
	0	$Byte_0$	$Byte_3*$	$Byte_0$	$Byte_3*$
[8 8 16]	3	$Byte_3$	$Byte_3$	$Byte_3$	$Byte_2$
	2	$Byte_2$	$Byte_2$	$Byte_2$	$Byte_1$
	1	$Byte_1$	$Byte_1$	$Byte_1$	$Byte_0$
	0	$Byte_0$	$Byte_0$	$Byte_0$	$Byte_3*$
[16 8 8]	3	$Byte_3$	$Byte_2$	$Byte_3$	$Byte_3$
	2	$Byte_2$	$Byte_1$	$Byte_2$	$Byte_2$
	1	$Byte_1$	$Byte_0$	$Byte_1$	$Byte_1$
	0	$Byte_0$	$Byte_3*$	$Byte_0$	$Byte_0$
[8 8 8 8]	3	$Byte_3$	$Byte_3$	$Byte_3$	$Byte_3$
	2	$Byte_2$	$Byte_2$	$Byte_2$	$Byte_2$
	1	$Byte_1$	$Byte_1$	$Byte_1$	$Byte_1$
	0	$Byte_0$	$Byte_0$	$Byte_0$	$Byte_0$

must be decided. This representation is also used as the configuration of the comparator. Since there are 5 different sets of comparison widths, plus the possibility of a disabled comparator, 3 bits suffice to represent these 6 different cases. We chose to use just 3 bits in order to minimize the size of the control signals. The binary representation of the comparator modes for W=32 are shown in table 3.2.

The hardware design design of the comparison value constructor for W=32 is depicted in Figure 3.11. It consists of an array of 4 8-bit wide 4-to-1 multiplexors. Each multiplexor selects one byte of the input according to the multiplexor control signals. In case the selected byte must be converted to 0, the AND-gates are used.

The multiplexor control signals are computed based on the start_byte and the comparator mode, according to the set of functions described above. The logic needed for that is also shown on Figure 3.11. To calculate the necessary subtractions $start\_byte - i$, where $i \in [1, 3]$, no subtractor will be used. Since the start_byte is just 2-bits wide and the i is always known, the $start\_byte - i$ are easily calculated through one level logic functions.

Table 3.2: The binary representation of comparator modes for incoming address width of 32 bits. The case of a disabled comparator is not dealt by the comparison value constructor.

Comparator mode	Encoding
disabled	111
[32]	100
[16 16]	011
[8 8 16]	010
[16 8 8]	001
[8 8 8 8]	000

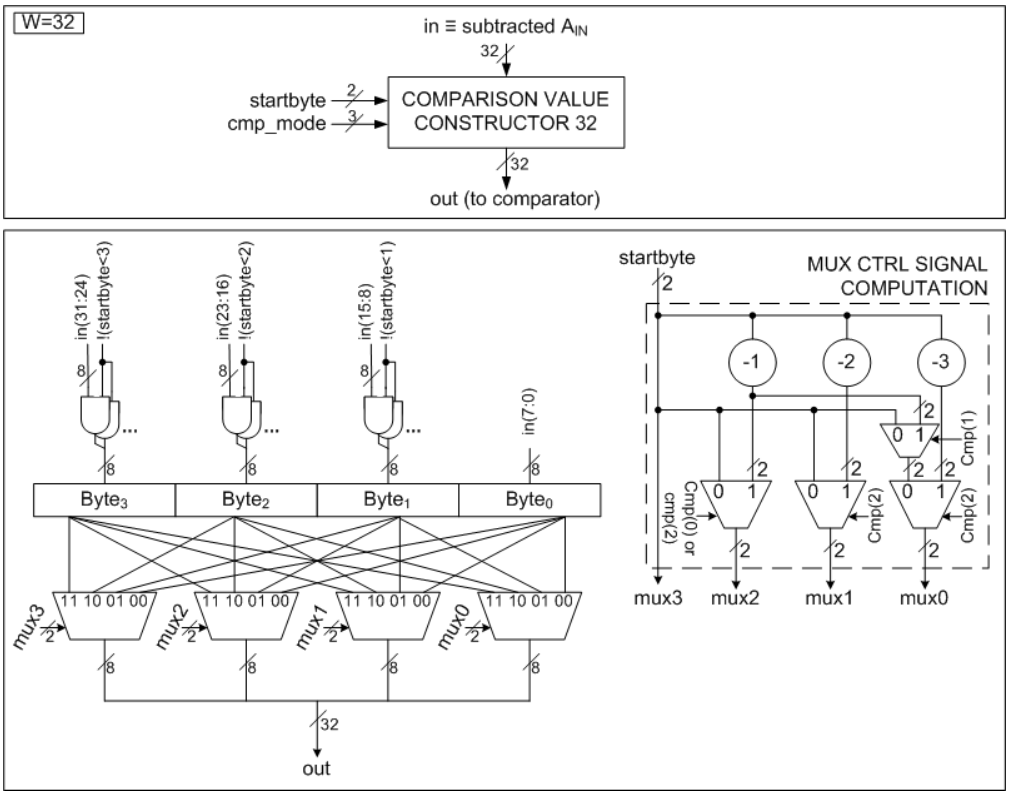


Figure 3.11: The comparison value constructor for incoming address width of 32-bits. In the top of the figure, the block diagram of the unit is depicted. The usage of the unit is described in Table 3.1. In the bottom, the specific hardware design is shown for the case of 32-bits wide incoming addresses, along with the module that computes the multiplexor control signals. The allowed values of the 2-bit start_byte are [0, 3]. The 3-bit cmp_mode must follow the values in Table 3.2. The AND-gates' control signals (start_byte $< i$) and the subtractions start_byte $- i$ are calculated through simple logic functions not shown on the figure.

Comparison value constructor for W=64 and W=128: The hardware designs of the comparator value constructors for incoming address widths (W) of 64 and 128 bits share the same principles as the one for W=32: there will be an array of $W/8$ 8-bit wide

($W/8$)-to-1 multiplexors, along with AND-gates to turn bytes of the incoming subtracted A_{IN} into 0 and logic to compute the multiplexors control signals for byte selection.

The main difference is in the computation of the multiplexor control signals due to the different allowed values of *start_byte* and comparison widths. For W=64, the *start_byte* may be in $[0, 7]$ and for W=128 in $[0, 15]$. The allowed comparison widths for W=64 and W=128 are actually a combination of the allowed comparison widths for $W/2$ plus an allowed full-width comparison W. Since the allowed comparison widths for W=32 are [32], [16 16], [8 8 16], [16 8 8], [8 8 8 8], then i.e. for W=64 the comparison widths may be [64], [32 : 32], [16 16 : 16 16], [8 8 8 8 : 8 8 8 8], [32 : 16 16], [8 8 8 8 : 16 16], etc. I.e. for W=128 they may be [128], [64 : 64], [32 32 : 32 32], [16 16 16 16 : 64], [32 32 : 8 8 16 32], etc. This means that the number of possible sets of comparison widths are $1 + 5 * 5 = 26$ for W=64 and $1 + 26 * 26 = 677$ for W=128. These numbers are actually 27 and 678 due to the possibility of a disabled comparator.

The expected operation of the comparison value constructor for a given set of comparison widths and a *start_byte* follows the same principles as for W=32 (see Table 3.1), so it will not be repeated here.

An important design decision had to be made at this point regarding the binary representation of the comparison widths sets (comparator modes). Previously, the 6 possible comparator modes for W=32 were represented minimally using 3 bits. The same approach could be followed here by using 5 bits (or 10 bits) to represent the 27 (or 677) possible comparator modes for W=64 (or W=128). That way the number of bits to store a comparator mode is minimized. The drawback is that the logic functions for identifying a comparator mode get more complex, along with the extra difficulty and time needed to analyze beforehand the 27 and 678 different possibilities.

Instead of the previous approach, another approach was used for the binary representation of the comparator modes for W=64 and W=128. Since a comparator mode for W=64 and W=128 is a combination of the comparator modes for $W/2$, then a concatenation of the binary representations for $W/2$ may be used. This means that $2*3 = 6$ bits (or $2*6 = 12$ bits) are needed to represent a comparator mode for W=64 (or W=128) out of two 3-bits (or 6-bits) comparator modes for W=32 (or W=64). The resulting binary representations of comparator modes for W=64 and W=128 may be seen in Tables 3.3 and 3.4 respectively. Although this approach uses more bits to represent a comparator mode than the previous one, it is considered beneficial since it simplifies the design process (existing logic from the $W/2$ designs may be used) with a minimized extra cost of 1 (or 2) bits per comparator mode.

The hardware designs of the comparison value constructors are depicted in Figures 3.12 and 3.13 for W=64 and W=128 respectively. These designs have an array of $W/8$ 8-bit wide ($W/8$)-to-1 multiplexors that select a byte according to the multiplexor control logic (using the control logic of the design for $W/2$). Also, there is an array of AND-gates that may turn a $Byte_i$ into a 0-value, when $start\_byte < i$ for $1 \leq i \leq W/8 - 1$.

As mentioned, the selection of bytes for a given W may be performed using the existing logic from the design for $W/2$. As the comparator mode for W is a concatenation of two comparator modes for $W/2$, two instances of the selection logic for $W/2$ will be used to drive the byte selection for W. Each instance will be fed with the corresponding comparator mode half. Both instances will be fed with the $log_2(W/16)$ least significant

*Table 3.3: The binary representation of comparator modes for incoming address width of 64 bits. Each comparator mode is a combination of the possible comparator modes for W=32 (see Table 3.2), except the new comparator mode [64]. The binary representation is a concatenation of the respective binary representations for W=32 (see Table 3.2), except the full-width comparison mode [64] which is represented as 100**101** to avoid collision with other existing representations. The case of a disabled comparator is not dealt by the comparison value constructor.*

Comparator mode	**Encoding**
disabled	111111
[64]	100101
[32 : 32]	100 100
[32 : 16 16]	100 011
[32 : 8 8 16]	100 010
[32 : 16 8 8]	100 001
[32 : 8 8 8 8]	100 000
[16 16 : 32]	011 100
[16 16 : 16 16]	011 011
[16 16 : 8 8 16]	011 010
[16 16 : 16 8 8]	011 001
[16 16 : 8 8 8 8]	011 000

Comparator mode	**Encoding**
[8 8 16 : 32]	010 100
[8 8 16 : 16 16]	010 011
[8 8 16 : 8 8 16]	010 010
[8 8 16 : 16 8 8]	010 001
[8 8 16 : 8 8 8 8]	010 000
[16 8 8 : 32]	001 100
[16 8 8 : 16 16]	001 011
[16 8 8 : 8 8 16]	001 010
[16 8 8 : 16 8 8]	001 001
[16 8 8 : 8 8 8 8]	001 000
[8 8 8 8 : 32]	000 100
[8 8 8 8 : 16 16]	000 011
[8 8 8 8 : 8 8 16]	000 010
[8 8 8 8 : 16 8 8]	000 001
[8 8 8 8 : 8 8 8 8]	000 000

bits of the *start_byte*. These instances may choose one out of $W/16$ bytes, so extra logic must be added to ensure the correct selection out of $W/8$ bytes. The needed extra logic narrows down to computing an extra control bit per multiplexor (the most significant bit of each multiplexor control signal) that dictates whether to select the byte from the upper half or the lower half of the incoming subtracted A_{IN}.

This extra bit (for W) is computed as a function of the *start_byte* and the computed multiplexor control signals (for $W/2$) per multiplexor ($mux_i^{W/2}$). There are three conditions that need to be identified: (a) if $start\_byte \geq W/16$, (b) if $mux_i^{W/2} > start\_byte\%(W/16)$, (c) if the comparator mode is $[W]$. In general, whenever only one of these conditions is true, then the extra bit is set to 1 (the byte is selected from the upper half of the incoming subtracted A_{IN}), otherwise it is set to 0 (the byte is selected from the lower half). For a more specific description of the control signal computation, see the logic functions in Figures 3.12 and 3.13.

Although it is more complex to compute the multiplexor control signals for larger incoming address widths, no delay is added to the complete design due to this extra complexity. This is the case because the computation is not on the critical path of the iteration design. What affects the delay is the extra multiplexor delay due to its increased depth.

To conclude, the comparison value constructor is the final hardware unit needed to

*Table 3.4: The binary representation of comparator modes for incoming address width of 128 bits. Each comparator mode is a combination of the possible comparator modes for W=64 (see Table 3.3), except the new comparator mode [128]. The binary representation is a concatenation of the respective binary representations for W=64 (see Table 3.3), except the full-width comparison mode [128] which is represented as 100100100**110** to avoid collision with other existing representations. The case of a disabled comparator is not dealt by the comparison value constructor. The presented comparator modes in the table are not all the possible ones.*

Comparator mode	**Encoding**
disabled	111111111111
[128]	100100100110
[64 : 64]	100101 100101
[64 : 32 32]	100101 100100
[64 : 32 16 16]	100101 100011
...	...
[32 32 : 64]	100100 100101
[32 32 : 32 32]	100100 100100
[32 32 : 32 16 16]	100100 100011
...	...
[32 16 16 : 64]	100011 100101
[32 16 16 : 32 32]	100011 100100
[32 16 16 : 32 16 16]	100011 100011
...	...
...	...

perform the selection of the parts of the incoming address. The integration of the comparison value constructor with the left-shifter and the subtractor unit will be described in the next section, along with the way to configure the signals that drive the operation of these units, according to the node information.

3.1.1.4 Integrating the units for "selecting parts of addresses"

Up to this point, the necessary hardware units for the selection of parts of addresses were detailed (the *left-shifter* and the *comparison value constructor*). Also, the *subtractor unit* was presented that performs (if needed) an alignment to the incoming address. Since the target is a Range Trie that may have an incoming address width (W) of 32, 64 or 128 bits, all the necessary variations of the units were detailed. In this section, the way to integrate and configure all of these units is detailed in order to perform successfully the selection of parts of the incoming address A_{IN}.

Figure 3.14 depicts the way to integrate these units for a given incoming address width (W) and a given memory bandwidth (BW). A single left-shifter unit is fed by the incoming address A_{IN}. The shifted A_{IN} is passed to the subtractor unit for W. Finally, the subtracted A_{IN} is the input to $k-1$ comparison value constructor for W. Since the number of comparators to be used in an iteration is $k = BW/W$ (where W may be 256, 512 or 1024 bits) and the k-th comparator performs only full-length comparisons (to be

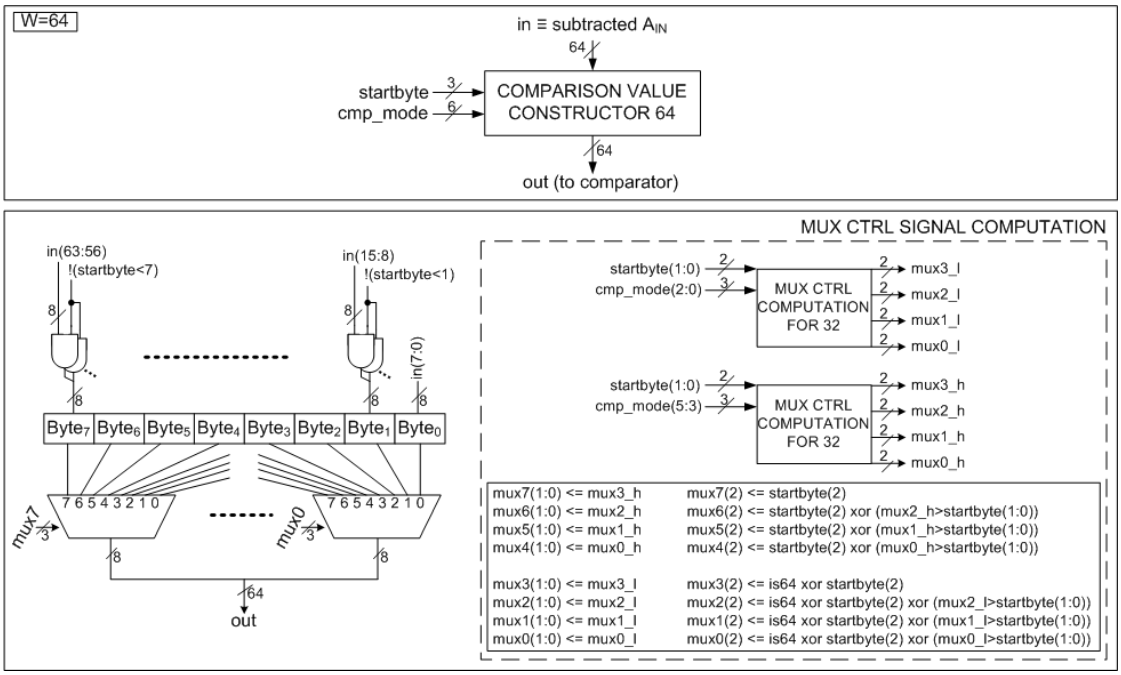


Figure 3.12: The comparison value constructor for incoming address width of 64-bits. In the top of the figure, the block diagram of the unit is depicted. In the bottom, an abstract hardware design is shown for the case of 64-bits wide incoming addresses. Note the use of logic from the design for W=32 (see Figure 3.11) to compute the multiplexor control signals. The allowed values of the 3-bit start_byte are [0, 7]. The 6-bit cmp_mode must follow the values in Table 3.3. The AND-gates' control signals (start_byte < i) and the 2-bit comparisons ($mux_i > start\_byte(1:0)$) are calculated through simple logic functions not shown on the figure.

used only if all other $k-1$ comparators perform also full-length comparisons), it suffices to employ $k-1$ comparison value constructors. Each comparison value constructor feeds the respective comparator to perform the designated comparisons. The k-th comparator does not need any form of comparison value preparation and it is fed directly with the incoming A_{IN}. The same applies for the prefix/suffix comparator.

For the correct selection of the address parts, the control signals of the units must be set up as directed by the node information. The node information consist of: the possible common prefix length (CP), the possible common suffix length (CS), the possible subtract value and the comparison widths to be performed per comparator. The specifications of these values were presented in the beginning of Section 3.1.1. These values are translated into the necessary control signals as follows:

- $Shift\_ctrl$ (2-bits wide):
  If $SUB = 0$, then $shift\_ctrl = (CP\%8)/2$. Otherwise, $shift\_ctrl = 0$ (if $CS\%8 = 0$) or $shift\_ctrl = (8 - CS\%8)/2$ (if $CS\%8 \neq 0$)

- $Start\_byte$ ($log_2(W/8)$-bits wide):
  If $SUB = 0$, then $start\_byte = (W/8 - 1) - CP/8$.
  Otherwise, $start\_byte = CS/8 + (CS\%8 \neq 0)$.

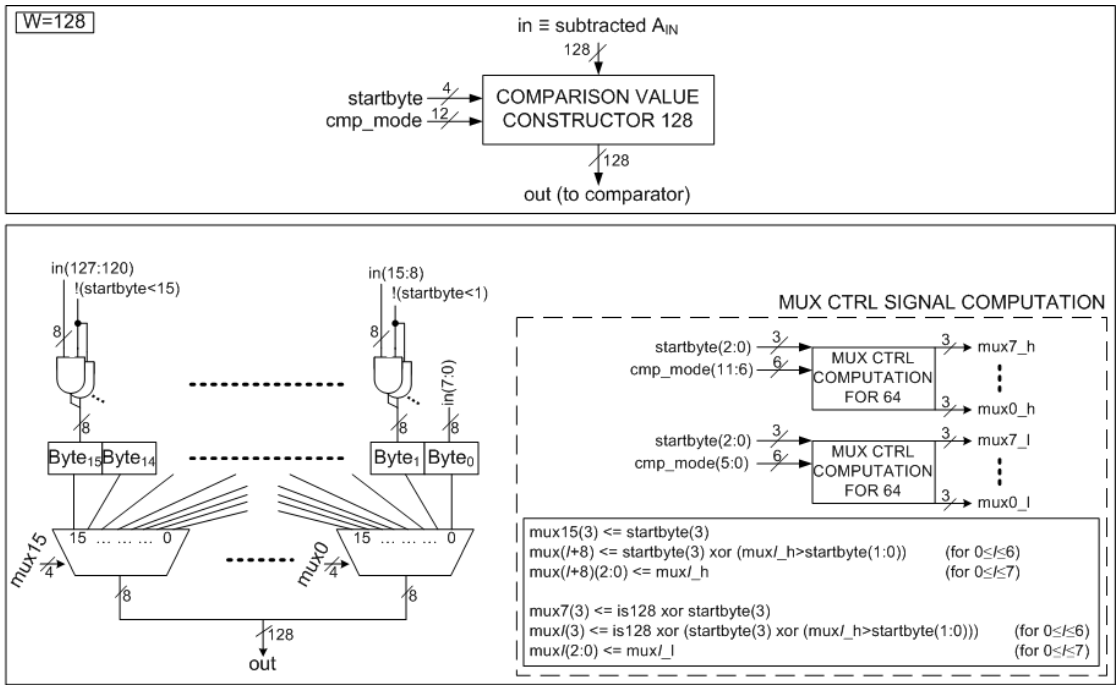


Figure 3.13: The comparison value constructor for incoming address width of 128-bits. In the top of the figure, the block diagram of the unit is depicted. In the bottom, an abstract hardware design is shown for the case of 128-bits wide incoming addresses. Note the use of logic from the design for W=64 (see Figure 3.12) to compute the multiplexor control signals. The allowed values of the 4-bit start_byte are [0, 15]. The 12-bit cmp_mode must follow the values in Table 3.4. The AND-gates' control signals (start_byte < i) and the 3-bit comparisons ($mux_i > start\_byte(2:0)$) are calculated through simple logic functions not shown on the figure.

- SUB ($(W/4)$-bits wide):
  This is the 2's-complement of the value to be subtracted. In case there is no subtraction, it must be set to 0.

- Cmp_mode_i (z-bits wide):
  The comparator mode that each of the $k-1$ comparators operates in. According to the comparison widths to be performed per comparator, the Cmp_mode_i (where $1 \leq i \leq k-1$) is represented in binary according to Tables 3.2, 3.3 and 3.4 for W=32, 64 and 128 respectively. The width (z) of Cmp_mode_i is 3-bits (for W=32), 6-bits (for W=64) or 12-bits (for W=128).

The selection of parts of addresses may now be performed correctly and the result is the values to be compared afterwards in the array of the k comparators. The required comparator units for performing the comparison are presented in the coming section.

3.1.2 Performing the comparisons

The basic operation during a Range Trie iteration is performing a number of comparisons between parts of the incoming address A_{IN} and predetermined values, according to the

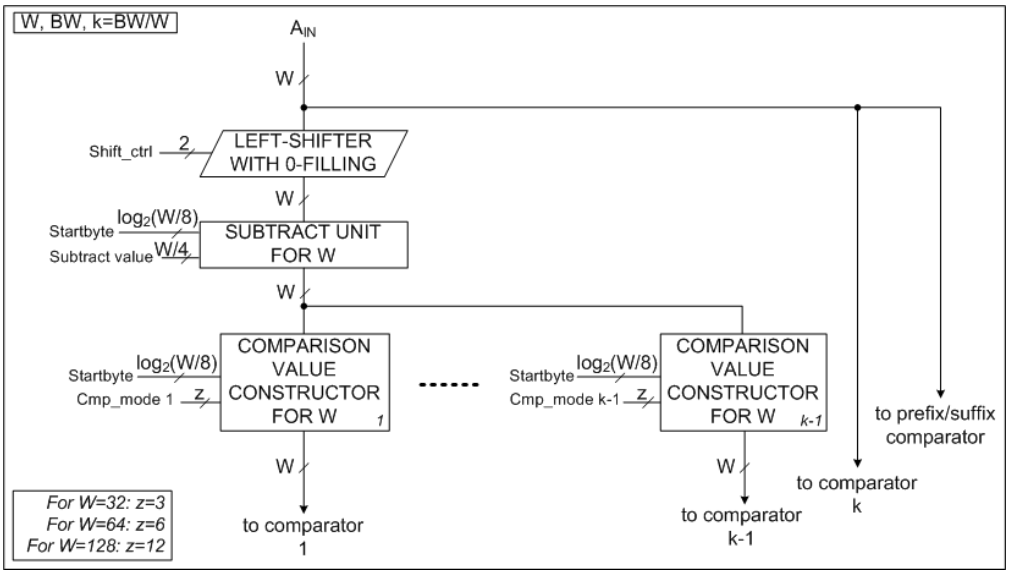


Figure 3.14: The integrated hardware design for selecting parts of addresses for a given incoming address width (W) of 32, 64 or 128 bits and a given memory bandwidth (BW) of 256, 512 or 1024 bits. The employed hardware units are the corresponding variations that were presented previously in this section.

node information (see Figure 3.15). The purpose of the comparisons is to determine which node to visit next. In the previous section the way to select and prepare the parts of A_{IN} for comparison was presented. In this section, the necessary hardware for performing the comparisons will be presented. In later sections, the hardware required for interpreting the results of the comparisons and deciding which node to visit next will be discussed.

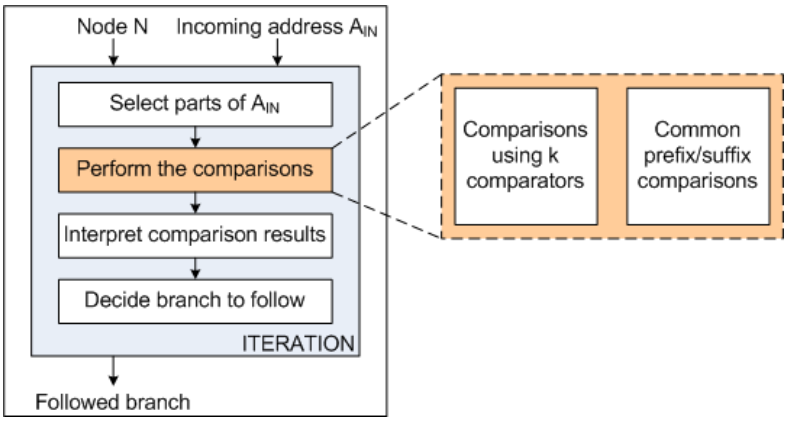


Figure 3.15: An abstract block diagram of the Range Trie iteration focusing on iteration step 2.

It has been mentioned that a Range Trie iteration uses an array of $k = BW/W$ comparators for a given incoming address width (W) and a given memory bandwidth (BW).

Each of these comparators performs a number of variable-width comparisons (except the k-th full-width comparator) according to the possible comparison widths (see Tables 3.2, 3.3 and 3.4). Since the parts of the incoming address to compare have a variable width, each comparator must be able to compare 8, $2*8$, $4*8$, ..., and W bits and output all the possible comparison results. At this point, the possible comparator modes are not used (each comparator outputs all the possibly needed results). At a later stage the comparator mode will be used to interpret the comparison results.

At the same time, there is the possibility of needed prefix/suffix comparisons (according to the node information). For that reason, apart from the k comparators, special comparators for the possible shared prefix/suffix must be designed. These comparators are different than the variable-width comparators, since a single comparison result is required and the maximum width of the comparison is $W-8$ bits. Also, to perform the correct comparison the prefix and suffix of the incoming address must be first extracted and then compared to the shared prefix and suffix..

All of the aforementioned comparators will operate in parallel. Before presenting their design, it must be noted that the result of the k-th full-width comparator will be used afterwards only when the rest $k-1$ comparators are also configured as full-width comparators. In that case there is no point for a common prefix/suffix comparison and its result is disregarded.

In the rest of this section, the two required hardware units (the variable-width comparators and the prefix/suffix comparators) will be described for all possible incoming address widths (32, 64, 128 bits). The given memory bandwidth does not affect these components, apart from the times that they are going to be used in the complete design. After presenting each unit, the way to integrate all of them will be shown, along with the means to ensure a correct operation.

3.1.2.1 The variable-width comparators

As mentioned, the variable-width comparators must be able to compare 8, $2*8$, $4*8$, ..., and W bits and output all the possible comparison results, in order to support all the possible comparator modes of Tables 3.2, 3.3 and 3.4. Since the minimum comparison width is 8-bits and all the possible comparison widths are 2^i*8, a W-bits wide comparator is designed as an array of $W/8$ 8-bits wide comparators. Every subsequent pair of 8-bits wide comparators is connected (with simple connection logic) to form a $2*8$-bits comparator, etc. For each performed comparison between values X and Y the following results must be output:

- Equal (E): If $X=Y$, then $E=1$. Otherwise, $E=0$.
- Greater equal or less (GE'/L): If $X \geq Y$, then $GE'/L=0$. Otherwise, $GE'/L=1$.

The hardware design of the variable-width comparator for incoming address width (W) of 32-bits is depicted in Figure 3.16. Note that the 8-bit comparison results are connected in an inverted tree manner to form the wider comparison results. Since the effort was to minimize the latency of the comparator, the 8-bit comparators were designed as two connected 4-bit comparators.

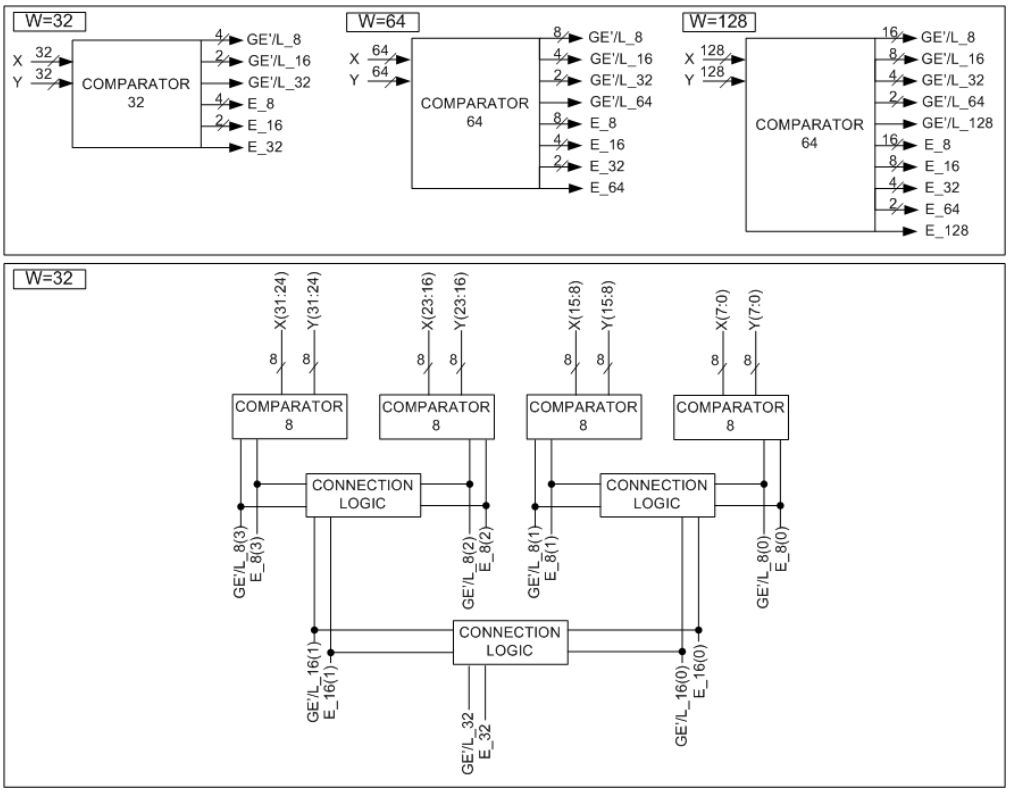


Figure 3.16: The variable-width comparator for an incoming address width (W). In the top of the figure, the block diagram of the units for W=32, 64 and 128 are depicted. In the bottom, an abstract hardware design is shown for the case of 32-bits wide incoming addresses. The 8-bits wide comparators and the connection logic are implemented as simple logic functions not presented here.

The variable width-comparators for W=64 and W=128 are designed in the same way as the one for W=32. Since the design is modular, two instances of the variable-width comparator for $W/2$, along with the same connection logic, may be used to form the W-bit comparator. This means that the inverted tree depth and the number of outputs increase, since the possible comparison results are more (see block diagrams in Figure 3.16).

3.1.2.2 The prefix/suffix comparators

The prefix/suffix comparator unit must be able to perform two comparisons on a possible shared common prefix and suffix, according to the node information. The necessary node information are: (a) the common prefix length (CP) and value (CP_{value}), (b) the common suffix length (CS) and value (CS_{value}). It has been established that CP and CS are a multiplicand of 2 and that $CP + CS \leq W - 8$. The purpose of the prefix/suffix comparator is to perform two comparisons: (a) the common prefix comparison (between the CP-bits prefix of A_{IN} and the CP_{value}) and (b) the common suffix comparison (between the CS-bits suffix of A_{IN} and the CS_{value}). Out of each comparison the GE'/L

and E results are required (as were defined before).

For the prefix/suffix comparisons, the worst-case comparison width is $W-8$ bits. For that reason two parallel $W-8$ bits wide comparators will be used. These comparators are constructed in a similar way as the aforementioned variable-width comparators. They consist of an array of $(W/8)-1$ 8-bits wide comparators connected in an inverted tree manner to form a $W-8$ bits wide comparator. Only the results of the complete comparison are output.

Since the used comparators are $W-8$ bits wide and the actual comparisons are CP and CS bits wide (where CP and CS might be less than $W-8$), the compared values must be extended to $W-8$ bits while filling the rest bits with zeros. This is achieved using a prefix mask and a suffix mask. Both masks are $W-8$ bits wide. The CP most significant bits of the prefix mask are set to ones, while the CS least significant bits of the suffix mask are set to ones. The rest bits of the masks are set to zeros. To create the masks the CP and CS values must be encoded in binary and fed to the prefix/suffix comparator unit.[1]

Applying the prefix mask on the $W-8$ most significant bits of the incoming address A_{IN} retrieves the prefix of A_{IN}, while filling the rest bits with zeros. In a similar reverse manner the suffix of A_{IN} is obtained.

A similar preparation occurs for the CP_{value} and the CS_{value}. Since $CP{+}CS \leq W{-}8$, the CP_{value} and CS_{value} are stored in a single $W-8$ bits entry (CP/CS_{values}) in order to minimize the memory requirements. Applying the prefix mask and the suffix mask separately to this entry is enough to retrieve the compatible forms of the CP_{value} and CS_{value}. At this point, all the comparison operands are correct and $W-8$ bits wide.

The hardware design of the prefix/suffix comparator unit is depicted in Figure 3.17. It consists of two decoders that generate the prefix and suffix masks based on the encoded prefix/suffix mask. The corresponding parts of A_{IN} and the CP/CS_{values} are obtained through the AND-gates, while using the prefix/suffix masks. Then the two $W-8$ comparators are used to perform the comparisons between the masked values. Although the depicted design is an example for W=32, it may easily adapted for W=64 and W=128.

3.1.2.3 Integrating the units for "performing the comparisons"

In the previous paragraphs the necessary hardware units for performing the necessary comparisons were detailed (the *variable-width comparators* and the *prefix/suffix comparators*). Since the target is a Range Trie that may have an incoming address width (W) of 32, 64 or 128 bits, all the necessary variations of the units were designed. In this section, the way to integrate and configure all of these units is detailed in order to perform successfully the comparisons between parts of the incoming address A_{IN} and the predetermined values, according to the node information.

Figure 3.18 depicts how to employ the required comparators in parallel for a given incoming address width (W) and a given memory bandwidth (BW). Just an array of

[1] Since CP and CS are always multiplicands of 2, then it suffices to encode the $CP/2$ and $CS/2$ values, as they require less bits. Therefore, CP and CS require $\lceil log_2(W-8)/2 \rceil$ bits each. I.e. the encoded mask for both CP and CS will require 8 bits for W=32, 10 bits for W=64 and 12 bits for W=128.

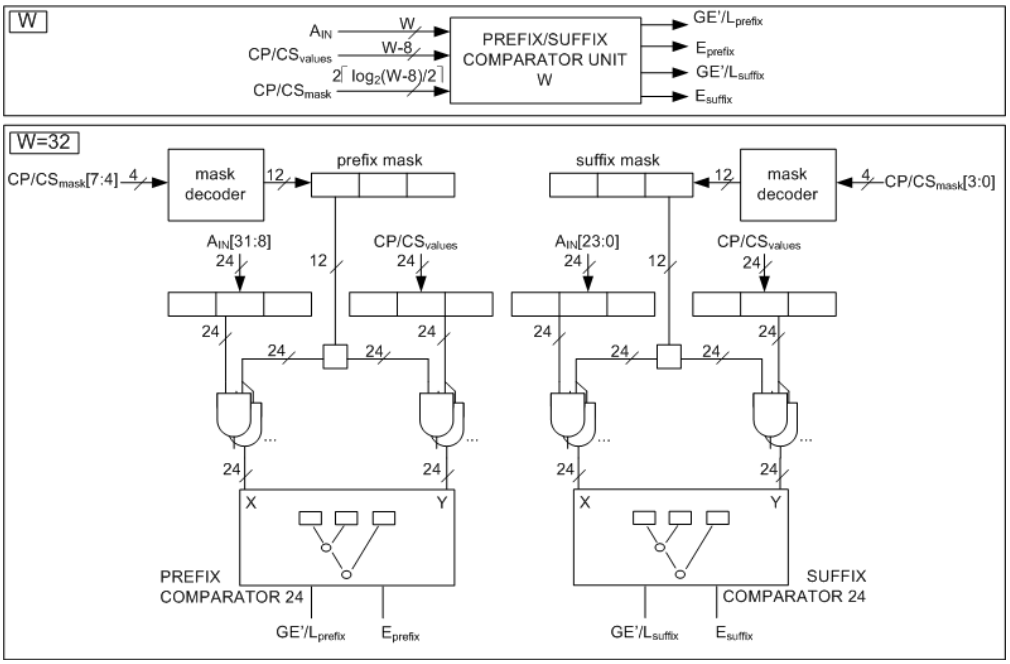


Figure 3.17: The prefix/suffix comparator unit for incoming address width of W-bits. In the top of the figure, the block diagram of the unit is depicted. In the bottom, an abstract hardware design is shown for the case of 32-bits wide incoming addresses. The mask decoders are implemented as simple priority decoders. The used comparators are implemented similarly to the variable-width comparators of Figure 3.16. Note that the incoming address prefix and suffix are connected to X of each comparator, while the common prefix and suffix are connected to Y.

$k = BW/W$ variable-width comparators for W is needed, along with a prefix/suffix comparator unit for W. Note that the prepared parts of A_{IN} are connected as the X operand of the comparators, while the predetermined comparison values are the Y operand of the comparators.

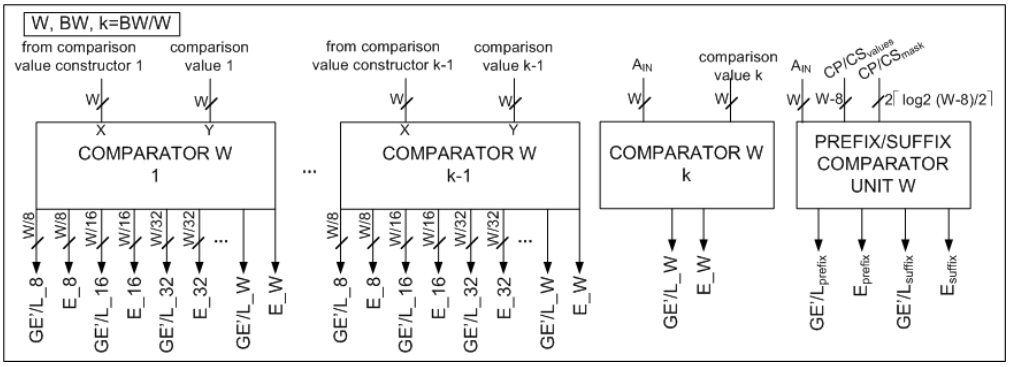


Figure 3.18: The integrated hardware design for performing the required comparisons for a given incoming address width (W) of 32, 64 or 128 bits and a given memory bandwidth (BW) of 256, 512 or 1024 bits. The employed hardware units are the corresponding variations that were presented previously in this section.

To perform the comparisons as required by the Range Trie node, a number of values need to be configured to make the units operate correctly. All of the following values are predetermined and are set before starting the operation of the Range Trie hardware.

- Comparison values i: Each of the k comparators performs a number of variable-width comparisons according to a specified comparator mode. The first operands (X) of the comparisons, the parts of A_{IN}, have been prepared accordingly by the hardware design. The second operand of the comparisons (Y), the comparison values i, must hold the respective values that will be compared according to the node information. Each comparison value i must be W-bits wide. Assuming that comparator i performs s comparisons according to its comparator mode $[w_1, ..., w_s]$, then the comparison value i is a concatenation of s comparison values, each w_j bits wide. In case a comparison of length w_j needs to be disabled, then the corresponding w_j bits of the comparison value i must be set to zeros. Comparison value k for the k-th full-width comparison must be set correctly only when this comparator is to be used, otherwise it suffices to set its comparator mode into *disabled*.

- Common prefix/suffix values (CP/CS_{values}): This is a $W - 8$ bits wide value holding the common prefix (CP bits wide) and common suffix (CS bits wide) of the Range Trie node. The CP most significant bits of CP/CS_{values} hold the common prefix value (if any). The CS least significant bits hold the common suffix value (if any). The rest bits are set to zeros.

- Common prefix/suffix encoded masks (CP/CS_{mask}): This is a $2 * \lceil log_2(W-8)/2 \rceil$ bits wide value holding the encoded prefix/suffix masks. The first half holds the binary encoding of $CP/2$, while the other half holds the binary encoding of $CS/2$, where CP and CS are the respective common prefix and suffix lengths. In case there is no shared prefix/suffix, then the respective half must be set to zeros.

To conclude, this hardware setup compares parts of A_{IN} with predetermined values and provides all the possible comparison results in the form of GE'/L and E signals. The first operand of the comparisons are the parts of A_{IN} and the second one are the predetermined values. This means that the comparison results tell if the corresponding parts of A_{IN} are greater-equal/less and equal/not-equal to the corresponding comparison values.

Not all of these comparison results will be used for deciding which node to visit in the next iteration. During the comparisons the comparator mode is not taken into account. This will happen later. Another example of this case is the use of the full-width k-th comparator and the prefix/suffix comparator unit. These two units operate in parallel all the time but their results will never be considered at the same iteration. Using the k-th comparator results may happen when all of the rest $k - 1$ comparators operate in full-width mode and thus there is no need for common prefix/suffix comparison.

All the comparisons results are now available and need to be interpreted according to the node information before deciding which node to visit next. In the next section, the hardware used for interpreting the comparison results will be presented.

3.1.3 Interpreting the comparison results

According to the Range Trie method, during an iteration a number (c) of comparisons is performed on parts of the incoming address A_{IN} against c predetermined values. Alongside, a prefix/suffix comparison may be performed on the prefix/suffix of A_{IN} against the common prefix/suffix. Depending on the results of the comparisons, the next Range Trie node to visit is determined.

Before proceeding into deciding the next node to visit, the results of the c comparisons must be interpreted into a single value (see Figure 3.19). This section describes the hardware design that interprets the comparison results into a single value. This single value represents the range that the part of A_{IN} belongs to out of the ranges defined by the c predetermined values. Assume that in an iteration the parts of A_{IN} are compared against c predetermined values. These values are the bounds of the $c+1$ ranges where the part of A_{IN} may belong to.

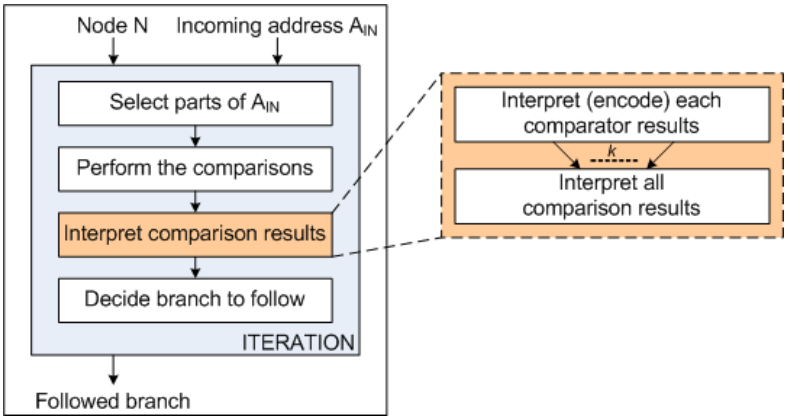


Figure 3.19: An abstract block diagram of the Range Trie iteration focusing on iteration step 3.

As described in the previous section, the c comparisons are performed in an array of $k = BW/W$ variable-width comparators, where BW is the available memory bandwidth and W is the incoming address width. Each of the k comparators outputs all the possible results for every possible allowed comparison width. For interpreting the comparison results, only the valid comparison results must be considered. The valid comparison results are obtained based on the comparator modes. I.e. assume that W=32 and the comparator mode for comparator i is [8 8 16], then we consider as valid comparison results the GE'/L and E coming out of the first 8-bit comparison, the second 8-bit comparison and the second 16-bit comparison of comparator i.

There is also the possibility of disabled comparisons within a comparator (identified by a 0-valued predetermined comparison value) and completely disabled comparators (identified by a comparator mode representing the disabled mode). If this is the case, the respective comparison results are disregarded.

The issue still remains on how to interpret the valid comparison results. Since the c predetermined values are compared in an increasing order in the k comparators, it suffices to add (or encode) the c valid GE'/L results into a single binary value ($range$).

Thus, it holds that $0 \leq range \leq c$ and it is possible to identify each of the $c+1$ matching ranges. This selected interpretation scheme means that the reported ranges are counted from right-to-left, meaning that: (a) if the result of comparison c is L, then the reported *range* is 0, (b) if the result of comparison 1 is GE, then the reported *range* is c, (c) otherwise if the result of comparison i is GE and $i+1$ is L (where $i < c$), then the reported *range* is $c - i$. This described process is a translation of Step 2 of the Range Trie method (see Section 2.2.3).

Alongside with the *range* computation it is required to interpret the E results of the valid comparisons. In a later stage it will be required to determine if the part of A_{IN} is equal to any of the predetermined values (for interpreting a possible common suffix match according to Range Trie Rule 3). Since it may be equal to only one of the predetermined values, it is enough to perform a logic-OR to all the valid E signals.

To conclude, the interpretation of the comparator results narrows down to (a) adding the *valid* GE'/L results and (b) performing a logic-OR of the *valid* E results. The results are the range that the part of A_{IN} belongs to ($0 \leq range \leq c+1$) and a signal indicating if the part of A_{IN} is equal to one of the c predetermined values. During this interpretation the possible prefix/suffix comparison results are not taken into consideration.

This whole process may be considered as an encoding task performed in two levels in hardware:

- First, there is a partial encoder per comparator that computes a *partial_range* value and an *partial_equal* signal based on the respective valid comparison results of the comparator. The valid comparison results are determined based on the comparator mode and the comparison values (to identify disabled comparisons when there are 0-valued comparison values). The resulting *partial_range* will be $\lceil log_2(W/8+1) \rceil$ bits wide[2] and is the sum of the valid GE'/L results of the comparator, except for the k-th comparator where the *partial_equal* is 1-bit wide[3]. The resulting *partial_equal* is the logic-OR of the valid E results of the comparator.

- Afterwards, there is a unit that adds the k *partial_ranges* values into the single *range* value, while calculating the global *equal* signal as a logic-OR of the *partial_equals*. The resulting *range* will be $\lceil log_2((k-1)*(W/8)+1) \rceil$ bits wide[4].

In the rest of this section the hardware units needed for interpreting the comparison results will be presented (the *partial encoder*, the *enable unit* and the *partial encodings adder*). All the necessary details will be discussed for designing these units for all possible memory bandwidths (256, 512, 1024 bits) and incoming address widths (32, 64, 128 bits). Afterwards, the way to integrate all of these units to perform the interpretation of the results will be presented.

[2]The maximum number of comparisons performed by a single comparator is $W/8$ (when only 8-bit comparisons widths are used), resulting into $W/8+1$ possible *partial_ranges*.

[3]The k-th full-width comparator performs only one comparison.

[4]The maximum number of comparisons performed in a Range Trie iteration are $(k-1)*(W/8)$ (when all $k-1$ comparators operate using only 8-bit comparisons widths), resulting into $(k-1)*(W/8)+1$ possible *ranges*.

3.1.3.1 Enable unit

It has been mentioned that c comparisons are performed using the k comparators during a Range Trie iteration. Each comparator is assigned with a comparator mode (see Tables 3.2, 3.3 and 3.4) that indicates which of its results should be considered for interpretation purposes. The c comparisons do not necessary use all the comparators. For that reason we introduced a comparator mode that disables a comparator completely (actually the comparator mode indicates that). Furthermore, there is the possibility that comparisons within a comparator are not utilized. The purpose of the *enable unit* is to tackle with this case and provide the information on whether a comparison is disabled within a comparator.

One solution would be to just add new comparator modes that indicate *disabled comparisons within a comparator*. This would result in extra bits needed for representing the comparator modes, the logic for identifying each comparator mode would become more complicated and the scalability to wider incoming address widths would be affected.

Instead we chose to hold the information of a disabled comparison within the comparison value itself. Whenever there is a comparison that is not needed, the respective bits must be set to zeros beforehand. This scheme allows for an easy detection of disabled comparisons (by performing logic-OR on the bits of the comparison values) while no extra bits are needed for indicating a disabled comparison.

We assumed that the c comparisons are assigned contiguously to the k comparators. This meant that only the right-most comparisons within a comparator may be disabled. I.e. for W=32, the following comparator modes are possible now: [16 X], [16 8 X], [8 X X X], [8 8 X X], [8 8 8 X], where X denotes a disabled comparison. Since the comparator modes must follow the ones in Table 3.2, the respective comparator modes are set to: [16 16], [16 8 8], [8 8 8 8], [8 8 8 8], [8 8 8 8]. At the same time the 16, 8, 24, 16, 8 least significant bits respectively of the comparison value are set to zeros.

As mentioned, the purpose of the enable unit is to identify if a comparison is disabled by performing a logic-OR on the respective bits of the comparison value. Figure 3.20 depicts the hardware design of the enable unit for W=32. This unit determines if the 8-bit comparisons or the second 16-bit comparison are enabled. The inspection of the 8 most significant bits of the comparison value is not needed for W=32 but it is present to assist the construction of the enable units for wider incoming address widths.

For W=64, an array of 2 enable units for W=32 will be used. As the comparator modes for W=64 are a concatenation of the comparator modes for W=32, the comparator modes with disabled comparisons must be set as for W=32. For that reason an extra rule was added: a disabled 32-bit wide comparison within a 64-bit comparator is translated into a [8 8 8 8] comparison. In a similar way the enable unit for W=128 is constructed as an array of 4 enable units for W=32.

The signals produced by the enable units will be used afterwards by the partial encoders to help determine which comparison results are valid. As there is one partial encoder employed per comparator, there is one enable unit per comparator, except the k-th full-width comparator which operates only in an enabled/disabled manner and may not have disabled comparisons within itself (since it performs just one comparison).

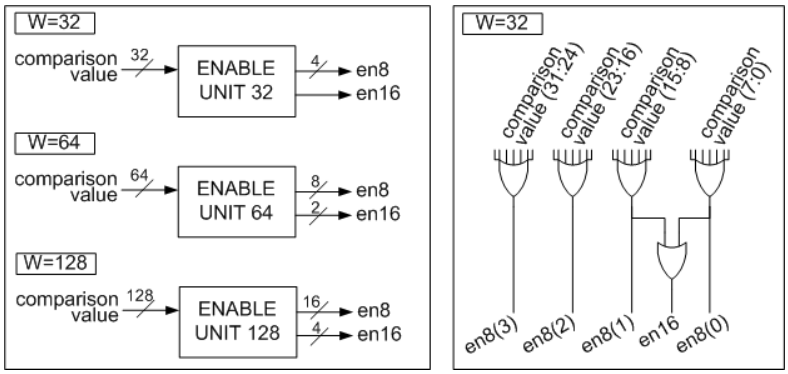


Figure 3.20: The enable unit for incoming address width of W-bits. In the left part of the figure, the block diagram of the unit is depicted for all possible W. In the right part, the hardware design is shown for the case of 32-bits wide incoming addresses.

3.1.3.2 Partial encoder

The partial encoder is responsible for the first level of interpretation of the comparison results. One partial encoder is deployed per comparator and interprets the comparison results per comparator. Based on which comparisons are valid (according to the respective comparator mode and the enable unit's output), it adds the valid GE'/L results and performs a logic-OR on the valid E results of the respective comparator. The output of the partial encoder are: (a) the *partial_range* indicating the range that the part of A_{IN} belongs to out of the ranges specified by the specific comparator comparisons and (b) the *partial_equal* indicating if the part of A_{IN} is equal to any of the values compared in the comparator.

The hardware design for the partial encoder for 32-bits wide incoming addresses is depicted in Figure 3.21. It consists of two parts, one responsible for adding the valid GE'/L results and one for performing the logic-OR of the valid E results.

First, regarding the addition of the valid GE'/L results, it must be mentioned that the maximum number of comparisons performed by a W-bit wide comparison is $W/8$. So, there are potentially $W/8$ GE'/L valid results that must be added resulting into a $\lceil log_2(W/8+1) \rceil$ bits wide number (the *partial_range*).

For the case of W=32, an adder that adds 4 bits into one 3-bit number was designed. The input to the adder must be only the valid GE'/L results. For that reason there is the logic that nullifies an invalid comparison result, based on the comparator mode and the enable unit signals. Also, since the adder adds 4 results, while the comparator results are more, groups of mutually exclusive signals were assigned to each input of the adder. Assuming that the adder adds bits a, b, c and d, the comparison results are assigned as follows:

- a is the GE'/L result of one of the following comparisons:
  - the 32-bit comparison (if the comparator mode is [32])
  - the first 16-bit comparison (if the comparator mode is [16 X])

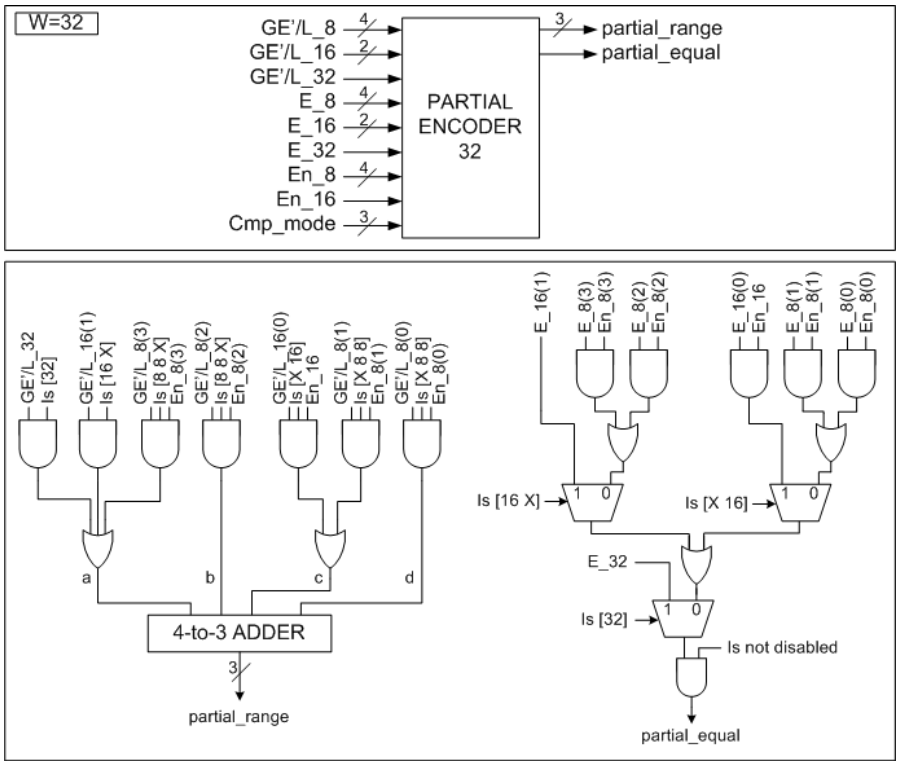


Figure 3.21: The partial encoder unit for incoming address width of 32-bits. In the top of the figure, the block diagram of the unit is depicted. In the bottom, the hardware design is shown consisting of logic for computing the partial_range and the partial_equal. The control signals detecting the comparator modes are computed using simple logic functions according to the binary representation of the comparator modes (see Table 3.2). The 4-to-3 adder is designed as three logic functions according to the logic table defining the required addition.

  - the first 8-bit comparison (if the comparator mode is [8 8 X] and the first 8-bit comparison is enabled)

- b is the GE'/L result of the second 8-bit comparison (if the comparator mode is [8 8 X] and the second 8-bit comparison is enabled)

- c is the GE'/L result of one of the following comparisons:

  - the second 16-bit comparison (if the comparator mode is [X 16])
  - the third 8-bit comparison (if the comparator mode is [X 8 8] and the third 8-bit comparison is enabled)

- d is the GE'/L result of the fourth 8-bit comparison (if the comparator mode is [X 8 8] and the fourth 8-bit comparison is enabled)

- a, b, c and d are zero when the comparator mode is set to disabled

A similar process is followed for calculating the logic-OR of the valid E signals.

Using levels of OR-gates, AND-gates and multiplexors (as depicted in Figure 3.21), the following functionality is achieved:

- If the comparator mode is set to disabled, then the *partial_equal* is set to 0.
- If the comparator mode is [32], then the *partial_equal* is the E of the 32-bit comparison, otherwise it is the logic-OR of *partial_equal_lower* and *partial_equal_upper*.
  - If the comparator mode is [16 X], then the *partial_equal_upper* is the E of the first 16-bit comparison.
    If the comparator mode is [8 8 X], then it is the logic-OR of the first and second 8-bit comparisons (if both 8-bit comparisons are enabled).
  - If the comparator mode is [X 16], then the *partial_equal_upper* is the E of the second 16-bit comparison (if the comparison is enabled).
    If the comparator mode is [X 8 8], then it is the logic-OR of the third and fourth 8-bit comparisons (if both 8-bit comparisons are enabled).

The partial encoder for W=64 and 128 is designed using the same principles. Each of these uses two instances of a partial encoder for $W/2$ (one for the upper half and one of the lower half of the comparison results) taking advantage from the way that the comparator modes were defined (see Tables 3.2, 3.3 and 3.4). Extra logic was added to form the actual *partial_range* and *partial_equal* out of the ones computed by the inner partial encoders for $W/2$.

To conclude, the partial encoder adds the valid GE'/L signals and performs a logic-OR on the valid E signals of the respective comparator to form the *partial_range* and *partial_equal* values. Such a partial encoder is deployed per each used comparator, except for the full-width k-th comparator where only one comparison is performed and the computation of its 1-bit *partial_range* and *partial_equal* reduces to a simple logic function of its 1-bit results with its 1-bit comparator mode (enabled or disabled).

3.1.3.3 Partial encodings adder (Top-level encoding)

The second level of the interpretation/encoding of the comparison results is performed by the *partial encodings adder*. This unit is responsible for adding/encoding the *partial_ranges* of the k comparators into the single desired *range* value. Alongside, it performs a logic-OR of the k *partial_equal* values to form the single *equal* value. As mentioned before, the *range* indicates the range that the part of A_{IN} belongs to out of the ranges specified by all the performed comparisons and *equal* indicates if the part of A_{IN} is equal to any of the compared values.

The hardware design of the partial encodings adder consists of an *adder* and a *logic-OR*. Performing the logic-OR of the k *partial_equals* is straightforward. On the other hand, the adder design depends highly on the number of used comparators per Range Trie instance, since it has to add $k = BW/W$ *partial_ranges*, where BW is the available memory bandwidth and W is the incoming address width. In particular, since the BW may be 256, 512 or 1024 bits and W may be 32, 64 or 128 bits, k may be one of the following values 2, 4, 8, 16, 32. This implied that a different adder should be designed for each possible k to facilitate the generation of every possible Range Trie hardware

instance. Before explaining the chosen approach for the adder designs, their expected input and output must be specified.

The adder must add k *partial_ranges*, where:

- The first $k-1$ *partial_ranges* are coming out of the first $k-1$ partial encoders. These *partial_ranges* are $\lceil log_2(W/8+1) \rceil$ bits wide each.
- The last 1-bit *partial_range* of the partial encoder for the k-th comparator. This should be added only when the k-th full-width comparator is enabled and the rest $k-1$ comparators are used as full-width comparators.

The output of the adder is a single *range* which is $\lceil log_2((k-1)*(W/8)+1) \rceil$ bits wide, because the maximum number of comparisons performed in a Range Trie iteration are $(k-1)*(W/8)$ (when all $k-1$ comparators operate using only 8-bit comparisons widths), resulting into $(k-1)*(W/8)+1$ possible *ranges*.

Instead of using regular adders to perform the addition of the k *partial_ranges*, an advanced addition technique was used based on *carry-save adders* [21]. A carry-save adder tree reduces the k numbers into 2 using $O(log_2 k)$ levels of carry-save adders[5]. Then a regular addition may be done to compute the final result. This is advantageous because each carry-save adder level has a delay of just a full-adder and the final addition is narrower. On the contrary, using regular adders would result in a delay equal to propagating the result through $k-1$ slow adders.

A further optimization was posed by integrating the addition of the 1-bit k-th *partial_range* into one of the rest $k-1$ *partial_ranges* and thus requiring the addition of one less number. This was possible since the k-th *partial_range* is present only when the rest $k-1$ comparisons are full-width and thus their *partial_ranges* hold useful information only in their least significant bit.

An example depicting the operation of the carry-save adder tree designed for W=32 and BW=256 is shown in Figure 3.22. The 7 *partial_ranges* are added using 4 carry-save adder levels. The 8-th *partial_range* is integrated with one of the 7 *partial_ranges*. Finally there is a 3-bit carry-lookahead adder to add the remaining 2 numbers into the *range*.

In order to ensure the minimum number of carry-save adder levels of the required partial encoder adder (and thus minimize the adder delay), the carry-save adder trees were designed for each of the possible values of k following the same principles.

To conclude, the partial encodings adder completes the interpretation of the comparison results. It computes the *range* value (by adding the k *partia_ranges* using the carry-save adder tree and a carry-lookahead adder) and the *equal* value (by performing a logic-OR on the k *partial_equals*). An example design for W=32 and BW=256 is depicted in Figure 3.23.

3.1.3.4 Integrating the units for "interpreting the comparisons results"

In the previous paragraphs the hardware design of the units for interpreting the comparison results was detailed. These units (the enable unit, the partial encoder and the

[5] A W-bit wide carry-save adder is an array of W full-adders. It adds 3 W-bit numbers resulting into 2 W-bit numbers which are the sum and carry outputs of the full adders.

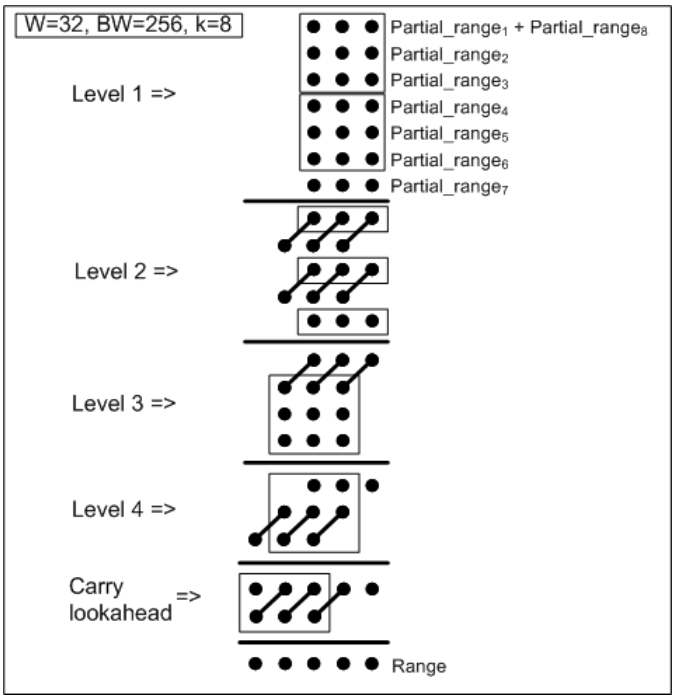


Figure 3.22: The operation of the carry-save adder tree used in the partial encodings adder unit for incoming address width of 32-bits and available memory bandwidth of 256-bits. In such a case there are $k = 8$ comparators employed in the iteration requiring for the addition of 8 partial_ranges. There are 4 levels of carry-save adders. Each carry-save adder is denoted by a box that adds the three numbers in the box into two numbers (connected with edges). In the final level there is a carry-lookahead adder to add 3-bits of the level 4 results. The operation is depicted using the dot-notation, meaning each bit is represented as a dot. To translate this diagram into hardware. Note that the 1st partial_range is integrated with the 8-th partial_range using simple logic.

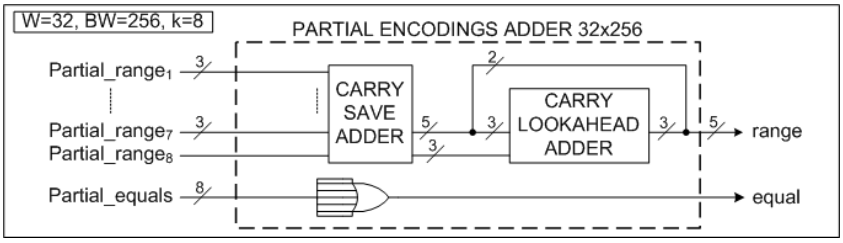


Figure 3.23: The partial encodings adder unit for incoming address width of 32-bits and available memory bandwidth of 256-bits. In such a case there are $k = 8$ comparators employed in the iteration requiring for the addition of 8 partial_ranges and the logic-OR of 8 partial_equals. The carry-save adder adds the 8 partial_ranges according to the designed 4-level tree of Figure 3.22. The carry-lookahead adder performs the final addition of Figure 3.22.

partial encodings adder) were designed for all possible incoming address widths (W) and available memory bandwidths (BW) to facilitate the generation of every possible Range Trie instance. In this paragraph the way to integrate all of these units will be presented.

Figure 3.24 depicts the interconnection between the units. There is an array of $k-1$ enable units that compute which comparisons within a comparator are enabled. There is no need for a k-th enable unit, sine the k-th full-width comparator operates in a disabled/enabled fashion determined directly by its comparator mode. Afterwards, there is an array of k partial encoders that are fed by the k comparator results. Each partial encoder computes a *partial_range* and a *partial_equal* based on the respective comparator mode and the enable unit's results. Finally, all the *partial_ranges* and *partial_equals* are processed by the partial encodings adder to produce the final *range* and *equal* signals.

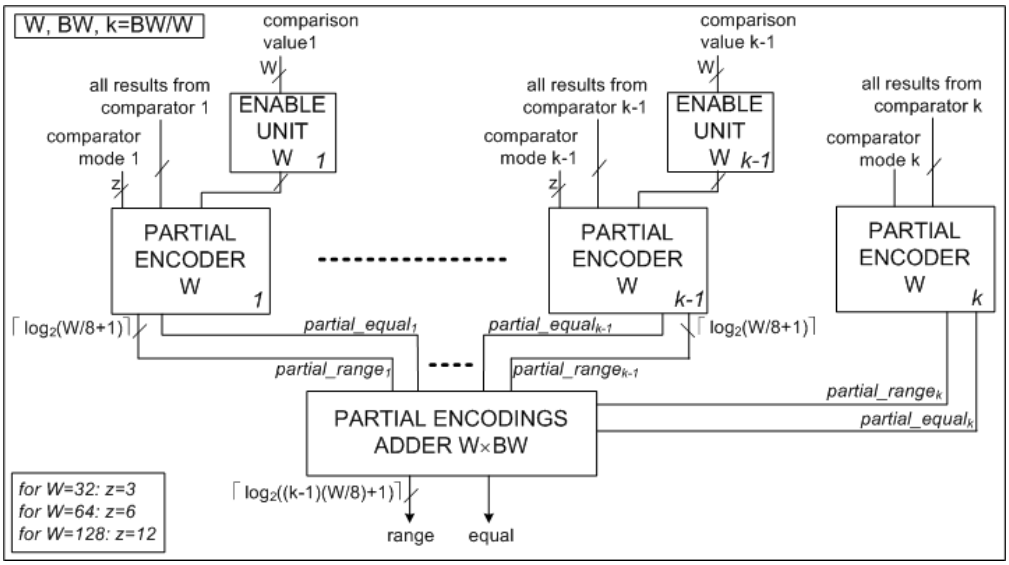


Figure 3.24: The integrated hardware design for interpreting the comparisons results for a given incoming address width (W) of 32, 64 or 128 bits and a given memory bandwidth (BW) of 256, 512 or 1024 bits. The employed hardware units are the corresponding variations that were presented previously in this section. Note that the k-th partial encoder is not the same as the rest as it implemented using simpler logic because of the simplified incoming results and comparator mode.

For the correct operation of these units, the comparator modes and the comparison values must be set accordingly. Actually, these values have already been set for using them with precedent hardware units (the comparison value constructors and the comparators). The only extra detail is to make sure that whenever a comparison within a comparator is disabled, the respective bits of the comparator value must be set to zeros and the respective comparator mode must be one of the valid ones (see Tables 3.2, 3.3 and 3.4).

To conclude, the interpretation of the comparison results considers only the valid comparison results of the c comparisons performed in an iteration and provides with (a) the *range* that the part of A_{IN} belongs to out of the $c+1$ ranges defined by the c comparisons and (b) the *equal* signal that states if the part of A_{IN} is equal to one of the c compared values. Note that the $c+1$ ranges are counted from right-to-left starting from 0 until c.

Up to this point, only the c comparison results coming out of the k comparators were taken into consideration. In the next step, where the next node to visit is decided, the possible prefix/suffix comparison results will also be considered.

3.1.4 Deciding which branch to follow

In every iteration of a Range Trie, a Range Trie node is visited, a set of comparisons is performed and the next node to visit must be decided. Up until now, only the c comparisons performed in the k comparators were considered for computing the *range* that the part of the incoming address A_{IN} belongs to out of the $c+1$ ranges defined by the c comparisons. Since there is the possibility of common prefix/suffix comparisons, their results should also be considered (see Figure 3.25) for making the final decision on the *next_range* that the part of A_{IN} belongs to out of the $c+1$ possible ones (in other words, the node to visit next). This section discusses the design details regarding choosing the next node to visit.

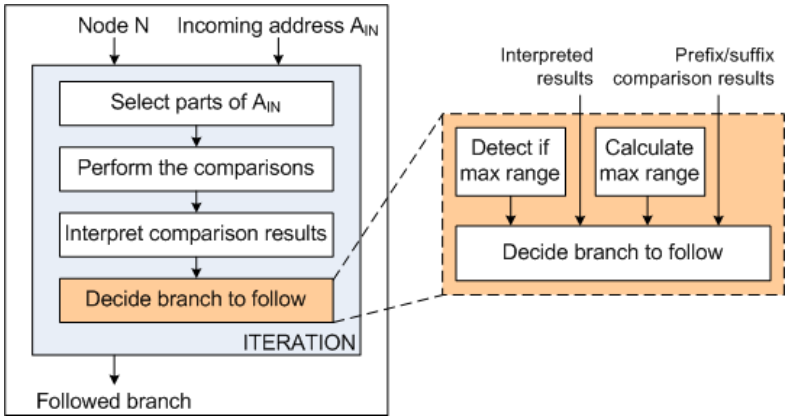


Figure 3.25: An abstract block diagram of the Range Trie iteration focusing on iteration step 4.

The process of deciding the *next_range* based on all the comparison results was described in the Range Trie method in Section 2.2.3. Since the *next_ranges* are counted in hardware from right-to-left, some modifications were done on the decision process resulting into the following process:

- Interpreting the c comparisons results indicate a *range* to visit next.
- If there is a common prefix comparison, then:
  - If the result of the comparison is L, the *next_range* is c.
  - If the result of the comparison is G, the *next_range* is 0.
  - If the result of the comparison is E, the *next_range* is the same as *range*.
- If there is a common suffix comparison, then:
  - If the result of the comparison is GE, the *next_range* is the same as *range*.

– If the result of the comparison is L, the *next_range* is $range + 1$. If *range* is equal to c, then *next_range* is *range*.

Out of this process, it may be seen that there is a set of values that may be needed to decide the *next_range*. These are the following:

- The *range* that was computed before while interpreting the results of the c comparisons. This *range* is already available.
- The number of comparisons c that are performed during an iteration. This number will be called *max_range* from now on.
- A signal *is_max_range* indicating if the *range* is equal to c (*max_range*).
- The $range + 1$.
- The results of the common prefix/suffix comparisons and an indication if these results are valid.

Computing the *max_range***:** The simple solution for being aware of the *max_range* value would be to store it in memory, but this would result in extra memory requirements. Instead we decided to calculate it on the fly, since its computation was not residing on the critical path of the hardware.

As mentioned, the *max_range* is actually the number of comparisons performed in the k comparators. It may computed in the same way that the *range* was computed in Section 3.1.3 assuming that all the k comparators result into $GE'/L = 1$. For that reason two new hardware units were designed for every possible incoming address width (W) and available memory bandwidth (BW):

- The *max range partial encoder* unit: This is identical to the partial encoder (see paragraph 3.1.3.2) assuming that all GE'/L are set to 1. This led to a simplification of its underlying logic. A max range partial encoder is also employed per comparator and computes a *max_partial_range* based on the respective comparator mode and the results of the respective enable unit. Since the *partial_equal* is not needed for computing the *max_range*, the respective logic was removed from the max range partial encoder.
- The *max range partial encodings adder* unit: This is identical to the adder used in the partial encodings adder (see paragraph 3.1.3.3). One instance of this unit is employed to add the k *max_partial_ranges* to calculate the *max_range*. The *max_range* will be $\lceil log_2((k-1) * (W/8) + 1) \rceil$ bits wide.

Computing the *is_max_range***:** Determining whether the *range* is equal to *max_range* could be done using a comparator after the *range* is computed, but this would add extra delay to the critical path of the computations. Instead, a more simple solution would be to inspect directly the results of the comparisons, in particular the result of the left-most

comparison performed in comparator 1. If the GE'/L result of this comparison is 1, then the *range* is equal to c (max_range).

To perform this check, a new unit was designed for every possible W that inspects the correct GE'/L result of the left-most comparison. This unit, called *max range detect* unit, is employed only once per iteration. Based on the comparator mode of comparator 1 and the results of comparator 1, it decides whether *range* is equal to max_range. I.e., for W=32, if the comparator mode of comparator 1 is [32] it inspects if the GE'/L result of the 32-bit comparison is 1; if the comparator mode is [16 X], it inspects the GE'/L result of the first 16-bit comparison; if the comparator mode is [8 8 X], it inspects the GE'/L result of the first 8-bit comparison.

Validating the prefix/suffix comparison results: To decide whether the value of the $next\_range$ will be *range* or 0 or max_range or $range + 1$, the prefix/suffix comparison results are required. These are directly accessible from the prefix/suffix comparator unit. Furthermore, to make sure that these results are valid the prefix/suffix mask must be inspected. The prefix/suffix mask holds a binary representation of the lengths of the possible shared prefix/suffix. In case the length of the prefix (or suffix) is zero (determined by a logic-OR on the respective half of the prefix/suffix mask), then the prefix (or suffix) comparison results should not be considered for deciding the $next\_range$.

Computing $range + 1$: To compute $range + 1$ an incrementor was designed using the Ling's approach to fast addition [21] resulting into an incrementor that requires three levels of logic gates. The designed incrementor outputs either *range* or $range + 1$ (if the common suffix comparison result is L and $range \neq max\_range$).

The hardware design integrating all the required units for computing the $next\_range$ is depicted in Figure 3.26. Note that *range*, max_range, is_max_range, the prefix/suffix comparisons results and the prefix/suffix mask are connected to the *next range unit*, where the decision is made according to the Range Trie method.

To conclude, the hardware design that was presented in this section calculates the $next\_range$ that the part of A_{IN} belongs to while taking into account all the comparison results (including the common prefix/suffix comparisons). This $next\_range$ actually indicates which branch of the current node to follow and, subsequently, which node of the next Range Trie level to visit in the next iteration. This decision is taken according to the Range Trie algorithm and rules (as presented in Section 2.2). This step concludes the Range Trie iteration tasks. What needs to happen afterwards is to calculate the location that the Range Trie node resides in the memory hierarchy. This will be explained later in Sections 3.2 and 3.3.

3.1.5 Top-level iteration module

In the previous four sections, the necessary hardware units for performing a Range Trie iteration were detailed. Each unit was designed for all possible incoming address widths

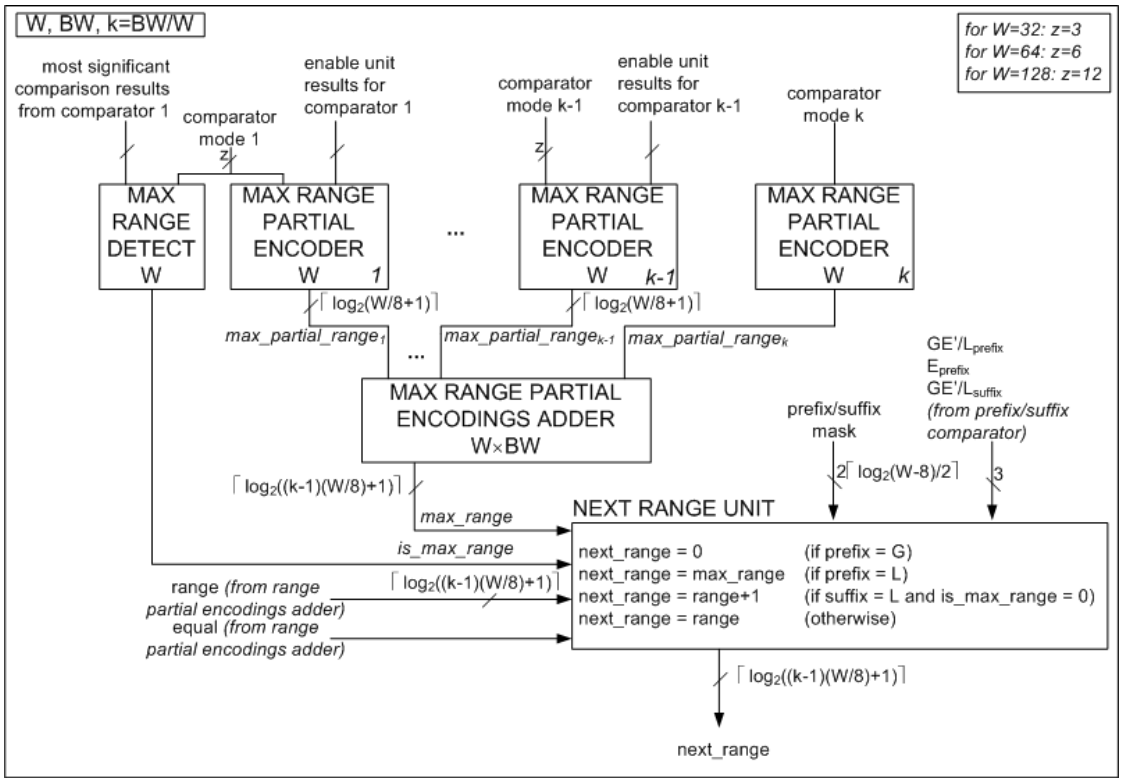


Figure 3.26: The integrated hardware design for deciding the next range value according to the results of all the comparisons (including the prefix/suffix comparisons) for a given incoming address width (W) of 32, 64 or 128 bits and a given memory bandwidth (BW) of 256, 512 or 1024 bits. The next range unit consists of multiplexors and an incrementor to choose the correct value of next_range. The prefix/suffix comparison results are considered only if they are valid (the common prefix/suffix length are not zero). Note that the max range partial encoder unit for the k-th comparator is reduced to passing the comparator mode of the k-th comparator directly to the max range partial encodings adder.

(32, 64 and 128 bits) and available memory bandwidths (256, 512 and 1024 bits). This section concludes the design of the *Range Trie iteration module* by integrating all the units and by formulating the top-level module.

The complete Range Trie iteration module performs the following operation: given an incoming address A_{IN} and a Range Trie node (represented as a set of values), it compares parts of A_{IN} according to the node information and computes the range that A_{IN} belongs to out of the ranges defined in the node. The answer that the iteration module provides is actually the outgoing branch of the current node that must be followed for the next iteration of the Range Trie method. The branches are counted from right-to-left starting from 0 until c (the number of comparisons performed in the current iteration).

Figure 3.27 depicts the top-level iteration module to be used in a Range Trie design with a given incoming address width (W) and a given available memory bandwidth (BW). These two parameters affect the underlying design of the iteration module and the size of the values required to define the iteration. Figure 3.28 shows the integrated

design of all the units needed to perform a Range Trie iteration.

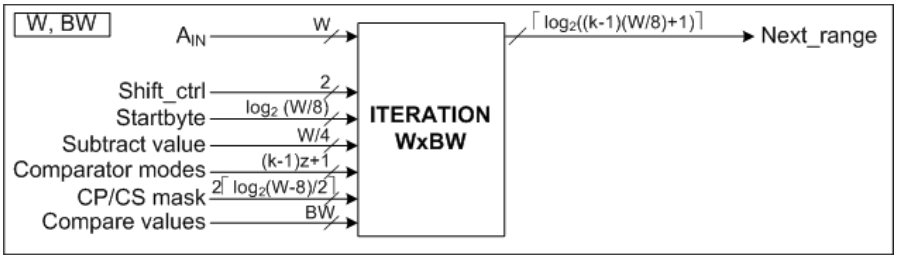


Figure 3.27: The top-level Range Trie iteration module for a given incoming address width (W) and a given available memory bandwidth (BW). The design of the iteration is depicted in Figure 3.28 as an integration of all the units presented in the previous sections.

Apart from the W-bits wide incoming address A_{IN}, the Range Trie iteration hardware is driven by the following values that define which parts of A_{IN} to compare in an iteration against which values:

- Shift_ctrl (2-bits wide) and start_byte ($log_2(W/8)$ bits wide): Based on the possible common prefix length (CP), common suffix length (CS) and the existence of a required subtraction, the shift_ctrl and start_byte determine the position within the A_{IN} that the part to select for comparison starts.
- Subtract value ($W/4$ bits wide): In case an address alignment must be performed, the subtract value is set as the 2's-complement of the value to be subtracted from the selected part of A_{IN}.
- The common prefix/suffix mask ($2 * \lceil log_2(W-8)/2 \rceil$ bits wide): The first half is the binary representation of $CP/2$, while the second half is the representation of $CS/2$.
- The comparator modes for the $k = BW/W$ comparators consisting of:
  - The $k-1$ comparator modes that determine the widths of the comparisons performed in each of the $k-1$ comparators. The allowed comparator modes may be seen in Tables 3.2, 3.3 and 3.4. The width of each comparator mode is 3-bits (for W=32), 6-bits (for W=64) or 12-bits (for W=128). In case a whole comparator is not used, then the comparator mode must be set to disabled. In case there are disabled comparisons within a comparator, a valid comparator mode according to the tables must still be used. The comparisons must be performed starting from the left-most comparator 1.
  - The k-th comparator mode that determines if the k-th comparator is enabled or disabled. The k-th comparator performs only full-width comparisons and it may be used only when all other comparators are configured also as full-width comparators.
- The comparison values for the $k = BW/W$ comparators (BW bits wide): As the comparator modes defined the widths of the comparisons to be performed per

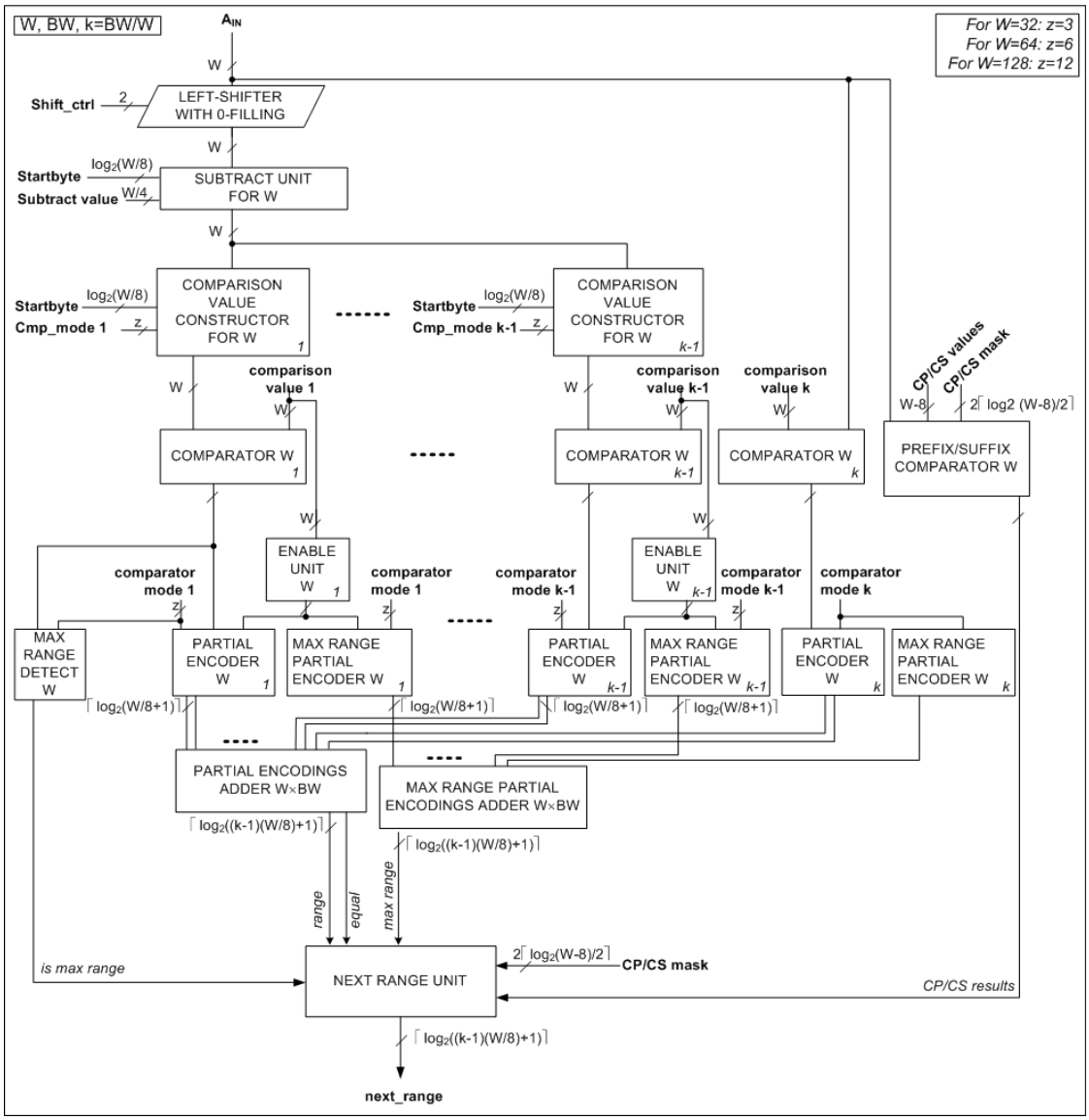


Figure 3.28: The complete hardware design of the Range Trie iteration for a given incoming address width (W) and a given available memory bandwidth (BW). The corresponding variations of the units (presented in the previous sections) are selected and connected to form the complete iteration hardware.

comparator, the comparison values must be set accordingly to indicate against which value to compare the selected parts of A_{IN}. In case a comparison within a comparator is disabled, then the respective bits of the comparison value must be set to zeros. Note that the comparison values are aa long as the available memory bandwidth. The first $(k-1)*W$ correspond to the values to be compared in the $k-1$ comparators. The W least significant bits hold either the value to be compared in the k-th comparator (only if it used), either the $W-8$ bits wide

common prefix/suffix to be compared in the prefix/suffix comparator.

All of these values actually represent the node information transformed into a data structure that is compatible with our Range Trie iteration hardware design. Such data structures are going to be stored and retrieved from the adjacent memory structure, whenever we want to visit a Range Trie node and perform an iteration. More on these will be presented in later sections (Sections 3.2 and 3.3).

This section concluded the Range Trie iteration hardware design by integrating all the required units for all possible values of W and BW. During the design process, the target was a fast, efficient and scalable design in terms of the given W and BW. The underlying units were designed in a way that minimizes the required time to perform the computations within an iteration. Alongside, care was taken to minimize the information size that drive the iteration in order to maximize the utilization of the given memory bandwidth and reduce the memory requirements.

At this point it is possible to employ the Range Trie iteration unit for the purposes of designing the complete Range Trie hardware. The iteration units will be used as a mean to perform the required comparisons, according to the current node information residing in the memory structure, and decide which node to visit in the next iteration. The details regarding the organization of the memory hierarchy and the way to employ the iteration modules will be presented in the coming sections.

3.2 Storing the Range Trie nodes in memory

The Range Trie method, presented in Section 2.2, directs how to traverse the Range Trie structure in order to perform address lookup. It consists of three basic steps: (a) visit a node, (b) perform the required comparisons (as defined by the node information), (c) decide which node to visit next based on the comparison results. These three steps are repeated until reaching a leaf node that reports the matching range.

In a hardware design, the "visit a node" step is translated into retrieving from memory the node data in order to proceed with steps (b) and (c) afterwards (the range Trie iteration). In this section, all the details of storing the Range Trie data structure in memory will be discussed. In particular, the following will be presented in the following four sections: (a) how the nodes are organized into a memory structure, (b) how the memory structure is addressed, (c) how the node information are encoded into a node data structure to be stored in the memory units and (d) what is the hardware design of the employed memory units.

3.2.1 Memory organization

In this section the way that the Range Trie nodes are organized into a memory structure will be described. The main motivations behind the proposed memory organization is to minimize the used memories size both in terms of memory width and memory depth (number of memory entries) and minimize the unused memory space.

Assume a Range Trie that is L levels deep (it needs $L-1$ iterations at-most to match an address to a range). We propose a memory organization consisting of $L-1$ memory

units, each one storing the nodes of the respective Range Trie level. Outside of the Range Trie design there is also the L-th memory unit (*action array*) that stores the actions to be performed per matching range.

The memory organization scheme, that was used, stores the Range Trie nodes of a single level i contiguously to memory unit i. There are no empty memory entries between the stored nodes and, thus, the required memory depth is minimized (the required address width for addressing memory unit i is also minimized).

As an example, assume the Range Trie of Figure 3.29. This Range Trie is 4 levels deep. A range will be matched in at-most 3 iterations. So, 3 memory units will be used to store the Range Trie nodes of level 1 to 3. Also, we assumed the existence of a 4th memory unit outside of the hardware design (action array) that stores the matching ranges' actions. Each memory unit i will store the nodes of level i starting from the right-most ones towards the left-most ones. Before storing the nodes into the memory structure, the Range Trie must be transformed into a more proper format.

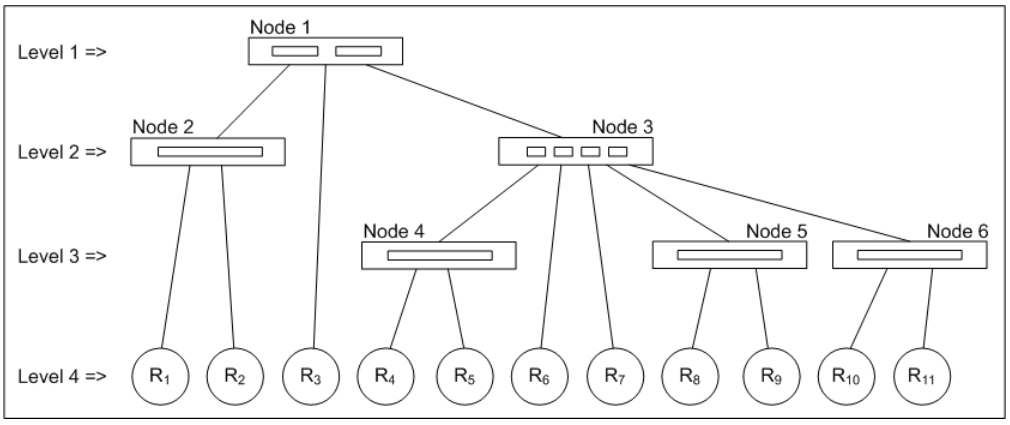


Figure 3.29: A generic Range Trie that is 4 levels deep. This will be annotated (as shown in Figure 3.30) in order to be stored in memory as depicted in Figure 3.31.

First, *extra leaf nodes* must be added wherever there is a range match that does not need L-1 iterations to be matched (see Figure 3.30). In the example Range Trie of Figure 3.29, extra leaf nodes for ranges R_1, R_2, R_3, R_6 and R_7 must be added in levels 2 and 3. These nodes will hold a pointer to memory unit 4 (action array) where the matched range's action resides. The addition of such extra leaf nodes is necessary, since we have to retrieve the correct matching range from the action array. An alternative solution would be to store the action for i.e. range R_3 directly in the level 2 memory. This would require knowledge of the actions per matching range. This solution was dismissed since we wanted the design to output the matching range address and then retrieve from the action array the corresponding match action to be performed.

Secondly, since the range nodes are stored contiguously in the memory units, it is necessary to know which are the children of a node. In particular, which is the right-most children of a node. For that reason the Range Trie nodes of Figure 3.29 are annotated with pointers to their right-most child (see Figure 3.30). Such a pointer is not needed for the root node, since the first entriy in memory unit 2 will always be its right-most

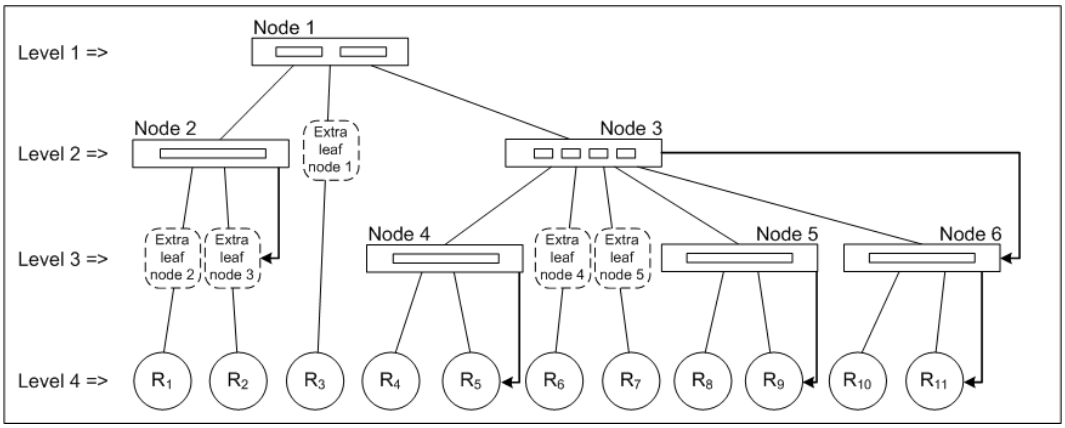


Figure 3.30: The generic Range Trie of Figure 3.29 annotated with extra leaf nodes and pointers to the right-most child of each node in order to store it in memory according to the followed memory organization.

child. This annotation with right-most children pointers must occur after adding the extra leaf nodes.

After annotating the original Range Trie in such a way to facilitate its storage in the memory units, it is possible to store it in the L memory units. As mentioned, the Range Trie nodes of each level i, will be stored contiguously in the corresponding memory unit i. One last thing to mention is that the Range Trie's top level always has just one node, the root node, so its corresponding memory unit suffices to store just one entry. Instead of using a memory unit of 1 memory entry, the equivalent memory element of a flip-flop is used. The storage of the Range Trie of Figure 3.29, after annotating it as depicted in Figure 3.30, is shown in Figure 3.31. The children pointers are also shown.

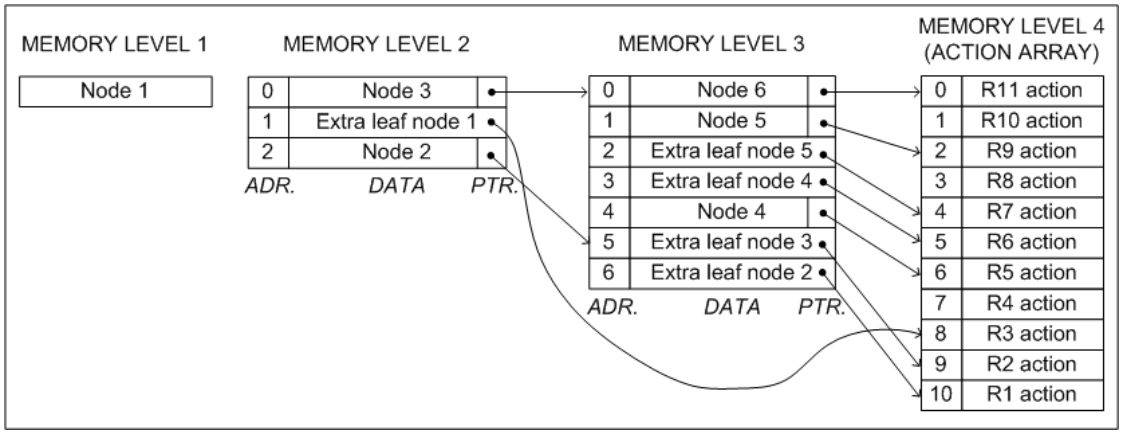


Figure 3.31: A Range Trie stored in memory according to the followed memory organization. It is based on the annotated Range Trie of Figure 3.30. The action array is shown to complete the memory organization, although it's not included in the Range Trie design.

To summarize, for a L level Range Trie that performs a range match at-most in $L-1$

iterations, L memory units M_i will be used. Extra leaf nodes must be added whenever a node branch points directly to the bottom level. Each memory unit M_i will hold the Range Trie nodes of level i starting from the right-most node towards the left-most node. Memory unit M_1 will hold only one entry, the single root node. Memory unit M_L is the action array and will hold all the matching ranges' actions. Memory unit M_L is not included in the Range Trie design, but the output of the design may be used to address it.

Since each Range Trie level holds more nodes than its previous one, the depths of the memory units increase from level i to level $i+1$. Since the nodes are stored contiguously, then the number of empty memory entries is minimized. If the number of nodes and extra leaf nodes in level i is N_i, then the number of entries in memory unit M_i will be $2^{\lceil log_2 N_i \rceil}$ resulting in a needed address of $\lceil log_2 N_i \rceil$ bits and $2^{\lceil log_2 N_i \rceil} - N_i$ empty memory entries.

An alternative memory organization solution would be to get rid of the children pointers and store nodes in a non-contiguous fashion. This would mean that nodes should be always stored in fixed locations resulting in a simpler addressing scheme. This memory organization would be equivalent to the used one, if the Range Trie was completely balanced and each node had the maximum possible outgoing branches. It is evident that such a memory organization could lead to vast amounts of unused memory entries. As a simple example of this, assume that the Range Trie of Figure 3.30 has a maximum branching factor of 5. If nodes are to be stored in a non-contiguous way, then memory unit M_1 should store 1 entry, M_2 $2^{\lceil log_2 5 \rceil} = 8$ entries, M_3 $2^{\lceil log_2 25 \rceil} = 32$ entries and M_4 $2^{\lceil log_2 125 \rceil} = 128$ entries. On the contrary, the used scheme required 1, 4, 8, 16 entries per memory unit respectively. The benefit of such a memory organization is a trivial addressing scheme, as it will be explained in the next section, at the expense of significantly bigger needed memory elements.

3.2.2 Memory addressing scheme

In the previous section, the way to store a Range Trie into the chosen memory organization was described. This organization was chosen since it minimizes the memory sizes and the unused memory area. In this section the way to address the proposed memory organization will be presented.

Assume a Range Trie node N that that has k children nodes N_j (see Figure 3.32(a)) and is stored in the memory according to the chosen memory organization (see Figure 3.32(b)).

It can be seen that the right-most child N_k of node N will reside in the next memory level in the memory entry with address equal to the *pointer* stored in node N. The second-to-right child N_{k-1} of node N will reside in memory entry $pointer + 1$, etc. The left-most child N_1 of node N will reside in memory entry $pointer + k - 1$. It is evident that in order to calculate the address of the next memory entry to be retrieved (in other words, the next node to visit) an addition should be performed. Assume that the k outgoing branches of node N are counted from right to left starting from 0. Then, if branch b is followed ($0 \leq b \leq k-1$), the next node N_{k-b} to visit is stored in the memory entry $pointer + b$ of the next memory level. This means that in every iteration of the

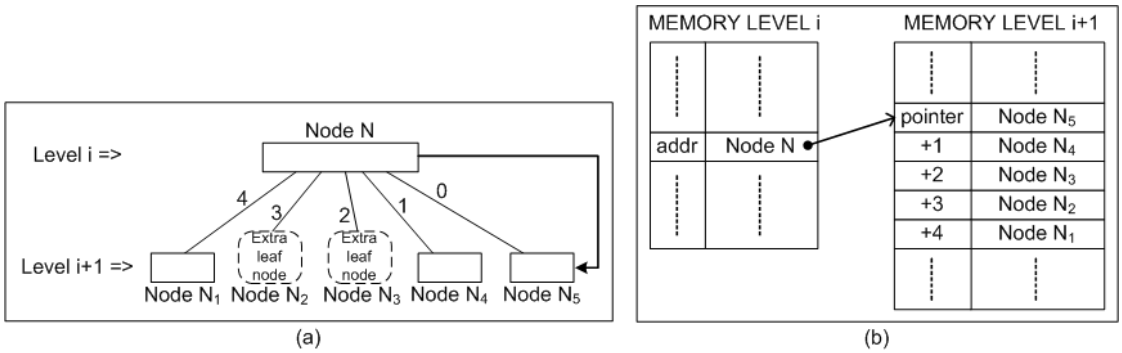


Figure 3.32: A Range Trie node N (of Range Trie level i) and its children N_j (of Range Trie level $i+1$) organized in the memory units i and $i+1$ (according to Section 3.2.1). In (a), N is annotated with extra leaf nodes and the pointer to its right-most child. Note that the outgoing branches of N are counted from right to left starting from 0. In (b), the way that N and its children are stored in the memory units for levels i and $i+1$ is shown.

Range Trie method an addition must be performed between the *pointer* of the current node and the number of followed branch b (offset).

Offset adder: For the purposes of calculating the location of the next node to visit, the *offset adder* was designed. The offset adder adds the number of the followed branch ($offset$) to the *pointer* to the next memory level. The $offset$ is the output of the iteration unit that operates according to the current node information and it is $\lceil log_2((k-1) * (BW/8) + 1) \rceil$ bits wide, since it depends on the memory bandwidth (BW) and incoming address width (W) parameters of the current Range Trie instance. The *pointer* to the right-most child of the current node depends on the next memory level depth and is as wide as the next memory level address.

We tried to minimize the cost in time of performing this addition by using advanced adder techniques. Figure 3.33 depicts the designed offset adder. Assume that the width of $offset$ is c and the width of *pointer* is d. The offset adder is designed as a two level carry-select adder taking advantage from the fact that $c \leq d$. The first level of the carry-select adder is a c-bit wide fast carry lookahead adder that adds $offset$ to the c least significant bits of *pointer*. The second level must add the carry-out of the first level to the $d - c$ most significant bits of *pointer*. This indicated that the second level of the offset adder may be designed using a fast incrementor.

Since the design of the offset adder depends on BW and W, such an adder should be designed for every combination of BW and W. Furthermore, the offset adder depends on the address width of the next memory level, which is not known in advance. That made it impossible to design beforehand all the possible offset adder variations for every possible parameters. Instead, a script was written to automatically generate the required offset adders for a given Range Trie instance.

The aforementioned offset addition could be avoided, if we used the alternative memory organization scheme (described in 3.2.1) where the nodes are stored in a non-contiguous fashion. In that particular case, the pointer would not be needed and to cal-

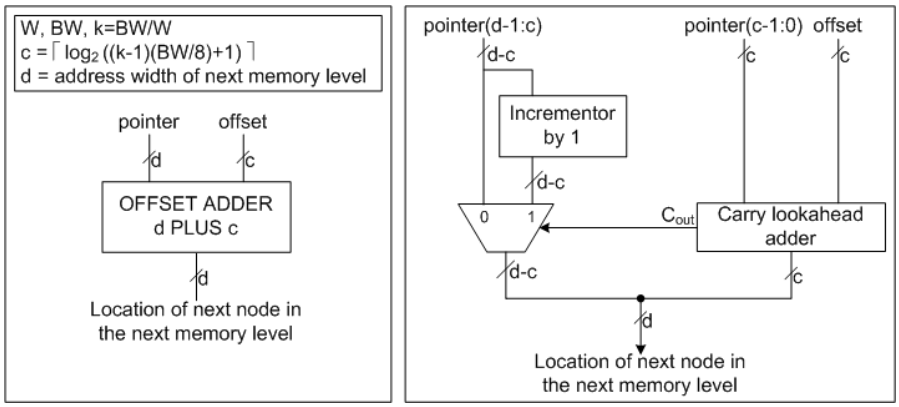


Figure 3.33: The offset adder that adds an offset (c-bits wide) to a pointer (d-bits wide) in order to compute the location of the next node to visit in the next memory level. The block diagram of the unit is depicted along with its abstract hardware design.

culate the address of the next node would narrow down to appending the offset (branch to be taken) at the end of the current node address. Although such an addressing scheme is an attractive solution, it was dismissed immediately since the required memory organization would result into wider memory addresses and thus prohibitive memory sizes with low utilization.

3.2.3 Range Trie node data structure

In the previous sections it was explained how the Range Trie nodes are organized into the memory units, along with the required addressing scheme to retrieve the next node to visit. In this section, the way to represent a Range Trie node into a data structure to be stored in the memory entries will be described.

Throughout section 3.1 the signals that are needed to drive the correct operation of an iteration were defined, according to the visited node information. All these signals form the data structure that must be stored in a corresponding memory entry of the memory hierarchy. While defining the memory organization, an extra value was added to the data structure (the *pointer* to the right-most child of the current node).

The node data structure fields may be classified into the three following parts (also shown in Figure 3.34(a)):

- Compare values: These are the values to be compared against the parts of the incoming address A_{IN}. During the development of the Range Trie method the effort was on performing as many comparisons as possible for a given memory bandwidth (BW). The purpose of having the memory bandwidth parameter in the Range Trie design is to investigate the scalability of the Range Trie design for a set of possible available memory bandwidths. This indicated that the compare values part should be BW-bits wide.

- Control values: These are the control signals that dictate what comparisons to perform on which parts of A_{IN}. It consists of the following signals, as de-

fined throughout section 3.1: $shift\_ctrl$, $start\_byte$, $subtract\_value$, CP/CS_{mask}, $comparator\_modes_i$. The size of these signals depends on both BW and W parameters. During the design of the iteration hardware, the effort was on minimizing the size of these control signals.

- Pointer value: This is the pointer for the right-most child of the node (see Section 3.2.2) and it depends on the depth of the next level memory. The pointer widths are not known in advanced and depend on the given Range Trie structure.

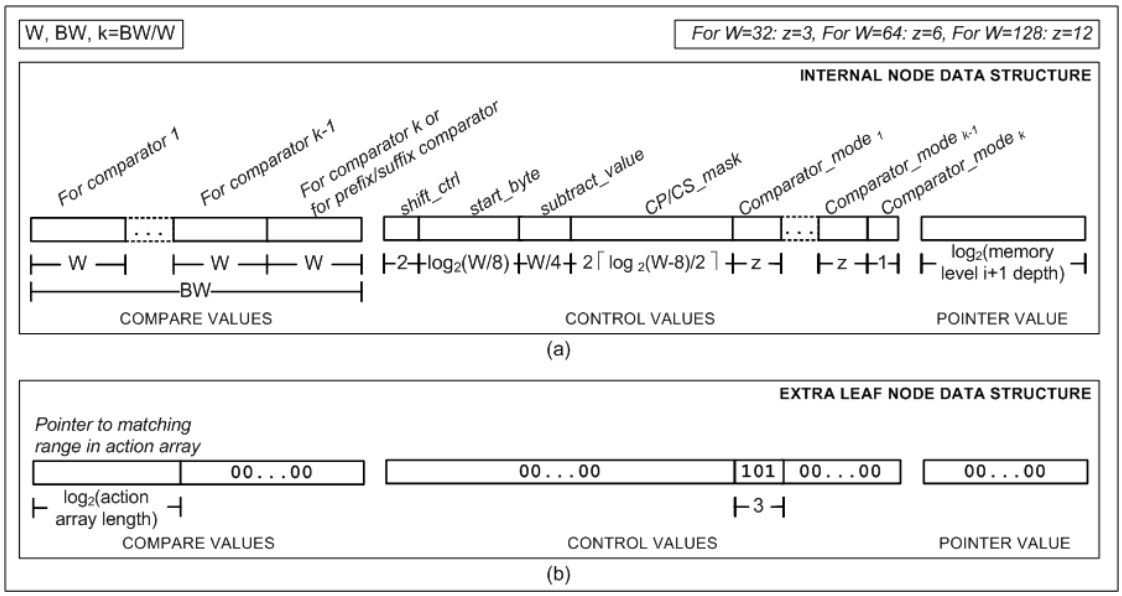


Figure 3.34: The data structure of (a) an internal node and (b) an extra leaf node residing in memory level i for a given memory bandwidth (BW) and incoming address width (W). The sizes of each field were determined during the design of the iteration hardware (see Section 3.1). Both data structures occupy the same number of bits. Note that the size of the pointer value requires knowledge for the depth of the next level memory unit. Such a pointer is not required for the root node's representation.

It is evident that the node data structure size depends on the values of W, BW and the next memory level depth. Table 3.5 shows the resulting data structure sizes for the possible values of BW and W, assuming a given pointer size w_{i+1}, since the latter is not known in advance.

The node data structure that was just presented was for the internal Range Trie nodes. It is also needed to store the *extra leaf nodes* in the memory structure. Since both internal nodes and extra leaf nodes are stored in the same memory units, then both node types are going to occupy the same number of bits (see Figure 3.34(b)), even though the extra leaf nodes require less bits. The necessary information to be stored for the extra leaf nodes are:

- A value indicating that it is a leaf node: The 3-bit value "101" must be stored in the 3 most significant bits of $comparator\_mode_1$ field. That value was chosen,

Table 3.5: The node data structure sizes for all possible values of available memory bandwidth (BW) and incoming address width (W). The sizes are in bits and are represented as a sum of the three parts' sizes: compare + control + pointer value sizes. The next memory level $i+1$ is assumed to be addressed using w_{i+1} bits.

		BW		
		256	**512**	**1024**
W	**32**	256+42+w_{i+1}	512+66+w_{i+1}	1024+114+w_{i+1}
	64	256+50+w_{i+1}	512+74+w_{i+1}	1024+122+w_{i+1}
	128	256+63+w_{i+1}	512+87+w_{i+1}	1024+135+w_{i+1}

since it does not clash with the existing binary representations of the comparator modes (see Tables 3.2, 3.3 and 3.4).

- The pointer to the action array: This value is as wide as the predefined output width of the Range Trie. It is stored in the most significant bits of the compare values part and points to the the location in the action array that the matching range resides.

To conclude, the memory structure was defined. Starting from a Range Trie structure, it is possible to configure the memory structure in a way to facilitate the operation of the Range Trie design. To sum up, in every iteration, a node is retrieved from the memory structure, the comparisons are performed according to the node data structure (compare values and control values) and the location of the next node to visit is computed (using the comparison results and the pointer value). Before employing the memory structure along with the iteration hardware to form the complete pipeline design that performs the Range Trie method, some hardware design details of the used memory units will be discussed in the following section.

3.2.4 Memory units' hardware details

In the previous three sections, the way to organize, address and store the Range Trie nodes into a number of memory units was described. This section concludes the discussion on the memory related issues by describing some hardware design details of the employed memory units.

As mentioned before, one of the parameters of the Range Trie design is the available memory bandwidth (BW). This parameter serves as a design limitation to dictate the maximum possible bandwidth of the used memory units and to evaluate different Range Trie instances on different memory organizations. In the previous section, where the node data structure was described, it may be seen that the compare values part is already BW-bits wide. The addition of the control values and pointer value parts leads to the need for a bandwidth larger than BW bits.

To overcome this limitation, we designed each memory unit as a set of three parallel memory subunits: one for storing the compare values, one for the control values and

one for the pointers. These three subunits are addressed by the same address. This separation also allows for manipulating the three value categories independently of each other. It is possible now to change a pointer without changing the respective compare values, etc. For comprehension reasons, in the rest of this thesis, the memory units will be mentioned as one unit instead of the three subunits.

Figure 3.35 depicts an example of a memory unit design. The proposed memory units allow the reading or writing of memory entries and the disabling of the memory units using the following signals:

- Write enable (WEN): This signal allows the writing of memory entries. It was added to the design in order to be able to configure the memory units with a given Range Trie structure (according to the Sections 3.2.1-3.2.3) and to allow the modification of memory entries while the Range Trie hardware is in operation. Each subunit is driven by a separate WEN, allowing for the independent manipulation of their data.

- Chip enable (CEN): This signal enables or disables the memory unit. Since there is a chance that a memory unit might not need to be accessed, then it may be disabled. When a memory unit is disabled, it does not consume power. Thus, disabling unused memory units minimizes the power dissipation by avoiding unneeded memory accesses. This ability is further utilized by the fact that the memory units actually consist of a number of smaller memory blocks. When accessing a specific memory block within a memory unit, the rest memory blocks are disabled to avoid unnecessary power dissipation.

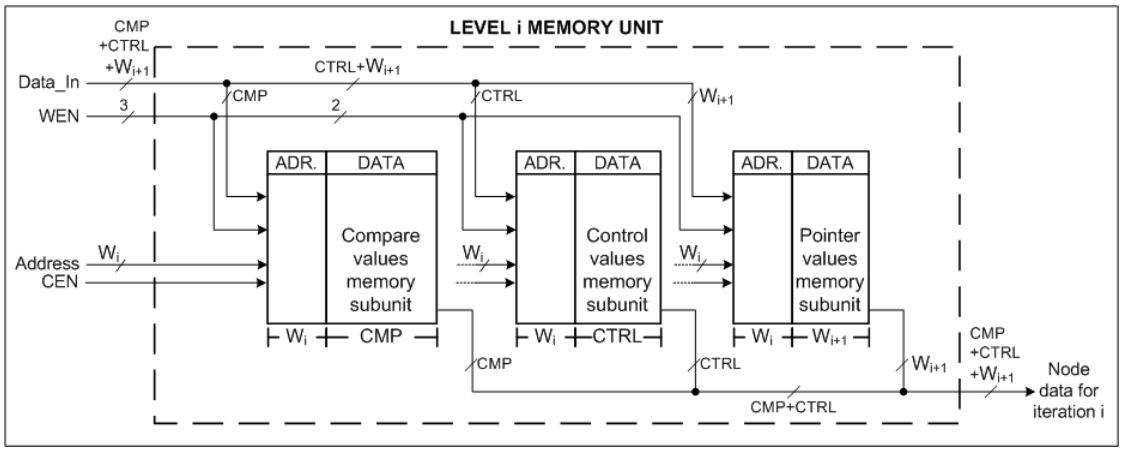


Figure 3.35: An abstract hardware design of the level i memory unit for a given memory bandwidth (BW) and incoming address width (W). Assume that level i memory unit is addressed by w_i bits and level $i+1$ memory unit by w_{i+1} bits. According to BW, W and w_{i+1}, the widths (CMP, CTRL, PTR) of each memory subunit (compare, control, pointer values) are computed (see Section 3.2.3).

To conclude, a number of memory units (as the ones described in this section) will be

employed in the complete Range Trie design, along with the iteration functional units (as described in Section 3.1). The memory units will be organized/addressed according to the chosen memory hierarchy (as described in Sections 3.2.1-3.2.2) and will hold the Range Trie nodes represented in a data structure (as explained in Section 3.2.3). In the next section, the Range Trie design will be completed by integrating the memory structure with the iteration hardware into a complete pipeline.

3.3 The complete Range Trie design

The Range Trie method (as presented in Section 2.2.3) is an iterative process consisting of three steps: (a) visit a node, (b) perform a number of comparisons (according to the node information) and (c) decide which node to visit next. This process ends when a leaf node is reached (the incoming address A_{IN} is matched to a range). In Sections 3.1 and 3.2, the *iteration functional unit* (performing steps (b) and (c)) and the chosen *memory structure* (that stores the nodes to be visited) were presented.

In this section both of these will be integrated in a pipeline fashion to form the complete Range Trie hardware design. In particular, the chosen pipeline scheme will be presented, along with its hardware design details according to the given Range Trie parameters (that were presented in the beginning of this chapter). Afterwards, the complete Range Trie top-level module and its usage will be described, concluding with some high level details of the complete Range Trie design.

3.3.1 Integrating the iteration hardware and the memory structure into a pipeline

The iterative nature of the Range Trie method indicated that the complete design should be designed as a pipeline consisting of a number of pipeline stages. This section presents the complete Range Trie design as an integration of the *iteration functional units* (see Section 3.1) and the *memory structure* (see Section 3.2) into a pipeline.

By pipelining the traversal of the Range Trie structure, a high throughput of one packet per clock cycle may be sustained. Using a pipeline allows the matching of an incoming address A_{IN} to a range per clock cycle, instead of waiting for a number of clock cycles before providing a new A_{IN}. To further increase the throughput of the design, we separated the iteration hardware and the memory accesses into two different pipeline stages. In a pipeline the clock cycle is determined by the slowest pipeline stage. Separating the iteration hardware and the memory accesses into two pipeline stages results into two quicker pipeline stages with a smaller clock delay.

Assume that the given Range Trie structure is L-levels deep, meaning that an incoming address A_{IN} is matched to a range in at-most $L-1$ iterations. The pipeline that is designed for the given Range Trie has $(L-1)*2-1$ pipeline stages (as depicted abstractly in Figure 3.36). Each iteration consists of two pipeline stages: (a) one for accessing the memory unit i (to retrieve a node from level i of the Range Trie) and (b) one for performing the computations according to the retrieved node (to decide which node to retrieve from memory unit $i+1$). Note that memory unit 1 does not have a

pipeline stage of each own, since it holds just one entry (the root node) and may be stored as a pipeline register[6].

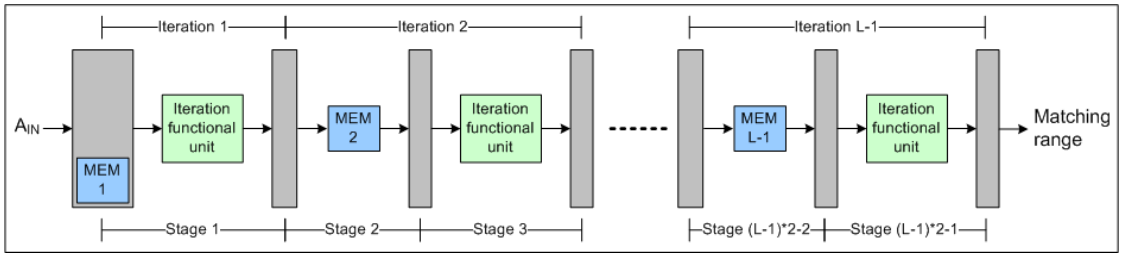


Figure 3.36: The abstract pipeline of a Range Trie design. The given Range Trie is L-levels deep and matches a range in at-most $L-1$ iterations and thus requiring $(L-1)*2-1$ pipeline stages. The pipeline stages are separated with the required pipeline registers.

There are many issues regarding the actual design of the pipeline, not evident in Figure 3.36. Figure 3.37 depicts the complete Range Trie pipeline design that also deals with the following issues:

- Matching a range in less than $L-1$ iterations: If a node retrieved from memory level i is an extra leaf node (detected by inspecting the 3 most significant bits of the comparator mode 1 field in the node data structure), then the rest pipeline stages (including the memory units) are disabled and the pointer to the action array (stored in the most significant bits of the node data structure) is propagated to the output (through the next address pipeline registers). Note that in any case the output of the pipeline is a pointer to the action array holding the action for the matched range (the action array is not included in the pipeline).

- Invoking the offset adder: If a node retrieved from memory level i is an internal Range Trie node, then in the next pipeline stage the iteration functional unit operates according to the retrieved node information. The output of the functional unit is added to the *pointer* (stored in the retrieved node) by using an offset adder. The result of the addition is the memory entry to retrieve from the next pipeline stage (memory level $i+1$). Note that this addition is not needed for the first iteration.

- Disabling the unused memory units: The memory units might need to be disabled if a matching range has already been found or there is no incoming address to match. The enable pipeline register propagates a signal that enables/disables the memory units accordingly.

- Variable number of iterations (bypassing iteration 1): In general, a Range Trie pipeline is tailored to a specific Range Trie structure. To offer an extra degree of flexibility, the ability to bypass iteration 1 was added. This reduces the number of

[6]The pipeline registers are placed between pipeline stages and their use is to propagate useful values (such as the A_{IN}) from one stage to the next.

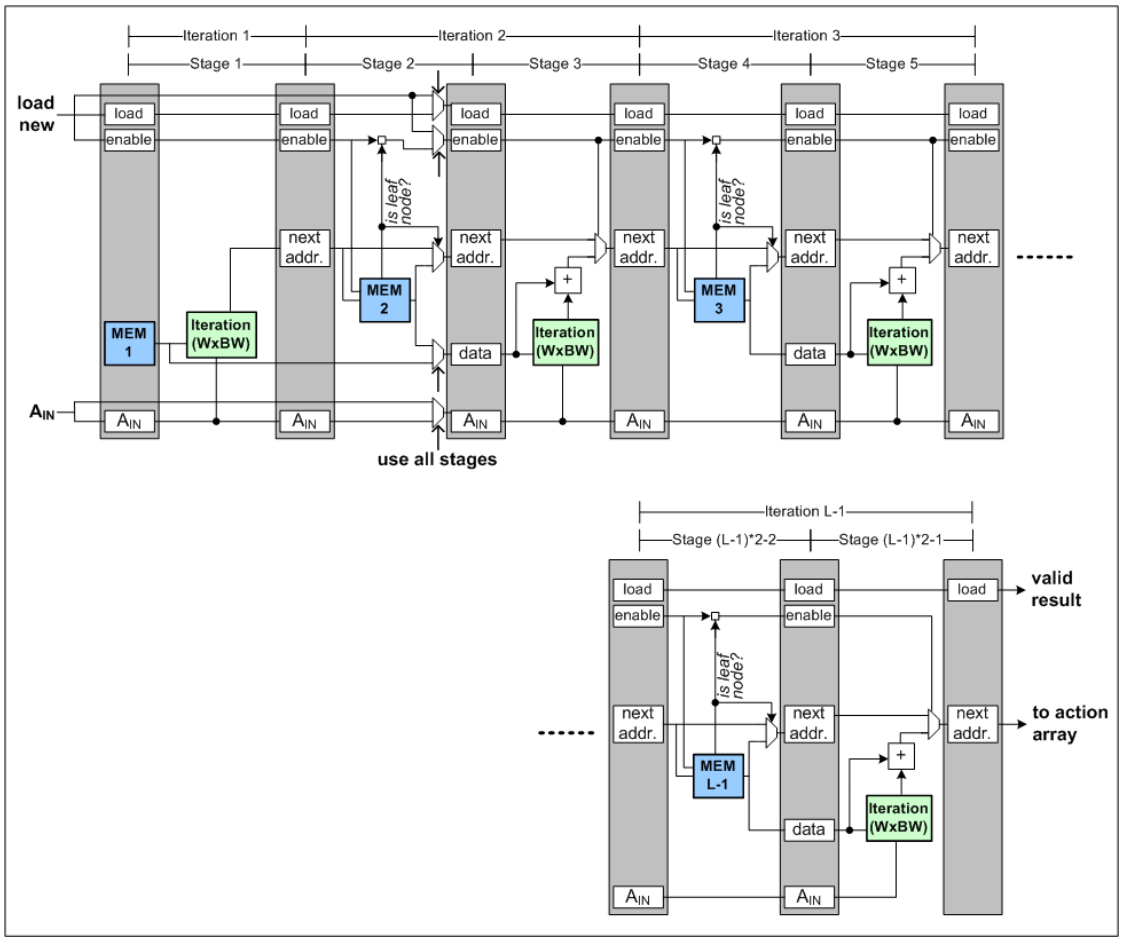


Figure 3.37: The complete Range Trie pipeline hardware design for a given set of parameters. The given Range Trie is L-levels deep and matches a range in at-most $L-1$ iterations and thus requiring $(L-1)*2-1$ pipeline stages. The pipeline stages are separated with the required pipeline registers. Note that not all the design details are depicted in the figure. I.e. the simple logic that controls the memory configuration is not shown.

pipeline stages to $(L-2)*2-1$ and may be used for at-most $L-1$ levels deep Range Tries. To support this, an array of multiplexors was added in pipeline stage 2 to control (by the *use_all_stages* signal) the input of the next pipeline registers.

- Supporting memory units writing: To allow the configuration/alteration of memory entries extra memory control signals were added that allow to write a memory entry to one of the available memories at a time.

- Range Trie handshaking signals: To offer the user of the Range Trie a complete interface during the operation of the design, the *load_new* and *valid_output* control signals were added. The first one must be set when a new incoming address is to be matched. The latter one is set whenever the output of the design is a valid one. Note that the output will be set correctly $(L-1)*2$ clock cycles after a new input.

From Figure 3.37 it may be seen how the pipeline depends on the set Range Trie instance parameters. These parameters are the incoming address width (W), the available memory bandwidth (BW), the number of iterations (L), the address width of each memory level (w_2, w_3, ..., w_{L-1}) and the output width (w_L). All of these parameters are tailored according to the given Range Trie structure that the hardware design is going to implement. Since these parameters are not known in advance, a script was created that automatically generates a Range Trie pipeline.

During the pipeline design process, the target was again an efficient, scalable, fast Range Trie design. Since the complete design has now been explained, it is possible to move up one level of abstraction and present the Range Trie top-level module in the next section.

3.3.2 The Range Trie top-level module

In the previous sections, all the details have been presented to design all possible instances of a Range Trie design. This section finalizes the Range Trie design by presenting its top-level module and the way to use it for address lookup.

Assuming an incoming address width (W) that may be 32, 64 or 128-bits and the available memory bandwidth (BW) that may be 256, 512 or 1024-bits, a Range Trie structure is constructed for a given set of R ranges using the heuristic construction methods of Section 2.2.4. The resulting Range Trie structure is the one to be translated into an equivalent hardware design. Out of the Range Trie structure it is possible to define all the parameters of the respective Range Trie design. In particular, the number of iterations (L), the address widths of each memory level (w_2, w_3, ..., w_{L-1}) and the output width ($w_L = \lceil log_2 R \rceil$). After defining the parameters, the complete Range Trie pipeline may be generated following the one presented in Section 3.3.1.

The resulting Range Trie design is represented by the top-level module depicted in Figure 3.38. This module may be used to perform address lookup on the specified Range Trie. Before starting the operation of the Range Trie hardware, the memory units must be configured to store the Range Trie node data structures according to the chosen memory organization scheme (see Section 3.2).

Using the Range Trie hardware is as simple as placing a new incoming address A_{IN} to be matched in every clock cycle, while setting the *load_new* control signal. After $(L-1)*2$ clock cycles, the Range Trie reports the range that A_{IN} belongs to out of the R given ranges. The reported range is actually the location in the action array where the matched range action resides. The action array is not included in the design, since the actions per range are not known in advance and because they depend on the system where the Range Trie is employed.

The Range Trie design may now be used as a lookup engine is every system/application that requires a form of address lookup. I.e. a backbone router may employ a Range Trie for performing the address lookup of the incoming packets' destination addresses and then decide what action to perform on the incoming packet.

This section concluded this chapter and completed the description of the Range Trie design that was the main focus of this thesis. The iteration functional units and the

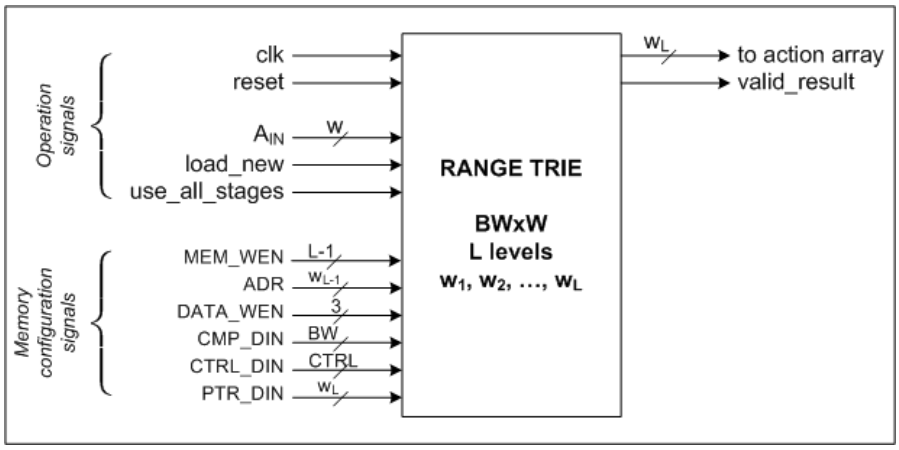


Figure 3.38: The Range Trie top-level module for a given Range Trie structure (that determines the parameter values). Before starting the operation of the Range Trie, the memory units must be configured through the memory configuration signals. The w_i denotes the address width of memory unit i. The memory unit L (action array) is not included in the design and is addressed by the Range Trie output.

memory units were integrated into forming the complete design in a pipeline fashion. The following chapters are going to focus on the evaluation of the proposed design in terms of various performance metrics.

3.4 Summary

In this chapter, the complete hardware design of the Range Trie address lookup method was presented in a bottom-up fashion. Starting from the iteration functional unit (in Section 3.1) and the chosen memory structure (in Section 3.2), a complete pipeline was formed (in Section 3.3) that implements the Range Trie address lookup method (as it was formulated in Section 2.2). All required hardware units were described that are needed for designing every possible instance of a Range Trie, according to its parameters. During the design, the effort was to exploit the Range Trie method inherit characteristics into building a fast, efficient and scalable hardware design with low memory requirements.

The Range Trie method is an iterative method, where (a) a node is visited, (b) a number of comparisons is performed on parts of the incoming address and (c) the next node to visit is decided, until a leaf node is reached and a match is reported. In Section 3.1, the hardware design of the Range Trie iteration was presented. This hardware performs the following four actions: (1) select parts of the incoming address for comparison and (possibly) perform address bound alignment, (2) perform the comparisons and (possibly) the shared prefix/suffix comparisons, (3) interpret the comparison results and (4) decide which outgoing branch of the current node to follow, according to the comparison results, following the Range Trie method's decision criteria. The required hardware units were designed to perform these actions. All required variations of the units were designed (and presented) in order to support the generation of every possible Range Trie instance. During the design process the necessary signals that control the

iteration were defined, according to the current node information. This resulted in the formulation of the way that the current node information must be represented for the correct operation of a given iteration.

In Section 3.2, the memory structure was defined where the Range Trie structure must be stored. This memory structure is used for retrieving the node information (in every iteration) stored as a specified data structure. In particular, a memory structure was defined that consists of as many memory levels as the Range Trie levels. The way to organize the Range Trie nodes into the memory levels was presented, along with the way to address the memory levels in order to retrieve the next node to visit. Finally, the node data structure was formally specified in order to store a Range Trie node into the memory hierarchy. During the specification of the memory structure the motivation was to minimize the memory requirements and maximize the memory utilization.

Finally, in Section 3.3, the complete Range Trie design was presented as an integration of the iteration hardware and the memory structure. The used approach was to pipeline the iteration hardware and the memory structure to design the iterative Range Trie method, while creating a design that offers high throughput and that is scalable in terms of the incoming address width and the number of ranges in the lookup table.

4 Evaluation

The previous chapter presented the complete Range Trie design as a pipeline of interleaving iteration stages and memory access stages. All the details were discussed on how to design every possible Range Trie instance given a set of its defining parameters. In this chapter a number of Range Trie instances are generated and synthesized for ASIC implementation (90nm and 130nm) in order to retrieve the design's performance metrics. The goal of this chapter is to evaluate the Range Trie design's scalability in terms of the incoming address width and the number of ranges in the lookup table. Along with the evaluation based on the performance metrics, other interesting issues regarding the Range Trie design will also be investigated.

After presenting the experimental setup that was used for the synthesis purposes in Section 4.1, a variety of iteration stages is generated, synthesized and evaluated in Section 4.2. Afterwards, in Section 4.3, a number of complete Range Trie designs is generated, synthesized and evaluated in terms of operating frequency, area, power consumption and memory requirements. Alongside, the Range Trie method is compared with existing solutions to show its improvements.

4.1 Experimental setup

In this section the experimental setup will be presented in order to define the details surrounding the used synthesis process. That way it will be possible to synthesize and evaluate the designed hardware in the following sections.

It was decided to evaluate the performance of the Range Trie on an ASIC implementation. In particular, we used the 90nm and 130nm ASIC processes. The synthesis was performed using the Synopsys Design Compiler tool. The synthesis process [2] that was followed was a basic one, meaning that we performed the necessary steps to acquire timing, power and area estimates of the designs under consideration. We did not followed the rest steps in the ASIC design flow that lead to a complete implementation (i.e. for tape-out purposes).

To be able to perform the synthesizes we obtained the respective foundry libraries for 90nm and 130nm technologies. We used the standard performance libraries (90nm_SP and 130nm_SP) of UMC-Faraday [1].

Alongside, we used technology libraries of the same type that define memory units in order to implement the memory structure of the Range Trie. Since the available memory units were not tailored to match exactly our needs, we opted to generate the required memory units out of the available ones. This meant that the generated memories were not the optimal ones because logic was replicated (such as the address selection logic) and because memory elements were unused (since the generated memory units could not match exactly the size of the required units).

During the synthesis process, the designs were constrained in terms of clock time in order to minimize the synthesized critical path and increase the operating frequency, as high speed was the target during the whole design process. The constraints were chosen in a way that did not make the tool to produce unnecessary large power consumption and area. A number of synthesizes were performed per design in order to retrieve the best results. To ensure the correctness of the synthesis results, each synthesized design was validated beforehand.

The synthesis results that we were interested in were: the clock cycle (operating frequency), the estimated power consumption[1] and the area[2] of the designs. In the rest of this chapter the synthesis results for a variety of hardware designs will be presented in order to evaluate the Range Trie design.

4.2 Evaluation of the iteration stage

The Range Trie design (as presented in Chapter 3) is a pipeline design consisting of interleaving iteration and memory access stages. In this section the iteration stage hardware will be evaluated in terms of various metrics (operating frequency, area, power consumption). The hardware under consideration was synthesized for all possible values of incoming address width (W=32, 64, 128 bits) and available memory bandwidth (BW=256, 512, 1024 bits) to investigate on the design's scalability. Furthermore, a set of variations of the iteration stage hardware was also synthesized to research on the effect of various design choices.

The iteration stage was chosen to be evaluated independently of the complete Range Trie pipeline since it is the core of the Range Trie method and it is where the novelties of the Range Trie method are evident. By introducing and evaluating some variations of the iteration stage, that are actually variations of the Range Trie method, the effect of certain design choices is examined. In particular the following iteration stage variations were generated for evaluation purposes:

1. *Full iteration stage*: This is the complete iteration stage that was chosen for the final complete design.

2. *Full iteration stage with a variable-width subtractor* and *without a subtractor*: To investigate on the effect of the bound alignment subtractor, two variations of the iteration stage were generated: one allowing for a wider variable-width subtraction and one where no bound alignment is performed.

3. *Iteration stage without memory addressing hardware*: To investigate on the cost of computing the next memory entry to retrieve, a variation of the iteration stage was generated consisting of just the iteration hardware.

4. *Basic iteration stage without a subtractor and memory addressing hardware*: This is a basic variation of the iteration stage where the memory addressing units and

[1]The power consumption results reported by the Synopsys Design Compiler are calculated as an estimation and not based on the actual usage of the underlying logic.

[2]The reported area corresponds to the area occupied by the logic cells.

the internal subtractor (for bound alignment) were eliminated. Its purpose is to measure the efficiency of a stripped down version of the iteration stage.

These variations are depicted abstractly on Figure 4.1 and where chosen that way since they consist of modifications that have an effect on the critical path of the iteration stage and therefore the speed of the resulting design.

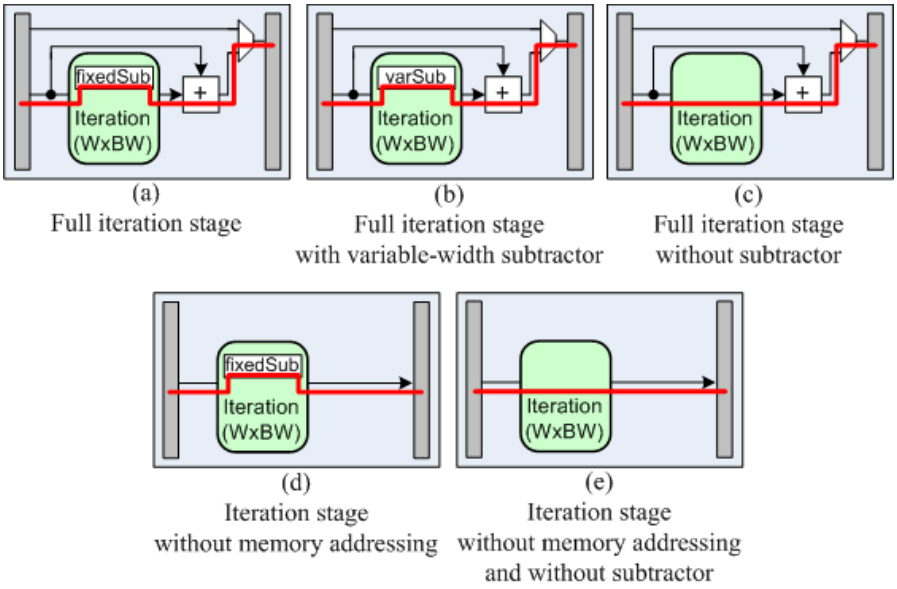


Figure 4.1: The five generated variations of the iteration stage for evaluation purposes. In (a), the full iteration stage is depicted as it was used in the complete pipeline design (see Figure 3.37). In (b)-(e), the iteration stage has been modified by varying the subtractor used for bound alignment or by eliminating the hardware required for the chosen addressing scheme. The red line on the diagrams depicts abstractly the critical path of each variation.

Full iteration stage: The full iteration stage, consisting of the Range Trie iteration hardware (for a given W and BW) followed by the offset adder and a multiplexor (as depicted in Figure 4.1(a)) was synthesized for all possible values of W and BW in both 90nm and 130nm ASIC technologies. The obtained results (operating frequency, area and power consumption) may be seen in Tables 4.1 and 4.2 for the 90nm and 130nm technologies respectively. These results are depicted graphically in Figures 4.2 and 4.3 respectively.

Out of these results there are some observations to be made. Moving to larger BW decreases the operating frequency, while the power consumption and area increase. This is expected because the increase of BW for a given W increases the number of used comparators in an iteration. Using more comparators means that the logic to interpret the comparison results gets more complicated with an increased critical path. At the same time, more hardware units are replicated resulting in the increase of area and power consumption.

Another thing to be noticed is that moving to wider incoming address widths (W) for a given BW decreases the operating frequency, while the power consumption and area

Table 4.1: The operating frequency, area and power consumption results obtained after synthesizing the full iteration stage for all possible values of W and BW using 90nm ASIC technology.

Full iteration stage (90nm)				
W	**BW**	**Frequency** (MHz)	**Area** (mm^2)	**Power** (mW)
32	256	729.93	0.046	11.49
	512	645.16	0.077	17.30
	1024	591.72	0.142	30.56
64	256	714.29	0.095	19.85
	512	595.24	0.149	26.45
	1024	534.76	0.242	40.74
128	256	621.12	0.208	36.59
	512	555.56	0.281	45.06
	1024	478.47	0.432	62.54

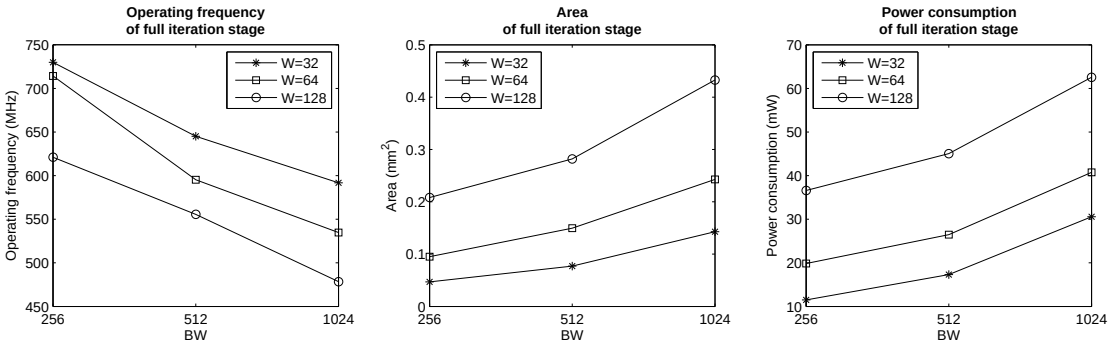


Figure 4.2: Graphic depiction of the operating frequency, area and power consumption results obtained after synthesizing the full iteration stage for all possible values of W and BW using 90nm ASIC technology. The exact result values may be seen in Table 4.1.

consumption increases. This was also expected because a wider W means more complex selection logic, more complex comparators and slower interpretation hardware.

The above observations are evident for both ASIC technologies (90nm and 130nm). One last thing to notice is that moving from 90nm to 130nm leads to lower operating frequencies (almost half), roughly the same area and reduced power consumptions. These changes were expected because smaller technology widths result into higher operating speed at the cost of higher power consumption. It is now a decision of the system designer to choose the ASIC technology that fits its needs, where more factors are taken into consideration, such as the manufacturing cost. In the rest of this section, the rest iteration variations are evaluated only for 90nm ASIC technologies.

To conclude, the most efficient design out of the 9 synthesized ones is the one with the narrower W and BW; the 32x256 iteration stage design. This doesn't necessarily rule out the rest design points. As it will be explained in Section 4.3, choosing the parameters of the complete Range Trie design is a process coupled with a given Range

Table 4.2: The operating frequency, area and power consumption results obtained after synthesizing the full iteration stage for all possible values of W and BW using 130nm ASIC technology.

Full iteration stage (130nm)				
W	**BW**	**Frequency** (MHz)	**Area** (mm^2)	**Power** (mW)
32	256	321.54	0.043	7.26
	512	288.18	0.076	11.39
	1024	250.00	0.139	18.96
64	256	315.46	0.083	12.31
	512	263.85	0.138	16.77
	1024	235.29	0.238	26.24
128	256	284.09	0.189	22.90
	512	245.70	0.276	30.48
	1024	207.47	0.421	38.00

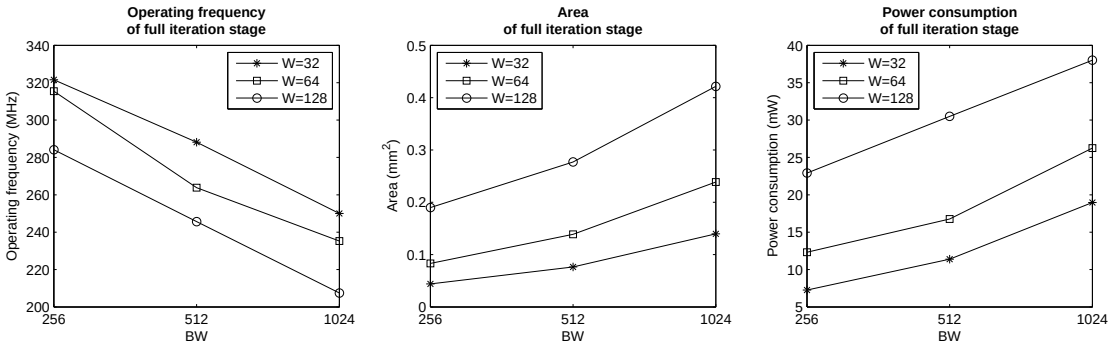


Figure 4.3: Graphic depiction of the operating frequency, area and power consumption results obtained after synthesizing the full iteration stage for all possible values of W and BW using 130nm ASIC technology. The exact result values may be seen in Table 4.2.

Trie structure, where the target is a fast design with as few pipeline stages as possible and minimized memory requirements.

Effect of the bound alignment subtractor: As described in Section 3.1.1.2, the Range Trie iteration uses a subtractor to align the address bounds of a Range Trie node according to Rule 5 of the Range Trie method (see Section 2.2.2). The purpose of Rule 5 is to further reduce the depth of a Range Trie structure, while utilizing optimally the available memory bandwidth.

Performing a subtraction is a slow operation whose speed depends on the width of the operands. For that reason, it was limited to $W/4$ bits in the full iteration stage. According to Rule 5, there may be benefits into using a wider subtractor. In this paragraph, the cost of using the subtractor is evaluated. In particular, the following three variations of the iteration stage are compared:

- The *full iteration stage*, that performs fixed $W/4$ wide subtractions. This is the

variation that was evaluated previously and the one used in the complete Range Trie design.

- The *full iteration stage with a variable-width subtractor* (varSub) that allows $W/4$ or $W/2$ wide subtractions[3].

- The *full iteration stage without a subtractor* (noSub), which means that Rule 5 is never used while constructing a Range Trie structure.

The *full iteration stage with a variable-width subtractor* and the *full iteration stage without a subtractor* (as depicted in Figure 4.1(b-c)) were synthesized for all possible values of W and BW in 90nm ASIC technologies. The obtained results (operating frequency, area and power consumption) may be seen in Tables 4.3 and 4.4 for each of the two variations respectively. These results are compared against the *full iteration stage* results in Figure 4.4.

Table 4.3: The operating frequency, area and power consumption results obtained after synthesizing the full iteration stage with a variable-width subtractor for all possible values of W and BW using 90nm ASIC technology.

Full iteration stage with a variable-width subtractor (90nm)				
W	**BW**	**Frequency** (MHz)	**Area** (mm^2)	**Power** (mW)
32	256	641.03	0.046	10.47
	512	595.24	0.079	16.54
	1024	531.91	0.147	28.51
64	256	588.24	0.083	14.88
	512	523.56	0.140	22.75
	1024	476.19	0.255	38.36
128	256	512.82	0.162	23.31
	512	476.19	0.258	36.22
	1024	416.67	0.421	53.25

For each of the synthesized variations it still holds that moving to wider incoming address widths (W) and wider given memory bandwidths (BW) results in a decrease in the operating frequency and an increase in power consumption and area.

Out of Figure 4.4 some more expected observations may be made regarding the resulting performance metrics. It may be seen that not using a subtractor results into significantly higher operating frequencies (up to 840 MHz for 32x256). Using a variable-width subtractor has a big impact on operating frequency due to the need for a two times wider subtraction than using a fixed-width subtractor. As expected, the full iteration stage with the fixed-width subtractor lies between the other two in terms of operating frequency. Regarding the power consumption and the area, not using a subtractor is

[3]The variable-width subtractor design was presented in Section 3.1.1.2

Table 4.4: The operating frequency, area and power consumption results obtained after synthesizing the full iteration stage without a subtractor for all possible values of W and BW using 90nm ASIC technology.

Full iteration stage without a subtractor (90nm)				
W	**BW**	**Frequency** (MHz)	**Area** (mm^2)	**Power** (mW)
32	256	840.34	0.044	11.55
	512	751.88	0.084	19.76
	1024	657.89	0.151	31.74
64	256	833.33	0.062	15.82
	512	684.93	0.110	23.39
	1024	625.00	0.206	38.28
128	256	729.93	0.071	15.17
	512	625.00	0.146	26.40
	1024	540.54	0.299	46.45

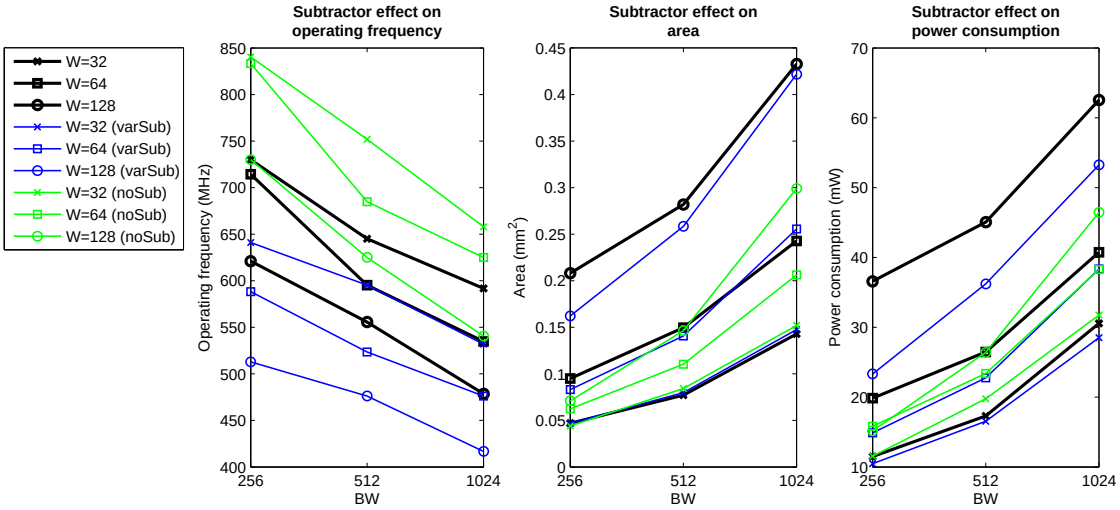


Figure 4.4: Evaluating the effect of bound alignment on the full iteration stage. The operating frequency, area and power consumption results are depicted. These were obtained after synthesizing three variations of full iteration stage for all possible values of W and BW using 90nm ASIC technology. The variations are: full iteration stage with a fixed-width subtractor (black lines), full iteration stage with variable-width subtractor (blue lines) and full iteration stage without subtractor (green lines). The exact result values may be seen in Tables 4.1, 4.3 and 4.4 respectively.

the only approach resulting into lower area and power consumption, especially for larger values of W and BW.

Evaluating which approach is the best depends on the Range Trie structure under consideration. Not using a subtractor is definitely an attractive solution but it means

eliminating bound alignment during the construction of a Range Trie. This effectively results into a deeper Range Trie (as proven in [6]) that requires more pipeline stages and underutilizes the memory. On the contrary, using a variable-width subtractor allows for wider subtraction and, thus, more possibilities of applying the bound alignment rule. That way the number of Range Trie levels may be reduced at the cost of a slower iteration stage. For the complete Range Trie design we opted for the middle solution, the fixed-width subtractor that offers the advantages of using the bound alignment without a high cost on the performance metrics. However, the other two approaches may be used if considered beneficial of a given Range Trie structure.

Effect of the memory organization scheme: The full iteration stage that was evaluated so far also includes the required memory addressing hardware, according to the chosen memory organization scheme (Section 3.2). In particular, the memory addressing hardware consists of the offset adder and a multiplexor that is used for dealing with leaf nodes. In this section, the memory addressing hardware is eliminated from the iteration stage to evaluate the cost of the chosen memory organization scheme.

The *iteration stage without the memory addressing hardware* (as depicted in Figure 4.1(d)) was synthesized for all possible values of W and BW in 90nm ASIC technologies. The obtained results (operating frequency, area and power consumption) may be seen in Table 4.5. These results are compared against the *full iteration stage* results in Figure 4.5.

Table 4.5: The operating frequency, area and power consumption results obtained after synthesizing the iteration stage without the memory addressing hardware for all possible values of W and BW using 90nm ASIC technology.

Iteration stage without memory addressing hardware (90nm)				
W	**BW**	**Frequency** (MHz)	**Area** (mm^2)	**Power** (mW)
32	256	862.07	0.044	12.73
	512	751.88	0.078	21.31
	1024	680.27	0.144	34.28
64	256	826.45	0.096	23.28
	512	689.66	0.142	29.22
	1024	617.28	0.252	49.64
128	256	729.93	0.216	43.66
	512	636.94	0.292	52.91
	1024	540.54	0.417	66.10

From Figure 4.5 the benefits of eliminating the memory addressing hardware may be seen. The operating frequency is higher, the power consumption is less and the area remains almost similar. These observations indicate the benefits of using a simple memory addressing scheme where no offset addition needs to be performed. However choosing such a scheme would require a fixed location of Range Trie nodes in the memory struc-

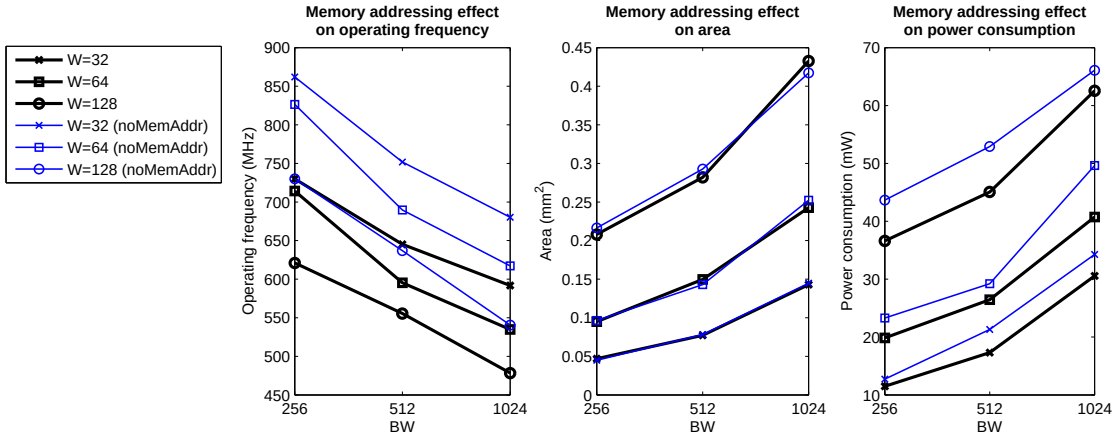


Figure 4.5: Evaluating the effect of the chosen memory addressing hardware on the iteration stage. The operating frequency, area and power consumption results are depicted. These were obtained after synthesizing (a) the full iteration stage (black lines) and (b) the iteration stage without memory addressing hardware (blue lines) for all possible values of W and BW using 90nm ASIC technology. The exact result values may be seen in Tables 4.1 and 4.5 respectively.

ture. That would require significantly larger memory units where most of the memory entries would not be used. Furthermore, eliminating the memory addressing hardware removes the support of leaf nodes in intermediate Range Trie levels. This issue may be dealt by allowing Range Trie structures that contain leaf nodes only on the bottom level.

Basic iteration stage: The last iteration stage variation that was generated is a basic one where both the alignment subtractor and the memory addressing hardware are eliminated. From the previous discussion we concluded that it is better not to remove these elements, since that would lead in a deeper pipeline with larger memory requirements. However, such a basic iteration stage is still viable and it would be interesting to see how efficiently an iteration is performed in that case.

The *basic iteration stage* (as depicted in Figure 4.1(e)) was synthesized for all possible values of W and BW in 90nm ASIC technologies. The obtained results (operating frequency, area and power consumption) may be seen in Table 4.6. These results are compared against the *full iteration stage* results in Figure 4.6.

As expected, the synthesize results of the basic iteration stage are significantly better against the ones of the full iteration stage. It is worth noting the best possible operating frequency was encountered in this variation (1.01 GHz for 32x256). However, as mentioned, using this iteration hardware variation has negative effects on the complete Range Trie design.

To conclude this section, five different variations of the iteration stage were generated and evaluated in terms of operating frequency, area and power consumption. The generation of these variations was targeting the modification of the critical path in order to evaluate mainly the effect on the operating frequency under various design scenarios. The resulting operating frequencies varied from 416.67 MHz up to 1010.10 MHz. We

Table 4.6: The operating frequency, area and power consumption results obtained after synthesizing the iteration stage without the memory addressing hardware and without the subtractor for all possible values of W and BW using 90nm ASIC technology.

Basic iteration stage without memory addressing hardware and without a subtractor (90nm)				
W	**BW**	**Frequency** (MHz)	**Area** (mm^2)	**Power** (mW)
32	256	1010.10	0.041	13.12
	512	877.19	0.081	22.33
	1024	746.27	0.150	36.03
64	256	952.38	0.053	14.73
	512	800.00	0.106	24.93
	1024	709.22	0.214	44.87
128	256	819.67	0.067	15.94
	512	714.29	0.150	31.87
	1024	606.06	0.301	51.73

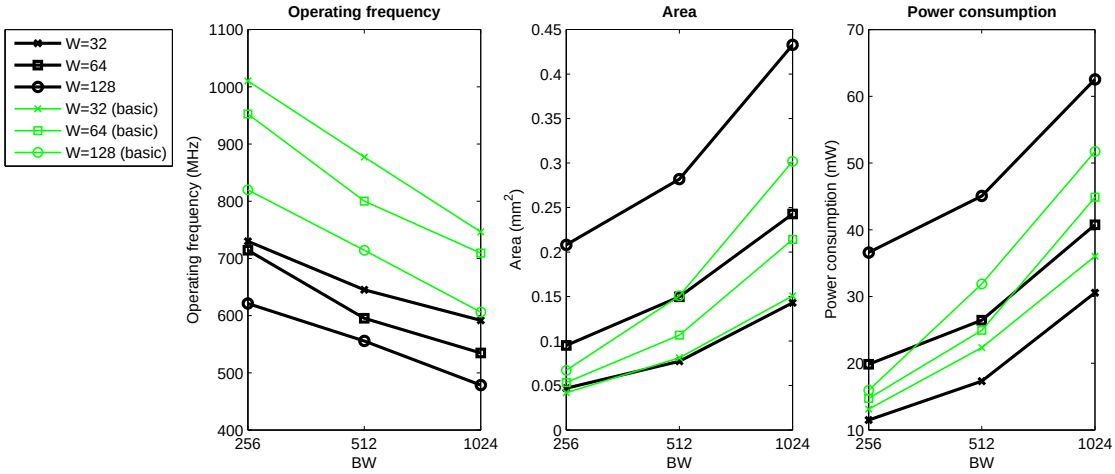


Figure 4.6: Comparing the full iteration stage (that was used in the complete pipeline) against the basic iteration stage (where the subtractor and memory addressing schemes were removed). The operating frequency, area and power consumption results of both variations are depicted after synthesizing (a) the full iteration stage (black lines) and (b) the basic iteration stage (green lines) for all possible values of W and BW using 90nm ASIC technology. The exact result values may be seen in Tables 4.1 and 4.6 respectively.

opted to use the first evaluated variation (the full iteration stage) that offers operating frequencies that range between 478.47 MHz and 729.93 MHz, along with a nice tradeoff on the resulting memory requirements and number of pipeline stages. In the next section the complete Range Trie pipeline will be evaluated, while using as an iteration stage the

full iteration stage.

4.3 Evaluation of the complete design

In this section the complete Range Trie hardware design (as presented in Chapter 3) is evaluated in terms of operating frequency, area, power consumption and memory requirements. Along with the evaluation based on the performance metrics, other interesting issues regarding the Range Trie design will also be discussed. Since the Range Trie design is parameterizable in many of its aspects (incoming address width, available memory bandwidth, number of iterations, memory sizes per level and output width), a set of representative Range Trie instances was generated and synthesized for ASIC to investigate on the scalability of the Range Trie design.

Choosing and generating the Range Trie instances for evaluation: Before presenting the synthesis results for the generated Range Trie instances, the chosen Range Trie instances will be defined, along with the used process to generate them.

In general, a Range Trie hardware instance is tailored to a specific Range Trie structure that performs lookup on a predefined lookup table consisting of a number of ranges. To be more specific, in order to obtain a compatible Range Trie instance, this process must be followed:

1. Given a lookup table consisting of N ranges, the heuristic construction methods of [6], that were described briefly in Section 2.2.4, must be employed to generate a Range Trie structure. The heuristic construction methods must be tailored to the specific hardware implementation, meaning that the same incoming address width (W) and available memory bandwidth (BW) are given as a parameter to the construction method. As a result a Range Trie structure is obtained that is L levels deep and contains in every node the required information in order to perform the lookup.

2. Given the obtained Range Trie structure, the parameters of a specific Range Trie hardware instance may be decided: the incoming address width (W), the available memory bandwidth (BW), the number of iteration stages in the pipeline ($L-1$), the required memory entries per memory level (m_1, ..., m_{L-1}) and the output width ($\lceil o = log_2 N \rceil$).

3. The chosen parameters are then used to *automatically* generate a Range Trie instance that may perform the address lookup based on the initial lookup table[4].

For the evaluation purposes, we decided to generate a set of representative Range Trie instances in order to investigate on the scalability of the design. The chosen instances vary in terms of incoming address width (W=32, 64, 128 bits), available memory bandwidth (BW=256, 512, 1024 bits) and number of ranges in the lookup table (N=256, 512, ..., 256K, 512K ranges). To obtain the respective Range Trie instances the previous

[4]Before starting the operation of the hardware, the memory units must be configured according to the Range Trie node information and the chosen memory organization.

Table 4.7: The 39 generated Range Trie instances that vary in terms of incoming address width (W), available memory bandwidth (BW) and number of ranges in the lookup table (N). The rest of the design parameters, such as the number of iteration stages (L-1) and the sizes of the memory units (m_i), were decided after constructing the respective Range Trie structures. These design instances were synthesized for ASIC and the synthesis results will be presented afterwards.

W	**BW**	**N**	**L-1**	m_1	m_2	m_3	m_4	m_5	m_6	m_7	**o**
32	256	256-1K	3	1	32	512					20
		2K-32K	4	1	32	512	8K				20
		64K-512K	5	1	32	512	8K	64K			20
32	512	256	2	1	64						20
		512-8K	3	1	64	1K					20
		16K-256K	4	1	64	1K	32K				20
		512K	5	1	64	1K	32K	64K			20
32	1024	256-1K	2	1	128						20
		2K-64K	3	1	128	4K					20
		128K-256K	4	1	128	4K	16K				20
		512K	4	1	128	4K	32K				20
64	256	256	3	1	32	256					20
		512-4K	4	1	32	256	2K				20
		8K-64K	5	1	32	256	2K	32K			20
		128K-256K	6	1	32	256	2K	32K	128K		20
		512K	6	1	32	256	2K	32K	256K		20
64	512	256-4K	3	1	64	1K					20
		8K-128K	4	1	64	1K	32K				20
		256K	5	1	64	1K	32K	64K			20
		512K	5	1	64	1K	32K	128K			20
64	1024	256-512	2	1	128						20
		1K-32K	3	1	128	4K					20
		64K-128K	4	1	128	4K	32K				20
		256K-512K	4	1	128	4K	64K				20
128	256	256-512	4	1	32	128	512				20
		1K-8K	5	1	32	128	512	8K			20
		16K-64K	6	1	32	128	512	8K	64K		20
		128K	7	1	32	128	512	8K	64K	128K	20
		256K	7	1	32	128	512	8K	64K	256K	20
		512K	7	1	32	128	512	8K	64K	512K	20
128	512	256-1K	3	1	64	512					20
		2K-32K	4	1	64	512	16K				20
		64K-128K	5	1	64	512	16K	64K			20
		256K	5	1	64	512	16K	128K			20
		512K	5	1	64	512	16K	256K			20
128	1024	256-16K	3	1	128	4K					20
		32K-128K	4	1	128	4K	32K				20
		256K	4	1	128	4K	64K				20
		512K	4	1	128	4K	128K				20

process was followed[5]. Table 4.7 shows the 39 resulting designs for every possible value of W, BW and N. Note that a common output width ($o = 20$ bits) was assumed that suffices to represent the number of the matched range. Also, many instances with a different N were grouped into one, as the number of iterations was the same for each N.

Synthesis results: Each of the 39 Range Trie instances of Table 4.7 was synthesized for 90nm and 130nm ASIC technologies. The resulting operating frequency, area and power consumption for every synthesized Range Trie are shown in Table 4.8. Figures 4.7, 4.8 and 4.9 depict graphically the operating frequency, area and power consumption results respectively of all the synthesized Range Trie instances at 90nm ASIC technology. Figures 4.10, 4.11 and 4.12 depict graphically the operating frequency, area and power consumption results respectively of all the synthesized Range Trie instances at 130nm ASIC technology.

By inspecting the synthesis results various observations may be made. First, regarding the operating frequency of the Range Trie, it may be seen that it mirrors the findings during the evaluation of the iteration stage (Section 4.2). Designs with a wider incoming address (W) and wider available memory bandwidths (BW) have a smaller operating frequency (Figures 4.7 and 4.10), as it was the case with the iteration stage itself. It may be seen that the operating frequency scales nicely with the increase of W. Moving from designs with W=32 (IPv4) towards W=128 (IPv6) decreases the operating frequency with 100MHz on average, although the W has been quadrupled. In general, the operating frequency ranges from 442MHz up to 694MHz for 90nm ASIC technology and from 191MHz up to 296MHz for 130nm.

It was found that the longest critical path of the complete design sometimes lies within the iteration stage and sometimes within the memory access stage (especially when large memory units are used). This indicated that the pipeline was balanced. To further improve on the operating frequency of the complete design another variation of the iteration stage may be used, as the ones presented in Section 4.2, at the cost of different memory requirements and number of iterations.

Note that in the complete design the operating frequency is smaller than the ones from the respective iteration stage (Table 4.1) because other issues come into play, such as the wiring fanout. This is also evident by the decrease of the operating frequency when moving to larger lookup tables that require larger memories and, thus, more wiring. In this case the operating frequency also scales well. As an example for the Range Trie design with W=32 and BW=256, moving from 32K ranges to 512K ranges results in just 30MHz operating frequency reduction.

In general, it may be claimed that the operating frequency scales efficiently when moving to wider incoming addresses and larger lookup tables, as depicted also in Figures 4.7 and 4.10, where the operating frequencies of all synthesized instances are depicted for 90nm and 130nm technologies respectively. One last thing to mention is that changing the ASIC technology has the expected effect on operating frequencies; moving to the slower 130nm process results in less than half operating frequency.

[5]The bottom-up heuristic construction method with variable length comparisons was used for obtaining the Range Trie structures. The original lookup tables were assumed to consist of N ranges distributed uniformly in the address space. Also, the address bound alignment was activated.

Table 4.8: The operating frequency, area and power consumption results obtained after synthesizing the 39 Range Trie instance of Table 4.7 using 90nm and 130nm ASIC technologies.

W	BW	N	90nm			130nm		
			Freq. (MHz)	**Area** (cm^2)	**Power** (W)	**Freq.** (MHz)	**Area** (cm^2)	**Power** (W)
32	256	256-1K	694.44	0.018	0.461	296.74	0.062	0.821
		2K-32K	666.67	0.082	1.026	280.11	0.139	1.399
		64K-512K	632.91	0.590	5.076	275.48	0.658	3.370
32	512	256	625.00	0.011	0.233	271.00	0.060	0.733
		512-8K	628.93	0.031	0.515	249.38	0.117	1.344
		16K-256K	609.76	0.479	3.957	247.52	0.601	3.039
		512K	574.71	1.363	10.542	245.70	1.557	6.603
32	1024	256-1K	558.66	0.021	0.403	219.78	0.113	1.142
		2K-64K	546.45	0.124	1.088	220.26	0.294	2.246
		128K-256K	529.10	0.529	3.811	219.30	0.751	3.571
		512K	540.54	0.934	6.664	217.39	1.210	5.099
64	256	256	671.14	0.014	0.295	271.74	0.066	0.794
		512-4K	645.16	0.032	0.462	268.10	0.105	1.183
		8K-64K	621.12	0.285	2.489	265.25	0.377	2.207
		128K-256K	628.93	1.296	10.620	262.47	1.469	6.658
		512K	588.24	2.306	17.636	262.47	2.560	11.101
64	512	256-4K	561.80	0.033	0.476	233.64	0.119	1.218
		8K-128K	546.45	0.481	3.649	230.41	0.602	2.785
		256K	543.48	1.366	9.989	229.89	1.560	6.052
		512K	549.45	2.269	16.452	232.02	2.539	9.504
64	1024	256-512	526.32	0.023	0.378	213.22	0.115	1.162
		1K-32K	507.61	0.126	0.956	207.04	0.296	2.151
		64K-128K	487.80	0.937	6.200	207.04	1.213	4.761
		256K-512K	502.51	1.747	12.130	202.43	2.131	7.644
128	256	256-512	571.43	0.030	0.573	246.31	0.102	1.168
		1K-8K	543.48	0.094	1.008	238.66	0.184	1.545
		16K-64K	555.56	0.599	4.618	239.23	0.727	3.544
		128K	543.48	1.611	11.862	237.53	1.820	7.698
		256K	540.54	2.621	18.069	237.53	2.911	11.843
		512K	500.00	4.640	31.727	236.41	5.093	20.108
128	512	256-1K	537.63	0.039	0.706	215.05	0.125	1.212
		2K-32K	500.00	0.269	2.130	213.68	0.374	1.919
		64K-128K	495.05	1.179	8.012	220.75	1.362	5.159
		256K	505.05	2.096	13.940	216.45	2.360	8.523
		512K	497.51	3.921	25.717	212.77	4.344	15.245
128	1024	256-16K	450.45	0.135	0.904	190.48	0.305	1.986
		32K-128K	446.43	0.968	5.805	191.94	1.237	4.573
		256K	442.48	1.799	10.751	191.57	2.167	7.340
		512K	442.48	3.461	20.650	191.20	4.030	12.949

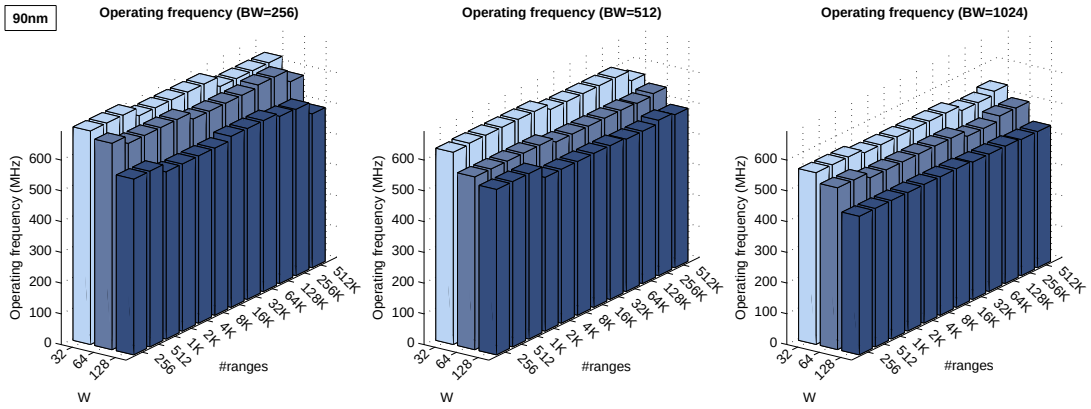


Figure 4.7: Graphic depiction of the operating frequency results obtained after synthesizing all Range Trie instances using 90nm ASIC technology. The exact result values may be seen in Table 4.8.

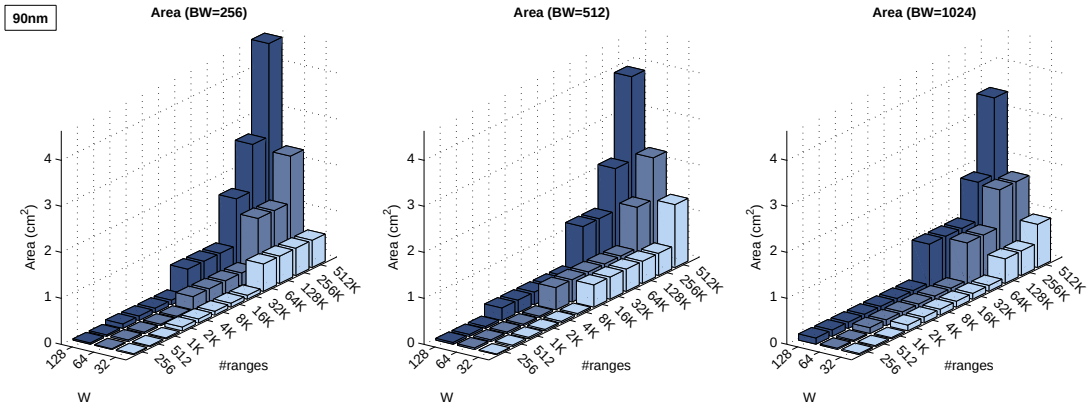


Figure 4.8: Graphic depiction of the area results obtained after synthesizing all Range Trie instances using 90nm ASIC technology. The exact result values may be seen in Table 4.8.

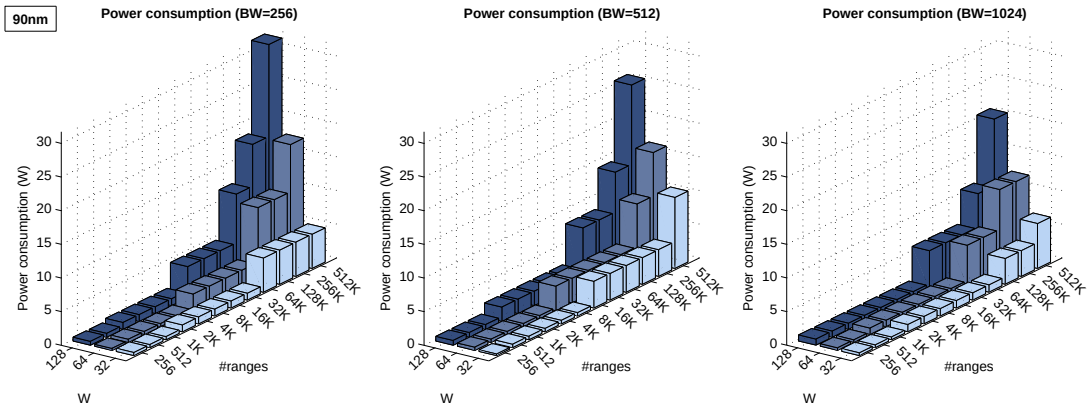


Figure 4.9: Graphic depiction of the power consumption results obtained after synthesizing all Range Trie instances using 90nm ASIC technology. The exact result values may be seen in Table 4.8.

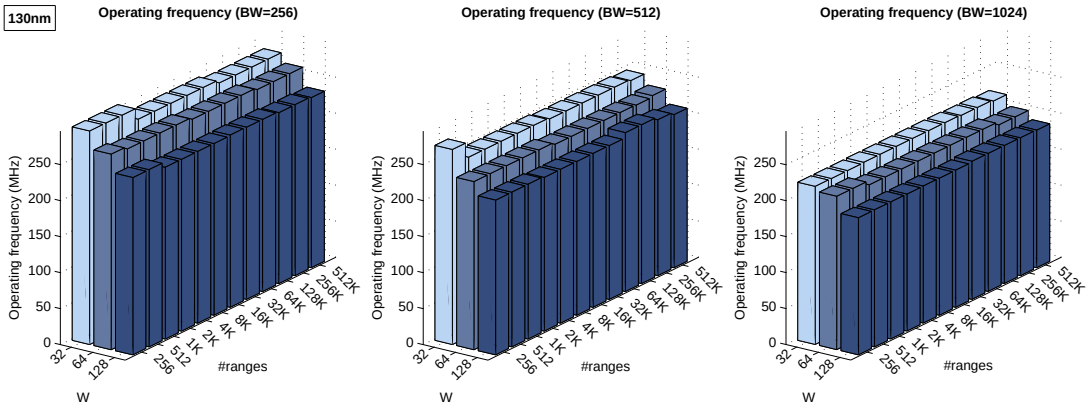


Figure 4.10: Graphic depiction of the operating frequency results obtained after synthesizing all Range Trie instances using 130nm ASIC technology. The exact result values may be seen in Table 4.8.

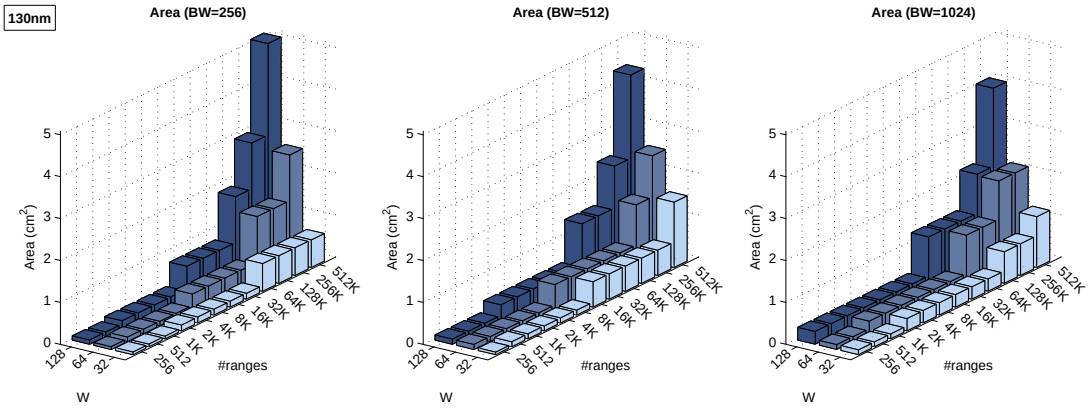


Figure 4.11: Graphic depiction of the area results obtained after synthesizing all Range Trie instances using 130nm ASIC technology. The exact result values may be seen in Table 4.8.

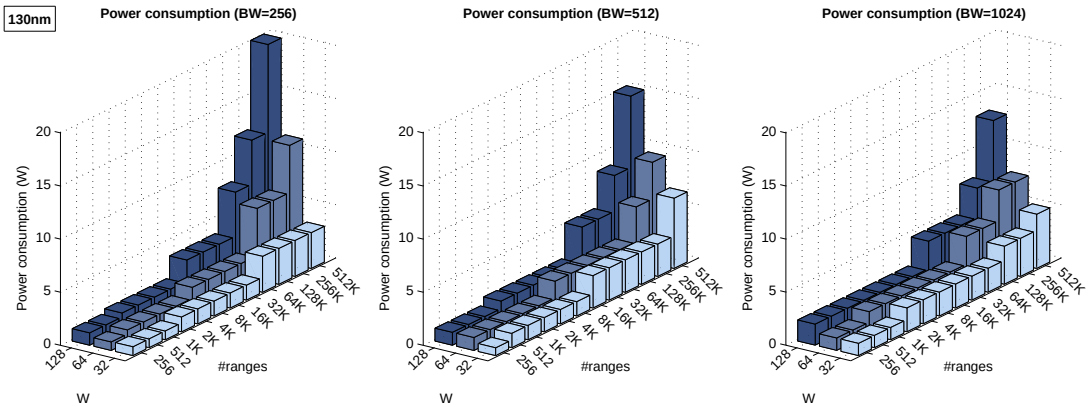


Figure 4.12: Graphic depiction of the power consumption results obtained after synthesizing all Range Trie instances using 130nm ASIC technology. The exact result values may be seen in Table 4.8.

While the operating frequency results were following the results of the iteration stage evaluation and depend mainly on the incoming address width and less on the lookup table size, the case is different for the area and power consumption results. The area and power consumption depend highly on the lookup table size and the incoming address width (Figures 4.8-4.9 and 4.11-4.12). Increasing the number of ranges results in the need for deeper Range Trie structures that require more pipeline stages and, thus, larger memory units per stage. As the vast amount of the occupied area consists of non-combinational (memory) elements, moving to larger lookup tables increases the power consumption and the area because the memory requirements increase. This increase is more evident when moving to the larger sizes of the lookup tables (64K - 512K) and at the same time moving to wider incoming addresses. As an example, for W=32, the power consumption and area remain low for most of the lookup table sizes, but given a large lookup table size (512K), moving from W=32 to W=64 increases the area and power consumption by 3.5 times and moving from W=64 to W=128 increases them by 2 times.

Not definite conclusions based on the synthesis results can be made for the scaling of area and power consumption because the obtained area and power consumption results were also affected by the used memory units. As mentioned in Section 4.1, the used memory units were designed out of a set of available memory units for the specific ASIC technologies. This meant the replication of unnecessary logic and the employment of unnecessary memory elements, resulting into larger than needed area and power consumption. The use of inappropriate memory units (that differed for 90nm and 130nm technologies) was also the reason that moving from 90nm to 130nm technology provided incoherent area and power consumption results; for some designs the power consumption was larger in 130nm and for others it was smaller (as it should be). Especially for the power consumption the results are prohibitively large for some designs. This situation was furthermore intensified by the fact that the Synopsys Design Compiler measures the power consumption as an estimation, instead of calculating based on the actual usage of the hardware, and that rendered out the design optimizations for saving power (such as disabling the memory units that don't need to be accessed). However, the area and power consumption results still give an indication regarding the real scaling when moving to wider incoming addresses and larger lookup tables.

Before concluding the evaluation of the synthesis results, it is useful to present another set of obtained results (Figure 4.13). This time the synthesized designs were under-constrained to retrieve the area and power consumption results as a function of throughput (measured in Gbps). The target was to see how the designs performed in the following actual wire speeds: in OC-48 (2.5Gbps), OC-192 (10Gbps), OC-768 (40Gbps) and OC-3072 (160Gbps). Since this set of designs targets real conditions, the IPv4 and IPv6 variants (with an available memory bandwidth of 256 bits) were synthesized that support 256K and 512K number of ranges in the lookup table. A minimum sized packet of 40 bytes was assumed.

Figure 4.13 offers the chance to see how the power consumption and area results are affected depending on the throughput. It may be seen that area remains unaffected, while the power consumption decreases in orders of magnitude when a low throughput is required. This also points that a designer may generate a design that is more fitting with

his needs, if for example he is interested in a lookup mechanism with low throughput.

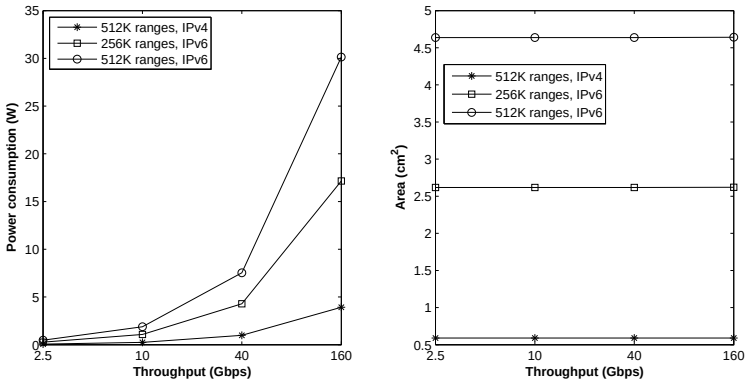


Figure 4.13: The power consumption and area results as a function of throughput. The results correspond to IPv4 and IPv6 address lookup using lookup tables ranging from 256 to 512K entries. The used ASIC technology was the 90nm one.

To conclude the evaluation of the synthesis results, the presented results showed that the Range Trie's operating frequency scales well in terms of incoming address width and lookup table size. A maximum operating frequency of 694MHz was achieved. Moving from IPv4 addresses (32-bits) to the wider IPv6 ones (128-bits) results on average in a 100MHz moderate decrease of the operating frequency. Using larger lookup tables results on very small decreases of operating frequency. As an extreme case, moving from 256 to 512K lookup tables decreases the operating frequency by a value ranging between 8MHz to 90MHz, depending the specifics of the design. The worst achieved operating frequency was 442MHz, when the lookup table size and incoming address width were maximized. All of these meant that, assuming a 40 bytes sized packet, the Range Trie design may support OC-3072 (160Gbps) wire speeds even for 512K ranges.

Regarding the area and power consumption, moving to significantly larger lookup tables (64K-512K ranges) and to wider incoming addresses (when already using a large lookup table) results in a big increase of these two performance metrics. In the rest cases, area and power consumption scaled efficiently in terms of the address width and the lookup table size. In any case, the area and power consumption depend mainly on the memory requirements and scale according to the memory requirements scaling. So, as the operating frequency was shown to scale efficiently enough, it remains to show the nice scaling of the memory requirements (and number of iterations).

Evaluation of memory requirements and number of iterations: Another important aspect of the Range Trie design is the total required memory to store a Range Trie structure and the number of required iteration stages. As mentioned before, these two parameters affect greatly the occupied area and the power consumption. Furthermore, the number of required iteration stages indicates the total number of pipeline stages and therefore the latency to match an incoming address to a range. In the following text

some results are going to be presented regarding the scaling of the memory requirements and the number of levels in terms of the incoming address width and the lookup table size.

The main idea behind the Range Trie method was to design a lookup mechanism with low latency, high troughput and low memory requirements. At the same time it is essential that these properties scale efficiently for an increasing incoming address width and lookup table size.

Figure 4.14 depicts the number of required iterations for the synthesized Range Trie instances. These values were decided based on the number of levels of the respective Range Trie structure (Table 4.7). It may be seen that the number of iterations ranged from 2 up to 7. In general, the number of iterations scaled well for the synthesized designs. Moving to a wider incoming address (W) might inflict at-most one extra iteration, while moving from the smallest (256) to the largest (512K) lookup tables inflicts at-most three extra iterations. At the same time, using a wider memory bandwidth (BW) resulted on average to one less iteration.

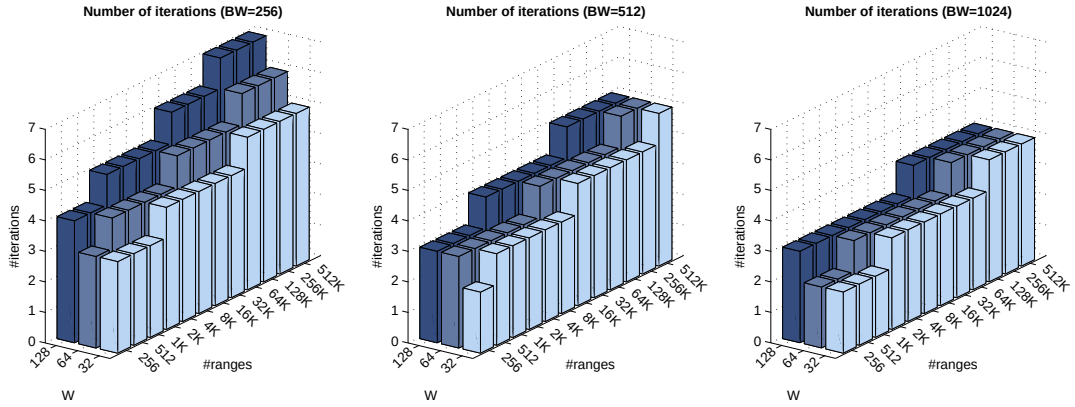


Figure 4.14: The number of iterations of the synthesized Range Trie designs. They were decided based on the depth of the respective Range Trie structure (see Table 4.7).

Although the number of iterations scales well, it is desirable to have as few as possible iterations. This is desired for two reasons. First, less iteration stages means a narrower pipeline with lower latency. Secondly, the addition of even just one iteration requires a respective memory unit which size gets even bigger for deeper iteration levels. The absolute lookup latency and the memory requirements for the synthesized designs are depicted in Figures 4.15 and 4.16.

The reason for presenting the absolute lookup latency results is that in case a quick match is required instead of a high throughput, then a different design point may be better to be chosen. From Figure 4.15 it may be seen that there are cases were different design points share almost the same absolute lookup latency.

Regarding the memory requirements (Figure 4.16), they were found to scale well, except for the larger lookup tables (64K-512K ranges) and wider incoming addresses (when already using a large lookup table). The worst-case encountered memory requirements (when having 512K ranges and 128-bits wide addresses) were approximately 24MBytes.

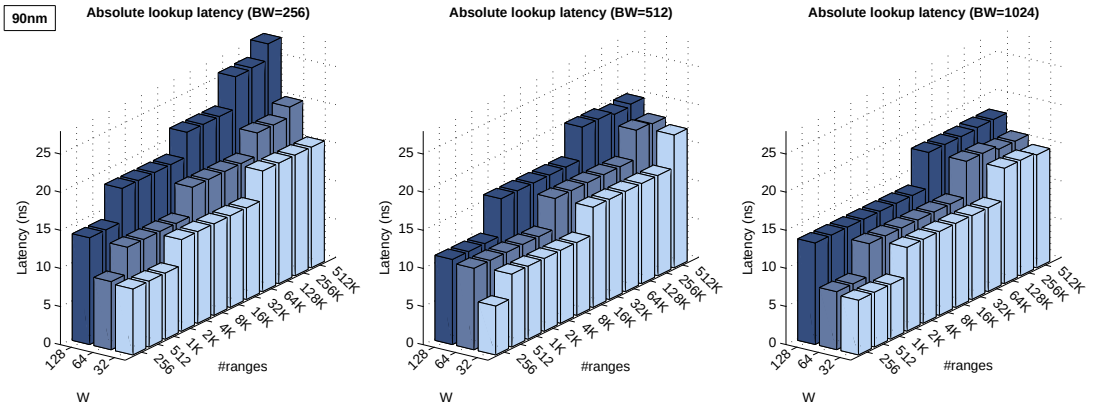


Figure 4.15: The absolute lookup latency of the synthesized Range Trie designs using 90nm ASIC technology. It is calculated based on the total number of pipeline stages $(L-1)*2$, where L is the Range Trie structure depth, and the duration of each clock cycle.

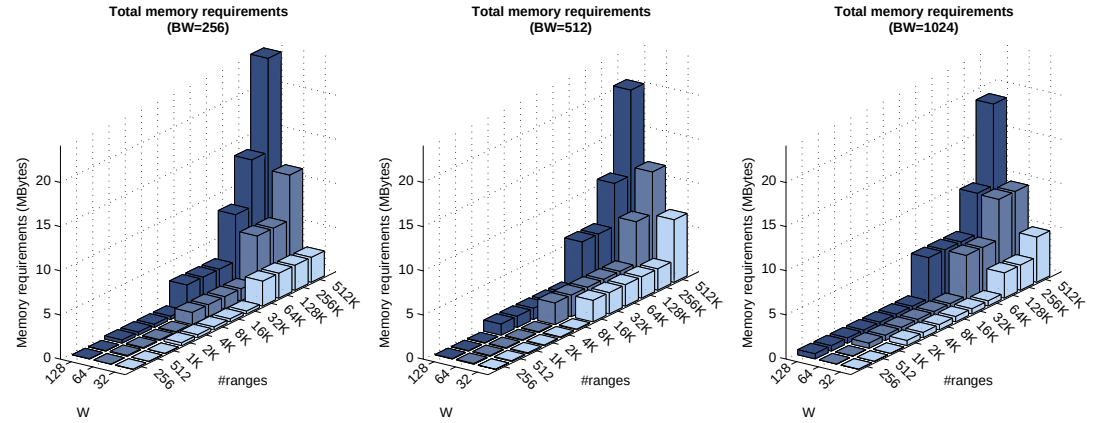


Figure 4.16: The Memory requirements of the synthesized Range Trie designs. They are calculated based on the required memory entries per iteration (see Table 4.7) for the respective Range Trie structure.

In the best-case the memory requirements were in the order of a few KBytes. The area and power consumption was found to scale in the same manner as the memory requirements.

The Range Trie hardware design could not be compared directly to other existing hardware solutions due to the different platforms that were used by others. Other solutions were implemented on various FPGAs, others using different ASIC technologies, while others were just software-based solutions. To get an idea where the Range Trie stands compared to other solutions, it may be compared based on the characteristics that matter most to the Range Trie design; the number of tree levels and the subsequent memory requirements.

Since the purpose of the Range Trie is to be used for real-life address lookup, it is interesting to see how it compares against existing lookup methods (presented in Section

2.1) when using real routing tables from existing internet backbone exchange points. In Figures 4.17 and 4.18 the number of Range Trie structure levels and its memory requirements are compared respectively against other methods. It may be seen that in both cases the Range Trie outperforms the existing solutions. Moreover, taking into account that the number of iterations and the memory requirements of the Range Trie scale efficiently in terms of the incoming address width and the lookup table size, places the Range Trie method as a future-proof address lookup solution.

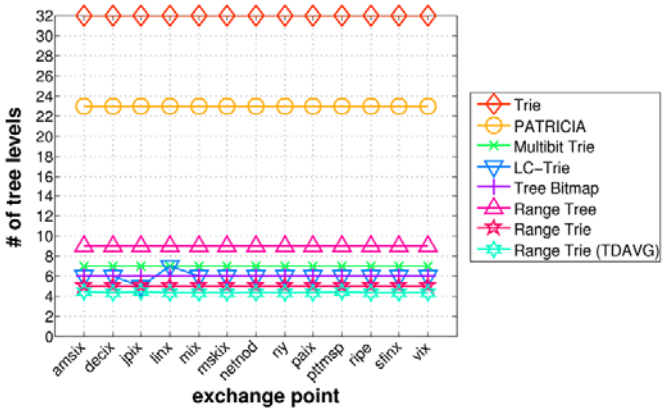


Figure 4.17: Comparison of the number of tree levels of various lookup methods under real routing tables of some actual backbone internet exchange points [25].

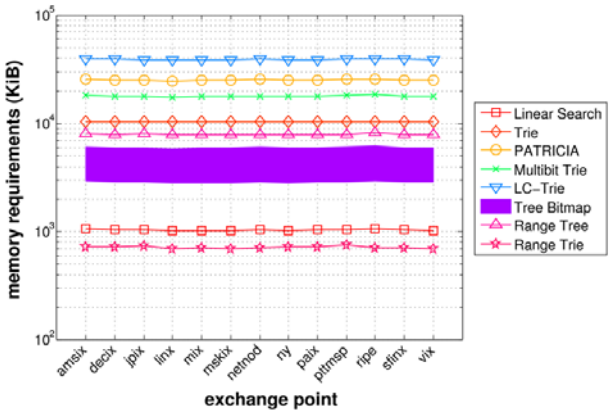


Figure 4.18: Comparison of the memory requirements of various lookup methods under real routing tables of some actual backbone internet exchange points [25].

To conclude this chapter, the proposed Range Trie design was evaluated in terms of operating frequency, power consumption and area. It was found out that it performs the address lookup process with low latency, high throughput, small memory requirements and that all of these properties scale efficiently when the address width and the number of ranges in the lookup table increase. Furthermore, as the design is highly parameterizable,

it offers the possibility of exploring the design space to choose a Range Trie instance that fits the specific needs of the system to be used in.

4.4 Summary

In this chapter the proposed Range Trie design was evaluated in terms of operating frequency, power consumption and area, after synthesizing a variety of Range Trie instances for 90nm and 130nm ASIC technologies. The target was to investigate on the Range Trie's performance and its scalability for increasing address widths and lookup tables.

Section 4.1 set up the required background regarding the experimental setup that was used for performing the synthesizes. The Synopsys Design Compiler tool was used for synthesizing the design for 90nm and 130nm ASIC technologies using the respective foundry libraries of UMC-Faraday.

Before evaluating the complete Range Trie design, that is a pipeline of interleaving iteration stages and memory access stages, the iteration stage was evaluated by itself in Section 4.2. A set of variations of the iteration stage were generated and synthesized for every possible values of incoming address width (W) and available memory bandwidth (BW). This was done to investigate the effect of various design choices on the iteration stage's performance. In particular, we investigated on the cost of the bound alignment subtractor (fixed-width subtractor, variable-width subtractor or eliminated subtractor) and the chosen memory organization scheme. This investigation targeted mainly the operating frequency. The variations of the iteration stages performed as expected. Out of the five generated variations we opted to choose the one that has a fixed-width bound alignment subtractor and that uses the chosen memory organization scheme. It's operating frequency ranged from 478MHz up to 729MHz. The best achieved operating frequency was 1.01GHz for the basic iteration stage variation, where the bound alignment subtractor and the memory addressing hardware were eliminated.

In Section 4.3 the complete Range Trie pipeline design was evaluated. First, a set of representative Range Trie instances was generated to cover a variety of design points. The resulting instances supported incoming address widths of 32, 64 and 128 bits and lookup tables consisting of 256 up to 512K ranges. The Range Trie instances were synthesized and their operating frequency, power consumption and area was reported. All of the findings indicated that the Range Trie method is a future-proof address lookup solution with low latency, high throughput and small memory requirements, while all of these properties scale efficiently when the address width and the number of ranges in the lookup table increase.

In particular, a maximum operating frequency of 694MHz was achieved. Moving from IPv4 addresses (32-bits) to the wider IPv6 ones (128-bits) resulted on average in a 100MHz decrease of the operating frequency. Using larger lookup tables resulted on very small decreases of operating frequency. The worst achieved operating frequency was 442MHz, when the lookup table size and incoming address width were maximized. At the same time, the memory requirements were found to scale also well, except for the larger lookup tables (64K-512K ranges) and wider incoming addresses (when already using a large lookup table). The worst-case encountered memory requirements were approximately 24MBytes. The area and power consumption was found to scale in the same

manner as the memory requirements. Also, it was found out that the Range Trie design may support OC-3072 (160Gbps) wire speeds even for 512K ranges. Finally, the Range Trie method was compared against existing lookup methods in real-life conditions and it was shown that the number of Range Trie structure levels and its memory requirements outperform the existing solutions.

5 Conclusions

The rapid growth of internet traffic, the increase in the number of network devices and the subsequent growing size of routing tables make more difficult for address lookup to keep pace with the increasing need for faster processing rates posed by the technological advancements in communication speed and bandwidth. Furthermore, the transition from the 32-bits wide IPv4 addresses to 128-bits wide IPv6 addresses demands for address lookup solutions that may scale efficiently in terms of the address width. Unfortunately, the currently available lookup solutions have started lagging behind and address lookup tends to become the bottleneck in the systems that they use it. This indicated that a method is desired for address lookup that has *low latency*, *high throughput* and *low memory requirements*. At the same time, these properties should *scale efficiently* when the address width and/or the number of ranges increases.

The problem of designing such a method was the main issue of this thesis. This thesis focused on the Range Trie algorithm introduced by I. Sourdis in [26] that promises to solve these problems. In particular, the Range Trie method was designed here for a hardware implementation in order to end up with an efficient hardware design and implementation of the Range Trie.

This chapter summarizes in Section 5.1 the issues that were addressed by this thesis, along with the proposed solution and the findings of the performed research. Section 5.2 presents the contributions of the performed research, while Section 5.3 concludes this thesis with some suggestions for future works.

5.1 Summary

In Chapter 2, a representative set of related designs and algorithms for address lookup that exists in the literature was presented. These were a mixture of algorithmic approaches to address lookup and hardware-targeting address lookup designs. The algorithmic approaches were classified based on the dimension of the performed search ("search on length" or "search on values") and on the type of the search traversal (sequential or binary). Because these approaches have started lagging behind and lack efficient scaling properties, the novel address lookup approach of the Range Trie was decided to be designed for a hardware implementation in this thesis. The Range Trie is between the "search on length" and "search on values" approaches and delivers a method with (a) low latency, (b) high throughput, (c) low memory requirements and (d) good scalability in terms of address width and lookup table size. The Range Trie is a specific tree structure with a multitude of nodes per level, along with a specific algorithm to traverse the tree structure. In each node, comparisons are performed on parts of addresses based on 5 Range Trie rules that formulate the fundamental concepts behind the Range Trie development. Based on the 5 rules, the parts of addresses to compare are minimized

by sharing common prefix/suffix comparisons, by omitting unnecessary comparisons and by aligning the addresses to be compared. Minimizing the parts of addresses to compare, results in a higher utilization of the given memory bandwidth, in an increase of the branches per node and in a decrease of the Range Trie depth. Generating a Range Trie structure is done in an automated way based on heuristic methods that exploit the 5 Range Trie rules.

The Range Trie method is an iterative method, where (a) a node is visited, (b) a number of comparisons is performed on parts of the incoming address and (c) the next node to visit is decided, until a leaf node is reached and a match is reported. In Chapter 3, the complete hardware design of the Range Trie address lookup method was presented in a bottom-up fashion. Starting from the iteration functional unit and the chosen memory structure, a complete pipeline was formed. All required hardware units were described that are needed for designing every possible instance of a Range Trie, according to its parameters. During the design, the effort was to exploit the Range Trie method inherit characteristics into building a fast, efficient and scalable hardware design with low memory requirements.

Specifically, also in Chapter 3, the hardware design of the Range Trie iteration was presented. This hardware performs the following four actions: (1) select parts of the incoming address for comparison and (possibly) perform address bound alignment, (2) perform the comparisons and (possibly) the shared prefix/suffix comparisons, (3) interpret the comparison results and (4) decide which outgoing branch of the current node to follow, according to the comparison results, following the Range Trie method's decision criteria. During the iteration design process the necessary signals that control the iteration were defined, according to the current node information. This resulted in the formulation of the way that the current node information must be represented for the correct operation of a given iteration. Afterwards, the memory structure was defined where the Range Trie structure must be stored. This memory structure is used for retrieving the node information (in every iteration) stored as a specified data structure. In particular, a memory structure was defined that consists of as many memory levels as the Range Trie levels. The way to organize the Range Trie nodes into the memory levels was presented, along with the way to address the memory levels in order to retrieve the next node to visit.

In Chapter 4 the proposed Range Trie design was evaluated in terms of operating frequency, power consumption and area. First, the iteration stage was evaluated by itself. A set of variations of the iteration stage were generated and synthesized to investigate the effect of various design choices on the iteration stage's performance, such as the address alignment subtractor and the memory addressing scheme. The variations of the iteration stages performed as expected.

Afterwards, a set of representative Range Trie instances was generated and synthesized for 90nm and 130nm ASIC technologies to cover a variety of design points. The resulting instances supported incoming address widths of 32, 64 and 128 bits and lookup tables consisting of 256 up to 512K ranges. Based on the resulting performance metrics, we investigated on the Range Trie's performance and its scalability for increasing address widths and lookup tables. All of the findings indicated that the Range Trie method is a future-proof address lookup solution with low latency, high throughput and small mem-

ory requirements, while all of these properties scale efficiently when the address width and the number of ranges in the lookup table increase.

In particular, a maximum operating frequency of 694MHz was achieved. Moving from IPv4 addresses (32-bits) to the wider IPv6 ones (128-bits) resulted on average in a 100MHz decrease of the operating frequency. Using larger lookup tables resulted on very small decreases of operating frequency. The worst achieved operating frequency was 442MHz, when the lookup table size and incoming address width were maximized. At the same time, the memory requirements were found to scale also well, except for the larger lookup tables (64K-512K ranges) and wider incoming addresses (when already using a large lookup table). The worst-case encountered memory requirements were approximately 24MBytes. The area and power consumption were found to scale in the same manner as the memory requirements. Also, it was found out that the Range Trie design may support OC-3072 (160Gbps) wire speeds even for 512K ranges. Finally, the Range Trie was compared against existing lookup methods in real-life conditions and it was shown that the number of Range Trie method levels and its memory requirements outperform the existing solutions.

5.2 Contributions

The main issue addressed by this thesis was the design and implementation of the Range Trie algorithm in a way that exploits optimally the inherit characteristics of the Range Trie method, which are: (a) low lookup latency, (b) high throughput, (c) low memory requirements, (d) good scalability of (a)-(c) in terms of the lookup address width and number of address ranges.

The contributions of this thesis are:

- **A hardware design of the Range Trie algorithm:** For the first time, the Range Trie method was successfully designed for a hardware implementation. All the required units were designed to form the complete Range Trie design. During the design process, extra effort was put to ensure the efficiency of the designed hardware. Every element was designed while having in mind to maintain the low latency, the high throughput and the low memory requirements. The resulting design is parameterizable in terms of address width, memory bandwidth and number of processing stages in order to accommodate the address lookup needs of the application under consideration. At the same time, the proposed design was well structured to allow easily its generation according to the parameters.

- **A complete design flow for hardware implementation and validation:** A complete design flow was created for generating Range Trie instances. Starting from a Range Trie structure, the design parameters are defined and a Range Trie hardware design may be generated that is tailored to that specific Range Trie structure. Alongside, all the necessary issues were addressed regarding the ASIC implementation, the validation and the configuration of the hardware design.

- **Evaluation of the Range Trie design:** A set of representative Range Trie instances was generated and synthesized for 90nm and 130nm ASIC technologies to

cover a variety of design points. The resulting instances supported incoming address widths of 32, 64 and 128 bits and lookup tables consisting of 256 up to 512K ranges. Each design point was evaluated in terms of operating frequency, area, power consumption and memory requirements. This design space exploration investigated and showed the Range Trie's scalability for increasing address widths and lookup tables. Apart from that, the evaluation presented the trade-offs between different Range Trie instances in order to allow the selection of the suitable instance for a particular application. All of these placed the Range Trie method as a future-proof address lookup solution

5.3 Future suggestions

This section concludes this thesis by suggesting some ideas for future works and improvements on the Range Trie hardware design. Some of these proposals for future work are the following:

- **Incremental updating**: In order to make the Range Trie method more complete, the support for *updating* the stored Range Trie structure must be added. The updating mechanism must be fast enough and be able to handle fast update rates. The benefits of the Range Trie design should be preserved. At the same time, the updating mechanism should be designed in a way that does not block the lookup operation while updating, or at least block it for the minimum possible time.

- **Longest prefix matching**: The Range Trie design may be augmented to support also longest prefix matching. The current design supports only exact range matching.

- **Generalizing the design**: It would also be interesting to investigate on the possible benefits of making the Range Trie hardware more general, in order to become as general as the Range Trie method. As an example question, having a completely variable-width comparator that could compare any possible part of an incoming address would yield any improvements, without hampering the design's performance, or not?

- **Memory organization schemes**: Other memory organization schemes may be developed that allow for a more uniform memory distribution, where all memory levels have a similar smaller size. This could be achieved by using a circular pipeline. Alongside, other memory addressing schemes could be investigated, such as using hashing to address a memory, in order to avoid the offset addition and utilize more the memory units.

- **Pipeline improvements**: The used pipeline may be re-invented by splitting the current iteration stage into more stages in order to increase the operating frequency. Another pipeline approach could be to use a circular pipeline where a lookup may be initiated at any possible stage.

- **ASIC design process**: Finally, the whole synthesis process may be done more precisely by reaching up to the final steps of the ASIC design process and by going through other steps, like exact floorplanning. At the same time, even smaller ASIC technologies may be used to reach higher operating frequencies. It would also be beneficial to use more appropriate memory units that do not lead in excessive area and power results.

Bibliography

[1] *The square of the Microelectronics Training Center of IMEC*, November 2008, `http://www.mtc-online.be/`.

[2] Himanshu Bhatnagar, *Advanced ASIC chip synthesis: Using synopsys design compiler, physical compiler and prime time*, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[3] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Commun. ACM **13** (1970), no. 7, 422–426.

[4] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese, *An improved construction for counting bloom filters*, ESA'06: Proceedings of the 14th conference on Annual European Symposium (London, UK), Springer-Verlag, 2006, pp. 684–695.

[5] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi, *IP address lookup made fast and simple*, Tech. report, 1999.

[6] Ruben de Smet, *Range trie heuristics for variable-size address region lookup*, Master's thesis, Computer Engineering, Delft University of Technology, 2009.

[7] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink, *Small forwarding tables for fast routing lookups*, Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '97) (New York, NY, USA), ACM, 1997, pp. 3–14.

[8] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor, *Longest prefix matching using bloom filters*, Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03) (New York, NY, USA), ACM, 2003, pp. 201–212.

[9] Sarang Dharmapurikar, Haoyu Song, Jonathan Turner, and John Lockwood, *Fast packet classification using bloom filters*, Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (ANCS '06) (New York, NY, USA), ACM, 2006, pp. 61–70.

[10] William N. Eatherton and Zubin D. Dittia, *Tree bitmap data structures and their use in performing lookup operations*, United States Patent 7,249,149 B1, July 2007.

[11] V. Fuller, T. Li, J. Yu, and K. Varadhan, *Classless inter-domain routing (CIDR): an address assignment and aggregation strategy*, 1993.

[12] P. Gupta, S. Lin, and N. McKeown, *Routing lookups in hardware at memory access speeds*, vol. 3, Mar-2 Apr 1998, pp. 1240–1247 vol.3.

[13] P. Gupta and N. McKeown, *Algorithms for packet classification*, Network, IEEE **15** (2001), no. 2, 24–32.

[14] Pankaj Gupta and Nick Mckeown, *Packet classification using hierarchical intelligent cuttings*, Proceedings of Hot Interconnects VII, 1999, pp. 34–41.

[15] Jahangir Hasan and T. N. Vijaykumar, *Dynamic pipelining: making IP-lookup truly scalable*, SIGCOMM Comput. Commun. Rev. **35** (2005), no. 4, 205–216.

[16] Manolis G. H. Katevenis, *The future of computing: Essay in memory of Stamatis Vassiliadis*, ch. Interprocessor Communication Seen As Load-Store Instruction Generalization, pp. 55–68, Delft, The Netherlands, September 28, 2007.

[17] S. Kaxiras and G. Keramidas, *IPStash: a set-associative memory approach for efficient IP-lookup*, vol. 2, March 2005, pp. 992–1001 vol. 2.

[18] Sailesh Kumar, Michela Becchi, Patrick Crowley, and Jonathan Turner, *CAMP: fast and efficient IP lookup architecture*, Proceedings of the 2006 ACM/IEEE symposium on Architecture for Networking and Nommunications Nystems (ANCS '06) (New York, NY, USA), ACM, 2006, pp. 51–60.

[19] Donald R. Morrison, *PATRICIA: Practical algorithm to retrieve information coded in alphanumeric*, J. ACM **15** (1968), no. 4, 514–534.

[20] S. Nilsson and G. Karlsson, *IP-address lookup using LC-tries*, IEEE Journal on Selected Areas in Communications **17** (1999), no. 6, 1083–1092.

[21] Behrooz Parhami, *Computer arithmetic: Algorithms and hardware designs*, Oxford University Press, New York, 2000.

[22] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, *Survey and taxonomy of IP address lookup algorithms*, Network, IEEE **15** (2001), no. 2, 8–23.

[23] Keith Sklower, *A tree-based packet routing table for berkeley UNIX*, Proceedings of 1991 Winter USENIX Conference (Dallas, Texas), January 21-25 1991.

[24] Haoyu Song, Jonathan Turner, and Sarang Dharmapurikar, *Packet classification using coarse-grained tuple spaces*, Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (ANCS '06) (New York, NY, USA), ACM, 2006, pp. 41–50.

[25] Ioannis Sourdis, Ruben de Smet, and Georgi N. Gaydadjiev, *Range trees with variable length comparisons*, 2009.

[26] Ioannis Sourdis, Ruben de Smet, George Stefanakis, and Georgi N. Gaydadjiev, *Data structure, method and system for address lookup*, Patent Reference NL2002799, filed on 24th April 2009.

[27] E. Spitznagel, D. Taylor, and J. Turner, *Packet classification using extended TCAMs*, Nov. 2003, pp. 120–131.

[28] V. Srinivasan and G. Varghese, *Fast address lookups using controlled prefix expansion*, ACM Transactions on Computer Systems **17** (1999), no. 1, 1–40.

[29] George Varghese Subhash Suri and Priyank Warkhede, *Multiway range trees: scalable IP lookup with fast updates*, IEEE Global Telecommunications Conference (GLOBECOM 2001) **3** (2001), 1610–1614.

[30] Bret Swanson and George Gilder, *Estimating the exaflood - the impact of video and rich media on the internet - a 'zettabyte' by 2015?*, January 2009.

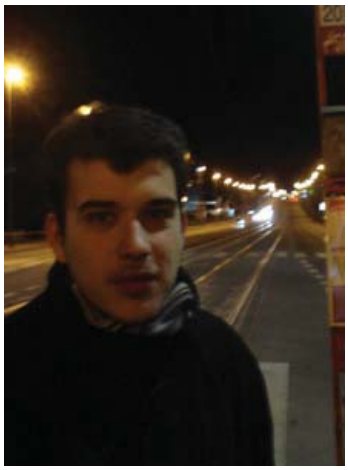
[31] David E. Taylor, *Survey and taxonomy of packet classification techniques*, ACM Comput. Surv. **37** (2005), no. 3, 238–275.

[32] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, *Scalable high-speed prefix matching*, ACM Transactions on Computer Systems **19** (2001), no. 4, 440–482.

[33] Priyank Warkhede, Subhash Suri, and George Varghese, *Multiway range trees: scalable IP lookup with fast updates*, Computer Networks **44** (2004), no. 3, 289–303.

[34] T.Y.C. Woo, *A modular approach to packet classification: algorithms and results*, vol. 3, Mar 2000, pp. 1213–1222 vol.3.

Curriculum Vitae



Georgios Stefanakis was born in Athens, Greece, on the 16th of November in 1984. In 2002, he graduated from the Platon High School in Athens with GPA 19.2/20.0 (Excellent).

The same year he enrolled as an undergraduate student into the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens, Greece. The duration of his bachelor studies was 4 years and he received his Bachelor in August 2006 with GPA 7.4/10.0 (Very Good). His thesis title was "Data mining from web for use in personalized web applications" under the supervision of Professor Yannis Ioannidis.

In September 2006, he enrolled as a master student into the MSc program of the Computer Engineering laboratory of the Delft University of Technology where he is an MSc student until now. During the development of his master thesis project he filled a patent application entitled "Data Structure, Method and System for Address Lookup" with the collaboration of Ioannis Sourdis, Ruben de Smet and Georgi N. Gaydadjiev.

His research interests include, but are not limited to, networks-on-chip, reconfigurable hardware, fault-tolerant computing, embedded systems and operating systems.