RMIT
UNIVERSITY

A Null Convention Logic based platform for high speed low energy IP packet forwarding

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

Prashant DABHOLKAR
B.E. Electronics Engineering, Mumbai University, India
MTech, Electrical and Electronics Communication Engineering, IIT, Kharagpur, India

School of Engineering
College of Science Engineering and Health
RMIT University

September 24, 2018

Declaration of Authorship

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Prashant Prabhakar Dabholkar

September 24, 2018

"Do not go where the path may lead. Go instead where you may leave a trail."

Ralph Waldo Emerson

Abstract

By 2020, it is predicted that there will be over 5 billion people and 38.5 billion Internet-of-Things devices on the Internet. The data generated by all these users and devices will have to be transported quickly and efficiently. Routers forming the backbone of this Internet already support multiple 100 Gbps ports meaning that they would have to perform upwards of 200 Million destination addresses lookups per second in the packet forwarding block that lies in the router 'data-path'. At the same time, there is also a huge demand to make the network infrastructure more energy efficient.

The work presented in this thesis is motivated by the observation that traditional synchronous digital systems will have increasing difficulty keeping up with these conflicting demands. Further, with reducing device geometries, extremes in "process, voltage and temperature" (PVT) variability will undermine reliable synchronous operation. It is expected that asynchronous design techniques will be able to overcome many of these problems and offer a means of lowering energy while maintaining high throughput and low latency. This thesis investigates existing address lookup algorithms and investigates the possibility of combining various approaches to improve energy efficiency without affecting lookup performance. A quasi delay-insensitive asynchronous methodology - Null Convention Logic (NCL) - is then applied to this combined design. Techniques that take advantage of the characteristics of the design methodology and the lookup algorithm to further improve the area, energy and latency characteristics are also analysed.

The IP address lookup scheme utilised here is a recent algorithmic approach that uses compact binary-tries and was selected for its high memory efficiency and throughput. The design is pipelined, and the prefix information is stored in large RAMs. A Boolean synchronous implementation of the algorithm is simulated to provide an initial performance benchmark. It is observed that during the address lookup process nearly 68% of the trie accesses are to nodes that contained no prefix information. Bloom filter structures that use non-cryptographic hashes and single-bit memory are introduced into the address lookup process to prevent these unnecessary accesses, thereby reducing the energy consumption. Three non-cryptographic hashing

algorithms (CRC32, Jenkins and Murmur) are also analysed for their suitability in Bloom filters, and the CRC32 is found to offer the most suitable trade-off between complexity and performance.

As a first step to applying the NCL design methodology, NCL implementations of the hashing algorithms are created and evaluated. A significant finding from these experiments is that, unlike Boolean systems, latency and throughput in NCL systems are only loosely coupled. An example Jenkins hash implementation with eight pipeline stages and a cycle time of 3.2 ns exhibits a total latency of 6 ns, whereas an equivalent synchronous implementation with a similar clock period exhibits a latency of 25.6 ns. Further investigations reveal that completion detection circuits within the NCL pipelines impair throughput significantly. Two enhancements to the NCL circuit library aimed particularly at optimising NCL completion detection are proposed and analysed. These are shown to enable completion detection circuits to be built with the same delay but with 30% smaller area and about 75% lower peak current compared to the conventional approach using gates from the standard NCL library. An NCL SRAM structure is also proposed to augment the conventional 6-T cell array with circuits to generate the handshaking signals for managing the NCL data flow. Additionally, a dedicated column of cells called the Null-storage column is added, which indicates if a particular address in the RAM stores no Data, i.e., it is in its Null state. This additional hardware imposes a small area overhead of about 10% but allows accesses to Null locations to be completed in 50% less time and consume 40% less energy than accesses to valid Data locations.

An experimental NCL-based address lookup system is then designed that includes all of the developed NCL modules. Statistical delay models derived from circuit-level simulations of individual modules are used to emulate realistic circuit delay variability in the behavioural modules written in Verilog. Simulations of the assembled system demonstrate that unlike what was observed with the synchronous design, with NCL, the design that does not employ Bloom filters, but only the Null-storage column RAMs for prefix storage, exhibits the smallest area on the chip and also consumes the least energy per address lookup. It is concluded that to derive maximum benefit out of an asynchronous design approach; it is necessary to carefully select the architectural blocks that combine the peculiarities of the implemented algorithm with the capabilities of the NCL design methodology.

Acknowledgements

While this PhD is a very personal journey, there are many individuals I have encountered along the way, who have influenced how it has unfolded.

Most importantly, the journey would not have been possible without the support and guidance of my supervisor A/Prof Paul Beckett. I was fortunate to have worked with a supervisor who was always available and willing to treat me as a colleague rather than a student. I must have spent countless hours discussing everything from the performance of transistors inside NCL gates to NCL system architecture. His inputs on "What is research?" were also invaluable whenever I felt lost.

I would also like to thank Dr Omid Kavehei and Dr Glenn Matthews who were on my supervisory team at different times. They provided me with valuable guidance and direction. I am grateful to Dr Karina Gomez Chavez for the discussions on packet processing and IP networks and for her inputs on the manuscripts I wrote during this work. A special thanks to Justin Spangaro and Wave Semiconductor Inc. for providing the NCL compilation and synthesis tools.

The tools by themselves would be of no use without the people who demonstrated how to use them and also created some of the designs that helped me in my work. I am thankful to Matthew Kim and Jeeson Kim for creating the NCL gate library, Kashfia Haque and Andrew Przybylski for their work on reconfigurable NCL processors, Renuka Sovani and Jing Yu for being ever present to resolve my queries with the Cadence tools and to Aratrika Ghosh, Nhan Truong Duy, Long Tran and Duc Nguyen for their company. All of these individuals not only helped me professionally but also became very close friends.

I am also thankful to Dr Sathya Chandrasekharan, Dr Kagiso Magowe, Dr Akram Hourani, A/Prof Madhu Bhaskaran and A/Prof Sharath Shriram who either through personal interactions or their actions showed me how to excel at research. A special mention of the amazingly helpful research support officers, Beth, Laurie and Claire in the School of Engineering office.

Thanks are also due to my parents who set me up on this path of learning and curiosity and to my father-in-law for supporting me when I needed him. A big thank you to my wife Sulu and my son Ekagra for understanding what I was going through and for putting up with my absence on weekends.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Internet - Challenges for the future . . . 1
- 1.2 Internet Routers . . . 3
  - 1.2.1 Overview . . . 3
  - 1.2.2 Performance vs Energy . . . 4
- 1.3 Asynchronous logic . . . 6
- 1.4 Motivation and scope . . . 7
- 1.5 Research questions and methodology . . . 8
- 1.6 Outcomes and Contributions . . . 10
  - 1.6.1 Publications . . . 11
- 1.7 Organisation . . . 11

2 Background and literature review 13
- 2.1 Address lookup in Internet routers . . . 14
  - 2.1.1 TCAM-based IP address lookup . . . 15
  - 2.1.2 Algorithmic SRAM-based IP address lookup . . . 16
- 2.2 Bloom filter theory . . . 18
  - 2.2.1 Bloom filter operation . . . 18
  - 2.2.2 Hash function requirements . . . 22
  - 2.2.3 Hash function selection . . . 23
- 2.3 Asynchronous logic systems . . . 24
  - 2.3.1 Classification of asynchronous techniques . . . 25
  - 2.3.2 Null Convention Logic (NCL) . . . 28
  - 2.3.3 State of the art in NCL-based systems . . . 33
  - 2.3.4 NCL limitations . . . 34

3 Compact-trie with Bloom filters in Boolean logic 37
- 3.1 Enhanced Compact-trie with epsilon nodes . . . 38
  - 3.1.1 Architecture . . . 38
  - 3.1.2 Trie creation and trie search algorithms . . . 41
- 3.2 Enhanced Compact-trie with epsilon links and Bloom filters . . . 46
  - 3.2.1 Architecture . . . 46
  - 3.2.2 Operation . . . 50
- 3.3 Hardware design considerations . . . 52
  - 3.3.1 Trie node structure . . . 52

3.3.2 Pipelined Bloom Filter . . . 53
3.3.3 System architecture . . . 54
3.4 Results and discussions . . . 55
3.4.1 Experimental setup . . . 55
3.4.2 Software simulations . . . 57
3.4.2.1 Effect of P_{trie} on memory utilisation of $E - Ctrie_{\epsilon}$ . . . 58
3.4.2.2 $E - Ctrie_{\epsilon}$ versus binary trie search . . . 61
3.4.2.3 $E - Ctrie_{\epsilon}$ and Bloom filters . . . 62
3.4.3 Circuit simulations for target FPGA . . . 65
3.4.3.1 Resource utilisation: Monolithic versus Pipelined Bloom Filter . 66
3.4.3.2 Total Resource utilisation and power consumption with pipelined Bloom filters . . . 66
3.4.3.3 Power distribution down trie levels and targeted Bloom filtering 67

4 Bloom Filters and Hashing Functions 73
4.1 Hash function implementation . . . 74
4.2 Hash function - performance evaluation . . . 79
4.3 Completion Detection circuits - architecture . . . 86
4.3.1 Conventional Completion Detection circuits with NCL gates (CoCD) . . . 86
4.3.2 Completion Detection with complementary gates and external feedback (CD-CG) . . . 88
4.3.3 Completion Detection with Complementary Smith-gates and external feedback (CD-CSG) . . . 91
4.4 Completion detection circuits - performance evaluation . . . 91

5 NCL SRAM with early completion detection and Null-storage column 99
5.1 NCL SRAM32x16 unit . . . 100
5.1.1 Read and Write completion detection . . . 101
5.1.2 SRAM unit with NULL storage . . . 103
5.1.3 SRAM32x16 unit Write and Read operation . . . 105
5.1.3.1 Write operation . . . 106
5.1.3.2 Read Operation . . . 108
5.2 1024x16 SRAM with address decoder, read-write completion and Null-storage column . . . 109
5.3 Simulation and Performance Measurements . . . 111
5.3.1 Write Performance . . . 112
5.3.2 Read Operation . . . 114

6 Compact trie with Bloom filters in Null Convention Logic 117
6.1 Design considerations . . . 118
6.2 Results and Discussion . . . 121
6.2.1 Simulation setup . . . 121
6.2.2 Simulation Results . . . 122
6.2.2.1 Cycle Time . . . 122
6.2.2.2 Energy Consumption . . . 125
6.2.2.3 Area . . . 127

7 Conclusion and the way forward 133

A Code listing 139

Bibliography 141

List of Figures

1.1 Projected growth of the internet from 2017 to 2021 Source: CISCO Visual Networking Index Report for 2017 . . . 2
1.2 Router block diagram showing the routing and forwarding function interacting with the Forwarding Information Base (FIB) . . . 3
1.3 Distribution of packet sizes in anonymised internet traces captured at core routers. 4

2.1 CAM-based implementation of the example prefix table of Table 2.1 . . . 16
2.2 Binary trie implementation of the example prefix table of Table 2.1 . . . 17
2.3 Difference in the TCAM-based and algorithmic SRAM-based approaches to destination address lookup . . . 18
2.4 Showing bits sets in a 16-bit Bloom filter programmed with three elements x, y and z and the results of querying the Bloom filter for two test elements $q1$ and $q2$ 20
2.5 False-positive probability characteristics of Bloom filter . . . 21
2.6 Simplified block diagram of an asynchronous system with timing diagrams showing the signal flow in 2-phase and 4-phase handshaking protocols . . . 25
2.7 Flow of Null and Data wavefronts through an NCL 2-of-3 threshold gate (TH23) 29
2.8 TH23 gate transistor schematic . . . 30
2.9 NCL gate library . . . 30
2.10 NCL pipeline model . . . 31
2.11 Wavefront propagation in 4-cycle NCL pipeline . . . 32

3.1 Process of insertion of prefix P10 in the $E - Ctrie_{\epsilon}$ structure . . . 44
3.2 Complete Enhanced Compact-trie with epsilon links ($E - Ctrie_{\epsilon}$) after insertion of all prefixes from the Enhanced Compact Prefix Table of Table 3.2 . . . 45
3.3 CRC8 generator block diagram and programmed Bloom filter . . . 49
3.4 Node structure in an $E - Ctrie_{\epsilon}$ . . . 53
3.5 FPGA block diagram of an $E - Ctrie_{\epsilon}$ implementation with Bloom filter with details of the match stage block . . . 56
3.6 Distribution of prefixes, length of $|x|$ and $|z|$ in the $E - Ctrie_{\epsilon}$ implementation of real routing tables . . . 59
3.7 Average number of memory accesses for varying Bloom filter scaling factors and P_{trie} values . . . 64

4.1 Murmur Hash block diagram . . . 76
4.2 Jenkins Hash block diagram . . . 77
4.3 CRC32 hash block diagram . . . 78
4.4 Test setup for measuring latency and throughput of an NCL design . . . 78
4.5 Cycle time variation at $0\,^{\circ}\mathrm{C}$, $85\,^{\circ}\mathrm{C}$, slow and fast process corners for Data and Null wavefronts in NCL implementations of hash functions . . . 80
4.6 Latency and cycle time of a 32-bit Booth Wallace multiplier for three different pipelining conditions . . . 82
4.7 Jenkins hash - latency and cycle time variation at $27\,^{\circ}\mathrm{C}$ for a set of test vector . . 84
4.8 Jenkins hash - supply current drawn during hash computation . . . 85
4.9 (a) Conventional TH44 gate schematic (b) Completion detection circuit using conventional THxx gates . . . 87

4.10 (a) Complementary TH44 gate schematic (b) Complementary TH44 gate symbol (c) Completion detection circuit using complementary and conventional THxx gates (CoCD) . . . 89
4.11 Input and output waveforms for the Complementary TH44 gate . . . 90
4.12 (a) THC4D gate schematic (b) THC4D gate symbol (c) Completion detection circuit using complementary Smith gates (CD-CSG) . . . 92
4.15 Supply current drawn by the three completion detection circuits during the Data-to-Null and Null-to-Data transitions at the input . . . 97

5.1 Circuit diagram of the 6-Transistor SRAM cell and organisation of the SRAM cell array . . . 100
5.2 Block diagram of the 32 words x 16 bits SRAM unit . . . 101
5.3 Schematic of the circuits generating the WRCMPLT and WRINCMPLT signals . . 103
5.4 Architecture block diagram of the SRAM unit (32 words x 16 bits) with a Null storage column . . . 104
5.5 Waveforms showing the order in which the data and control signals in the SRAM32x16 unit toggle when a Data value is being written to a location that initially contained a Null value . . . 107
5.6 Block diagram of the 1024x16 bit NCL RAM . . . 110

6.1 NCL wrapper around a Boolean module to produce the 'NCL-in-Verilog' modules used for simulations . . . 121
6.2 Wavefront propagation diagram demonstrating the effect of reading a Null-storage column, on the cycle time of the system . . . 124
6.3 Observed cycle time (with 10000 test vectors) at the input of an NCL implementation of the $E - Ctrie_{\epsilon}$ with and without Null-storage column (MGM routing table) . . . 126
6.4 Distribution of cycle times at the input of an NCL implementation of the $E - Ctrie_{\epsilon}$ implementation (a) with Null-storage column and (b) without Null-storage column (MGM routing table) . . . 126
6.5 Energy consumption down trie levels in an $E - Ctrie_{\epsilon}$ implementations of both the (a) LYS and (b) MGM tables for the following cases: **WOBF_WONC**: without Bloom filter and without Null-storage column, **WBF_WONC**: with Bloom filter and without Null-storage column and **WOBF_WNC**: without Bloom filter but with Null-storage column, **WBF_WNC**: with Bloom filter and with Null-storage column . . . 128
6.6 Area overhead of the Bloom filter approach and the Null-storage column approach at each level in an $E - Ctrie_{\epsilon}$ implementation for both the LYS and MGM tables. The difference between the two approaches is expressed as a percentage of the Bloom filter area overhead. . . . 129

List of Tables

2.1 Example prefix table . . . 14
2.2 State of art in NCL-based designs . . . 35

3.1 Definition of terms used in the Enhanced Compact Prefix Table (E-CPT) and the Enhanced Compact-trie ($E - Ctrie$) algorithm . . . 39
3.2 Original prefix table with prefix entries decomposed into individual sub-strings according to the $E-Ctrie_{\epsilon}$ algorithm and the resultant Enhanced Compact Prefix Table ($E - CPT$). . . . 40
3.3 Definition of terms used in the $E - Ctrie_{\epsilon}$ algorithm . . . 46
3.4 Enhanced Compact Prefix Table with CRC8 and BF indices . . . 48
3.5 Illustrating the Bloom filter computation during the search process of the key '110001010001' (MSB_k = 1, M_k = 2 and AP_k = '0001010001') in the $E - Ctrie_{\epsilon}$ structure of Figure 3.2 . . . 51
3.6 Sizes of real IPv4 routing tables [103] . . . 57
3.7 Total node count, memory utilisation in Mbits and the depth in the $E - Ctrie_{\epsilon}$ (min/median/max) where the LPM is found for various P_{trie} values . . . 60
3.8 Comparison of binary trie and $E - Ctrie_{\epsilon}$ implementations (P_{trie} = 2) for all routing tables . . . 63
3.9 Comparison of $E - Ctrie_{\epsilon}$ implementations for all routing tables in three cases - without Bloom filter, with monolithic Bloom filter and with pipelined Bloom filter ($P_{trie} = 2$, Bloom filter scaling factor $M = 8$) . . . 65
3.10 Comparison of memory requirement of monolithic Bloom filter and pipelined Bloom filter (BF scaling factor M = 8) . . . 67
3.11 Comparison of $E - Ctrie_{\epsilon}$ implementation for LYS and MGM tables with and without Bloom filters . . . 67
3.12 Comparison of an $E - Ctrie_{\epsilon}$ implementation for the MGM routing table in three cases - with targeted Bloom filtering, with total Bloom filtering and without Bloom filtering . . . 70

4.1 Comparison of NCL implementation of Jenkins and CRC32 hash . . . 83
4.2 Comparison of proposed completion detection circuit architectures against conventional architecture with transistors sized for a fixed delay value of ∼230 ps . . . 96

5.1 Truth table for write completion detection circuit . . . 103
5.2 Write operation: Effect of per-bit write completion detection circuit on latency, cycle time and energy per operation . . . 113
5.3 Write operation: Effect of Null storage column on cycle time, latency and energy consumption per operation . . . 114
5.4 Read operation: Effect of Null storage column on cycle time, latency and energy consumption per operation . . . 115

6.1 Comparison of energy consumption per lookup in NCL implementations of the $E - Ctrie_{\epsilon}$ for the (a) LYS and (b) MGM routing tables . . . 125

Dedicated to the two wonderful women in my life, my mother and my wife, for their unwavering belief and constant encouragement

Chapter 1

Introduction

1.1 Internet - Challenges for the future

The Internet has grown remarkably over the past two decades. From a simple tool of scientists and engineers, it now influences the lives of a significant percentage of the world's population. With the rise of the Internet of Things, it is rapidly becoming a vital component of the world's socio-economic infrastructure.

However, this rapid expansion has thrown up numerous challenges, in particular, those arising due to the large volume of data that needs to be handled efficiently by the individual network nodes. A 2017 CISCO Visual Networking Index Report [1] estimates global IP traffic to be in the order of 1.2 ZB per year [1], generated by an average of 2.3 networked devices per capita each running at an average speed of around 27.5 Mbps. As Figure 1.1 depicts, it is predicted that by 2021 a person connected to the Internet would have about 3.5 devices, each accessing the data at an average speed of 53 Mbps generating total traffic of 3.3 ZB. This rapid growth will inevitably fuel an enormous demand for bandwidth and increasingly complex communication equipment.

At the same time, there has also been a push towards a more energy efficient (*greener*) ICT infrastructure, requiring the network equipment to handle this increased traffic while consuming less energy than before. While remarkable energy efficiency gains have been achieved in the traditional areas of illumination, heating and cooling, the energy density in equipment used to process and communicate data has been increasing every year [2]. A 2012 Greenpeace report analysing the energy efficiency of modern data centres found that if the data centres involved

[1] 1 ZB = 1000 Exabytes [EB]; 1 EB = 1 Billion Gigabytes [GB]

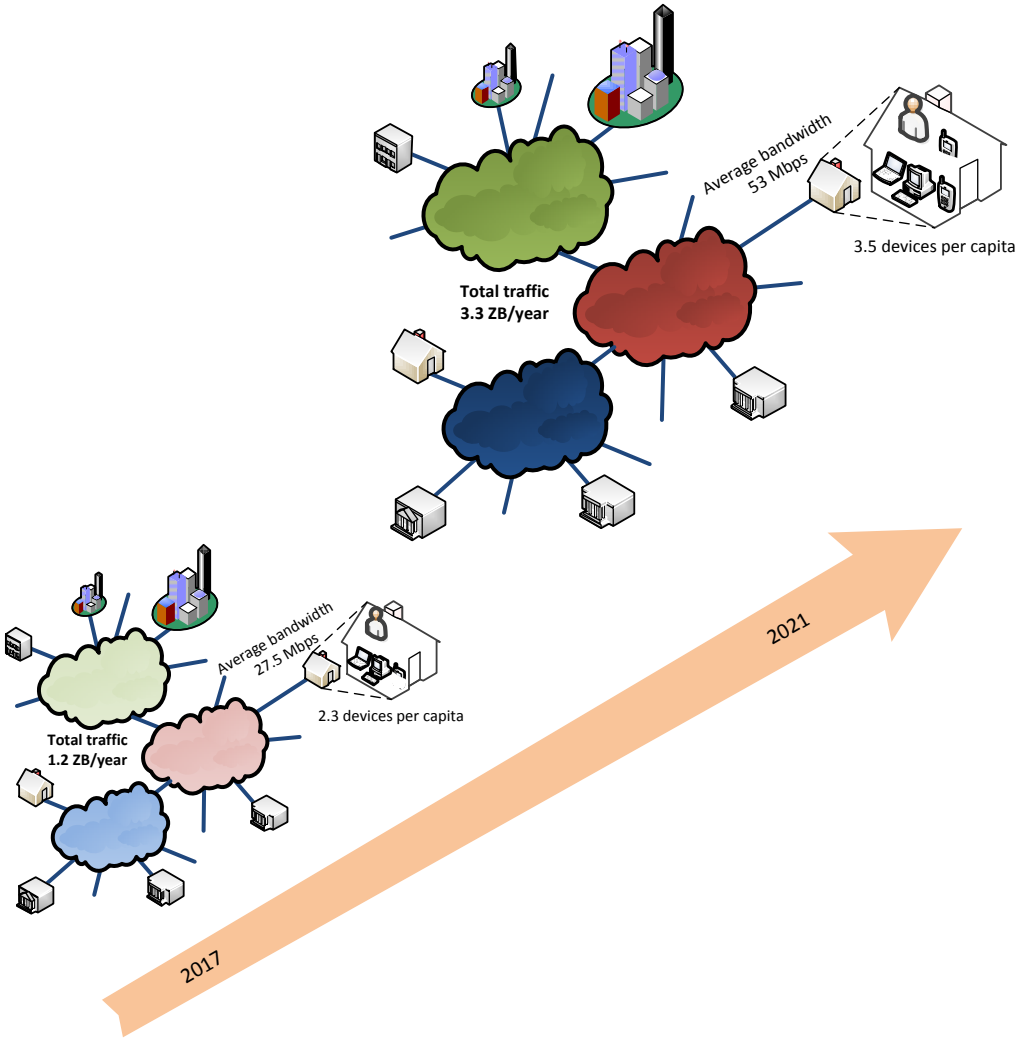


Figure 1.1: Projected growth of the internet from 2017 to 2021
Source: CISCO Visual Networking Index Report for 2017

in cloud computing were a country then their total energy consumption would be the 5^{th} highest after the US, China, Russia and Japan and about 10% higher than the energy consumption of India or Germany [3]. The same study has also identified packet routers — the networking equipment within these data centres — as the single largest consumer of energy per square foot of space.

It is this conflict between performance and energy requirements in routers that is the primary motivation for this thesis.

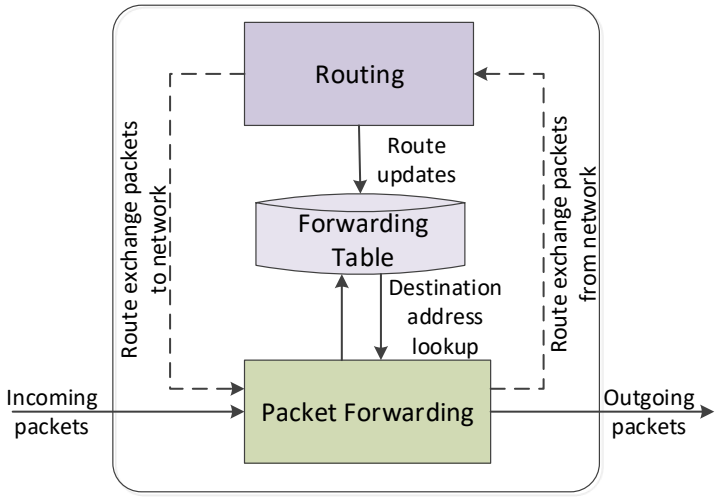


Figure 1.2: Router block diagram showing the routing and forwarding function interacting with the Forwarding Information Base (FIB)

1.2 Internet Routers

1.2.1 Overview

Packet routers are the basic building blocks of the Internet and have existed since around 1969 [4], starting off as simple Interface Message Processors (IMPs) designed to transfer messages between remote computer networks. For something that has been around for so long, the basic functionality of a router has hardly changed. It is still a deceptively simple collection of network interfaces and internal logic [5], [6] that performs two fundamental and important tasks: routing and packet forwarding (Figure 1.2) [7].

The first step in the process is the creation of a *topology map*, an image of the network topology based on route exchange packets shared with neighbouring nodes. *Routing* is the process of using this map to find the best possible path between two nodes in the network and storing this information in a database known as the forwarding table. For each packet that enters the router, the destination address is extracted from the packet header. Packet *forwarding* is the process of using the forwarding table to identify the correct egress port for the packet based on its destination address and moving the packet to the output queue. Routing is thus a control plane function, while forwarding is a data plane function of the router. The performance of the forwarding function significantly affects the overall performance of the router offering the most potential for improvement and is, therefore, the primary focus of this thesis.

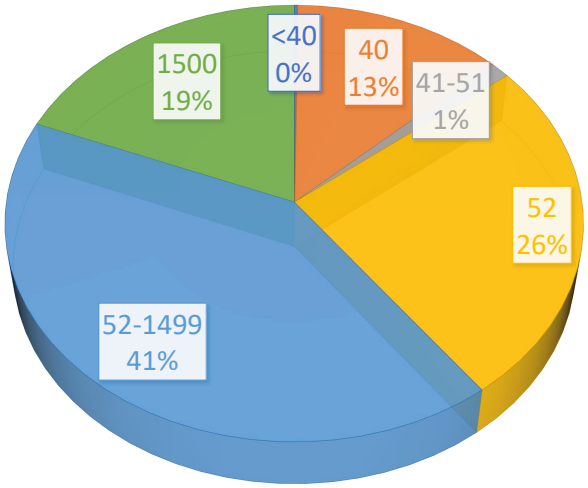


Figure 1.3: Distribution of packet sizes in anonymised internet traces captured at core routers.

As in the case of a complete router [6], the performance of the address lookup function can be measured in terms of:

1. *Throughput*– the number of destination address lookups performed per second, conventionally a complex function of clock frequency and logic complexity.

2. *Latency*– the delay from when the packet enters the address lookup block to when it exits with the correct egress port information, which depends on traffic volumes, input/output queue congestion, memory performance and the number of pipeline stages.

3. *Area*– including the area of any on-chip memory used to store next hop information and the logic that is needed to access one or more memory locations for each lookup.

4. *Energy*– consumed for each lookup. As the address lookup engine in the router is always on and has to run for every packet, any improvement to the energy consumption of the address lookup block will significantly improve the energy consumption of the router as well.

1.2.2 Performance vs Energy

Of the four router performance metrics mentioned above, throughput, latency and energy consumption of the packet forwarding module are all dependent on the size of the packet itself.

A plot of core router packet size data (Figure 1.3) obtained from anonymised Internet traces collected by Caida [8] illustrates that IP packets occur in a variety of sizes. However, *TCP ack* packets that are 40 bytes in length constitute almost 13% of the total packets handled by a router. At 100 Gbps line rates, these short, 40-byte packets will be received entirely in the order of 6.7 ns and must be forwarded as fast as they are received. A router must, therefore, be capable of looking up the destination address in its forwarding table before the next packet arrives, suggesting that upwards of 150 Million [5] or perhaps up to 200 Million [9] energy-efficient address lookups must be performed per second.

At first, when routers were little more than embedded software processors with the lookup tables residing in standard RAM and the machine programmed to manage the routing rules this job was relatively easy. The division of the architecture into *control* and *data* plane segments allowed the development of special purpose architectures (ASICs) to perform the most compute–intensive parts of the routing process. Recently, in an attempt to improve the performance of simple RAM–based routers, Ternary Content Addressable Memory (TCAM)s had been introduced to perform the IP lookup function [10]–[13]. While these can achieve high throughput, single cycle lookup operation, they are much more complex than standard SRAMs and are, therefore, much costlier than SRAMs of equivalent capacity, and they also consume significantly more power per bit.

The algorithmic lookup techniques that use either a binary tree, bit traversal trie or hashes stored in SRAMs have been used traditionally for IP lookup and have, however, been around for as long as the BSD kernel [14]. Numerous variants of the basic binary trie have been proposed and shown to perform IP lookup efficiently [15]–[26]. On the one hand, Bloom filters have been used to improve the memory access overhead in algorithmic lookup techniques based on hash tables [27], [28] or binary tries [29]–[31]. Details of the Bloom filter architecture and the existing trie-based schemed are discussed later in Chapter 2. On the other hand, Compact-trie[2] is an example of the sort of trie-based structures that have been proposed to reduce the average depth of tries and to improve the memory efficiency of the lookup algorithm [32]–[34]. These newer algorithmic approaches to IP lookup, while not as fast as TCAMs, have been shown to be energy efficient and to be able to achieve high throughput while making effective use of SRAMs and on-chip memory [28].

[2]This is referring to the specific trie architecture and is, therefore, written as 'Compact-trie' and not 'compact-trie'.

However, in spite of these improvements not surprisingly these designs, as all digital architectures face the same problem: power (and energy) consumption is a more-or-less linear function of logic switching activity. Even as far back as 1998, when clock speeds were around a few hundred MHz, clock circuits were already consuming 40-—60% of the total power in high performance integrated circuits [35]. This percentage has not improved with time [36]. It is likely that traditional synchronous digital systems will have increasing difficulty keeping up with these high transfer speeds while meeting tight energy and latency constraints. In addition, extremes in "process, voltage and temperature" (PVT) variability at small device dimensions will increasingly conspire to force excessive timing margins on the clock signal and to prevent reliable synchronous timing closure.

Asynchronous design techniques may offer a straightforward way to solve these issues, firstly by operating at a rate determined by the logic itself rather than a fixed, precomputed clock frequency that must allow for worse-case variability. Secondly, the approach might represent a way to reduce the density of switching activity, thereby lowering energy while maintaining high throughput and low latency, although this does not automatically follow. In fact, it entirely possible for the switching behaviour of an asynchronous system to approach or exceed that of an equivalent synchronous system [3], resulting in a higher power (albeit with other advantages such as low latency and un-correlated current waveforms and lower *peak* power).

1.3 Asynchronous logic

Asynchronous logic systems comprise of blocks of logic separated by registration stages that synchronise and communicate data using handshaking signals [37]. This is in contrast to Boolean systems where a global clock signal performs the task of coordinating the outputs of individual logic expressions that make up the system state [38]. In an asynchronous system, the outputs of logic expressions are coordinated locally without attempting to have a single global signal available to every logic expression in the system.

Null Convention Logic (NCL), introduced in 1996 by Fant and Brandt [39], is one such asynchronous logic system. NCL uses a library of circuit templates called *threshold gates* that (to quote [39]) *"...implement logical expressions on sufficiently expressive variables"*. Unlike Boolean variables that have a 'True' or 'False' value, NCL variables have either a Data or a Null (no

[3] Wave Semiconductor, personal communication, 2016

Data) value[4]. The threshold gates in the NCL library are essentially asymmetric variants of the fundamental Muller C-elements [40]. The basic idea behind these threshold gates is that when a certain number of inputs are in the Data state, the output transitions to the Data state and when all the inputs at the input of the gate transition to the Null state, the output transitions to the Null state. NCL is considered to be a Quasi Delay Insensitive (QDI) technique because, under certain circuit delay assumptions, the correct operation of the system is insensitive to the delay of the individual circuit elements and the system functions correctly by design, without requiring explicit timing analysis and constraints.

A typical NCL system comprises Null–Convention combinatorial blocks separated by registration stages all built using the threshold gates, thus creating a network of NCL pipelines. The registration stages in NCL allow logic functions to be split into smaller blocks to increase the throughput of the system. Just as for a single NCL gate, when a specific number of inputs (as specified by the implemented logic function) of the combinatorial block are in the Data state, the outputs will transition to the Data state. The presence of a Data value at the output is an indication to the next block in the pipeline that the inputs have been completely processed and that Data is now available downstream. This 'output complete' detection also acts as an acknowledge (*ack*) signal for the circuits that source the input variables to this combinatorial block. On receiving the *Data-ack* signal, the source circuits may clear the Data value on the inputs and transition to the Null state. Once all the inputs to the combinatorial block are at Null, the output transitions to a Null state and a *Null-ack* is generated and sent back to the source circuits. The source circuits will then send the next Data value if available, and the cycle will continue. This sequence of Null and Data values on the NCL variables appear to flow as waves through the pipeline, and that is why they are known as Null and Data *wavefronts*. The wavefronts flow downstream from input to output, and their flow is controlled by the acknowledge signals that flow upstream.

1.4 Motivation and scope

This work is motivated by the observation that traditional synchronous digital systems will have increasing difficulty keeping up with the high transfer speeds demanded of routers while still meeting strict energy and latency constraints. It is already clear that the IP address lookup

[4]In the remainder of this thesis, when referring to the state/value of an NCL variable, the first letter will be in the capital, as 'Null' and 'Data'.

function is a critical component that determines the performance of Internet routers and therefore the ultimate performance of the Internet itself. This work addresses the bottleneck imposed by the lookup process at both an algorithmic and a circuit level.

Firstly, the work has investigated techniques that could potentially offer a significant reduction in the number of memory operations per IP address operation without compromising the throughput performance. By combining some existing approaches in novel ways, it was expected that energy efficiency could be improved without affecting lookup performance. Further, given the inherent data flow behaviour of routers, asynchronous design techniques such as NCL seem like an excellent match to the problem of address lookup, offering a means of lowering energy while maintaining high throughput and low latency. However, as Jens Sparsϕ says in [41] *"...it takes more than knowledge to design efficient asynchronous circuits. "Just going asynchronous" will typically result in larger, slower and more power consuming circuits. The key is to use asynchronous techniques to exploit characteristics in the algorithm and architecture of the application in question"*. Thus, the work in this thesis explores how the existing approaches interact and also identifies tradeoffs between the characteristics of the underlying algorithm and the capabilities of the NCL design methodology.

It needs to be noted that the results presented here are based on IPv4 routing tables alone (rather than IPv6). While it is true that for studies where the objective is to evaluate lookup algorithms, an approach that works on IPv4 may not work on IPv6 and vice versa, when it comes to comparing hardware design methodologies, the two will not require significantly different implementations [27]. Given a specific lookup algorithm, IPv6 will only need more of every resource rather than mandating an entirely different approach.

1.5 Research questions and methodology

Due to the sheer number of devices that are connected today, the sizes of lookup tables within the routers have grown huge, making the lookup task even more challenging. As a result, it is not just the speed at which lookups must be performed, but also the number of entries in the forwarding table that must be searched in the lookup process that is a challenge. The research was thus set up to address the following questions:

1. **How might existing IP address lookup techniques be adapted to improve their energy consumption and/or latency?**

   The majority of existing address lookup techniques rely on the synchronous design methodology and an algorithmic SRAM-based trie structure. In all these approaches, memory accesses contribute most of the latency and energy within the lookup process. There are two existing approaches that attempt to reduce the number of memory accesses needed during address lookup. One of them is to combine a Bloom filter with a binary trie structure to filter out unnecessary memory operations and thus improve either latency or energy consumption or both. The second approach is to use a Compact-trie that has an improved mechanism to use memory efficiently thereby reducing the number of accesses required per lookup. This thesis proposes the idea of combining these two techniques and evaluates whether a Bloom filter, when used with a Compact-trie, achieves a better performance than either of them individually.

2. **Will applying an NCL-based asynchronous design paradigm further improve energy and performance compared to an equivalent synchronous implementation and will these systems have to be structured differently from the original implementations?**

   Asynchronous techniques have been around for many years and have been shown to be superior to their equivalent synchronous designs in terms of latency vs throughput trade-offs and peak energy consumption (e.g., [42]). To further improve the energy performance of the address lookup process, this thesis proposes and analyses the effect of applying the NCL design style to the Bloom filter–enhanced Compact-trie design. However, given that a naive application of NCL was seen as unlikely to offer significant improvements, the modules used in the asynchronous design are individually enhanced, and novel architectures for NCL memory and acknowledgement generation circuits are introduced. These individual modules are assembled in various combinations (with and without either the Bloom filter or the NCL memory) to take advantage of the unique characteristics of the Compact-trie structure. The performance of these enhanced designs is then compared against the unmodified NCL designs.

1.6 Outcomes and Contributions

The work reported in this dissertation has resulted in the following specific outcomes.

1. SRAM-based Compact-trie lookup algorithm with improved prefix handling capability and energy consumption.

   (a) The proposed enhanced algorithm handles a range of prefixes not handled correctly by the original algorithm and also presents a correct estimate of the resource utilisation.

   (b) A technique to employ Bloom filters with Compact-trie for Longest Prefix Matching to improve lookup performance. Experiments on a pipelined implementation of the algorithm show that for cases where the prefix information is small enough to be stored on-chip, a pipelined Bloom filter can lead to significant power savings.

   (c) A "targeted" Bloom filtering approach that has a significantly smaller area overhead with only a small reduction in the energy savings as compared to filtering at all levels in the Compact-trie.

2. Analysis of the latency versus throughput performance of a pipelined NCL implementation of Murmur, Jenkins and CRC32 hashing algorithms for Bloom filters.

   (a) NCL implementations of Jenkins, Murmur and CRC32 hash algorithms that have the same throughput as equivalent Boolean logic implementations, but a much lower latency.

   (b) Two modified NCL threshold gate architectures for use in completion detection circuits. These gates have a complementary behaviour to conventional gates, but they can be used to build completion detection circuits that generate outputs in phase with the input signals while consuming lower energy and occupying a smaller area.

3. Null Convention Logic SRAM design with Null-storage column and early completion detection.

   (a) An NCL SRAM architecture with separate read and write completion detection circuits that generate the correct handshaking signals to enable its use in NCL pipelines.

(b) The idea of a single-bit "Null" flag column to indicate whether the stored word is Data or Null. This feature helps reduce both the cycle time and energy consumption whenever address locations that store Null are accessed.

4. NCL-based design of a compact-trie with Bloom filters and Null-storage column RAMs for address lookup.

   (a) A complete compact-trie design that assembled the NCL-based hashing algorithm, the modified completion detection circuits, Bloom filters and the SRAM with Null-storage column.

   (b) Results, showing that unlike the Boolean implementation the best cycle time (throughput), area and energy consumption is achieved with an NCL implementation that does not include the Bloom filters but uses only the Null-storage RAMs.

1.6.1 Publications

A few of the outcomes mentioned before have been reported in the following publications.

1. P. Dabholkar, R. Sovani, and P. Beckett, "A low latency asynchronous Jenkins hash engine for IP lookup," Proc. - IEEE Int. Symp. Circuits Syst., vol. 2016, July, pp. 2663–2666, 2016.

2. P. Dabholkar and P. Beckett, "A High Throughput, Low Latency Null Convention Logic 16x16-bit Multiplier," in 10th International Conference on Signal Processing and Communication Systems (ICSPCS), 2016, pp. 378–385.

3. P. Dabholkar and P. Beckett, "Optimised Completion Detection Circuits for Null Convention Logic Pipelines", in 2017 IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics, 2017, pp 1-4.

1.7 Organisation

This Chapter presented the motivation and outcomes of the research and introduced some key ideas related to IP routers, the address lookup process and energy-efficient logic design. The remainder of this dissertation proceeds as follows. The following chapter (Chapter 2) presents an analysis of the existing literature on algorithms for IP address lookup in present-day routers. The pros and cons of the various processes are discussed, and the algorithm that this work

uses is identified. The theory required to understand the operation of these algorithms is also presented. This is followed by a discussion of the different types of asynchronous design techniques and current applications of the NCL design approach.

Chapter 3 presents the discussion on the Compact-trie based address lookup algorithm being evaluated. Enhancements are proposed to the actual trie generation process, and this is followed by an exploration of various Bloom filtering schemes to improve the energy consumption of the lookup process. In Chapter 4, an analysis of the latency, throughput and energy consumption of NCL-based implementations of three hashing algorithms that can be used for computing Bloom filter indices is presented. Evaluation of the performance of these algorithms leads to the development of asynchronous circuit elements that use modified NCL gates to improve the area and energy consumption. Chapter 5 proposes and analyses the NCL SRAM architecture created to improve latency and energy consumption of the prefix lookup process.

Chapter 6 draws together the work of the previous Chapters (3, 4 and 5) and describes the design of an NCL-based Compact-trie design for address lookup. The cycle time (throughput), area and energy consumption under different conditions - with and without the specially designed NCL RAM, and with and without the Bloom filter to control memory access is also presented.

Finally, Chapter 7 concludes the thesis and offers some directions for future work.

Well begun is half done

Aristotle

Chapter 2

Background and literature review

Packet processing is the single most critical function that determines the throughput and energy performance of core routers in the Internet. Most modern routers already include multiple 100 GigE (Gigabit Ethernet) ports [43]. While most of the traditional demand for bandwidth has been from the computing and entertainment industries, with the proliferation of the Internet of things and cryptocurrencies, there are now even newer data sources. This growth in demand has come as a challenge to the existing clocked Boolean systems that are finding it increasingly difficult to scale. This chapter presents an overview of the existing approaches to address lookup in Internet routers and the theory behind Bloom filters which are used to regulate memory accesses in address lookup algorithms. This is followed by a discussion on the working of various types of asynchronous systems. Null Convention Logic which is one particular type of asynchronous design methodology is studied in detail, and existing literature in NCL system design is described.

Table 2.1: Example prefix table

Prefix	Next Hop Information (NHI)
1010*	Port 1
10101*	Port 2
101111*	Port 3
01011*	Port 2
01*	Port 4
1010000*	Port 1
1*	Port 3
1111*	Port 1

2.1 Address lookup in Internet routers

A central component of the forwarding process in Internet routers is the destination address lookup. In this process, the destination address of an IP packet is extracted and matched against entries in a prefix table also known as a Forwarding Information Base (FIB). Each FIB entry consists of a network identifier that corresponds to a contiguous block of IP addresses and next hop information (NHI) which identifies the router port that the packet should be forwarded to so that it reaches its final destination. The network identifier is also known as a **prefix**, and network identifiers in a FIB may be of different lengths, and they are allowed to overlap. This means that a destination in the network is reachable through different paths or different egress ports [44]. The router then has the job of identifying the route corresponding to the prefix entry that is most specific. This process is known as Longest Prefix Match (LPM) address lookup.

Table 2.1 shows an example prefix table for a small 4-port router using dummy 10-bit IP addresses. The '*' in the prefix entry implies that the remaining bits of the prefix are a "don't care". The prefix table has 8 entries, and it can be seen that the entries at location 1 and 2 in the table have prefixes that include a range of overlapping IP addresses. Prefix 1 covers addresses 1010-000000 to 1010-111111, while Prefix 2 covers the more specific addresses 10101-00000 to 10101-11111. For a destination IP address that lies between 10101-00000 to 10101-11111, Prefix 2 would be the longest matched prefix, even though both Prefix 1 and Prefix 2 are matching and so the packet would be forwarded to Port 2 and not Port 1. A similar situation will also occur for address between 1010000-000 to 1010000-111 when both Prefix 1 and Prefix 6 match the addresses in this range, but Prefix 6 is the LPM.

IP address lookup algorithms implemented on Hardware may be classified into two broad categories [6]:

1. direct and TCAM based;

2. algorithmic and SRAM based.

2.1.1 TCAM-based IP address lookup

Ternary Content Addressable Memory (TCAM) is a structure that allows its memory to be accessed by its contents rather than by the address. A single TCAM cell can store a binary "0', "1" or a "don't care" value. To be used for a lookup operation, the TCAM cells are organised into a multi-bit wide multi-word structure similar to SRAMs. When a binary value is presented at the input of the TCAM, it is simultaneously compared against all entries in the TCAM, and it returns with a list of all entries where the data is found. During the comparison operation, a don't care value is considered as matched irrespective of the input bit. In some implementations, TCAMs may also implement some form of arbitration scheme to chose the 'best' address[1]. This will require that the TCAM entries be sorted in some order when programmed [45]. Figure 2.1 shows the TCAM organisation for the prefixes of Table 2.1. For an input IP address '1010111111' to be looked up, the TCAM matches the input with the prefix entries present in its memory and outputs an address, Addr2 in the present case because the longest match is with the second location in the TCAM. This address is used to obtain the Next Hop Information (in this case Port 2) from the NHI memory structure which is typically a smaller SRAM.

This capability of a TCAM to match multiple entries makes them attractive for storing prefixes for the LPM address lookup process. TCAMs are designed to store an entire routing table or a smaller subset of the most commonly accessed entries and allow the simultaneous comparison against the destination address of the packet being routed. Using TCAMs to perform address lookup has been a very popular approach [10]–[13] because of their ability to perform lookups in a single cycle. This latency gain, however, comes at the cost of a massively parallel architecture that exhibits significant switching activity in all its cells for each address lookup. This leads to higher power consumption than conventional SRAMs [29]. The parallel architecture and the "don't care" also mean that TCAM cells require more chip area, making them

[1]For address lookup, **best** means the entry that matches the most number of bits starting at the MSB and ignoring the don't care bits

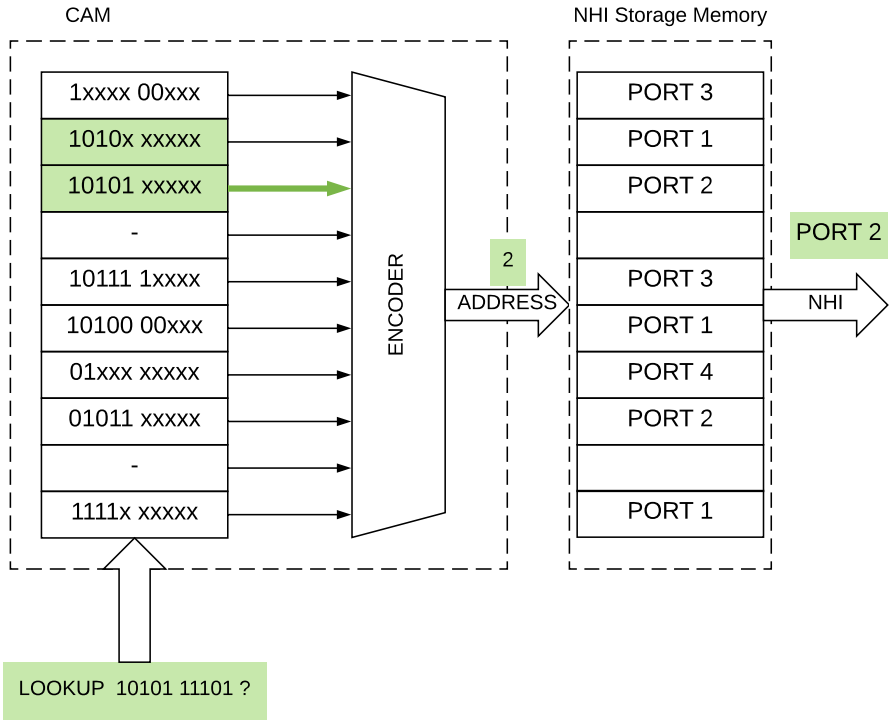


Figure 2.1: CAM-based implementation of the example prefix table of Table 2.1

costlier than SRAMs. The extra logic and capacitive loading also increase their access times [33]. Improvements to the performance of IP address lookup using TCAMs have either tried rearranging prefixes in the TCAM [46] adding extra logic to reduce the number of access to the TCAM [47] or modifying the TCAM circuit itself [48], [49]. Interestingly, there have also been attempts to achieve a TCAM-like behaviour using an SRAM-based architecture [50], [51]

2.1.2 Algorithmic SRAM-based IP address lookup

An alternative to the brute force method of TCAMs is the algorithmic approach that stores prefix information in conventional SRAMs and uses a multi-step algorithm to identify the best matching prefix and obtain correct next hop information. These algorithmic approaches may be hash-based [20], involve a binary value search in trees [16], [21], use tree bitmaps [17], [25], or a bit traversal in binary or multi-bit tries [18], [19], [22], [26] or a combination of these techniques [23], [24]. Of these various approaches, the bit traversal in trie-based structures is popular. To illustrate the working of a lookup algorithm employing bit traversal in a binary trie, consider the binary trie representation of the prefix table in Table 2.1 presented in Figure 2.2. A search for the IP address '101011111' will start at the head node and examine each bit of the IP address moving down the trie. The path through the trie is determined by the value of the bit being

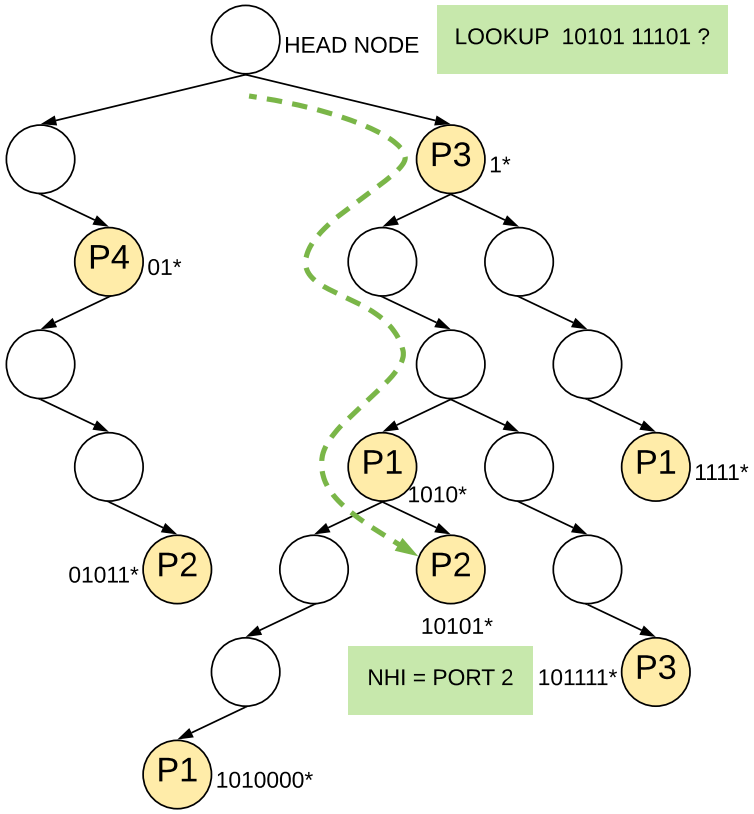


Figure 2.2: Binary trie implementation of the example prefix table of Table 2.1

examined. At the head node, the MSB is examined, and because it is '1', the right branch out of the head node is followed to its child. At the next level, the next bit in the address is '0' and therefore the left branch is taken and so on until the node with no further children is encountered (as shown by the green dashed-line). The last node with prefix information in the path is the best matching prefix (in this case Port 2). A summary of the differences in the TCAM-based and algorithmic SRAM-based approach is presented in Figure 2.3.

It may be noted that while TCAMs are definitely power hungry, the energy consumption in the SRAMs can also be significant due to the sheer amount of storage and computation that may be required. One way of reducing the energy consumption of the algorithmic hardware based LPM scheme is to employ additional circuits on-chip to minimise the number of memory accesses [52]. Dharmapurikar et al. [27] introduced the idea of storing prefixes in SRAM hash tables and using Bloom filters to test membership of a prefix before accessing the hash table in memory. The idea of combining a binary trie structure with Bloom filters to achieve a reduction in the number of memory accesses required has also been explored in [28]–[31], [53]. This scheme was extended by [23], which proposed a hash-tree bitmap scheme using a combination

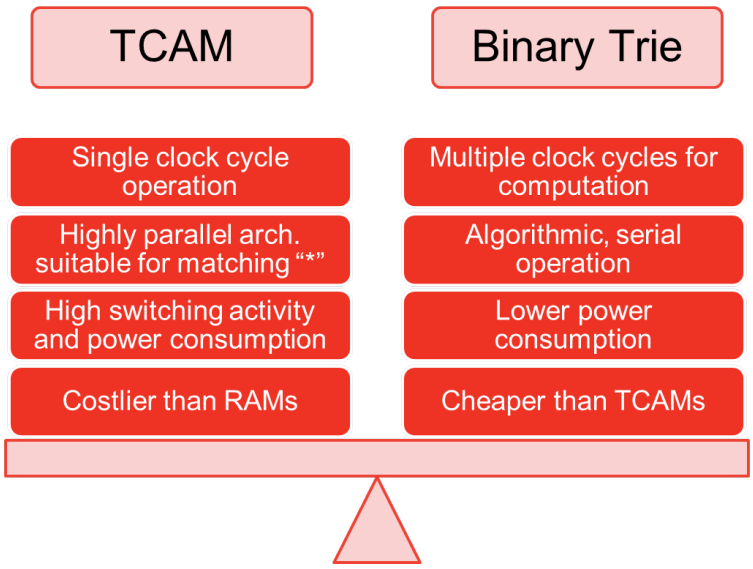


Figure 2.3: Difference in the TCAM-based and algorithmic SRAM-based approaches to destination address lookup

of hash search and standard trie based processing using a Tree Bitmap. This scheme, implemented on an FPGA, demonstrated reduced resource utilisation compared to either TCAM or Trie-based solutions. Compact-trie is one of the newer trie structures that has been proposed by Erdem et al. [32]–[34] and has been shown to have a better memory efficiency (bits/prefix) and a better throughput than some of the other popular algorithms including Tree Bitmap [17] and Flashtrie [24].

2.2 Bloom filter theory

Bloom filters are named after Burton Bloom who introduced the concept of a probabilistic data structure to verify if an element being tested is a member of a specific pre-defined set of elements. A useful property of Bloom filters is that if an element is a member of the set, then the result is always positive, while if the element is not a member, then there is a very low probability that a false-positive result may be returned, making them very suitable for use in IP routers to filter out unwanted memory accesses during the address lookup. Bloom filters also find applications in other areas of networking such as packet classification, filtering, security and routing [54].

2.2.1 Bloom filter operation

Before a Bloom filter can be used to test membership of a set, it needs to be programmed with all the members of the set. Consider a set A consisting of n members as follows:

$$A = \{a_1, a_2, a_3, \ldots a_n\}$$

Each member of A (a_i) is x-bits long and is the input 'key' for a hash function h that processes the bits in the 'key' to generate a hash value $h(a_i)$ between 0 and $m-1$. The hash value is y-bits, where $m = 2^y$ and $x > y$. The elements in A are sparsely distributed within the much larger address space (2^x addresses) represented by x bits. A Bloom filter employs an m-bit memory vector and k hash functions $h_1(\ )$, $h_2(\ )$, ... $h_k(\ )$ to produce k hash values. Each of these k values addresses one bit in the m-bit vector, and the process of programming the Bloom filter involves setting each of these addressed bits to 1 for all elements in the set A. At the end of the programming step then, bits at the following addresses would be set:

$$h_1(a_1), h_2(a_1), \ldots h_k(a_1), h_1(a_2), h_2(a_2), \ldots h_k(a_2), \ldots h_1(a_n), h_2(a_n), \ldots h_k(a_n).$$

Because hashing is essentially a mapping from the larger 2^x address space to the smaller space with 2^y values, multiple elements in A may have the same hash value generated by different hash functions (e.g., $h_1(a_2)$ may be equal to $h_6(a_{18})$). Thus a bit in the m-bit vector may be set by more than one element of the set A, and therefore it is not possible to delete an element from a Bloom filter once it is programmed. Figure 2.4 shows a 16-bit Bloom filter. A set of elements, $x1, x2, x3$, is programmed into the Bloom filter and the indices for the three prefixes are denoted by the purple, red and bloom arrows respectively. It can be seen that for $x2$ and $x3$, one of their Bloom filter indices is common (15^{th} bit) [2]. The figure also shows a situation when the Bloom filter is queried for the presence of two keys $q1$ and $q2$. The Bloom filter indices for $q1$ map to locations in the Bloom filter that have their bits set, thus resulting in a match, while the querying process for $q2$ results in a negative result.

Querying a Bloom filter for membership of an element a_{test} starts off in a manner similar to the programming process. Given a_{test}, the same k hash functions are employed to generate k hash values. The bits at the k locations in the m-bit Bloom filter are checked. If any of the bits are 0, then a_{test} is definitely **not** a member of the set A. The certainty of a non-membership when a 0 is found, is because if the element a_{test} were a member, then the k bits would definitely have

[2] This Bloom filter diagram has been inspired by a similar figure at `https://en.wikipedia.org/wiki/Bloom_filter`

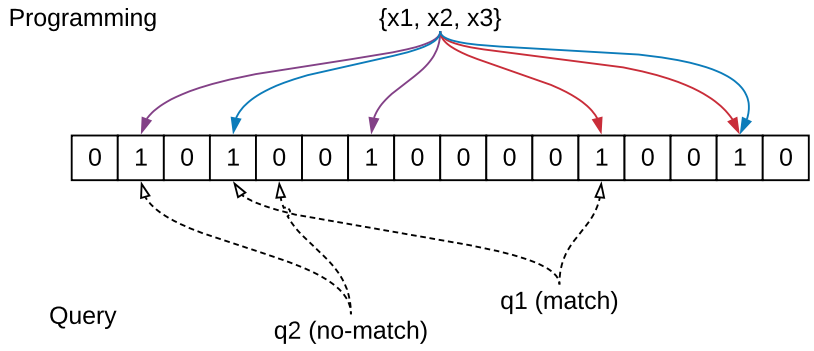
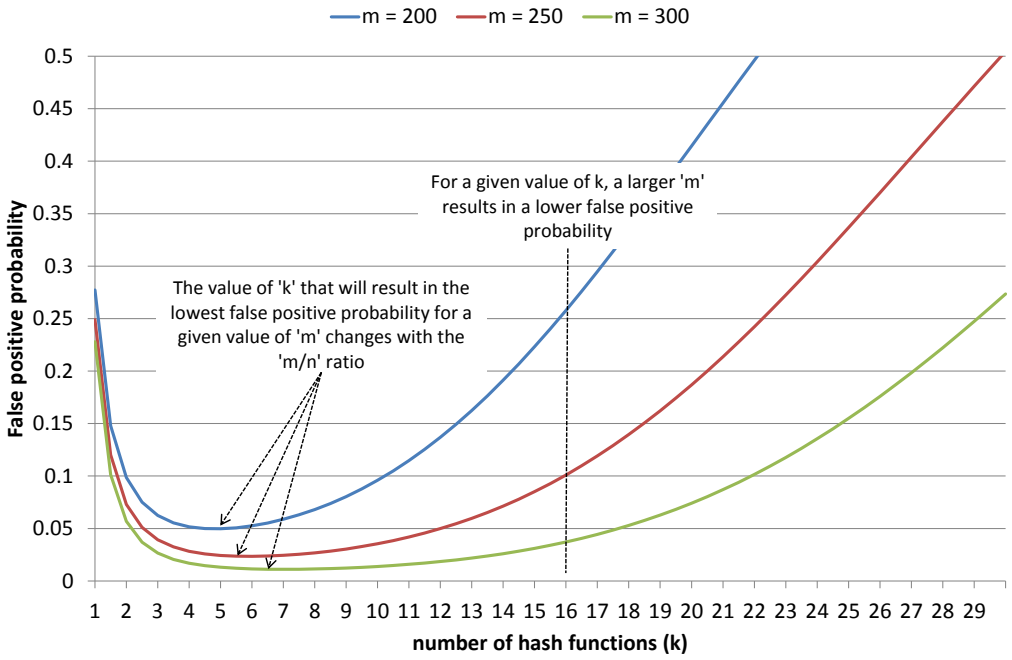


Figure 2.4: Showing bits sets in a 16-bit Bloom filter programmed with three elements x, y and z and the results of querying the Bloom filter for two test elements $q1$ and $q2$

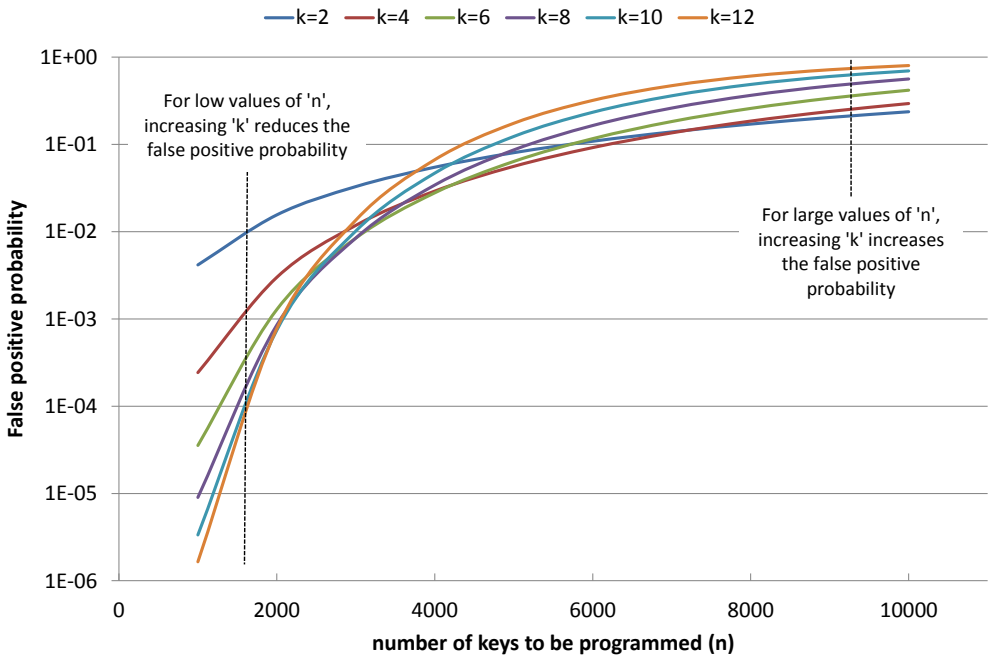
been set during programming. This is a useful property to filter memory accesses when the given IP address does not match any of the prefixes in the prefix table. However, when all k bits are 1, i.e., the test comes out positive, the element a_{test} may be a member. This ambiguity in membership for a positive test is due to the possibility that some of the k-bits may have been set by any of the other members of A. This probability of a test element being declared a member of the set A by the Bloom filter, when in fact it is not, is termed as the false-positive probability and is given by:

$$p_f = (1 - e^{-kn/m})^k \tag{2.1}$$

The false-positive probability of a Bloom filter is dependent on its size, the number of keys programmed into it and the number of hash functions. The change in the false-positive value due to a change of one or more of these characteristics of the Bloom filter is illustrated in Figure 2.5. It can be observed in Figure 2.5a that when the number of elements in a Bloom filter (n) is fixed, the lowest false-positive probability value for different sizes of Bloom filters is different and is achieved for different values of k (i.e., the number of hash functions). However, for a fixed value of k, (possibly dictated by the available computing resources in the system) the false-positive probability may be improved by increasing the number of bits in the Bloom filter. Figure 2.5b meanwhile indicates that in systems where the filter memory (m) is fixed, for low n/m values, a larger k results in a lower number of false-positives. However, as the number of elements programmed into the Bloom filter starts to increase it is better to have a smaller value of k to improve the false-positive probability.



(a) False-positive probability for different sizes of Bloom filter (m) vs number of hash functions (k) for fixed number of entries (n = 32)



(b) False-positive probability for variable number of hash functions (k) vs number of keys in (n) for fixed Bloom filter size (m = 30000)

Figure 2.5: False-positive probability characteristics of Bloom filter

2.2.2 Hash function requirements

Bloom filters used in networking applications typically emphasize worst case over average performance [55], but at the same time, they are also sensitive to latency. In such a situation it is essential to carefully choose a hash function that balances the worse case performance with the complexity of its hardware implementation and latency. As Knuth wrote in [56], *"...we need a great deal of faith in probability theory when we use hashing methods, since they are efficient only on the average, while their worst case is terrible."*. Similarly, Broder and Mitzenmacher [54] caution that *"The false positive rate (of Bloom filters) assumes that the hash functions are perfectly random. However, the question of what hash function to use in practice remains an interesting open question."*

Hashing methods available in literature can be classified as cryptographic or non-cryptographic. The primary purpose of cryptographic hashes is to obfuscate the data in such a manner that a malicious third party cannot determine the original message by just examining or 'reverse-engineering' the hashed data. Non-cryptographic hashes, on the other hand, exist to provide collision detection of non-malicious inputs and to detect accidental changes in the data. While it may be tempting to use a cryptographic hash function for Bloom filtering because of their superior obfuscation, it has been shown by [55], [57] that simple non-cryptographic hashing functions can also achieve a low false-positive probability.

Hash functions used in Bloom filters, in fact, have to satisfy the following properties to minimise the probability of false-positives:

1. Uniform distribution: the hash value should be uniformly distributed, i.e., each hash value should be equally likely given a random distribution of the input data.
2. Avalanche property: every input bit should affect every output bit about 50% of the time. Conversely, every bit of the output should depend on every bit of the input.
3. Multiple hash values: a single implementation of the algorithm should be able to generate multiple hash values for the same input data through a change in the seed.

Additionally, in a pipelined hardware implementation of the address lookup process in IP routers, the hash function may need to process a key that is just a bit longer or different from the previous key processed. A hash function that can store and reuse the value generated in a previous computation to quickly generate a new hash value reflecting the effect of the changed/added bit will obviously perform better than one that does not have this capability.

2.2.3 Hash function selection

Of all the non-cryptographic hashes available in the literature [58]–[62], the vast majority of networking applications use a limited set encompassing either the Murmur, Jenkins or CRC32 hash [27], [29], [30], [63], [64]. As a result, only these three algorithms have been considered for evaluation. The basic characteristics of these three algorithms can be described as follows:

1. *Murmur hash*

   The Murmur hash is a non-cryptographic hash function, proposed by Austin Appleby in 2008 [60], that uses a combination of recursive multiply, rotate and exclusive-or (XOR) operations to arrive at the final hash value. The Murmur hash has excellent avalanche behaviour.

2. *Jenkins Hash (Lookup3)*

   Jenkins Hash is also a non-cryptographic hash that was initially proposed in 1997 by Bob Jenkins [61]. A newer version of this hash was released in 2006. Just as in the Murmur hash, the Jenkins hash also uses a combination of addition, rotate and XOR operations but no multiplication and is much more regular in its implementation.

3. *CRC32 Hash*

   The CRC32 is the most trivial hashing algorithms of the three. It uses only bitwise XOR and rotate operations, and a large number of hash indices can be obtained regardless of input length [64]. It is one of the hashes from a family of Cyclic Redundancy Check functions that are used in communication systems for detecting accidental changes to data [65] and is used in many popular standardised digital networks such as the IEEE802.3 and the ITU-T G.706 standards [62], [66]. The hash generated using CRC32 does not have a uniform distribution and also has a poor avalanche behaviour [67], but is very simple to implement.

The false positive probability computation in a Bloom filter assumes that the hash functions are perfectly random, but finding a hash function that satisfies this criterion is surprisingly difficult [54]. A review of the literature comparing the three hashes considered here suggest that Jenkins and Murmur hash have a very low bias (12% and 2.1%) when compared to CRC32 (50%) [67]. This implies that in CRC32 with a change of input, some bits in the output have a 100% chance while others have a 0% chance of flipping. In Jenkins and Murmur hashes, on the other hand,

the probability is around 50% for all the bits. This evaluation was, however, performed for streaming data where the number of bytes that need to be hashed is quite large, and the overhead in terms of the number of cycles required to compute the hash value is negligible. For application where one is trying to compute the hash of a small 32-bit or 128-bit IP address, the overhead of a complex hashing algorithm can be significant. It should also be noted that when hashing operations are being performed in hardware, a single cycle operation may not always be possible given the pipelined nature of the hardware and also because synchronisation of various sections of the circuit can become difficult. Although a study [55] demonstrated that the Jenkins hash had a much more uniform distribution of values across the complete range of outputs, it also highlighted that the Jenkins and Murmur hash are more suited to a software implementation and do not meet the area and latency requirements of a hardware implementation where speed is of essence and limited chip area is available for the filtering functions. A detailed discussion on the hardware requirements of a hash and the function eventually used in the present work is available in Section 4.1

While there has been considerable research effort in implementing cryptographic hashes in hardware to improve execution times, non-cryptographic hashes have not received the same kind of attention. In [55], [67]–[69] hardware hash computations have been shown to be a viable alternative to software hashes. However, the ability of hardware to efficiently implement the computationally intensive hashing algorithm is a key variable in determining the performance of the particular scheme. Interestingly, while Bloom filters have generally been found to be useful, recent research [70] in Bloom filter applications to cache sharing suggest that in some cases, not using Bloom filters may actually be better.

2.3 Asynchronous logic systems

Asynchronous systems have been around for a number of years as an alternative to the synchronous design methodology to overcome the primary problem with synchronous designs, which is the global clock tree that is always ‘on’ and toggling. Although most large digital systems have local clock domains and regional clock trees, the clocks in these local clock domains are still synchronising a few hundred gates, and hence the switching activity is not trivial. A further challenge with synchronous systems is the rate at which clock speeds can actually

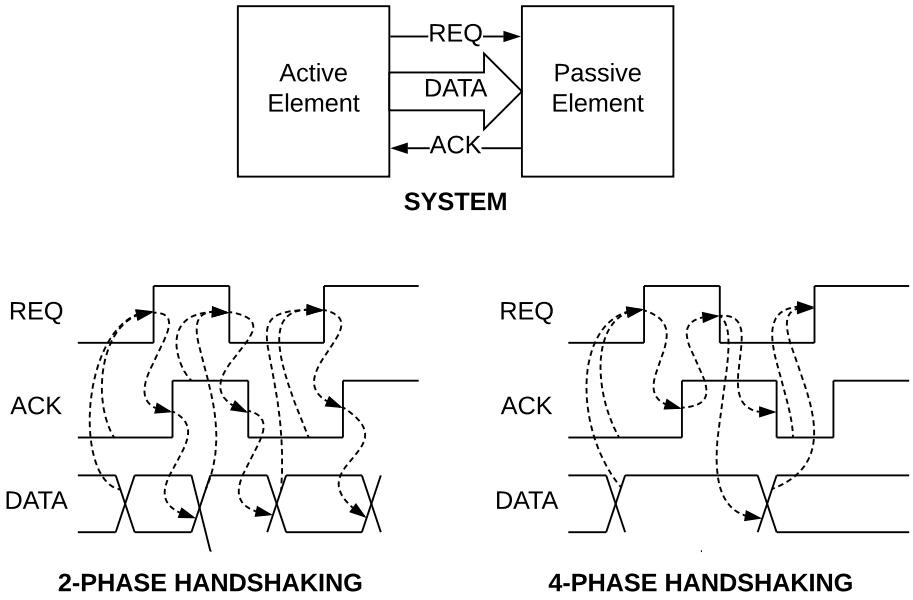


Figure 2.6: Simplified block diagram of an asynchronous system with timing diagrams showing the signal flow in 2-phase and 4-phase handshaking protocols

increase year-on-year. According to the 2013 International Technology Roadmap for Semiconductors (ITRS) report, the on-chip local clock is expected to increase to just about 6.69 GHz by 2018 and 7.96 GHz by 2020, implying its growth is constrained to no more than 8% per year [71]. Inevitably, increasing clock speeds would lead to an increase in the power consumption in these circuits. [72]. It is expected that asynchronous logic design methods could be one of the alternatives to designing systems that are fast and energy efficient at the same time. In fact, the 2015 ITRS report on system integration expects asynchronous logic design techniques to relax the timing requirements on circuits, thereby allowing voltages to be scaled down to reduce the energy requirements [73]

2.3.1 Classification of asynchronous techniques

A digital circuit is asynchronous when there exists no clock signal synchronising the sequence of events [74]. Instead, all asynchronous systems use some form of handshaking mechanism to coordinate the flow of data between elements of the circuit. The most basic and intuitive handshaking signals are *request* (req) and *acknowledge* (ack). All asynchronous protocols involve an active element sending a *req* to synchronise with a passive element, which issues an *ack* when it is ready to communicate [74]. Depending on whether this communication is sensitive to the

levels or the edges of the handshaking signals, asynchronous protocols are classified as either **4-phase** or **2-phase** signalling (Figure 2.6). In 2-phase handshaking whenever a new data value is available for transfer the *req* signal transitions. Once the data is received by the passive element, the *ack* signal transitions resulting in the initiation of a new data transfer. In 4-phase handshaking, only a high level on the *req* signal results in a data transfer. Once the data transfer is complete, the *ack* signal is asserted, which causes the *req* and then the *ack* to be subsequently deasserted. It is only when the *req* and *ack* have transitioned through 4-phases that a new data value can be transferred. In both 2-phase and 4-phase handshaking, the data and control signals are arranged into *channels*, and 'data' flows along these communication channels as a series of entities variously referred to as tokens [37], [41] or wavefronts [38], [75].

Asynchronous techniques may also be classified on the basis of how the data and handshaking signals are combined, into **bundled-data systems** or **dual-rail systems**. These two techniques are a trade-off between robustness against timing variations, power and performance. In a bundled-data system, the data is carried in Boolean encoded variables, and the handshaking signals that form the control path are also encoded in binary and bundled together with the data. Appropriate delays are introduced in the control path to delay them as much as the data path signals that implement the logic functions. An advantage of bundled-data systems is that logic is designed as conventional Boolean systems, and this results in circuits that occupy a small area. Bundled-data designs, however, require greater design and timing validation efforts to ensure that the timing constraints are met [75]. These systems assume worst case delay in the data path from one register stage to the other. This is similar to a synchronous system where the worst case delay in the clock tree determines the maximum operating frequency. A bundled-data system simply localises the problem by forcing the designer to calculate appropriate system delay values. If the systems being designed are complex, then designers would be required to compute the critical delays of all the paths in the circuit for the bundled-data technique, something that may be time-consuming and potentially error-prone. Dual-rail systems, on the other hand, encode the request signal into the data signal itself using two wires per bit of information [41]. For each Boolean bit in this systems, there are two wires. One wire is used to signal a logic '1' or *true* value and the other wire is used to signal a logic '0' or *false*. N dual-rail pairs may be used to carry an N-bit Boolean value.

Data signals in dual rail circuits also contain the encoded control signals, and therefore performing computation with these signals requires one to take extreme care in translating the

functional specifications into circuits. This requirement has led to the creation of different hardware templates as follows:

1. Delay Insensitive Minterm Analysis (DIMS) [41], [76].
2. Weak Conditioned Half Buffer (WCHB) [77].
3. Pre-charged Half Buffer (PCHB) [77].
4. Null Convention Logic (NCL) [39].

At the implementation level, correct operation of asynchronous systems is dependent on these hardware templates working correctly under appropriate assumptions of circuit delays leading asynchronous circuits to also be classified based on their delay assumptions as **self-timed**, **speed-independent (SI)** or **delay-insensitive (DI)** [41]. An exhaustive coverage of the theory behind these classes of circuits is presented in [76], [78]. Unfortunately, true delay-insensitive circuits are limited in their functionality because of their stringent requirements on wire delays as well. Instead, if it is assumed that that wire forks in the system are isochronic, i.e., the delay of the two prongs of the fork are equal then the result is what is known as quasi-delay insensitive (QDI) circuits. Typically, circuits designed using the 4-phase dual-rail approach are all quasi-delay insensitive. QDI systems are an important class of asynchronous systems because they require little timing analysis and can be made to be correct by construction [72].

Achieving delay insensitivity in 4-phase handshaking dual-rail logic systems is typically not possible using ordinary logic gates and flip-flops available in standard cell libraries used to design Boolean systems [79]. Instead, these systems use a fundamental state holding element known as a Muller C-element first introduced by [40]. The C-element and dual-rail encoding technique, both individually and together, have been used in various popular asynchronous circuit design techniques. However, as identified by Fant in [38], they do not represent a coherent, easily understandable and adaptable conceptual foundation as they still rely on Boolean logic. It is possible to devise a coherent logic system that completely and unambiguously expresses the behaviour of the system without depending on any supplemental temporal information.

2.3.2 Null Convention Logic (NCL)

Null Convention Logic (NCL) presents the essential framework for easy adoption of the 4-phase handshaking, dual-rail, quasi-delay insensitive design technique by completely expressing the behaviour of a system in terms of logical expressions and proposes a library of m-of-n threshold gates [39]. To be precise, NCL operators, also known as NCL gates, may be defined as a threshold logic function [80] with positive weights assigned to inputs, coupled with a hysteresis mechanism that guarantees QDI behaviour [74]. Defined mathematically, a threshold logic function t is an n-variable unate function with a threshold T, and weight w_i assigned to each variable x_i such that:

$$t = \begin{cases} 1 & : \sum_{i=1}^{n} w_i.x_i \geq T \\ 0 & : otherwise \end{cases}$$

Additionally, NCL defines the two states of a rail as 'Null' state and 'Data' state. 'Null' represents the no Data state, i.e. it acts as a space between two consecutive Data values on the rail. Two such rails can be viewed together as a dual-rail NCL variable that forms a codeword, representing the 'zero' and 'one' value of a conventional single-bit Boolean variable. The two rails for a Boolean variable A may, therefore, be represented as $A.0$ and $A.1$.

$\{A.1, A.0\} = \{1, 0\}$ and $\{A.1, A.0\} = \{0, 1\}$ represent the 'logic 1' and 'logic 0' values for Boolean variable A.

$\{A.1, A.0\} = \{0, 0\}$ indicates the absence of a value or a 'space' between two consecutive Data values for A.

$\{A.1, A.0\} = \{1, 1\}$ is an illegal value, and the designer has to ensure that this condition never occurs within an NCL circuit.

If the system contains an m-bit Boolean signal (A_0,A_1,...A_m), then in dual-rail NCL it is coded as 'm' dual-rail signals ($A_0.0, A_0.1, A_1.0, A_1.1, A_m.0, A_m.1$). However, if the objective of the system is communication rather than computation, and the Boolean logical variable A is m-ary, i.e., the variable can have only one of 'm' possible value, then it is possible to have an m-rail NCL variable, ($A.0$, $A.1$, ... $A.m$). For such a variable, only one of the 'm' rails can be in the Data state at any time.

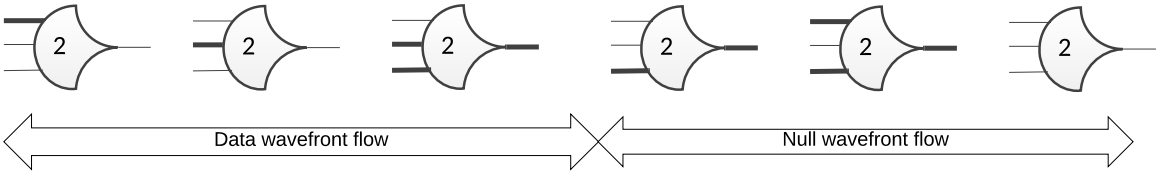


Figure 2.7: Flow of Null and Data wavefronts through an NCL 2-of-3 threshold gate (TH23)

As mentioned before, NCL also defines a library of operators. Just as with Boolean operators such as 'And' 'Or', etc., the purpose of NCL operators is to resolve data sets. Since a single NCL rail can only have a Null or Data value, the only property that a set of NCL rails can have is "How many rails have a Data value?" Null Convention Logic is, therefore, a threshold logic, and its operators are M of N threshold operators with state-holding behaviour [39]. Consider the '2 of 3' NCL operator shown in Figure 2.7. Bold lines indicate that the rail is carrying a Data value and thin lines represent a Null value.

If one assumes that the initial condition of the inputs and output of the gate is all Null, the output will not transition to the Data value, unless 2 out of 3 inputs transition to the Data value. Once the output transitions to Data, it shall not transition back to Null until all the inputs have transitioned to Null. The operator thus has a threshold behaviour, when transitioning from Null to Data and a 'hysteresis' or state-holding capability once the output is asserted [81]. This transfer of the Data values from the input to the output of a gate or NCL combinatorial function is known as Data wavefront flow, while the flow of the Null values is known as the Null wavefront flow. A transistor schematic of the TH23 gate is shown in Figure 2.8, where A, B and C are the inputs and Y is the output. The schematic illustrates the basic operation of the gate (e.g., goto data, goto Null) and how the feedback from the Y output leads to hysteresis behaviour.

Figure 2.9 shows a few of the operators from the NCL gate library with the equations under the gates representing the Null to Data transition function. Since NCL utilises 27 fundamental state-holding gates for circuit design, rather than only C-elements, it has a greater potential for optimisation than other delay-insensitive paradigms [81]. The NCL library is a covering set of four-input threshold functions that also map to all four-variable unate Boolean expressions [38]. Note here that a four-variable NCL function is not the same as a four-variable function in Boolean logic. A function of four Boolean variables would map into a function of at the most

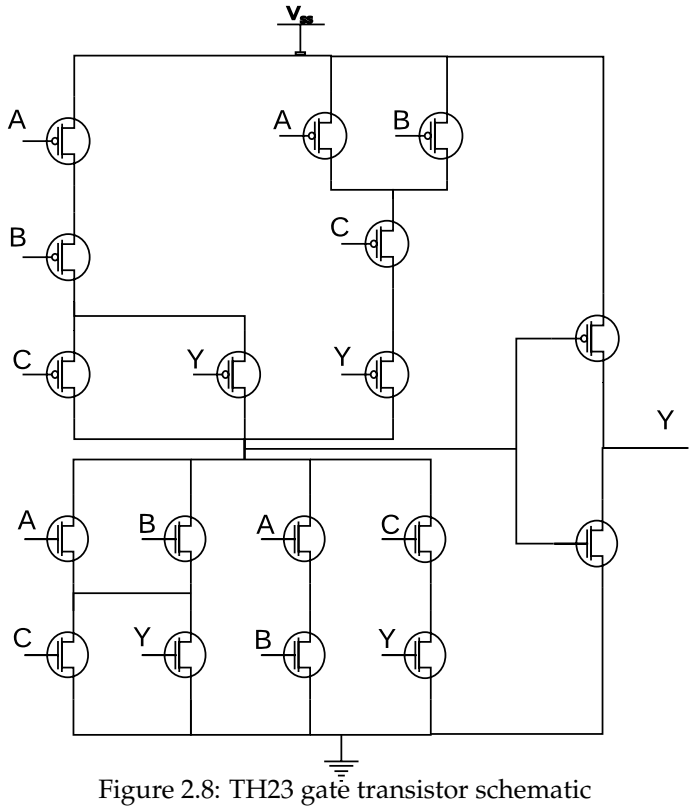


Figure 2.8: TH23 gate transistor schematic

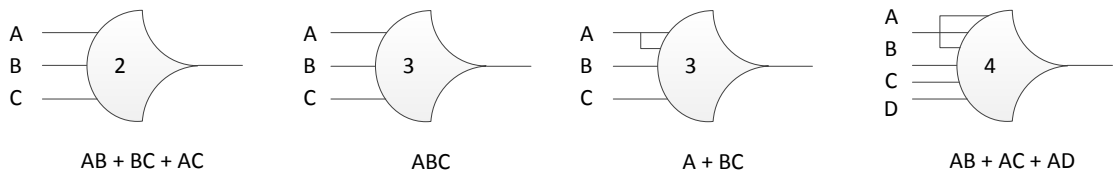


Figure 2.9: NCL gate library

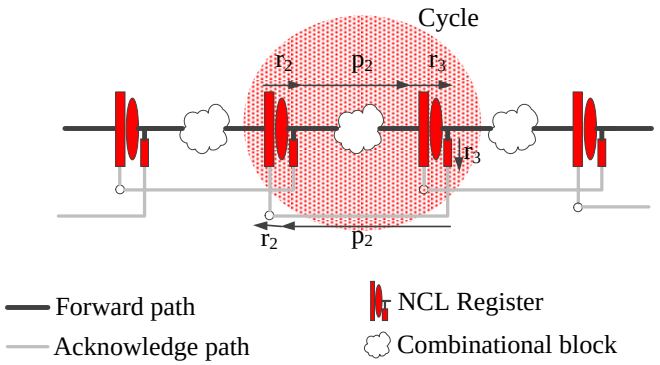


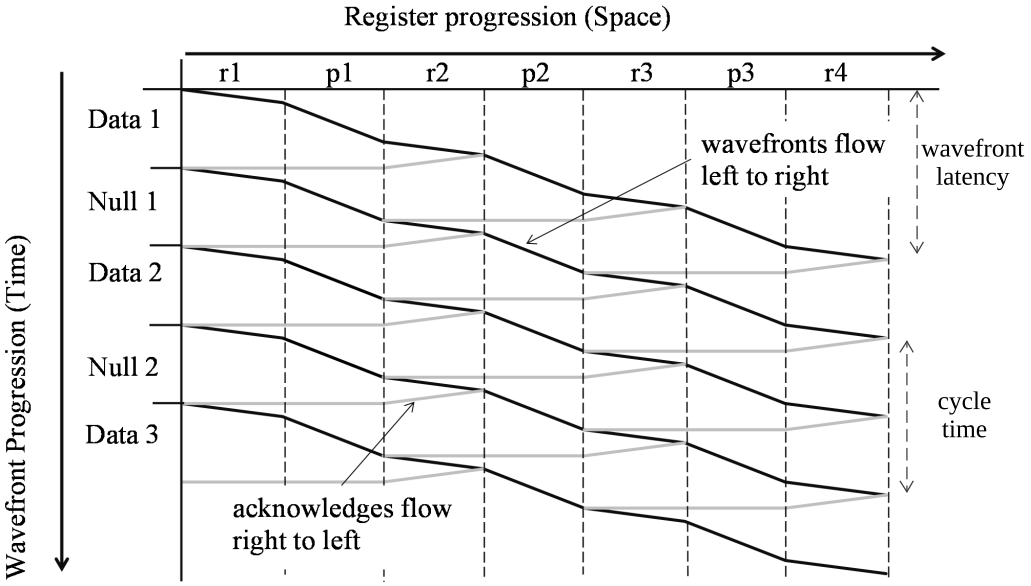
Figure 2.10: NCL pipeline model

eight NCL variables, with one set of four variables being the Boolean complements of the other set.

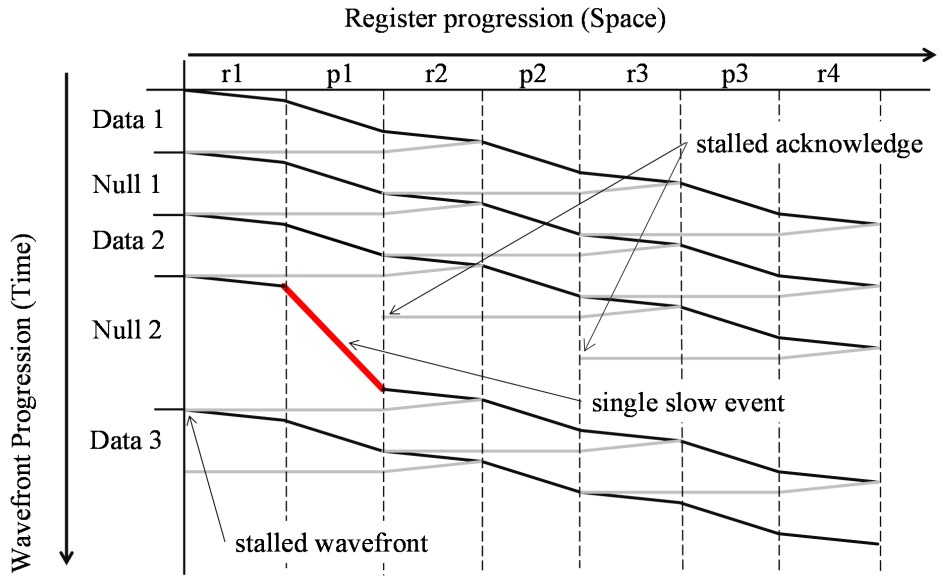
A simple pipeline of NCL variables is shown in Figure 2.10. In this figure, the bold line represents the path along which data flows from left to right through the NCL combinations stages and NCL registers. The grey lines represent the path for the flow of the acknowledge signals from right to left. In NCL terms, the part of the circuit between and including two NCL registers is called a cycle, and it is highlighted in the figure. Each cycle has a wavefront path that determines the forward latency and an acknowledge path that determines the reverse latency. A two-dimensional representation of wavefront propagation through a 4-register pipeline is illustrated in Figure 2.11. In this figure, the bold lines again represent the flow of data, while the thin grey lines represent the flow of acknowledge signals. The amount by which the data or acknowledge signals move down the 'time' axis when flowing through the individual components (R1,P1,R2,P2...) of the pipeline, indicate the delay they experience in those components.

Figure 2.11a shows the situation with a uniform delay for all wavefronts through the various stages. The wavefronts are pulled into the input of the pipeline and leave the output at a constant rate. The time from when a wavefront enters the pipeline to when it exits at the output is the latency of the system. Cycle time meanwhile is defined as the time between two successive Data or Null wavefronts. The cycle time determines the throughput of an NCL system and is analogous to the clock period in Boolean systems.

It is well known that delay performance of Boolean gates is dependent in part on the data being switched, and in the case of multi-input gates, it may also depend on the order in which the



(a) uniform delays in all stages



(b) single slow event in one stage

Figure 2.11: Wavefront propagation in 4-cycle NCL pipeline

inputs switch. Boolean gates are typically characterised for various input patterns and different load conditions, and the characteristic data are used to perform static timing analysis of bigger circuits that use these gates. The maximum frequency of operation of a Boolean circuit is, therefore, defined by the worst case timing characteristic of the worst gate in the circuit.

In NCL pipelines, however, though the delay of each gate is dependent on the input data and also on whether the gate is transitioning from Data to Null or Null to Data, the instantaneous delay does not at design time determine the final performance of the system. It is acceptable in an NCL pipeline for one of the stages to have a slightly longer delay for one particular data pattern. This longer delay will propagate through the pipeline, and once it has passed, subsequent Data wavefronts can propagate as fast as possible through the system. This concept is explained using a hypothetical scenario in Figure 2.11b, where the single slow event in one of the wavefronts, causes all forward propagating wavefronts and backward propagating acknowledge signals originating from that stage in the pipeline to be delayed. However, after the single slow event has passed, subsequent forward wavefronts and backward acknowledge signals flow through the pipeline at the fastest possible rate.

2.3.3 State of the art in NCL-based systems

The NCL paradigm and the asynchronous design approach have been used in a number of designs commercially developed by Theseus Logic Inc. [82]. A number of case-study circuits have also been implemented that demonstrate the capacity of complex circuits to be designed using the NCL approach. In [83], a Multiply and Accumulate Unit in NCL was designed and characterised. It was concluded that the NCL based design outperformed other synchronous and asynchronous approaches with respect to average cycle times even though the NCL based design incorporated additional features such as conditional rounding, scaling and saturation. It has also been demonstrated [84] that NCL logic-element based reconfigurable devices can be implemented and conventional Boolean logic designs could be mapped to these devices. A complete arithmetic logic unit implemented using both dual-rail and quad-rail NCL has been described [85]. Meanwhile, in the commercial space, Wave Semiconductors (now Wave Computing) developed their own version of NCL and built a multi-core processor that can operate at speeds in excess of 5GHz [86].

As far as the storage elements in asynchronous systems are concerned, existing asynchronous SRAM designs use either a bundled-delay approach or a Speed Independent (SI) approach. The

bundled delay approach was adopted by [87] and [88], while the speed independent approach was adopted by [89] and [90]. The bundled delay approach is susceptible to timing and voltage variations and requires careful timing analysis. The approach in [89] incorporates SI design principles on the read side, while the write side requires the design to satisfy timing assumptions at all PVT corners. The SRAMs developed in this work follow the approach suggested by [90].

There have also been multiple attempts at developing globally asynchronous locally synchronous (GALS) as well as fully asynchronous Network-on-Chip routers [91]–[94]. And although the basic building blocks for asynchronous packet routers were introduced by Nedelchev et al. [95] in 1994, the commercial 72-port 10G Ethernet switch developed by Davies et al. [96] is perhaps the only implementation of a fully asynchronous router for IP applications. All these activities show that complex systems may be built with NCL and the potential advantages in terms of throughput, latency and immunity to PVT variations are definitely realisable. Table 2.2 presents a state of the art in NCL-based designs.

2.3.4 NCL limitations

NCL was first proposed by Fant et al. in 1996[39]. As illustrated in section 2.3.3 there have been a number of attempts at designing complex systems using the NCL design paradigm. However a survey of the literature also suggests that designing with NCL is not particularly easy, primarily because of the lack of design tools. During this project attempts at using existing EDA tools to design NCL systems has been particularly challenging. There are no commercial or academic tools available to synthesize a behavioural description of an NCL design to a functionally correct netlist.

Another potential advantage of NCL is the expected saving in power consumption because of the absence of the clock tree. However it has been shown in [102] that in NCL circuits, although the peak power consumption is lower than that observed in conventional Boolean circuits, the average power consumption is comparable.

Table 2.2: State of art in NCL-based designs

Authors	Year	Title	Performance Goal/Objective
Smith et al. [83]	2002	Null convention multiply and accumulate unit with conditional rounding, scaling and saturation	Lowest cycle times in MAC unit (12.7 ns)
Smith S. [72]	2007	Design of an FPGA Logic Element fir implementing Asynchronous Null Convention Logic circuits	Design of a NCL logic element requiring the smallest area
Bandapati et al. [85]	2007	Design and characterization of Null Convention arithmetic logic units	Design of a 4-bit 8-operation ALU in both dual-rail and quad-rail versions.
Joshi et al. [97]	2007	NCL Implementation of Dual-Rail 2S Complement 8x8 Booth2 Multiplier using Static and Semi-Static Primitives	Evaluation of the area, power and speed of the design.
Duggannapally et al. [98]	2008	Design and Implementation of FPGA Configuration Logic Block Using Asynchronous Static NCL	Design of a FPGA configurable logic block (CLB) using asynchronous static NULL convention logic (NCL) Library. The proposed design uses three static LUT's for implementing NCL logic functions.
Moreira et al. [99]	2013	NCL+: Return-to-one Null Convention Logic	Modification of the NCL paradigm to support the return-to-one protocol resulting in an interesting tradeoff between power and propagation delay.
Pryzbylski et al. [100]	2016	The Bel array: An asynchronous fine-grained coprocessor for DSP	Design of a single-bit compute element based DSP in 28nm FDSOI process implementing a FIR filter
Dabholkar et al. [42]	2016	A high throughput, low latency null convention logic 16x16-bit multiplier	A fast 16x16 bit Multiplier in 28nm FDSOI achieving 609 Mops at a latency of 2.26 ns
Caberos et al. [101]	2017	Area-efficient CMOS implementation of NCL gates for XOR-AND/OR dominated circuits	Area efficient implementation of NCL circuits (14% smaller) when compared against conventional gate designs

Summary

This chapter has presented the background and prior work on address lookup techniques in modern routers and has discussed the advantages of the algorithmic SRAM-based approaches over brute-force TCAM-based approaches. The theory of Bloom filters and how they can be used to reduce unnecessary accesses to memory during the address lookup process has been explained. The focus of this thesis is to evaluate design methodologies that reduce energy consumption and area while maintaining the throughput performance. Asynchronous design techniques have been identified as a technique with the potential to reduce energy consumption. The details of Null Convention Logic - a QDI asynchronous design methodology - follow, leading to a discussion on state of the art in NCL-based systems. The next chapter will present details of a recent algorithmic lookup approach and the manner in which Bloom filters can be used to improve the energy consumption of this algorithm even in the Boolean logic system.

If I have seen further, it is by standing on ye shoulders of Giants.

Isaac Newton

Chapter 3

Compact-trie with Bloom filters in Boolean logic

Algorithmic SRAM-based approaches to address lookup have been extensively researched over a number of years. While there are a number of different memory structures used in these algorithms, the binary trie is a popular ordered tree data structure that is used to store prefix information. The next hop information for a prefix is stored at the leaf nodes in the trie, and the lookup algorithm reaches these by traversing the trie on the basis of the bits in the destination address, looking for the longest matching prefix. While binary tries are simple, they occupy large memories, are difficult to update, and the lookup process is slow. A number of compressed trie structures have been explored, and the Compact-trie structure proposed by Erdem et al. [33] has been shown to be an efficient alternative to the binary trie. This chapter presents enhancements to this original Compact-trie structure and also evaluates the performance of the address lookup function when Bloom filters are added before the Compact-trie architecture to reduce the required number of memory accesses.

3.1 Enhanced Compact-trie with epsilon nodes

3.1.1 Architecture

As outlined above, the Compact-trie is a popular structure for storing IP prefixes associated with a packet forwarding function. The key idea originally proposed by Erdem et al. in [32] and extended in [34] is the representation of a prefix p as the concatenation of three sub-strings x_p, y_p and z_p such that $p = x_p y_p z_p$. MSB_p is the value of the most-significant bit of the prefix and LSB_p is the value of the least-significant bit, determined by the length of the prefix. For example, for a prefix of length 18, the most significant bit is at position 31, while the least significant bit is at position 14. In Erdem's algorithm, the sub-string x_p is defined as a sequence of MSB_p bits followed by a single complementary bit. Similarly, z_p is a string of LSB_p bits preceded by a single complementary bit. The length of these sub-strings is $|x_p|$ (also denoted by M_p) and $|z_p|$ (also denoted by L_p) respectively, so that any prefix can then be completely represented by a 5-tuple $p = \{MSB_p, LSB_p, |x_p|, y, |z_p|\}$. MSB_p and LSB_p are 1-bit values while $|x_p|$ and $|z_p|$ each can have a maximum length of $log_2 W$, where W is the maximum length of the prefix. [34] also describes the steps to be followed for prefix insertion and address lookup in the Compact-trie. However, the mechanism for checking for the longest matching prefix is complex, and the handling of prefixes of type 111* or 111000* is incorrect. While the number of such prefixes in real IPv4 routing tables is limited, if these prefixes are not inserted in the correct location in the Compact-trie, it will result in incorrect lookups. The original algorithm [32] also underestimates the total memory utilisation of the Compact-trie structure.

A modified version of the original algorithm is proposed here that has the following enhancements:

1. Support for types of prefixes not handled correctly in the original algorithm.

2. A more accurate estimate of the total memory utilisation.

The enhanced algorithm will be referred to as Enhanced Compact-trie ($E-Ctrie$) to distinguish the proposed algorithm from the original algorithm in [34]. In this enhanced algorithm, MSB_p and LSB_p are once again the bits at the MSB and LSB position of the prefix p, represented as p = $xe_p ye_p ze_p$ i.e., a concatenation of three sub-strings xe_p, ye_p and ze_p. However, in the enhanced algorithm, the sub-strings are defined differently from the original and these new definitions

are shown in table 3.1 along with definitions of the strings xe_k and ye_k for the keys. A key 'k' to be searched in the Enhanced Compact-trie is represented as a concatenation of these two sub-strings xe_k ye_k, with MSB_k and LSB_k defined as the bits in the MSB and LSB positions of the key. In all these definitions, the '+' sign implies a non-zero repetition of a bit value, while '*' implies that the bit is repeated zero or more times (in the sense of a regular expression). When splitting a prefix p into sub-strings, bits are first allocated to xe_p followed by the ze_p sub-string and finally to ye_p. This is a key difference between Erdem's and the proposed Compact-trie algorithm.

A sample prefix table along with the prefix decomposition into sub-strings is shown in Table 3.2. It can be seen that in the Enhanced Compact Prefix Table (E-CPT), because of the new sub-string definitions, MSB_p, LSB_p and $|xe_p|$ are defined for all the prefixes. This modification to the sub-string definitions guarantees the correct lookup of some of the prefixes that may be incorrectly looked up with Erdem's algorithm. The work presented here also corrects the resource utilisation calculation presented in [34].

When compared to a binary trie algorithm, the *E-Ctrie* algorithm uses the active part (AP) of a prefix alone to determine its location in the trie[1]. As the AP is a sub-string of the complete prefix, there may be a number of prefixes in the routing table having the same active part but

Table 3.1: Definition of terms used in the Enhanced Compact Prefix Table (E-CPT) and the Enhanced Compact-trie ($E-Ctrie$) algorithm

Term	**Explanation**
xe_p = MSB_p+	a continuous string of bits of the prefix with the same value as the MSB, including the MSB.
ze_p = LSB_p*	a continuous string of bits of the prefix with the same value as the LSB, including the LSB.
ye_p = [01]*	also known as the active part of the prefix (AP_p) is a string of ones and zeros lying between xe_p and ze_p. The sub-string ye_p may be of zero length.
M_p	length of xe_p sub-string
L_p	length of ze_p sub-string
xe_k = MSB_k+	a continuous string of bits in the key with the same value as the MSB, including the MSB.
ye_k = [01]*	also known as the active part of the key (AP_k) is a string of ones and zeros. The sub-string ye_k may be of zero length
M_k	length of xe_k sub-string

[1]This is the same mechanism as Erdem's original Compact-trie algorithm

Table 3.2: Original prefix table with prefix entries decomposed into individual sub-strings according to the $E - Ctrie_{\epsilon}$ algorithm and the resultant Enhanced Compact Prefix Table ($E - CPT$).

Prefix Table		Prefix Table decomposed				Enhanced Compact Prefix Table						
NH	Prefix	NH	xe_p	ye_p	ze_p	NH	MSB_p	LSB_p	$\|xe_p\|$	ye_p	$\|ze_p\|$	P_{Type}
P1	11*	P1	11	*	-	P1	1	1	2	*	0	1
P2	111*	P2	111	*	-	P2	1	1	3	*	0	1
P3	11101*	P3	111	0*	1	P3	1	1	3	0*	1	3
P4	0011110*	P4	00	1111*	0	P4	0	0	2	1111*	1	3
P5	11110*	P5	1111	*	0	P5	1	0	4	*	1	2
P7	110101*	P7	11	010*	1	P7	1	1	2	010*	1	5
P8	1110100010*	P8	111	010001*	0	P8	1	0	3	010001*	1	5
P9	111011*	P9	111	0*	11	P9	1	1	3	0*	2	3
P10	110111*	P10	11	0*	111	P10	1	1	2	0*	3	3
P17	110001*	P17	11	000*	1	P17	1	1	2	000*	1	5
P18	111010*	P18	111	01*	0	P18	1	0	3	01*	1	4
P19	11101000*	P19	111	01*	000	P19	1	0	3	01*	3	4
P20	111010001*	P20	111	01000*	1	P20	1	1	3	01000*	1	5
P21	1001*	P21	1	00*	1	P21	1	1	1	00*	1	4
P22	0011010*	P22	00	1101*	0	P22	0	0	2	1101*	1	5
P23	1110111*	P23	111	0*	111	P23	1	1	3	0*	3	3

different MSB_p, LSB_p, $|xe_p|$ or $|ze_p|$ values. All such prefixes are stored in the same node within the trie. However, this approach affects the memory efficiency of a hardware implementation of the algorithm because memory has to be pre-allocated to each node when it is first inserted into the trie. This implies that each node has to have the same number of bits as the node with the greatest number of conflicted prefixes for a particular table. The original Compact-trie algorithm already handled this shortcoming through the use of epsilon links such that if the total number of prefixes to be stored at a node are greater than a threshold value (P_{trie}), the node is split in to sequentially connected small and fixed size nodes, each storing a maximum of P_{trie} prefixes. The connection between these nodes is called an epsilon link. The enhanced algorithm here also uses the same idea, and the corresponding trie structure is called Enhanced Compact-trie with epsilon links ($E - Ctrie_\epsilon$). This modification improves the memory efficiency at the cost of a slight increase in depth.

3.1.2 Trie creation and trie search algorithms

Consider the Enhanced Compact Prefix Table (E-CPT) from Table 3.2. To insert the prefixes from this table into an $E - Ctrie_\epsilon$ the AP_p of each prefix is first identified. For a given prefix the bits in the AP_p are used to traverse the trie starting at the head of the trie(the current node $Node_{cur}$). If $Node_{cur}$ does not have an outgoing ϵ link, the most significant bit in AP_p is examined, and the left or right branch out of $Node_{cur}$ is taken depending on whether the bit is a '0' or '1'. The node so reached is the new $Node_{cur}$, and a new AP_p is created by shifting the old AP_p one bit to the left (effectively discarding the MSB that was just examined). The MSB of the new AP_p is now examined, and a decision on the branch to be followed out of the new $Node_{cur}$ is taken. This process of examining the MSB of the AP_p, moving to a downstream node and shifting the AP_p a bit to the left is continued, until a node is reached that has no successor nodes (children) in the direction indicated by the MSB of the AP_p. A new node is then created and connected to the last $Node_{cur}$, and the trie traversal continues down to the newly created node. The process so far is similar to a conventional binary trie traversal, except that in the present case a sub-string (AP_p) of the prefix is used instead of the complete prefix (Line 3 in Algorithm 1. The special case in Compact-trie traversal occurs when a node with an outgoing epsilon link is encountered. At this node, the bit value is not examined, and the AP_p is not shifted, and only the search moves to the node at the next level. This is because, at ϵ connected nodes, there is only one link out and thus no decision needs to be taken.

Algorithm 1: Enhanced Compact-trie construction algorithm

Data: Enhanced Compact Prefix Table (E-CPT) consisting of set of prefixes P^i where P^i = $\{AP^i_p, (MSB^i_p, LSB^i_p, M^i_p, L^i_p, NHI^i_p)\}$, $0 \leqslant i < N$
Data: Root node P_{root} of Ctrie, $P_{root}.left$ = NULL , $P_{root}.right$ = NULL, $P_{root}.size$ = 0
Data: Maximum node size P_{trie}
Result: CT_ϵ trie structure

```
1  initialise i = 0;
2  foreach prefix p_i in CPT do
3      P_node = binary_trie_traversal (AP^i_p);
4      where P_node is the node reached through binary trie traversal
5      Prefix information P^i is to be stored at P_node
6      if P_node.size < P_trie then
7          P_node.info = (MSB^i_p, LSB^i_p, M^i_p, L^i_p, NHI^i_p) ;
8          P_node.size ++ ;
9      else
10         if P_node.eout == TRUE then
11             P_node = P_node.left ;
12             Go to Line 5;
13         else
14             Create a new node P_new ;
15             Copy all prefix and pointer information from P_node to P_new ;
16             P_node.left = P_new ;
17             P_node.eout = TRUE ;
18             P_node.info = (MSB^i_p, LSB^i_p, M^i_p, L^i_p, NHI^i_p) ;
19             P_node.size = 1
```

The trie traversal will finally reach a node, and the AP_p has no more bits to be examined. It is at this $Node_{cur}$ that the prefix has to be inserted (Lines 3-5 in Algorithm 1) . If the total number of prefixes stored at this node is less than the P_{trie} value, then the new prefix information is stored at the node (Lines 6-8 in Algorithm 1). However, if the node already contains P_{trie} number of prefixes then a new node is created (Line 14 in Algorithm 1), with an epsilon link from this node to the $Node_{cur}$(Lines 10-12 in Algorithm 1), and the link from the parent of the $Node_{cur}$ is now connected to the newly created node (Lines 15-17 in Algorithm 1). The prefix information is then copied into the new node and the size of the node is set (Lines 18-19 in Algorithm 1).

The algorithm used for the creation of the Enhanced Compact-trie is listed in Algorithm 1 and an illustration of the insertion process for prefix P10(110111*) is presented in Figure 3.1. The $E - CTrie_\epsilon$ as it was before insertion of prefix P10 is shown in Figure 3.1a. The AP_p of prefix P10 is examined starting at the head node, moving down the trie, until the insertion location is identified as shown in Figures 3.1a to 3.1d . Since the node at the insertion location already contains 2 prefixes and the P_{trie} value for the current example is 2, the node cannot hold any

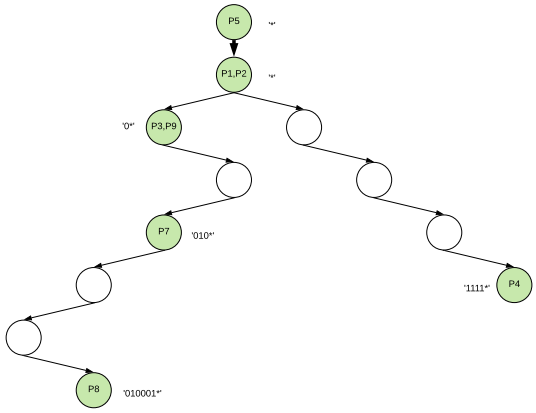
more prefixes. A new node is created, and prefix P10 is stored in this node and the links reordered so that the left child of Node(P1,P2) is now Node(P10) and there is an epsilon link from Node(P10) to Node(P3,P9), denoted by a single thick vertical arrow out of the node in Figure 3.1e. The trie traversal process during the insertion of prefix P10 is indicated by the dotted arrows. The complete $E - Ctrie_{\epsilon}$ for the E-CPT of Table 3.2 is shown in Figure 3.2.

Algorithm 2: $E - Ctrie_{\epsilon}$ search algorithm w/ Bloom filter

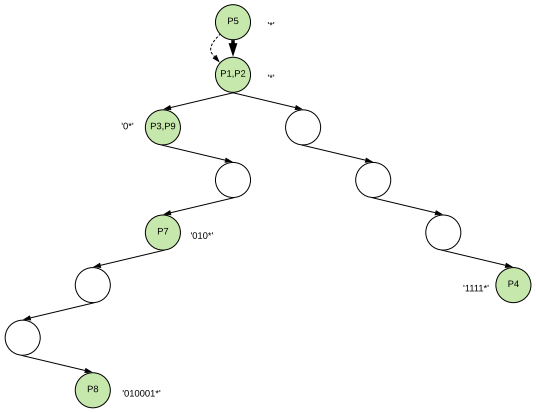
Data: P_{root} is the root node of the $E - Ctrie_{\epsilon}$
Data: Key is the destination IP address to be searched
Result: NHI_{key} is the longest matching next hop information

For head node with L_p == 0: *matchCondition* is
($(MSB_k == MSB_p)$ **and**
$(M_k \geqslant M_p)$)
For all other nodes: *matchCondition* is
($(MSB_k == MSB_p)$ **and**
$(M_k == M_p)$ **and**
$(B_0 == L_p)$ **and**
$(MAP_k \geqslant L_p \geqslant matchLength)$)

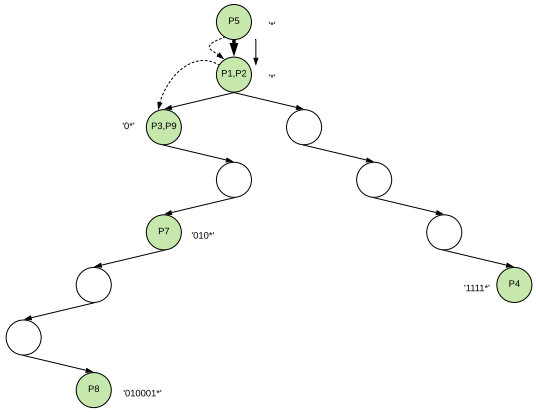
```
1  Initialise
   NHI_k = 0,
   trie_level = 0,
   P_node = P_root
   matchLength = 0 ;
2  while P_node is not NULL do
3      foreach prefix in P_node do
4          if matchCondition is TRUE then
5              NHI_k = NHI_p ;
6              matchLength = M_p + L_p + trie_level ;
7          else
8              no updates to NHI_k and matchLength ;
9      if P_node.eout == TRUE then
10         Pnext_node = P_node.left ;
11         No updates to matchLength, trie_level or AP_k;
12     else
13         if B_0 == 0 then
14             Pnext_node = P_node.left
15         else
16             Pnext_node = P_node.right
17         reset matchLength to 0 ;
18         increment trie_level Shift AP_k one bit left ;
19     P_node = Pnext_node
20 Return NHI_k
```



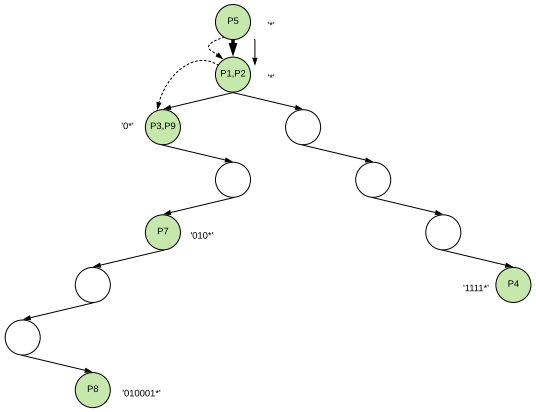
(a) $E - Ctrie_{\epsilon}$ before insertion of prefix P10



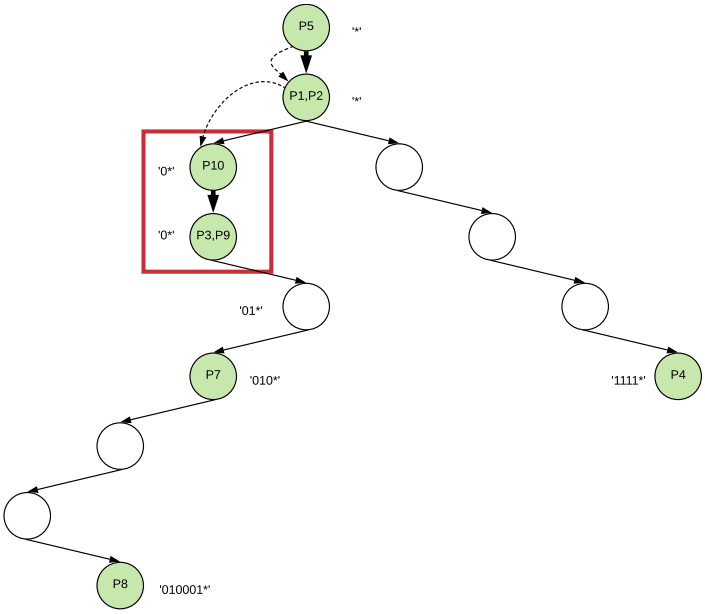
(b) Insert P10, current node has epsilon link, jump to epsilon connected node



(c) current node has regular links, examine AP=0*, follow left link down the trie, new AP=*



(d) current node has regular links, no more bits in AP. Insert prefix here, check P_{trie} value



(e) $E - Ctrie\epsilon$ after insertion of prefix P10

Figure 3.1: Process of insertion of prefix P10 in the $E - Ctrie_{\epsilon}$ structure

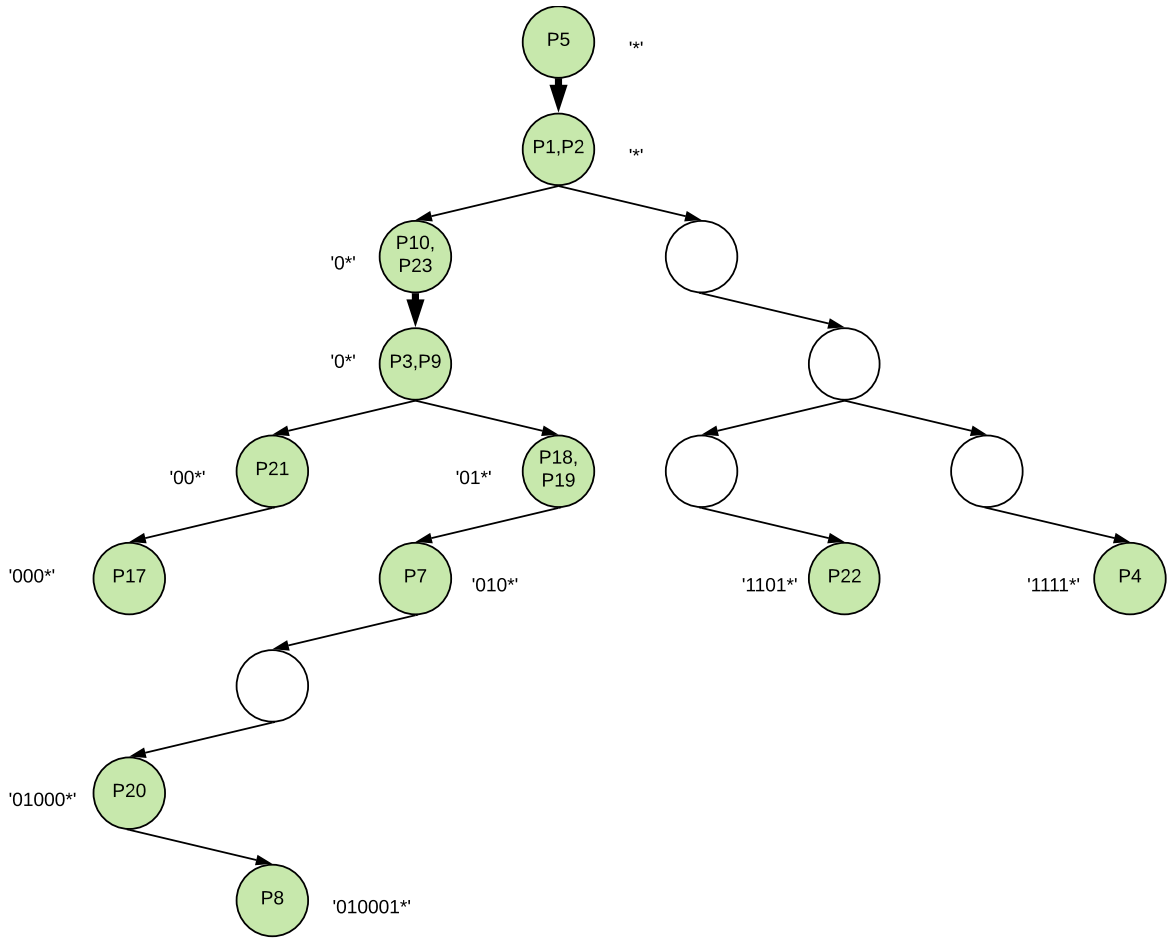


Figure 3.2: Complete Enhanced Compact-trie with epsilon links ($E-Ctrie_{\epsilon}$) after insertion of all prefixes from the Enhanced Compact Prefix Table of Table 3.2

Table 3.3: Definition of terms used in the $E - Ctrie_{\epsilon}$ algorithm

Term	Definition
B_0	Most significant bit in AP_k
MAP_k	lengthof({B_0}*), where {B_0}* is a continuous string of bits with the same value as B_0 and including B_0
$trie_{level}$	Current level in the trie search. The head node is level 0
$matchLength$	length of the matched prefix
$P_{node}.\epsilon out$	condition that the node has an epsilon link to the next stage

The search procedure for the proposed Enhanced Compact-trie algorithm is listed in Algorithm 2, while the terms and definitions used in the Algorithm have been explained in Table 3.3. As an example of the search process, consider that the IP address (key) to be searched is '110001010001'. For this key MSB_k = 1, M_k = 2 and AP_k = '0001010001'. IP lookup in an $E - Ctrie_{\epsilon}$ starts from the head node and traverses the trie using the bits in the AP_k. At each stage, a node in the trie is examined, and the stored prefixes are checked for a prefix match by evaluating the $matchCondition$ requirements specified in the Algorithm. If a match is found, the next hop information is stored, and the search moves on to the next level. The address of a child node in the next-level of the trie is determined by the 'left' or 'right' child pointer in the current node, based on the value of B_0. If the node is an epsilon node, then the present B_0 value is not checked, and the epsilon link is taken. While progressing the prefix search to the child node, the AP_k value is shifted one bit to the left if the child is connected over a regular link, but is kept unchanged for epsilon-connected nodes.

The discussion so far suggests that the process of decomposing prefixes into the three substrings helps create a trie structure that is shorter than a binary trie although the storage requirement at each node in the trie is higher. The search through this Compact-trie structure is expected to be faster than a binary trie. However, it may be noted that the search process still involves accessing the prefix information at each level in the trie. The next section shall discuss further additions to the design that limit the number of accesses.

3.2 Enhanced Compact-trie with epsilon links and Bloom filters

3.2.1 Architecture

An issue present in the Compact-trie is that, at leaf nodes with more than one stored prefix, multiple memory access cycles have to be incurred to fetch information for each prefix before a

decision on LPM can be made. This leads to an increase in the latency, a decrease in throughput and an increased power consumption per address lookup. In addition, because the Compact-trie uses only a sub-string of the prefix, there could be prefixes that are not within related IP subnets stored in the same node or in the same branch of the trie, resulting in unnecessary memory accesses during the search process. It was shown earlier in Chapter 2 that Bloom filters can be effective in reducing the number of memory accesses in binary trie address lookup implementations. It is, therefore, possible that a similar application of Bloom filters might reduce the number of memory accesses in the Compact-trie.

A regular $E - Ctrie_{\epsilon}$ structure was first created as discussed in the previous section. Prefixes were inserted at the appropriate location in the trie and also programmed into a Bloom filter. The Bloom filter indices were generated by a hash computation block that receives the following bit string as its input $din_{hash} = \{MSB_p\ M_p\ AP_p\ LSB_p\}$. To illustrate the operation consider the prefix 1001*. The din_{hash} has 1 bit for the MSB_p, 3 bits for M_p, as many bits as required for the AP_p and 1 bit for the LSB_p. For the prefix 1001*, the MSB_p is 1, the M_p is 001 (in binary), AP_p is 00 (in binary) and LSB_p is 1, concatenating all these together we get 1001001 which is the din_{hash} for the prefix. Also, for the example prefix table with 16 entries (n = 16), a single CRC8 generator is used, and it is assumed that a 64-bit Bloom filter is available (m = 64). The CRC8 polynomial used was $x^8 + x^5 + x^4 + 1$. A block diagram of the CRC8 generator is shown in Figure 3.3. The minimum number of hash indices k required to achieve the minimum false positive probability p_f for Bloom filters given the number of elements n and m bits in the Bloom filter is given by Dharmapurikar et al. [27] as:

$$k = \frac{m}{n} ln(2) \tag{3.1}$$

In the present case, the number of Bloom filter indices required is two, and these are obtained from a single CRC hash value by selecting the lower and upper 6 bits of the CRC8. The din_{hash} bit string and the hash indices generated for the prefixes in the E-CPT are shown in Table 3.4 and the final state of the 64-bit Bloom filter after all prefixes have been programmed is shown beside the CRC8 generator in Figure 3.3. For the example prefix table, the Bloom filter bits at indices 6, 10, 16, 19, 20, 23, 27, 29, 31, 33, 36, 37, 41, 42, 43, 44, 45 and 51 are set.

In Bloom filters for $E - Ctrie_{\epsilon}$, multiple prefixes with the same Active Parts (AP_p) are stored at the same node or at epsilon connected nodes. It is possible that some of these prefixes may

Table 3.4: Enhanced Compact Prefix Table with CRC8 and BF indices

Enhanced Compact Prefix Table							BF input	BF output	
Prefix	NH	MSB_p	LSB_p	$\|xe_p\|$	ye_p	$\|ze_p\|$	din_{hash}	CRC8	Bloom filter indices
11*	P1	1	1	2	*	0	10101	109	27, 45
111*	P2	1	1	3	*	0	10111	43	10, 43
11101*	P3	1	1	3	0*	1	101101	83	20, 19
0011110*	P4	0	0	2	1111*	1	001011110	134	33, 06
11110*	P5	1	0	4	*	1	11000	95	23, 31
110101*	P7	1	1	2	010*	1	10100101	144	36, 16
1110100010*	P8	1	0	3	010001*	1	10110100010	151	37, 23
111011*	P9	1	1	3	0*	2	101101	83	20, 19
110111*	P10	1	1	2	0*	3	101001	170	42, 42
110001*	P17	1	1	2	000*	1	10100001	179	44, 51
111010*	P18	1	0	3	01*	1	1011010	165	41, 37
11101000*	P19	1	0	3	01*	3	1011010	165	41, 37
111010001*	P20	1	1	3	01000*	1	10110100001	93	23, 29
1001*	P21	1	1	1	00*	1	1001001	173	43, 45
0011010*	P22	0	0	2	1101*	1	001011010	165	41, 37
1110111*	P23	1	1	3	0*	3	101101	83	20, 19

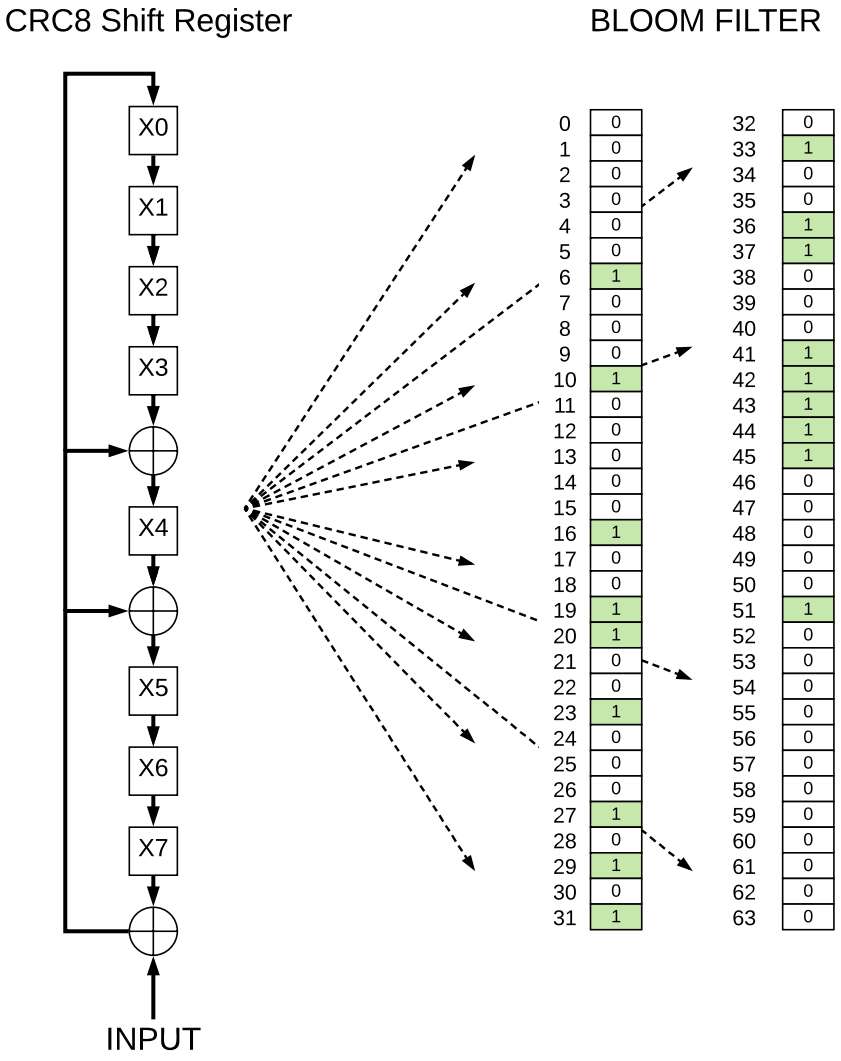


Figure 3.3: CRC8 generator block diagram and programmed Bloom filter

have the same M_p values as can be seen with prefixes P3, P9 and P23 and also with prefixes P18 and P19. Table 3.4 indicates that for all such prefixes, the Bloom filter indices are the same and these contribute only one entry to the Bloom filter. This is unlike a binary trie, where no two prefixes are stored in the same node, and therefore their Bloom filter indices would be different. Thus in case of binary tries, different bits would be set in the Bloom filter, causing it to fill up much faster than in the case of the Compact-trie. It can be seen from (3.1) that if the number of elements that have to be stored in the Bloom filter increase for the same size of Bloom filter, the false positive probability degrades. Thus, since the number of distinct elements that have to be stored in Bloom filters for Compact-trie is less than those in Bloom filters for binary trie, it

may be expected that the number of false positive indications from the Bloom filter would also reduce.

3.2.2 Operation

The trie creation algorithm for an Enhanced Compact-trie with epsilon links in the presence of Bloom filters is essentially the same as that in Algorithm 1. The only difference now is that in addition to inserting the prefix in the Compact-trie, the prefix tuple information is also fed to a hash computation block that generates the Bloom filter indices and these are used to program the corresponding bits in the Bloom filter.

Algorithm 3: $E-Ctrie_{\epsilon}$ search algorithm w/ Bloom filter

Data: P_{root} is the root node of the $E-Ctrie_{\epsilon}$
Data: Key is the destination IP address to be searched
Data: BF is the Bloom filter for $E-Ctrie_{\epsilon}$
Result: NHI_{key} is the longest matching next hop information

For $E-Ctrie_{\epsilon}$ **with Bloom filters:** All *matchConditions* are the same as those without Bloom filters

Perform Initialisation

```
while P_node is not NULL do
    din_hash = {MSB_k, M_k, most significant (trie_level + 1) bits of AP_k} ;
    Compute Bloom filter indices h_k1, h_k2, ... h_kn;
    Check if Bloom filter matches ;
    if Bloom filter result is TRUE then
        ACCESS PREFIX INFORMATION;
        foreach prefix in P_node do
            Evaluate matchCondition and update NHI and matchlength if match found;
    else
        DO NOT ACCESS PREFIX INFORMATION
    if P_node.eout == TRUE then
        Pnext_node = P_node.left ;
        No updates to matchLength, trie_level or AP_k;
    else
        if B_0 == 0 then
            Pnext_node = P_node.left
        else
            Pnext_node = P_node.right
        reset matchLength to 0 ;
        increment trie_level Shift AP_k one bit left ;
    P_node = Pnext_node
Return NHI_k
```

Table 3.5: Illustrating the Bloom filter computation during the search process of the key '110001010001' (MSB_k = 1, M_k = 2 and AP_k = '0001010001') in the $E - Ctrie_\epsilon$ structure of Figure 3.2

$trie_{level}$	din_{hash}	CRC8	Bloom filter indices	BF Match
0	1 010 0	225	56,63	NO
1	1 010 0	225	56,63	NO
2	1 010 00	252	63,60	NO
3	1 010 00	252	63,60	NO
4	1 010 000	126	31,62	NO
5	1 010 0001	179	44,51	YES

The prefix search algorithm for an $E - Ctrie_\epsilon$ with Bloom filter is also very similar to the one without the filter and is listed in Algorithm 3. The lookup process starts from the root node and traverses the trie using the bits in the AP_k and the 'left', 'right' and 'epsilon' link information present at each node. However, at each level in the trie, Bloom filter indices are also computed, and the Bloom filter is accessed. Prefix information is examined only at those nodes in the trie that return a positive result from the Bloom filter query. As discussed in Section 3.2.1 the input string to the hash computation block during prefix insertion is given by $din_{hash} = \{MSB_p\ M_p\ AP_p\ LSB_p\}$ and the complete AP_p is used in the hash computation. However, in the process of prefix search, only the sub-string of AP_k that is one more than the length of the AP_p for prefixes stored at that level is used in the hash computation. Thus $din_{hash} = \{MSB_k\ M_k\ AP_k[trie_{level} + 1bits]\}$.

As an example, consider that the IP address (key) '110001010001' is to be searched for the longest matching prefix in the prefix table 3.2. For this key MSB_k = 1, M_k = 2 and AP_k = '0001010001'. The Bloom filter computation and the prefix search process through the trie is illustrated in Table 3.5. For this illustration, please also refer to the $E - Ctrie_\epsilon$ of Figure 3.2 and to the programmed Bloom filter in Figure 3.3

In this case, the Bloom filter produces a negative result for the first five levels. Because a Bloom filter may produce a false positive but never a false negative, the search process continues to move down the trie without accessing any of the prefixes in the first five levels. It is only at the sixth level in the trie that the Bloom filter indicates a positive and the prefix information is accessed. The match condition is then evaluated, and the Next Hop Information is updated when a match is found.

3.3 Hardware design considerations

It is necessary to evaluate the memory requirements of the $E-Ctrie_{\epsilon}$ and the Bloom filter when implemented as a fully pipelined design on an FPGA to be able to comment on the effectiveness of the proposed technique. This section discusses the hardware design considerations such as the structure of the nodes in the trie, the trade-off between a monolithic and a pipelined Bloom filter and the complete system architecture that might affect throughput, energy savings and memory utilisations of the final implementation.

3.3.1 Trie node structure

In a pipelined hardware implementation, memory size is fixed, and there is no equivalent of the software mechanism of dynamically allocating memory to each node from a large heap when new prefixes are added to the trie. Instead, fixed memory blocks have to be pre-allocated for each pipeline stage during the design process. As a result, the size of the node and in turn the total memory requirement for the $E - Ctrie_{\epsilon}$ is a function of the number of bits in the pointers and the prefix specific information that needs to be stored at each node. Figure 3.4 shows the structure of a typical node in the $E - Ctrie_{\epsilon}$ structure.

The prefix-specific information (shown in blue in Figure 3.4) can be stored in a separate memory, either on or off-chip. The value in brackets represents the number of bits needed to represent a particular field. Thus the MSB and LSB require one bit each, the M and L fields require 'm' and 'l' bits respectively, while the child pointers need 'lc' and 'rc' bits. The actual number of bits needed for these fields is determined by the total number of nodes in the routing table, the distribution of these prefixes and the depth at which a particular node is located in the trie. If two routing tables exist having almost equal numbers of prefixes but with different distributions, such that one of the tables results in a deeper trie with a higher number of intermediate non-prefix nodes, then that routing table would require more memory resources on the FPGA. This is not just because of the greater total number of nodes in the trie, but also because more nodes imply a larger address space that, in turn, requires more bits for the pointers in each node. It is, however, not possible to tweak the number of bits in the node structure for every routing table and therefore fixed values are chosen.

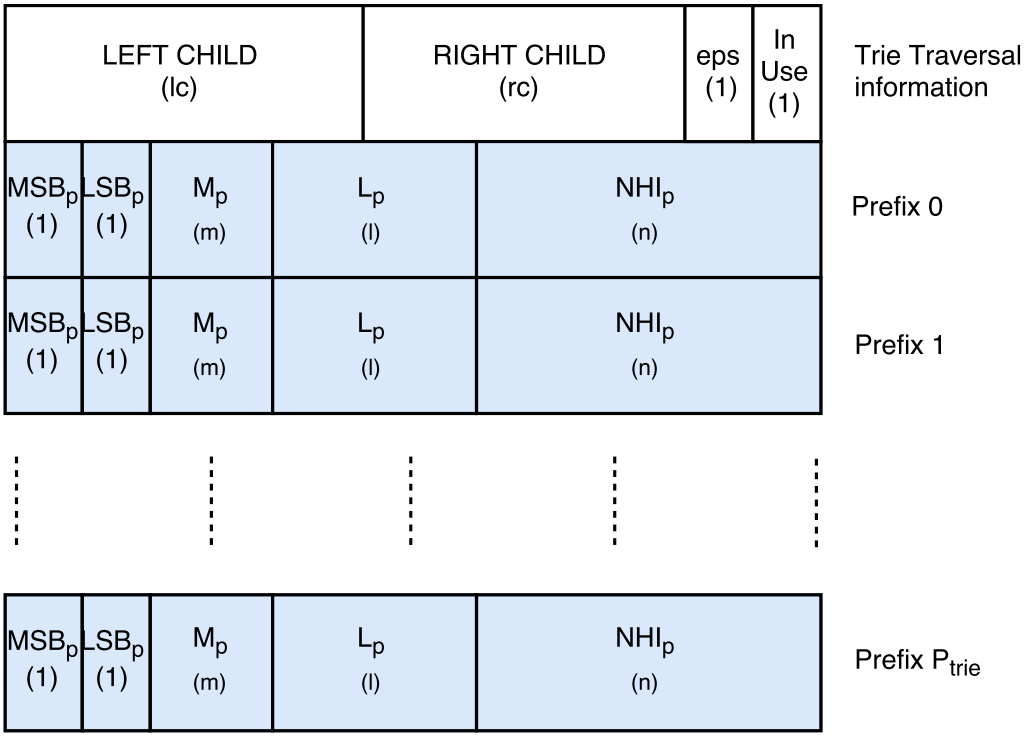


Figure 3.4: Node structure in an $E - Ctrie_\epsilon$

3.3.2 Pipelined Bloom Filter

A CRC generator is used as the hash function for generating Bloom filter indices to evaluate the performance of the $E-Ctrie_\epsilon$ algorithm with Bloom filtering using real routing tables. Because the input to the hash function consists only of MSB_p, M_p AP_p and LSB_p i.e., a reduced and transformed set of the information uniquely identifying a prefix, there exists the possibility of a collision in the generated hash values. If all prefixes are programmed in a single Bloom filter (also referred to as monolithic Bloom filter), a key being searched at a certain level in the trie will generate Bloom filter indices that match bits set by prefixes at other levels, resulting in a false positive and therefore unnecessary prefix memory accesses.

It could be expected that setting up a separate Bloom filter for each level in the trie would exhibit better performance, albeit with potentially greater area overheads. This approach lends itself quite well to a hardware implementation as the Bloom filter, and the address lookup function can now be pipelined. A separate Bloom filter per pipeline stage means each Bloom filter can now be much smaller than the single large Bloom filter that would otherwise have been

needed. This, in turn, results in fewer bits needed to access the Bloom filter memory, thereby simplifying the design of the address decoder and would also result in a much distribution of the index values since all indices are generated by selecting sub-strings from a single CRC value. Setting up a single Bloom filter that can be accessed by all stages in a pipelined implementation additionally requires a multi-port memory, which is difficult to achieve in an FPGA. Thus having a pipelined Bloom filter is considered appropriate.

An additional design consideration for the use of a pipelined Bloom filter is the ease of CRC computation at each stage of the pipeline. It can be seen from the CRC-8 diagram in Figure 3.3 that the value of the CRC shift register after an input bit is received is dependent on the initial value loaded into the shift register and the value of the input bit. The hash computation required for Bloom filters uses a key that has MSB_k, M_k as its first two fields, followed by the bits in the AP_k. At each level in the trie, the key fed to the CRC block is just one bit longer than the key fed to the CRC block at the previous level. Thus, at each stage in the design, the CRC value computed in the previous pipeline stage can be used as the initial value for the CRC shift register and feeding just the one extra bit from the AP_p or AP_k allows the new CRC value to be generated in a single clock cycle. For the first pipeline stage, the CRC of the string $MSB_k\ M_k$ is computed separately in a single clock cycle before the actual prefix search enters the first stage.

3.3.3 System architecture

Figure 3.5 shows the block diagram of an FPGA implementation of the $E - Ctrie_\epsilon$ that includes a Bloom filter. This is very similar to the CT_ϵ architecture proposed by [34]. For the $E - Ctrie_\epsilon$ hardware implementation, the number of stages in the lookup pipeline is dependent on the maximum number of bits in the active part of the prefixes for a routing table and also on the P_{trie} value. A single 'match_stage' in the $E-Ctrie_\epsilon$ pipeline comprises three parts: a 'match_module' that implements the prefix match and trie traversal algorithm, 'node_storage' that stores the prefix and trie traversal information at that level and a 'Bloom filter' that controls accesses to the prefix information RAM. The 'node_storage' module (Figure 3.5), which includes both the prefix information RAM and the trie traversal information RAM, is implemented on FPGAs in block RAMs (BRAM) or LUT RAMs depending on how the synthesis tool responds to the required size. The prefix information RAM may also be implemented in an external off-chip SRAM when a large number of bytes need to be stored per prefix. Bloom filter

bits are always stored on-chip to ensure fast operation. The present research follows Song et al. [28] and a number of others where complete tries and prefix storage were implemented inside FPGA memory. In contrast, the work by Lim et al. [30] and Dharmapurikar et al. [27], both of whom use Hash Tables to store prefix information, expect these to be held in external memory. The primary reason for using Bloom filters in these cases then is the underlying assumption that the cost of an external memory access is high and must be avoided, while the cost of a Bloom filter access in terms of power consumption and latency is insignificant. All these address lookup schemes thus use multiple Bloom filter computations to save memory accesses and improve latency, throughput and power consumption.

3.4 Results and discussions

The results presented in this section seek to understand the point at which the time and power consumption associated with hash computations and Bloom filter accesses are no longer trivial when compared with external SRAM accesses. The performance of the proposed algorithm was evaluated first through software simulations in Python and then through Modelsim® simulations of a Verilog design targeted for Xilinx® Virtex-7 FPGA. The software simulations demonstrate the effect of the Bloom filter on average and worst case prefix memory accesses and also reveal the impact that changes in Bloom filter size and P_{trie} value have on the lookup performance. The Modelsim simulations for the target FPGA were used to evaluate the resource utilisation and power consumption of representative pipelined $E - Ctrie_{\epsilon}$ implementations.

3.4.1 Experimental setup

Six different IPv4 routing tables downloaded from Packet Clearing House [103] on 01-April-2017 were used for testing. Two of these tables have more than 300k prefixes (big tables), two have between 40k and 100k prefixes (medium table), and two have less than 10k prefixes (small tables). These different sizes are chosen to verify that the algorithm performs correctly over a range of different table sizes. Table 3.6 shows the number of prefixes in each of the routing tables. Synthetic packet traces were generated that contained roughly five times the total number of prefixes in the routing table, with destination IP addresses distributed uniformly over the range of addresses covered by the prefixes.

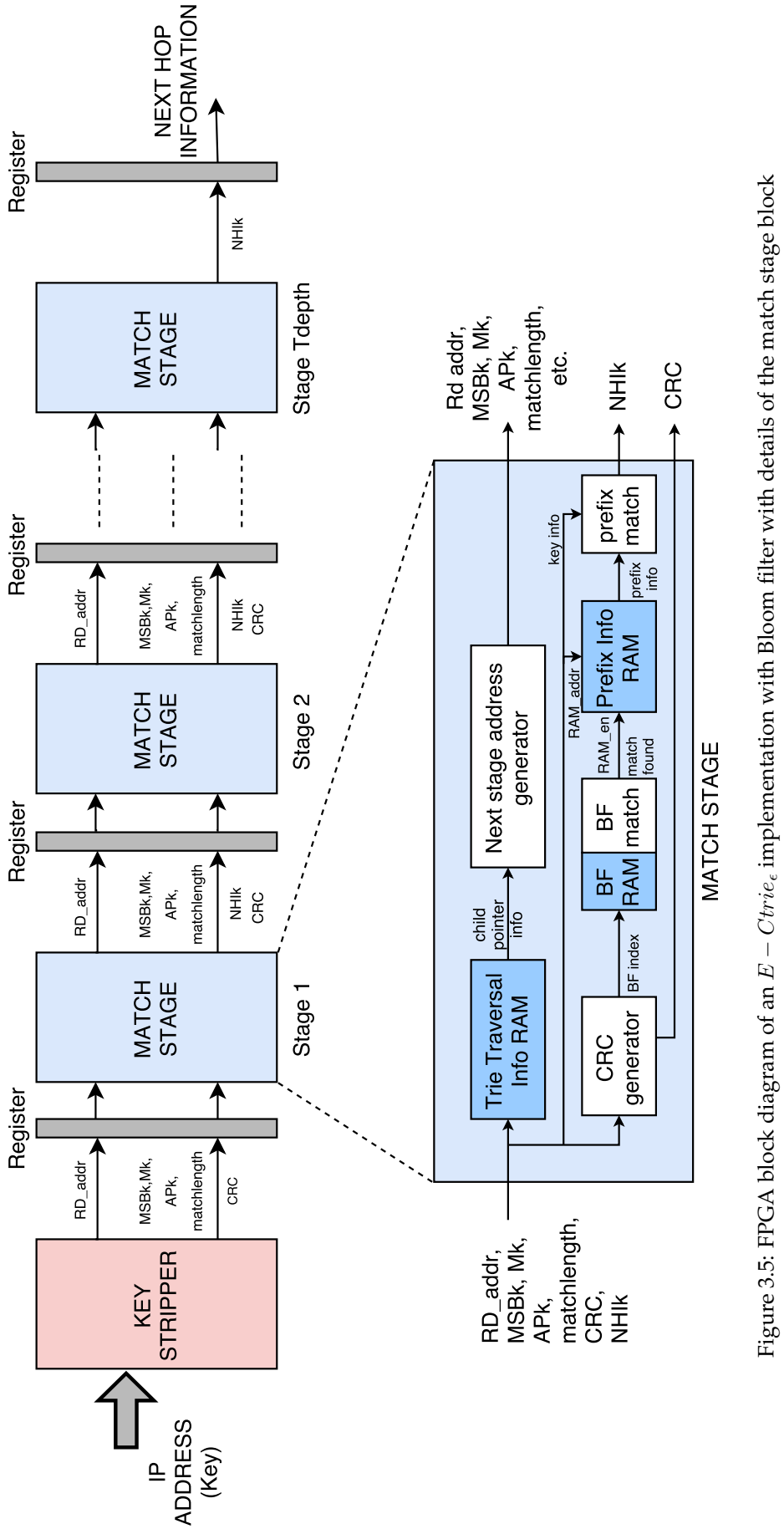


Figure 3.5: FPGA block diagram of an $E - Ctrie_{\epsilon}$ implementation with Bloom filter with details of the match stage block

Table 3.6: Sizes of real IPv4 routing tables [103]

Exchange Name	Number of prefixes
CAI	3180
MAN	6304
LYS	44426
MGM	96232
IAD	388773
PAO	629539

In the Bloom filter, the hash function generator uses the IEEE 802.3 CRC32 polynomial [104].

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

The number of Bloom filter indices used has been set to 2, and the indices are obtained by selecting $log_2 n$ bits from different locations in the generated CRC, where n is the size of the Bloom filter. Lookup performance was evaluated for two P_{trie} values (2 and 3), five Bloom filter sizes and a (M_p, L_p) = (3, 5). The Bloom filters are designed M_{bf} times the number of prefixes in a routing table N rounded off to the nearest higher power of '2'.

$$BF_{size} = M_{bf} * 2^{\lceil log_2 N \rceil}$$

Thus for a LYS routing table with $N = 44426$ prefixes, a Bloom filter with $M_{bf} = 2$ will require 131072 locations if a single Bloom filter is used. On the other hand, for a pipelined implementation the Bloom filter sizes will depend on the number of prefixes at each level. While the size cannot be selected differently for every table that the router needs to handle, based on the understanding that the number of prefix nodes at shallower levels are much less than the number of nodes deeper in the trie, the Bloom filter sizes in pipelined Bloom filters may be adjusted. For the same LYS table, there are 10276 prefixes at the 22^{nd} pipelining stage. The Bloom filter for this stage will, therefore, have 32768 elements for a M_{bf} value of 2.

3.4.2 Software simulations

Software simulations were performed with appropriate data structures created for the $E - Ctrie_\epsilon$ and Bloom filters in Python 3.6. The simulations were used to evaluate the performance of the insertion and search algorithms and were carried out both for the monolithic and

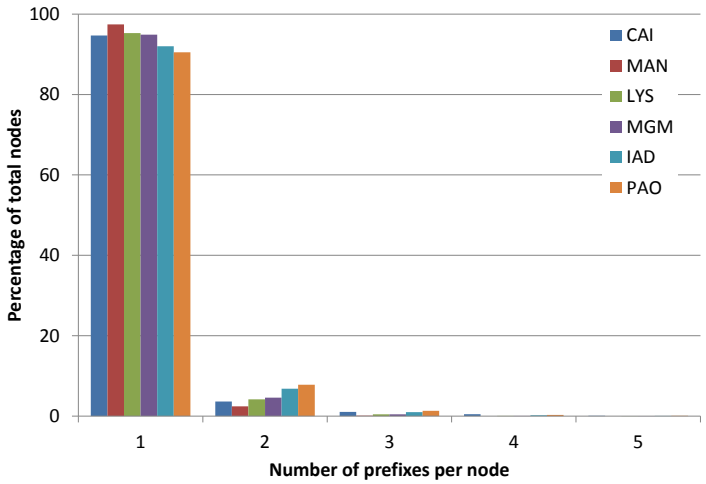
pipelined Bloom filter. The saving in the number of memory accesses and the overhead in terms of total memory requirements are reported.

3.4.2.1 Effect of P_{trie} on memory utilisation of $E - Ctrie_{\epsilon}$

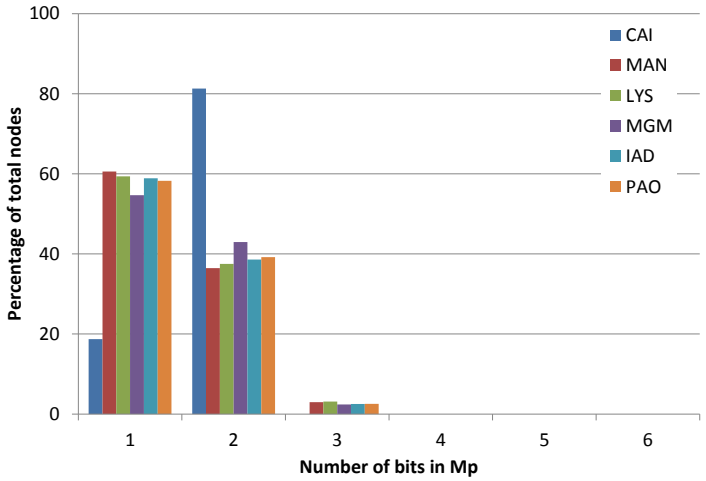
The memory consumption of the Enhanced Compact-trie ($E - Ctrie$) and Enhanced Compact-trie with CT_{ϵ} nodes ($E - Ctrie_{\epsilon}$) were evaluated for the routing tables in Table 3.6. Both these observations were taken without Bloom filters, and these observations extend the results reported for the Ctrie structure in [34], where the effect of P_{trie} variation was evaluated only for one specific routing table. Figure 3.6a shows a distribution of the number of nodes versus the number of prefixes per node in $E - Ctrie$s for the different routing tables. For the large tables, 98% of the nodes have two or fewer prefixes, while for the medium and small tables, the figure is nearly 99.8%. Limiting the number of prefixes per node through the use of the CT_{ϵ} structure can, therefore, improve the memory efficiency.

Memory savings can also be effected by a careful selection of the number of bits allocated to store the $|xe_p|$ and $|ze_p|$ values for prefixes (i.e., M_p and L_p). For the routing tables examined in this work, Figure 3.6b and Figure 3.6c plot the percentage of total prefixes against the number of bits required to represent $|xe_p|$ and $|ze_p|$ values respectively. It can be seen that for all routing table sizes, $|xe_p|$ needs a maximum of three bits, while $|ze_p|$ requires up to five bits. It can also be seen that for prefixes in the smaller tables, $|xe_p|$ values can possibly be represented with two or fewer bits. The correctness of the algorithm in determining the longest matching prefix depends on its ability to correctly distinguish prefixes on the basis of their $|xe_p|$ and $|ze_p|$ values. The choice of the number of bits allocated for $|xe_p|$ and $|ze_p|$ in prefix nodes is critical and therefore bound by the maximum number of bits typically required for any routing table.

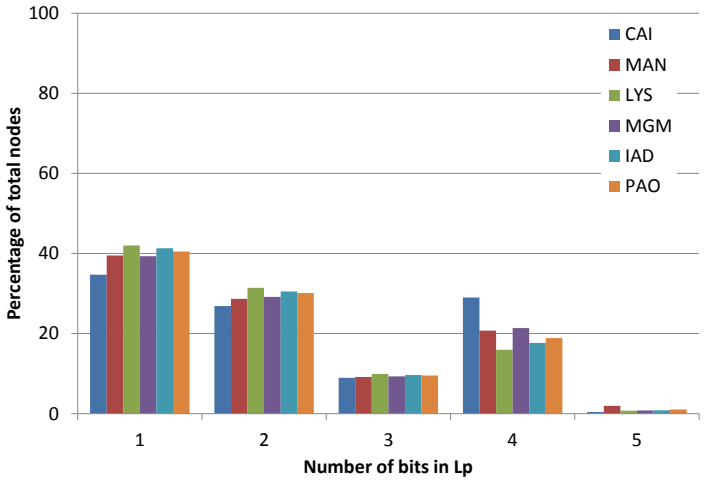
Table 3.7 shows the number of nodes, total memory required and the minimum, maximum and median number of PM stages encountered during a lookup in an $E - Ctrie_{\epsilon}$ for one small, medium and large routing table using different P_{trie} values and (M_p, L_p) = (3, 5) and assuming 8 bits of storage for the output port information (i.e., NHI_p) in prefix nodes [32],[30]. The total memory required is dependent on the number of bits in the child pointers required to correctly address downstream nodes in the trie. If the complete trie structure is stored in a single memory module, the address width is dependent on the total number of nodes. As an example, in the CAI table with about 6000 nodes (3180 prefixes), a child pointer will need 13



(a) Number of prefixes per node in $E - Ctrie$



(b) Bits required for storing $|x|$



(c) Bits required for storing $|z|$

Figure 3.6: Distribution of prefixes, length of $|x|$ and $|z|$ in the $E - Ctrie_\epsilon$ implementation of real routing tables

Table 3.7: Total node count, memory utilisation in Mbits and the depth in the $E - Ctrie_\epsilon$ (min/median/max) where the LPM is found for various P_{trie} values

Table	Metrics	P_{trie}					
		1	2	3	4	5	10
CAI	Nodes (K)	6.00	5.82	5.78	5.77	5.77	5.77
	Memory bits (M)	0.23	0.33	0.39	0.41	0.43	0.41
	LPM found depth	9/25/42	9/24/33	9/23/32	9/23/32	9/23/32	12/23/32
MAN	Nodes (K)	21.0	20.8	20.8	20.8	20.8	20.8
	Memory bits (M)	1.0	1.4	1.7	2.1	2.5	3.6
	LPM found depth	21.17/27	20.86/24	20.85/23	20.80/23	20.79/23	20.82/23
LYS	Nodes (K)	90.27	88.25	88.07	88.04	88.04	88.04
	Memory bits (M)	4.31	5.84	7.18	7.79	8.24	8.25
	LPM found depth	8/24/41	8/23/30	8/23/32	8/23/32	8/23/32	8/23/32
MGM	Nodes (K)	188.4	183.7	183.3	183.3	183.2	183.2
	Memory bits (M)	10.2	13.2	16.5	19.8	23.1	33.0
	LPM found depth	23.04/41	21.54/30	21.29/26	21.26/25	21.25/24	21.25/23
IAD	Nodes (K)	-	540.45	536.79	536.06	535.89	535.83
	Memory bits (M)	-	38.19	47.99	57.65	67.04	61.86
	LPM found depth	-	12/25/53	12/24/43	11/24/40	11/23/37	11/23/33
PAO	Nodes (K)	864.2	808.2	800.2	798.5	798.0	797.9
	Memory bits (M)	50.1	61.4	75.2	89.4	103.7	146.8
	LPM found depth	51/33/116	32/42/64	27/36/47	24/30/38	23/12/35	21/65/31

bits, including a margin for future increases in the number of prefixes. However, for the IAD table with 540450 nodes (∼389k prefixes), each child pointer will need to be at least 20 bits long. The actual number of memory bits required in each case is calculated by inserting the appropriate numbers in the node structure of Figure 3.4. The size of each node in the $E-Ctrie_{\epsilon}$ may, therefore, be set to vary from $26+(18*P_{trie})$ for the smaller tables to $40+(18*P_{trie})$ for the larger tables. The total memory requirement estimated with these expressions is much more accurate than the memory estimated by Erdem et al. [33], because they do not consider the effect of multiple prefixes stored at a single trie node. As the P_{trie} value is increased, a greater number of prefixes can be stored at each node, and therefore the total number of nodes required decreases. However, the memory required per node increases and this, in turn, increases the total memory for the trie. Smaller values of P_{trie} have the opposite effect. If a specific routing table, say LYS, with P_{trie} = 3, is considered, it can be seen from Table 3.7 that the memory required is 22.8% more than that needed when P_{trie} = 2. The P_{trie} change, however, does not significantly alter the average and worst case depth in the trie where lookup finishes. On the other hand, decreasing P_{trie} to 1 brings the memory requirement down by 26.2%. This also increases the median depth for lookup to finish just slightly but in the worst case the depth at which lookup finishes increases by almost 27%. A P_{trie} value of 2 or 3, therefore, achieves an adequate balance between memory efficiency and the number of stages in the trie that need to be traversed.

3.4.2.2 $E - Ctrie_{\epsilon}$ versus binary trie search

The proposed enhanced algorithm has also been benchmarked against a binary trie search algorithm in terms of total memory requirement, trie density (ratio of prefix nodes to total nodes) and the average number of PM stages. The benchmarking is undertaken against a binary trie search and not against some of the other lookup algorithms in the literature [34],[105], [24] [18] for the reason that previous reports on the performance of IP lookup algorithms on FPGAs have been based on a set of routing tables that are quite different from the ones used here. A direct comparison with these results would be unreliable because both the distribution of prefixes within the table as well as the overall size of the table significantly affect the memory requirements. In addition, throughput numbers depend greatly on the characteristic of the FPGA device and its underlying technology node. While it is sometimes possible to scale

performance numbers of simple designs from one FPGA device to another based on the average node performance (perhaps related to the clock frequency), for a complex design such as a Packet Lookup algorithm there are far too many variables such as pipeline depth, memory organisation, storage requirement per node that can affect the throughput, latency and memory utilisation results. To keep the evaluation fair, the present implementation is compared against a basic binary trie (BT) implementation applied to the same set of routing tables and targeted on the same FPGA, just as complex circuit delays in microprocessors are benchmarked against the delay of an inverter with a fanout of 4 (FO4) in the same process node. A direct comparison with the original Compact-trie algorithm in terms of throughput and area is also not undertaken because the original algorithm cannot handle all types of prefixes and the memory calculations are incomplete.

Table 3.8 illustrates the performance of the enhanced algorithm versus a binary trie search. It can be seen that the proposed algorithm exhibits from 17% to 50% improvement in memory utilisation while encountering a marginally lower number of PM stages during a prefix lookup operation. It can also be seen that the $E - Ctrie_\epsilon$ structure has a much higher density than a conventional binary trie for all routing table sizes and the density is in fact much better for the larger tables. A comparison of the original Compact-trie algorithm against a Binary trie [34] has also demonstrated similar improvements in memory utilisation. The memory efficiency as measured by the number of bits required per prefix for the IAD table (388k prefixes) is around 97, while for an equivalent sized table using the Flashtrie algorithm [24] and the Tree Bitmap algorithm [17] the memory efficient is 39 and $\sim$70 respectively. A comparison of the memory efficiencies suggests that the proposed algorithm requires slightly more memory than other algorithms available in the literature.

3.4.2.3 $E - Ctrie_\epsilon$ and Bloom filters

The performance of the Enhanced Compact-trie structure is then evaluated with Bloom filters with the same set of routing tables and the same synthetic packet traces. Figure 3.7 shows the average number of prefix memory accesses required for one each of the small, medium and large routing tables for different values of Bloom filters. It can be observed that the average number of accesses reduce considerably with increasing Bloom filter sizes until a scaling factor of 'M_{bf}' = 8. Increasing the Bloom filter size beyond this does not result in any significant

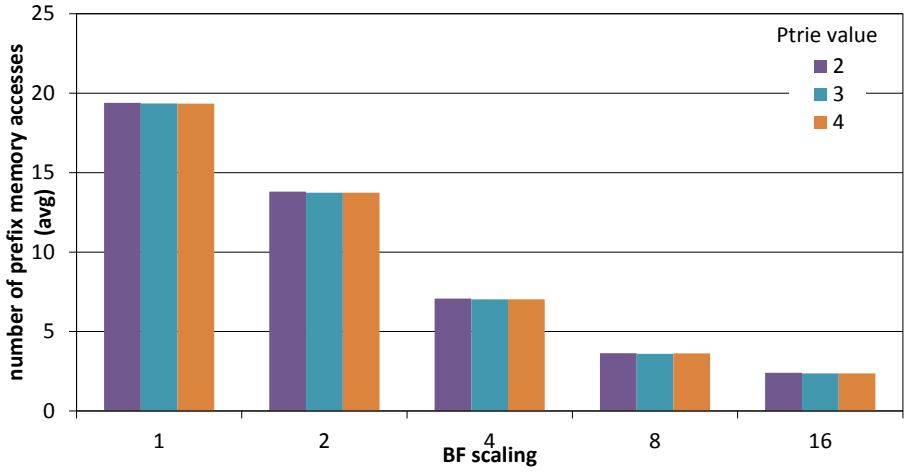
Table 3.8: Comparison of binary trie and $E - Ctrie_{\epsilon}$ implementations (P_{trie} = 2) for all routing tables

Table	Number of Nodes(k) / Total Memory (Mbits)		Trie density (%)		Median number of PM stages	
	$Btrie$	$E - Ctrie_{\epsilon}$	$Btrie$	$E - Ctrie_{\epsilon}$	$Btrie$	$E - Ctrie_{\epsilon}$
CAI	16.2/ 0.6	5.8/ 0.3	19.6	51.5	25	24
MAN	41.3/ 1.7	20.8/ 1.2	15.2	29.5	25	23
LYS	198.8/ 8.7	88.3/ 5.8	22.3	48.1	25	23
MGM	460.7/ 21.2	183.7/12.2	20.8	49.8	25	23
IAD	1517.7/ 75.9	540.5/38.2	25.6	66.5	25	25
PAO	2453.4/127.6	808.2/56.4	25.6	70.9	25	25

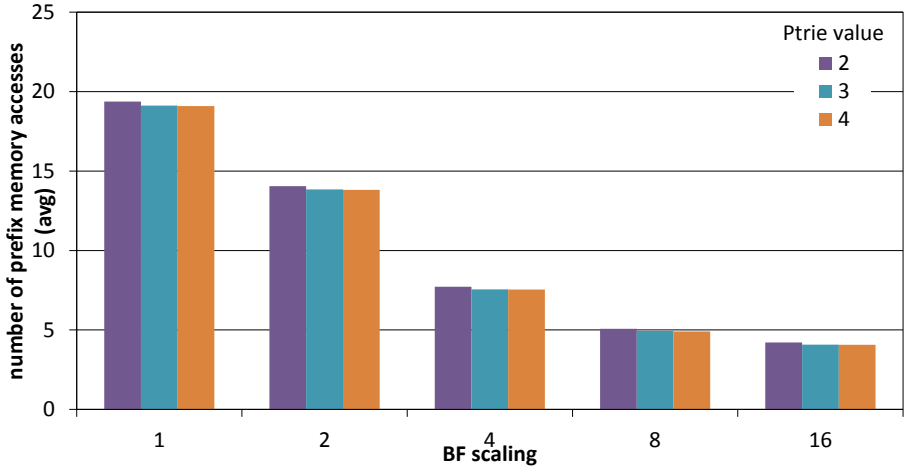
reduction in the number of memory accesses. On the contrary, making the Bloom filters any larger will increase the number of bits required to address the individual Bloom filter locations. This will adversely affect the frequency of operation of the Bloom filter and also make it difficult to accommodate the Bloom filter memory on-chip. It is also seen that the number of accesses required is not affected significantly by the P_{trie} value.

Table 3.9 shows a comparison of the number of prefix memory accesses (average and maximum) for all the prefix tables first without Bloom filters and then with monolithic and pipelined Bloom filters. The table also shows the number of false memory accesses. It can be seen that irrespective of the size of the routing table, both the average and the maximum number of prefix memory accesses required to find the longest prefix match reduce with the introduction of a Bloom filter. With a monolithic Bloom filter, the reduction in average prefix memory accesses is around 80% for the small and medium tables, while for the large tables the reduction is smaller – 67% for IAD and 53% for PAO. The worst case number of prefix memory accesses improve about 56% for the small tables and about 30% for the large tables. The experiments on a pipelined Bloom filter demonstrate a greater saving for the larger tables than the smaller tables. For the IAD table the average and worst case number of prefix memory accesses are 74% and 53% better with Bloom filters, while for the PAO table, the same metrics show an improvement of 71% and 57%.

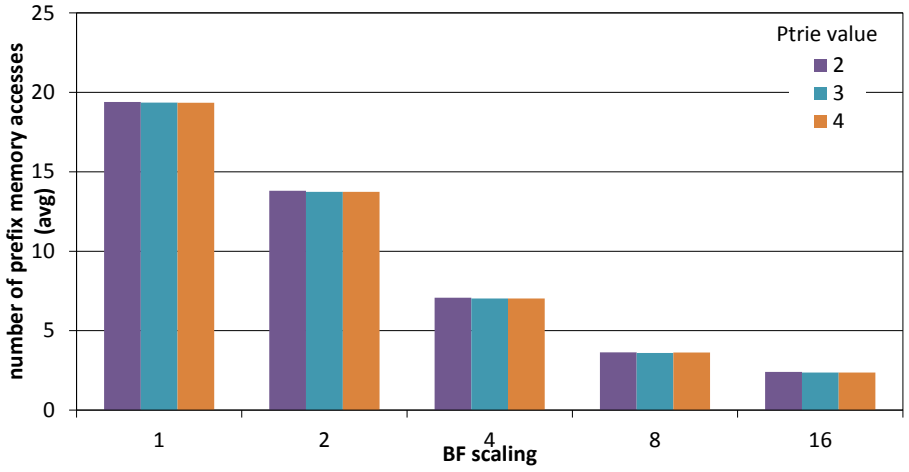
As for the false accesses, the first number in these columns are the accesses due to the conventional 'false positive' effect of Bloom filters. This is when the Bloom filter indicates a false positive because the indexed bits have been set by other prefixes in the table. The number of such accesses ranges from 0.2 per lookup for the small CAI table to about 0.6 per lookup for the



(a) MAN



(b) MGM



(c) IAD

Figure 3.7: Average number of memory accesses for varying Bloom filter scaling factors and P_{trie} values

Table 3.9: Comparison of $E - Ctrie_\epsilon$ implementations for all routing tables in three cases - without Bloom filter, with monolithic Bloom filter and with pipelined Bloom filter ($P_{trie} = 2$, Bloom filter scaling factor $M = 8$)

Prefix Table	trie nodes		total lookups	Number of Prefix Memory accesses							
	total	prefix		w/o BF		monolithic BF			pipelined BF		
				avg	max	avg	max	false	avg	max	false
CAI	5820	3001	15910	22.3	32	5.1	14	0.3, 2.7	4.94	14	0.2, 2.6
MAN	20827	6148	31568	20.9	24	2.4	9	0.4, 1.2	2.30	9	0.3, 1.1
LYS	88259	42413	222215	21.8	31	3.6	14	0.3, 2.2	3.56	14	0.4, 2.2
MGM	183709	91534	482084	21.5	30	4.2	14	0.4, 2.8	3.99	13	0.4, 2.6
IAD	540452	359272	1942110	25.6	52	8.4	36	0.4, 6.8	6.62	24	0.4, 5.0
PAO	808183	573555	3144908	32.4	64	15.1	46	0.1, 13.2	9.31	27	0.6, 7.6

large PAO table with pipelined Bloom filters. The second number in the same column is the number of false positives that occur due to a peculiarity of the Compact-trie. CRC generators used to compute the Bloom filter indices in a Compact-trie receive at their input a string composed only of the MSB_k, M_k and AP_k information of the key. With large tables, it is possible that a key being searched may not have a matching prefix, but may have the same MSB_k, M_k and AP_k as an existing prefix. As a result, the CRC generator will generate the Bloom filter indices that would result in a match. This is different from the conventional false positive behaviour because here the key used to generate the indices itself is matching and there is nothing in the Bloom filter to filter out this false access. The number of such accesses ranges from 2.6 per lookup for the small table to as high as 7.6 accesses per lookup for the large table with pipelined Bloom filters.

3.4.3 Circuit simulations for target FPGA

The idea of using Bloom filtering with external memory has already been shown to improve lookup latency in lookup algorithms that use hash tables and trie structures [27], [28], [30]. While the present work demonstrated through software simulations that a similar benefit could be achieved by applying Bloom filters to the Compact-trie, to understand if this approach would lead to different observations if the on-chip memory was used, it was necessary to target the algorithm for an FPGA.

Verilog prototypes for pipelined $E - Ctrie_\epsilon$ structures with and without Bloom filters were created for the medium-sized LYS and MGM tables on a Xilinx Virtex-7 xc7vx330tffg1157-3 FPGA using Vivado 2017.1. This device has 204000 LUTs and 750 blocks of 36Kb RAM, which means

that the large prefix tables could not be accommodated on this device and were, therefore, not evaluated. In the FPGA design, trie-traversal information RAM, prefix information RAM and Bloom filter RAM were all implemented using the on-chip 36Kb block-RAMs. All blocks were fully pipelined, enabling a prefix lookup every clock cycle. A consequence of using on-chip RAM blocks and a fully pipelined design is that the time required to access the trie traversal information and the time required to access prefix memory is the same. This means that while the trie-traversal information is accessed, the prefix information can also be accessed and the expected benefit of Bloom filtering to improve latency is not really achieved. However, it was predicted that the reduced activity in prefix information RAMs as a result of Bloom filtering would lead to power savings, as was observed in the results.

3.4.3.1 Resource utilisation: Monolithic versus Pipelined Bloom Filter

A comparison of the resource utilisation between a single Bloom filter and a pipelined Bloom filter is shown in Table 3.10. For the pipelined Bloom filter, the table shows both the largest Bloom filter in the pipeline and also the total size of all Bloom filters, followed in brackets by the percentage penalty (+ve) or benefit (-ve) of pipelining. It can be seen that for both the smaller prefix tables (CAI and MAN), the pipelined Bloom filter actually consumes more memory bits than a single Bloom filter. However, for the medium sized prefix tables and also for the larger PAO table, the pipelined Bloom filter consumes less memory than the single Bloom filter as seen in Table 3.10 while performing better as seen in Table 3.9. For the large IAD exchange table, however, the pipelined Bloom filter is larger than the single Bloom filter. These results suggest that the memory penalty of a pipelined Bloom filter is as much a function of the prefix length distribution as the size of the prefix table.

3.4.3.2 Total Resource utilisation and power consumption with pipelined Bloom filters

The resource utilisation and power consumption of the FPGA implementations with and without filtering are detailed in Table 3.11. As expected, it can be seen that the presence of the Bloom filter increases the logic as well as block RAM utilisation on the device. However, the effect of Bloom filtering on power consumption within block RAMs and logic is different. While the average power consumed by the prefix RAM reduces by around 70%, there is now a new Bloom filter RAM that contributes to the total power consumption. This new component consumes

Table 3.10: Comparison of memory requirement of monolithic Bloom filter and pipelined Bloom filter (BF scaling factor M = 8)

Prefix Table	monolithic BF	pipelined BF	
	Total (kb)	largest BF (kb)	Total (kb)
CAI	32.7	8.1	35.0 (6.9%)
MAN	65.5	16.3	67.6 (3.3%)
LYS	524.3	131.1	493.9 (-5.8%)
MGM	1048.6	262.1	1013.0 (-3.4%)
IAD	4194.3	524.3	4595.0 (9.6%)
PAO	8388.6	524.3	7246.1 (-13.6%)

Table 3.11: Comparison of $E - Ctrie_\epsilon$ implementation for LYS and MGM tables with and without Bloom filters

		LYS		MGM	
		w/o BF	w/ BF	w/o BF	w/ BF
LUTs	Logic	6951	8527	6774	8406
	Mem	2147	3330	1844	2973
Registers		9460	13191	8674	12404
Block RAMs (36 kb)		259.0	284.5	556.0	595.5
Power (mW)	LUT	79	105	84	119
	Prefix RAM	607	140	1068	324
	BF RAM	-	155	-	178
	Total	1723	1531	2774	2487
Max Freq. (MHz)		318	311	307	304

about 17–26% of the power consumed by the original prefix RAM. The actual CRC computation increases the logic power by about 35–40%. Overall the addition of Bloom filter decreases the total chip power including signal power, IO power and clock power by around 10–12% and marginally decreases the overall operating frequency by 1–2%.

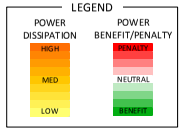
3.4.3.3 Power distribution down trie levels and targeted Bloom filtering

Another important effect to be considered when applying Bloom filtering is illustrated in Figure 3.8, which shows a map of the power consumption in the RAM components at different stages in the pipelined $E - Ctrie_\epsilon$ with and without Bloom filters. The results have once again been presented for the LYS and MGM tables. For the pipelined $E - Ctrie_\epsilon$ implementation without a Bloom filter, the power consumed in the prefix information RAM alone is shown.

Trie level	W/o BF PFX RAM	W/ BF PFX RAM	W/ BF BF RAM	W/ BF total	difference	BF size
0	1	1	5	6	5	0.0
1	2	1	5	6	4	0.0
2	2	1	5	6	4	0.0
3	2	1	5	6	4	0.0
4	2	1	5	6	4	0.0
5	4	1	5	6	2	0.1
6	4	0.2	5	5.2	1.2	0.1
7	4	0.2	5	5.2	1.2	0.1
7	4	0.2	5	5.2	1.2	0.5
8	4	0.2	5	5.2	1.2	0.5
9	5	0.2	5	5.2	0.2	1.0
10	11	1	5	6	-5	2.0
11	11	2	5	7	-4	4.1
12	22	4	5	9	-13	4.1
13	22	4	5	9	-13	8.2
14	48	11	5	16	-32	16.4
15	48	13	5	18	-30	16.4
16	47	13	10	23	-24	32.8
18	88	28	10	38	-50	32.8
19	83	5	14	19	-64	65.5
20	76	29	14	43	-33	65.5
21	63	4	11	15	-48	65.5
22	46	15	9	24	-22	131.1
23	3	0.2	1	1.2	-1.8	32.8
24	0.2	0.2	0.2	0.4	0.2	8.2
25	0.2	0.2	0.2	0.4	0.2	4.1
26	0.2	0.2	0.2	0.4	0.2	1.0
27	0.2	0.2	0.2	0.4	0.2	1.0
28	2	1	0.2	1.2	-0.8	0.0
29	2	1	0.2	1.2	-0.8	0.0
30	1	1	0.2	1.2	0.2	0.0

All power numbers are specified in mW. Bloom filter size is in kbits.

(a) LYS



Trie level	W/o BF PFX RAM	W/ BF PFX RAM	W/ BF BF RAM	W/ BF total	difference	BF size
0	1	2	5	7	6	0.0
1	2	0.2	5	5.2	3.2	0.0
2	2	0.2	5	5.2	3.2	0.0
3	2	0.2	5	5.2	3.2	0.0
4	2	0.2	5	5.2	3.2	0.1
5	4	0.2	5	5.2	1.2	0.1
6	4	0.2	5	5.2	1.2	0.3
7	4	0.2	5	5.2	1.2	0.5
8	4	0.2	5	5.2	1.2	1.0
9	4	0.2	5	5.2	1.2	2.0
10	5	0.2	5	5.2	0.2	2.0
11	11	2	5	7	-4	4.1
12	22	4	5	9	-13	8.2
13	49	10	5	15	-34	16.4
14	49	11	5	16	-33	16.4
15	96	24	11	35	-61	32.8
16	95	24	16	40	-55	65.5
17	184	51	16	67	-117	65.5
18	151	54	15	69	-82	65.5
19	126	41	14	55	-71	131.1
20	104	43	12	55	-49	131.1
21	82	35	9	44	-38	131.1
22	58	18	7	25	-33	262.1
23	4	0.2	2	2.2	-1.8	65.5
24	0.2	0.2	0.2	0.4	0.2	8.2
25	0.2	0.2	0.2	0.4	0.2	2.0
26	0.2	0.2	0.2	0.4	0.2	0.5
27	0.2	0.2	0.2	0.4	0.2	0.5
28	0.2	0.2	0.2	0.4	0.2	0.3
29	2	2	0.2	2.2	0.2	0.0

All power numbers are specified in mW. Bloom filter size is in kbits

(b) MGM

Figure 3.8: Power consumption in prefix information RAM (with and without Bloom filtering) and Bloom filter RAM (with Bloom filtering) at different levels in the pipelined trie for two routing tables

While for the implementation with Bloom filters the power consumption in both the prefix information RAM and the Bloom filter RAM is shown along with the total RAM power consumption. It is observed that for the 'with Bloom filter' case, the total power consumption in the RAMs is greatest in the stages that have the largest Bloom filter. Figure 3.8 also shows the difference in total power consumption with and without Bloom filters at each level in the trie. A negative value for the difference means that the total power consumed with Bloom filters is less than the total power consumed without Bloom filters. For these prefix tables, the Bloom filter at levels closest to the head node in the trie add a significant power penalty, while Bloom filters at levels 10–23 result in power savings. Of the existing modified trie traversal algorithmic approaches, [18] is one of the few that reports the power consumption of their non-pipelined, simple-pipelined and memory-balanced pipelined implementations. While it would be inappropriate to compare the numbers directly, since the results have been obtained with different routing tables, it can be seen that the minimum power consumption reported in [18] with optimised parameters is $\sim$ 1000 mW in the simple pipelined case and $\sim$ 4000 mW for the memory-balanced pipelined case. The power consumption of the proposed Bloom filtering approach is also around 2000 mW, which is roughly the same order of magnitude. A direct comparison is also difficult because [18] has not reported on the frequency of their clock. An analysis of FPGA hot-spots was also not undertaken and is outside the scope of the thesis

Finally, in light of the energy consumption gradient observed down levels of the trie, a *targeted* Bloom filtering approach was explored, where the filters are instantiated only on specific levels to achieve a better trade-off between performance, power and area. The experiment was set up with an $E - Ctrie_{\epsilon}$ implementation of the MGM exchange table with Bloom filters instantiated only from levels 11 to 22, which correspond to the power saving levels of Figure 3.8. For all other stages in the pipelined trie, the prefix information is accessed without filtering. The results presented in Table. 3.12 show that the LUT, Register and Block RAM sizes are almost the same as with filtering at all levels (only $\tilde{4}$% smaller in each case). Additionally, both the block RAM and total power consumption are significantly lower in the targeted approach compared with the corresponding figures without Bloom filtering (75% and 25%, respectively) and similarly lower compared to the total Bloom filtering case (37% and 16%, respectively). Further, the resulting small increase in the average number of prefix memory accesses will not affect the throughput as the system is still able to achieve one lookup per clock cycle because of the pipelined design.

Table 3.12: Comparison of an $E - Ctrie_\epsilon$ implementation for the MGM routing table in three cases - with targeted Bloom filtering, with total Bloom filtering and without Bloom filtering

		BF instances		
		none	targeted	all levels
LUTs	Logic	6774	7921	8406
	Mem	1844	2954	2973
Registers		8674	11901	12404
Block RAMs (36 kb)		556.0	579.5	595.5
Power (mW)	LUT	84	111	119
	Prefix RAM	1068	206	324
	BF RAM	-	106	178
	Total	2774	2081	2487

Summary

This chapter has first provided an enhanced prefix decomposition technique for an existing algorithm to ensure that the correct next hop information was looked up for incoming IP packets in all IPv4 routing tables. Comprehensive software simulations show that the Enhanced Compact-trie with epsilon links ($E - Ctrie_{\epsilon}$) performs significantly better than a binary trie implementation. A Modelsim simulation for the target FPGA with real IPv4 routing tables achieves a throughput of ∼300 million lookups per second while requiring on an average the same number of prefix match stages as a binary trie implementation.

The performance of the trie search algorithm was further improved by the addition of Bloom filtering. Software simulations demonstrate that an appropriately sized filter reduces worst case memory accesses to prefix storage RAM by about 50-60% and the average access rate by almost 80% in some routing tables. If it is assumed that prefix storage is in external memory, then the experiments presented here suggest that the application of Bloom filters to the Compact-trie could result in an improvement in latency.

The specific implementation of the algorithm with Bloom filtering on FPGA, however, assumed on-chip storage only and so latency improvements were not available. However, in cases where the prefix information is compact enough to be stored on-chip, a pipelined Bloom filter can lead to useful power savings. An important observation from the experiments is that power saving with Bloom filters is not uniform down the trie. A targeted Bloom filtering approach can be used to further reduce power consumption while using fewer on-chip resources and without affecting lookup performance.

It is essential to evaluate if an NCL implementation of the Compact-trie demonstrates a similar improvement in power consumption after the addition of a Bloom filter. While the final NCL design evaluation is discussed in Chapter 6, the next chapter discusses the latency versus throughput tradeoff in NCL circuits and proposes a small number of modified NCL circuit designs to improve throughput.

> It doesn't stop being magic just because you know how it works.
>
> Terry Pratchett, The Wee Free Men

Chapter 4

Bloom Filters and Hashing Functions

Since its invention almost 50 years ago, the Bloom filter has been applied to a wide variety of memory intensive processes, from advanced cache controllers to search algorithms for extremely large databases. As outlined in Chapter 2, this probabilistic data structure has an important characteristic that makes it particularly useful in IP address lookups: false positive matches are possible, but false negatives are extremely unlikely. As a result, it can be used to greatly reduce the number of lookups required compared to a system that does not use them while eliminating the risk of missing an address that is actually present.

This chapter describes specifics of the NCL implementation of hashing algorithms used in Bloom filters for address lookup in IP routers. The algorithms are first evaluated for their appropriateness to an NCL implementation, and the latency, throughput and energy performance of the NCL designs are contrasted against equivalent metrics obtained for Boolean circuits. The effect of pipelining on the latency and cycle time of specific blocks within the NCL design has also been explained.

NCL systems, in common with all other asynchronous logic systems, do not require low-skew, high performance, high-power clock trees. However, they do have the local *Completion Detection* (CD) tree structures that generated the signals required for handshaking. The throughput of NCL systems is thus dependent not only on the delay of the forward path but also on the delay of these Completion Detection circuits in the reverse path. The CD circuits span the width of the data path and can result in extremely large fan-in, path delays and occupy a large area on the die. Going by the guiding principle of not just applying 'asynchronous techniques' to synchronous designs, optimisations to the completion detection circuits are also explored and evaluated in this chapter.

4.1 Hash function implementation

The focus of the present work is the lookup of 32-bit IPv4 destination addresses and therefore only 32-bit truncated versions of all mathematical operations are implemented[1]. It is assumed that the hash values index into a 1024-location Bloom filter. The hash output, therefore, needs to be 10 bits long. Hashes used in Bloom filters should be able to generate multiple hash values from the same key value. Kirsch et al. [106] has shown that if there exists a set of hash values, generated using a class of hash functions; newer hash values can be generated by specific mathematical and logical operations on these hash values or by selecting subsets of a single hash value without having to design multiple function generators. Interestingly, the three hash generators considered here (Jenkins, Murmur and CRC32) already have the ability to generate multiple hash values from the same key with different seeds, without needing a new generator implementation.

Taking into consideration arguments based on the available literature, it became clear that the CRC32 hash has an acceptable hashing performance, its latency and area utilisation is appropriate, and its avalanche behaviour may be sufficient for the particular application studied in this thesis. However, it was also considered possible that an NCL implementation of the Jenkins and Murmur hash might throw up some unexpected results with regards to the circuit operation and timing characteristics and so, along with the CRC32 hash, these were also implemented in Null Convention Logic and their cycle time and latency measured and compared.

Boolean and NCL circuits were designed for the individual building blocks in the hashes and assembled to build the complete hash function. The XOR function is relatively simple to implement while hardware rotate and shift functions involve only a reordering of the nets. The adder used in the designs has a Kogge-Stone architecture. The multiplier, however, is the most complex block and has the greatest latency. The multiplier consists of a Booth partial product generator, Wallace-Tree adder and a Kogge-Stone adder. The Wallace-tree adder is truncated at 32 bits, which reduces its complexity and latency, as the number of compressor stages is reduced [42]. Three different pipelined versions of the multiplier were designed to control the

[1]While IPv6 is outside the scope of the work, it is possible to make some specific observations about the Jenkins, Murmur and serial CRC32 algorithms. Hashing an IPv6 address would involve feeding in the 128-bit IP address in 32-bit chunks, increasing the latency to 4× the latency of the IPv4 implementation. For the parallel CRC32 implementation the latency would be the same for IPv4 and IPv6. However, an implementation of the latter would need a significantly larger area as the XOR operations performed at each stage of the CRC32 shift register would now have a greater number of inputs. However, the overall organisation of the architecture will remain the same.

cycle time in the final design. One of the versions had no internal registers but only had the input and output registration stages. Another version was the coarse-pipelined multiplier that had, in addition to the registers at the input and output, registers after the Booth encoder, Wallace tree adder and the Kogge-Stone adder. The final version was the fine-pipelined multiplier that had registration stages inside the adders and partial product generators. The multiplier served as a useful test system for evaluating the effect of pipelining granularity on the cycle time and latency of NCL circuits.

Figure 4.1 is a block diagram of the Murmur hash engine implemented as an NCL pipeline. The block diagram shows a registration stage after each complex block such as a multiplier or an adder. The XOR and rotate functions are not registered individually. Registration stages are instead added after a combination of two or three such low latency stages. This helps maintain uniform cycle time. The multiplier used in the final design is the non-pipelined multiplier. The Jenkins hash engine has a structure very similar to the Murmur hash engine, but it uses only addition, XOR, rotate and shift operations. The Jenkins hash can, therefore, be expected to be faster than the Murmur hash due to the absence of the multiplication operation. A block diagram of the Jenkins hash is shown in Figure 4.2 while a block diagram of a serial implementation of the CRC32 hash algorithm is shown in Figure 4.3. Of course, one problem with the serial implementation is that the 32 bits of the IP address (key) have to be fed in serially and would, therefore, need 32 Data and Null wavefronts to pass through before the CRC32 value is obtained. In the final design, the CRC32 is instead generated using a parallel implementation of the CRC32 polynomial as in [107].

The 'auto-produce' and 'auto-consume' modules are important components of the NCL test infrastructure that were also developed for these designs. While the basic concept has been explained by Fant [38], the idea has been extended here so that it generates multi-bit wide pseudo-random Data and Null wavefronts to the DUT. The auto-produce block is important because it generates and makes test vectors available to the 'Device Under Test' (DUT) as soon as the technology and operating conditions will allow. This functionality is achieved by designing a multi-bit NCL shift register/memory that is loaded with a sequence of Data and Null vectors through a separate data port immediately after reset as part of the initialisation process. The ports used to load the test vectors are seen on the top left side of the block in Fig. 4.4. While the test vectors are being loaded, the DUT is kept under reset. Once the test vectors have been

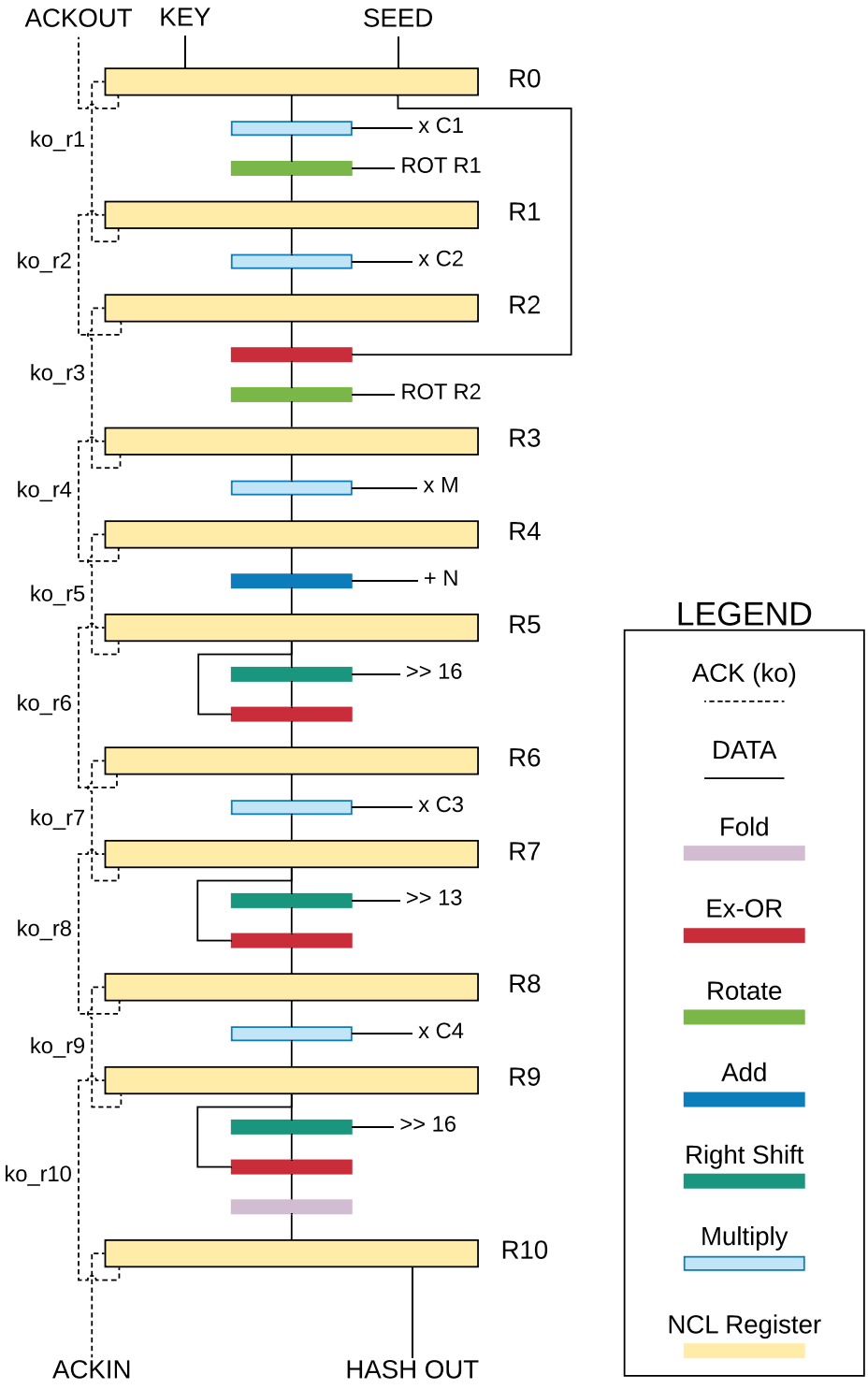


Figure 4.1: Murmur Hash block diagram

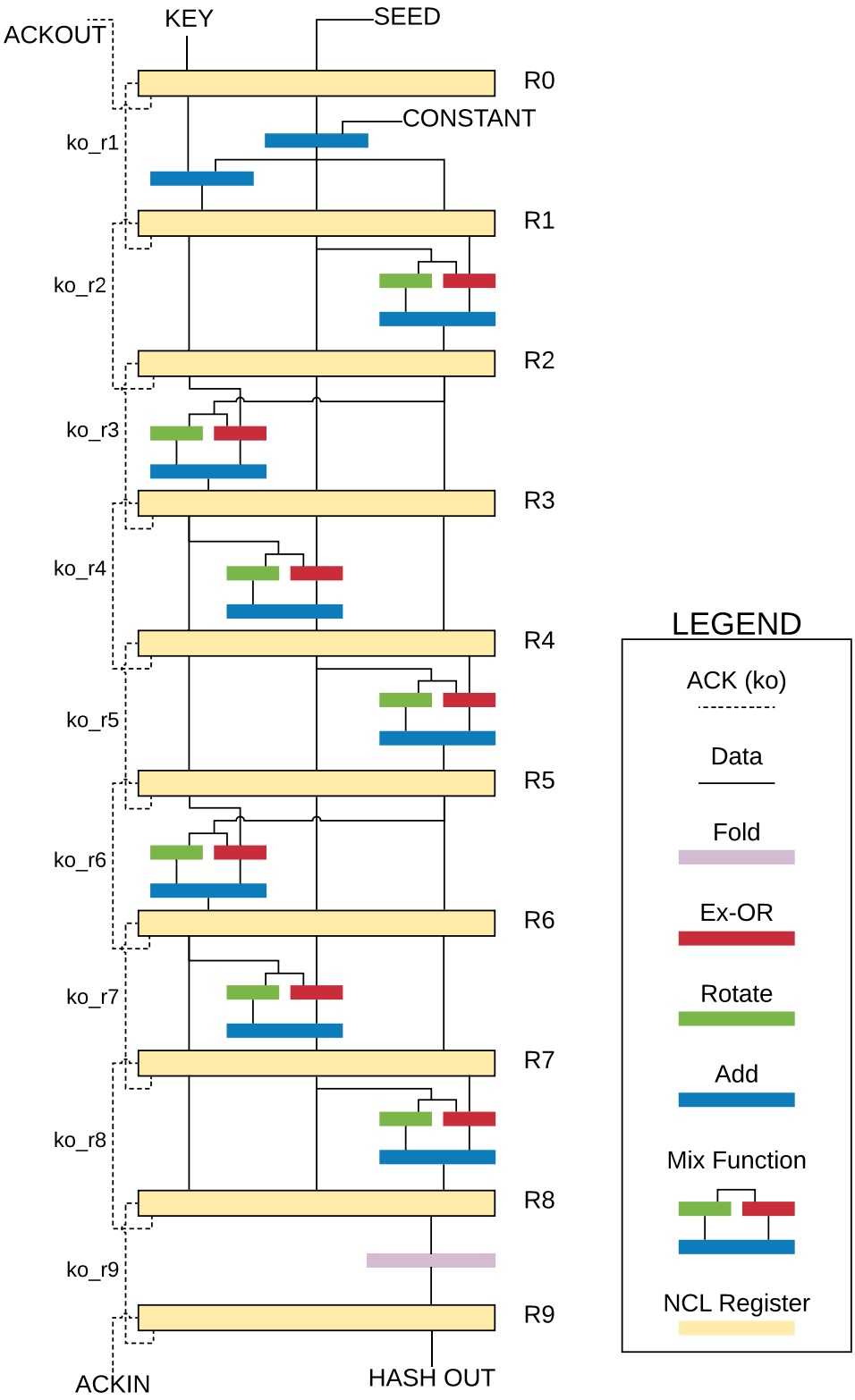


Figure 4.2: Jenkins Hash block diagram

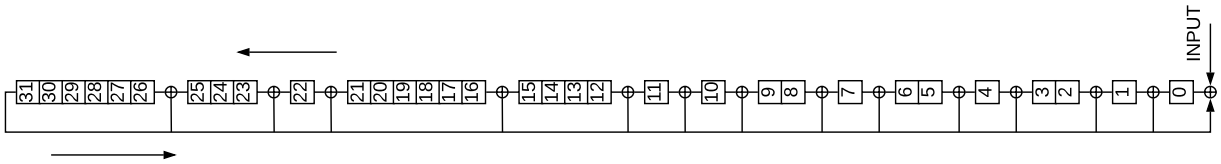


Figure 4.3: CRC32 hash block diagram

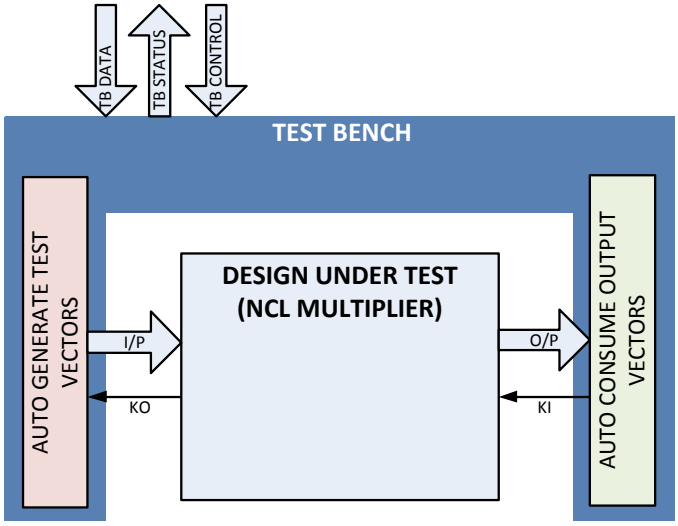


Figure 4.4: Test setup for measuring latency and throughput of an NCL design

loaded, the DUT is taken out of reset. Since the DUT is initialised to an all Null condition, the moment it comes out of reset it requests a Data value from the test bench. The auto-produce block in the test bench is ready to supply this value with negligible latency. The Data wavefront is processed within the DUT, and after it passes the first registration stage, the Null request is generated for the Test bench, which is also supplied immediately. This sequence of Data and Null wavefronts then flow continuously and automatically without requiring any additional inputs from the test bench.

The auto-produce block on its own is, however, not enough to achieve a self-timed operation. Unless a Data value is being requested at the output of the DUT the wavefront produced by the auto-produce block will not be *pulled* through the input ports. For this reason, an 'auto-consume' block that tests for output Data completeness and generates the appropriate request signals for the DUT is needed. The auto-produce and auto-consume blocks together ensure that

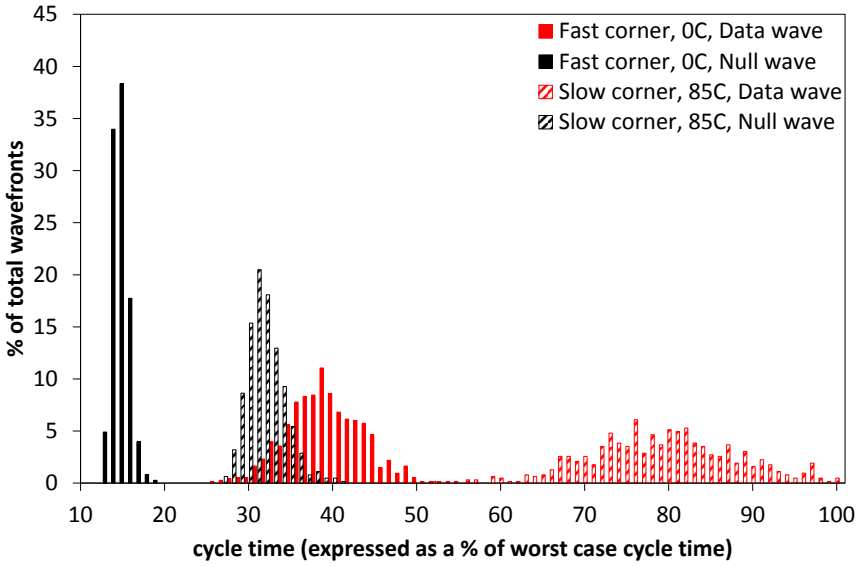
the flow of Data and Null wavefronts in the NCL pipeline are self-timed and not synchronised to an external signal (e.g., a test clock), as is the case in synchronous test benches.

4.2 Hash function - performance evaluation

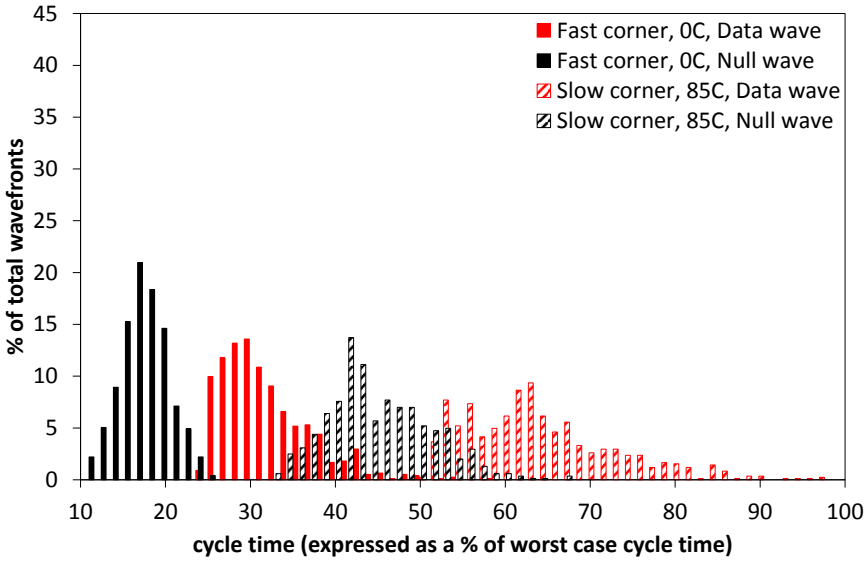
For the purpose of this study, all three hash functions were coded as behavioural NCL in a proprietary Hardware Description Language known as "NELL", and translated using a synthesis tool into structural Verilog netlist instantiating components from a library of NCL threshold gates. Both the HDL and the synthesis tool were provided by Wave Semiconductor [108]. The threshold gate circuits were themselves designed in-house (not by the author) in Cadence Virtuoso using a 1V, 28nm ultra-thin body and BOX silicon-on-insulator (UTBB-SOI) process [109].

Synchronous implementations were also created in Verilog and translated using Cadence RTL Compiler®and synthesised using the digital standard cell library from the same UTBB-SOI process kit to enable a comparison of the performance of the NCL circuits with their Boolean counterparts. The netlists were then imported into Cadence Virtuoso®and simulated using Cadence UltraSim®at $0\,^{\circ}\mathrm{C}$, $27\,^{\circ}\mathrm{C}$ and $85\,^{\circ}\mathrm{C}$. The absolute cycle times and latencies of the NCL and Boolean implementations were then measured. In the NCL implementation, the DUT is initialised to the Null state immediately after reset and is fed test vectors from the auto-produce block in the test bench. The test vectors are 32-bit pseudo-random values.

Figure 4.5 shows a comparison of the spread of cycle times for the Jenkins and Murmur hash engines at $0\,^{\circ}\mathrm{C}$ and $85\,^{\circ}\mathrm{C}$ and two different process corners. It can be seen that cycle times vary between a maximum and minimum value for different input vectors and also vary for the Data and the Null wavefronts. It is also clear that for the Jenkins hash engine, the cycle times are distributed uniformly around a mean value, and the spread of the Data wavefront is higher than the spread of the Null wavefront. For the Murmur hash engine, the cycle times for the Data wavefronts are clustered towards the left (shorter cycle times), and the spread has a long tail to the right (longer times). This behaviour is due to the presence of the multiplier stages followed by the XOR and rotate stages in the Murmur hash. The multiplier stages exhibit extended cycle times for some input vectors, whereas the XOR and rotate stages have a much smaller and uniform cycle time for all input vectors. It is also clear that the cycle times of all three engines are dependent on the operating condition. The 'fast 0C' timing model results in a smaller cycle time for both the Data and Null wavefronts than the 'slow 85C' timing model. It is to be



(a) Jenkins hash



(b) Murmur hash

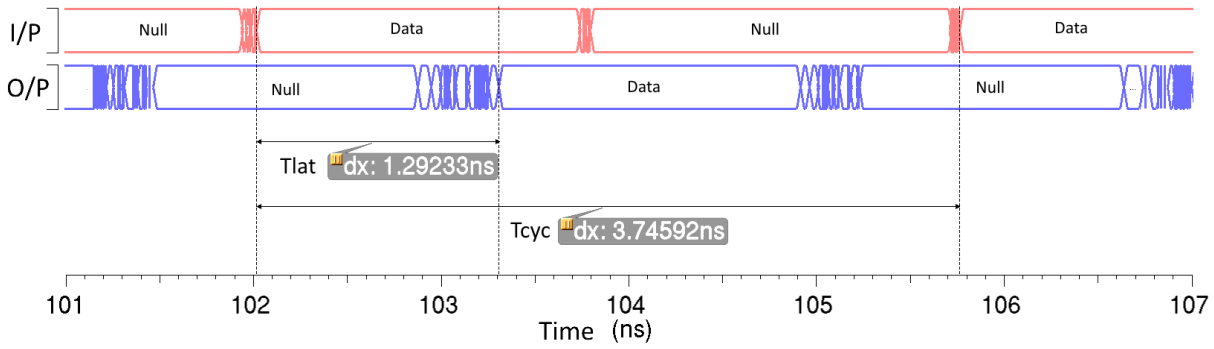
Figure 4.5: Cycle time variation at $0\,^{\circ}\mathrm{C}$, $85\,^{\circ}\mathrm{C}$, slow and fast process corners for Data and Null wavefronts in NCL implementations of hash functions

expected that the cycle times for typical operating conditions would lie between these two extremes and the throughput of the system is 'self-regulated'. This is in contrast to synchronous systems, where the operating condition that results in the worst case timing as obtained by static timing analysis would dictate the maximum allowable clock frequency. The CRC32 implementation while demonstrating different cycle times at the different operating conditions, did not the kind of spread demonstrated by the Jenkins and Murmur hash implementations because of the relatively simpler structure.

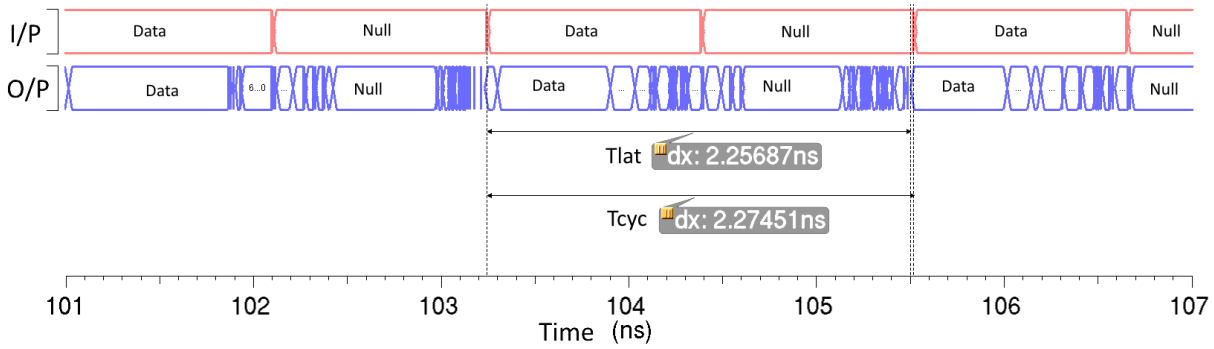
An important consideration in the cycle time and latency of NCL circuits is the number of pipeline stages in the design. This idea can be explained better by observing the behaviour of a single multiplier in the circuit as shown in the wave propagation in Figure 4.6. In this case, as the number of pipeline stages in the multiplier is increased, the forward delay increases only slightly by an amount equal to the delay of the additional registration stage. However, the total cycle time reduces significantly. The latency and cycle times for each of the three pipelining cases have been plotted in Fig. 4.6. It can be seen for the non-pipelined cases, that the latency is extremely low, but the cycle times are high. Interestingly in the case of a fully-pipelined design, the cycle times have improved, but the degradation in latency is small. Latency and cycle times for the coarsely pipelined design sit somewhere in between these two extremes. A comparison of the latency and cycle times for the non-pipelined and fully-pipelined designs also suggests that in NCL systems with wide data paths, the delay in the reverse (acknowledge) path is comparable to the delay in the forward paths. The completion detection network is the only logical component in the reverse path and therefore improving its performance presents an opportunity to improve the throughput of the overall system.

In the fully pipelined multiplier design, which includes 15 pipelining stages the cycle time is half of the non-pipelined design, equivalent to a doubling of the throughput. However, the latency increases to only $\sim$1.4 times the cycle time (i.e., 2.26 ns in this case). This can be compared with a clocked Boolean system, where the addition of 15 pipelining stages would increase the latency to 15 times the clock period. If we assume the Boolean system to work with a clock period of 1.6ns, i.e a throughput of $\sim$610 Mops/sec, it would have a latency of approximately 25 ns, an order of magnitude larger than the asynchronous case.

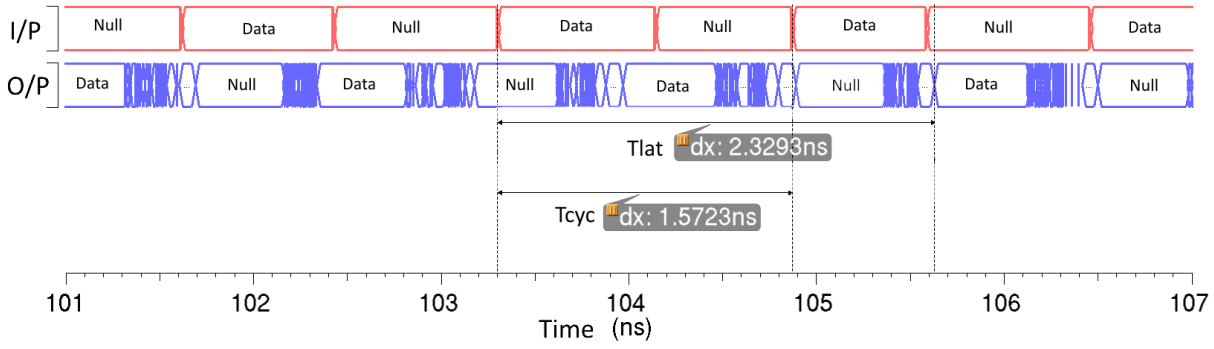
When comparing the behaviour of the complete hashing algorithms, it is to be expected that both the Jenkins and the Murmur hash implementations will behave in an almost identical



(a) no pipelining



(b) coarse pipelining



(c) fine pipelining

Figure 4.6: Latency and cycle time of a 32-bit Booth Wallace multiplier for three different pipelining conditions

Table 4.1: Comparison of NCL implementation of Jenkins and CRC32 hash

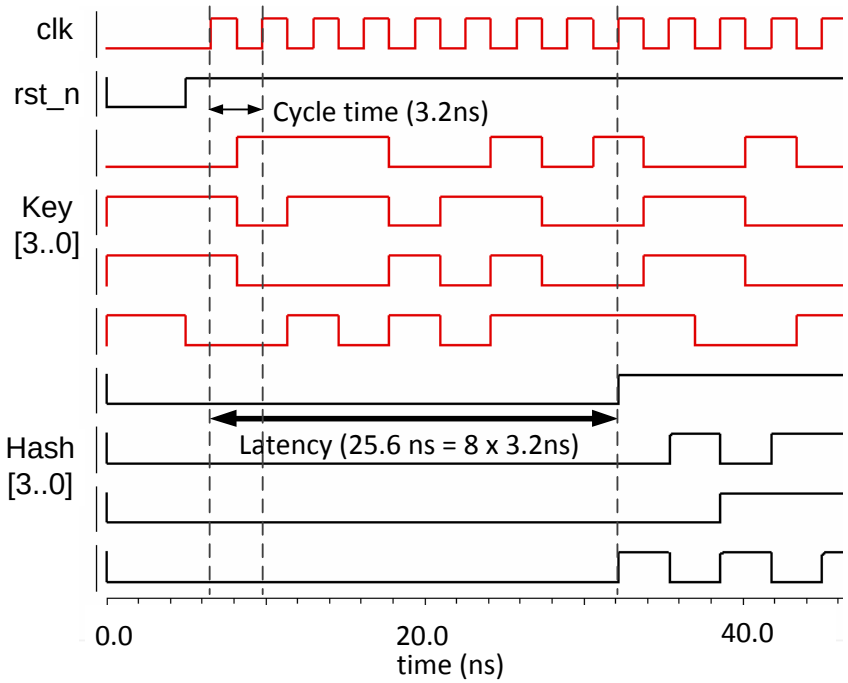
Hash	cycle time(ns)	latency	peak supply current (mA)	size
Jenkins	3.1	5.2	11.9	42549
CRC32	2.3	0.8	5.7	11545

manner as their structures are similar. Because the Jenkins hash implementation has a better average cycle time and latency, it was chosen for further evaluation below. A comparison of the Boolean and NCL implementations of the Jenkins hash is shown in Figure 4.7. Under similar operating conditions and for similar throughput targets, the latencies of the NCL circuits are much smaller than those of the Boolean circuits. It is seen that for the same average cycle time of approximately 3.2 ns, the NCL implementation has a latency of just over 5.1 ns , while the synchronous latency is 25.6 ns, made up of eight pipeline stages of 3.2 ns each. The latency of the NCL implementation is thus almost a fifth that of the equivalent synchronous implementation ($\sim$5.4 ns vs $\sim$25.6 ns).

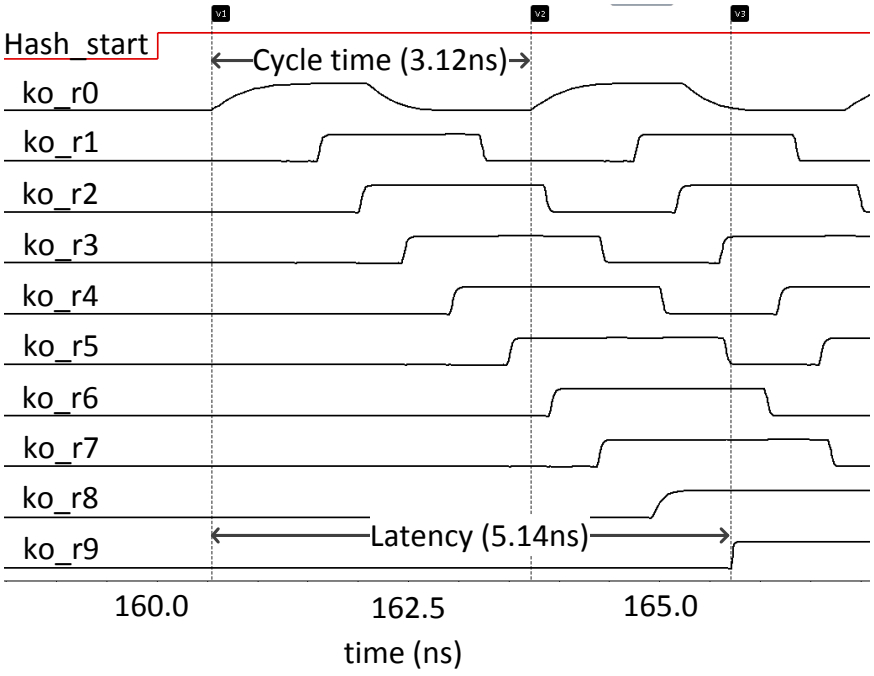
A comparison of the supply current drawn by the synchronous and NCL implementations is presented in Figure 4.8. It is interesting to note that the peak supply current for the synchronous design is almost 18 times the peak supply current of the NCL design. This is because in an NCL system different sections of the circuit are switching at different times, unlike a synchronous system in which all gates toggle in response to the clock edge. This is a significant advantage of the NCL approach and implies that an NCL design would not require the large decoupling capacitors that are typically needed in synchronous systems.

Table 4.1 then shows the average cycle times and supply current in NCL implementations at $27\,^{\circ}\mathrm{C}$ for the Jenkins and CRC32 hash implementations. The sizes of the two hash implementations measured as a multiple of the area of an NCL TH22 gate are also mentioned. It can be seen that the CRC32 has a 25% better cycle time, a 52% better peak supply current and almost a quarter of the area of a Jenkins hash implementation, suggesting that for the final implementation, using a CRC32 hash for computing Bloom filter indices may be advantageous.

The following sections present a discussion on the enhancements made to the threshold gate designs and completion detection circuits to improve the system performance.

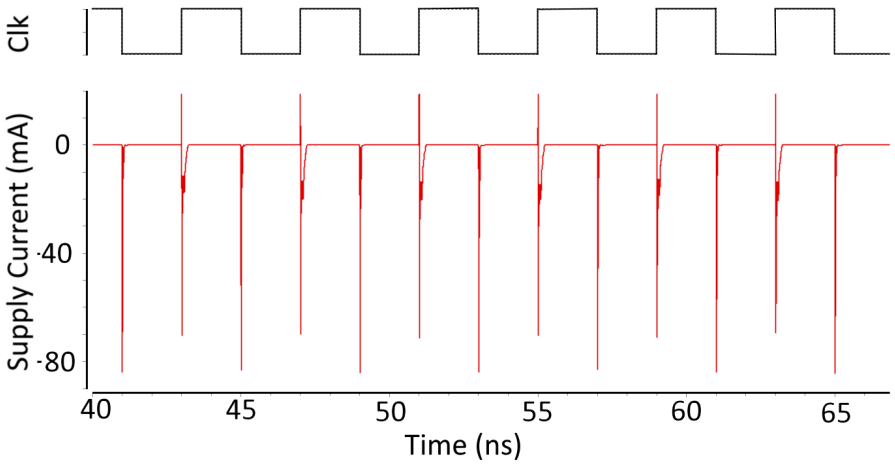


(a) synchronous implementation

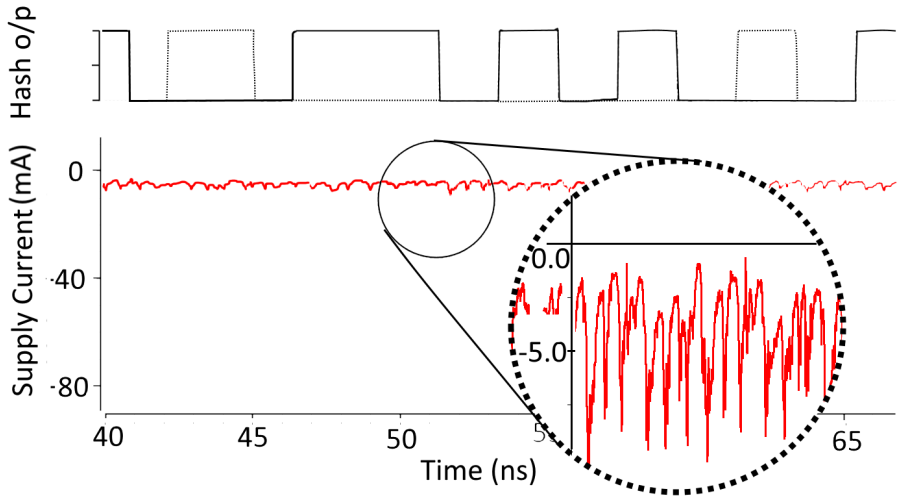


(b) asynchronous implementation

Figure 4.7: Jenkins hash - latency and cycle time variation at 27 °C for a set of test vector



(a) synchronous implementation



(b) asynchronous implementation

Figure 4.8: Jenkins hash - supply current drawn during hash computation

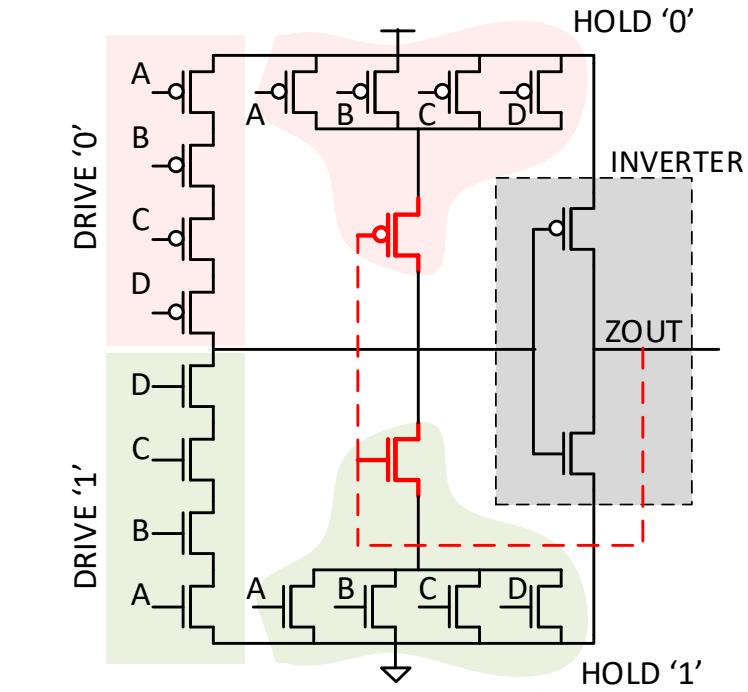
4.3 Completion Detection circuits - architecture

4.3.1 Conventional Completion Detection circuits with NCL gates (CoCD)

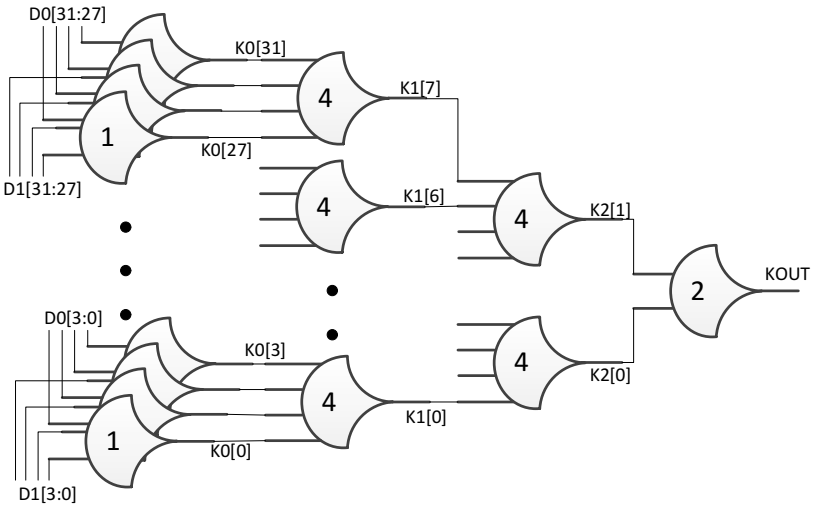
NCL systems essentially comprise multi-bit data paths made up of combinational functions and registers built using NCL gates. For the sustained flow of data through the pipeline, the inputs of the combinational logic block have to monotonically transition from the complete Null state to complete Data state and back. These transitions are known as wavefronts, and their flow from input to output is controlled by the acknowledge signals that flow between NCL registers in the reverse direction. An NCL pipeline register is a bank of TH22 gates spanning the data path that allows a Data or Null value at the input to flow out based on the Data or Null request received from downstream registers. This bank of gates is followed by a Completion Detection (CD) circuit. The CD circuit, as its name suggests, detects whether all the outputs are in the "complete Data" or "complete Null" state and denotes this through a single-rail acknowledge signal. The acknowledge signal is inverted and fed as a Null or Data request into the upstream registers. In this way, the registers enclose NCL combinational functions in a so-called *logical determination* boundary. For lightly pipelined systems with large combinational functions and narrow data paths, the CD circuit delays are insignificant. However, as the data path widths increase and systems are heavily pipelined to improve throughput, the delays of these circuits may be comparable to combinational path delays.

A conventional completion detection (CoCD) circuit comprises an array of TH12 gates followed by stages of cascaded TH44 and TH22 gates. The number of stages is determined by the width of the data path. For example, a CoCD circuit on a 32-bit data path requires a total of 32 TH12 gates (Fig. 4.9b) to generate 32 completion signals, followed by two stages comprising eight and two TH44 gates respectively that reduce it down to two signals. These two signals are then combined in the final TH22 gate to produce a single rail acknowledge signal.

Fig. 4.9a shows the schematic of an NCL TH44 gate used in the conventional completion detection circuit. The gate has five sections - Drive 1 (Data) Drive 0 (Null), Hold 1 (Data), Hold 0 (Null) and an inverter. The inverter stage is necessary to generate outputs in phase with the input, i.e. if the inputs are in the Null state, the output also need to be in the Null state, and if the 'threshold' number of inputs are in the Data state, the output should be in the Data state. In an NCL circuit, these in-phase outputs are necessary, when they have to be fanned out to



(a)



(b)

Figure 4.9: (a) Conventional TH44 gate schematic (b) Completion detection circuit using conventional THxx gates

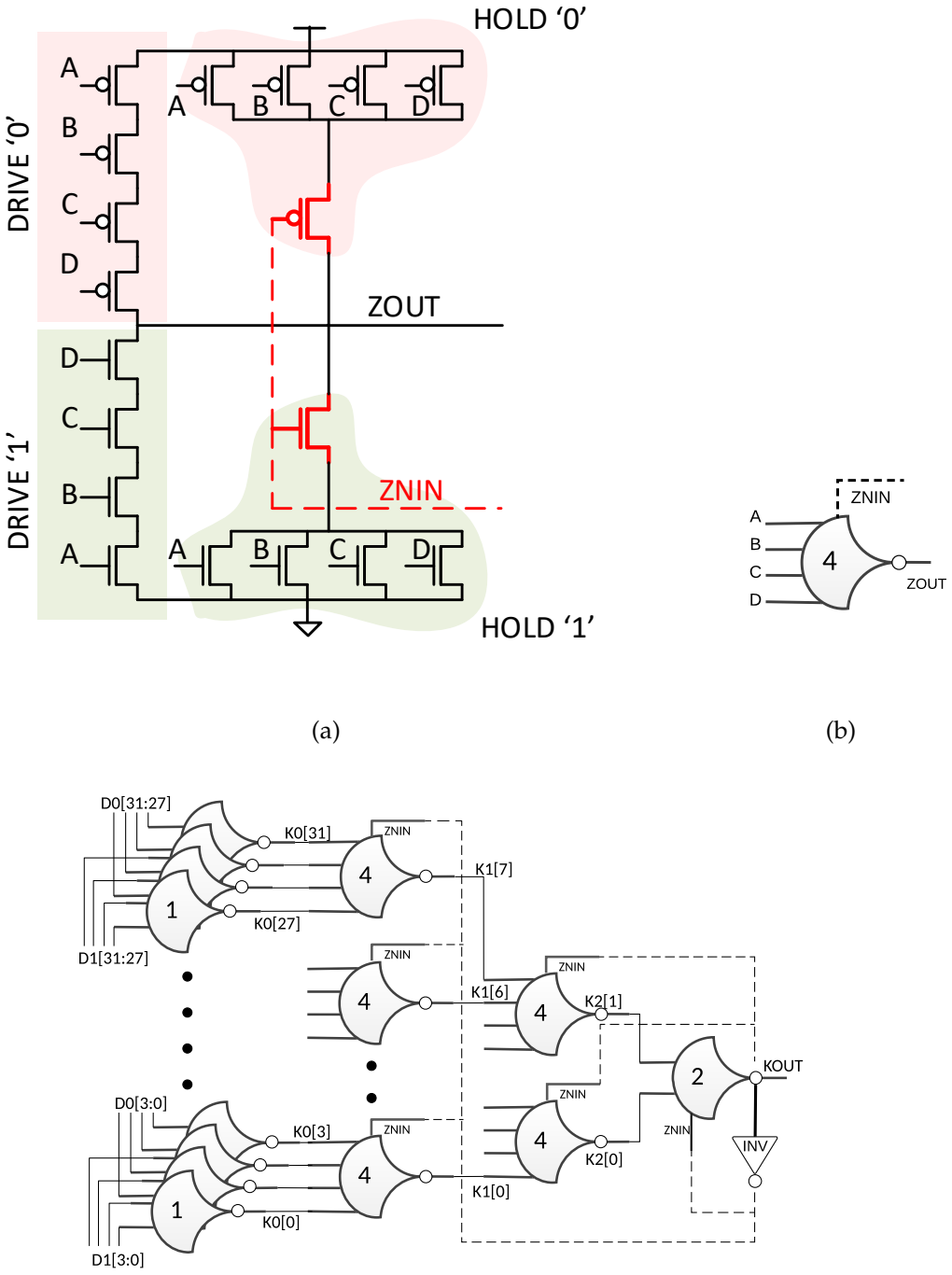
other combinational blocks. However, in a completion detection circuit, the outputs of the first stage are only being fed to the next stage, and the outputs of the next stage are fed to the third stage till a single rail acknowledge signal is generated. It is only at the output of the completion detection circuit that the output needs to be in phase with the input. If it is ensured that the completion detection circuit as a whole meets the "completeness of input" and "observability" criteria[38], then the individual NCL gates may be redesigned with outputs, not in phase with the inputs. This work presents two enhanced architectures for completion detection circuits for 32-bit dual-rail pipelines that explore this idea. The emphasis being on reducing the area, and energy consumption of the circuit, while maintaining throughput performance.

4.3.2 Completion Detection with complementary gates and external feedback (CD-CG)

The proposed completion detection circuit will make use of 'complementary' TH12, TH22 and TH44 gates to achieve the required functionality. These 'complementary' gates do not have the final inverter normally present in conventional THxx gates. As a result, they generate a Null output when the required number of inputs as specified by the threshold condition are in the Data state, and they generate a Data output when all the inputs transition to a Null. This modification to the gate operation, however, implies that the output of the gate cannot be fed back to the hysteresis transistor shown in red in Figure 4.9a. Instead, the hysteresis (state holding) behaviour for the TH44 gate with 'complementary' output (TH44Co) has to be achieved through an additional input (ZNIN) as shown in the schematic of Figure 4.10a. This input needs to be driven by a signal capable of maintaining the correct phase relation inside each gate. When referring to the outputs of these gates, we will refer to the in-phase output as conventional output and the out-of-phase output as complementary output. The gate symbol for the complementary gates is drawn with a bubble on its output to signify the inverted phase of the output and a separate ZNIN port as shown in Figure 4.10b.

The input and output waveforms of a complementary output TH44 gates are shown in Fig. 4.11. In the waveforms, a 'high' value indicates the Data state, while a 'low' value indicates the Null state and the behaviour of the gate may be explained as follows.

1. At time 't0', all inputs except B and the state holding input ZNIN are in the Null state. The output is therefore pulled down through the Hold '1' part of the circuit and remains at Null



(a) (b)

(c)

Figure 4.10: (a) Complementary TH44 gate schematic (b) Complementary TH44 gate symbol (c) Completion detection circuit using complementary and conventional THxx gates (CoCD)

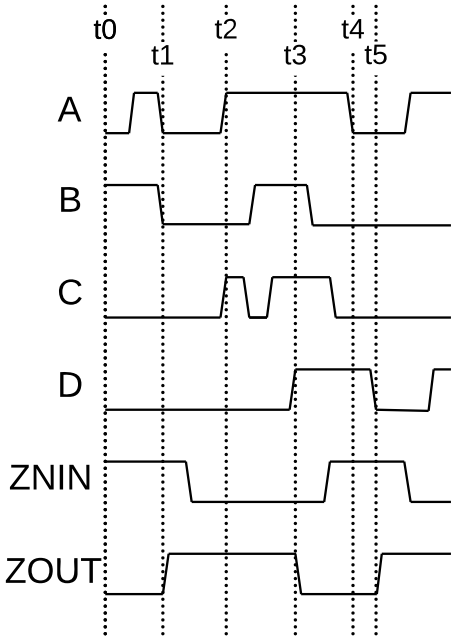


Figure 4.11: Input and output waveforms for the Complementary TH44 gate

2. At time 't1', all inputs are now in the Null state. And though ZNIN is still high, the Drive '0' section of the circuit pulls up the ZOUT line to a 'high' (Data) value.

3. At time 't2', two of the inputs have transitioned to the Data state. However, the output ZOUT remains in the Data state.

4. It is only at time 't3' when all inputs are in the Data state that the output transitions to the Null state.

5. At time 't4' the Data value on the ZNIN input maintains the output at Null because input D is still in the Data state. This is similar to the behaviour at time 't0'.

6. Finally at time 't5', when all inputs have again transitioned to the Null state, the output goes up to the Data state.

Moreira et al. [110] recently proposed an NCL+ gate, which is similar to the complementary gate idea described here. The objective of NCL+ gates is to simplify NCL design akin to what is being attempted here and to allow conventional synthesis tools to be able to handle NCL behavioural code.

The circuit diagram of completion detection with complementary output gates is similar to the conventional completion detection circuit, except for the external feedback paths through the

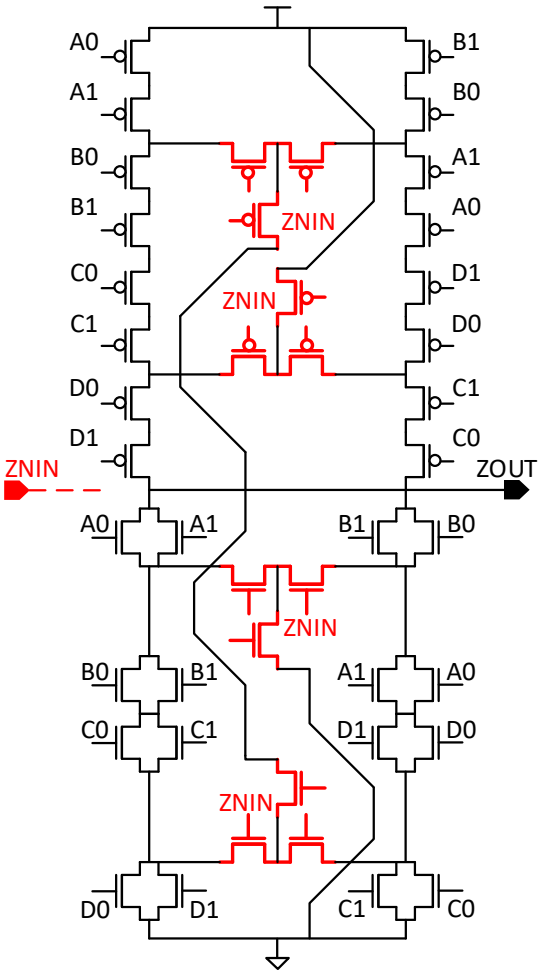
circuit to drive the correct phases into the hysteresis transistors. Thus a Data-complete at the input of the CD-CG circuit generates the following signals in the downstream stages: a Null-complete at the TH12 stage, a Data-complete at the first TH44 stage, a Null-complete at the second Th44 stage and finally a Data-complete at the last TH22 stage. The final TH22 gate needs to generate both the inverted and non-inverted value of the output. The complementary output is fed back to the second TH44 stage and is also the final Completion Detection signal. The regular output is fed back to the first TH44 stage. For a data path width N, this integrated CD circuit has $log_2(N) - 1$ fewer inverter stages, potentially resulting in a smaller area and higher throughput performance. The completion detection circuit using complementary output gates is shown in Figure. 4.10c

4.3.3 Completion Detection with Complementary Smith-gates and external feedback (CD-CSG)

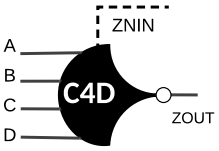
In [111], Parsan and Smith proposed a static gate architecture that merges the "Drive" and "Hold" sections of a gate into a single circuit to improve the delay characteristic and area. Gates that follow this architecture shall be referred to as *Smith-gates* in the rest of this work. A somewhat similar idea of merging sections of completion detection circuits for QDI PCHB designs was proposed by Ho et al. [112]. In the original design of [111], the switching impedances for the various inputs in a TH44 gate are not uniform, and therefore the delay performance depends on the order in which the inputs transition to the all-Data or all-Null state. Additionally, no attempt was made to balance the propagation delays in the rise and fall directions. The modification proposed here borrows the idea of the merged drive and hold circuits, but redesigns the two arms of the pull-up and pull-down networks, so that the impedances in each of the arms is uniform. The inverter stages have also been removed, and finally, four TH12 and one TH44 have been merged into a single structure to generate a completion signal spanning 4 dual-rail signals (THC4D). These modified Smith gates all have a complementary output and a ZNIN port to feed the correct phase of the completion signal to the hysteresis transistors. Fig. 4.12a shows the schematic of the THC4D gate and Fig. 4.12c shows the completion detection circuit built using the complementary Smith gates.

4.4 Completion detection circuits - performance evaluation

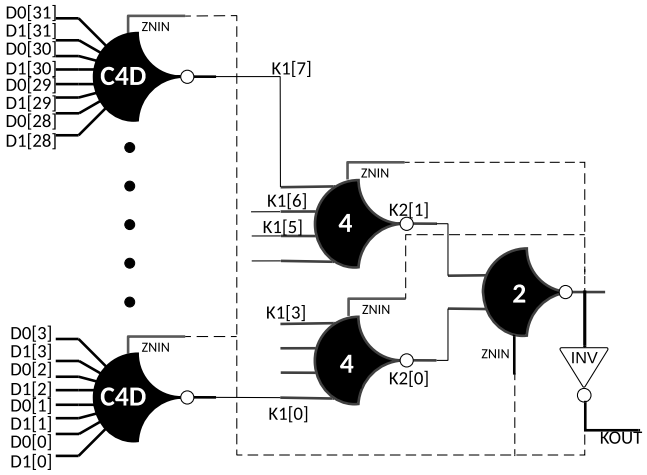
The performance of the proposed completion detection circuits is evaluated in Cadence Virtuoso® using the same 1V, 28nm ultra-thin body and BOX silicon-on-insulator (UTBB-SOI) process kit



(a)



(b)



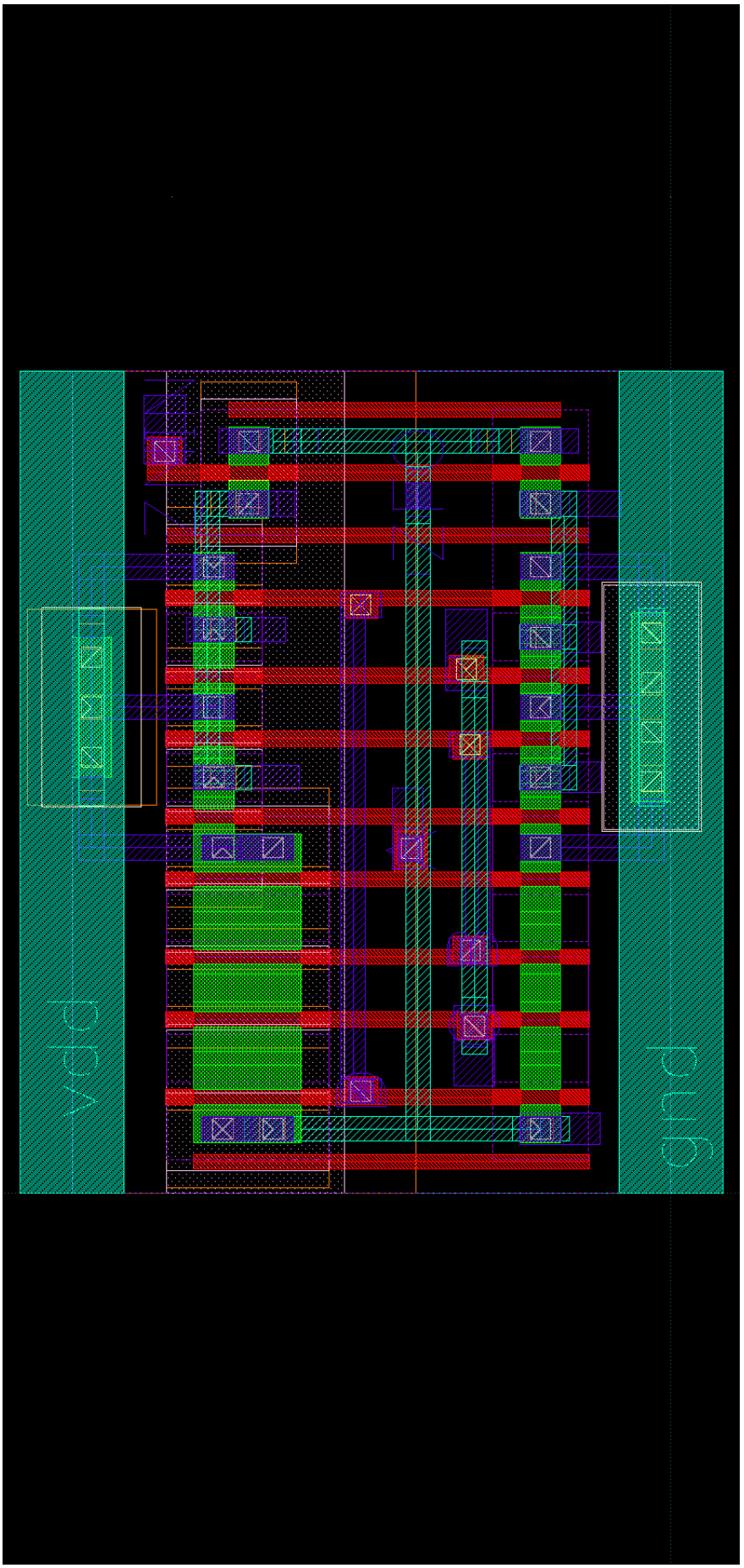
(c)

Figure 4.12: (a) THC4D gate schematic (b) THC4D gate symbol (c) Completion detection circuit using complementary Smith gates (CD-CSG)

from ST Microelectronics mentioned previously. Schematics for all three gate styles were created, and the gates were then assembled into the completion detection circuits so that latency and throughput performance could be evaluated. Note that the CD-CSG gate has a stack of 8 PMOS transistors. Such a tall stack while prohibitive in bulk CMOS technology is possible in UTBB-FDSOI as the substrate bias effect that prevents it in bulk CMOS does not exist in fully depleted SOI. The average propagation delays were estimated using Cadence UltraSim® simulations at $27\,^{\circ}\mathrm{C}$. Simulations were also performed at $0\,^{\circ}\mathrm{C}$ and $75\,^{\circ}\mathrm{C}$ and over a range of voltages between 0.7V and 1.2V. The proposed completion detection circuits were observed to operate correctly over the full voltage and temperature ranges outlined above. Each of the individual gates was also laid out using Cadence Virtuoso Layout-XL® to create standard cells. The layouts of the TH44 complementary gate and the THC4D gate are shown in Figures 4.13a and 4.14a. The total area of the complete circuit was then obtained by running a trial *Place and Route* through Cadence Encounter Digital Implementation® (EDI) using these standard cells. Tool (license) limitations did not allow extraction and back-annotation of electrical characteristics of the EDI layouts into circuit simulations.

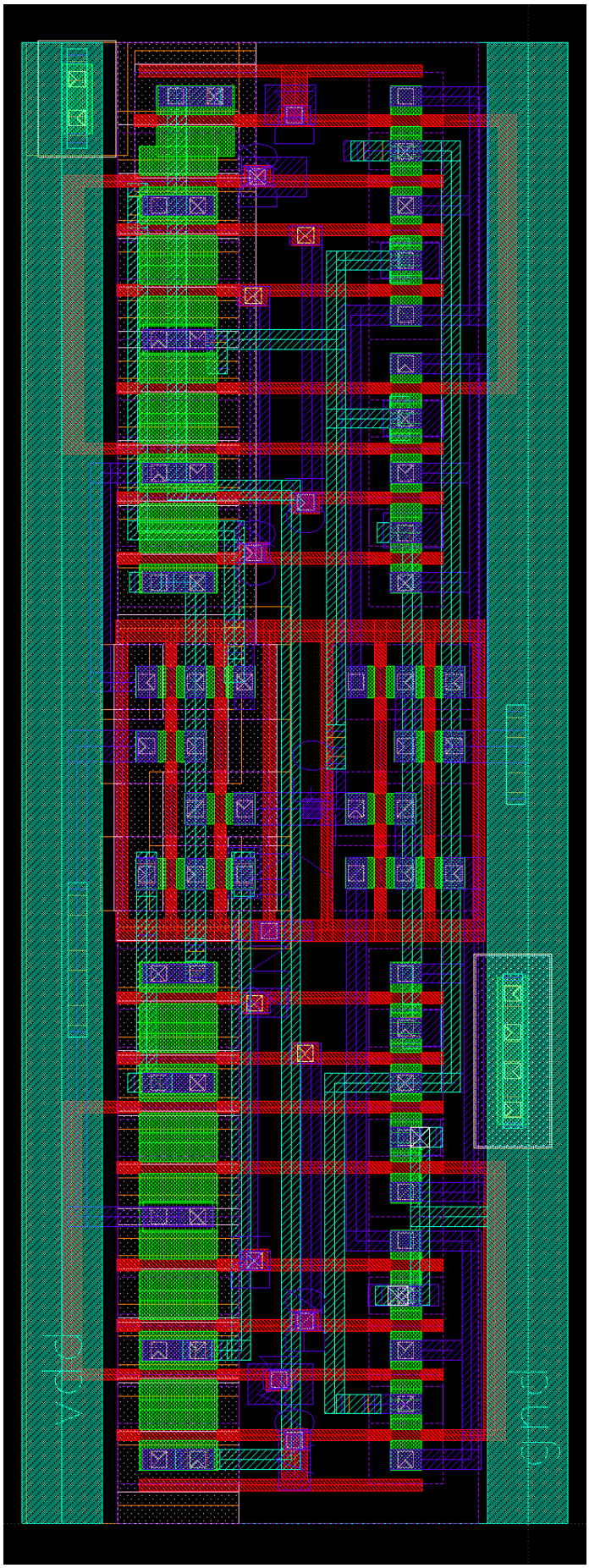
A target propagation delay of approximately 240 ps was set and the transistor sizes (width/length) adjusted to achieve the target for both the Null-Data and Data-Null transitions to achieve a fair comparison between the three styles. Energy per operation and peak supply current measurements, as well as actual propagation delays for all three circuits, were obtained through circuit level simulation. The comparison results are presented in Table 4.2. The percentage improvement for CD-CDG and CD-CSG over the original CoCD circuit have been specified in brackets next to the absolute values. It can be seen that both the CD-CG and CD-CSG circuits occupy a smaller area than the original CoCD circuit and also have a lower energy consumption per operation. The area of the CD-CSG circuit is smaller than the CoCD circuit even though the number of transistors is higher, because of the careful sizing of the transistors in the THC4D gate. It may also be noticed from the circuit diagrams and the layouts that the circuits of the individual CD-CG gates are simpler and smaller than those of the CD-CSG gates. The CD-CSG gates are much more complex, and the presence of the ZNIN-fed hysteresis gates between the two arms make them harder to lay out within the strict height restrictions of standard cells. However, the energy-delay product of the CD-CG circuits is higher than the CD-CSG circuit.

An interesting characteristic of the proposed circuit styles is the peak current drawn by these circuits. In a conventional completion detection circuit, because the outputs of all the gates



(a)

(b) Layout of the TH44 complementary gate created using Cadence Layout-XL®



(a)

(b) Layout of the THC4D gate created using Cadence Layout-XL®

Table 4.2: Comparison of proposed completion detection circuit architectures against conventional architecture with transistors sized for a fixed delay value of ∼230 ps

	CoCD	CD-CG	CD-CSG
Delay (ps)	230	237 (-3.04)	221 (3.91)
Transistor Count	404	320 (20.7)	420 (-3.96)
Area (μm^2)	67.2	50.4 (25.0)	46.8 (30.3)
Energy per operation (fJ)	200	130 (35.0)	100 (50.0)
Peak supply current (mA)	2.15	0.58 (73.0)	0.54 (74.8)

are in phase, the dynamic current waveform exhibits significant peaks. On the other hand in the CD-CG and CD-CSG circuits, the outputs are allowed to transition out of phase from the inputs, and therefore the peak currents are much smaller. Fig. 4.15 shows the supply current waveform for one Null-Data-Null cycle of the input for all three CD circuit designs. It is observed that the peak supply current drawn in CD-CG and CD-CSG circuits is ∼75% lower than that measured in conventional completion detection circuits. This results in significant savings (∼35% and ∼50% respectively) per completion detection operation in both the CD-CG and CD-CSG circuits. Fig. 4.15 illustrates the improvement observed in CD-CG and CD-CSG circuits over the conventional design with respect to peak current consumption.

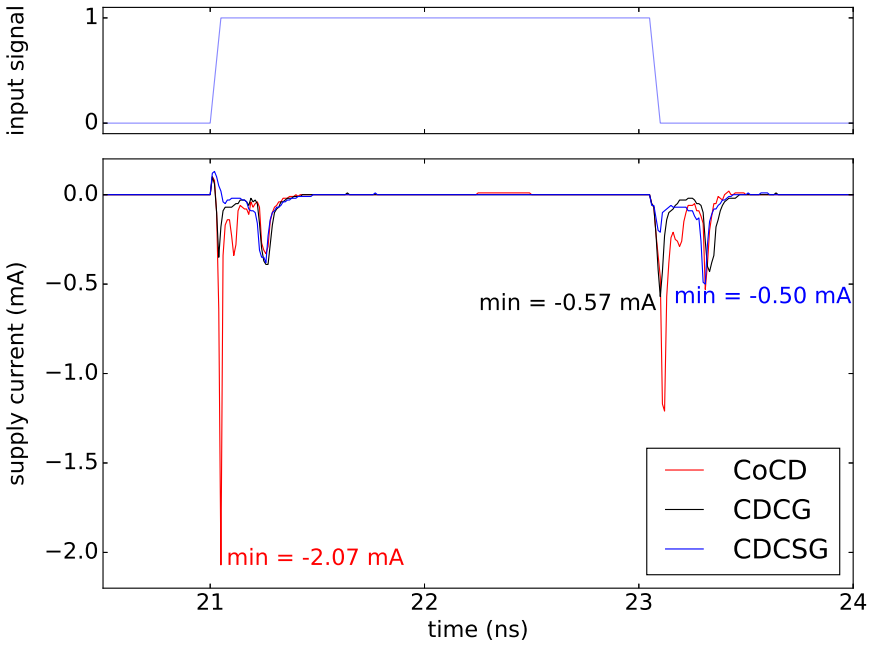


Figure 4.15: Supply current drawn by the three completion detection circuits during the Data-to-Null and Null-to-Data transitions at the input

Summary

This chapter has discussed the background work and infrastructure development that was undertaken before tackling the main problem of packet lookup in IP routers. Three hashing algorithms were analysed for their suitability to NCL implementations. The appropriateness of the hash functions themselves in terms of their uniform distribution and avalanche behaviour was not evaluated as part of this work, and instead, the existing literature was relied upon.

A comparison of the NCL implementations of Jenkins and Murmur hash demonstrated how cycle times vary over a range of values with changing inputs. The experiments also demonstrated that NCL demonstrates a robustness in the face of Process and Temperature variations and that NCL circuits work correctly without requiring any additional timing constraints. A comparison of Boolean and NCL implementations of the Jenkins hash algorithm then showed that for a given throughput, the latency of the NCL implementation is almost 1/5th that of the equivalent synchronous implementation (5.4ns vs 25.6ns). Finally, it was seen that a CRC32 implementation was better than a Jenkins or Murmur hash implementation in terms of the cycle time, peak supply current as well as area.

Experiments undertaken on hashing algorithm implementations also led to the identification of one of the primary bottlenecks in NCL data paths - completion detection circuits. Two optimised completion detection circuits, Completion Detection with Complementary gates (CD-CG) and Completion Detection with Complementary Smith Gates (CD-CSG) were proposed. The performance of these circuits was evaluated, and it was observed that both the CD-CG and CD-CSG circuits demonstrated an improvement in area (25% and 30%) and energy/operation (35% and 50%) over conventional completion detection circuits. The Complementary Smith Gates, however, have a complex circuit as well as layout. It is concluded that the CD-CG circuits with their right balance of energy saving and design and layout complexity are appropriate for use in complex circuits needed for the address lookup operations.

> Absorb what is useful, reject what is useless, but most importantly add what is specifically your own.
>
> Bruce Lee

Chapter 5

NCL SRAM with early completion detection and Null-storage column

An important class of destination address lookup algorithms encompasses those based on a trie or tree data structure. These algorithms rely implicitly on the availability of numerous and fast SRAMs to achieve throughput comparable to alternative approaches based on costly, power-hungry but fast TCAMs. While these SRAMs have traditionally been external to the chip performing the actual computation, it is possible to place most of the packet forwarding information for small, medium and even for some large routing tables within on-chip memory. Only the actual packet data is so large it still needs to be stored off-chip in external SRAM.

In this chapter, a static RAM organisation is proposed and analysed that works in a manner similar to other NCL circuits, in that it executes the read or write operation only when all the inputs signals are in the complete Data state, and it holds the output in the Data state, till all inputs transition to the Null state. The proposed RAM can, therefore, be instantiated wherever a storage element is required. It also does not require any complex interfaces to transform NCL to Boolean logic and vice versa. The addition of the early completion detection and the special Null-storage column results in a small cycle time and low energy consumption when address locations containing Null values are accessed.

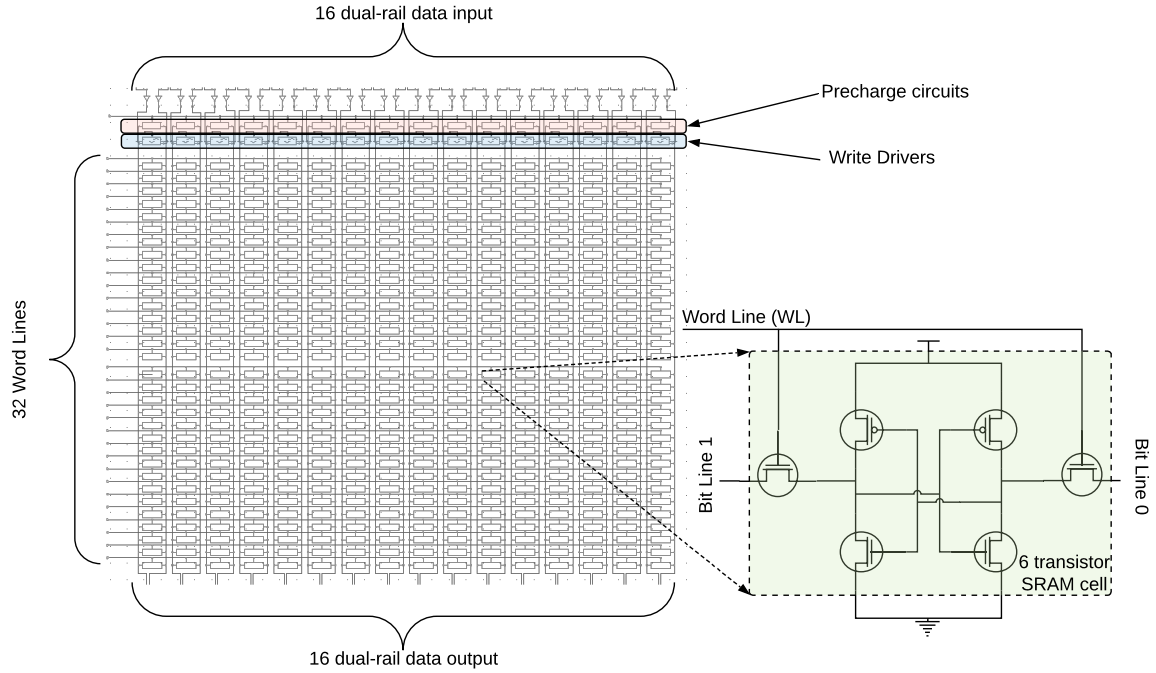


Figure 5.1: Circuit diagram of the 6-Transistor SRAM cell and organisation of the SRAM cell array

5.1 NCL SRAM32x16 unit

The NCL compatible RAM cell is based on the conventional 6-transistor (6T) cell widely used in existing SRAMs, comprising the 6T cross-coupled storage elements and bit-line access transistors enabled by the decoded word line (WL) signal. These cells are organised into fundamental units of 32 words x 16 bits (SRAM32x16), which include precharge and bit-line driver circuits, out of which larger more complex SRAM structures can be built. The 6-transistor SRAM cell and the SRAM cell array is shown in Figure 5.1. The number of 6T cells required in an NCL SRAM is the same as that required in conventional Boolean SRAMs. This is because the complementary bit lines in the 6T cell serve as the 'zero' and 'one' rails of an NCL variable.

Two important enhancements are proposed to this conventional SRAM structure:

1. Read and Write completion detection;
2. a Null-storage column.

Both these are described in detail in the following sub-sections.

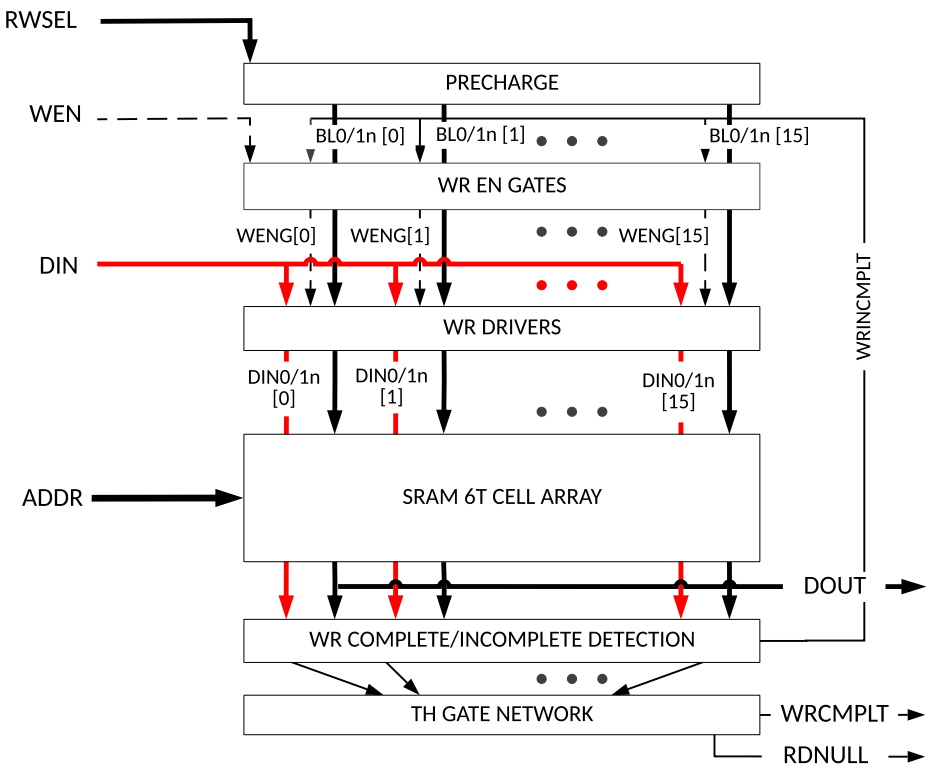


Figure 5.2: Block diagram of the 32 words x 16 bits SRAM unit

5.1.1 Read and Write completion detection

An important requirement for an NCL RAM is the ability to handle requests for Null and Data wave-fronts from downstream elements and generate the appropriate acknowledge signals towards the upstream elements in the pipeline. The SRAM must, therefore, be capable of detecting when the read and write cycles are completed. In this work, instead of using a combined read/write controller for each bit, as for example in [90], the read and write circuits have been separated. This is based on the observation that during a write operation the cycle ends when the data is written into the SRAM cell so that write completion can be detected at the cell or unit level. On the other hand, a read operation cycle is complete only when data is read from the cell and, if necessary, multiplexed with the output data from other SRAM32x16 units in the bank. Thus the read completion circuit for an SRAM bank has to encompass the whole depth and width of the bank and not just the individual SRAM32x16 units and is therefore external to the units. In fact, the read completion circuit is similar to the regular completion detection circuits that are discussed in Chapter 4. For this reason, the write and read completion detection systems can be developed and optimised separately.

An architectural block diagram of an SRAM32x16 unit with the write completion detection circuit is shown in Fig. 5.2. If x denotes the bit number in the word, DIN0[x] and DIN1[x] are the 'zero' and 'one' rail of one bit of the input word. These input rails are passed through inverters to generate signals DIN0n[x] and DIN1n[x] that are fed to the write driver block. While there are many possible designs for the write driver, the circuit proposed by Dama et al. [89] was found suitable for the proposed design. BL0n[x] and BL1n[x] are the internal 'zero' and 'one' bit lines for column *'n'*. When neither a read or write operation is active, these lines are pulled up by the precharge circuit and thus are active low. During an SRAM access, these bit lines may be driven either by the write driver during a write cycle or by the SRAM cell array during a read cycle. The BL(0/1)n[x] are therefore also the outputs of the SRAM unit. The SRAM32x16 units are generally not used in isolation, but a number of these units are combined to build wider and deeper SRAM blocks. It is at the output of these combined structures that the active low data outputs are merged and the levels inverted to enable the downstream NCL blocks to receive the correct logical values.

The write completion detection circuits, shown in Fig. 5.3 are placed one per column within the SRAM cell array. These circuits operate on the active low BL0/1n and DIN0/1n signals and produce two active high outputs WRCMPLT and WRINCMPLT. Table 5.1 gives the truth table for the write completion detection circuit. It can be seen that the WRCMPLT signal is not simply the inverse of the WRINCMPLT signal, and both the signals are necessary for the correct operation of the SRAM. The situation when both BL1n and BL0n are 'zero' is considered illegal, whereas when both these signals are 'one' means the write data is not available and therefore WRINCMPLT and WRCMPLT are both 'zero'.

The case where either of DIN0n and DIN1n is low indicates that the SRAM is in its write cycle. In the write cycle, the WRINCMPLT signal goes high when either DIN0n $\neq$ BL0n or DIN1n $\neq$ BL1n, while the WRCMPLT signal high when both DIN0n = BL0n and DIN1n = BL1n. A careful analysis of the circuit shows that there will be periods when the BL0n and BL1n are transitioning from their old values to their new values when both WRCMPLT and WRINCMPLT may be high. However, this period is extremely short and once the bit lines transition to the new value, the WRINCMPLT signal would go low. The WRCMPLT outputs of the write completion detection circuit for all columns are combined using a bank of THxx gates (e.g., TH12, TH14) to generate the final write complete signal for the SRAM32x16 unit.

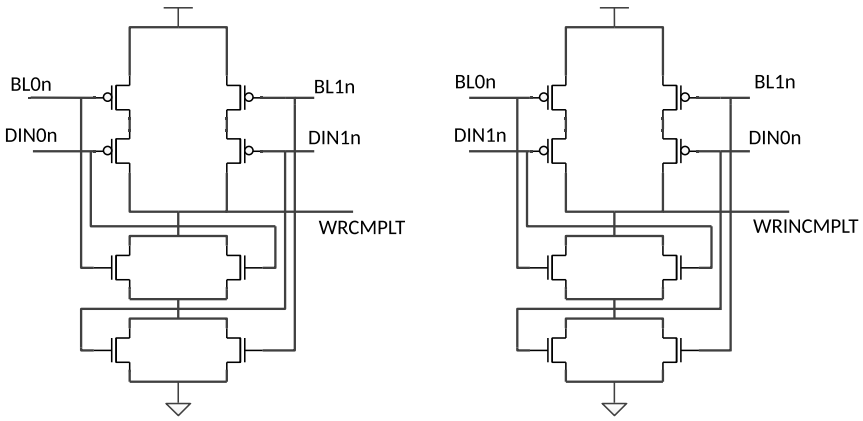


Figure 5.3: Schematic of the circuits generating the WRCMPLT and WRINCMPLT signals

Table 5.1: Truth table for write completion detection circuit

Inputs				Outputs		Comments
DIN0n	BL0n	DIN1n	BL1n	WRCMPLT	WRINCMPLT	
0	0	1	1	1	0	Write is complete
1	1	0	0	1	0	Write is complete
0	1	1	0	0	1	Write initiated.
1	0	0	1	0	1	Write initiated.
0	x	0	0	-	-	Illegal inputs
1	x	1	x	0	0	write inputs unavailable. WRCMPLT and WRINCMPLT at Null
Any other input combination				inputs transitioning. Write in progress		

As mentioned above, the read completion detection circuit is not included in the SRAM32x16 unit. The read completion signal is generated after the outputs of all the individual units in a bank are multiplexed and inverted.

5.1.2 SRAM unit with NULL storage

The second proposed enhancement to the SRAM is the addition of a Null storage column that stores a single bit Null flag (WNULL) per word indicating whether the word contains a Null value (WNULL = 1) or a valid Data value (WNULL = 0). This idea was somewhat inspired by the cache 'dirty-bit' concept, which flags whether the corresponding block of memory has

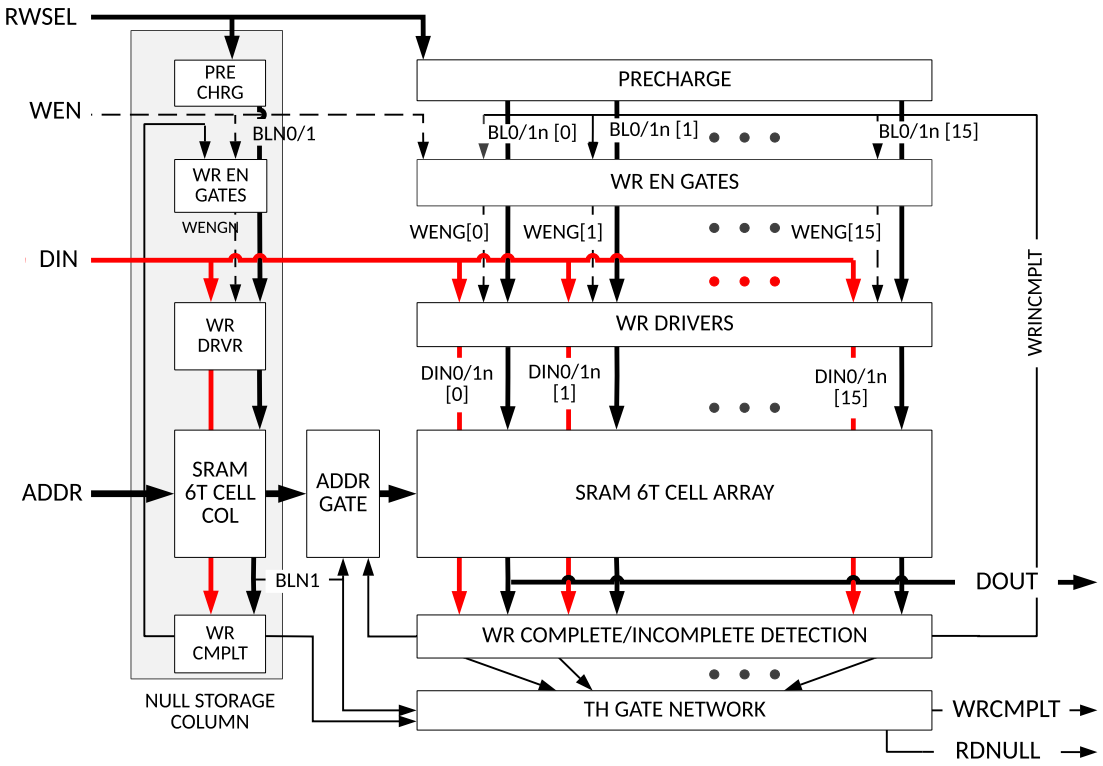


Figure 5.4: Architecture block diagram of the SRAM unit (32 words x 16 bits) with a Null storage column

been modified and therefore requires further action before replacement. Although it imposes a small overhead on the memory system, the operation of this Null flag may result in shorter, more energy efficient read and write cycles in certain scenarios and can result in an overall reduction of the latency and cycle times in the address lookup function.

The architectural block diagram of the SRAM32x16 unit with the Null storage column is shown in Figure 5.4. DNIN is the input port of the Null storage column and DNIN0 and DNIN1 are the 'zero' and 'one' input rails of this input variable. A high value on DNIN0 indicates that the 'Word is not Null' (WNULL = 0), while a high value on DNIN1 indicates a 'Word is Null' condition (WNULL = 1). DNIN0n and DNIN1n are the inverted versions of DIN0 and DNIN1 and are the inputs to the Null column write driver block. The outputs of the write driver block are the inverted bit lines BLN0n and BLN1n connected to the 6-transistor storage elements in the Null storage column. When the WL input for an address in the SRAM unit is asserted, the corresponding WNULL bit is checked first and used to gate the WL signal to the SRAM cell array. This ensures that during the read and write operations, unnecessary accesses to the main array are avoided, thereby reducing the energy consumption of those operations.

An essential difference between the SRAM cell array and the SRAM Null storage column is the drive strength of the 6T cells and the write driver. Because the output of the Null column is used to gate the address line towards the 6T cell array and the write drivers, the transistors need to have a higher drive strength. However, it is only the BLN0n bit line that has the additional load and so additional buffers are needed along the BLN0n line to be able to drive the long line of the 'address gate' circuitry. In contrast, the BLN1n line only has the read completion detection circuit and the external read data multiplexer as its load, so it does not require these additional buffer stages.

In SRAM32x16 units with Null storage column, if a Data value is being written (WNULL = 0), the write completion signal is generated by combining the WRCMPLT signals generated per column as discussed in section 5.1.1. On the other hand, if a Null value is being written (WNULL = 1), only the WRCMPLT signal of the Null storage column is used to generate the WRCMPLT for the entire unit. Similarly, the read completion is detected either on the whole data word if WNULL = 0 or only on the output of the Null storage column if WNULL = 1. A detailed description of the read/write operation of the SRAM32x16 unit and the behaviour of its various signals follows next.

5.1.3 SRAM32x16 unit Write and Read operation

The general read/write operation of an SRAM cell array can be explained as follows. In the idle state, BL0n and BL1n lines are pulled high by a precharge circuit controlled by the read/write select (RWSEL) signal of the SRAM32x16 unit. When an SRAM location is to be read, RWSEL goes high, and the precharge is disabled, allowing the bit lines to float. A specific address in the SRAM unit is accessed by enabling the appropriate WL. Depending on the value stored in the 6T cell, one of the bit lines will be driven low, while the other stays high. Conventional synchronous SRAMs use sense amplifiers to detect the difference between these bit lines and drive a single bit output. However, in the asynchronous SRAM for dual-rail systems, correct voltage levels are required on both the bit lines and so sense amplifiers are not used [89]. During a write operation, a strong SRAM write driver is enabled, which drives the bit lines to the required state. This value gets latched in the SRAM cell array when the Word Line goes low. The SRAM32x16 unit with Read and Write Completion Detection and the Null-storage column operates as a conventional SRAM cell array with some additional circuitry that controls how the read and write completion signals and the data outputs are generated in specific cases.

5.1.3.1 Write operation

Starting in the idle state with all inputs Null, both the WRCMPLT and WRINCMPLT signals will also be NULL. The write cycle starts with the data to be written being asserted on DNIN0n, DNIN1n, DIN0n and DIN1n along with RWSEL, WEN and the correct WL. The RWSEL assertion disables the precharge circuit while the WL signal enables the pass transistor for a specific word in the SRAM cell array.

Write Operation

When a Data value is to be written to the SRAM, the {DNIN1, DNIN0} lines have a value {0, 1}, while the 16 data bit lines are driven to the correct Data values.

1. *Address to be written currently has WNULL = 0*

   When the WL corresponding to this address is driven high, the pass transistors of the SRAM cell are enabled, and the stored bit values flow out on the BL0n and BL1n lines. Now there are two possible scenarios for each bit in the word - either the bit value being written {DIN0n, DIN1n} is different from the value already present in the storage element {BL0n, BL1n}, or the values are the same. If the DIN1n and DIN0n values for a bit are different from the value of the corresponding BL0n and BL1n lines, the WRINCMPLT signal for that bit will be asserted. This *per-bit* WRINCMPLT signal plus the WEN input of the SRAM unit are fed to the bank of TH22 gates which generate WR_EN. The output of the WR_EN gate is the write enable signal (WEN_G[x]), for each column x in the SRAM array (SEL input of the write driver). When both WEN and WRINCMPLT[x] are high, the WEN_G[x] goes high and stays high till both inputs are de-asserted. Once de-asserted, the WEN_G[x] does not go high if the WEN goes high, but the WRINCMPLT remains low. The presence of the TH22 gate, therefore, prevents the write driver from being enabled immediately on the assertion of the external WEN input, but only if the data being written is different from the data already present in the SRAM word. Once enabled, the write driver will drive the BL1n[x] and the BL0n[x] lines to the same value as the DIN1n[x] and DIN0n[x] causing the WRCMPLT signal to go high and the WRINCMPLT signal to go low. The WRCMPLT signal is used to indicate when the write operation is complete. In the case when the DIN0n,DIN1n value is equal to the {BL0n,BL1n} value for any column, the WRCMPLT signal will be generated immediately, without the write driver being enabled. Thus, in this case, the write operation is shortened, and the completion detection signal is generated earlier than would have been otherwise.

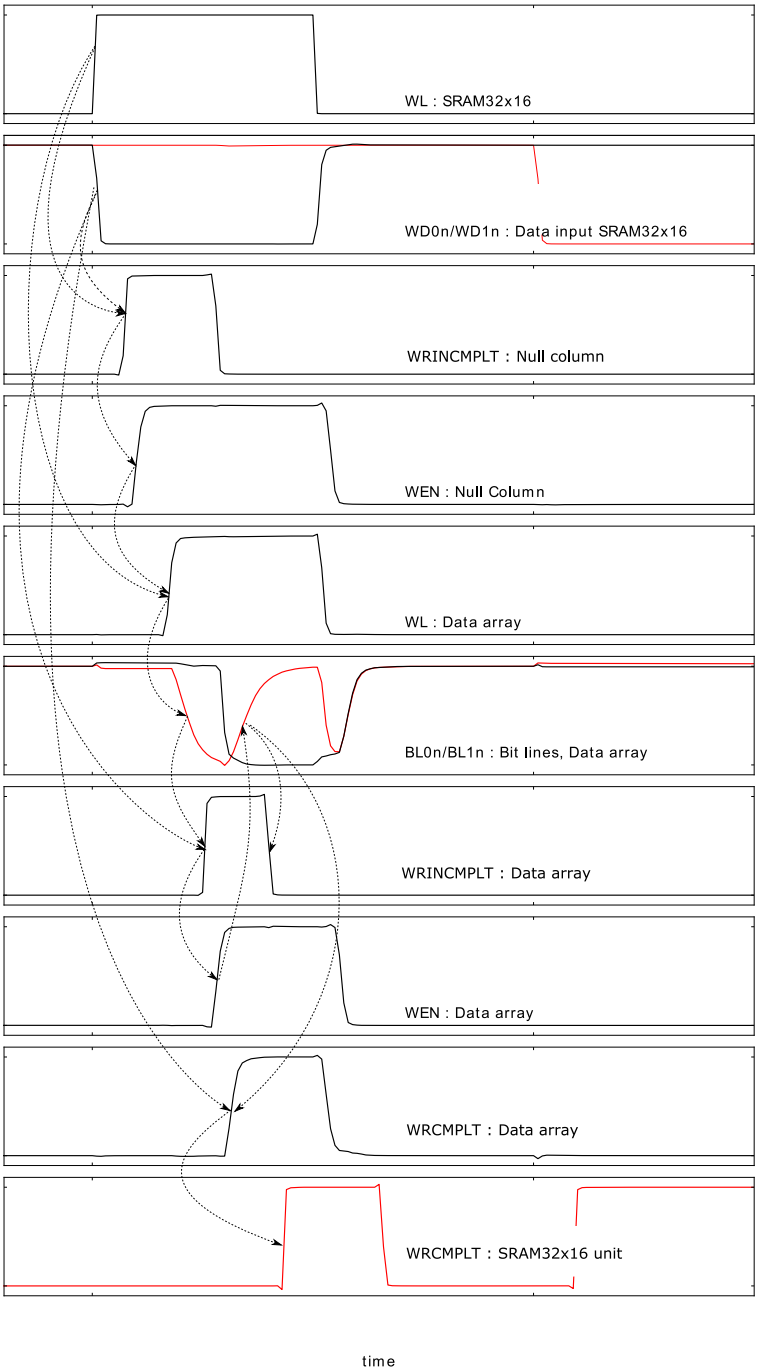


Figure 5.5: Waveforms showing the order in which the data and control signals in the SRAM32x16 unit toggle when a Data value is being written to a location that initially contained a Null value

2. *Address to be written has WNULL = 1*

   In this case, enabling the WL input of the SRAM32x16 unit allows the Null storage column to be written with the new WNULL = 0 value, but the WL to the SRAM cell array now remains disabled until the write to the Null storage column completes. Once this value is written, the address line is enabled, and the write cycle proceeds as in the previous case with WNULL = 0. The waveform in Fig. 5.5 shows this behaviour. A Data value on the WL and Write data lines at the input of the SRAM32x16 unit triggers the WRINCMPLT signal of the Null column high, which enables the write driver for the Null column. Once the Null column is written with a non-Null value, the Word Line for the SRAM array is asserted. In this specific case, the bit value being written is different from the bit value already stored, which causes the WRINCMPLT signal for the SRAM array bit to go high, enabling the write driver for the column. When the data is correctly written i.e., the bit lines have the same value as the write data lines, the WRCMPLT signal for the column will go high, ultimately triggering the write complete signal for the SRAM32x16 unit.

Write Null value

Writing a Null value to a memory location involves changing the state of the WNULL bit in the Null storage column word from WNULL = 0 to WNULL = 1. To achieve this, the {DNIN1, DNIN0} lines are driven with the value 1,0. The other 16 data lines are kept in the Null state i.e., {DIN1[x],DNIN0[x]} = {0, 0}. The corresponding WL signal is asserted, and the Null state bit is stored in the Null storage column cell. Because the 16 data bits are Null, no write operation is performed on the SRAM cell array. The write cycle can, therefore, be 'short-circuited' and marked complete by the write-complete detection on the Null column alone, without considering the state of the WRCMPLT signal for other bits in the word. This results in shorter cycle times as well as savings in energy.

5.1.3.2 Read Operation

During a read cycle the Word Line for the address to be read is asserted and the value of the WNULL bit is available. Depending on this value, there are two possible scenarios:

Read Data

If the address being read has WNULL = 0, then the WL towards the SRAM cell array is asserted,

which causes the pass transistors to open, and the stored value flows out on the BL0n and BL1n lines as in the case of a conventional SRAM. This data flows through the multiplexing and inversion logic and then to the regular completion detection circuit, and the read cycle is completed.

Read Null

If the data being accessed has WNULL = 1 in the Null storage column, the WL towards the SRAM arrays remains de-asserted, none of the rows in the SRAM cell array are enabled, the BL0n and BL1n lines remain in the Null state (high, because they have an inverted sense) and the output of the SRAM32x16 unit remains at Null. The RAM unit, however, still needs to indicate that the read cycle is complete. This is achieved by routing the read complete of the Null storage column alone instead of the completion detection output of the wide data path to the upstream nodes. Because this is a faster path, the read cycle completes sooner when a Null value is read.

5.2 1024x16 SRAM with address decoder, read-write completion and Null-storage column

The SRAM32x16 unit designed in the previous section can then be used to build larger SRAM designs, such as the 1kx16 block that is examined here. The SRAM1kx16 block includes the required pipeline stages, plus an address decoder to decode dual-rail address into individual WL signals that are input to the SRAM32x16 blocks. The additional circuitry required to combine the read and write completion signals is also a part of the SRAM1kx16 block. All these details have been shown in the architectural block diagram of Figure 5.6.

The incoming dual rail address ADDR 0/1 [9:0] bus is first registered to ensure that all bits of the address are in the Data state before the address is decoded. The lower five bits of the address are decoded into 32 single rail address lines AD0-AD31, while the upper five bits are decoded into 32 UNIT_SEL rails. The set of addresses AD0-AD31 are fanned-out into 32 address latches each enabled by one UNIT_SEL line, generating a total of 1024 single rail word select lines AIN0–AIN1023 for the SRAM1024x16 bank. The address is completely decoded instead of fanning out the AD0-AD31 lines directly to the SRAM32x16 units to avoid unnecessarily enabling the WL in all SRAM32x16 units, which would lead to an increase in energy

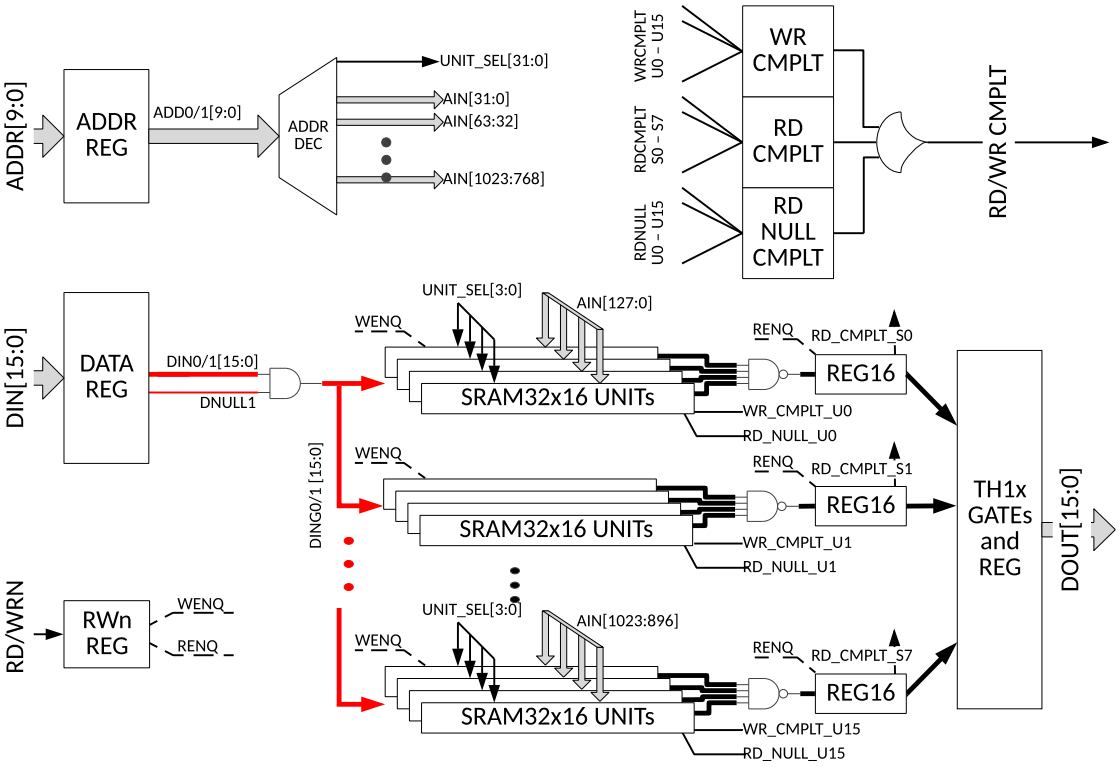


Figure 5.6: Block diagram of the 1024x16 bit NCL RAM

consumption. For a given address, only one UNIT_SEL rail will go high, and this is used to enable the corresponding SRAM32x16 unit. The UNIT_SEL signal is also used to suppress the DIN inputs of the SRAM32x16 unit that are not being addressed, which helps save energy during write operations. The DIN inputs are also disabled if the DNULLIN input bit is true further increasing the energy savings.

On the read side, the output data from each of the four SRAM32x16 units are combined using banks of NAND4 gates to give a total of eight sets of 16-bit dual rail data. The SRAM32x16 units drive their bit lines active low so only one of the four data inputs to the NAND4 gate will have valid active-low data, while all other buses will be pulled high by their precharge circuits. The outputs of the NAND4 gates are registered to split the single long read cycles into multiple smaller cycles to reduce the cycle time and improve throughput. The outputs of these NCL registers are then combined using cascaded TH14 and TH12 gates. The final 16-bit dual-rail output data is again latched in an NCL register. There are thus a total of three registration stages on the read side and one registration stage on the write side. The improvement in throughput comes at the cost of a marginal increase in latency. If this increase in latency is not acceptable in a particular application, then one of the registration stages on the read side can be removed. It is likely that the read and write cycle times can be further improved with minimal impact on latency by introducing an additional pipelining stage after the address decoding unit, although this idea has not been investigated as part of the current work.

5.3 Simulation and Performance Measurements

As with the hashing and completion detection circuits, the SRAM designs were implemented in Cadence Virtuoso® using a 1V, 28nm ultra-thin body and BOX silicon-on-insulator (UTBB-SOI) process kit from ST Microelectronics. The threshold gates used in the SRAM had been designed and tested separately [113], while the Boolean gates are from a standard cell library from ST Microelectronics. Two different 32x16 units were designed: one with Null Column storage (SRAM32x16N) and the other without (SRAM32x16). The SRAM units were then combined to build 1kx16 banks - one with the Null storage column (SRAM1kx16N) and the other without (SRAM1kx16). Both the designs had write completion detection. The SRAM designs were simulated using Cadence UltraSim® at 27 °C. The average propagation delay, cycle time and energy consumption per operation for both the read and write cycles were determined using Virtuoso® Visualisation and Analysis XL Calculator.

5.3.1 Write Performance

There are two primary effects that need to be considered when evaluating the write performance of the SRAM: firstly, the effect of the early completion detection circuit and secondly the effect of the Null storage column. Intuitively, it can be expected that the early completion detection should improve write latency and cycle times, while the addition of the Null-storage column should negatively affect the same. Two separate experiments were conducted to distinguish between these effects. The effect of the early completion detection was evaluated by simulating the two 1k SRAM designs for the following cases:

A) SRAM word overwritten by new word where all bits have changed;
B) SRAM word overwritten by new word where none or a small number of the bits have changed.

In the second set of experiments, the following data vectors were generated and fed to only the SRAM1kx16N (1K SRAM with Null storage column) in order to evaluate the effect of the Null storage column:

C) SRAM word location containing valid data changed to Null;
D) SRAM word location initially at Null changed to Not-Null with the remaining data bits not changed;
E) SRAM word location initially at Null changed to Not-Null with a corresponding change in the actual data bits stored. in the SRAM cell array.

Cases D and E together are the inverse operation of Case C but have to be treated as two distinct cases because of the way the NCL SRAM with Null storage is constructed.

It has been discussed in section 5.1.3 that whenever a WNULL = 1 value is present in the Null storage column, write access to the SRAM array is disabled. This ensures that the data value that was originally present in the particular word location in the SRAM array is not disturbed. Consider that at a subsequent time such a word location is to be reverted back to a WNULL = 0 value and the data to be stored at this location is the same as the old data value. Now, as soon as the WNULL = 0 value is written, the write access to the SRAM array would be enabled, and the write operation would complete sooner compared to the case where the data bits were changed because of the early write completion detection feature in the main SRAM

Table 5.2: Write operation: Effect of per-bit write completion detection circuit on latency, cycle time and energy per operation

	w/o Null Column		w/ Null Column	
	All bits changed	No bits changed	All bits changed	No bits changed
Cycle Time (ps)	1348	1269 (5.8%)	1445	1373 (5.0%)
Latency (ps)	787	715 (9.1%)	886	818 (7.7%)
Energy (pJ)	2.64	2.41 (8.7%)	2.81	2.47 (12.1%)

array. This is why the two situations have to be evaluated separately. The particular benefit of this organisation will become apparent in a later section, in which SRAM accesses in tries used for address lookup are discussed.

Table 5.2 shows the write performance of the SRAM banks (SRAM1kx16 and SRAM16x16N) for case A and case B. Two observations can be made from these data: firstly for both the SRAM banks (with and without Null storage column) there is a small but useful reduction in throughput delay, latency and energy when none of the bits in a word are being overwritten, compared to the situation when all the bits in the word are changing. This is an effect of the per-bit completion detection circuits. In practice, for most of the writes, the number of bits that change their value would be somewhere between the two extremes so that the average performance will also be between these two extremes. Thus, the effect on cycle times, latency and energy consumption will be marginal, at best. It is worth noting, however that this difference implies that the average throughput and latency of the SRAM will not be dependent on the worst case propagation delays as is the case in clocked SRAMs, but will vary with the input data pattern. Secondly, it can be seen that for SRAMs with the Null storage column, the overheads incurred by the additional per-bit write completion detection hardware are of approximately the same order ($\sim$5% throughput, $\sim$7% latency and $\sim$12% energy).

The measurements from the second set of experiments involving the Null storage column are presented in Table 5.3. The numbers reported in this table for the SRAM1kx16 bank i.e., SRAM without Null-storage column, correspond to the average value obtained for input data having a random number of bits different from the value already in the SRAM. For the SRAM bank with the Null-storage column SRAM1kx16N, in the case where existing data is overwritten

Table 5.3: Write operation: Effect of Null storage column on cycle time, latency and energy consumption per operation

	w/o Null column	w/ Null column		
		Data to Data	Null to Data	Data to Null
Cycle Time (ps)	1334	1436 (-7.6%)	1491 (-11.7%)	1141 (14.5%)
Latency (ps)	775	878 (-13.2%)	930 (-20%)	673 (13.1%)
Energy (pJ)	2.52	2.64 (-4.7%)	2.68 (-6.3%)	1.25 (50%)

with new data (both Null $\rightarrow$ Data and Data $\rightarrow$ Data conditions), a random number of bits is assumed to be changing.

It can be seen that the presence of the Null column slightly increases the cycle time and energy consumption during Data write operations but results in comparable savings in latency and cycle time and significant energy savings during a Null write. However, in routing applications, the SRAM will be written only when the IP table is initially constructed or updated. Further, the rate of updates is much slower (at most a few hundred updates per second [114]) than the rate at which address lookup operations are performed ($\sim$ 200 Million per second [9]). The write operations will, therefore, be only a minor component of the overall performance and the additional latency and energy consumption because of the Null storage column, or the write completion detection circuit will be largely insignificant.

5.3.2 Read Operation

As with the write performance, the read performance also depends on whether a Null or Data location is being accessed. Table 5.4 shows a performance comparison of the two SRAM designs during a read operation. The absolute values in the table for the SRAM1kx16N bank are followed by the percentage gain (+ve) or penalty (-ve) over equivalent values obtained from the SRAM1kx16 bank. It can be seen that the SRAM with Null columns exhibits significantly shorter cycles consuming much lower energy whenever Null locations are accessed. Conversely, accessing regular (non-Null) locations incurs a slight penalty due to the additional gating of the Null bit. While it may seem wasteful to have to spend more energy and time to access data location to be able to access Null locations faster, it will be shown in Chapter 6 that this specific property of being able to store Null values and associating a lower penalty with Null accesses offers significant advantages in the design of a packet lookup engine.

Table 5.4: Read operation: Effect of Null storage column on cycle time, latency and energy consumption per operation

	w/o Null column	w/ Null column	
		Data Read	Null Read
Cycle Time (ps)	1649	1733 (-5.1%)	1016 (38.4%)
Latency (ps)	747.5	824 (-10.2%)	547 (26.8%)
Energy (pJ)	2.69	2.75 (-2.2%)	1.10 (59.1%)

Summary

This chapter has presented the architecture and performance evaluation of an NCL-based RAM design that includes read and write completion detection and a single-bit storage to indicate the Null state of each word in the SRAM. Although NCL uses two rails to represent one Boolean variable, an NCL SRAM has the same number of 6T cells as a Boolean SRAMs and this is a significant advantage of the architecture. The only additional resources in the NCL SRAM are the 6T cells in the Null Column and the completion detection circuit. These additional hardware blocks allow the RAM to be used in pipelined NCL circuits without having to undergo NCL to Boolean transformation and vice-versa. Secondly, the additional hardware results in situations where there is marginal penalty (-5.1%) in accessing regular data elements in the SRAM, while resulting in a significant benefit when reading (38.4%) Null locations. The energy saving while accessing the Null locations in RAM structures without the Null-storage column is also significantly lower (59.1%) than in structures where the Null-storage column is not present. A comparison with Boolean SRAMs is not appropriate as the benefits of using the NCL SRAM can only be achieved inside a system that is completely designed in Null Convention Logic. It is predicted that an SRAM1kx16N based design of the address lookup function in a router will exhibit better performance than a design based on the SRAM1kx16 block because of the number of Null locations that are accessed in a binary-trie based lookup, and this is discussed in Chapter 6.

While technology is important, it's what we do with it that truly matters.

Muhammad Yunus

Chapter 6

Compact trie with Bloom filters in Null Convention Logic

The work presented in Chapter 3 showed that a Boolean implementation of the Compact-trie Lookup algorithm results in a memory structure that has a better memory efficiency without compromising on the lookup performance. It was also demonstrated that Bloom filters could be used to prevent unnecessary accesses to the prefix storage SRAMs in these tries. A careful selection of the levels in the trie on which Bloom filtering was carried out can result in additional power saving compared to a total Bloom filtering approach. This chapter demonstrates that in an NCL-based design while, the addition of Bloom filters does result in improvement in the energy consumption as it did in the Boolean design, a greater improvement is achieved through the use of NCL RAMs with Null-storage column to store prefix information without using a Bloom filter. This suggests that a straight translation of a Boolean logic design to Null Convention Logic is not necessarily the best possible approach.

6.1 Design considerations

The design and the theory of operation of a Compact-trie have already been discussed in Chapter 3. Before 'ncl-ising' the complete Compact-trie, its components such as the hash function, completion detection circuit and memory were implemented using the NCL approach and their performance evaluated and presented in Chapters 4 and 5 respectively. The final step in the process is to incorporate the appropriate individual elements in different combinations and identify the one combination that demonstrates the best balance of energy, area and performance.

The architecture of an Enhanced Compact-trie with epsilon links ($E - Ctrie_{\epsilon}$) implemented in NCL is no different from that implemented in Boolean logic and follows the block diagram of Figure 3.5. The Bloom filter in the NCL implementation also uses a CRC32 hash algorithm for generating the indices. The CRC32 algorithm is preferred because it can use the hash value from the previous trie level and the MSB of the prefix's active part to complete the hash computation in a single cycle. The complete $E - Ctrie_{\epsilon}$ is pipelined, and the registration stages use the completion detection circuits with complementary gates (CD-CG) instead of using the completion detection circuits with conventional gates (CoCD) as it has been shown in Chapter 4 that the CD-CG circuits have a better energy performance.

The decision on whether to use SRAMs with or without the Null-storage column is, however, critical because the Null-storage column has an area overhead and the SRAM with Null-storage column should be used only in situations where the energy savings due to Null location accesses are expected to be significant. In the case of a Boolean SRAM storing any information, a zero value and a 'no data' (Null) value are either considered the same or the 'no data' value has to be stored as a unique bit pattern not present in regular data values. In the case of NCL, however, the 'no data', i.e. the Null value is readily available and may be used to indicate the absence of the requested information.

The $E - Ctrie_{\epsilon}$ has three memory elements - the next child information RAM, Bloom filter RAM and prefix storage RAM. An analysis of the trie density numbers in Table 3.8 reveals that for the trie-traversal information RAM, all the internal nodes will have at least one valid child address and it is only the final node of a path in the trie that has a Null value. It is therefore expected that implementing this memory with the extra Null-storage column will

not provide any significant energy or throughput improvement. The Bloom filter memory also does not need the additional Null storage as it is only a single-bit wide memory. For the prefix information memory, however, a large percentage of internal nodes store a zero or Null value in their prefix information fields. Experiments conducted on the NCL SRAM have already shown that accesses to locations that have a Null value in the RAM can be completed much faster and consume less energy than accesses to locations containing a Data value. It is possible therefore that the prefix information memory implemented with NCL SRAMs having Null-storage column will lead to a better performance than that observed in Boolean logic.

To test system performance, the blocks in the hardware block diagram of Fig. 3.5 are implemented in NCL in either of the following two ways.

1. The behaviour is coded in a proprietary Hardware Description Language known as "NELL"[1], and translated using a proprietary synthesis tool into a structural Verilog netlist. The synthesis process instantiates components from a library of NCL threshold gates. The 'key_stripper' module and the CRC computation block within the 'bloom_filter' module were designed in this manner.

2. If the behaviour of a block is better expressed as a flow of Data and Null wavefronts through NCL threshold gates, registers, or existing NCL modules, then the block is described as a structural netlist of these NCL components using Cadence Virtuoso® schematic capture. The match_stage, match_module and next_child modules were designed in this fashion.

The energy consumption and latency of the complete NCL-based implementation of the $E - Ctrie_\epsilon$ may be obtained through a circuit-level simulation in Cadence ADE-XL®. However, the design has multiple RAM blocks that hold the prefix information, trie-traversal information and Bloom filter bits. Since the objective of this work is to evaluate only the latency and energy consumption in an NCL implementation of an address lookup algorithm and not evaluate the lookup algorithm itself, the memory is pre-configured with prefix and trie-traversal information at the start of the simulation. Performing this initialisation in Cadence ADE-XL® will increase simulation times and the complexity of the testbench infrastructure in Cadence ADE-XL®. To work around this situation each of the individual modules in the design were simulated in Cadence ADE-XL® under different input conditions and the delay and energy

[1] supplied by Wave Semiconductors Inc.

consumption for each operation were characterised. It was observed that for all the modules except the key_stripper, the input to output delay and input to ko (completion detection signal) delay are uniformly distributed around a mean with the mean and standard deviation for the Null and Data wavefronts being different. These values obtained through circuit simulations are incorporated into delay and energy models for the individual modules and are used subsequently.

It may be noted that Boolean implementations of the modules in the $E - Ctrie_{\epsilon}$ have already been designed and tested to obtain the results reported in Chapter 3. These Boolean modules take in a clock signal and single-rail inputs and generate single-rail outputs 'x' clock cycles after the input, where 'x' is the number of registration stages in the Boolean design.

It is then possible to wrap each Boolean module in the $E - Ctire_{\epsilon}$ in NCL wrapper modules to create as many 'NCL-in-Verilog' modules. This new module is written in Verilog but has an NCL behaviour instead of the conventional Boolean behaviour. For the registration stages, instead of wrapping Boolean registers in NCL wrappers, the 'NCL-in-Verilog' modules were created as a structural Verilog netlist instantiating behavioural models of the NCL threshold gates. Both of these 'NCL-in-Verilog' modules were used to build the complete $E - Ctrie_{\epsilon}$ design in Verilog but with an *NCL like* behaviour and use it for simulation in Cadence NC-Verilog®. Figure 6.1 shows a block diagram of the NCL wrapper instantiating the Boolean design. As shown, the wrapper also contains a converter that translates dual rail signals to single rail signals at the input (d2s_reg) and a single rail to dual-rail converter (s2d_reg) at the output. The d2s_reg and s2d_reg modules also detect the 'complete Data' or 'complete Null' state on the input or output signals as appropriate and generate the handshaking signals.

The 'complete-Data' and 'complete-Null' signal at the input enables the NCL delay and energy model that generates a random output delay and a random *ack*-signal delay value drawn from the uniform distribution models created previously through Cadence ADE-XL® simulations. These delays are of the order of 100s of picoseconds and are introduced in the outputs and the *ack*-signal using the '#delay' facility in Verilog. The completion signal at the input is also passed to a clock generator that generates the appropriate number of clock pulses for the Boolean module. The Boolean design needs a clock to function correctly, while the NCL design does not have any clock. The local clock generator when triggered, generates the clock signal with a period (typically 1 ps) that is much smaller than the delay of the NCL design.

This clock generator ensures that the Boolean module generates the functionally correct single rail outputs. The clock generator is triggered by the input-complete detect signal because the Boolean module should receive the clock signal only when the input is complete and not otherwise. The functionally correct single rail output is passed to the s2d_reg block that produces the dual-rail outputs with the appropriate delay only when the inputs are complete, and the next stage in the pipeline is requesting a Data value as indicated by the ki input. The outputs when produced also cause the generation of the 'output-complete' signal which is delayed in the delay insertion block (Figure 6.1) to generate the ko (*ack*) signal towards the upstream node.

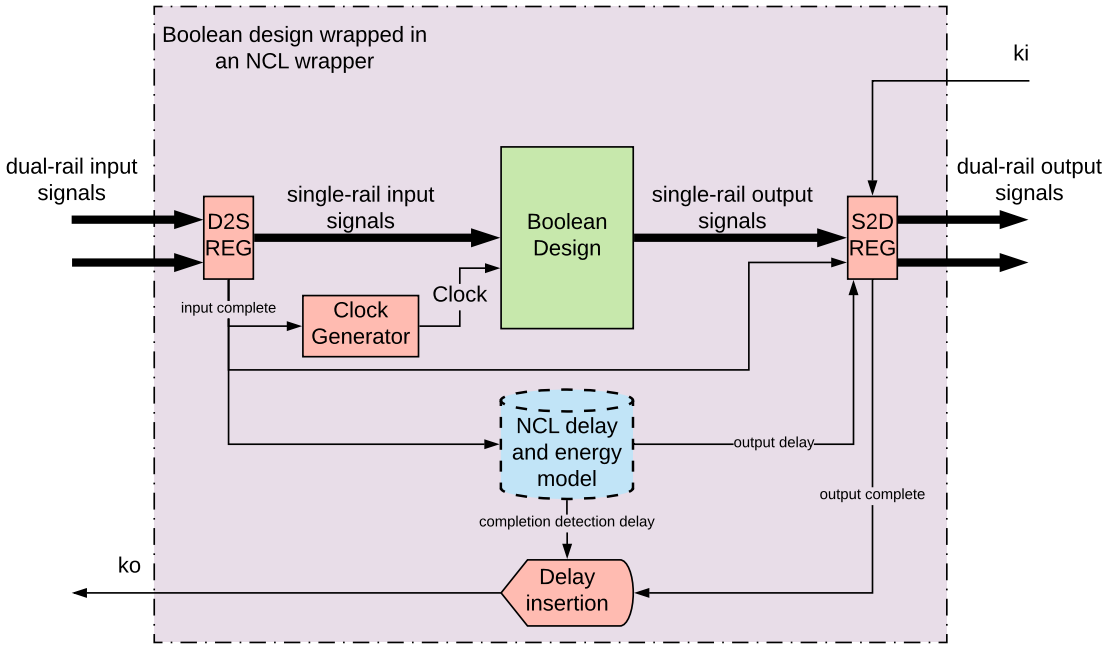


Figure 6.1: NCL wrapper around a Boolean module to produce the 'NCL-in-Verilog' modules used for simulations

6.2 Results and Discussion

6.2.1 Simulation setup

The performance of the proposed algorithm was evaluated using a combination of software simulation in Python, and Cadence NC-Verilog® with delay energy and area numbers obtained through designs generated in Cadence Virtuoso® and simulated using Cadence UltraSim®.

In the simulations of the synchronous design in Chapter 3, of the six IPv4 routing tables downloaded from Packet Clearing House [103] on 01-April-2017, two (MGM and LYS) were used as

representative tables for evaluating the effect of Bloom filtering. In the present NCL simulations, the same two tables have been used. Synthetic packet traces were also generated that contained roughly five times the total number of prefixes in the routing table, with destination IP addresses distributed uniformly over the range of addresses covered by the prefixes.

It has been demonstrated in Chapter 3 that for an enhanced compact trie lookup, the addition of Bloom filters improved the energy consumption in the trie. The Bloom filters, however, occupy additional area resources and also increase the latency of the system. Meanwhile, in Chapter 5, it was shown that accessing a Null location results in shorter cycle times and lower energy consumption, while increasing the cycle times for Data accesses and also occupying additional area. The performance of the following four designs that include different combinations of Bloom filter and Null-storage column SRAM are evaluated:

- **WOBF_WONC**: without Bloom filter, without Null-storage column in prefix memory;
- **WBF_WONC**: with a Bloom filter, without Null-storage column in prefix memory;
- **WOBF_WNC**: without Bloom filter, with Null-storage column in prefix memory;
- **WBF_WNC**: with Bloom filter, with Null-storage column in prefix memory.

6.2.2 Simulation Results

6.2.2.1 Cycle Time

The cycle time behaviour of the NCL pipelined structure can be analysed keeping in mind the following important characteristic of NCL pipelines mentioned by Fant in [38] *"As a general rule an occasional fast cycle in the pipeline will always be shadowed by other cycles in the pipeline except in cases where many such fast cycles occur together and do not fall in the shadow of cycles with regular cycle-times."*

If we consider the specific case here, the module that can have an occasional fast cycle is the SRAM with Null-storage column used for the prefix information storage. Figure 6.2 is a wavefront propagation diagram that demonstrates the situation when a Null value is read. In this figure P1, P2, P3, P4 are the combinational stages in the pipelines, while R1, R2, R3, R4 and R5 are the registration stages. The Data and Null wavefronts are indicated by the solid black lines, while the *ack* signals are indicated by the solid red lines. To distinguish between the flow of

signals between the same registration and combinational stages, $R1_f$ and $R1_b$ denote the wavefront and *ack* signal flow respectively through the registration stage R1. Similar terms may be defined for the combinational stages.

The wavefronts thus flow from left to right as:

$$R1_f \rightarrow P1_f \rightarrow R2_f \rightarrow P2_f \rightarrow R3_f \rightarrow P3_f \rightarrow R4_f \rightarrow P4_f \rightarrow R5_f \rightarrow P5_f,$$

and the *ack* signals flow from right to left. At each registration stage, the *ack* signal received from the downstream stage has to first flow left to right through the registration stage, along with the wavefront and the new acknowledge signal generated then continues to flow right to left. The path for the acknowledge signal is thus:

$$R5_b \rightarrow P4_b \rightarrow R4_b \rightarrow R4_f \rightarrow R4_b \rightarrow P3_b \rightarrow ... \rightarrow R2_f \rightarrow R2_b \rightarrow P1_b \rightarrow R1_b,$$

and is shown as a blue dashed line in Figure 6.2.

If it is considered that stage P3 is the prefix storage RAM, and that for the second and third data wavefronts, the prefix RAM reads a Null location, then it is seen that the corresponding cycles complete much quicker than for the other data wavefronts when Data locations are read. However, the *ack* signals flow back through $R4_b \rightarrow P3_b \rightarrow R3_b$ and wait for the next wavefront to arrive. In effect, the fast cycle is shadowed by the slower cycles that come after it, and its presence does not significantly affect the cycle times at the input.

A plot of the cycle times (as seen at the input) for a set of ten thousand IP addresses fed to two $E - Ctrie_\epsilon$ implementations of the MGM routing table are presented in Figure 6.3. The first implementation is when the prefix information memory has the Null-storage column and the second is when it does not. The distribution of the cycle times for both implementations is presented in Figure 6.4. It is apparent from the two Figures 6.3 and 6.4 that the average cycle times with and without Null-column storage is almost the same ($\sim$ 3065 ps), with about the same standard deviation ($\sim$ 60 ps). The distribution of cycle times is left-skewed with a long tail to the right suggesting that there are only a few instances when cycles with short periods occurred at multiple stages in the pipeline resulting in overall short cycle time for the system. While in a large number of instances, the slow cycles dominated the system and the cycle time of the complete system was longer. It may also be concluded that though NCL RAMs with

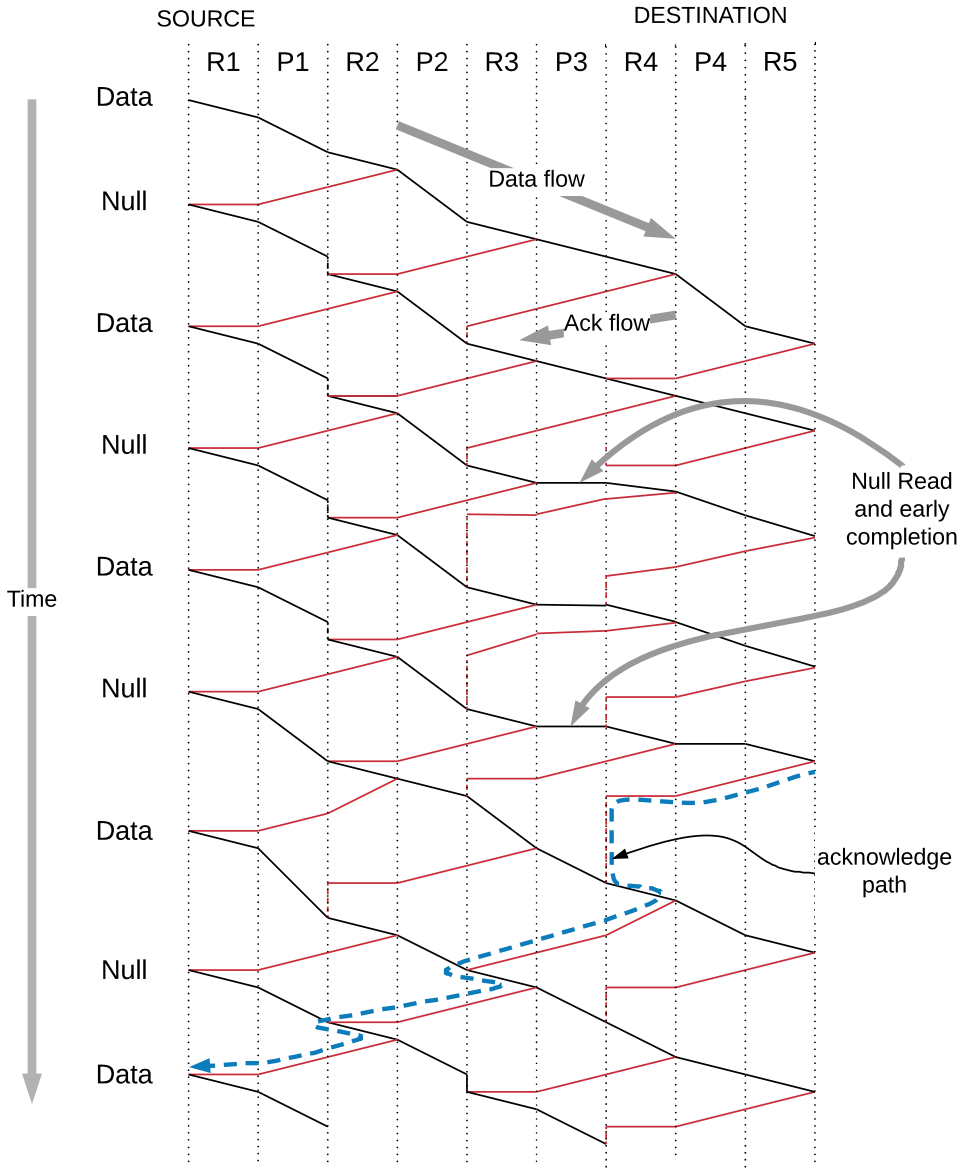


Figure 6.2: Wavefront propagation diagram demonstrating the effect of reading a Null-storage column, on the cycle time of the system

Null storage have a smaller cycle time when Null locations are accessed, their presence in the system does not affect the overall average cycle time.

6.2.2.2 Energy Consumption

The energy consumption of the address lookup process is also measured with the same set of random test vectors. The energy consumed on average per address lookup for both the LYS and MGM tables is presented in Table 6.1.

Table 6.1: Comparison of energy consumption per lookup in NCL implementations of the $E - Ctrie_\epsilon$ for the (a) LYS and (b) MGM routing tables

	w/o Bloom	w/ Bloom
w/o Null	38.35 pJ	29.80 pJ
w/ Null	22.69 pJ	29.23 pJ

(a)

	w/o Bloom	w/ Bloom
w/o Null	37.48 pJ	30.05 pJ
w/ Null	22.61 pJ	29.30 pJ

(b)

For both the routing tables, the highest energy consumption per lookup is seen in the case where neither the Bloom filter nor the Null-storage column is present in the design. In the results obtained with the synchronous Boolean logic implementation of the $E - Ctrie_\epsilon$ presented in Section 3.4.3.2, the presence of the Bloom filter helped reduce the energy consumption. An identical behaviour is seen in the NCL implementation, where the addition of the Bloom filter alone results in a reduction in the energy consumed per lookup (22.2% for LYS and 19.5% for MGM). With the Bloom filter present, adding a Null-storage column to the prefix storage memory does improve the average energy consumption slightly from 29.80 pJ to 29.23 pJ in LYS and from 30.05 pJ to 29.30 pJ in the MGM routing tables. However, this improvement is not significant because the Bloom filters have already filtered out the unnecessary accesses to the prefix storage memory. The only memory reads happening now are either the ones needed to access the correct prefix information or those arising due to the false positives of the Bloom filter. The saving in the energy consumption through accessing Null locations in memory is thus marginal. The least energy consumed per lookup is achieved in the case where instead of using the Bloom filter for filtering out accesses, only the Null-storage column RAM is used for the prefix storage. The energy consumed per lookup, in this case, is significantly less than the case when neither Bloom filter nor the Null-storage column RAM is used ($\sim$ 41% less for LYS and $\sim$39% for MGM). This happens because the Bloom filter prevents unnecessary accesses to prefix locations containing Null. However, it consumes energy to compute the Bloom filter

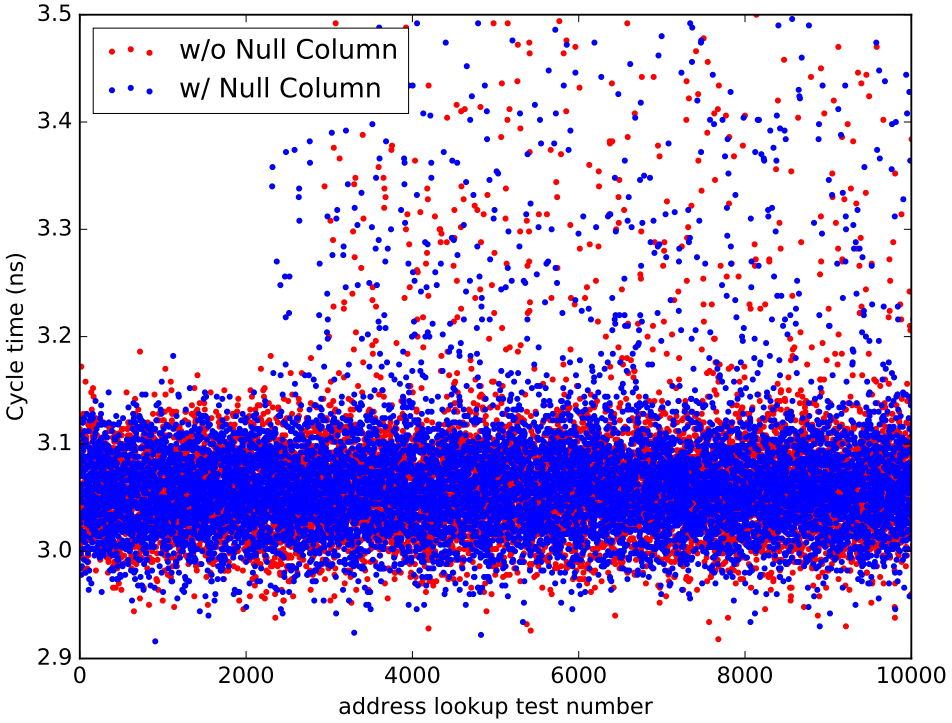
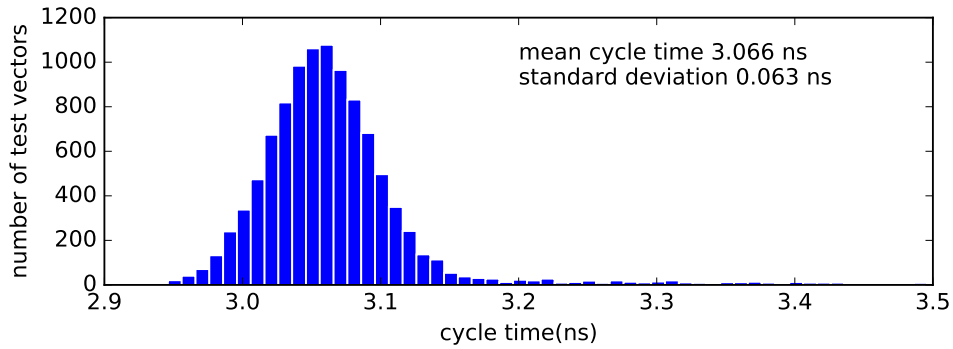
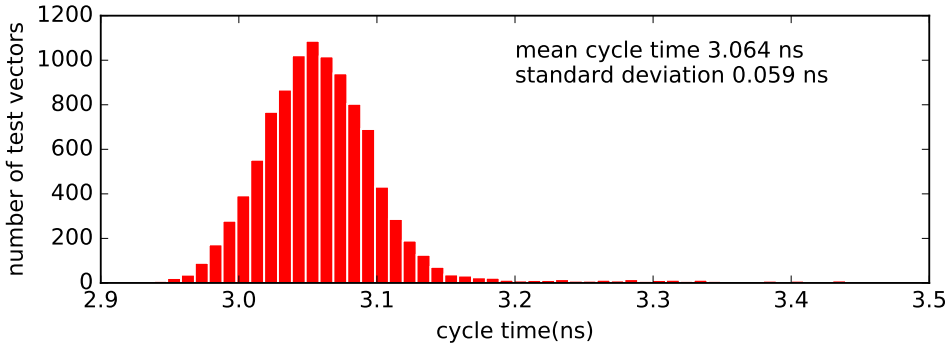


Figure 6.3: Observed cycle time (with 10000 test vectors) at the input of an NCL implementation of the $E - Ctrie_{\epsilon}$ with and without Null-storage column (MGM routing table)



(a) with Null-storage column



(b) without Null-storage column

Figure 6.4: Distribution of cycle times at the input of an NCL implementation of the $E - Ctrie_{\epsilon}$ implementation (a) with Null-storage column and (b) without Null-storage column (MGM routing table)

indices. The Bloom filter RAM also consumes power. With a Null-storage column RAM alone, there is no additional energy being consumed by the Bloom filter, and the prefix storage RAM anyway consumes less energy whenever a Null location is accessed resulting in a much larger reduction in the energy consumption.

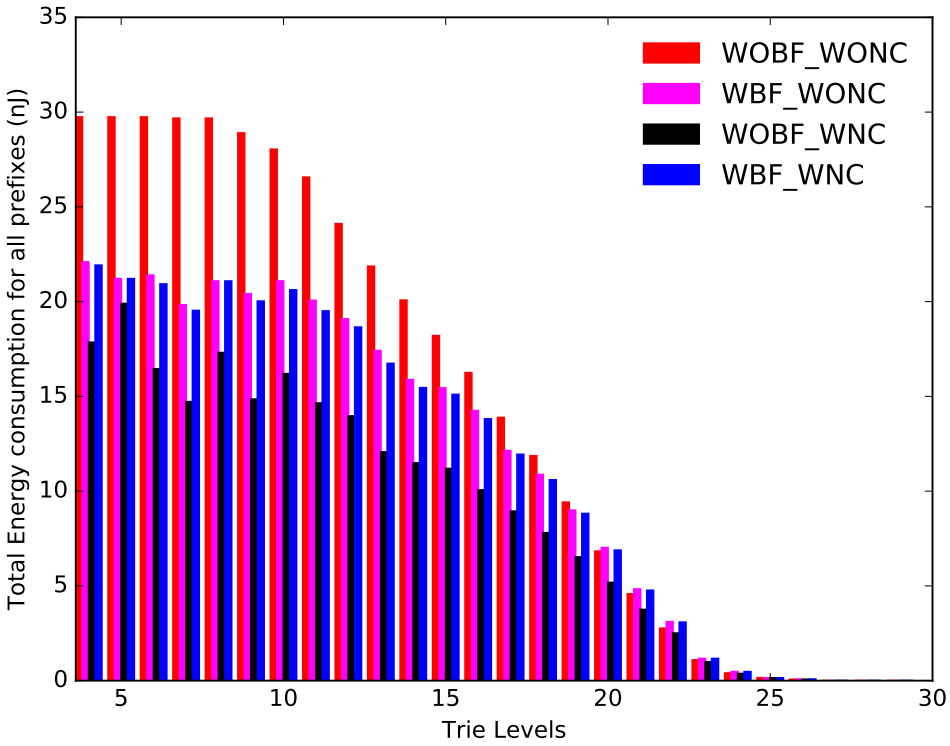
This result is significant as it suggests that in case of an NCL implementation, it is not necessary to have the Bloom filter to reduce energy consumption. Better energy savings can be obtained by utilising the capabilities of the NCL design methodology.

Figure 6.5 also illustrates the energy consumption for all prefixes together at each level in the trie for all four cases. The first four levels of the trie have been omitted from the figure and also from the energy numbers presented in Table 6.1 because at these levels in the trie, there are not much energy savings to be achieved through either technique, as the number of prefixes stored at this level is limited.

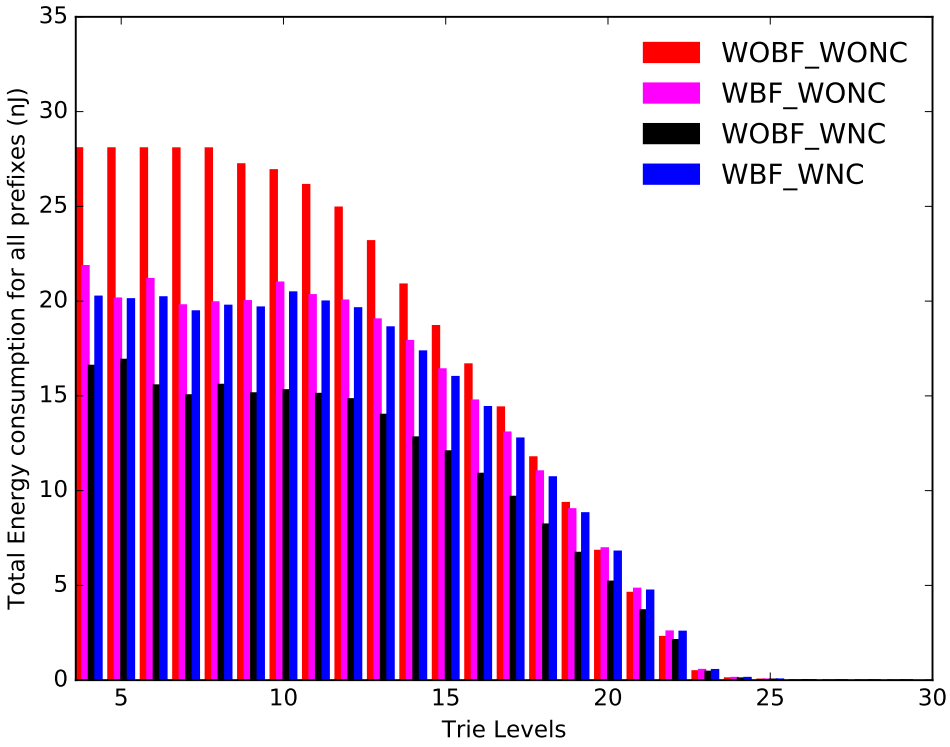
6.2.2.3 Area

The final metric on which the four cases are compared is the area. Because of limitations of the tool, the area figures are not obtained through a place and route of the whole design. Instead, the area of each component used in the design was first estimated through an approximate placement performed in Cadence Layout-XL®. The total area of the designs was then obtained through calculations that used these individual area estimates.

Figure 6.6 illustrates the area overhead because of the Bloom filter and the extra Null-storage column in the prefix RAM as we move down levels of the trie. The tables also list the ratio of the two overheads expressed as a percentage. It can be seen that as we move down the trie, the overhead because of the Null-storage column is much smaller than the BF area (as low as $\sim$5.5% of the Bloom filter area at level 15 of the trie for the MGM routing table). It has already been shown in Section 6.2.2.2 that the energy consumption per lookup with Null-storage RAM is lower than the energy consumption with Bloom filters, the area overhead numbers suggest that deeper in the trie, the amount of additional on-chip area required to reduce energy consumption through the use of the Null-storage column is only a small percentage of the area needed to achieve a poorer energy consumption through the use of the Bloom filter.



(a) LYS



(b) MGM

Figure 6.5: Energy consumption down trie levels in an $E - Ctrie_{\epsilon}$ implementations of both the (a) LYS and (b) MGM tables for the following cases: **WOBF_WONC**: without Bloom filter and without Null-storage column, **WBF_WONC**: with Bloom filter and without Null-storage column and **WOBF_WNC**: without Bloom filter but with Null-storage column, **WBF_WNC**: with Bloom filter and with Null-storage column

Trie Level	BF area overhead (um2)	Prefix RAM area overhead (um2)	Ratio PFX RAM / BF (%)
0	3303.1	2108.6	63.8
1	3303.1	2108.6	63.8
2	3303.1	2108.6	63.8
3	3303.1	2108.6	63.8
4	3303.1	2108.6	63.8
5	3303.1	2108.6	63.8
6	3303.1	2108.6	63.8
7	3303.1	2108.6	63.8
8	5987.3	2108.6	35.2
9	11355.8	2108.6	18.6
10	22092.7	2108.6	9.5
11	43566.7	4217.2	9.7
12	43566.7	4217.2	9.7
13	86514.5	6325.9	7.3
14	86514.5	8434.5	9.7
15	172410.2	10543.1	6.1
16	172410.2	12651.7	7.3
17	172410.2	16869.0	9.8
18	344201.5	21086.2	6.1
19	344201.5	25303.4	7.4
20	344201.5	27412.1	8.0
21	344201.5	25303.4	7.4
22	344201.5	23194.8	6.7
23	86514.5	8434.5	9.7
24	22092.7	2108.6	9.5
25	11355.8	2108.6	18.6
26	3303.1	2108.6	63.8
27	3303.1	2108.6	63.8
28	3303.1	2108.6	63.8
29	3303.1	2108.6	63.8
30	3303.1	2108.6	63.8

(a) LYS

Trie Level	BloomFilter area overhead (um2)	Prefix RAM area overhead (um2)	Ratio PFX RAM / BF (%)
0	3303.1	2108.6	63.8
1	3303.1	2108.6	63.8
2	3303.1	2108.6	63.8
3	3303.1	2108.6	63.8
4	3303.1	2108.6	63.8
5	3303.1	2108.6	63.8
6	3303.1	2108.6	63.8
7	3303.1	2108.6	63.8
8	5987.3	2108.6	35.2
9	11355.8	2108.6	18.6
10	22092.7	2108.6	9.5
11	43566.7	4217.2	9.7
12	86514.5	6325.9	7.3
13	172410.2	10543.1	6.1
14	172410.2	14760.3	8.6
15	344201.5	18977.6	5.5
16	344201.5	27412.1	8.0
17	687784.3	35846.6	5.2
18	687784.3	42172.4	6.1
19	687784.3	50606.9	7.4
20	687784.3	54824.1	8.0
21	687784.3	56932.8	8.3
22	687784.3	52715.5	7.7
23	172410.2	10543.1	6.1
24	22092.7	2108.6	9.5
25	11355.8	2108.6	18.6
26	3303.1	2108.6	63.8
27	3303.1	2108.6	63.8
28	3303.1	2108.6	63.8
29	3303.1	2108.6	63.8

(b) MGM

Figure 6.6: Area overhead of the Bloom filter approach and the Null-storage column approach at each level in an $E - Ctrie_{\epsilon}$ implementation for both the LYS and MGM tables. The difference between the two approaches is expressed as a percentage of the Bloom filter area overhead.

It may be concluded that in an NCL implementation of the lookup algorithm using the $E - Ctrie_{\epsilon}$ structure, the most energy saving is achieved with the least area overhead without affecting the throughput of the system when the system uses a prefix memory storage that has a Null-storage column and no Bloom filter.

Summary

This chapter has provided an analysis of four NCL implementations of an existing SRAM based compact trie algorithm. The simulation results demonstrate that the behaviour of the NCL implementations is different from the Boolean implementation where the addition of Bloom filters reduced worst-case memory accesses to prefix storage RAM thereby improving the power. NCL allows the use of a Null-storage column in the RAM, which can act as a flag to indicate a Null location. This capability of the RAM can be exploited in the prefix storage memory that is typically only sparsely filled because the number of prefix nodes is smaller than the total number of nodes in the trie. The use of the Null-storage RAM for prefix information results in the same throughput as would be achieved without this RAM. However, the advantage of the Null-storage RAM is that it uses much less area and consumes less energy as compared to a Bloom filter.

> Every once in a while, a new technology, an old problem, and a big idea turn into an innovation
>
> Dean Kamen

Chapter 7

Conclusion and the way forward

Since it first came into being, the growth of the Internet has been relentless, and one of the key drivers of this growth has been the capability of the packet router. The router that started off as a software program running on a general purpose microprocessor handling a few kilobits per second is now a standalone system transporting hundreds of terabits of data streaming in through multiple ports. The energy consumption of routers and other networking equipment in modern data centres has now reached alarming proportions and therefore a router not only has to handle immense amounts of data, but it also has to do it fast and in a way that consumes the least energy. In this thesis, an asynchronous Null Convention Logic based solution for performing the critical destination address lookup function in routers has been presented. The solution demonstrates that taking advantage of the unique characteristics of the algorithm and the asynchronous methodology can achieve energy consumption much lower than would have been possible by a naive translation of the Boolean logic design to NCL.

Existing algorithmic SRAM-based approaches to destination address lookup in IP routers were first explored and a recent algorithm, the Compact-trie lookup, was chosen for its superior performance over other RAM–based approaches. The work in this thesis extends the capabilities of the original algorithm through enhancements to the prefix decomposition technique and the application of Bloom filters to improve lookup performance. Current state-of-art in the application of Bloom filters to binary trie structures points to improvement in the lookup latency when the number of memory accesses is reduced. Experiments presented here demonstrate that the application of Bloom filters to a Compact-trie result in a similar reduction in the number of prefix memory accesses When evaluated on FPGA devices where the prefix information is stored on-chip, the presence of the Bloom filter increases the memory utilisation and the logic power as expected. However, the reduction, due to filtering, in the average power consumed

by the prefix RAM is also significant. An evaluation of the effect of Bloom filters on power consumption in trie-based lookup with on-chip storage as presented here has not been attempted before and is, therefore, a contribution of this thesis. Another important observation from these experiments is that power saving with Bloom filters is not uniform down the trie. A targeted Bloom filtering approach can be used to reduce power consumption further while using fewer on-chip resources, without affecting lookup performance. This experiment thus answers the first research question:

How might existing IP address lookup techniques be adapted to improve their energy consumption and/or latency

by demonstrating that the energy consumption of an existing algorithmic address lookup algorithm can be improved by the selective application of a Bloom filter to control the number of memory accesses required.

In the future, with denser and deeper tries, the complexity and the number of hash computations must increase. This implies a non-trivial increase in the delay and power of the Bloom filter necessitating a careful evaluation of the power and delay trade-off in the Bloom filter index computation and prefix memory access. A useful extension to the current work, therefore, would be to produce an a priori 'quality factor' that would determine the advantage of Bloom filtering at a level in the trie, using a more analytical approach.

It is clear from the applications of NCL and other asynchronous design techniques in the literature that just 'going asynchronous' does not really improve the performance. It is essential to understand the algorithm being implemented and uncover its characteristics that can take advantage of the asynchronous design methodology. To evaluate the performance of an NCL-based address lookup system, three popular non-cryptographic hash algorithms, Jenkins, Murmur and CRC32, were first selected, implemented and compared with an equivalent synchronous version. The objective was to identify whether in an NCL implementation the more complex Jenkins or Murmur hash would perform better than the CRC32 algorithm. It was found that even with NCL, the CRC32 implementation had a much smaller cycle time and latency and was, therefore, the most suitable of the three for generating the Bloom filter indices. The analysis of the Murmur and Jenkins hash algorithms showed that even in complex systems, for a given throughput, the latency of the NCL implementation could be as low as

20% of the equivalent synchronous implementation (5.4ns vs 25.6ns in the specific experiments here). The NCL implementation also demonstrates an 800% improvement in the peak current characteristics. The result leads to the conclusion that in case of a Bloom filter, translating the Boolean design into an NCL design improved the latency and peak power consumption while demonstrating a robustness in the face of temperature and process variations.

The experiments also demonstrated that with increasing data path widths and fine-grained pipelining, there is a need for completion detection circuits that occupy a smaller area, consume less energy and have small propagation delays. Two optimised completion detection circuits that used modified threshold gates were then proposed to satisfy this need. The first of these designs was based on complementary threshold gates and exhibited a propagation delay of 238 ps with an area reduction of 25% and an energy/operation improvement of 35% over traditional completion detection schemes. The performance gains were increased further in the second design where the TH12 and the first level of TH44 gates were combined into one THC4D gate and the "Drive" and "Hold" sections of a threshold gate were merged into a single composite network. These changes to the design resulted in a circuit that is about 30% smaller, consumes ∼50% less energy per operation and draws a peak current around 25% that of the conventional Completion Detection circuits while exhibiting a marginally smaller propagation delay (i.e., ∼4%).

Because the present work uses an algorithmic SRAM-based lookup algorithm, an NCL memory was proposed conforming to NCL behaviour. The proposed architecture had completion detection circuits and single-bit "Null" flag per location. The architecture resulted in shorter, energy-efficient read and write cycles for empty locations in memory. In the example implementations presented with Null-storage column, designed using a 28 nm FDSOI process, the technique results in a cycle time reduction of approximately 38.4% and an energy reduction of 59.1% when a Null location is read over the equivalent numbers obtained in memories that do not have the Null-storage column. This saving in the individual memory accesses will reflect in an overall improvement in the performance of the systems only when the memory accesses that encounter such Null locations are a significant proportion of the total memory accesses.

An evaluation of the complete Compact-trie address lookup system with and without these Null-storage column RAMs for the prefix memory, plus the Bloom filters, reveals a rather interesting characteristic. It is seen that for both the routing tables evaluated, the architecture not

employing any Bloom filters, but having only the Null-storage column RAMs for the prefix storage exhibit the smallest area on the chip and also consume the least energy per address lookup. This is because the Null-storage column saves much more energy while consuming lesser area than that saved by the Bloom filter with a larger area overhead. This observation is important because it answers the second research question that this thesis set out to answer, which is:

Will applying an NCL-based asynchronous design paradigm further improve energy and performance compared to an equivalent synchronous implementation and will these systems have to be structured differently from the original implementations?

It may be concluded that the application of an NCL design methodology does improve the energy consumption and area utilisation of the IP address lookup function in Routers when compared with synchronous implementations. However, it is not just a straight translation of the synchronous design that results in these gains, but a careful selection of the architectural blocks that combine the peculiarities of the underlying algorithm with the capabilities of the NCL design methodology. This result is indeed significant and non-obvious and is a major finding of the thesis. The initial hypothesis was that an NCL implementation of a Bloom filtered trie implementation would lead to a significant saving in terms of power and latency. However it was realised that in fact the NCL SRAM itself filtered off the accesses to trie locations that did not contain any prefixes and therefore the Bloom filter was not necessary

A complete IP packet forwarding block in Internet router includes in addition to the address lookup function, packet classification, filtering and flow control. The present work chose to evaluate the performance of an NCL-based implementation of the address lookup function. Hardware based packet classification and filtering use trie-structures similar to the ones used for destination address lookup, with the difference being in the size of the lookup key and the information stored per prefix/entry. The results reported here for address lookup to obtain next hop information suggest that the application of NCL or an equivalent asynchronous design method to the algorithms and structures used for packet filtering, classification and flow control in the packet processor in Internet routers may also result in similar performance improvements. The capability of the Null-storage column RAM may be used in other areas of computing where sparsely populated RAM structures are frequently accessed. Both of these paths of exploration could be an activity for the future.

Thus, although synchronous Boolean systems are the norm at present, it is extremely likely that the routers of the future would be (at least partially) asynchronous to meet the ever-growing demand for more bandwidth, higher performance and lower energy.

The perfect is the enemy of the good.

Voltaire

Appendix Chapter A

Code listing

Simulations of the complete NCL design are performed on Boolean implementations of the individual modules of the compact trie wrapped in NCL wrapper modules to create a 'NCL-in-Verilog' module. These modules are written in Verilog but have an NCL behaviour instead of the conventional Boolean behaviour. The code used for the NCL simulations is available at https://gitlab.com/pdabholkar/IPLookup and is organised as follows.

```
IPLookup
├── ncl
│   └── All design modules written in NELL
├── ncl_as_v
│   └── Nell-in-Verilog modules
├── python
│   ├── Python scripts for parsing routing tables
│   ├── Python scripts for analysing simulation results
│   ├── memMap_LYS - memory initialization files for the LYS routing table
│   └── memMap_MGM - memory initialization files for the MGM routing table
├── routeFiles
│   └── routing table downloaded from PCH
└── rtl
    └── RTL designs in verilog for the compact trie with epsilon links
```

Bibliography

[1] Cisco, "Cisco Visual Networking Index : Forecast and Methodology 2016-2021", Tech. Rep., 2017.

[2] M. P. Mills, "The cloud begins with coal - big data, big networks, big infrastructure, and big power : an overview of the electricity used by the global digital ecosystem", Tech. Rep., 2013.

[3] G. Cook, "How clean is your cloud - catalysing an energy revolution", Greenpeace, Tech. Rep. April, 2012, p. 52.

[4] D. C. Walden, *Looking back at the ARPANET effort, 34 years later*, 2003.

[5] C Partridge, P. P. Carvey, E Burgess, I Castineyra, T Clarke, L Graham, M Hathaway, P Herman, A King, S Kohalmi, T Ma, J McAllen, T Mendez, W. C. Milliken, R Pettyjohn, J Rokosz, J Seeger, M Sollins, S Storch, B Tober, G. D. Troxel, D Waitzman, and S Winterble, "A 50-Gb/s IP router", *Networking, IEEE/ACM Transactions on*, vol. 6, no. 3, pp. 237–248, 1998.

[6] T. Ganegedara, "Algorithms and architectures for high-performance IP lookup and packet classification engines", English, PhD thesis, Ann Arbor, 2013, p. 185, ISBN: 9781303693137.

[7] D. Medhi and K. Ramasamy, *Network Routing*. Morgan Kaufmann Publishers, 2007, ISBN: 9780120885886.

[8] The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces, Ed., *The CAIDA UCSD Chicago A 2014-05-16*.

[9] Y. H. E. Yang, Y. Qu, S. Haria, and V. K. Prasanna, "Architecture and performance models for scalable IP lookup engines on FPGA", in *IEEE International Conference on High Performance Switching and Routing, HPSR*, 2013, pp. 156–163, ISBN: 9781467346207.

[10] V. C. Ravikumar and R. N. Mahapatra, "TCAM architecture for IP lookup using prefix properties", *Micro, IEEE*, vol. 24, no. 2, pp. 60–69, 2004.

[11] T Mishra, S Sahni, and G Seetharaman, "PC-DUOS: Fast TCAM lookup and update for packet classifiers", in *Computers and Communications (ISCC), 2011 IEEE Symposium on*, 2011, pp. 265–270, ISBN: 1530-1346.

[12] T. Banerjee, S. Sahni, and G. Seetharaman, "PC-DUOS plus : A TCAM Architecture for Packet Classifiers", English, *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1526–1539, 2014.

[13] T Banerjee, S Sahni, and G Seetharaman, "PC-TRIO: A Power Efficient TCAM Architecture for Packet Classifiers", *Computers, IEEE Transactions on*, vol. 64, no. 4, pp. 1104–1118, 2015.

[14] K. Sklower, "A tree-based routing table for Berkeley Unix", in *Proceedings of the Winter USENIX Conference*, Berkley, USA, 1991.

[15] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups", pp. 3–14, 1998.

[16] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search", *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 324–334, 1999.

[17] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates", *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[18] W. Jiang and V. K. Prasanna, "Data Structure Optimization for Power- Efficient IP Lookup Architectures", *Computers, IEEE Transactions on*, vol. 62, no. 11, pp. 2169–2182, 2013.

[19] J. Matoušek, M. Skačan, and J. Kořenek, "Towards Hardware Architecture for Memory Efficient IPv4 / IPv6 Lookup in 100 Gbps Networks", in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013 IEEE 16th International Symposium on*, 2013, pp. 108–111, ISBN: 9781467361361.

[20] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups", *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 25–36, 1997.

[21] H. Le and V. K. Prasanna, "Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning", *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 1026–1039, Jul. 2012.

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

[23] J Tobola and J Korenek, "Effective hash-based IPv6 longest prefix match", in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2011 IEEE 14th International Symposium on*, 2011, pp. 325–328.

[24] M Bando, Y. L. Lin, and H. J. Chao, "Flash trie: Beyond 100-Gb/s IP route lookup using hash-based prefix-compressed trie", English, *IEEE/ACM Transactions on Networking*, vol. 20, no. 4, pp. 1262–1275, 2012.

[25] S. Sahni and H. Lu, "Dynamic tree bitmap for IP lookup and update", *Proceedings of the Sixth International Conference on Networking, ICN'07*, no. October, 2007.

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

[27] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters", *Networking, IEEE/ACM Transactions on*, vol. 14, no. 2, pp. 397–409, 2006.

[28] H Song, F Hao, M Kodialam, T. V. Lakshman, I. C. Society, and Ieee, "IPv6 lookups using distributed and load balanced bloom filters for 100Gbps core router line cards", English, in *28th Conference on Computer Communications, IEEE INFOCOM 2009*, Rio de Janeiro, 2009, pp. 2518–2526, ISBN: 0743166X (ISSN); 9781424435135 (ISBN).

[29] H. Lim and N. Lee, "Survey and Proposal on Binary Search Algorithms for Longest Prefix Match", *Communications Surveys & Tutorials, IEEE*, vol. 14, no. 3, pp. 681–697, 2012.

[30] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On Adding Bloom Filters to Longest Prefix Matching Algorithms", *Computers, IEEE Transactions on*, vol. 63, no. 2, pp. 411–423, 2014.

[31] J. H. Mun and H. Lim, "New Approach for Efficient IP Address Lookup Using a Bloom Filter in Trie-Based Algorithms", *Computers, IEEE Transactions on*, vol. 65, no. 5, pp. 1558–1565, 2016.

[32] O. Erdem, A. Carus, and H. Le, "Compact trie forest: Scalable architecture for IP lookup on FPGAs", in *2012 International Conference on Reconfigurable Computing and FPGAs*, IEEE, Dec. 2012, pp. 1–6, ISBN: 9781467329217.

[33] O. Erdem and A. Carus, "Clustered linked list forest for IPv6 lookup", English, in *Proceedings - IEEE 21st Annual Symposium on High-Performance Interconnects, HOTI 2013*, San Jose, CA: IEEE Computer Society, 2013, pp. 33–40, ISBN: 9780768551036.

[34] O Erdem, A Carus, and H Le, "Large-scale SRAM-based IP lookup architectures using compact trie search structures", English, *Computers and Electrical Engineering*, vol. 40, no. 4, pp. 1186–1198, 2014.

[35] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon, "High-performance microprocessor design", *Solid-State Circuits, IEEE Journal of*, vol. 33, no. 5, pp. 676–686, 1998.

[36] S. A. Butt, S Schmermbeck, J Rosenthal, A Pratsch, and E Schmidt, "System Level Clock Tree Synthesis for Power Optimization", in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, 2007, pp. 1–6.

[37] P. A. Beerel, "Asynchronous circuits: An increasingly practical design solution", *Proceedings - International Symposium on Quality Electronic Design, ISQED*, vol. 2002-Janua, no. May 2002, pp. 367–372, 2002.

[38] K. M. Fant, *Logically Determined Design : Clockless System Design with NULL Convention Logic(TM)*. New Jersey: John Wiley & Sons Inc., 2005, ISBN: 0471684783.

[39] K. M. Fant and S. A. Brandt, *NULL Convention Logic TM: a complete and consistent logic for asynchronous digital circuit synthesis*, 1996.

[40] D. E. Muller and W. S. Bartky, "A Theory of Asynchronous Circuits", in *International symposium on Theory of Switching*, vol. 1, Harvard University Press, 1959, pp. 204–243.

[41] J. Sparsø and S. Furber, *Principles Asynchronous Circuit Design*. Springer, 2002, ISBN: 0792376137.

[42] P. Dabholkar and P. Beckett, "A High Throughput, Low Latency Null Convention Logic 16x16-bit Multiplier", in *10th International Conference on Signal Processing and Communication Systems (ICSPCS)*, 2016, pp. 378–385, ISBN: 9781509009411.

[43] Juniper Networks, *T Series Core Routers*.

[44] A. S. Tanenbaum, *Computer Networks*, 4th. Prentice Hall Inc, 2003.

[45] B. Agrawal and T. Sherwood, "Ternary CAM power and delay model: Extensions and uses", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 5, pp. 554–564, 2008.

[46] A Rasmussen, A Kragelund, M Berger, H Wessing, and S Ruepp, "TCAM-based high speed Longest prefix matching with fast incremental table updates", in *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*, 2013, pp. 43–48, ISBN: 2325-5552.

[47] S. Yan and K. Min Sik, "A Hybrid Approach to CAM-Based Longest Prefix Matching for IP Route Lookup", in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, 2010, pp. 1–5, ISBN: 1930-529X.

[48] M. J. Akhbarizadeh, M Nourani, D. S. Vijayasarathi, and P. T. Balsara, "A nonredundant ternary CAM circuit for network search engines", English, *Ieee Transactions on Very Large Scale Integration (Vlsi) Systems*, vol. 14, no. 3, pp. 268–278, 2006.

[49] B. Gamache, Z. Pfeffer, and S. Khatri, "A fast ternary CAM design for IP networking applications", *Proceedings. 12th International Conference on Computer Communications and Networks*, vol. 00, no. C, pp. 434–439, 2003.

[50] Z Ullah, M. K. Jaiswal, and R. C. C. Cheung, "Z-TCAM: An SRAM-based Architecture for TCAM", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2014.

[51] W. Jiang, "Scalable Ternary Content Addressable Memory implementation using FPGAs", in *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, 2013, pp. 71–82.

[52] H. Yi-Mao, C. Yuan-Sun, C. Chao-Yang, H. Chung-Hsun, and Y. Hsi-hsun, "A high throughput ASIC design for IPv6 routing lookup system", in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, 2013, pp. 505–508, ISBN: 0271-4302.

[53] W. Quan, C. Xu, A. V. Vasilakos, J. Guan, H. Zhang, and L. A. Grieco, "TB2F: Tree-bitmap and bloom-filter for a scalable and efficient name lookup in Content-Centric Networking", *2014 IFIP Networking Conference, IFIP Networking 2014*, 2014.

[54] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey", *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[55] N. Hua, E. Norige, S. Kumar, B. Lynch, H. Nan, E. Norige, S. Kumar, B. Lynch, N. Hua, E. Norige, S. Kumar, and B. Lynch, "Non-crypto hardware hash functions for high performance networking ASICs", *Proceedings - 2011 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2011*, pp. 156–166, 2011.

[56] D. E. Knuth, *The art of computer programming vol 3: sorting and searching*, 3rd. Addison Wesley, 1998.

[57] M. Mitzenmacher and S. Vadhan, "Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream", in *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '08, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008, pp. 746–755.

[58] A. Fiessler, D. Loebenberger, S. Hager, and B. Scheuermann, "On the Use of (Non-)Cryptographic Hashes on FPGAs", in *Applied Reconfigurable Computing*, S. Wong, A. C. Beck, K. Bertels, and L. Carro, Eds., Cham: Springer International Publishing, 2017, pp. 72–80, ISBN: 978-3-319-56258-2.

[59] C. Pesyna, *Choosing a Good Hash Function*.

[60] A. Appleby, *Murmur Hash*.

[61] B. Jenkins, "ALGORITHM ALLEY-What makes one hash function better than another? Bob knows the answer, and he has used his knowledge to design a new hash function that may be better than what you're using now.", *Dr Dobb's Journal-Software Tools for the Professional Programmer*, vol. 22, no. 9, pp. 107–110, 1997.

[62] IEEE, *IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation*, Dec. 2017.

[63] M. A. Gosselin-Lavigne, H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "A performance evaluation of hash functions for IP reputation lookup using bloom filters", *Proceedings - 10th International Conference on Availability, Reliability and Security, ARES 2015*, pp. 516–521, 2015.

[64] K. Lim, K. Park, and H. Lim, "Binary Search on Levels Using a Bloom Filter for IPv6 Address Lookup Categories and Subject Descriptors", in *ANCS*, 2009, pp. 185–186, ISBN: 9781605586304.

[65] W. W. Peterson and D. T. Brown, "Cyclic Redundany Codes for Error Detection", *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–236, 1961.

[66] International Telecommunication Union, *G.706 Standard*, 1988.

[67] F Yamaguchi and H Nishi, "Hardware-based hash functions for network applications", in *Networks (ICON), 2013 19th IEEE International Conference on*, 2013, pp. 1–6.

[68] J Kastil, V Kosar, and J Korenek, "Hardware architecture for the fast pattern matching", in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013 IEEE 16th International Symposium on*, 2013, pp. 120–123.

[69] K. Deokho, O. Doohwan, and W. W. Ro, "Design of power-efficient parallel pipelined bloom filter", *Electronics Letters*, vol. 48, no. 7, pp. 367–369, 2012.

[70] O Rottenstreich and I Keslassy, "The Bloom Paradox: When Not to Use a Bloom Filter", *Networking, IEEE/ACM Transactions on*, vol. 23, no. 3, pp. 703–716, 2015.

[71] Internet Technology Roadmap for Semiconductors (ITRS), "2013 Overall Roadmap Technology Characteristics (ORTC) Table", Tech. Rep., 2013.

[72] S. C. Smith and J. Di, *Designing Asynchronous Circuits using NULL Convention Logic (NCL)*, S. M. U. Mitchell A.Thornton, Ed. Morgan & Claypool, 2009, ISBN: 9781598299816.

[73] Internet Technology Roadmap for Semiconductors (ITRS), "International Technology Roadmap for Semiconductors 2.0 2015 Edition System Integration", Tech. Rep., 2015, pp. 1–21.

[74] P. Brazil, "Proposal of an Exploration of Asynchronous Circuits Templates and their Applications", no. 077, 2014.

[75] S. M. Nowick and M. Singh, "High-performance asynchronous pipelines: An overview", *IEEE Design and Test of Computers*, vol. 28, no. 5, pp. 8–22, 2011.

[76] C. J. Myers, *Asynchronous Circuit Design*. New York: Wiley, 2001.

[77] A. M. Lines, "Pipelined Asynchronous Circuits", *Computer Science Technical Reports*, 1998.

[78] S. Hauck, "Asynchronous design methodologies: An overview", *Proceedings of IEEE*, vol. 83, no. 1, pp. 69–93, 1995.

[79] M. T. Moreira, C. H. M. Oliveira, R. C. Porto, and N. L. V. Calazans, "Design of NCL gates with the ASCEnD flow", *2013 IEEE 4th Latin American Symposium on Circuits and Systems, LASCAS 2013 - Conference Proceedings*, pp. 1–4, 2013.

[80] S. L. Hurst, "An introduction to threshold logic: a survey of present theory and practice", *Radio and Electronic Engineer*, vol. 37, no. 6, pp. 339–351, Jun. 1969.

[81] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle - An RTL Approach to Asynchronous Design", in *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*, 2012, pp. 65–72, ISBN: 1522-8681.

[82] M Ligthart, K Fant, R Smith, A Taubin, and A Kondratyev, "Asynchronous design using commercial HDL synthesis tools", in *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, 2000, pp. 114–125, ISBN: 1522-8681.

[83] S. C. Smith, R. F. DeMara, J. S. Yuan, M Hagedorn, and D Ferguson, "NULL convention multiply and accumulate unit with conditional rounding, scaling, and saturation", *Journal of Systems Architecture*, vol. 47, no. 12, pp. 977–998, 2002.

[84] S. C. Smith, "Design of an FPGA logic, element for implementing asynchronous NULL convention logic circuits", English, *Ieee Transactions on Very Large Scale Integration (Vlsi) Systems*, vol. 15, no. 6, pp. 672–683, 2007.

[85] S. K. Bandapati and S. C. Smith, "Design and characterization of NULL convention arithmetic logic units", English, *Microelectronic Engineering*, vol. 84, no. 2, pp. 280–287, 2007.

[86] E. Esteve, *WTL Leverage FDSOI to Achieve Both Low Power AND High Speed*, 2014.

[87] T. Soon-hwei, L. Poh-Yee, and M. S. Sulaiman, "A 160-Mhz 45-mW Asynchronous Dual-Port 1-Mb CMOS SRAM", in *2005 IEEE Conference on Electron Devices and Solid-State Circuits*, vol. 00, Dec. 2005, pp. 351–354, ISBN: 0780393392.

[88] M. F. Chang, S. M. Yang, and K. T. Chen, "Wide Vdd Embedded Asynchronous SRAM With Dual-Mode Self-Timed Technique for Dynamic Voltage Systems", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 8, pp. 1657–1667, Aug. 2009.

[89] J. Dama and A. Lines, "GHz Asynchronous SRAM in 65nm", *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pp. 85–94, 2009.

[90] A. Baz, D. Shang, F. Xia, and A. Yakovlev, "Self-timed SRAM for energy harvesting systems", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6448 LNCS, no. 2, pp. 105–115, 2011.

[91] J. Tse and A. Lines, "NanoMesh: An asynchronous kilo-core system-on-chip", *Proceedings - International Symposium on Asynchronous Circuits and Systems*, pp. 40–49, 2013.

[92] E Kasapaki, Spars, x00D, and J., "Argo: A Time-Elastic Time-Division-Multiplexed NOC Using Asynchronous Routers", in *Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on*, 2014, pp. 45–52, ISBN: 1522-8681.

[93] N Onizawa, A Matsumoto, T Funazaki, and T Hanyu, "High-Throughput Compact Delay-Insensitive Asynchronous NoC Router", *Computers, IEEE Transactions on*, vol. 63, no. 3, pp. 637–649, 2014.

[94] D Rostislav, V Vishnyakov, E Friedman, and R Ginosar, "An asynchronous router for multiple service levels networks on chip", in *Asynchronous Circuits and Systems, 2005. ASYNC 2005. Proceedings. 11th IEEE International Symposium on*, 2005, pp. 44–53, ISBN: 1522-8681.

[95] I. M. Nedelchev and C. R. Jesshope, "Basic building blocks for asynchronous packet routers", in *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV '94, Proceedings., Fourth Great Lakes Symposium on*, 1994, pp. 184–187.

[96] M Davies, A Lines, J Dama, A Gravel, R Southworth, G Dimou, and P Beerel, "A 72-Port 10G Ethernet Switch/Router Using Quasi-Delay-Insensitive Asynchronous Design", in *Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on*, 2014, pp. 103–104, ISBN: 1522-8681.

[97] M. V. Joshi, S. Gosavi, V. Jegadeesan, A. Basu, S. Jaiswal, W. K. Al-Assadi, and S. C. Smith, "Ncl implementation of dual-rail 2s complement 8x8 booth multiplier using static and semi-static primitives", in *2007 IEEE Region 5 Technical Conference*, Apr. 2007, pp. 59–64.

[98] I. P. Dugganapally, W. K. Al-Assadi, T. Tammina, and S. Smith, "Design and implementation of fpga configuration logic block using asynchronous static ncl", in *2008 IEEE Region 5 Conference*, Apr. 2008, pp. 1–6.

[99] M. T. Moreira, C. H. M. Oliveira, R. C. Porto, and N. L. V. Calazans, "NCL+: Return-to-one Null Convention Logic", in *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, 2013, pp. 836–839, ISBN: 1548-3746.

[100] A. Przybylski, K. Haque, and P. Beckett, "The bel array: An asynchronous fine-grained co-processor for dsp", in *2016 10th International Conference on Signal Processing and Communication Systems (ICSPCS)*, Dec. 2016, pp. 1–7.

[101] A. Caberos, S. Huang, and F. Cheng, "Area-efficient cmos implementation of ncl gates for xor-and/or dominated circuits", in *2017 IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia)*, Oct. 2017, pp. 37–40.

[102] P. Dabholkar, R. Sovani, and P. Beckett, "A low latency asynchronous Jenkins hash engine for IP lookup", *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2016-July, pp. 2663–2666, 2016.

[103] *Packet Clearing House (PCH) Routing data daily snapshots*.

[104] P. Koopman, "32-Bit Cyclic Redundancy Codes for Internet Applications", *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 459–468, 2002.

[105] H Fadishei, M Saeedi, and M. S. Zamani, "A fast IP routing lookup architecture for multi-gigabit switching routers based on reconfigurable systems", *Microprocessors and Microsystems*, vol. 32, no. 4, pp. 223–233, 2008.

[106] A. Kirsch and M. Mitzenmacher, "Less Hashing , Same Performance : Building a Better Bloom Filter", *Wiley InterScience Journal*, vol. 33, no. 2, pp. 456–467, 2006.

[107] Cypress Semiconductors, "Parallel Cyclic Redundancy Check (CRC) for HOTLink", Cypress Semiconductors, Tech. Rep. March, 1994, pp. 1–7.

[108] *Wave Computing*.

[109] J. Kim, M. M. Kim, and P. Beckett, "Static leakage control in null convention logic standard cells in 28 nm UTBB-FDSOI CMOS", in *ISOCC 2015 - International SoC Design Conference: SoC for Internet of Everything (IoE)*, 2016, pp. 99–100, ISBN: 9781467393089.

[110] M. Moreira, P. Beerel, M. L. L. Sartori, and N. Calazans, "NCL Synthesis With Conventional EDA Tools: Technology Mapping and Optimization", *IEEE Transactions on Circuits and Systems I: Regular Papers*, no. 1, pp. 1–13, 2018.

[111] F. A. Parsan and S. C. Smith, "CMOS implementation comparison of NCL gates", in *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, 2012, pp. 394–397, ISBN: 1548-3746.

[112] W.-g. Ho, K.-s. Chong, B.-h. Gwee, J. S. Chang, and M.-f. Yee, "A Power-Efficient Integrated Input / Output Completion Detection Circuit for Asynchronous-Logic", pp. 376–379, 2011.

[113] M. M. Kim, J. Kim, and P. Beckett, "Area Performance Tradeoffs in NCL Multipliers using Two-Dimensional Pipelining", in *International SOC Design Conference*, 2015, pp. 125–126, ISBN: 9781467393089.

[114] Y. Tong, X. Gaogang, D. Ruian, S. Xianda, and K Salamatian, "Towards practical use of Bloom Filter based IP lookup in operational network", in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, 2014, pp. 1–4.