University of Alberta

LOW POWER, HIGH THROUGHPUT INTERNET ROUTING TABLE LOOKUP USING MULTI-BIT TRIES OF SRAM

by

Jason W. Klaus 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2008

Your file Votre référence
ISBN: 978-0-494-55122-6
Our file Notre référence
ISBN: 978-0-494-55122-6

In loving memory of my father...
who taught me to dream in binary

Abstract

A solution to the Internet routing table lookup problem for "backbone" routers is presented. It leverages the power efficiency and low cycle times of Static Random Access Memory (SRAM) to construct a pipelined fixed-stride multi-bit hardware trie on a proposed chip, with very low power consumption. The parallelism in the design proves to be far more scalable than the industry standard Ternary Content Addressable Memory (TCAM), requiring 16.7% of the TCAM chip area and 1.1% of the power, per lookup, for the same number of lookups per second. Unlike other hardware trie implementations, the design also boasts well bounded worst case update times.

Acknowledgements

I would like to thank my professor, Dr. Duncan Elliott, for his guidance, instruction and support. I would also like to thank my colleagues for their suggestions and assistance: Tyler Brandon, John Koob and Leendert van den Berg.

This research was funded by Alberta Ingenuity, the Alberta Informatics Circle of Research Excellence (iCORE), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the University of Alberta.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AS	Autonomous System
ASIC	Application Specific Integrated Circuit
BCAM	Binary Content Addressable Memory
BGP	Border Gateway Protocol
CAM	Content Addressable Memory
CIDR	Classless Inter-Domain Routing
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
IC	Integrated Circuit
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
LUT	Look-Up Table
NAT	Network Address Translator
NNTP	Network News Transfer Protocol
SMTP	Simple Mail Transfer Protocol
SOC	System On a Chip
SRAM	Static Random Access Memory
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction to Internet Routing

This thesis presents a solution to the routing table lookup problem. This chapter describes the problem in detail by first explaining the Internet Protocol (IP), then how computers are addressed and organized into networks, and finally how routers forward packets between computers. The chapter also describes how the current IP version 4 (IPv4) address space grew, the problem of its eventual exhaustion, the motivation for adopting the new IP version 6 (IPv6), and the relevance of solutions to IPv4 routing table lookups to IPv6 routing table lookups. Finally an overview of the proposed solution is presented. A history of the Internet and the IP can be found in appendix A and should be read first by anyone not familiar with them.

1.1 Internet Protocol (IP)

The Internet Protocol (IP) defines the most fundamental details about how computers communicate over a network. Each computer's interface on the network is assigned a different and unique IP address. To communicate, computers send data to each other in the form of packets. Each packet has a two parts: a header which contains IP information, and a body which contains the actual data. Each IP header contains, among other information, a source and a destination address, corresponding to the computer that sent the packet and the computer that is to receive the packet, respectfully. In IP version 4 (IPv4), the version predominantly used today, IP addresses are four bytes (32 bits) long [53]. IPv4 addresses are often expressed

as four tuples of decimal numbers between 0 and 255 each, such as 192.168.1.1.

For very large IP networks, such as the Internet, it is impractical to have every computer connected to every other computer directly. A common solution to this problem is to group a bunch of local computers together into a smaller network, called a subnet. These computers are all connected to a common router that controls the subnet, which in turn can be connected to other routers. Several subnets are then connected together through a router to form a larger subnet, usually called a site, and so on, until the entire network is connected. Computers on the same subnet are often assigned a contiguous range of IP addresses, such as 192.168.1.1, 192.168.1.2, ... to 192.168.1.15 for a 15 computer network for example. Subnets are often represented as a prefix of the group of IP address of the computers they comprise, which is written as a normal IP address followed by a forward slash and the length of the prefix in bits out of 32. For example, 192.168.1.0/24 is a prefix that represents the 256 IP addresses between 192.168.1.0 and 192.168.1.255 inclusive.

To send a packet to another computer, a computer first forwards the packet to its local router, which analyzes the packets' destination IP address. If the packet is destined for a computer directly connected to the router it forwards it to that computer. If not, then the router looks at a set of rules to determine where the packet should go. Often the router will forward the packet to another router it thinks is closer to the destination computer. That router then uses its own rules to determine where the packet should go, and so on and so on, until the packet is ultimately forwarded to its destination. A slightly more complicated example of this can be seen in Figure 1.1.

1.2 Routing Table Lookups

As mentioned in the previous section, routers forward the packets they receive to their destinations. They do this by keeping routing tables that contain rules that match various fields in a packet's header (most commonly the packet's destination IP address) to output ports on the router. A router's ports might be connected to

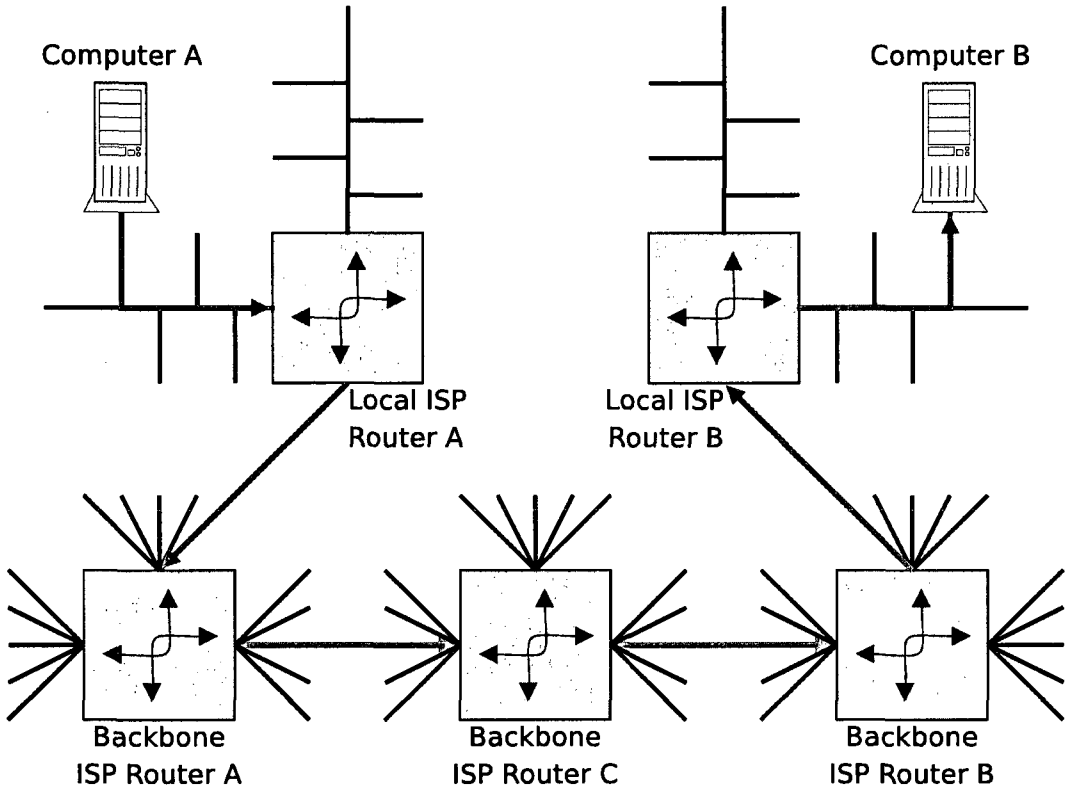


Figure 1.1: Internet Routing Example: Computer A sends a packet to Computer B on a different Internet Service Provider (ISP)

computers or even other routers. Rules that match on the destination IP address are expressed as IP address prefixes representing wildcarded ranges of IP addresses. If an incoming packet's destination IP address falls under the range of one of these address prefixes it is forwarded to the port indicated by that rule. If more than one prefix is a match for a particular destination IP address then the longest matching prefix is considered to be the best match and that rule used. For example, if a router's lookup table contains the address prefixes listed in Table 1.1, and it received a packet with destination IP address 192.168.5.2, then it would be forwarded to port 7, not port 12, since 192.168.5.0/24 is a longer matching prefix than 192.168.0.0/16.

Routing tables are also dynamic. As new computers, routers and subnets are added to the Internet, new entries must be added. Likewise, if old computers, routers or subnets leave the Internet, some entries must be removed. More com-

IP Address Prefix	Destination Port
192.168.0.0/16	Port 12
192.168.5.0/24	Port 7
192.169.0.0/16	Port 14

Table 1.1: Example Routing Table Lookup

Router Application	Lookup Table Entries	Packets Per Second	Table Updates
Small Subnet	10	100	1 per day
Small ISP	100	10,000	1 per minute
Medium ISP	1000	1,000,000	1 per second
"Backbone"	250,000	1,260,000,000	100 per second

Table 1.2: Different Router Requirements

monly, hardware failure, power outages or any number of reasons may cause a computer, router or subnet to temporarily leave the Internet, only to return later. In all these cases the routing tables must adapt to ensure packets reach their destinations whenever possible.

Routing requirements vary considerably depending on what role the router plays. A router that administers a small subnet of only a handful of computers might have only a handful of routing table entries, process only hundreds of packets a second and rarely need updating. In stark contrast, a so called "backbone" router that connects the largest Internet sites together can have 250 thousand routing table entries [27], processes 1.26 billion packets per second[1], and requires updating 100 times a second [22]. This disparity in routing requirements is summarized in Table 1.2. Clearly, designing solutions for routing table lookups for "backbone" routers is a considerable challenge.

1.3 IPv4 Address Space Growth and Exhaustion

When the Internet first began it was envisioned that only a handful of governments, universities and corporations around the world would ever utilize it. After all, these large organizations were the only ones that could afford to have computers and pay

[1]Current backbone routers support 1.2 Tbps switching [65]. Assuming an average packet size of 1000 bits [22]: $1.2Tb/s \times 1024Gb/Tb \div 1000bits/packet = 1.17Gp/s$

the enormous communications costs. To this end, the IPv4 32bit address space was allocated into three different classes to correspond roughly with the sizes of subnets that the Internet would comprise. Class A allotments were the largest, assigning a single 8 bit prefix to an organization for a total of $2^{24} = 16,777,216$ IP addresses. Class B allotments were the middle ground, assigning a single 16 bit prefix to an organization for a total of $2^{16} = 65,536$ IP addresses. Class C allotments were the smallest, assigning a single 24 bit prefix to an organization for a total of $2^8 = 256$ IP addresses [19].

This prediction didn't hold, however. Advances in technology soon brought computer and bandwidth prices crashing down, making it far more affordable to be on the Internet. Many companies started seeing the Internet as a new frontier for business and were eager to get connected. The number of organizations requesting class B allotments sky-rocketed, threatening to exhaust that class of addresses. In an effort to slow down this trend and provide a short term solution, Classless Inter-Domain Routing (CIDR) was introduced. CIDR abolished the three class system of address allocation, allowing organizations to be allocated any address prefix length that they required. Since many organizations needed more than 256 IP addresses (former class C) but far less than 65,536 IP addresses (former class B) several such organizations could split the space formerly reserved for one class B prefix instead of each requiring their own. An unfortunate side effect of CIDR was a complication in "backbone" routing table lookups, as subnet address prefixes could now be any length instead of falling into one of three classes, dramatically increasing the difficulty of the longest prefix matching problem.

Around the same time as CIDR a new specification emerged called Network Address Translator (NAT) [62]. NAT describes how a subnet of computers can use private IP addresses for communication amongst themselves, but then bind to globally unique IP addresses for connections to computers outside the subnet. This allows many organizations to reuse the same IP address ranges internally (often ranges that are not allowed to be used globally) while retaining a much smaller set of globally unique IP addresses for whenever external connections are required. The

end result is that organizations can make due with much smaller global IP address range allotments, further delaying the effects of IPv4 address space exhaustion.

1.4 Introduction to IP version 6 (IPv6)

While both CIDR and NAT greatly slowed the rate of IPv4 address space exhaustion, they are not long term solutions [19]. To solve the problem once and for all, the IP version 6 (IPv6) specification is being slowly adopted [15]. IPv6 calls for 128 bit IP addresses, and in its current form, reintroduces at least a partial partition to the address space once more [24]. The lower 64 bits of each IPv6 address represents a universal identifier for the specific device. The upper 64 bits of each IPv6 address designate the particular subnet the address belongs to, being split between a global IPv6 address prefix for the site the subnet resides in and an ID for the subnet within the current site. The exception to this rule is the section of IPv6 address space allocated to mapping to the old IPv4 address space in the name of backward compatibility.

At first glance it looks as though the requirements of IPv6 routing tables are significantly different than those of IPv4 routing tables. While the IPv4 address space is very densely allocated into prefixes of varying lengths, the IPv6 address space will be very sparsely allocated into prefixes of a few standard lengths. This calls into question the benefit of designing solutions to the IPv4 routing table lookup problem, as not only is IPv4 being phased out in favor of IPv6, but solutions for IPv4 may not readily also apply to IPv6. However, upon closer inspection, it is clear that IPv4 solutions to the routing table lookup problem are indeed still valuable for "backbone" routers. While preliminary versions of the IPv6 specification have been around for many years now adoption has been very slow; many newer devices are designed to support both IPv4 and IPv6, but large amounts of legacy IPv4 only software and hardware still remain. Furthermore, even if a push is eventually made to switch over the majority of traffic to IPv6 networks the IPv4 routing table will still need to remain part of the IPv6 routing table for many years after for backwards compatibility. Finally, since only a portion of the upper 64 bits of an IPv6 address

will be used to denote a site's prefix, it is envisioned that "backbone" routing tables will contain prefixes at most 50 bits long; far closer to existing 32 bit prefixes than initially expected for 128 bit IPv6 addresses.

1.5 The Proposed Solution

This thesis presents a solution to the routing table lookup problem for "backbone" routers. The proposed design offers far better scalability than previous solutions, boasting an average of 53.7 billion lookups/second, with only $7.85W$ of total power consumption, and a chip area of $71.1mm^2$. While the solution is presented for IPv4, it still remains relevant for IPv6.

The rest of the thesis is organized as follows: Chapter 2 summarizes all of the previously published solutions to the routing table lookup problem. Chapter 3 provides the motivation for this work and describes the entire design of the solution, including how lookups and updates are handled. Chapter 4 describes how the design was validated, including the details of the Field Programmable Gate Array (FPGA) implementation. Chapter 5 develops several metrics for comparing stride choices for the design, applies them to real "backbone" routing tables, then determines the performance of the design with the preferred strides. Finally, Chapter 6 develops a model for Ternary Content Addressable Memory (TCAM) to determine the performance of a comparable solution to the routing table lookup problem in TCAM, then compares it with the proposed design.

Chapter 2

Previous Work in Routing Table Lookups

Designing solutions to the Internet routing table lookup problem has been an active area of research for many years. Published solutions can be divided into two main categories: software solutions and hardware solutions. Of the software solutions, a large number employ a trie data structure, which is of particular interest to the author. Of the hardware solutions, the majority favor Ternary Content Addressable Memory (TCAM), which is the current industry standard.

2.1 Software Approaches

In general software implementations of routing table lookups define some sort of data structure to store the routing information and explain how lookups and updates can be performed on it. Software implementations generally assume they will be run on commodity CPUs with standard memory and caching architectures. The authors of software implementations compare their work based on the total amount of memory required, and on the number of memory accesses, actual CPU time or a bound on asymptotic time required for lookups and updates.

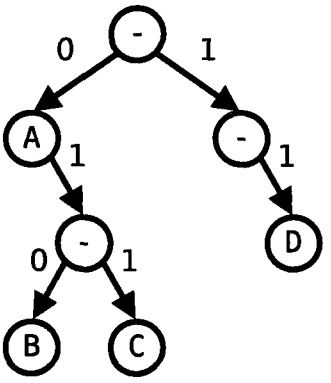
Software implementations are limited by the general purpose CPUs and memory they run on. While these continue to evolve and keep an attractive price point, they cannot keep pace with the every increasing demands on "backbone" routers, whose users themselves also get more powerful CPUs, demand larger amounts of

bandwidth and continue to increase in number. Since software solutions are also serial by nature, they cannot take advantage of hardware pipelining or parallelization to help close the performance gap. Moreover, specialized hardware can often be tailored for smaller areas and reduced power consumption when compared to general purpose hardware, further disadvantaging software solutions.

2.1.1 Background on Software Tries

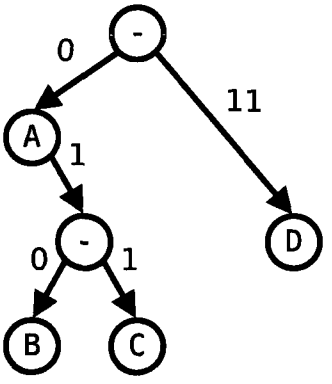
A basic binary trie (from re*trie*val) is a tree data structure similar to a binary tree, where each node stores data and has up to two child nodes: a left child and a right child. Unlike binary trees, however, binary tries always store data for a given key in a specific node in the trie. The root node of the trie corresponds to the zero length key. Its left and right children correspond to the one digit keys 0 and 1 respectfully. Similarly, the left and right children of the 0 keyed node correspond to the two digit keys 00 and 01 respectfully, and so on. When a new key-data pair is inserted into the trie, any missing parent nodes of the new node are added in with NULL data. An example of a simple binary trie can be seen in Figure 2.1. A search of a trie for a particular key starts at the root, and a comparison of the first (most significant) bit of the key. If the bit is a 0 then the search continues down the left child, otherwise it continues down the right child. This process continues until either all the bits of the search key are consumed (indicating the correct node has been found) or the indicated child node is absent from the trie (indicating the key is not in the trie). If the correct node has NULL data then the key is also not in the trie, otherwise there is an exact match.

A basic binary trie can be easily transformed into a binary Patricia trie by compressing paths traversing nodes with NULL data and only a single child. Figure 2.2 shows the binary Patricia obtained by transforming the example basic binary trie from Figure 2.1. The right child of the root node, corresponding to the 1 key with NULL data, has been compressed so that the 11 key becomes the new right child. The left child of the root node, corresponding to the 0 key, cannot be similarly compressed because while it has only one child it also stores the data value



Key	Data
0	A
010	B
011	C
11	D

Figure 2.1: Example Basic Binary Trie

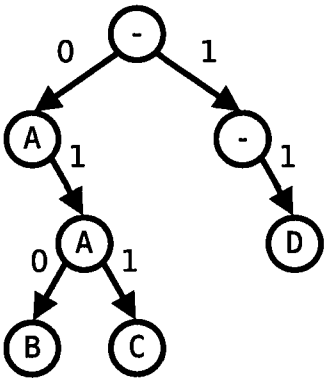


Key	Data
0	A
010	B
011	C
11	D

Figure 2.2: Example Binary Patricia Trie

"A". Likewise the right child of that node, corresponding to the 01 key, cannot be compressed because while it stores a NULL data value it has two child nodes. Patricia tries often reduce the number of nodes required compared to basic tries at the cost of additional complexity in storing the compressed paths.

A basic binary trie can also easily be adapted to store prefixes, and hence solve the routing table lookup problem. Instead of key-data pairs we now have prefix-port pairs, associating a given address prefix with the appropriate router output port. Shorter length prefixes are treated exactly like their shorter length key counterparts, being stored closer to the root than longer length prefixes. A parent node also propagates its own port value to all nodes below it that have NULL port values, filling in for more specific prefixes missing from the trie. An example of a simple prefix trie can be seen in Figure 2.3. Searching a prefix trie is similar to searching a binary trie, only all search keys are maximal length, and when a search stops



Prefix	Port
000/1	A
010/3	B
011/3	C
110/2	D

Figure 2.3: Example Basic Prefix Trie

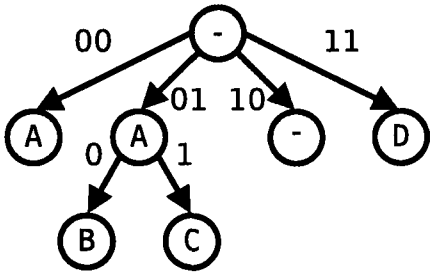
(either due to a non-existent child or consuming all the search key bits) the last node traversed has the longest matching prefix's port number stored in it.

A binary prefix trie can also be further extended into a multi-bit prefix trie. Instead of examining just a single bit at each level, multiple bits are examined. The number of bits examined at a time is called the stride. If a stride of x bits is examined at a given level, then the node at that level of the trie has 2^x children corresponding to the 2^x possible values those bits can have. A prefix that covers several children of a given node is expanded and its port number is stored in every child node covered by that prefix, unless of course a more specific prefix also covers the child. Multi-bit tries that always divide up an address into the same strides are called fixed-stride tries, where as those that vary the strides based on the contents of the trie are called variable-stride tries. An example of a fixed-stride $\{2,1\}$ prefix trie (a prefix trie whose first stride is 2 bits and second stride is 1 bit) is shown in Figure 2.4. While multi-bit tries are generally less efficient in storage than their binary counterparts, they often make up for it by being more efficient in processing operations.

2.1.2 Software Trie Approaches

The following section summarizes the previously published solutions to the routing table lookup problem that make use of software tries.

Chiueh and Pradhan [13] presented a modified fixed-stride $\{16,8,8\}$ trie combined with a caching scheme. It depends on the spatial and temporal locality of



Prefix	Port
000/1	A
010/3	B
011/3	C
110/2	D

Figure 2.4: Example Fixed-Stride $\{2, 1\}$ Prefix Trie

lookups for good performance, and it is unclear how updates are handled.

Nilsson and Karlsson [46] designed a Level Compressed (LC) trie which combines a variable-stride trie with a skip function to bypass sparse sections. The skip function reduces the memory required to store the trie and reduces the number of nodes that have to be analyzed per lookup. Its perceived benefit is diminished, however, by the requirement to store the original prefixes separately and compare against them with each skip to ensure that the skipped bits match the prefix. It also further complicates an already complex update process for variable-stride tries.

Lampson et al. [34] proposed a fixed-stride $\{16, 16\}$ trie where each second level lookup-table is replaced by sorted prefix start and end values that are binary searched to find the longest matching prefix. Unfortunately, this means the worst case number of memory accesses depends on the size of the lookup table, and updates may require moving many such values to keep them sorted.

Kijkanjanarat and Chao [30] presented two tries: one indexed by the start of the IP address being looked up and the other indexed by the end of the address. These two tries share leaf nodes, which somewhat reduces the number of nodes required compared to a single trie at the cost of increased complexity and memory accesses per lookup and update.

Yilmaz et al. [82] described Linked list Cascade Addressable Trie (LCAT) which uses a lot of extra memory to be able to rebuild the original prefixes and simplify updates. Of particular interest is their proposed default route for each node in the trie, which ensures update time is well bounded in the worst case. This idea has been incorporated into the design in this thesis.

Kobayashi and Murase [31] stored prefixes in an $\{8,8,8,8\}$ multi-bit trie, compressing some nodes using either an Index method or the Candidate Prefix Table (CPT) method. The Index method stores the multi-bit node's data as a compressed bitmap. The CPT method replaces one or more multi-bit nodes with a list of the prefixes they represent. While these compression schemes save memory they complicate lookups and especially updates, as adding or removing prefixes might require changing the type of compression used on a node.

Pak and Bahk [49] significantly improved over the Patricia trie by removing backtracking. Single bit compares are replaced by full prefix compares, and an 8 bit front index table is used to speed up lookups. They achieved better performance and memory utilization with fairly straightforward lookups and updates.

Wuu and Pin [78] stored prefixes in a $\{8,8,8,8\}$ multi-bit trie using bitmap compression on the nodes in the last two strides. The compression, while reducing memory consumption, complicates lookups and makes updates prohibitively expensive.

Oh and Ahn [47] created Bit-Map (BM) trie which transforms the routing table into a bit-map where a one in a particular position denotes a prefix covering that range. The bit-map is compressed by only keeping track of how many ones have been encountered up to a certain position. Lookups require few memory accesses to retrieve the compressed information at the cost of up to 76 additions and shifts, per lookup, to decode it.

Ahmand and Mahapatra [2] designed M-trie, a logic minimization structure where prefixes are stored in a ternary trie where each node has entries for 0, 1 or X (don't care). M-trie is very efficient for adding and removing prefixes, optimizing the stored prefixes as needed. Unfortunately, M-trie is not practical for lookups, with the authors suggesting to store the optimized prefixes in a TCAM. It is not clear, however, how incremental updates to the M-trie can be easily synchronized to a TCAM.

Most of the previously published software trie solutions to the routing table lookup problem trade more complex lookup and update procedures for reducing

the amount of memory needed to store the trie. In any event none of them can scale to the throughput required for "backbone" routing table lookups.

2.1.3 Other Software Approaches

The following section summarizes the previously published software solutions to the routing table lookup problem that do not make use of tries.

Doeringer et al. [18] presented a trie-like structure called Dynamic Prefix Trie (DP-Trie) where lookups are done using bit comparisons down the trie to locate a leaf then back up towards the root to find the longest matching prefix. DP-Trie provides good average-case lookup performance and deterministic tries after updates at the cost of complex nodes and operations on them.

Yazdani and Min [81] described how to store prefixes in a tree where each node has up to some M number of children. In this tree less specific prefixes are always stored above the more specific prefixes they contain. Lookups consist of searching down the tree until a leaf is found, keeping track of the last matching prefix. Unfortunately, keeping the tree somewhat balanced is difficult for updates. The worst case lookup time is also still the number of bits in an IP address for the case where a length 32 prefix has every possible parent prefix added to the tree as well.

Wang et al. [73] transformed prefixes into address ranges that are sorted and stored in a table. An initial table looks up the first 16 bits of an address to identify which ranges should be binary searched to locate the range that contains the address. Unfortunately, updates require rebuilding the entire table, and the approach doesn't scale well with the number of prefixes.

Berger [8] stored prefixes in a binary tree where each node is a prefix. Lookups mostly require log(N) memory access since updates occasionally try to re-balance the tree, but this is not guaranteed. The authors also propose some skip and indexing functions similar to what is used in tries but do not discuss their costs and benefits.

Bian and Khatri [9] employed an adaptation of the Expresso-MV algorithm to compress the routing table. Unfortunately, updates require a complete rebuild of the routing table. The authors also erroneously assume that IP address lookups not

matching any prefixes don't occur and optimize for this, when in fact packets with addresses not matching any prefixes are supposed to be dropped.

Futamura et al. [20] proposed two different algorithms each with their own benefits and limitations. The first, "Elevator - Stairs", builds a Patricia-trie out of the prefixes and creates a hash table to skip a certain number of levels into the trie for faster lookups at the cost of increased memory usage and more complicated updates. The second, "logW - Elevators", builds several such hash tables for various levels to always complete a lookup within a number of steps proportional to the logarithm of the length of the lookup addresses. It uses more memory and further complicates updates but has slightly better lookup performance.

Lim et al. [38] sorted and stored the prefixes based first on how many other prefixes encompass them, then by the value of their bits. A separate lookup table on the first 8 bits of an IP address determines where in the main table to start binary searching prefixes and how many to search. Once a matching prefix is found it may also have a pointer to the location of its encompassed prefixes which are similarly binary searched, and so on, until the last matching prefix, and hence the longest, is found. With large numbers of prefixes this scheme might require a large number of memory accesses per lookup, and updates to the main table are difficult unless a lot of empty entries are left for additions.

Wuu et al. [79] constructed a heap of all the prefixes in the form of a tree with the longest prefixes closest to the root of the tree. While lookups stop immediately once a matching prefix is found, this approach speeds up a few lookups at the cost of slowing down most of the others. The authors also proposed storing two prefixes per node in some cases to reduce the number of memory accesses, but at the cost of greater lookup and update complexity.

Dharmapurikar et al. [17] applied Bloom filters to longest prefix matching, resulting in an approach with few average case memory accesses per lookup and reasonable memory size. Unfortunately, updates are complicated and require additional processing and memory. Hasan et al. [23] expanded on Bloomier filters which extend Bloom filters, using extra storage and complexity to remove the possibility

of false positive matches on lookups and hence reduce the number of required memory accesses. Unfortunately, updates require additional processing and occasionally require reconstruction of the tables.

Most of the previously published software solutions to the routing table lookup problem, that don't use tries, also involve complex lookup and update procedures. In any event none of them can scale to the throughput required for "backbone" routing table lookups.

2.2 Hardware Solutions

In general hardware implementations of routing table lookups involve specialized memory architectures for storing routing prefixes and specialized computational hardware for handling lookups and updates. The authors of hardware implementations compare their work based on the total amount of memory or transistors required, as well as the lookup latency, throughput and power consumption.

Hardware implementations often involve custom circuits designed around commodity or even fully custom Integrated Circuit (IC) components. Designing, testing and fabricating an Application Specific Integrated Circuit (ASIC) is a very resource intensive proposition, so it's not surprising that many authors instead present their designs along with simulation results and configurable logic implementations such as a Field Programmable Gate Array (FPGA). While often insightful as to the expected properties of the actual ASIC if it were to be built, great care must be taken to ensure that these results are accurate. Moreover, since the cost of an ASIC is so high, custom hardware solutions must present a very compelling advantage over existing commodity hardware and software solutions to justify the investment.

2.2.1 Background on TCAM

In general, a Content Addressable Memory (CAM) is a two dimensional array of cells like a traditional memory, such as Static Random Access Memory (SRAM), in that it can be used to store data for retrieval at another time. In addition to read and write operations, a CAM can also perform searches of all of the values it contains, in

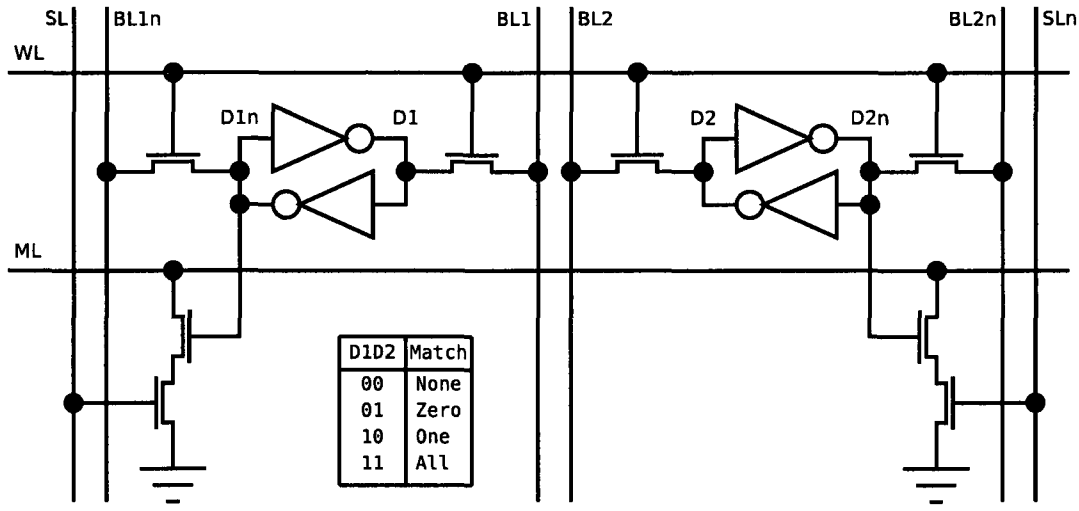


Figure 2.5: TCAM Cell

parallel, to check for matches against a key. Where as a Binary Content Addressable Memory (BCAM) can only look for keys that match its data exactly, a Ternary Content Addressable Memory (TCAM) can store and search against wild-carded entries, which offers far more flexibility. Not surprisingly this means a TCAM can be adapted to storing and searching routing tables prefixes, with shorter prefixes padded with don't care bits on their ends.

A standard TCAM cell is composed of two SRAM cells and additional match line circuitry. A transistor diagram of a TCAM cell is shown in Figure 2.5. Depending on the two bits stored in the cell, it can match nothing, a zero, a one, or both a zero and a one (wild-card). Several TCAM cells are generally connected together to form a word, sharing a common match line. To search, this match line is first pre-charged to V_{dd}, then the search data is applied to the search lines. Search data can also be any combination of ones, zeros, wild-cards and match only wild-cards. If one or more cells in a TCAM word mismatch the search data then they discharge the match line. If the match line of a TCAM word remains at V_{dd} then the word has matched the search. An entire TCAM is composed of many TCAM words, all of which are searched in parallel to look for matches.

In the case of Internet routing table lookups, the entries stored in a TCAM are address prefixes ending in wild-cards, and the search data is binary IP addresses.

A given IP address may match several prefixes, but the longest matching prefix is required. To that end the prefixes in the TCAM must be priority encoded such that the only match reported, if any, is the longest matching prefix. The index of the matching entry is then used to access a small standard memory to retrieve that entry's destination port number.

2.2.2 TCAM Approaches

TCAM solutions to the Internet routing table lookup problem are very popular in industry for a reason. A TCAM is very flexible and well suited to pattern matching. Storing each prefix as an entry is simple and makes it easy to gauge how close the TCAM is to being full. Storing extra information like quality of service metrics or expanding to IPv6 support is straightforward if the word size of the TCAM is increased. On the downside, TCAM cells are complicated compared to SRAM cells. Increasing the TCAM word size means increasing the match line capacitance, and increasing the number of entries in a TCAM increases the search and bit line capacitances. These higher capacitances mean a larger TCAM has dramatically slower performance and increased power consumption compared to a similar sized SRAM. Moreover, the full parallel search nature of a TCAM means significant power is expended with each search and it is impossible to conduct multiple searches of the same TCAM in parallel. A standard TCAM therefore does not scale well to the sizes demanded by "backbone" routers, prompting much research into improvements. Similar interest exists in coming up with efficient priority encoding schemes that make updating easy while keeping a TCAM memory efficient.

Pei and Zukowski [51] applied tries to the general routing problem (before the Internet was prevalent), observing that a single trie could be divided into several tries searched in parallel, with CAM being the fully parallel case. They found that in the worst case the CAM solution was superior in terms of speed and layout except where the address space is large and fully utilized. The CAM solution did, however, consume a lot more power.

McAuley and Francis [42] compared RAM, BCAM and TCAM implementations of generalized routing tables for three different types of addressing. They presented three BCAM and TCAM organizations: single cycle per lookup using a single memory, multiple cycles per lookup using a single memory, and single cycle per lookup using multiple memories. The single cycle per lookup, multiple memories approach assigns a unique priority to each memory to make prioritizing matches easier.

Kobayashi et al. [32] created a Vertical Logic operation with Mask encoded Prefix length (VLMP) for determining the longest matching prefix in TCAM. It removes the requirement to keep TCAM entries sorted at the cost of extra hardware and power consumption, and the approach doesn't scale well to large TCAMs.

Shah and Gupta [61] presented the Prefix-Length Ordering (PLO_OPT) and Chain-Ancestor Ordering (CAO_OPT) constraining algorithms that keep an external trie to calculate the swaps required to add a new prefix (or remove an existing prefix) from a TCAM while preserving an ordering for longest prefix matching. This approach greatly reduces the number of prefixes moved in the worst case without adding additional TCAM complexity at the cost of external processing.

Liu [41] described two methods of compressing the prefixes stored in a TCAM: pruning and mask extension. Unfortunately, both methods would require external processing of updates to re-compact the existing prefixes.

Arsovski et al. [6] designed a 12 transistor TCAM cell with asymmetric 4 transistor SRAM cells that uses a current-race sensing scheme where match lines are first grounded and then current is injected to drive them high. A reference full match line is used to determine which match lines have mismatches preventing them from charging as quickly. This approach saves power by only requiring matches to rise to half the high state voltage and by only needing to discharge uncommon full matches as opposed to pre-charging common mismatches.

Zane et al. [83] considered two different optimizations to TCAM. The first used a hashing function on lookup IP addresses to select a small subset of many TCAM blocks to search in parallel for a match. This approach significantly reduces search

power consumption provided the hashing function creates a good prefix distribution amongst the blocks, which can degrade as updates are made. The second optimization involves storing the prefixes in a trie, then dividing up the sub-tries amongst all the TCAM blocks with a small index TCAM to identify which need to be searched on a match. This approach again offers significant power savings at the cost of complicated updates that must occasionally re-partition the sub-tries at great expense.

Gamache et al. [21] presented a custom 512 bit matching, 512 bit storage TCAM with 168 blocks of 128 entries each. Searches are pipelined to first find a longest match in each block, then combine these results to determine the global longest match, then read the storage information of that match. Their TCAM also makes use of partitioned match lines to save power, and custom 9-phase wired-NOR logic for determining the longest matching prefix. The authors estimate their 21,504 entry TCAM would have a $18mm \times 18mm$ die size, a $200MHz$ pipelined lookup throughput and a power consumption of around $16W$ in a $0.1\mu m$ process. These estimated features provide a good baseline for what was possible for TCAM designs in the year 2003.

Kocak and Basci [33] divided prefixes into two or more TCAMs based on prefix length. Lookups consist of first searching the TCAM with the longest prefixes for a match, and only searching the TCAM with the next longest prefixes in the event of a mismatch. This approach saves power since not all lookups require searching all TCAMs, but its benefit is limited for backbone routers since a large percentage of the prefixes are all of length 24.

Pagiamtzis and Sheikholeslami [48] demonstrated two optimizations to TCAM to reduce power consumption. The first is to break up large match lines into segments where previous segments disable subsequent segments in the case of mismatch. The second is to amplify search data from small voltage swing global search lines to large swing local search lines.

Akhbarizadeh et al. [5] created Prefix Content Addressable Memory (PCAM), a custom TCAM cell for longest prefix matching that uses 22 percent less area with somewhat slower performance and increased power consumption.

Wu et al. [76] sorted the prefixes stored in a TCAM into different levels with empty entries between each group. Updates require external processing, with the average case requiring one or two entry changes, but in the worst case it can take much more than that. The authors also don't address what happens when the free space between groups gets filled up by updates.

Akhbarizadeh et al. [4] divided the prefixes into two groups: the disjoint set of prefixes that don't encompass any other prefixes, and the remaining prefixes that do encompass at least one other prefix. Since the prefixes in the first group don't overlap with each other at most one will match a given lookup IP address, eliminating the need for longest prefix match logic. In most cases there are far more prefixes in the first group so the number of TCAM entries requiring longest prefix match logic is significantly reduce. Unfortunately, updates are complicated by the need to distinguish between the two groups and the favorable distribution of prefixes between the two groups is not guaranteed.

Pao [50] observed that storing IPv4 and IPv6 addresses in 144bit or larger sized TCAM entries is wasteful since most matches occur on the first part of the address. He split address entries into two different partitions searched one after another, where often times the first search was all that was required to determine a match. This eliminated the need to even load the second part of the search address in such cases, increasing the lookup throughput especially in low pin count devices.

Kasnavi et al. [28] created a Hardware-based Longest Prefix Matching (HLPM) where TCAM entries are divided into four stages with previous stages disabling subsequent stages for each match to save power. Each entry also contains a special length field used by custom logic to determine the longest prefix match, removing the requirement for managing the order of entries in the TCAM.

Wu et al. [77] divided the prefixes into three groups: those that encompass other prefixes but are not encompassed themselves, those that are encompassed by other prefixes, and those that neither encompass other prefixes or are encompassed themselves. Lookups involve accessing an index TCAM (without longest prefix match) that stores the first group of prefixes. In the case of a match, the appropriate

TCAM (with longest prefix match) storing the contained prefixes of the matching prefix is accessed. In the case of a mismatch by the index TCAM a TCAM (without longest prefix match) containing all the prefixes in the third group is accessed. This approach reduces the power consumed per lookup at the cost of more complicated lookups and updates, and requires all of the prefix relationships to be tracked.

Akhbarizadeh and Nourani [3] designed Multi-Selector and Multi-Block Popular-prefix Table (MSMB-PT) which divides the prefixes into multiple TCAMs accessed by multiple Range Detectors (RDs) doing lookups in parallel. Each RD has a small cache that stores popular prefixes to help reduce contention among the RDs for popular TCAMs. Updates to the MSMB-PT are complicated by the need to keep the prefixes balanced between the TCAMs and to update the RDs as to the location of each prefix. Lin et al. [39] proposed a similar approach using an algorithm applied to a trie construction to determine which prefixes go into which TCAMs. It allows more straightforward distributing of update prefixes to the TCAMs with occasional re-balancing, but still requires external processing and memory.

Chang [12] demonstrated that tree-style AND-type match lines and segmented search lines help reduce TCAM search latency and energy.

Wu and Wang [75] ensured that prefixes were sorted in a TCAM such that a prefix always comes after those prefixes it encompasses, making the first matching prefix of a lookup the longest matching prefix. When a new prefix is added to the TCAM it is swapped with prefixes that encompass it until a prefix that isn't encompassed by any other prefix is obtained, which can be safely stored at the end of the TCAM.

Mohan and Sachdev [45] proposed a new TCAM cell architecture where only a single transistor loads the match line. While this single match line transistor is not fully enabled when a mismatch occurs in the cell, the extra search latency and power consumption this causes is more than offset by the reduced search latency and power consumption of the far less loaded match line. The authors also presented a method for sharing charge from early stages of a segmented match line with future stages to reduce latency and save power.

All of the previously discussed research attempts to improve upon the shortcomings of TCAM. Some researches investigated different cell designs or used pipelining to help reduce power consumption and area. Others presented custom priority encoding hardware for selecting the longest matching prefix, making updates easy. Still others presented advanced update schemes, often requiring external processing and memory, to preserve some prefix orderings to make priority encoding easy. While all of this research has helped alleviate some of the biggest problems with scaling TCAM designs to larger capacities, the parallel search nature of TCAM will always prevent it from increasing throughput and decreasing power consumption at the same fast pace of SRAM based designs.

2.2.3 Background on Hardware Tries

One of the very attractive aspects of software fixed-stride multi-bit prefix tries is that they lend themselves readily to hardware implementations. A node of a multi-bit trie that is indexed by a binary key maps perfectly onto a standard memory. Each entry in the memory can contain either a port number if the result is known, or a pointer into a new memory to continue the search. The next stride of the search IP address is used to index into this new memory, and the process repeats. An example of a fixed-stride $\{4,2,2\}$ hardware prefix trie populated with the prefixes from Table 2.1 is shown in Figure 2.6.

As an example, consider a search of the hardware trie in Figure 2.6 for the 8 bit IP address 01101111 (shown in bold). The first four bits of this address (0110) are used to index into the first memory. The retrieved memory entry is a pointer to the first bank for the next stride, so the search continues. The next two bits of the search address (11) are used to index into this first bank, yielding yet another pointer. The last two bits of the search address (11) are then used to index the first bank of the last stride, yielding port 2, which is the expected answer.

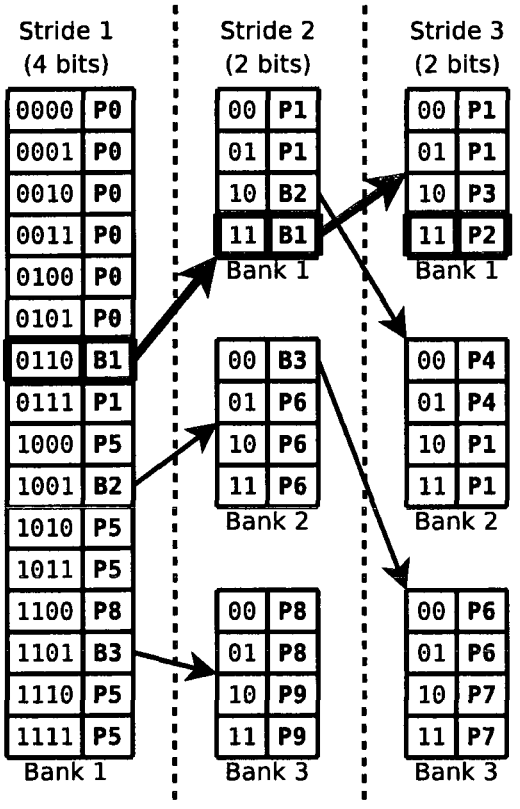


Figure 2.6: Example Fixed-Stride $\{4,2,2\}$ Hardware Prefix Trie

Prefix	Port	Prefix	Port
00000000/1	0	10000000/1	5
01100000/3	1	10010000/4	6
01101110/7	2	10010010/7	7
01101110/8	3	11000000/3	8
01101000/7	4	11011000/5	9

Table 2.1: Example 8 bit IP address prefixes

2.2.4 Hardware Trie Approaches

Hardware tries offer an interesting alternative to TCAM by exploiting the fact that SRAM is cheaper, smaller, faster and less power consuming than TCAM. By carefully choosing the strides to partition an IP address, hardware tries can offer fairly efficient storage of a large number of prefixes, especially in densely packed address spaces. This proves very promising as efforts to conserve IPv4 address space is resulting in very densely packed clusters of prefixes in "backbone" routing tables.

Unfortunately, in the worst case, tries can be extremely inefficient ways of storing address prefixes, especially in very sparse address spaces. This not only provides a challenge for the upcoming switch to IPv6, but also in allowing for future expansion of existing routing tables. Care must be taken in assuring that a good stride choice for one routing table isn't also a poor choice for another lookup table, or won't become a poor choice with future prefixes being added to the current table. Another problem hardware tries share with their software brethren is that they are often difficult to update, with incredibly long update times required in the worst case.

Gupta et al. [22] discussed a fixed-stride $\{24, 8\}$ multi-bit hardware trie using two DRAMs called DIR-24-8-BASIC. They also proposed DIR-24-8-INT which adds a second level of indexing to the second memory to save space at the cost of an extra memory access per lookup and added complexity. They suggested optimal strides for tries of depths 3 to 6 to optimize memory usage, and present a number of different update schemes requiring different amounts of processor and routing table update time. The architecture can be pipelined to handle one lookup per clock cycle, but updates have very large worst case times.

Huang and Zhao [26] designed a fixed-stride $\{16, 16\}$ RAM based trie with compressed second stride lookup tables requiring at most three memory accesses per lookup. The compression, while reducing memory consumption, requires specialized hardware to decode and makes incremental updating of the routing table impractical.

Uga and Shiomoto [69] presented a Patricia trie combined with three CAMs that index all level 8, 16 and 24 nodes in the trie. Each lookup involves searching all three CAMs in parallel to obtain a pointer into the Patricia trie from which to conduct a much shorter search of 9 nodes or less.

Wang et al. [72] proposed a $\{16, 16\}$ multi-bit hardware trie where entries in the second level memory are compressed based on common prefix bits. This technique reduces the required storage but reconstruction of the second memory is required on updates.

Sungkee et al. [64] designed a $\{16,8,8\}$ multi-bit hardware trie with compression. Lookups require up to four memory accesses while additions require anywhere from two to four 64 byte blocks to be regenerated. Unfortunately, additions require external processing and there is no mention of how removals are handled.

Sahni and Kim [58] created a dynamic programming algorithm to calculate the best partitions for a fixed-stride hardware trie for a given set of prefixes in order to minimize memory usage. While it is faster that previous algorithms (but not asymptotically so), it does not consider hardware overheads and assumes a single memory. Sahni and Kim [59] also proposed a similar algorithm for variable-stride hardware tries that was asymptotically faster that other algorithms.

Chang and Lim [11] implemented a fixed-stride multi-bit trie with one SRAM per stride. Each entry has a port number for each possible prefix length ending in that stride, as well as a pointer to the next stride's memory if needed. This approach requires a lot of extra memory to store this information and it somewhat complicates lookups, but makes updates very easy. They proposed a skip function that provides very little memory savings at the cost of added lookup and update complexity. They also proposed a compression scheme for the port information that further complicates lookups. Finally they recommended a $\{14,4,4,4,4,2\}$ partition for the single 40,000 prefix table they analyzed.

Taylor et al. [67] described Fast Internet Protocol Lookup (FIPL), which is a multi-bit trie that stores bitmaps of which entries have corresponding next hop ports and next stride lookup pointers. The corresponding ports and pointers are stored contiguously in a separate location, requiring up to 11 memory accesses per lookup. Updates require external processing and there is no discussion of how fragmentation is handled.

Wang et al. [74] stored prefixes in a compressed trie where leaf nodes are grouped and stored in a larger node structure, and intermediate nodes are grouped and stored in a larger pointer structure. While this reduces the required memory usage it dramatically complicates lookups and updates on the trie.

Qin et al. [54] presented a CAM that is used to match some subset of a lookup IP address' bits to one of many much smaller Patricia tries for further processing. They recommend TCAM over BCAM for more even prefix distributions among the tries, but even good initial prefix distributions might degrade over time with updates.

Wang et al. [71] built the prefixes into a trie, then stored each small sub-trie into fixed-stride multi-bit tries stored in DRAM. The root prefix of each small sub-trie is stored in a TCAM which is longest prefix matched on each lookup to determine which sub-trie to access. Updates consist of adding entries to the TCAM and occasionally regenerating complete sub-tries to reduce the number of entries in the TCAM, which requires external processing.

Almost all of the previously discussed hardware trie solutions attempt to reduce the required memory for storing the routing table. While some achieve some fairly large savings, most of the solutions have fairly complex lookup and update procedures. Furthermore, none of the solutions are presented with adequate investigation into how they perform using a variety of different routing tables; most present results for a single routing table if they present any results at all.

2.2.5 Other Hardware Approaches

Still other hardware solutions to the routing table lookup problem have been published that aren't based completely on TCAM or hardware tries. This subsection outlines these non-conventional hardware approaches.

Hsiao and Jen [25] mapped a routing table to compressed combinational logic and implement it in an FPGA. Similarly Sangireddy and Somani [60] used binary decision diagrams to generate their compressed logic for an FPGA. While these approaches are very FPGA resource efficient it is impossible to update the routing table without reprogramming the FPGA. Reprogramming the FPGA involves regenerating the combinational logic, recreating a bit file based on this new logic, and finally programming the FPGA with this new bit file. This whole process can take hours, requires additional computing resources, and takes the FPGA offline

during each reprogramming operation. An FPGA, while much more economical, is also much slower and consumes more power than an ASIC, and does depend on memory technology for its logic units despite what some authors suggest.

van Lunteren [70] described Balanced Routing Table Search (BaRTS) which consists of a trie-like structure where each stride hashes certain (not necessarily sequential) bit positions. A certain number of additional prefix comparisons must be made at each stride as well; the data for which is stored in a wide memory, requiring only a single memory access per node. He only briefly discussed how to deal with memory fragmentation issues and presented no concrete strategy.

Lin and Chang [40] proposed a TCAM to check for matches against all prefixes with length greater than 24, then a compact IP-routing block to store the remaining prefixes in a compressed form that complicates lookups. Unfortunately, updates to the TCAM and IP-routing block are not discussed despite being non-trivial.

Lim et al. [37] split up all the prefixes by length into different hash tables that are searched in parallel for lookups. Collisions are handled by storing all the colliding prefixes in a binary tree and binary searching the prefixes with each matching lookup. The lookup performance of each hash table depends on the quality of the hashing function which may degrade with updates, and the required size of each hash table is difficult to determine in advance when taking into account updates.

Kaxiras and Keramidas [29] designed IPStash: a set associative memory architecture with a few fixed levels to which all prefixes are mapped. Unfortunately, conflicts among prefixes are possible and cannot be handled by the architecture. Also, increasing the size of IPStash requires increasing its memory's size and associativity, which offers poor scalability.

Mohammadi et al. [44] created Hardware Assisted Software IP Lookup (HASIL), which involves adding three new instructions to a general purpose Central Processing Unit (CPU) to help speed up software lookups using Dynamic M-way Prefix (DMP) trees. It is unclear why only three custom instructions were decided on when a fully custom CPU would improve performance even further.

Ravikumar and Mahapatra [55] described a reconfigurable combinational logic block that analyzes incoming lookups and decides on one or more independently selectable TCAM arrays to continue the search in. This approach dramatically reduces power consumption since only a fraction of the total TCAM is searched, provided that the combinational logic can keep the prefixes well distributed and grouped. Unfortunately, updates are complicated and may require reprogramming of the combinational logic and the moving of many address prefixes. A very similar approach was also proposed by Zheng et al. [84]. Ravikumar et al. [56] replaced the combinational logic with a fixed division that selects exactly one TCAM for each prefix or lookup IP address based on their higher order bits, trading the flexibility of the combinational logic for the simplicity of a known partition.

Lim and Lee [36] proposed an Enhanced Binary Tree (EnBiT) which divides prefixes into a number of different balanced trees based on whether or not they encompass other prefixes. A TCAM is used to decide which sub-tree(s) must be further searched for each lookup. While the author claims that updates are straightforward it is unclear how they could based on what few details are discussed. Tang et al. [66] present a similar approach where the sub-trees are stored as compressed bitmaps in SRAM. How this approach can be updated is not discussed and most likely requires rebuilding substantial portions if not all of the sub-trees and TCAM entries.

Lim and Jung [35] split up prefixes by length into different RAM tables, storing them based on hashing. Any prefixes colliding with existing prefixes were stored in a separate small TCAM. A lookup is done in parallel in each RAM table and the TCAM to determine the longest matching prefix for a given IP address. Unfortunately, this approach relies heavily on the RAM table hashing function distributing prefixes evenly in the face of updates to avoid wasting RAM and requiring a larger TCAM.

Xu et al. [80] used the Comb Extraction Scheme (CES) to split each prefix into two smaller prefixes consisting of its even and odd bits respectively. These two tables are searched in parallel for each lookup and matching entries are compared

to determine which original prefixes completely match. While both of these tables are much smaller than the original table, collisions between prefixes quadratically increase the number of comparisons that must be made for each lookup. While the authors suggest a custom ASIC to handle these comparisons, the number of collisions in larger routing tables would quickly make even this approach impractical.

Mingfeng and Zhenghu [43] divided prefixes into groups based on length, then stored each group's prefixes in BCAMs based on a hashing function. Colliding and non-standard length prefixes are stored in a separate TCAM. Lookup IP addresses are hashed to determine which BCAM in each group they should be matched against and also looked up in the TCAM, with the results being combined to determine the longest matching prefix. This approach's usefulness is highly dependent on the quality of the hash function, which can degrade with new updates to the routing table.

Tzeng [68] proposed a Speedy Packet Lookup (SPaL) technique where the line cards of a router are re-designed so that each is responsible for a subset of the prefixes based on certain prefix bits. Each line card devotes part of its cache to its specific prefixes and the rest to the remaining prefixes. For lookups each line card checks its local cache, and in the case of a miss, uses a crossbar to check the cache of the line card responsible for the applicable prefixes before consulting the central routing table. This approach makes better use of the limited cache on each line card at the cost of the crossbar between line cards, and relies heavily on the spatial and temporal locality of lookup requests to be effective.

Baldwin and Ng [7] designed a router where each output port has its own TCAM that stores all of the prefixes that point to it. Each lookup involves matching the IP address against each of these TCAMs and choosing the TCAM with the longest matching prefix. While this approach removes the need to determine which port corresponds to the longest prefix match in a particular TCAM, simplifying each TCAM's design, it requires external logic to determine which TCAM has the longest matching prefix. This approach also requires extra TCAM space as prefixes are not guaranteed to be evenly distributed amongst the many ports of a router.

Finally this approach cannot support multiple lookups in parallel per clock cycle.

Sun and Zhao [63] converted prefixes into the ranges of addresses each port covers which are sorted and stored in a tree. Each tree node stores some compressed range endpoints and a pointer to the group of tree nodes below the current node. The authors describe how to build the tree but not how to update it, which presumably requires external processing to completely rebuild it each time.

Deng et al. [16] stored all prefixes of length 24 or less into a single stride multi-bit DRAM trie, and all longer prefixes into a TCAM. Each lookup accesses both the trie and the TCAM to determine the longest prefix match. While each update now operates on a smaller TCAM or multi-bit trie, updates on the single stride 24 bit trie can still be prohibitively expensive in the worst case.

There is a wide assortment of different hardware solutions to the routing table lookup problem that aren't completely TCAM or hardware trie based. Unfortunately, very few of them can be both pipelined, to reduce cycle time, and efficiently replicated, to support multiple lookups per cycle. The solutions that do, one FPGA based and one cache based, are impractical to update and require temporal and spatial locality in the lookups, respectively. Clearly better hardware solutions are needed to scale with the increasing demands of "backbone" routers.

Chapter 3

Design

Previous work on the Internet routing table lookup problem has yielded very few practical solutions for the demands of "backbone" routers. Software solutions running on general purpose hardware just aren't fast enough to handle the data rates. A TCAM, although the industry favorite, just doesn't scale well to the large sizes demanded, and consumes incredibly large amounts of power in the applications it is used in. Other hardware solutions, while innovative, often prove impractical or require further development.

With the rapid progression of Very Large Scale Integration (VLSI) techniques for Application Specific Integrated Circuit (ASIC) design and the continued decrease in Complementary Metal Oxide Semiconductor (CMOS) feature sizes, more and more transistors are becoming possible on a single chip. It is now commonplace for a System On a Chip (SOC) to combine many specialized processors, memories and logic together to solve a wide range of problems.

The goal of this research is to exploit the speed, compactness and power efficiency of SRAM through a hardware trie implemented as a SOC to solve the Internet routing table lookup problem for "backbone" routers. By implementing the hardware trie using many small pipelined banks of SRAM, new untapped possibilities for parallelism are exploited. This innovative approach to a hardware trie provides exceptional throughput and power efficiency that an existing TCAM just cannot match, while offering comparable latency and ASIC size. Plus new additions to the hardware trie structure ensure that updates to the lookup table are bounded to

take reasonable amounts of time in even the worst case.

This chapter begins by outlining the significant differences between the proposed implementation and an existing fixed-stride multi-bit hardware trie implementations. Next, each of the major components of the design are presented in much greater detail, including the lookup and update processes. Each component builds on the previous components, starting with each bank of memory and working up to the complete lookup table design and other high level components. This bottom-up approach was also used for the design work itself, but with a constant vision of the high level result kept in mind. It helps to refer back to Figure 3.1 in this section while reading the rest of the chapter.

3.1 Significant Differences

This section provides an overview of the significant differences between the proposed implementation and an existing fixed-stride multi-bit hardware trie implementations. First, the design trades some extra memory space for increased performance, replacing complicated compression schemes with less efficient, but simple to use, representations. Second, instead of storing all of the nodes used for one level of the trie in one large memory, each is implemented as its own concurrently accessible smaller memory. Third, each smaller memory is coupled with a register to store a default port number for the memory, with the goal of simplifying and bounding updates. Finally, each memory entry stores either a destination port number and the length of its corresponding prefix, or a pointer to a new memory. The addition of the prefix length field to the entries is a requirement of the new update procedures. An example simple fixed-stride $\{4,2,2\}$ hardware prefix trie with these modifications, populated with the prefixes from Table 3.1 is shown in Figure 3.1.

Consider three separate searches of the trie shown in Figure 3.1 for 8 bit IP addresses 00101001 (shown in light grey), 01101111 (shown in medium grey) and 11010100 (shown in dark grey). The first (light grey) search accesses the first memory and immediately returns port number 0 as an answer. The second (medium grey) search must first follow two pointers before arriving at port number 2 as an

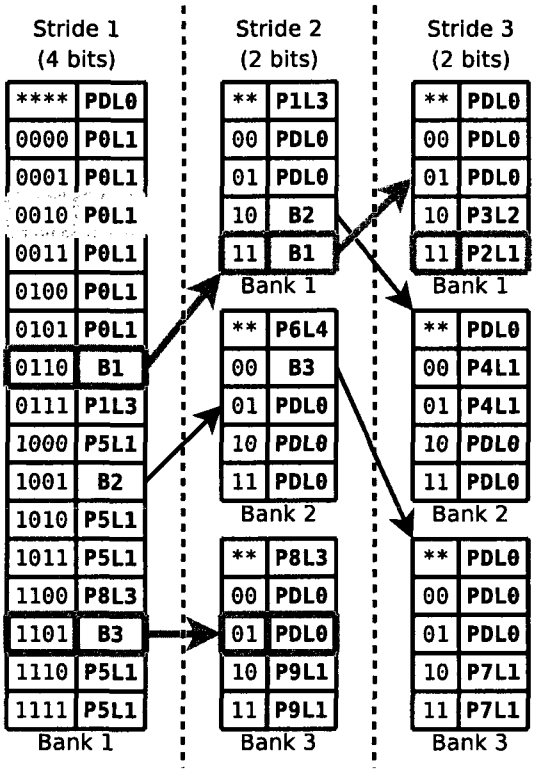


Figure 3.1: Example Fixed-Stride $\{4,2,2\}$ Hardware Prefix Trie With Default Ports

Prefix	Port	Prefix	Port
00000000/1	0	10000000/1	5
01100000/3	1	10010000/4	6
01101110/7	2	10010010/7	7
01101110/8	3	11000000/3	8
01101000/7	4	11011000/5	9

Table 3.1: Example 8 bit IP address prefixes

answer. Finally the last (dark grey) search must follow a single pointer to arrive at an answer of port number *default*, which indicates that the correct answer is the default answer for the memory, which is port number 8. Notice that all three of these lookups access different memories, with the exception that they all need to access the first memory. By making three copies of the first memory it is now possible to conduct all three of these searches in parallel. Throughput can be further increased by pipelining each level of the trie.

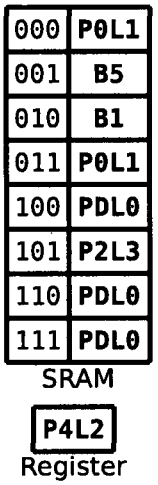


Figure 3.2: Example Lookup Bank

3.2 Lookup Bank: An SRAM Bank And A Register

The basic building block of the proposed hardware trie consists of an SRAM bank indexed by the appropriate stride of the IP address that is being looked up. Each entry in the SRAM has a single bit to identify whether it's a destination port number and the length of its corresponding prefix, or a pointer to a new memory. A port number and prefix length entry indicates that the answer to the lookup is known and no more searching is required. A pointer entry indicates that further searching is required with the next stride of the address.

In addition to the SRAM bank, each lookup bank includes a register that stores the default port number and associated prefix length for the entire bank. If any entry in the SRAM contains the special *default* port number then that entry is treated as having the register's port number instead. An example lookup bank for a stride of 3 bits is shown in Figure 3.2.

Note that the length of the appropriate prefix stored in an SRAM entry is actually the length of the prefix relative to the current stride. For example, when adding prefix 11011000/5 to the fixed-stride $\{4,2,2\}$ prefix trie in Figure 3.1, that prefix will be stored in the third and fourth entries of bank 3 of the second stride. The length of the 5 bit prefix relative to the second stride is $5 - first\_stride = 5 - 4 = 1$

bit so the length of this new prefix is actually stored as 1. Likewise, a prefix of length 6 would be stored in the second stride with a length of 2. The special length of 0 is reserved for use with entries that default to the port number stored in the register, as is the case for the first two entries of bank 3 of the second stride. These three lengths are the only valid lengths of prefixes that can be stored in the second stride (anything shorter would be stored in stride 1, anything longer would be stored in stride 3) so the optimal binary representation for the prefix lengths (in this case) takes two bits. This is a savings of two bits over the 4 bits that would be required to store a length value between 0 and 8 for the 8 bit prefixes.

3.3 Lookup Node: Lookup Bank For Multiple Agents

Every clock cycle, if enabled, a lookup bank processes a supplied address fragment (the section of the lookup IP address corresponding to the current stride) producing resolved data and default data. The resolved data is the bank's SRAM entry addressed by the address fragment, and is either a port number and prefix length, or a pointer to another memory bank. The default data is the contents of the bank's register, which is the default port number and prefix length for the bank.

To support multiple lookups in parallel, a lookup bank must be shared between several different lookup agents. Each of these agents may access the lookup bank, but at most one will do so during any given clock cycle. (How this is guaranteed is explained later in Section 3.11) Each agent supplies each lookup node with an enable signal which is high only if that agent wishes to access the node. These enable signals are logically ORed together to produce the enable signal for the node's bank. They are also used to multiplex the correct address fragment into the bank from amongst the address fragments supplied by each lookup agent. The block diagram for a lookup node is shown in Figure 3.3. It may also help to refer to the implementation of a lookup bank in Figure 3.2 on page 36.

While there are multiple lookup agents that may read from a lookup node there is only a single agent responsible for updating it. For clarity the signals required for updating a lookup node are omitted from Figure 3.3. These signals include a write

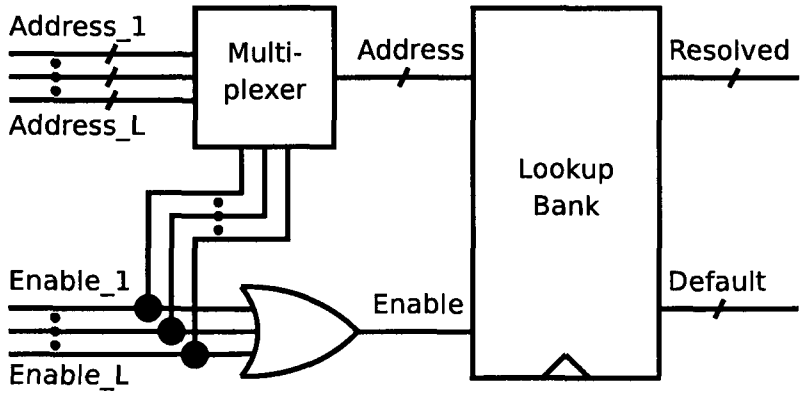


Figure 3.3: Lookup Node For L Lookup Agents

enable, update default (for selecting between updating the SRAM or the register), input SRAM data and input register data signals. The update agent also shares the first lookup agent's enable and address signals.

Since multiple lookup agents access the same lookup bank it may seem advantageous to use multi-port SRAM to reduce the complexity of the bused memory signals. Unfortunately, even dual-port SRAM tends to be 1.5 to 2 times the size of its single ported counterpart, easily countering a slight reduction in bus logic.

3.4 Background: Multiplexing Signals

In the lookup node, as well as later sections of this design, there arises a need to select a single signal from among many based on a supplied address or that signal's companion enable signal. There are several different ways of designing such a multiplexing circuit, and all have their strengths and weaknesses.

The three most popular methods to multiplex signals are: address based multiplexer trees, tristate buses, and enable based multiplexer trees. Examples of each of these types of multiplexers are shown in Figure 3.4. Address based multiplexer trees, while more complex than the other methods, provide built in address decoding which is perfect if individual enable lines aren't already available. Tristate buses, while poor at scaling for long wires, require the fewest transistors to implement. Enable based multiplexer trees offer a good compromise between the other two ap-

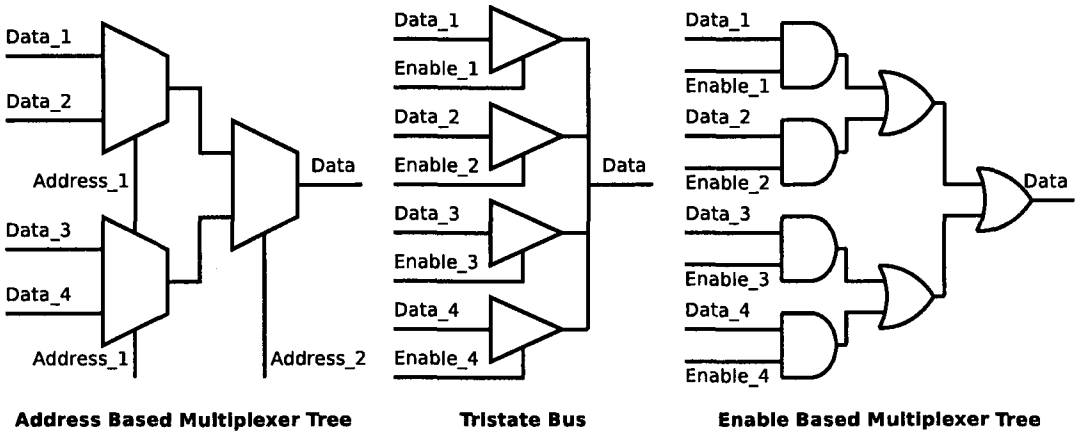


Figure 3.4: Three Different Multiplexing Schemes

Property	Address Based Multiplexer Tree	Tristate Bus	Enable Based Multiplexer Tree
Enable Type	Address	Individual	Individual
Cell Count	$inputs - 1$	$inputs$	$2 \times inputs - 1$
Cell Transistors	14	4	4
Total Transistors	$14 \times inputs - 14$	$4 \times inputs$	$8 \times inputs - 4$
Propagation Delay	$\propto log_2(inputs)$	$\propto inputs$	$\propto log_2(inputs)$
FPGA Usable	Yes	No	Yes

Table 3.2: Comparison Of Different Multiplexer Designs

proaches, and in some cases are even superior [14]. The benefits and limitations of each type of multiplexer are summarized in Table 3.2. Because this design must be FPGA implementable, only the address and enable based multiplexer trees are used. Which is used in each case depends on whether or not individual enable signals are available. The address multiplexing of the lookup bus is an ideal candidate for an enable based multiplexer tree, for example.

3.5 Lookup Bus: Connects Multiple Lookup Nodes

Just as many lookup agents may access a single lookup node, many lookup nodes may be accessed by a single lookup agent. Multiple lookup nodes are combined together and made available to all agents through a common lookup bus. Every clock cycle each of the L lookup agents provides an enable signal, the number of

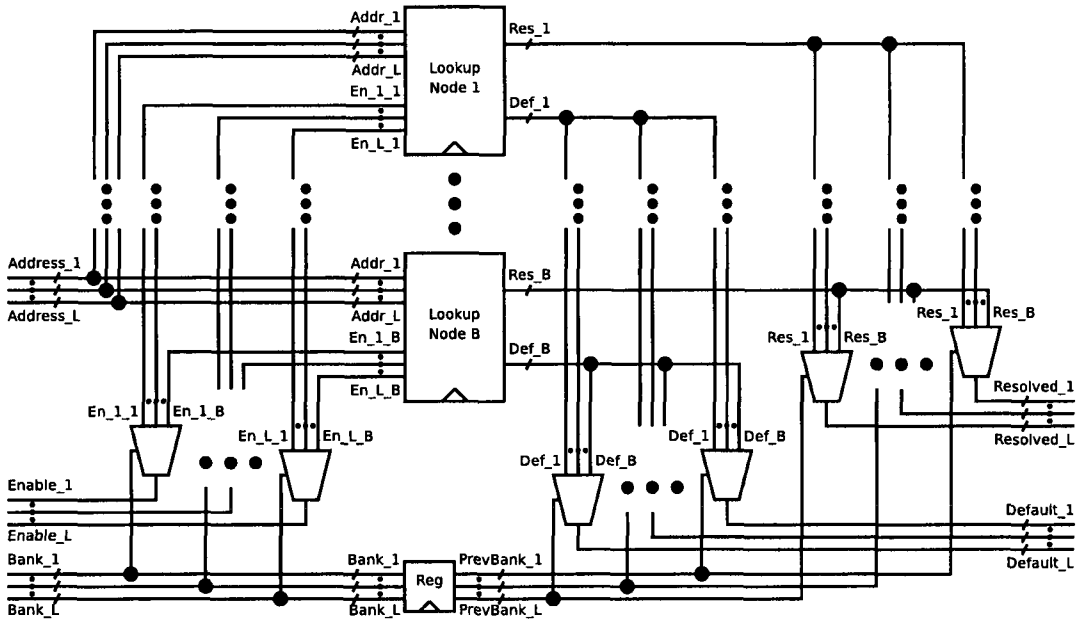


Figure 3.5: Lookup Bus For L Lookup Agents And B Lookup Nodes

the bank it wishes to access (if any) and the address within that bank it wishes to read. If the enable signal is high then the designated bank is read; otherwise nothing is read for the agent that clock cycle. The enable signal of each lookup agent is demultiplexed according to its bank number, passing the enable signal to only the designated node while passing low as the enable to all other nodes. The address of the lookup agent is simply passed to every lookup node and will be ignored by all nodes except possibly the one designated by the agent. The number of the designated bank for each lookup agent is also registered and used to multiplex out the designated node's resolved and default data back to the agent on the following clock cycle. The block diagram for a lookup bus is shown in Figure 3.5. It may also help to refer to the implementation of a lookup node in Figure 3.3 on page 38.

As with the lookup node the lookup bus also involves some update signals that are omitted from Figure 3.5 for simplicity. The update agent's write enable, update default, input SRAM data and input register data signals are simply connected to each update node. The update agent also shares the first lookup agent's enable, bank number and address signals.

Another option for generating the output resolved and default data signals for

each lookup agent would be to register the demultiplexed enable signals for each lookup agent instead of their bank numbers and use crossbars instead of multiplexers. While this would simplify the output bus designs and reduce the propagation delay on the output signals it would require more registers and increase the propagation delay on the demultiplexed enable signals. For this reason multiplexers were selected instead.

For this design to function correctly it is a requirement that multiple agents do not try and access the same memory bank during the same clock cycle. While there are several ways of ensuring this, this design uses an arbiter at the input to the lookup table to ensure that two lookups that could potentially access the same nodes never issue in the same clock cycle. The full details of the arbiter implementation and estimations of its effects on throughput are in Section 3.11. An alternative approach would be to have one of the lookup agents stall if such a conflict situation arose. Another would be for each memory bank to have an input queue, like some CPU functional units have, where lookups would wait until the specific memory was free. While these approaches are practical for other applications they would greatly complicate the lookup bus design, increasing search latency while offering little improvement in throughput and no improvement in worst case throughput, as will be shown in Section 3.11.

3.6 Lookup Stage: Lookup Bus With Agents

Routing table lookups enter a stage as five inputs: a lookup enable signal, a perform lookup signal, an IP address, a port number and a default port number. Each lookup is assigned to a lookup agent, which accesses the lookup bus, if required, then outputs the updated inputs from the stage. A block diagram of a lookup stage is shown in Figure 3.6. It may help to refer to the block diagram of the lookup bus in Figure 3.5 on page 40. The lookup enable signal indicates if the lookup agent is being supplied with work this clock cycle. The perform lookup signal indicates if this lookup agent should access the lookup bus to read a memory entry. The IP address is the full address that is being handled by the lookup. The port number

indicates where the packet with the given IP address should be routed to, if known. The default port number indicates where the packet with the given IP address should be routed to if the final port number turns out to be the special default port.

The first lookup agent also acts as an update agent for the stage, processing updates for all of the nodes on the lookup bus when required. Again for simplicity all of the update signals aren't shown in Figure 3.6. The actual update signals passed between stages will be covered in a subsequent section.

In this design a routing table lookup is passed from lookup agent to lookup agent as it progresses through the lookup table. An alternative approach would be to have a lookup assigned to a single lookup agent that accessed all the lookup buses and handled that lookup from start to finish. While this approach may seem more straightforward it greatly complicates the lookup bus as many more lookup agents would need to be attached to the nodes of a stage in order to maintain the same throughput. Intuitively, a lookup agent's bus connection for a particular stage would sit idle most of the time as the agent accessed lookup buses in other stages. This is why the current approach is used.

3.7 First Lookup Stage: A Special Case

The first lookup stage is slightly different from the other stages in several respects. Firstly, it has only a single lookup bank that is replicated so each lookup agent has a copy that it uses exclusively. Secondly, since there is only one bank per lookup agent there is no need for lookup agents to supply a bank number with their lookup requests. Thirdly, if an IP address is being processed by a first stage lookup agent then that agent will always perform a lookup, so no input perform signal is necessary. Fourthly, since this is the first lookup for a given IP address there is no existing port number or default port number data, so these inputs are also not needed. Lastly, to efficiently keep all copies of the first bank the same, the update agent performs identical updates on all banks in the first stage in parallel. A block diagram of the first stage is shown in Figure 3.7. As with previous figures, all update signals have been omitted for clarity.

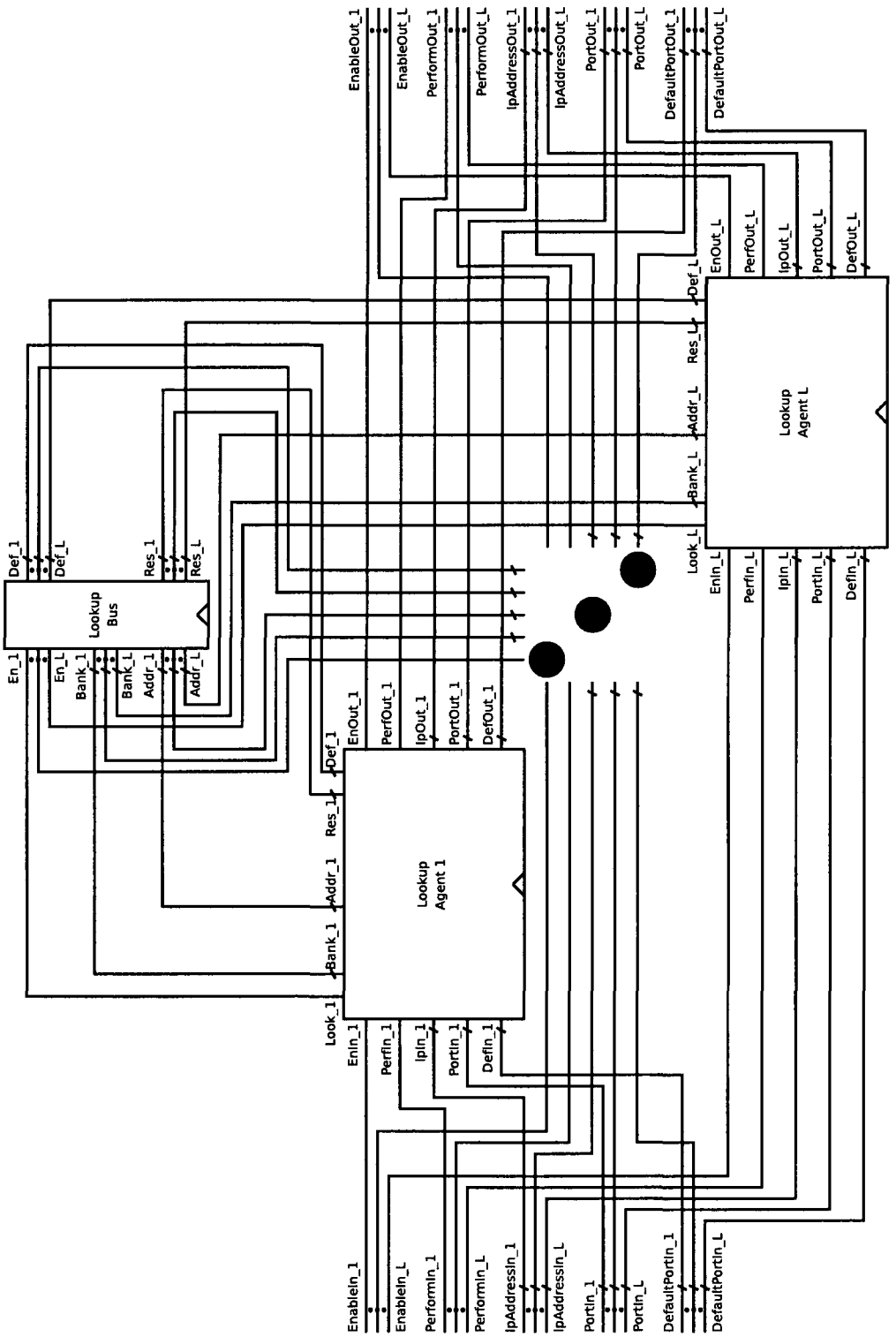


Figure 3.6: Lookup Stage For L Parallel IP Address Lookups

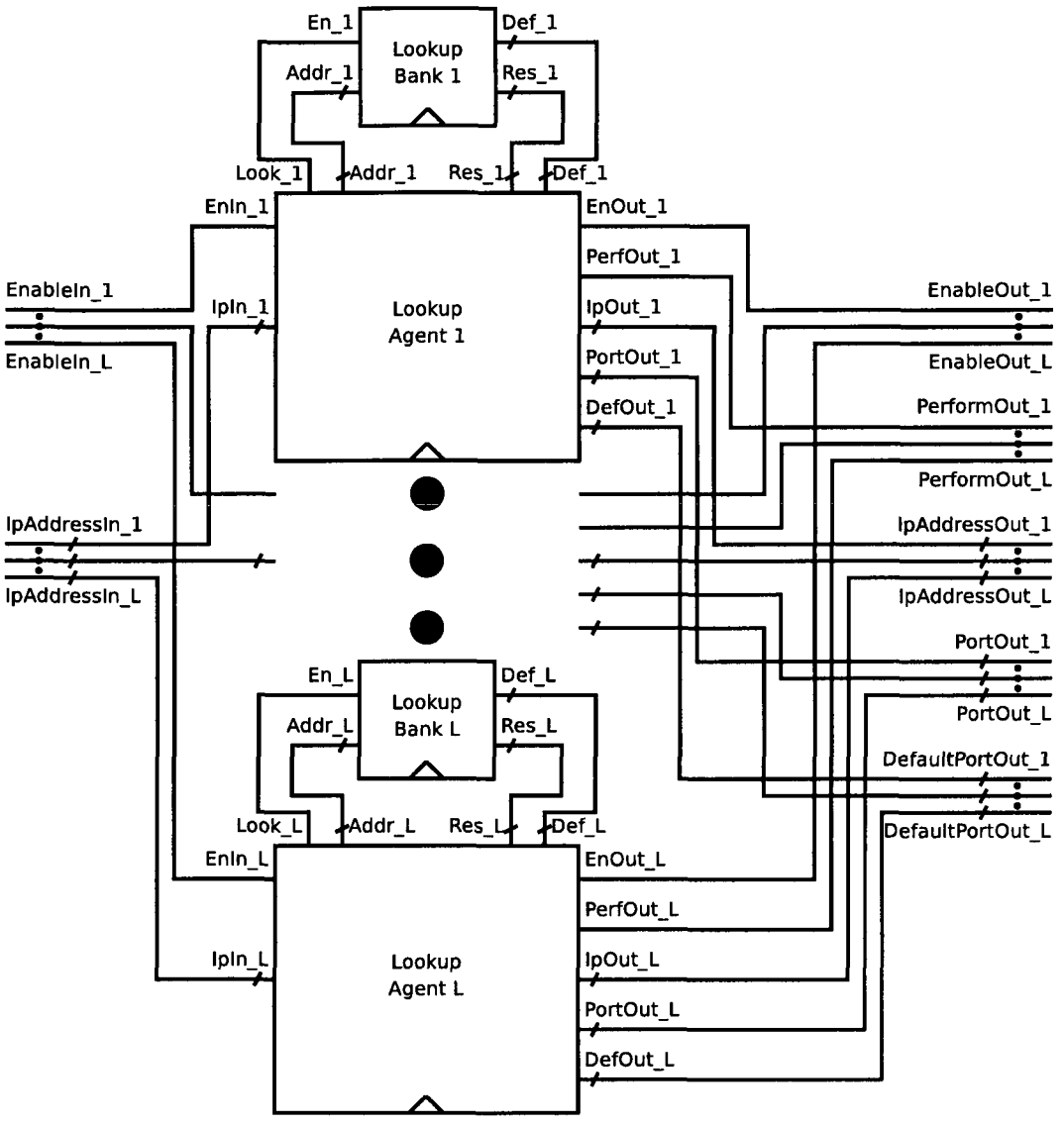


Figure 3.7: First Lookup Stage For L Parallel IP Address Lookups

While it would also be possible to replicate more than just the first stride memories to reduce conflicts and increase the design's throughput, it will be shown in Section 3.11, that for large enough first stride memories, the possible benefit of this approach is small compared to the added cost of replicating all of the second stride memories. Then, in Subsection 5.2.3, it will be shown that larger first stride memories result in more favorable chip areas. Thus only replicating the first stride memory was used for this design.

3.8 Lookup Table: Combines Multiple Stages Together

A complete lookup table consists of N lookup stages, corresponding to the N strides of the trie, chained together, followed by some result logic. The result logic transforms the output data from the final lookup stage into the results output from the lookup table. Each of these results consist of: a lookup signal that indicates if a lookup is being output this clock cycle, the IP address of the lookup being output, and the port number a packet with that IP address should be routed to. A block diagram of a lookup table is shown in Figure 3.8. As with previous figures, all update signals have been omitted for clarity.

3.9 Lookup Process

When an IP address of a packet to route enters the lookup table the lookup process begins. The IP address is handed to a lookup agent in the first lookup stage for processing on the first clock cycle. Every subsequent clock cycle the IP address is passed on to another lookup agent in the next lookup stage. After the last lookup stage, the port number to route the packet to has been determined and is output. A diagram of the lookup process is shown in Figure 3.9. Some examples of the lookup process in action can be found in Appendix B.1.

The IP address lookup is analyzed by a local lookup agent every time it is passed to a new stage. The first stage always performs a lookup, and therefore takes the first stride of the IP address and uses it to index into its only lookup bank. If the entry is a port number then no further lookups are necessary, and the answer is passed along unchanged through the other stages to the result logic. If the entry is a pointer then at least one more lookup is required, so the lookup agent instructs the following agent to carry out a lookup on the bank indicated by the pointer. In either case the default port number passed out is the default data of the first stage's bank.

In the second and all remaining lookup stages, the local lookup agent analyzes what the previous stage's lookup agent reported. If another lookup is required then the agent uses its stride of the IP address to index into the bank indicated by the

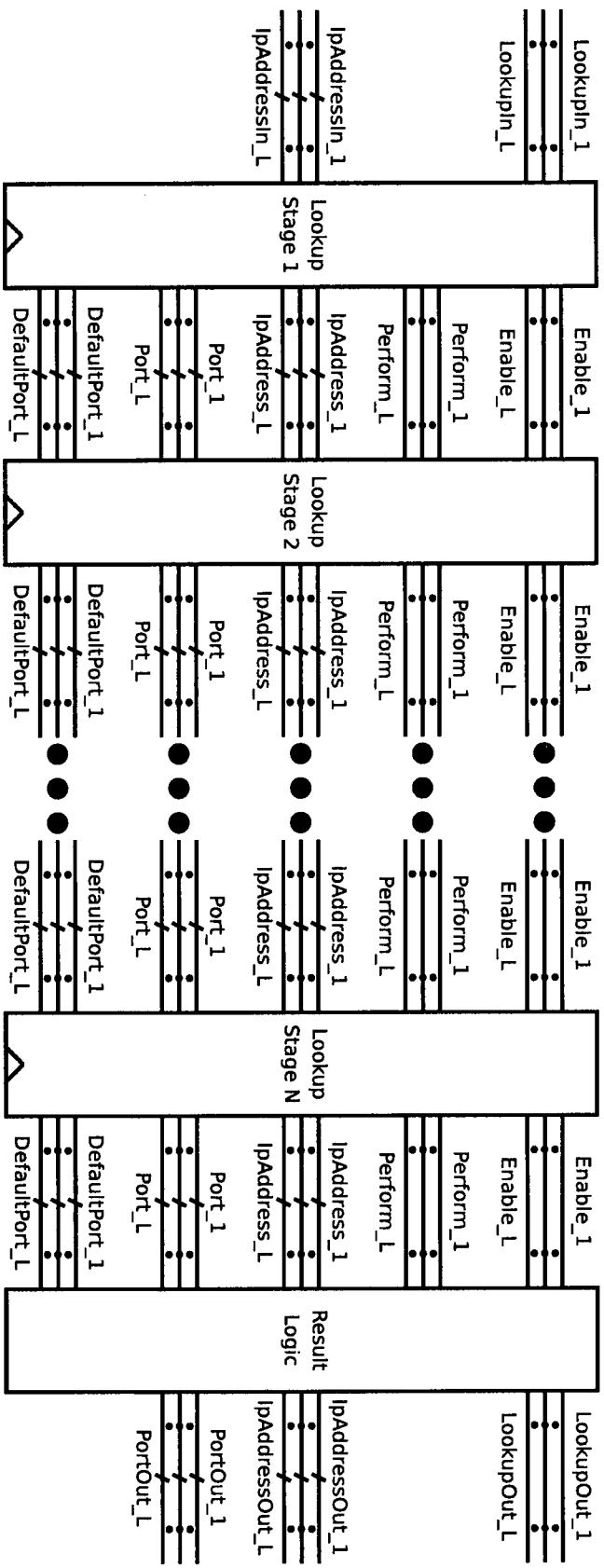


Figure 3.8: Lookup Table Composed of N Stages For L Parallel IP Address Lookups

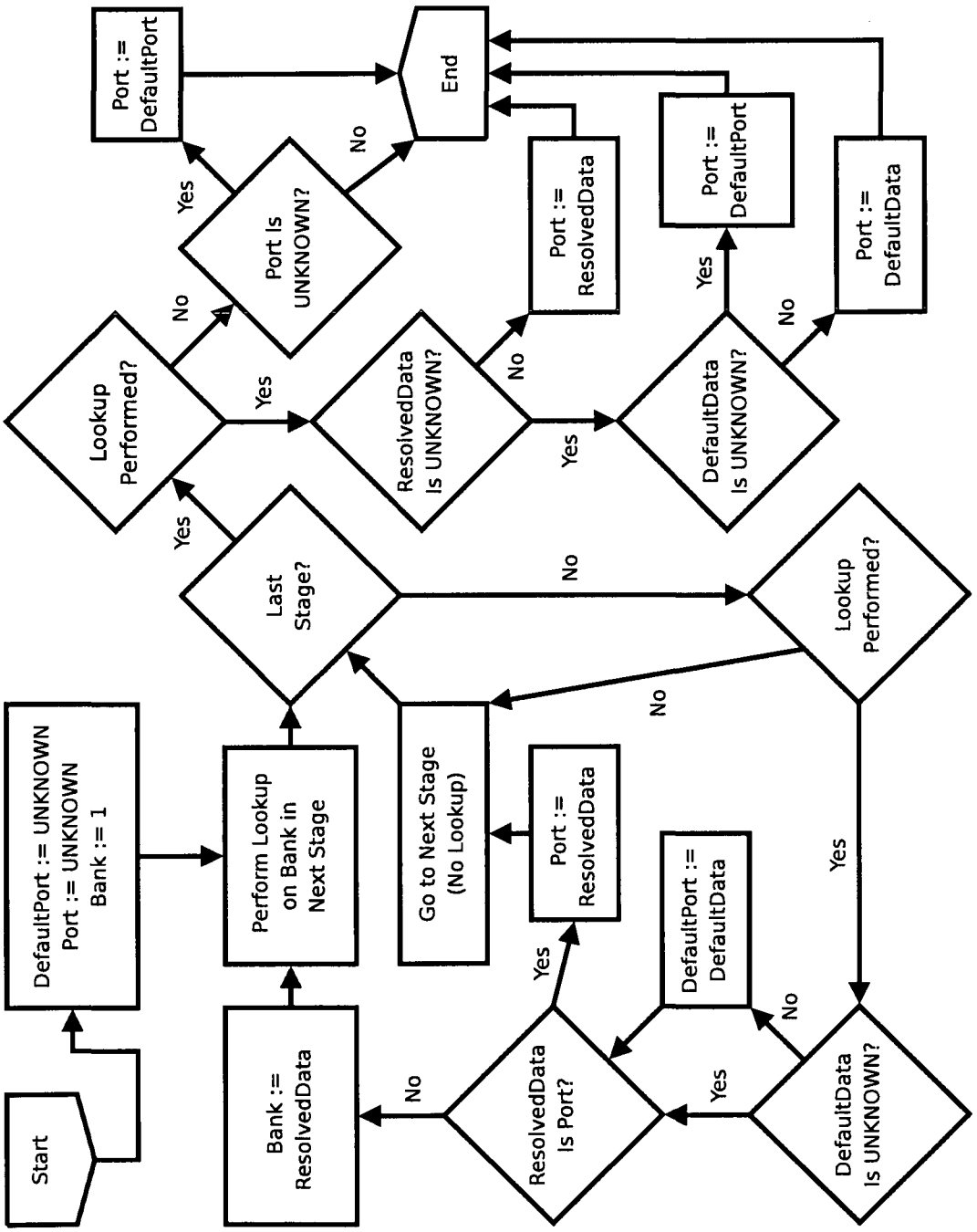


Figure 3.9: IP Address Lookup Process

previous stage. If the entry is a port number then again no further lookups are necessary. If the entry is a pointer then again at least one more lookup is required. In either case the new default port number is analyzed. If it is the special *default* (unknown) value then the new entry "defaults" to the value from the previous stage, otherwise the new default port number is used. If, on the other hand, no new lookup is required then an answer is already known and the port number and default port number from the previous stage are simply passed along to the next stage.

The port number and default port number output from the last stage are then analyzed by the result logic. If the port number is the special *default* value then the value of the default port number is used as the result. Otherwise the port number is used as the result.

An alternative approach would be to allow lookups that determine their port number early to leave the lookup table early, and hence out-of-order. While this would improve the latency of those lookups, it could actually increase the latency of other lookups started a cycle or two earlier if those lookups took longer to determine their port numbers. This is because, as will be discussed in Section 3.12, of the limited number of lookup results output by the chip per cycle. The early finishing, but later issued, lookup could take the spot of the later finishing, but earlier issued, lookup, making an already slow lookup even slower. Increasing the number of lookups retired per cycle would require more pins or faster I/O, without any added benefit to throughput. Furthermore, out-of-order completion further complicates the use of the design compared to in-order completion.

3.10 Update Process

The update process is actually the two separate processes of adding a new prefix to the lookup table, and removing an existing prefix from it. Where as lookups are pipelined and several are executed at once, only one update is processed at any one time. In addition, when the system is undergoing an update no new lookups are allowed, although existing lookups will finish. This ensures the system is always in a consistent state and that there are no update conflicts. Updates can take anywhere

between a few to many thousands of clock cycles, but are bounded to never run extremely long, as will be shown in Subsection 3.10.3. In general many simple update steps are preferred over several complicated ones so that the maximum clock frequency, and hence the lookup throughput of the design isn't adversely affected. This tradeoff of favoring lookups over updates is acceptable since updates occur far less frequently (hundreds/second) than lookups (billions/second)[1].

In each stage, the first lookup agent is extended to also perform the duties of update agent. Each update agent executes commands by reading and writing the lookup banks in its stage, and by querying data or sending commands to the previous or next stage's update agent.

An alternative approach would be to have a single update agent with read and write access to all of the lookup banks. While this would seem more straightforward, it would require an extra read connection on each lookup bus, which would result in decreased lookup performance. Since fast lookups are a much larger priority than simplified updates this approach was not used.

One of the limitations of the proposed architecture is with a prefix whose address spaces is completely covered by more specific prefixes. For example, prefix 0100/2 is completely covered by the prefixes 0100/3 and 0110/3. Because of longest prefix matching, no IP address will end up being routed by the covered prefix, only by perhaps those more specific prefixes. While having a completely covered, and hence redundant, prefix in the lookup table is of no benefit, it may come into play again if one of the more specific prefixes covering it were to be removed. Unfortunately, the proposed architecture has no way of storing redundant prefixes, so adding a redundant prefix will modify nothing and no record of it will be kept. Likewise if a previously stored prefix becomes redundant, through the addition of more specific prefixes that completely cover it, the redundant prefix will simply cease to be in the table without any warning given. While this behavior can be worked around and doesn't impact normal operation, it is still important to keep in mind when updating the lookup table.

[1] Recall Table 1.2 on Page 4

3.10.1 Addition Process

A diagram of the addition process is shown in Figure 3.10. Some examples of the addition process in action can be found in Appendix B.2.

When an update agent receives a new IP address prefix addition it checks to see if that prefix extends past the current stage. If it does then the update agent looks up its stride of the prefix in the appropriate lookup bank and analyzes the entry. If the entry is a pointer to a bank in the next stage then it is followed. If the entry is a port number then a new bank must be allocated in the next stage. This new bank's default entry is changed to be the port number and prefix length of the analyzed entry, and the analyzed entry is changed to point to the new bank. The pointer to the new bank is then followed. If the prefix does not extend past the current stage then a search of all the entries encompassed by the prefix is conducted. If a searched entry is a port number and prefix length then it is replaced by the new prefix's data if the existing prefix length is less than or equal to the new prefix's length, otherwise it is left unchanged. If a searched entry is a pointer to a bank in the next stage then that bank's default entry is checked. If the default entry's prefix length is less than or equal to the new prefix's length then it is replaced by the new prefix's data, otherwise it is left unchanged.

A special case of the addition process occurs when the prefix being added is zero length, representing a default route for the entire table. In this case the addition becomes a simple modification of the default entry of the replicated lookup bank in the first stage.

3.10.2 Removal Process

A diagram of the removal process is shown in Figure 3.11. Some examples of the removal process in action can be found in Appendix B.3.

When an update agent receives a new IP address prefix removal it checks to see if that prefix extends past the current stage. If it does then the update agent looks up its stride of the prefix in the appropriate lookup bank and analyzes the entry. If the entry is a port number, then there is an error, since the lookup table cannot

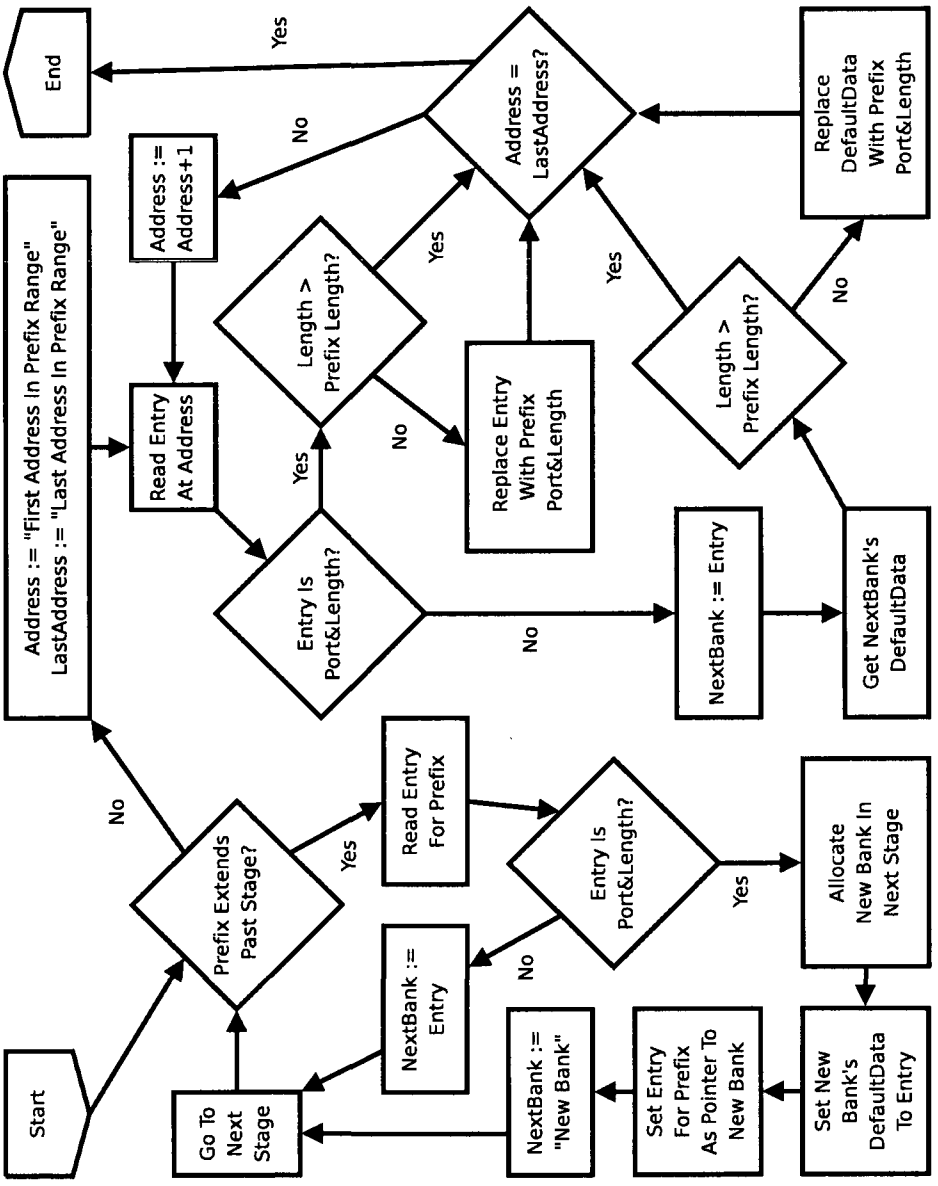


Figure 3.10: IP Address Prefix Addition Process

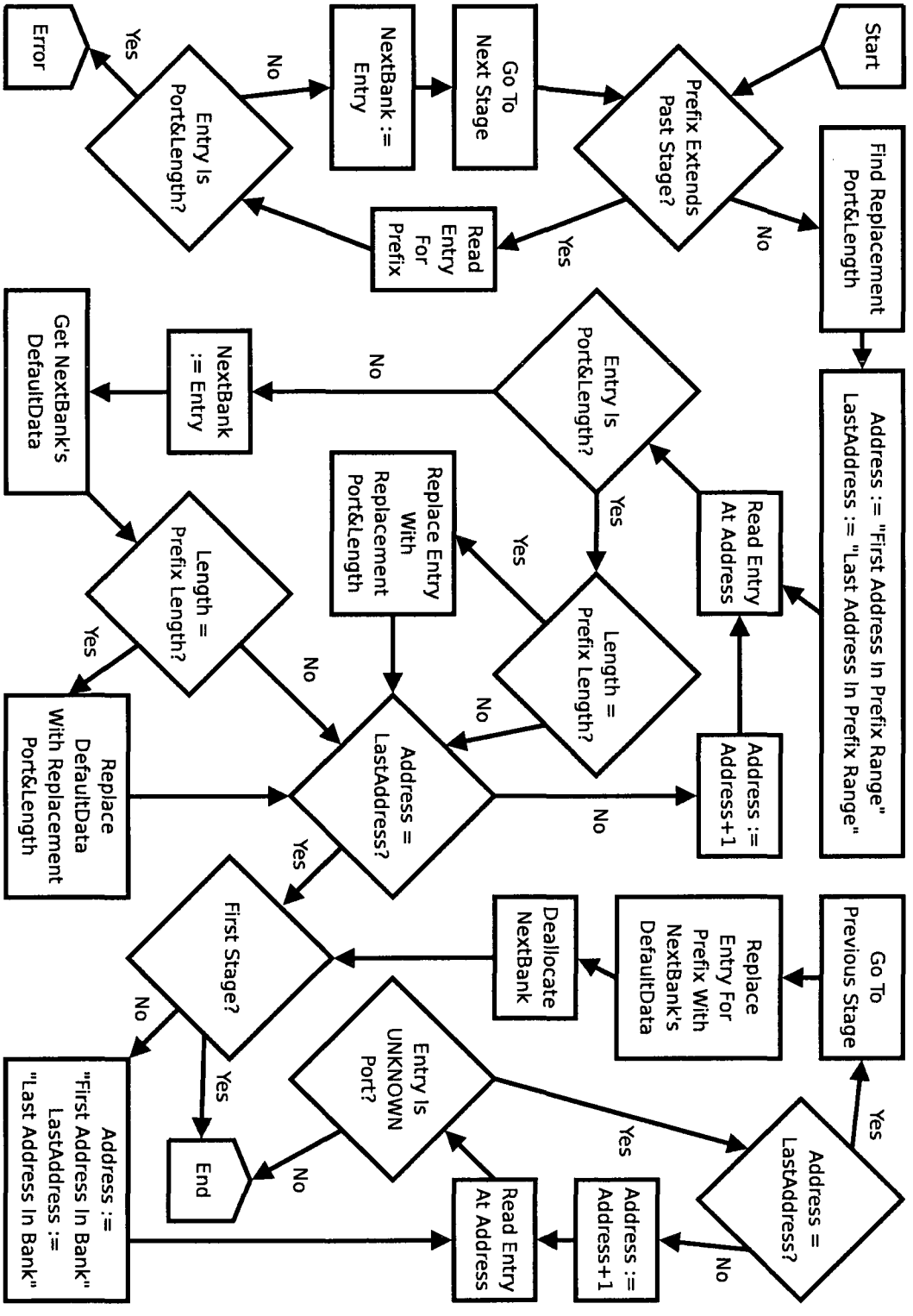


Figure 3.11: IP Address Prefix Removal Process

possibly be holding the prefix to be removed. If the entry is a pointer to a bank in the next stage then it is followed. If the prefix does not extend past the current stage then a search is conducted of entries within the current bank for prefixes that encompass the prefix to be removed. The most specific (longest) such prefix, if found, will be used as a replacement for any entries removed in the next step. If no such more specific prefix is found locally then every entry removed in the next step will be set to a port of *default* and length of 0. Next a search of all of the entries encompassed by the prefix to be removed is conducted. If a searched entry is a port number and prefix length that matches the prefix to be removed then it is set to the replacement entry, otherwise it is left alone. If a searched entry is a pointer to a bank in the next stage then that bank's default entry is checked. If the default entry matches the prefix to be removed then it is set to the replacement entry, otherwise it is left alone. After all of the entries have been processed then the bank must be checked to see if it can be deallocated. If the bank is in the first stage then this is not possible, otherwise each entry is checked to see if it contains the port number *default* and prefix length 0. If so then the bank is deallocated and the bank in the previous stage that pointed to it has its pointer entry changed to the default entry of the newly deallocated bank. Once again a check is performed of this previous stage bank to see if it can be deallocated, which continues until either the first stage is reached or a bank is found to contain a non-*default* port number or pointer, which means it can't be deallocated.

A special case of the removal process occurs when the prefix being removed is zero length, representing a default route for the entire table. In this case the removal becomes a simple modification of the default entry of the replicated lookup bank in the first stage, setting it to the port number *default*.

3.10.3 Worst Case Updates

One of the significant problems with previous software and hardware trie implementations is that in the worst case adding or removing a prefix can take an extremely long time. Prefixes can cover millions of entries, and in a dense address

space with enough prefixes it is possible that adding or removing a prefix might require checking and modifying millions of entries in the trie. If such a situation were to arise then the lookup table might be unable to handle new lookups for several seconds or more, shutting down the router for this time, which is completely unacceptable.

The update procedures presented for this design consist of the following four steps: navigate to the target bank, find the replacement entry, modify each entry covered by the prefix to reflect the change, and deallocate banks if possible. For prefix additions, only the first and third steps are required. For prefix removals, all four steps are required. In the worst case, navigating to the target bank is linear work with respect to the number of stages. Finding a replacement entry consists of searching up to half of a bank's entries. Deallocating banks if possible is, in the worst case, proportional to the sum of the sizes of a bank in each stage but the first. The big difference in this design comes with modifying the entries covered by a prefix to reflect the change. Because each modification involves either a port number and prefix length, or a default entry at the other end of a pointer, the work required is, in the worst case, proportional to the number of entries in a bank, not the number of entries in the trie! A prefix can cover at most half the entries in a bank (if it covers them all then it can be reflected in the default entry) so the work required is more precisely proportional to half a bank's entries in the worst case. In general the number of stages is far less than the number of entries in a bank, so the worst case update time for this design is either proportional to half the size of the largest bank, or to the sum of the sizes of a bank in each stage but the first. If the first stage has the largest bank, which is commonly the case, then the worst case update time is proportional to half the size of the first stage bank. For a first stride of 16 bits, for example, this would be some small multiple of $2^{16} \div 2 = 2^{15} = 32,768$ clock cycles, far less than the potential millions of clock cycles for previous designs.

When looking at real "backbone" routing tables, it is readily apparent that no entries exist for prefixes shorter than 8 bits [27]. If this was guaranteed to be the case then the design could be modified to make finding a replacement entry in the

first stage far more efficient, since prefixes of these shorter lengths could never be found. This design modification, combined with the fact that a first stage prefix could cover only $\frac{1}{2^8} = \frac{1}{256}^{th}$ of the entries in the first stage bank, would significantly reduce the worst case update time even further. However, since this lack of shorter prefixes is only a legacy left over from IPv4 classes and no guarantee for the future, the design does not make that assumption.

3.11 Arbiter

In previous sections it was assumed that no more than one lookup agent would access a given lookup node during the same clock cycle. It is the arbiter's job to structure the input of the lookup table such that this never occurs. Since in the first stage each lookup agent has its own copy of the lookup bank, conflicts are only possible between agents in subsequent stages. In order for such a conflict to occur, two lookup agents in the first stage must access the same entry during the same clock cycle. That entry must be a pointer, which would then direct two lookup agents in the second stage to access the same bank on the following clock cycle. In order for the two lookup agents in the first stage to access the same entry, the two lookups they are performing must have the same first stride bits. Therefore if two lookups with the same first stride bits are never allowed to issue in the same clock cycle there is no possibility of conflicts.

The arbiter receives all incoming IP address lookup requests serially. It allocates them into groups of however many can be performed in parallel by the lookup table. Every time the lookup table is ready for new input the arbiter supplies it the current group and starts a new one. If the current group is full the arbiter waits until a new one is started. If the next IP address would share the same first stride bits as another address already in the group the arbiter waits for a new group.

This arbiter design is easy to implement. Unfortunately, conflicts between the first stride bits of IP addresses being looked up reduces the throughput of the lookup table. Assuming uniformly random lookup requests, the chances that two IP addresses will have the same first stride bits is inversely exponentially proportional

to the length of the first stride of the lookup table. This is an important factor to consider when selecting the strides of a lookup table.

More formally, the average number of lookups completed per clock cycle, assuming a full load of random lookup requests, for a lookup table capable of L parallel lookups per clock cycle and a first stride of F bits using this simple arbiter design is: $\sum_{k=1}^{L-1}(\frac{k^2}{2^F}\prod_{j=1}^{k-1}\frac{2^F-j}{2^F}) + L\prod_{j=1}^{L-1}\frac{2^F-j}{2^F}$. As an example, a lookup table with a first stride of $F = 16$ bits that supports $L = 8$ parallel lookups would achieve an average of 7.9987 lookups per clock cycle. It is easy to see that, for larger sized first strides and uniformly random lookup requests, the loss in throughput from using the simple arbiter design is negligible and therefore a more complicated arbiter design isn't needed.

In reality the lookup requests of a router will not be uniformly random. In certain cases there could be large amounts of traffic directed at a small subset of the address space, limiting the throughput of the proposed design. In these cases even a more advanced arbiter design won't improve things much as most of the throughput will be lost to conflicts that cannot be avoided. One possible solution to this problem would be to have a small cache of popular lookup addresses to ensure that many identical lookup requests do not adversely affect the design's performance.

3.12 Packaging & I/O Signals

A lookup table capable of L parallel IP address lookups per cycle requires L input IP address and enable signals per cycle and outputs L output IP addresses, port numbers and enable signals per cycle. If IP addresses are each 32 bits, port numbers are each 6 bits and enable signals are each one bit then each lookup requires $32 + 1 = 33$ input pins and $32 + 6 + 1 = 39$ output pins for a total of $33 + 39 = 72$ pins per lookup done in parallel. This large number of pins per lookup and the serial nature of the arbiter are strong arguments for high speed serial input and output of lookups. In this scheme, a new lookup IP address is input and the result of a previous lookup is output every clock cycle. The arbiter combines L lookups into a single parallel input for the lookup table, which is clocked once every L input

clock cycles. The output of the lookup table is also serialized into L clock cycles of lookup results to be output. For an S stage (stride) lookup table, each lookup takes $S+2$ clock cycles (one cycle per stage plus one more for combining the results and another spent being deserialized on input and serialized on output) for a total latency of $L*(S+2)$ input cycles per lookup barring any conflicts the arbiter needs to resolve.

An alternative approach would be to assign an incrementing tag number of say 8 bits to each input lookup request. This tag number could then be output by the chip instead of the entire IP address, saving 24 output pins at the cost of 8 more input pins, reducing the pin count by 16. It would also makes it easier for the off chip logic to match the result port numbers to the original packets. While not implemented in this design this approach would be even more attractive if the design were modified to support 128 bit IPv6 addresses.

In the event of a conflict, the arbiter asserts a wait signal to pause lookup input until the conflict has been resolved. It also asserts the wait signal whenever an update is started until the update has finished and normal input can resume.

In the event of an error during an update, an error signal is asserted by the lookup table and the update is aborted. Updates in general also require a 2 bit input operation code (to identify a lookup, addition or removal), an input IP address prefix, an input prefix length, and an input port number (in the case of additions). The input IP address prefix can be loaded in on the same 32 input pins as the lookup IP addresses. The prefix length requires an additional 6 input pins (to represent a value between 0 and 32 inclusive) and the input port number requires another 8 input pins. Thus updates require an additional $1+2+6+8=17$ input pins.

In addition to power, ground, reset and clock pins the lookup table therefore requires $72+1+17=90$ lookup and update pins.

Chapter 4

Testing

4.1 VHDL Code

The entire design was implemented generically in VHDL (Very High Speed Integrated Circuit (VHSIC) Hardware Description Language), with properties such as the number of bits in an IP address, the number and size of each multi-bit trie stride, and the number of possible ports all being easily changed through generic constants. Not only did this make the implementation incredibly flexible so it could be used to try a wide variety of configurations, it also made it a lot easier to test each of these configurations as very little source code changed between them. By validating one configuration through extensive testing only a few corner cases remained to be validated with all the other configurations, drastically reducing testing time.

4.2 Functional Simulation

As each building block of the design was implemented it was functionally tested using ModelSim v6.0e from Mentor Graphics. The complete lookup table and arbiter were also functionally simulated and verified. Functional testing consisted of a combination of basic operation scenarios, specifically targeted corner cases, and thousands of randomly generated inputs. In particular, cases that covered all branches of the lookup and update process diagrams were developed. In all cases C++ programs were written to generate the expected output for each test case to be

Feature	Value
Logic Cells	99,216
Block Ram	7,992*Kb*
18×18 Multipliers	444
Digital Clock Management Blocks	12
Configuration Size	33.65*Mbits*
PowerPC 405 Processors	2
Max Available RocketIO Transceivers	20
Max Available User I/O Pins	1164

Table 4.1: Xilinx Virtex-II Pro XC2VP100 FPGA Features

compared against the output of the functional simulations.

4.3 FPGA Implementation

The FPGA development board used for validating the design was the AMIRIX Systems AP1100. The AP1100 combines a Xilinx Virtex-II Pro XC2VP100 FPGA with a large number of peripherals on a PCI card that can inserted into a host computer. The XC2VP100 has 99,216 logic cells, 7,992 Kb of block ram, and two embedded PowerPC 405 processors. The features of the XC2VP100 are listed in Table 4.1.

A $\{16,8,8\}$ partition with support for four simultaneous lookups was selected for the FPGA implementation. Unfortunately, due to the limited amount of block memory in less than ideal sizes and its use by other components of the system, only 4 first stride memories, 128 second stride memories and 32 third stride memories could be realized. While large enough to support some pretty elaborate routing table test cases it was clear that simulating full "backbone" routing tables, requiring thousands of memories per stride, would not be possible in a FPGA. Not surprisingly the lookup table was memory bound and took up only 16,065 (18%) of the 88,192 4 input Look-Up Tables (LUTs) and 5,166 (5%) of the 88,192 flip flops. A summary of the FPGA resource usage is in Table 4.2. The lookup table had a maximum throughput of 94.1MHz without any significant optimizations.

Testing the lookup table at almost 100MHz presented a fairly difficult challenge

FPGA Resource	Total Available	Total Used	Utilization
Look-Up Tables	88,192	16,065	18%
Flip Flops	88,192	5,166	5%
Block Rams	444	319	71%

Table 4.2: Lookup Table FPGA Resource Usage

due to large amount of input and output data required. Transferring in four 32 bit IP addresses and returning four 32 bit IP addresses and four 6 bit port numbers in the just over $10ns$ cycle time was out of the question for any of the standard peripherals on the development board; unfortunately, the board did not expose the FPGA's high speed serial links. This left three alternatives: generate the input within the FPGA, run the lookup table at a much slower clock frequency, or buffer input within the FPGA then release it into the lookup table at full speed. While generating the input within the FPGA would have been the most straightforward it would also be the least flexible and hardest to verify as the input generation and output verification would all be internal to the FPGA. The second alternative of running the lookup table at a lower clock frequency would offer the flexibility of externally generated and verified data, but wouldn't validate the extremely high throughput of the design. It's for these reasons that the third approach was selected: to offer the best of the first two approaches at the cost of some added complexity in its implementation.

Test input was sent to the FPGA through the evaluation board's serial port for simplicity. A simple text based command format was used to specify the sequence of IP address lookups, prefix additions and prefix removals for each test. Additional commands reset the lookup table, clearing its contents, and released all of the previously buffered commands to the lookup table at full speed. One of the embedded PowerPCs in the FPGA was used to parse the serial port input and supply it to a very wide internal memory. This very wide internal memory could buffer up to a few thousand commands, which were then dispatched together to the lookup table as fast as it could handle them. The output of the lookup table during a run was similarly buffered into a very wide internal memory to be later read by the PowerPC and sent back across the serial port. The number of clock cycles each run took was

also tracked and printed to the serial port, to verify how quickly the lookup table could process each sequence.

Testing of the FPGA implementation involved verifying basic operation, specifically targeted corner cases, thousands of randomly generated operations, lookup performance and worst case update performance. A lot of the test cases generated for the design's functional simulation in ModelSim were easily re-used here, and C++ programs were again used to validate the output of the FPGA.

Chapter 5

Quantitative Comparisons of Stride Choices

One of the most important implementation decisions for any multi-bit hardware trie design is the choice of how many strides, and the sizes of each. A poor choice of strides can dramatically increase the memory required to store a set of prefixes; it also affects the area, power consumption, latency and throughput of the final design in hardware. Additionally, a good choice of strides for one set of prefixes might be a very bad choice in strides for another set. Unfortunately, most of the previous work on multi-bit hardware tries doesn't address this important issue, or makes recommendations based on only one set of prefixes from a single routing table.

In this chapter all possible choices of strides for this lookup table design are compared using real prefixes from real "backbone" routing tables. First, the source of this routing table data is presented and analyzed. Next, several different metrics for evaluating stride choices are developed. Finally, these metrics are applied to all the possible stride choices, the results are compared, and the preferred choice of stride is presented.

5.1 Source of Routing Table Data

This section presents the source of the routing table data used for comparing stride choices. The largest routing table is analyzed, and the information about all the routing tables is condensed into a form that makes stride choice comparisons much

easier.

5.1.1 Border Gateway Protocol (BGP) Tables

The "backbone" routers at the heart of the Internet use the Border Gateway Protocol (BGP) to communicate routing information between each other to build their own routing tables. Each group of "backbone" routers under a single technical administration (for example, an Internet Service Provider) is assigned a unique Autonomous System (AS) number [57]. Each AS advertises the IP address ranges assigned to each of its internal networks to each of its AS neighbors, who then forward this information on to each of their neighbours, and so on, until each AS knows about every other AS. Each AS uses these advertisements to construct a BGP routing table mapping each advertised IP address range to the best path to the destination AS that advertised it. Each of these paths might involve one or more intermediate Autonomous Systems as every AS is not directly connected to every other AS. The BGP routing table can then be used to create an IP address lookup table, mapping IP address ranges to the appropriate port connected to the appropriate next-hop AS.

While the BGP routing table of an AS might not directly correspond to the IP address lookup table of one of that AS's routers it offers a very good approximation of the types of prefixes the lookup table would contain. While using the actual lookup tables of "backbone" routers would be ideal for stride comparisons, very few are made public, for security and other reasons. Fortunately a large number of AS BGP routing tables were made publicly available on http://bgp.potaroo.net/ as part of a project to study the growth of the BGP table, among other things. BGP tables from each of the Autonomous Systems in Table 5.1.1 were captured on three different dates: December 22, 2005; November 26, 2006; and May 27, 2007. Unfortunately, at the time of writing, it looks as though http://bgp.potaroo.net/ now only presents data on a single AS.

AS Numbers of BGP Routing Tables Captured
286 1221 1239 1668 2493 2497 2828 2905 2914 3257 3277 3292
3303 3333 3356 3549 3561 4513 5459 5511 5650 6079 6395 6453
6509 6539 6939 7018 7500 7660 8075 11537 11608

5.1.2 The Largest Routing Table

The AS with the largest number of prefixes on May 27, 2007 was AS7500 with 222,728. It started at 169,103 prefixes on December 22, 2005 and grew to 205,784 prefixes on November 26, 2006. Figure 5.1 shows the number of prefixes of each length AS7500 had in its routing table on each of the three capture dates. There are several interesting properties of BGP routing tables to take note of from this graph. First, that there are clusters of prefixes of lengths 8, 16 and 24, corresponding to the old classes before Classless Inter-Domain Routing (CIDR) was introduced. Second, there are no prefixes of length less than 8. While nothing in CIDR prevents such large ranges of IP addresses to be allocated to a single provider, it is unlikely to happen for the remaining use of IPv4, due to the scarcity of available address ranges. Lastly, there are a somewhat significant number of prefixes of length 25 or more. Ideally such specific prefixes could be aggregated into larger prefix ranges and not need to be advertised individually in the BGP table. Unfortunately, that is not the case and any solution to the routing table lookup problem must handle them as well.

5.1.3 Reducing the Complexity of Stride Choice Comparisons

While knowing the number of prefixes in the largest "backbone" routing table is enough to appropriately size a TCAM to handle all of the different routing tables, it's not so straight-forward for multi-bit hardware tries. For a given stride choice a routing table with a smaller number of prefixes might actually require more memory than a routing table with more prefixes. All of the routing tables must therefore be taken into consideration for all possible stride choices.

To simplify this problem, observe that, for a given set of routing prefixes P, there is certain subset of prefixes, P_n, containing all prefixes of length longer than

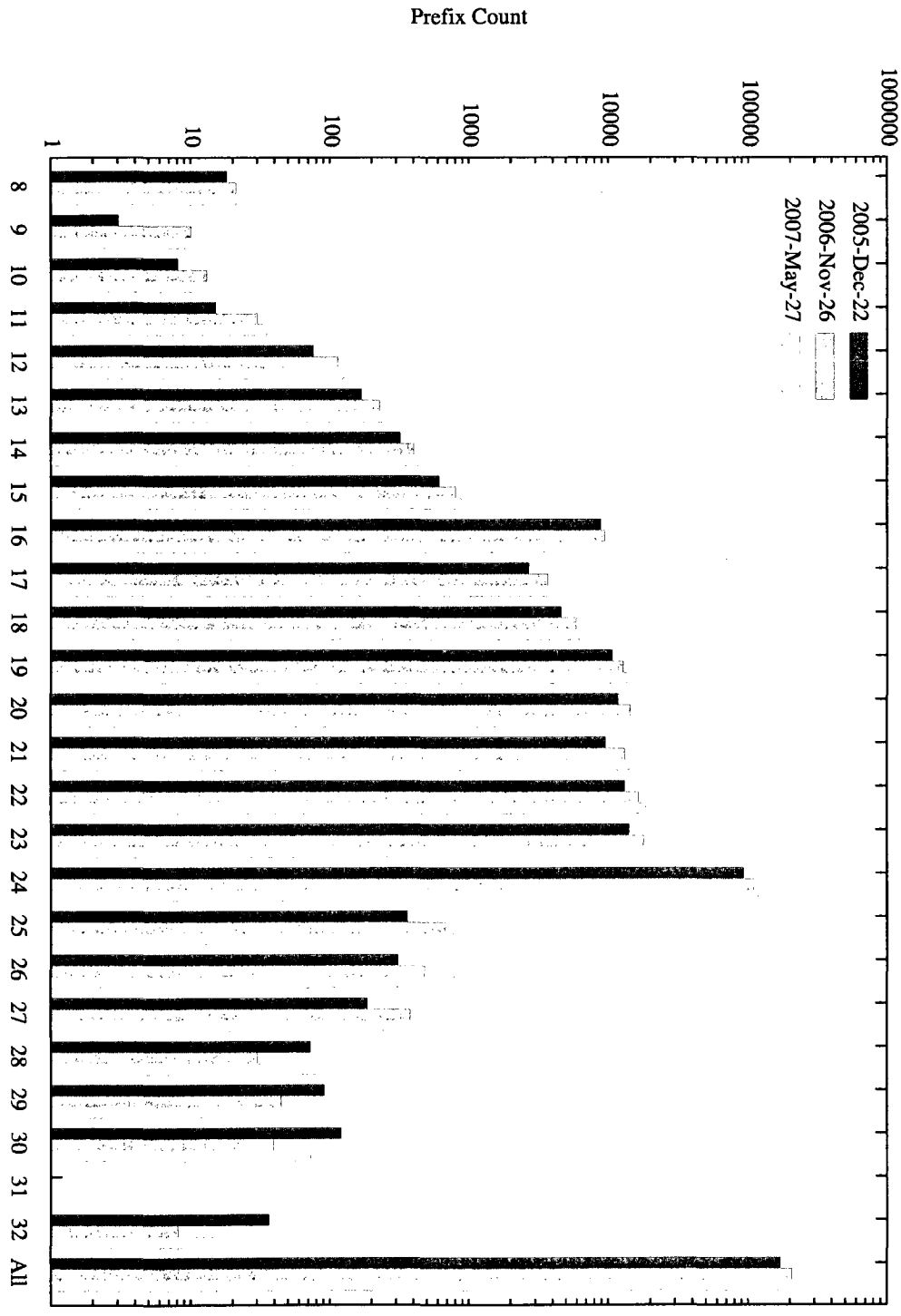


Figure 5.1: Prefixes of Each Length in the Routing Table of AS7500

n bits. If the prefixes P were to be stored in a two stride multi-bit trie whose first stride was n bits wide, then all of the prefixes in $P - P_n$ would be stored in that first stride, and the prefixes in P_n would need to be stored in the second stride. Adding each prefix in P_n to the lookup table involves looking up the first n bits of the prefix in the first stride lookup table and adding a pointer to a new second stride lookup table, unless of course a previous prefix in P_n shared the same first n prefix bits and has already added the pointer and new lookup table. The number $L_P(n)$ of second stride lookup tables required to store the P_n prefixes is therefore less than or equal to the number of prefixes in P_n. Now observe that if the first stride of n bits is partitioned into two strides of total length n, the $P - P_n$ prefixes will be stored in the first two strides, and the P_n prefixes will require $L_P(n)$ new lookup tables in the third stride. Also observe that if the second stride was partitioned and the first stride of n bits was not, the $P - P_n$ prefixes would still be stored in the first stride, while the P_n prefixes would still require $L_P(n)$ second stride lookup tables (and perhaps some third stride lookup tables as well). Thus no matter how many strides of how many bits the first n bits of the trie are partitioned into, and no matter how many strides of how many bits the subsequent strides are partitioned into, a set of prefixes P will always require $L_P(n)$ m^{th}-stride lookup tables when the first $m-1$ strides total n bits, for any value of m. By precomputing the $L_P(n)$ values, for all possible n for a set of prefixes P, the number of lookup tables needed for each stride for a given choice of strides is now easily determined.

The problem is even further simplified by now computing the $L_P(n)$ values, for each set of prefixes P, corresponding to each of the different routing tables. The maximum $L_P(n)$ value for a given n amongst all of the sets of prefixes, $L_{max}(n)$, is therefore the number of next-stride lookup tables required to handle all possible sets of prefixes. It, therefore, suffices to use the single set of $L_{max}(n)$ values to evaluate a given choice of strides, instead of each set of $L_P(n)$ values for each set of prefixes P.

The $L_{max}(n)$ values, for each value of n for each capture year, are plotted in Figure 5.2. Several important observations can be made by looking at this graph.

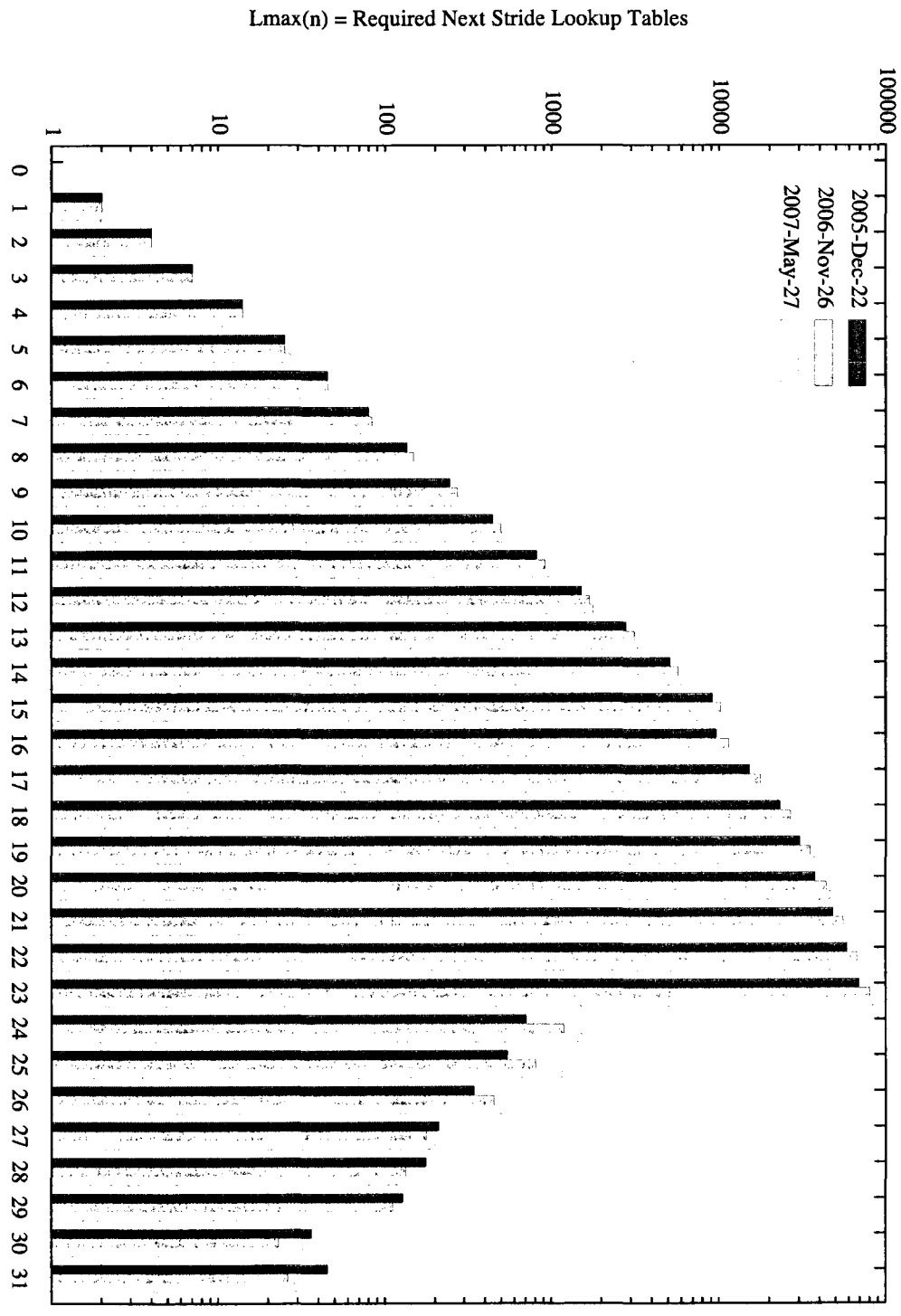


Figure 5.2: Required Next-Stride Lookup Tables For Each Total of All Previous Stride Bits to Support All Routing Tables

First, the growth of $L_{max}(n)$ is clearly not exponential with respect to n (especially for $n \geq 24$). This indicates that a single stride of 32 bits is not a very efficient choice for these prefix sets, and significant savings can therefore be realized by using more than one stride. Second, starting a new stride after the first 17 to 23 bits is clearly a bad idea as it would require well over 10,000 next-stride lookup tables. A single stride spanning this bit range clearly offers significant memory savings. Third, the first two observations present a trade-off: longer strides spanning certain ranges can reduce the required number of next-stride lookup tables to save memory, but at the same time longer strides are less efficient, allocating memory for address ranges not covered by any prefixes. Finally, it's clear that the number of next-stride lookup tables, $L_{max}(n)$, required for a given n is growing as time passes and the IPv4 address space gets closer to exhaustion. This growth, present in the $5 \leq n \leq 26$ range, is also stable, providing confidence that a multi-bit hardware trie implementation, with a particular choice in strides, will remain viable for years to come, as long as some extra lookup tables are provided for each stride past the first.

5.2 Developing Stride Choice Comparison Metrics

In this section three different stride choice comparison metrics are developed with varying levels of computational complexity.

5.2.1 Metric #1: Required Memory Bits

A poor choice in strides for a multi-bit hardware trie can drastically increase the required memory for a design. It seems only natural then that one of the chosen metrics should be the required number of memory bits of the design. With this metric the number of entries in a stride's lookup table is multiplied by the number of bits required to store each entry. The number of bits needed to store the default port information is added, then the resulting number is multiplied by the number of lookup tables required for that stride. This is repeated for each of the remaining strides, with the total being the number of memory bits needed for the entire design.

5.2.2 Metric #2: Required Memory Entries

This metric counts the number of memory entries required for a design, which is a simplification over counting the number of memory bits required for that design. Recall from Section 3.2 that each memory entry stores either a port number and relative prefix length, or a pointer to a next stride lookup table. Observe that each port number is a fixed number of bits. Also note that the number of bits required to store the relative prefix length is logarithmic with respects to the size of the stride. Finally, note that the number of bits required to store a pointer to a next stride lookup table is logarithmic with respects to the number of next stride lookup tables. It is, therefore, expected that the size of each memory entry varies only slightly between the different strides, and the added complexity of counting individual bits, instead of entries, is not required to draw the same conclusions.

5.2.3 Metric #3: Required Design Area

Far more complicated than either of the previous two metrics, the last metric seeks to estimate the actual chip area consumed by the design, which is predominately due to the SRAMs. It is expected that this approach will give the most accurate results, as it factors in several second order effects not considered by the other metrics. As an added benefit, this approach also allows estimation of the power consumption, latency and throughput of the final design, for little added cost. This provides several different comparison points for stride choices resulting in similar chip areas. It is important to note, however, that more complicated metrics, such as active power consumption and maximum throughput, are dependent on the actual workload of the router, and these metrics are therefore approximations. It is also important to remember that these estimates are not substitutes to results from an actual hardware implementation.

Just as there are a huge number of possible stride choices, there are a large number of different SRAM sizes required depending on those stride choices. The size of a stride determines the number of entries required, and the number of next stride lookup tables affects the size of each of those entries. Unfortunately, it is not prac-

	Value	Notes
Technology	45nm SOI	
Cell Height	$0.38\mu m$	
Cell Width	$0.83\mu m$	
Cell Area	$0.315\mu m^2$	
Macro Organization	$32K \times 16$	
Macro Height	$0.475mm$	1024 word lines
Macro Width	$0.482mm$	512 bit line pairs
Macro Area	$0.229mm^2$	
Macro Access Time	$450ps$	$0.9V$, $85°C$, Typical Process
Macro Active Power	$21mW$	$1GHz$, $1.0V$, $125°C$, Fast Process
Macro Leakage Power	$24mW$	$1GHz$, $1.0V$, $125°C$, Fast Process

Table 5.1: Performance of a State of the Art SRAM

tical to implement each of these possible combinations in hardware to gather the most accurate numbers, and previously published work covers only a small fraction of these configurations. To solve this problem, a published state of the art SRAM design is selected as a baseline, and the properties of all the other SRAM configurations are extrapolated from it. While this approach clearly introduces some error into the values, it is expected that they will still be accurate enough to draw some meaningful conclusions.

Pilo et al. [52] presented a SRAM macro in 45nm SOI technology with a fast access time and advanced power reduction techniques. The published performance of this SRAM macro are in Table 5.1.

Based on these figures and an accurate layout diagram of the macro cell the authors provided, it is possible to estimate how the macro overhead will change if the number of columns and rows in the macro were to be changed. The overhead affecting the width of the macro can be attributed to repeaters, which scale with the number of columns in the macro ($0.043\mu m/column$), and to word line drivers and row decoders, which mostly don't change with the number of columns ($27.8\mu m$). Likewise the overhead affecting the height of the cell can be attributed again to repeaters, which scale with the number of rows in the macro ($0.044\mu m/row$), and to column decoders and sense amplifiers, which mostly don't change with the number of rows ($47.4\mu m$). The required width and height of a custom SRAM macro

	Symbol	Formula
Width	S_W	$[(0.83\mu m + 0.043\mu m) \times columns + 27.8\mu m]$
Height	S_H	$[(0.38\mu m + 0.044\mu m) \times rows + 47.4\mu m]$
Area	S_A	$S_W \times S_H$
Cycle Time	S_{Tc}	$\frac{1}{2} \times (S_W/0.482mm + S_H/0.475mm) \times 410ps$
Active Power	S_{Pa}	$(S_A/0.229mm^2) \times 21mW$
Leakage Power	S_{Pl}	$(S_A/0.229mm^2) \times 24mW$
Total Power	S_{Pt}	$S_{Pa} + S_{Pl}$

Table 5.2: Formulas for Estimating the Performance of a State of the Art SRAM

cell is therefore estimated to be $[(0.83\mu m + 0.043\mu m) \times columns + 27.8\mu m]$ and $[(0.38\mu m + 0.044\mu m) \times rows + 47.4\mu m]$ respectively. These formula are summarized in Table 5.2.

Approximations of the new cycle times, active power, and leakage power of a scaled macro cell are also possible. Since the capacitance of each bit line scales linearly with the height of the macro, and the capacitance of each word line scales linearly with the width of the macro, it is expected that the cycle time of the macro, which is dominated by the capacitances of the bit lines and word lines, will scale linearly with the width and height of the macro as well. The active power is dominated by the charging of all the bit lines, with the number of bit lines scaling with the width of the cell and the length of each scaling with the height of the cell. Thus it is estimated that the active power will scale with the area of the cell. Finally the leakage power is dominated by the number of cells in the macro, and is therefore estimated to scale with the area of the macro cell as well. These formulas are summarized in Table 5.2.

To calculate the total area of the design, the estimated area of the SRAMs in each stride are summed. The area overhead of the non-SRAM logic is estimated to be $0.25mm^2$ for a single lookup per cycle, and expected to scale linearly with the number of additional lookups per cycle supported. This formula is summarized in Table 5.3.

To calculate the total power consumption of the design, the estimated active and leakage power consumed by each stride is summed. The estimated leakage

	Symbol	Formula or Description
Number of Strides	N	Number of strides in the design
Stride Memories	M^k	Number of memories in stride k
Stride Width	W^k	Number of bits in stride k
Stride Memory Area	A^k	Area of a memory in stride k
Max. Lookups/Cycle	L_{max}	Maximum number of lookups/cycle supported
Stride Area	T_A^k	$M^k \times A^k$
Stride Active Power	T_{Pa}^k	$L_{max} \times S_{Pa}^k$
Stride Leakage Power	T_{Pl}^k	$M^k \times S_{Pl}^k$
Stride Total Power	T_{Pt}^k	$T_{Pa}^k + T_{Pl}^k$
Average Lookups/Cycle	L_{avg}	$\sum_{k=1}^{L_{max}-1}(\frac{k^2}{2^{W^1}}\prod_{j=1}^{k-1}\frac{2^{W^1}-j}{2^{W^1}}) + L_{max}\prod_{j=1}^{L_{max}-1}\frac{2^{W^1}-j}{2^{W^1}}$
Design Cycle Time	D_{Tc}	$max_{k=1}^{N}(S_{Tc}^k) + 200ps$
Design Throughput	D_T	L_{avg}/D_{Tc}
Design Latency	D_L	$\sum_{k=1}^{N+3}(D_{Tc})$
Design Area	D_A	$\sum_{k=1}^{N}(T_A^k) + L_{max} \times 0.25mm^2$
Design Active Power	D_{Pa}	$\sum_{k=1}^{N}(T_{Pa}^k) + L_{max} \times 22.9mW$
Design Leakage Power	D_{Pl}	$\sum_{k=1}^{N}(T_{Pl}^k) + L_{max} \times 26.2mW$
Design Total Power	D_{Pt}	$D_{Pa} + D_{Pl}$

Table 5.3: Formulas for Estimating the Performance of a Design

power for a stride is the expected leakage power of a single memory for that stride multiplied by the number of memories in that stride. The estimated active power for a stride is the expected active power for a single memory for that stride multiplied by the number of lookups per cycle; This is the maximum number of memories that can possibly be active in that stride at one time. In actuality, on average, fewer than that will be active, as some lookups will be resolved in previous strides. This estimate, therefore, provides a good upper-bound on the maximum expected average power, and hence total power for the design. The active and leakage power overheads of the non-SRAM logic are estimated to be $22.9mW$ and $26.2mW$ respectively for a single lookup per cycle, and expected to scale linearly with the number of additional lookups per cycle supported. These formulas are summarized in Table 5.3.

To calculate the maximum throughput for the design, the stride with the largest memory cycle time dictates the highest frequency that the design could possibly operate at. In practice, however, the routing, logic and register overhead of connecting these memories up in the design will be added to that. Determining the exact value

of this overhead would require implementing the design in hardware, so an approximation of $200ps$ is used instead. The listed number of lookups per cycle indicates the maximum number of lookups that can be completed each clock cycle, but in practice the average number of lookups completed per cycle can be much less than that. As mentioned in Section 3.11, conflicts can occur between lookups that would cause them to access the same memories at the same time, which forces the arbiter to issue them in different clock cycles, reducing the design's throughput. The equation from that section is used to calculate the expected average lookups completed per cycle, which is then multiplied by the highest possible frequency to determine the maximum possible throughput for that stride choice. These formulas are summarized in Table 5.3.

To calculate the total latency of the design, the cycle time of the design is multiplied by the number of strides in the design plus 3. Each lookup spends one clock cycle in each stage (stride), one in the arbiter, one to determine the final lookup result, and finally one during input and output from the chip. This formula is summarized in Table 5.3.

5.3 Stride Choice Comparisons Using Three Metrics

In this section, each of the three metrics is applied to every single possible stride choice, for every possible number of strides. In addition the number of parallel lookups, per clock cycle, supported is varied in all cases and is factored into the results. Only the last capture taken on May 27, 2007 is used for this analysis to limit the size of the data set, although the results are comparable for the other capture dates as well. The most promising candidates from each metric are presented and compared to evaluate the effectiveness of each metric. Finally, the preferred stride choice is selected and presented.

5.3.1 Metric #2: Required Memory Entries

Figure 5.3 shows the design's preferred stride choices to minimize the number of memory entries. Not shown on this graph is the result for a single stride of 32 bits

which requires $(1+2^{32}) \times L = 4,294,967,297 \times L$ memory entries, where L_{max} is the maximum number of simultaneous lookups per cycle supported.

Based on the prefix distributions seen in Figure 5.1 it is unsurprising to see that almost all the preferred stride choices are based on separating the address space between the first 24 bits and the last 8 bits. The second less extreme drop between the number of length 25 to 27 prefixes to the length 28 to 32 prefixes is reflected in the benefit of partitioning the last 8 bits into two strides of 3 and then 5 bits respectively.

Increasing the number of strides from 1 can dramatically decrease the required number of memory entries initially, with most of the reduction realized with 5 strides. Although not shown on the graph, for one lookup per cycle, minor reductions continue logarithmically until 12 strides, at which point the overhead of having an extra default entry per memory overtakes any reductions from moving to more strides of smaller size.

As mentioned previously in Section 3.7 increasing the number of supported lookups per cycle requires replicating the first stride memory. Thus certain stride choices like $\{24,8\}$ might result in the fewest memory entries for 1, 2 or 4 lookups per cycle, but other stride choices like $\{22,10\}$, with smaller first stride memories, become better choices for 8 or more lookups per cycle.

Based on the number of required memory entries the preferred stride choice for up to 5 strides and 16 lookups per cycle is $\{09,08,04,03,08\}$ with $1,317,582$. This offers a factor of $50,000$ reduction in memory entries over the single stride of 32 bits for the same 16 lookups per cycle. Without any other constraints taken into account, it always makes sense to increase the number of parallel lookups using this metric as the incremental cost of an additional first stride memory will always be less than the incremental benefit of an additional lookup per cycle. This approach is not practical, however, as will be shown in Subsection 5.3.3.

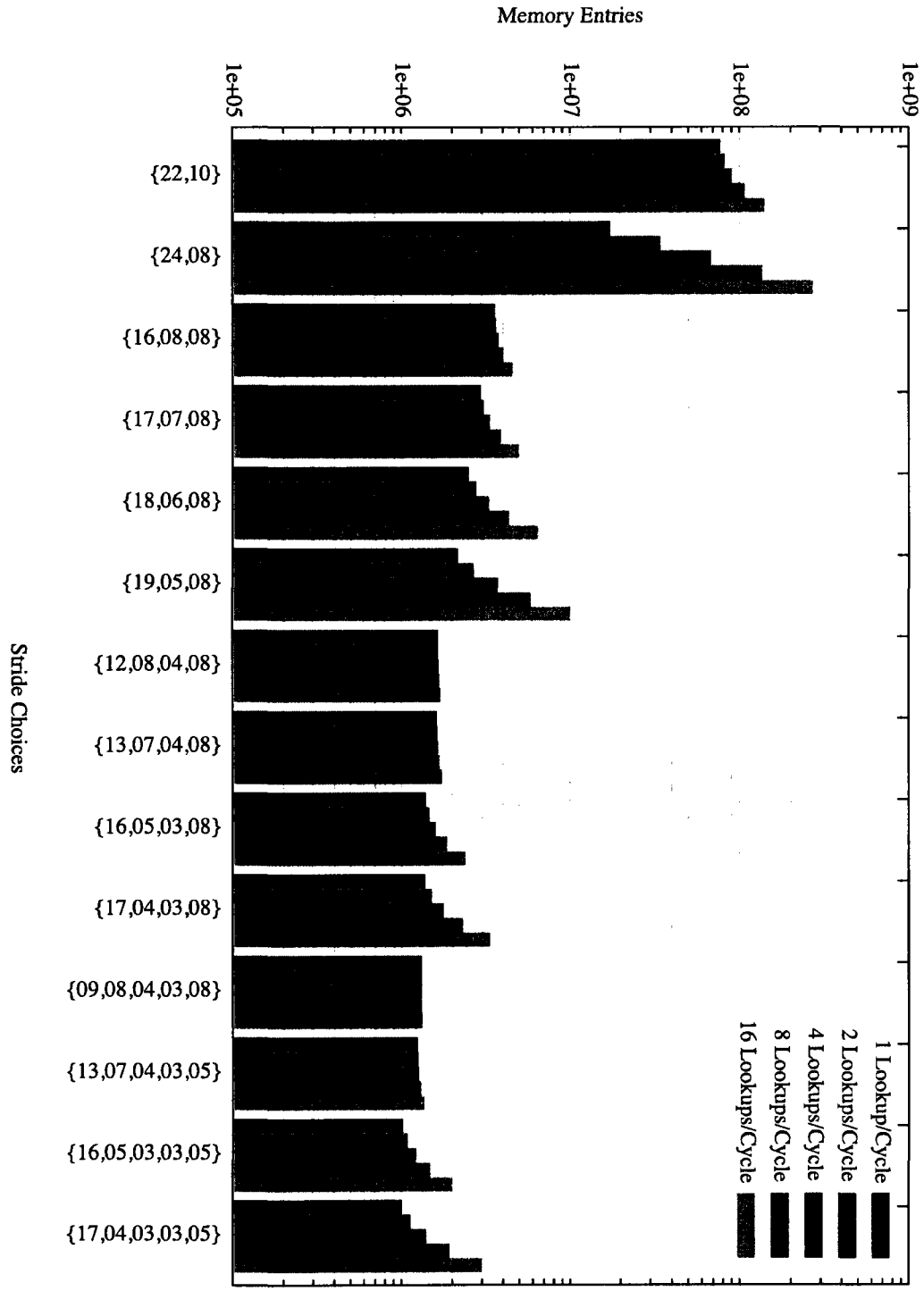


Figure 5.3: Preferred Stride Choices for Minimizing the Design's Required Memory Entries

5.3.2 Metric #1: Required Memory Bits

Figure 5.4 shows the design's preferred stride choices to minimize the number of memory bits. Not shown on this graph is the result for a single stride of 32 bits which requires $55,834,574,856 \times L$ memory bits, where L_{max} is the maximum number of simultaneous lookups per cycle supported.

It's clear that the same trends and observations that apply to the preferred stride choices for reducing memory entries also apply to the preferred stride choices for reducing memory bits. While a lot of the preferred stride choices for reducing memory bits are the same as those for reducing memory entries there are some slight differences; implying that counting memory entries is not a perfect approximation for counting memory bits.

Based on the number of required memory bits, the preferred stride choice for up to 5 strides and 16 lookups per cycle is $\{10,08,03,03,08\}$ with $16,980,853$. This also offers a factor of $50,000$ reduction in memory bits over the single stride of 32 bits for the same 16 lookups per cycle. Just as with the memory entries metric, without any other constraints taken into account, it always makes sense to increase the number of parallel lookups using this metric as the incremental cost of an additional first stride memory will always be less than the incremental benefit of an additional lookup per cycle. This approach is not practical, however, as will be shown in Subsection 5.3.3.

5.3.3 Metric #3: Required Design Area

Figure 5.5 shows the design's preferred stride choices to minimize the design's area, which includes not only the memory cell area, but the row and column overheads as well. Not shown on this graph is the result for a single stride of 32 bits, which requires $20,680 \times L\ mm^2$, where L_{max} is the maximum number of simultaneous lookups per cycle supported.

One of the most apparent differences between the results of this metric and the previous two is that larger first strides and smaller second strides are far less favorable for producing the smallest chip area. The reason for this is because the chip

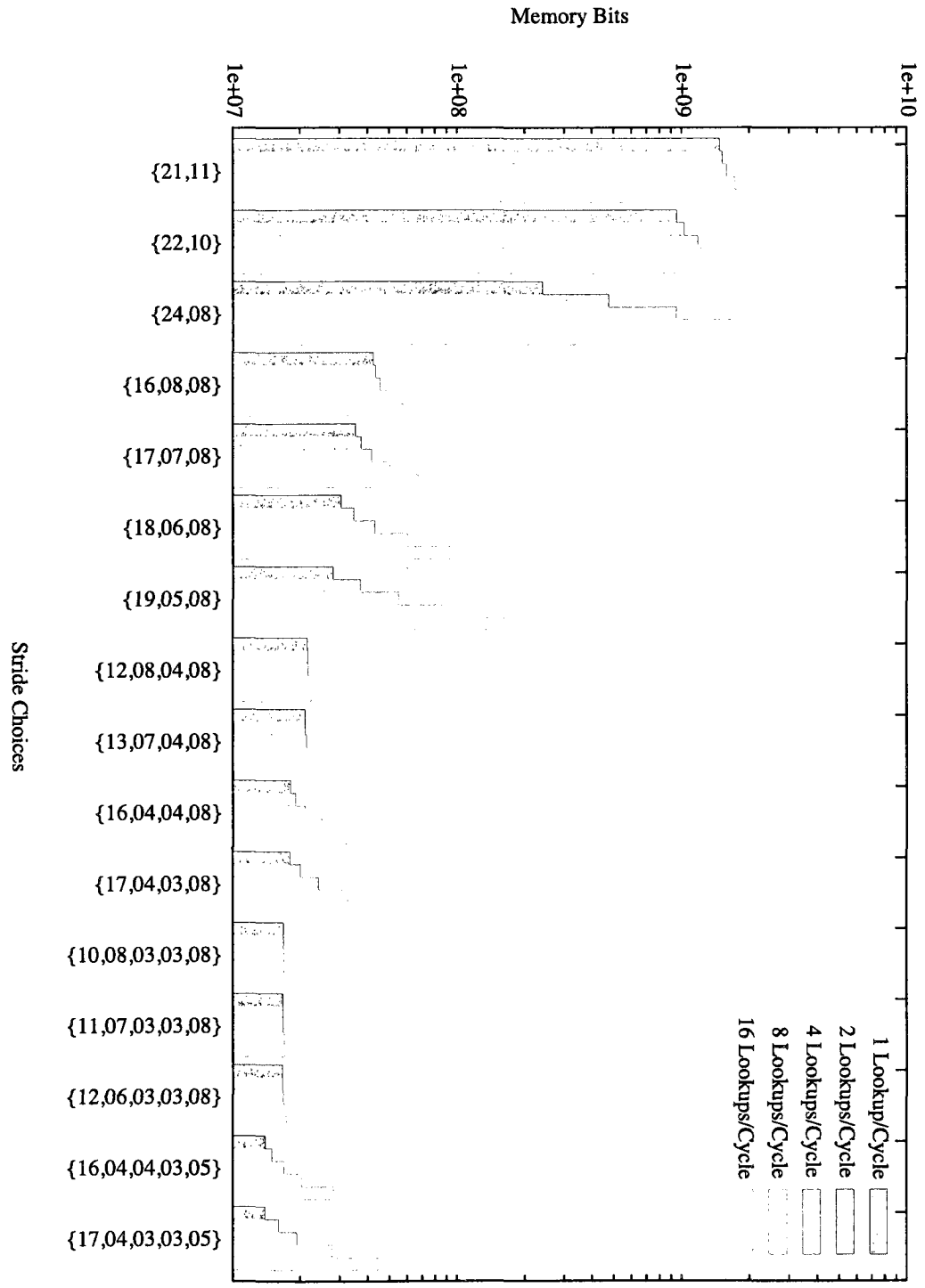


Figure 5.4: Preferred Stride Choices for Minimizing the Design's Required Memory Bits

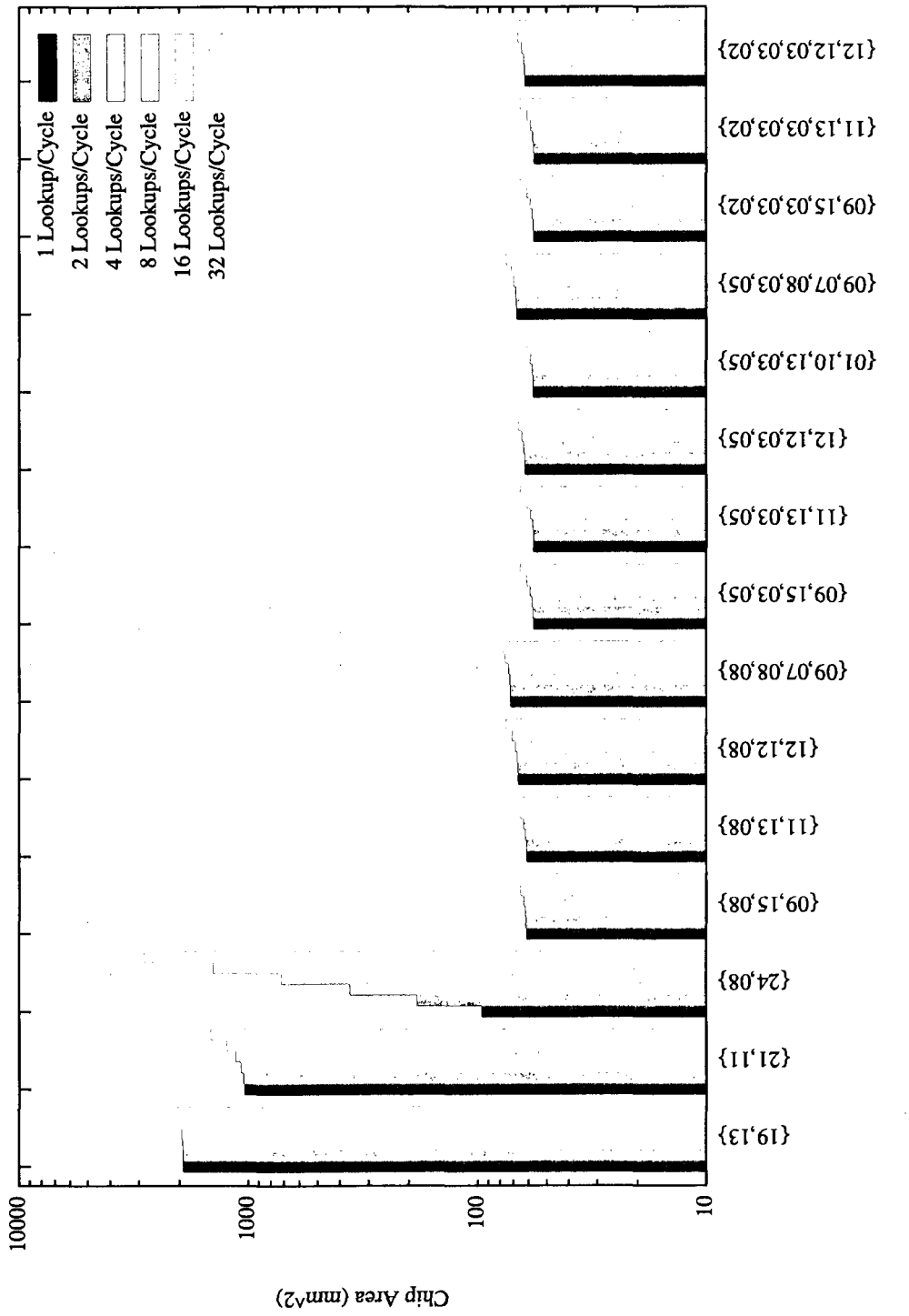


Figure 5.5: Preferred Stride Choices for Minimizing the Design's Chip Area

area metric takes into account the fact that there's a fixed width and height overhead added on to each memory, as explained in Subsection 5.2.3. This overhead makes it extremely inefficient to create small memories that store only a few hundred bits or less, and heavily favors larger memories that can amortize the cost over a much larger number of bits. Thus, while this actually makes large first stride memories more attractive, it also makes having lots of small second stride memories prohibitively expensive. Accordingly the preferred stride choices usually partition the first 24 bits fairly evenly into two larger strides, such as $\{11,13\}$.

The reduced size of the first stride also means that adding an extra first stride memory, to support an additional lookup per cycle, is a much cheaper incremental cost. Preferred stride choices such as $\{11,13,03,05\}$ remain attractive from 1 lookup per cycle all the way to 32 lookups per cycle, with smaller first stride choices like $\{09,15,03,05\}$ only providing a slight advantage in chip area for 16 or more lookups per cycle.

Figure 5.6 shows the expected total power consumption of each of the preferred stride choices for reducing the design's chip area. Not shown on this graph is the result for a single stride of 32 bits which requires $4,064 \times L\ W$, where L_{max} is the maximum number of simultaneous lookups per cycle supported.

As expected the preferred stride choices for reducing the design's memory chip area also have very good expected total power consumptions. This is because the often dominant leakage power consumption is directly proportional to the total design area. On the other hand, the incremental total power consumption for adding an extra lookup per cycle is more than the incremental chip area. This occurs because, while the increase in leakage power is directly proportional to the increase in area, the increase in active power consumption is directly proportional to the number of lookups per cycle, and hence increases much faster with more lookups per cycle.

Figure 5.7 shows the expected maximum throughput of each of the preferred stride choices for reducing the design's chip area. Not shown on this graph is the result for a single stride of 32 bits which is $7.83 \times L$ million lookups per second,

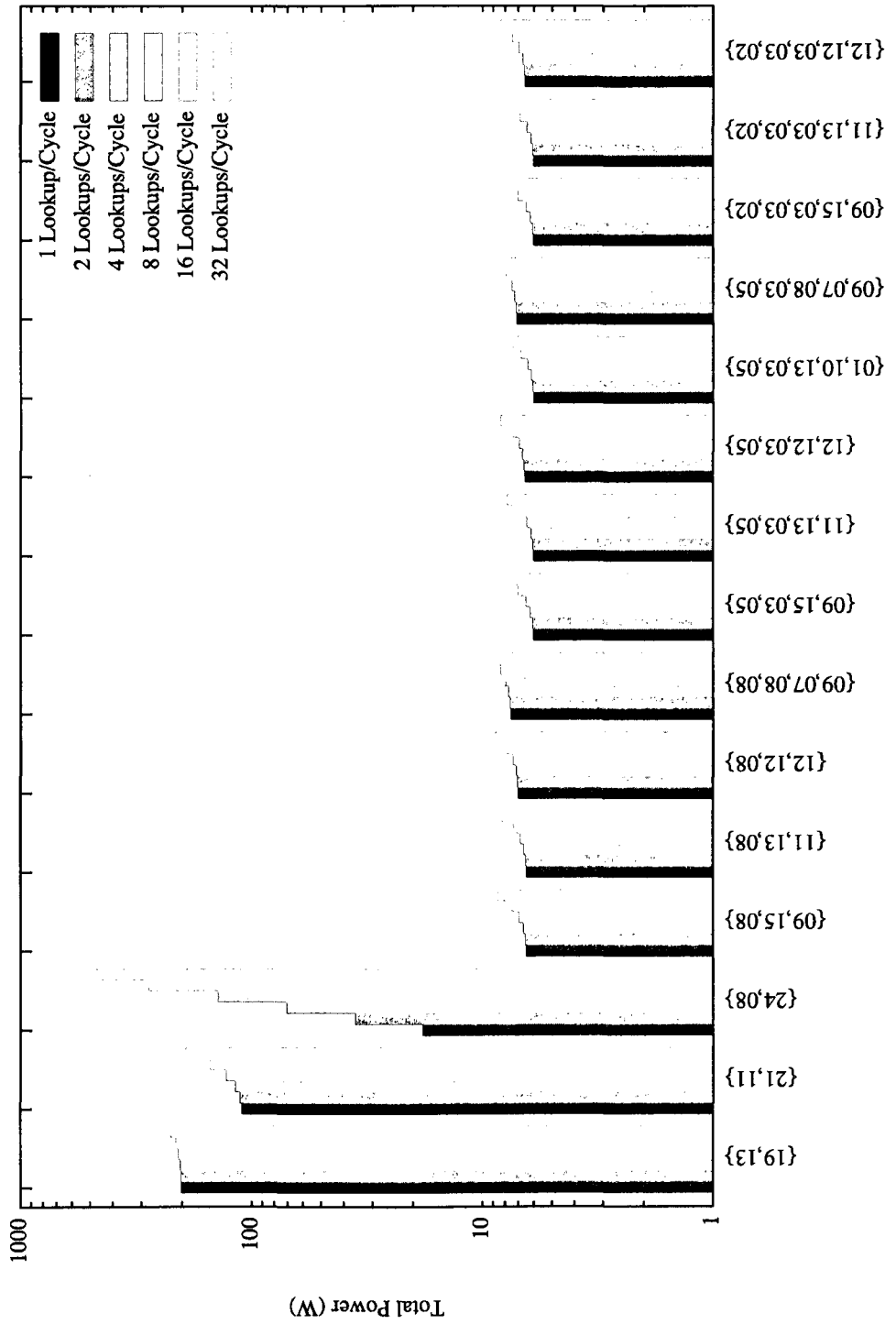


Figure 5.6: Total Power Consumption for the Preferred Stride Choices for Minimizing the Design's Chip Area

where L_{max} is the maximum number of simultaneous lookups per cycle supported.

As mentioned in Subsection 5.2.3 there are three main factors that influence the expected throughput of a stride choice for the design. The first is the largest cycle time of a stride, which dictates the maximum frequency that the design can be clocked at. It's clear from the graph that very large strides, like for $\{24,8\}$, have very large memory access times, low maximum frequencies and hence low lookup throughput. The second factor is the maximum number of lookups per cycle supported by replicating the first memory. The more lookups that can execute per cycle, the higher the expected throughput, with doubling the number of maximum lookups doubling the throughput in most cases. The final factor affecting throughput is the likelihood of conflicting lookups that need to be resolved by the arbiter. The smaller the first stride, the more likely it is that two random lookups will target the same entry, forcing a stall and decreasing the throughput. This is most apparent with the stride choice $\{01,10,13,03,05\}$ where the expected lookups per cycle reaches a maximum of 1.5, no matter what the maximum number of lookups per cycle is increased by. This is because the single bit first stride memory supports only two different lookups per cycle, and half the time the second lookup conflicts with the first and needs to be stalled. Clearly this stride choice would benefit from replicating not just its first stride memories, but its second stride memories as well. If this were done, however, this stride choice would require a lot more area to implement, and $\{11,13,03,05\}$ would remain a better choice.

Conflicts creating stalls that affect throughput are also apparent, however, as stride choices like $\{09,07,08,03,05\}$ see their incremental benefit of adding more lookups per cycle decay as the likelihood of conflicts increases in the 9 bit first stride memory. Replicating the 9 bit first stride memory from 8 times to 16 times to 32 times, for example, results in diminishing returns as the expected lookups per cycle goes from 7.84 to 14.8 to 23.8.

As mentioned in the previous two sections, the previous two metrics both favor increasing the maximum lookups per cycle arbitrarily large as the incremental cost is always less than the incremental benefit. While diminishing returns on through-

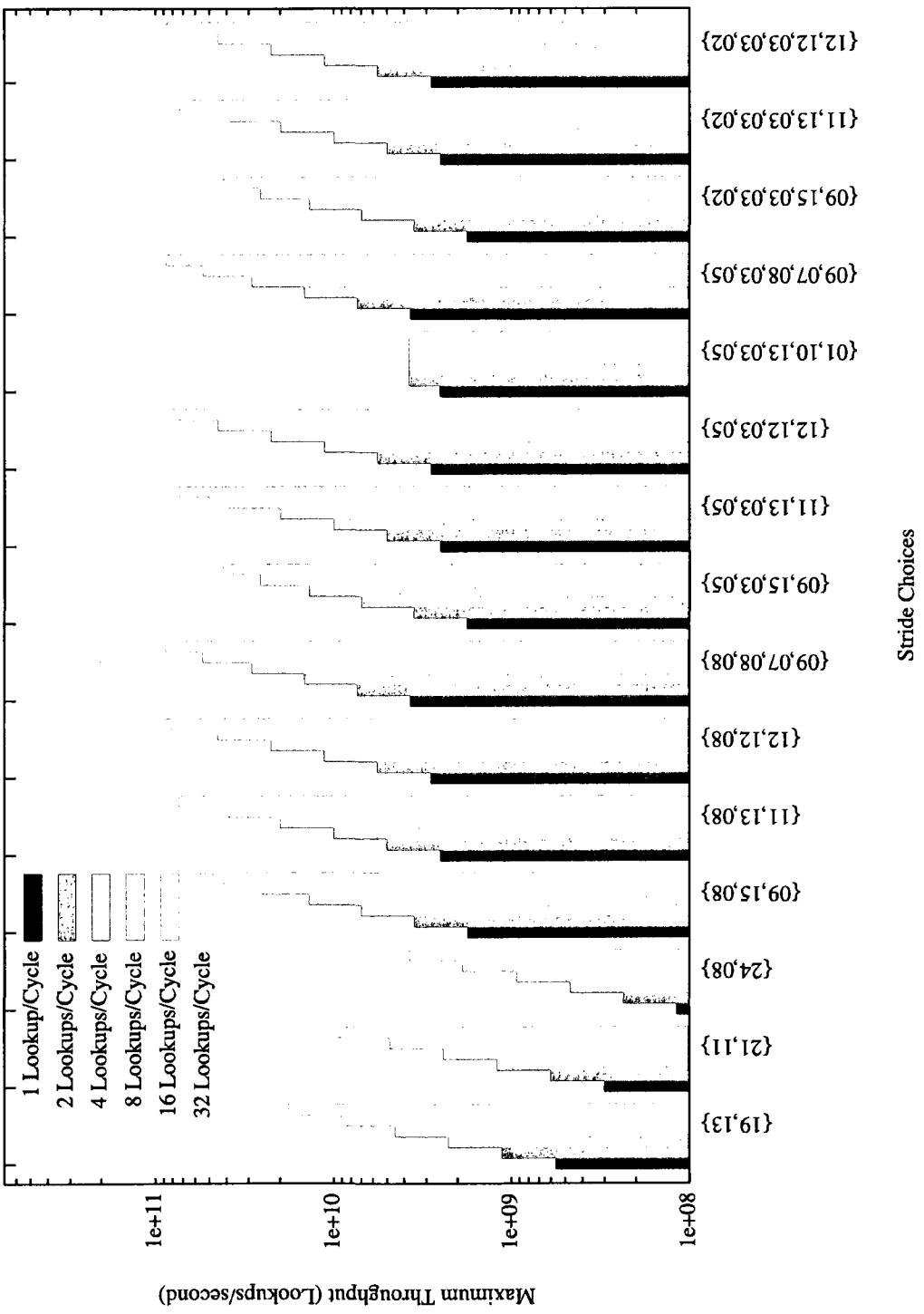


Figure 5.7: Maximum Throughput for the Preferred Stride Choices for Minimizing the Design's Chip Area

put from this third metric clearly show that this is not the case, there are several other factors that also limit the usefulness of arbitrarily large scaling as well. The most compelling of these is the added complexity to the non-memory parts of the design. Up until this point we assumed that the area, power and latency of the SRAM memory were the dominant factors in the overall design. While this is true for a small number of parallel lookups per cycle, it will no longer hold for large numbers of parallel lookups. While doubling the number of parallel lookups might only double a fraction of the chip memory area (the first stride memories) it will double the number of lookup agents and all of the routing logic required for them to access all of the existing memories. Furthermore, the added complexity of busing twice as many lookup agents to all the memories will increase the latency of every agent's access, requiring more pipeline stages to maintain the same lookup frequency. This in turn increases the power consumption of the design and overall lookup latency. The complexity of the arbiter is also dramatically increased as well.

Without implementing the design in hardware it is difficult to gauge the exact extent of these scaling limitations to parallelism. We feel that 16 lookup per cycle is a reasonable compromise: offering high throughput without overly complicating the busing and arbitration logic. With this in mind, a scatter plot of the best stride choices for minimizing the design's chip area while increasing the design's throughput was generated and is showing in Figure 5.8, for 16 lookups/cycle. From the plot, it is clear that stride choice $\{11,13,03,05\}$ offers the best throughput for the lowest chip area, while $\{09,07,08,03,05\}$ offers the smallest chip area for the highest throughput. The plot also shows that $\{12,12,03,05\}$ offers a good compromise between the two extremes. To determine which of these three stride choices is in fact the best, the ratios of throughput per area for all three were calculated, and are summarized in Table 5.4.

From Table 5.4, the design with the best throughput for a given chip area has a stride choice of $\{09,07,08,03,05\}$. It has a chip area of $71.1mm^2$, total power consumption of $7.85W$, maximum cycle time of $275ps$, average lookups per cycle of 14.8, and a maximum throughput of 53.7 billion lookups per second. These

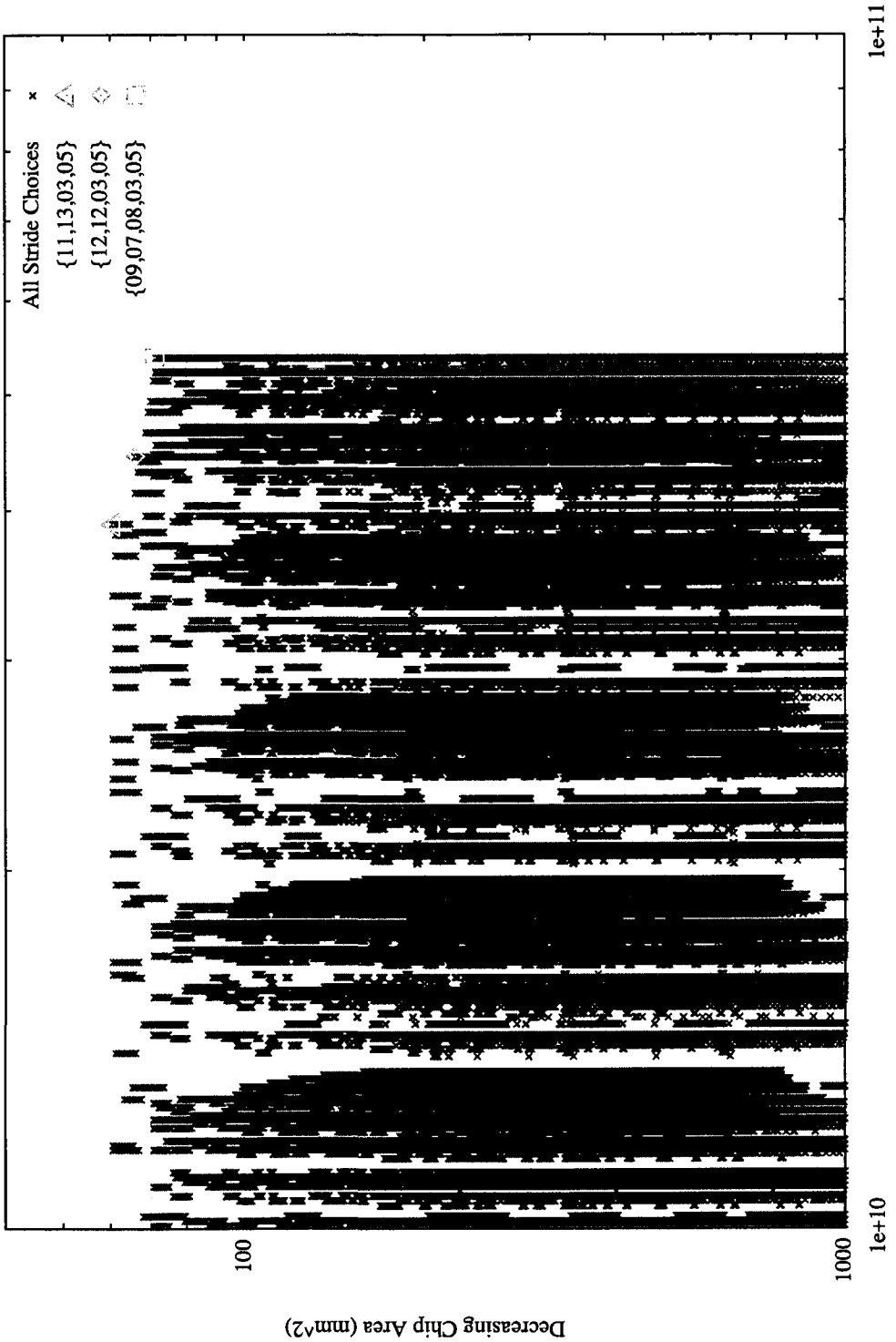


Figure 5.8: Design Chip Area vs. Maximum Throughput for All Stride Choices and 16 Lookups/Cycle

Stride Choice	Chip Area	Throughput	Lookups/sec/mm^2
$\{11,13,03,05\}$	$60.7mm^2$	38.9 billion lookups/sec	641 million
$\{12,12,03,05\}$	$66.0mm^2$	44.4 billion lookups/sec	673 million
$\{09,07,08,03,05\}$	$71.1mm^2$	53.7 billion lookups/sec	755 million

Table 5.4: Ratio of Throughput to Chip Area For the Best Three Stride Choices

	Value
Chip Area	$71.1mm^2$
Active Power	$0.40W$
Leakage Power	$7.45W$
Total Power	$7.85W$
Cycle Time	$275ps$
Lookup Latency	$1.92ns$
Maximum Lookups per Cycle	16.0
Average Lookups per Cycle	14.8
Average Lookups/Second	53.7 billion
Average Energy/Lookup	$0.157nJ$

Table 5.5: Performance of the Design with Preferred Stride Choice $\{09,07,08,03,05\}$

numbers are summarized in Table 5.5.

Chapter 6

Comparisons with TCAM

In this chapter a Ternary Content Addressable Memory (TCAM) solution to the same "backbone" routing table lookup problem is presented. The features of this TCAM solution are derived, then compared to the proposed solution.

6.1 Features of the TCAM Solution

While the features of SRAM cells implemented in 45nm process technologies have been widely published, such as in the paper described in subsection 5.2.3, there is an apparent lack of similar publications for TCAM cells. This is compounded by the fact that TCAM macros published for older processes are often very small capacity and low throughput, making them unsuitable for "backbone" routing table lookup applications.

Agrawal and Sherwood [1] address the lack of published design data by estimating the features of TCAM macros based on the desired configuration, capacity and technology process used. While their estimates are not perfect, they offer good insight into the expected features of TCAM macros in a 45nm process, providing a level playing field for comparing against the proposed design. They also provide a download-able program at http://www.cs.ucsb.edu/~arch/memmodel/ (Note: "memmodel" and not "mem-model" as in the paper) that takes in the desired TCAM parameters and process technology size and produces all of the relevant feature and performance estimates. This program was used to produce a lot of the parameters

presented in the following paragraphs.

From subsection 5.1.2 the largest "backbone" routing table available had 222,728 prefixes on May 27, 2007. A TCAM solution to the "backbone" routing table lookup problem would therefore need a capacity of 222,728 rows of 32 bit entries to store all of the required prefixes. Matches to each search would need to be priority encoded to determine the longest matching prefix, whose entry location would then be used to address a 222,728 entry SRAM that stored each 8 bit forwarding port number. For higher throughput, the three steps of TCAM search, priority encoding and forwarding port number lookup are all done in separate pipeline stages.

While the TCAM parameter calculating program does not provide estimates of the entire TCAM and priority encoder area, it does estimate a TCAM cell's size by linearly scaling a reference $0.18\mu m$ process TCAM. A $45nm$ process TCAM cell's width and height are estimated to be $4.33 \times process/0.18 = 4.33 \times 0.045/0.18 = 1.0825\mu m$ and $4.05 \times process/0.18 = 4.05 \times 0.045/0.18 = 1.0125\mu m$ respectively, for an area of $1.0825\mu m \times 1.0125\mu m = 1.096\mu m^2$. Thus a TCAM with 222,728 rows of 32 bits each has an area of $7.81mm^2$ from its bit cells alone, not counting sense amplifiers, row decoders, column decoders, bank overhead and priority encoding.

The TCAM parameter calculating program takes in the number of banks to divide the TCAM into. Since simply adding additional rows to a TCAM increases the length and hence capacitance of its search and bit lines, the performance of a TCAM macro cannot scale with its capacity. Thus, it is common to divide a TCAM into many smaller connected banks, trading added routing and priority encoding complexity for increased performance and similar capacity. To determine the appropriate number of banks for the TCAM solution, the search throughput and active power consumption for various numbers of banks are plotted in Figure 6.1.

The search throughput and active power consumption increase fairly equally until 256 banks, at which point the latency of the priority encoder becomes larger than the latency of the search itself. Increasing the number of banks past this point, therefore, offers no improvement in throughput. The search power does still in-

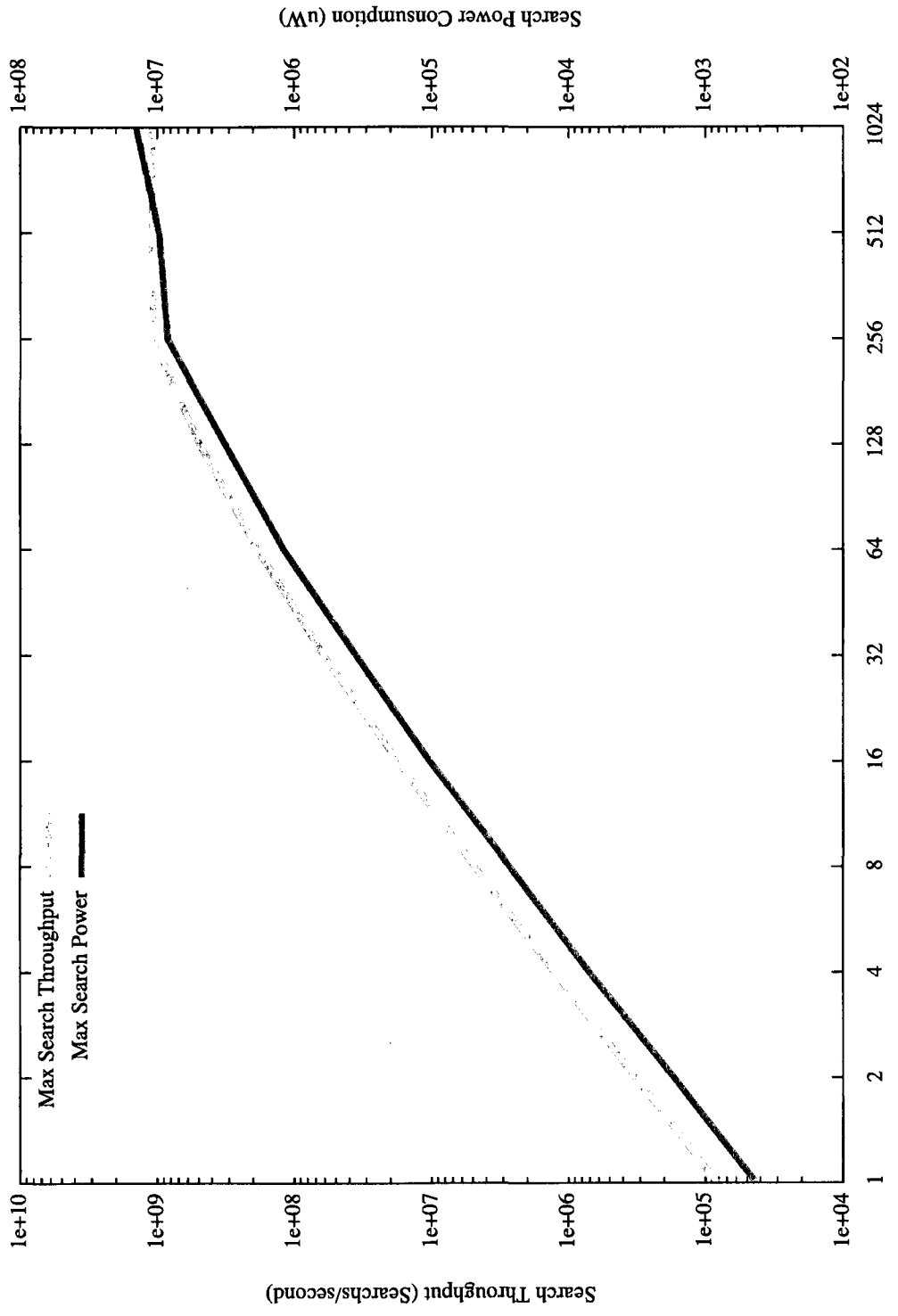


Figure 6.1: TCAM Search Throughput and Active Power Consumption for Various Numbers of Banks

Feature	Value
Memory Organization	$222,728 \times 32$
Memory Rows	1741
Memory Columns	1024
Memory Height	$0.786mm$
Memory Width	$0.922mm$
Memory Area	$0.724mm^2$
Memory Active Power	$66.4mW$
Memory Leakage Power	$75.9mW$
Memory Total Power	$142.3mW$
Memory Max Access Time	$803ps$

Table 6.1: Features of the Required Forwarding Next Port SRAM

crease slightly despite no increase in searches per second, however, because of the added power dissipated in the more complicated bank routing. Since adding more banks past 256 adds no additional throughput at the cost of added search power, 256 banks was selected for the TCAM solution. This gives the TCAM an estimated search latency of $930ps$ and an active power consumption of $7.83nJ/search$.

To calculate the leakage power of a TCAM, the program uses an internal list of technology parameters. Unfortunately, the program only has these parameters for $0.18\mu m$, $0.13\mu m$, $0.10\mu m$, and $0.07\mu m$ processes. The leakage power of the TCAM solution is plotted for each of these processes in Figure 6.2. It's clear from the graph that the leakage power of the TCAM solution steadily increases as the process technology size decreases, although predicting exactly what the value might be for $45nm$ is not straight-forward. As a very conservative estimate, the leakage power of 5.17W for the $70nm$ process is used to represent the TCAM solution.

The same model presented in subsection 5.2.3 is used to estimate the features of the required forwarding port number SRAM. Table 6.1 summarizes these estimated features.

This TCAM solution has a total TCAM bit cells area of $7.81mm^2$ and a forwarding SRAM area of $0.724mm^2$, for a total minimum area of $8.53mm^2$. The total latency of the TCAM search, priority encoding and forwarding SRAM lookup stages are $939ps$, $918ps$, and $803ps$ respectively, for a minimum cycle time of $939ps$ and

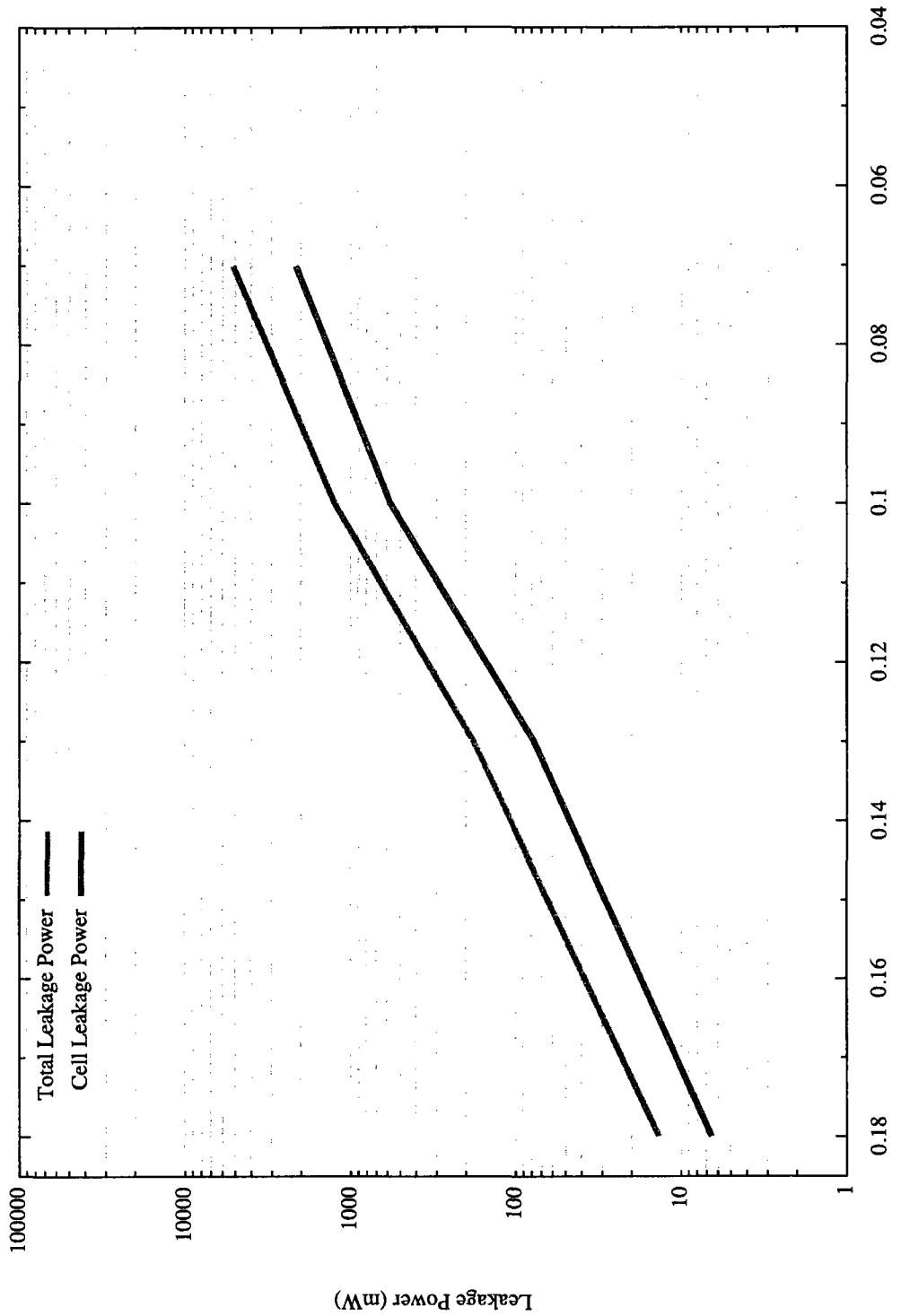


Figure 6.2: TCAM Leakage Power for Various processes

Feature	Value
TCAM Organization	256 banks of 871 words by 32 bits
TCAM Minimum Area	$8.53mm^2$
TCAM Active Power	$8.40W$
TCAM Leakage Power	$5.25W$
TCAM Total Power	$13.7W$
TCAM Cycle Time	$939ps$
TCAM Latency	$2.82ns$
TCAM Lookups/Second	1.06 billion
TCAM Energy/Lookup	$12.9nJ$

Table 6.2: Features of the TCAM Solution

a maximum throughput of 1.06 billion lookups/second. The search active power of the TCAM cells and priority encoder is therefore $8.33W$, combining with the active power of the forwarding SRAM ($66.4mW$) for a total search active power of $8.40W$. The minimum leakage power of the TCAM cells and priority encoder is $5.17W$, combining with the leakage power of the forwarding SRAM ($75.9mW$) for a total leakage power of $5.25W$. This means the entire TCAM solution has a total power of $13.7W$. A summary of the features of the entire TCAM solution is shown in Table 6.2.

6.2 Comparison of the TCAM Solution

From subsection 5.3.3 the hardware trie design proposed in this thesis has an average lookups per second of 53.7 billion. For a comparable search throughput, the TCAM solution would have to be replicated over 50 times, requiring a total area of at least $427mm^2$ and a total search power of $685W$. Clearly, with 16.7% the area and 1.1% the power, per lookup, for the same throughput, the proposed hardware trie design is a dramatic improvement over TCAM. These results are summarized in Table 6.3.

Feature	TCAM Solution	Proposed Solution	% of TCAM Value
Chip Area	$427mm^2$	$71.1mm^2$	16.7%
Active Power	$420W$	$0.40W$	0.01%
Leakage Power	$263W$	$7.45W$	2.83%
Total Power	$685W$	$7.85W$	1.15%
Cycle Time	$939ps$	$275ps$	29.3%
Latency	$2.82ns$	$1.92ns$	68.1%
Lookups/Second	53.0 billion	53.7 billion	101%
Energy/Lookup	$12.9nJ$	$0.157nJ$	1.22%

Table 6.3: Feature Differences Between TCAM Solution and the Proposed Solution

Chapter 7

Conclusion

This thesis presents a solution to the routing table lookup problem for "backbone" routers. Previously published software and hardware solutions lack the scalability required to keep pace with the ever growing demands placed on the Internet's "backbone" routers. The proposed design leverages the power efficiency and low cycle times of SRAM to construct a pipelined fixed-stride hardware trie on a chip. An innovative design allows multiple lookups to be done per cycle, providing incredibly high throughput with very low power consumption. Special design considerations ensure that updating the routing table is straight-forward and well bounded in even the worst case.

The design was implemented in VHDL to validate it, not only through functional simulation, but also through an FPGA implementation. Several different metrics were developed to evaluate the pros and cons of different stride choices using real "backbone" routing tables, with chip area estimation proving to be the most accurate. A stride choice of $\{09,07,08,03,05\}$ with 16 lookups/cycle proved to be the preferred in terms of chip area, power and throughput, boasting an average of 53.7 billion lookups/second, with only $7.85W$ of total power consumption, and a chip area of $71.1mm^2$. This solution is far more efficient than the industry standard TCAM; It requires 16.7% of the TCAM chip area and 1.1% of the power, per lookup, for the same throughput.

7.0.1 Future Work

The next step in this research is to implement and validate the proposed design in hardware. Different pipelining schemes could be experimented with during this process to help increase throughput at the cost of additional power and latency. The benefits and costs of adding a tag number to each lookup request could also be explored.

Another idea would be to look into improved ways of implementing both really small and really large memories. A really small memory could be replaced by a register file, for example, taking up less chip area than a handful of bit cells and all of the overhead sense amplifiers and decoders of an SRAM.

Other possibilities for improving throughput exist as well. By gaining access to real "backbone" packet to destination address traces, different arbiter designs could be more accurately evaluated, possibly motivating a more complicated arbiter to improve throughput. A small cache of previous lookups for the design might also offer some advantages, both by handling some additional lookups, and by reducing the number of conflicts that cause stalls.

Finally, the design could be extended to handle IPv6 addresses or more packet fields in addition to the destination address. While these sparser address spaces are not idea for trie based designs, the proposed design may prove to still be fairly efficient.

Bibliography

[1] B. Agrawal and T. Sherwood. Ternary cam power and delay model: Extensions and uses. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(5):554–564, May 2008.

[2] S. Ahmand and R. Mahapatra. M-trie: an efficient approach to on-chip logic minimization. In *IEEE/ACM International Conference on Computer Aided Design, 2004 (ICCAD-2004)*, pages 428–435, 2004.

[3] M.J. Akhbarizadeh and M. Nourani. Throughput increase in packet forwarding engines using adaptive block-selection scheme. *IEEE Communications Letters*, 9(9):838–840, 2005.

[4] M.J. Akhbarizadeh, M. Nourani, and C.D. Cantrell. Segregating the encompassing prefixes to enhance the performance of packet forwarding engines. In *IEEE Global Telecommunications Conference, 2004 (GLOBECOM'04)*, volume 3, pages 1612–1616, December 2004.

[5] M.J. Akhbarizadeh, M. Nourani, D.S. Vijayasarathi, and P.T. Balsara. Pcam: a ternary cam optimized for longest prefix matching tasks. In *Proceedings on the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004 (ICCD 2004)*, pages 6–11, October 2004.

[6] I. Arsovski, T. Chandler, and A. Sheikholeslami. A ternary content-addressable memory (tcam) based on 4t static storage and including a current-race sensing scheme. *IEEE Journal of Solid-State Circuits*, 38(1):155–158, January 2003.

[7] R.W. Baldwin and E. Ng. Technique to eliminate sorting in ip packet forwarding devices. In *Proceedings on the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004 (ICCD 2004)*,

pages 554–559, October 2004.

[8] M. Berger. Ip lookup with low memory requirement and fast update. In *Workshop on High Performance Switching and Routing, 2003 (HPSR)*, pages 287–291, June 2003.

[9] Jianjian Bian and S.P. Khatri. Ip routing table compression using espresso-mv. In *The 11th IEEE International Conference on Networks, 2003 (ICON 2003)*, pages 167–172, October 2003.

[10] V. Cerf, Y. Dalal, and C. Sunshine. Specification of internet transmission control program. Technical Report RFC675, The Internet Engineering Task Force, December 1974.

[11] R.C. Chang and B.H. Lim. Efficient ip routing table lookup scheme. *IEE Proceedings - Communications*, 149(2):77–82, April 2002.

[12] Yeim-Kuan Chang. A 2-level tcam architecture for ranges. *IEEE Transactions on Computers*, 55(12):1614–1629, December 2006.

[13] T. Chiueh and P. Pradhan. High-performance ip routing table lookup using cpu caching. In *Proceedings on the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99)*, volume 3, pages 1421–1428, March 1999.

[14] K. Choi and W.S. Adams. Vlsi implementation of a 256256 crossbar interconnection network. In *Proceedings on the Sixth International Parallel Processing Symposium, 1992*, pages 289–293, March 1992.

[15] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. Technical Report RFC2460, The Internet Engineering Task Force, December 1998.

[16] Yaping Deng, Ke Yin, and Lei Yu. High speed ip routing lookup algorithm based on ram and tcam. In *Proceedings on the 2006 International Conference on Communications, Circuits and Systems*, volume 3, pages 1677–1680, June 2006.

[17] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking*, 14(2):397–

409, April 2006.

[18] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.

[19] V. Fuller and T. Li. Classless inter-domain routing (cidr): The internet address assignment and aggregation plan. Technical Report RFC4632, The Internet Engineering Task Force, August 2006.

[20] N. Futamura, R. Sangireddy, S. Aluru, and A.K. Somani. Scalable, memory efficient, high-speed lookup and update algorithms for ip routing. In *Proceedings on the The 12th International Conference on Computer Communications and Networks, 2003 (ICCCN 2003)*, pages 257–263, October 2003.

[21] B. Gamache, Z. Pfeffer, and S.P. Khatri. A fast ternary cam design for ip networking applications. In *Proceedings on the The 12th International Conference on Computer Communications and Networks, 2003 (ICCCN 2003)*, pages 434–439, October 2003.

[22] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proceedings on the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '98)*, volume 3, pages 1240–1247, March-April 1998.

[23] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar. Chisel: A storage-efficient, collision-free hash-based network processing architecture. In *33rd International Symposium on Computer Architecture, 2006 (ISCA '06)*, pages 203–215, 2006.

[24] R. Hinden and S. Deering. Ip version 6 addressing architecture. Technical Report RFC4291, The Internet Engineering Task Force, February 2006.

[25] Ilion Yi-Liang Hsiao and Chein-Wei Jen. A new hardware design and fpga implementation for internet routing towards ip over wdm and terabit routers. In *Proceedings on the The 2000 IEEE International Symposium on Circuits and Systems, 2000 (ISCAS 2000 Geneva)*, volume 1, pages 387–390, May 2000.

[26] Nen-Fu Huang and Shi-Ming Zhao. A novel ip-routing lookup scheme and

hardware architecture for multigigabit switching routers. *IEEE Journal on Selected Areas in Communications*, 17(6):1093–1104, June 1999.

[27] Geoff Huston. As65000 bgp routing table analysis report. http://bgp.potaroo.net/as2.0/bgp-active.html, September 2007.

[28] S. Kasnavi, V.C. Gaudet, P. Berube, and J.N. Amaral. A hardware-based longest prefix matching scheme for tcams. In *IEEE International Symposium on Circuits and Systems, 2005 (ISCAS 2005)*, pages 3339–3342, May 2005.

[29] S. Kaxiras and G. Keramidas. Ipstash: a power-efficient memory architecture for ip-lookup. In *Proceedings on the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003 (MICRO-36)*, pages 361–372, December 2003.

[30] T. Kijkanjanarat and H.J. Chao. Fast ip lookups using a two-trie data structure. In *Global Telecommunications Conference, 1999. GLOBECOM'99*, volume 2, pages 1570–1575, 1999.

[31] M. Kobayashi and T. Murase. A processor based high-speed longest prefix match search engine. In *2001 IEEE Workshop on High Performance Switching and Routing*, pages 233–239, May 2001.

[32] M. Kobayashi, T. Murase, and A. Kuriyama. A longest prefix match search engine for multi-gigabit ip processing. In *2000 IEEE International Conference on Communications (ICC 2000)*, volume 3, pages 1360–1364, June 2000.

[33] T. Kocak and F. Basci. A low-power network search engine based on statistical partitioning. In *2004 Workshop on High Performance Switching and Routing (HPSR)*, pages 264–268, 2004.

[34] B. Lampson, V. Srinivasan, and G. Varghese. Ip lookups using multiway and multicolumn search. *IEEE/ACM Transactions on Networking*, 7(3):324–334, June 1999.

[35] Hyesook Lim and Yeojin Jung. A parallel multiple hashing architecture for ip address lookup. In *2004 Workshop on High Performance Switching and Routing (HPSR)*, pages 91–95, 2004.

[36] Hyesook Lim and Bomi Lee. A new pipelined binary search architecture for

ip address lookup. In *2004 Workshop on High Performance Switching and Routing (HPSR)*, pages 86–90, 2004.

[37] Hyesook Lim, Ji-Hyun Seo, and Yeo-Jin Jung. High speed ip address lookup architecture using hashing. *IEEE Communications Letters*, 7(10):502–504, October 2003.

[38] Hyesook Lim, Bomi Lee, and Wonjung Kim. Binary searches on multiple small trees for ip address lookup. *IEEE Communications Letters*, 9(1):75–77, January 2005.

[39] Dong Lin, Yue Zhang, Chengchen Hu, Bin Liu, Xin Zhang, and Derek Pao. Route table partitioning and load balancing for parallel searching with tcams. In *IEEE International Parallel and Distributed Processing Symposium, 2007 (IPDPS 2007)*, pages 1–10, March 2007.

[40] Po-Chou Lin and Chung-Ju Chang. A priority tcam ip-routing lookup scheme. *IEEE Communications Letters*, 7(7):337–339, July 2003.

[41] Huan Liu. Reducing routing table size using ternary-cam. In *Hot Interconnects 9, 2001*, pages 69–73, 2001.

[42] A.J. McAuley and P. Francis. Fast routing table lookup using cams. In *Proceedings on the Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies: Networking: Foundation for the Future (INFOCOM '93)*, pages 1382–1391, March-April 1993.

[43] Tan Mingfeng and Gong Zhenghu. High speed ip lookup algorithm with scalability and parallelism based on cam array and tcam. In *2004 IEEE International Conference on Communications*, volume 2, pages 1085–1089, June 2004.

[44] H. Mohammadi, N. Yazdani, B. Robatmili, and M. Nourani. Hasil: hardware assisted software-based ip lookup for large routing tables. In *The 11th IEEE International Conference on Networks, 2003 (ICON 2003)*, pages 99–104, October 2003.

[45] N. Mohan and M. Sachdev. Low-capacitance and charge-shared match lines for low-energy high-performance tcams. *IEEE Journal of Solid-State Circuits*,

42(9):2054–2060, September 2007.

[46] S. Nilsson and G. Karlsson. Ip-address lookup using lc-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[47] Seung-Hyun Oh and Jong-Suk Ahn. Bit-map trie: a data structure for fast forwarding lookups. In *IEEE Global Telecommunications Conference, 2001 (GLOBECOM'01)*, volume 3, pages 1872–1876, November 2001.

[48] K. Pagiamtzis and A. Sheikholeslami. A low-power content-addressable memory (cam) using pipelined hierarchical search scheme. *IEEE Journal of Solid-State Circuits*, 39(9):1512–1519, September 2004.

[49] Wooguil Pak and Saewoong Bahk. Flexible and fast ip lookup algorithm. In *IEEE International Conference on Communications, 2001 (ICC 2001)*, volume 7, pages 2053–2057, June 2001.

[50] D. Pao. Tcam organization for ipv6 address lookup. In *The 7th International Conference on Advanced Communication Technology, 2005 (ICACT 2005)*, volume 1, pages 26–31, February 2005.

[51] T.-B. Pei and C. Zukowski. Putting routing tables in silicon. *IEEE Network Magazine*, 6(1):42–50, January 1992.

[52] H. Pilo, V. Ramadurai, G. Braceras, J. Gabric, S. Lamphier, and Yue Tan. A 450ps access-time sram macro in 45nm soi featuring a two-stage sensing-scheme and dynamic power management. In *2008 IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 378–379,621, 2008.

[53] J. Postel. Internet protocol. Technical Report RFC791, The Internet Engineering Task Force, September 1981.

[54] G. Qin, S. Ata, I. Oka, and C. Fujiwara. Effective bit selection methods for improving performance of packet classifications on ip routers. In *IEEE Global Telecommunications Conference, 2002 (GLOBECOM'02)*, volume 3, pages 2350–2354, November 2002.

[55] V.C. Ravikumar and R.N. Mahapatra. Tcam architecture for ip lookup using prefix properties. *IEEE Micro*, 24(2):60–69, March-April 2004.

[56] V.C. Ravikumar, R.N. Mahapatra, and Laxmi Narayan Bhuyan. Easecam: an energy and storage efficient tcam-based router architecture for ip lookup. *IEEE Transactions on Computers*, 54(5):521–533, May 2005.

[57] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). Technical Report RFC4271, The Internet Engineering Task Force, January 2006.

[58] S. Sahni and Kun Suk Kim. Efficient construction of fixed-stride multibit tries for ip lookup. In *Proceedings on the The Eighth IEEE Workshop on Future Trends of Distributed Computing Systems, 2001 (FTDCS 2001)*, pages 178–184, November 2001.

[59] S. Sahni and Kun Suk Kim. Efficient construction of variable-stride multibit tries for ip lookup. In *Proceedings on the 2002 Symposium on Applications and the Internet. (SAINT 2002)*, pages 220–227, February 2002.

[60] R. Sangireddy and A.K. Somani. Binary decision diagrams for efficient hardware implementation of fast ip routing lookups. In *Proceedings on the Tenth International Conference on Computer Communications and Networks, 2001*, pages 12–17, October 2001.

[61] D. Shah and P. Gupta. Fast updating algorithms for tcam. *IEEE Micro*, 21(1): 36–47, January-February 2001.

[62] P. Srisuresh and K. Egevang. Traditional ip network address translator (traditional nat). Technical Report RFC3022, The Internet Engineering Task Force, January 2001.

[63] X. Sun and Y.Q. Zhao. An on-chip ip address lookup algorithm. *IEEE Transactions on Computers*, 54(7):873–885, July 2005.

[64] J. Sungkee, Sang-Hun Chung, Jung-Wan Cho, and Hyunsoo Yoon. A scalable and small forwarding table for fast ip address lookups. In *Proceedings on the 2001 International Conference on Computer Networks and Mobile Computing*, pages 413–418, October 2001.

[65] Cisco Systems. Cisco carrier routing system. PDF Document, October 2006. URL `http://www.cisco.com/application/pdf/en/us/guest/products/ps5763/c1031/cdccont_`

0900aecd800f8118.pdf.

[66] Yi Tang, Wei Lin, and Bin Liu. A tcam index scheme for ip address lookup. In *First International Conference on Communications and Networking in China, 2006 (ChinaCom'06)*, pages 1–5, October 2006.

[67] D.E. Taylor, J.W. Lockwood, T.S. Sproull, J.S. Turner, and D.B. Parlour. Scalable ip lookup for programmable routers. In *Proceedings on the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 2, pages 562–571, June 2002.

[68] N.-F. Tzeng. Spal: a speedy packet lookup technique for high-performance routers. In *International Conference on Parallel Processing, 2004 (ICPP 2004)*, volume 1, pages 284–291, 2004.

[69] M. Uga and K. Shiomoto. A fast and compact longest match prefix look-up method using pointer cache for very long network address. In *Proceedings on the Eight International Conference on Computer Communications and Networks, 1999*, pages 595–602, October 1999.

[70] J. van Lunteren. Searching very large routing tables in fast sram. In *Proceedings on the Tenth International Conference on Computer Communications and Networks, 2001*, pages 4–11, October 2001.

[71] P.-C. Wang, C.-T. Chan, R.-C. Chen, and H.-Y. Chang. Efficient entry-reduction algorithm for tcam-based ip forwarding engine. *IEE Proceedings - Communications*, 152(2):172–176, April 2005.

[72] Pi-Chung Wang, Chia-Tai Chan, and Yaw-Chung Chen. A fast ip routing lookup scheme. In *2000 IEEE International Conference on Communications (ICC 2000)*, volume 2, pages 1140–1144, June 2000.

[73] Pi-Chung Wang, Chia-Tai Chan, Shuo-Cheng Hu, and Yaw-Chung Chen. Routing interval: a new concept for ip lookups. In *Joint 4th IEEE International Conference on ATM (ICATM 2001) and High Speed Intelligent Internet Symposium, 2001*, pages 310–315, April 2001.

[74] Pi-Chung Wang, Chia-Tai Chan, Shuo-Cheng Hu, Yu-Chen Shin, and Yaw-Chung Chen. Hardware-based ip routing lookup with incremental update.

In *Proceedings on the Ninth International Conference on Parallel and Distributed Systems, 2002*, pages 183–188, December 2002.

[75] Weidong Wu and Ruixuan Wang. A tcam management scheme for ip lookups. *14th IEEE International Conference on Networks, 2006 (ICON '06)*, 1:1–4, September 2006.

[76] Weidong Wu, Bingxin Shi, and Feng Wang. Fast updating scheme of forwarding tables on tcam. In *Proceedings on the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004)*, pages 522–527, October 2004.

[77] Weidong Wu, Jian Shi, Ling Zuo, and Bingxin Shi. Power-efficient tcams for bursty access patterns. *IEEE Micro*, 25(4):64–72, July-August 2005.

[78] Lih-Chyau Wuu and Shou-Yu Pin. A fast ip lookup scheme for longest-matching prefix. In *Proceedings on the 2001 International Conference on Computer Networks and Mobile Computing*, pages 407–412, October 2001.

[79] Lih-Chyau Wuu, Kuo-Ming Chen, and Tzong-Jye Liu. A longest prefix first search tree for ip lookup. In *2005 IEEE International Conference on Communications (ICC 2005)*, volume 2, pages 989–993, May 2005.

[80] Zhen Xu, G. Damm, I. Lambadaris, and Y.Q. Zhao. Ip packet forwarding based on comb extraction scheme. In *2004 IEEE International Conference on Communications*, volume 2, pages 1065–1069, June 2004.

[81] N. Yazdani and P.S. Min. Fast and scalable schemes for the ip address lookup problem. In *Proceedings of the IEEE Conference on High Performance Switching and Routing, 2000 (ATM 2000)*, pages 83–92, June 2000.

[82] P.A. Yilmaz, A. Belenkiy, N. Uzun, N. Gogate, and M. Toy. A trie-based algorithm for ip lookup problem. In *IEEE Global Telecommunications Conference, 2000 (GLOBECOM'00)*, volume 1, pages 593–598, December 2000.

[83] F. Zane, Girija Narlikar, and A. Basu. Coolcams: power-efficient tcams for forwarding engines. In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 1, pages

42–52, March-April 2003.

[84] Kai Zheng, Chengchen Hu, Hongbin Liu, and Bin Liu. An ultra high throughput and power efficient tcam-based ip lookup engine. In *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, volume 3, pages 1984–1994, March 2004.

Appendix A

History of the Internet Protocol (IP)

While millions and millions of people use the Internet every day, very few of us know how it started or how it works. To understand the Internet, however, we also need to understand computers.

Computers speak to each other using various agreed upon methods of sharing and interpreting data called protocols, similar to how humans communicate using different languages. Often different computer tasks involve different operations, and hence use different protocols. Protocols are often organized into different levels, with lower level protocols handling very basic computer interactions, while higher level protocols offer much more specialized functionality by making use of functions provided by lower level protocols. This allows many higher level protocols to make use of the same lower level protocol and not have to worry about the low level details.

Prior to the Internet, many organizations developed their own protocols to handle interactions between their computers. A given computer was connected to a handful of other computers using the same protocol, forming what is called a network. Many different networks were formed, each using its own protocol, and each providing limited or no connectivity to other networks. The Internet as we know it today started as an effort to connect all of these different networks together. That effort culminated in the release of RFC675 in 1974 which specifies the Transmission Control Protocol (TCP) and the Internet Protocol (IP) version 1 [10].

The idea behind these new protocols was to standardize the lowest level protocols computers use to communicate with each other. Higher level protocols that provided services such as electronic mail (SMTP), news (NNTP), and file transfer (FTP) could then make use of these new low level protocols instead of defining their own. A network that implemented these low level protocols could be connected seamlessly with other similar networks, providing high level services to each other without being forced to change their existing infrastructure.

Appendix B

Process Examples

All of the following process examples use the same fixed-stride $\{4,2,2\}$ hardware prefix trie shown in Figure 3.1 on Page 35 and built from the prefixes in Table 3.1 on Page 35.

B.1 Lookup Process Examples

B.1.1 Lookup Example 1: IP Address 01101111

In this example IP address 01101111 is being looked up. In Figure B.1 the first stage lookup of 0110 in the first bank results in a pointer to the first bank in the second stage. The default data from the first bank is the *default* value so is not stored. In Figure B.2 the second stage lookup of 11 in the first bank results in a pointer to the first bank in the third stage. The default data from the first bank is port number 1 which is stored. In Figure B.3 the last stage lookup of 11 in the first bank results in a port number value of 2 which is output as the result.

B.1.2 Lookup Example 2: IP Address 11010010

In this example IP address 11010010 is being looked up. In Figure B.4 the first stage lookup of 1101 in the first bank results in a pointer to the third bank in the second stage. The default data from the first bank is the *default* value so is not stored. In Figure B.5 the second stage lookup of 00 in the third bank results in a port number value of *default* which is stored. The default data from the third bank is port number 8 which is stored. In Figure B.6 the last stage doesn't perform a lookup. Since the stored port number is the special *default* value the stored default port number is output as the result, which is 8.

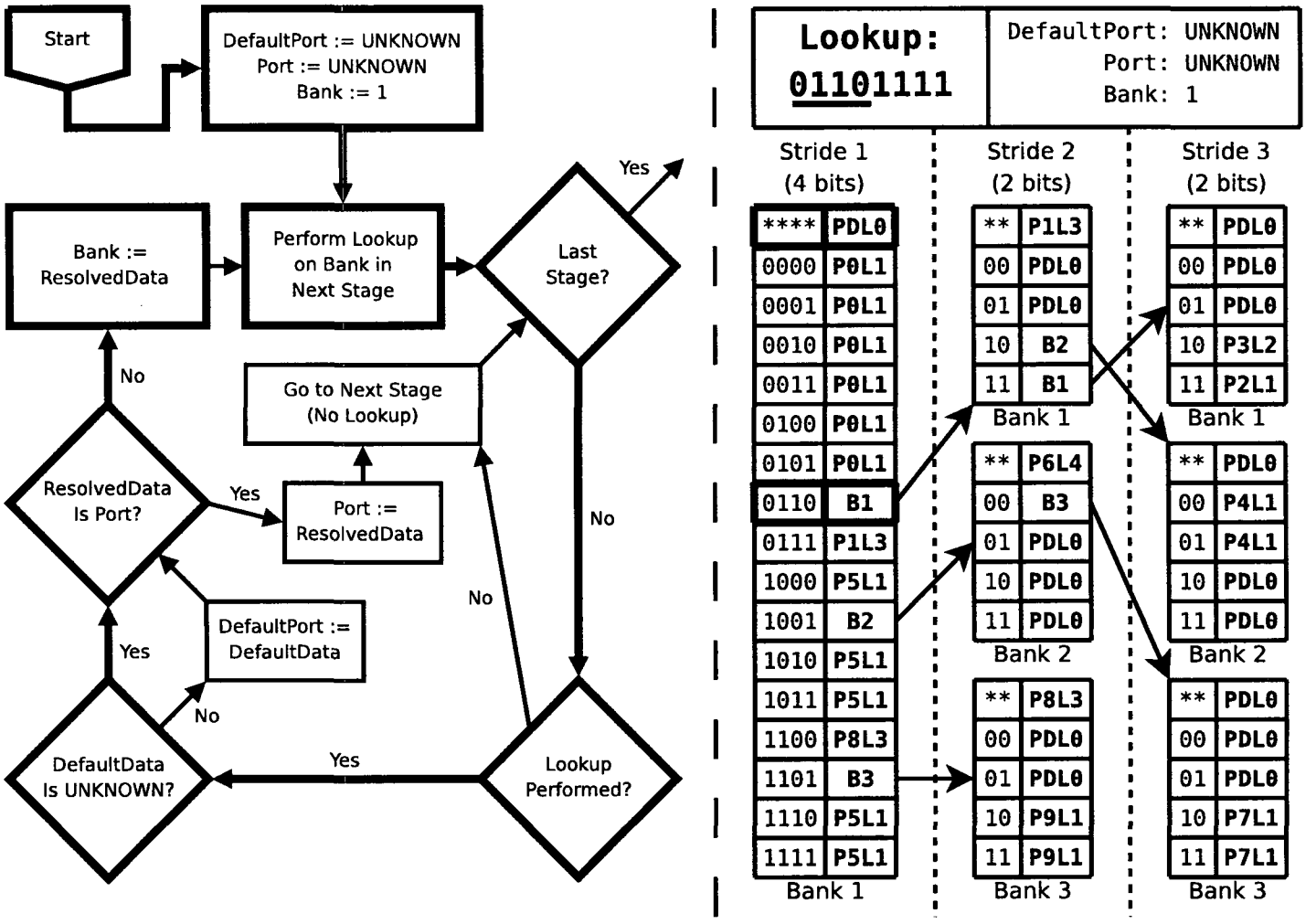


Figure B.1: Lookup Example 1: First Stage Lookup Returns A Pointer

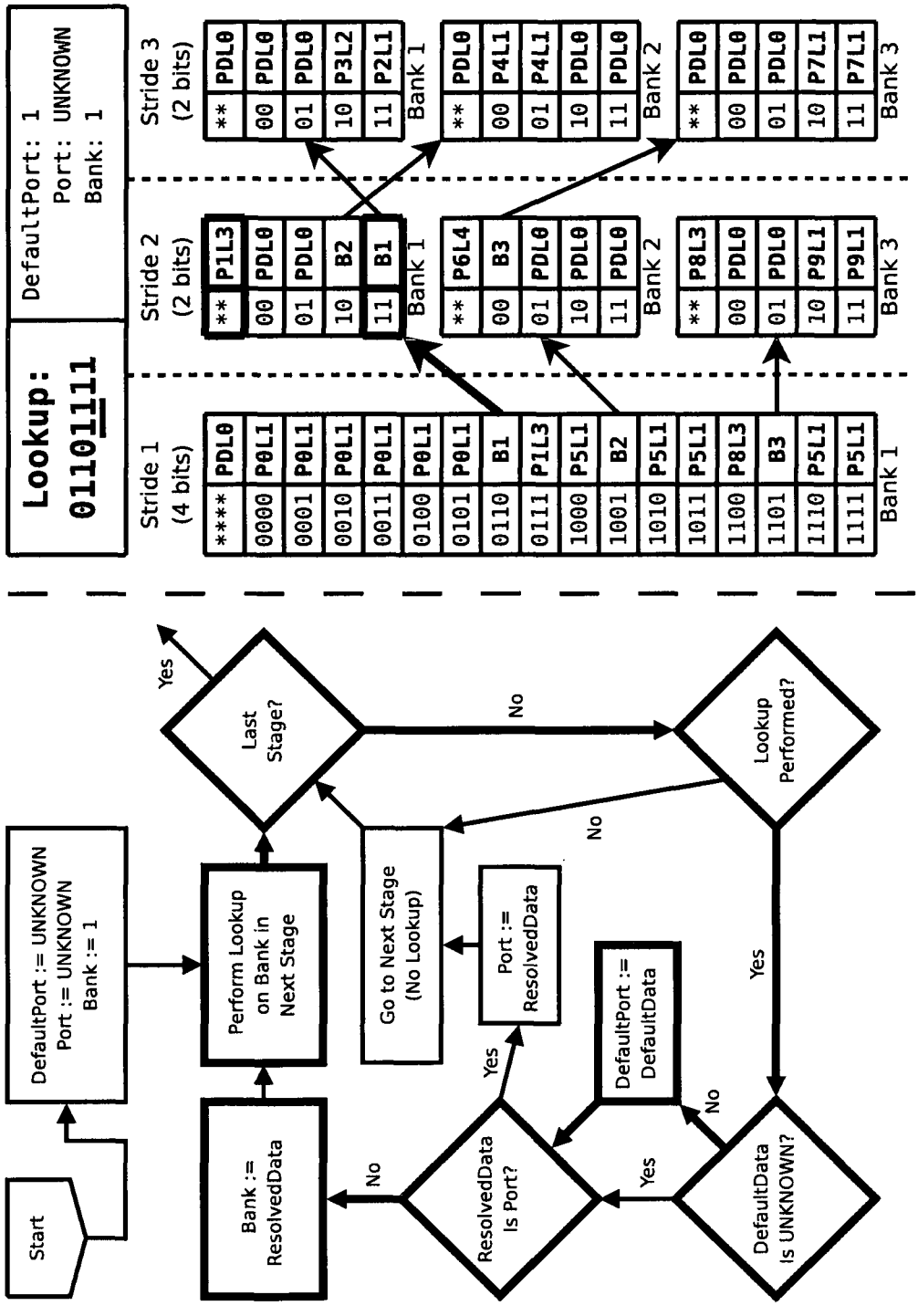


Figure B.2: Lookup Example 1: Second Stage Lookup Returns A Pointer

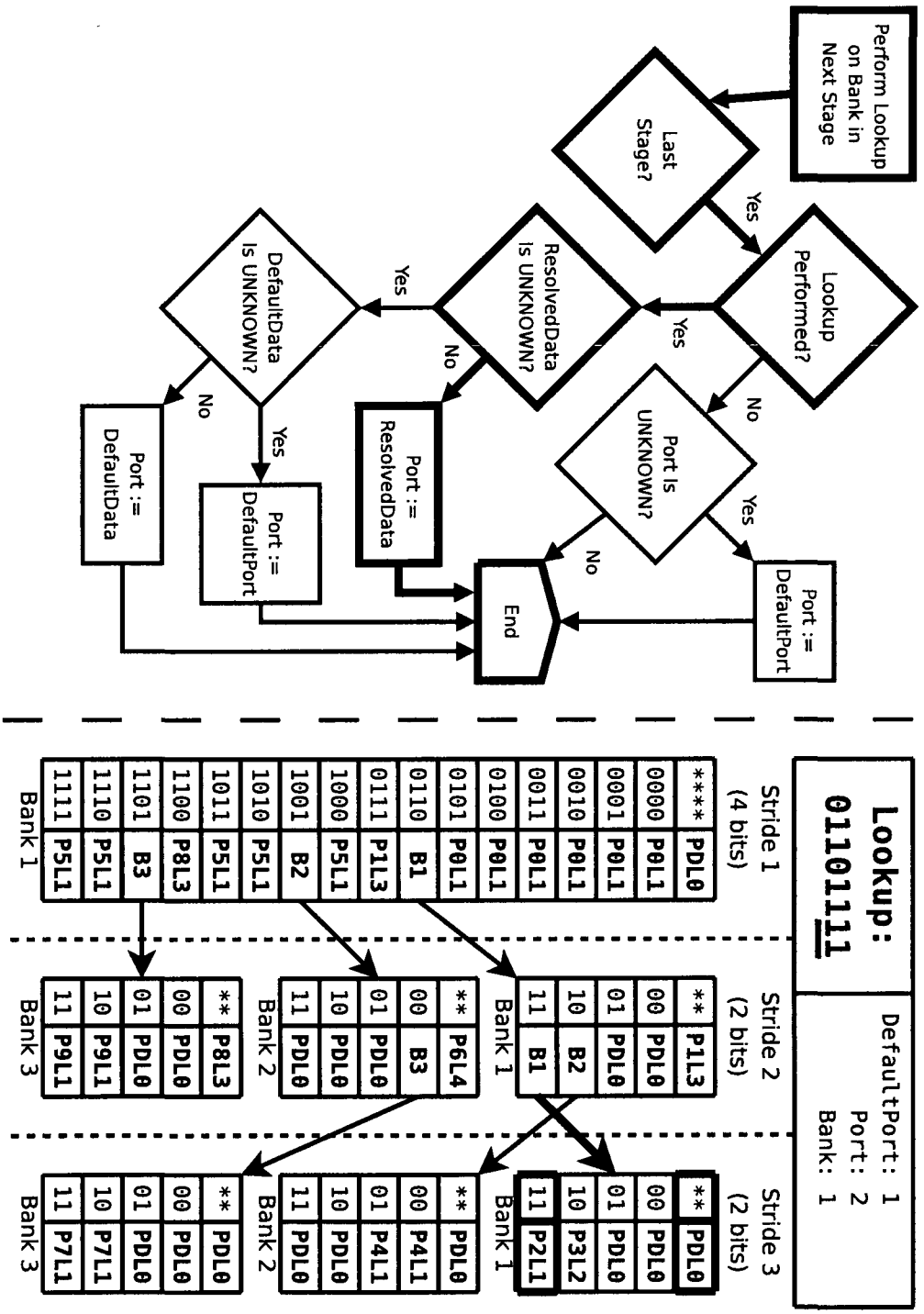


Figure B.3: Lookup Example 1: Last Stage Returns A Port Number

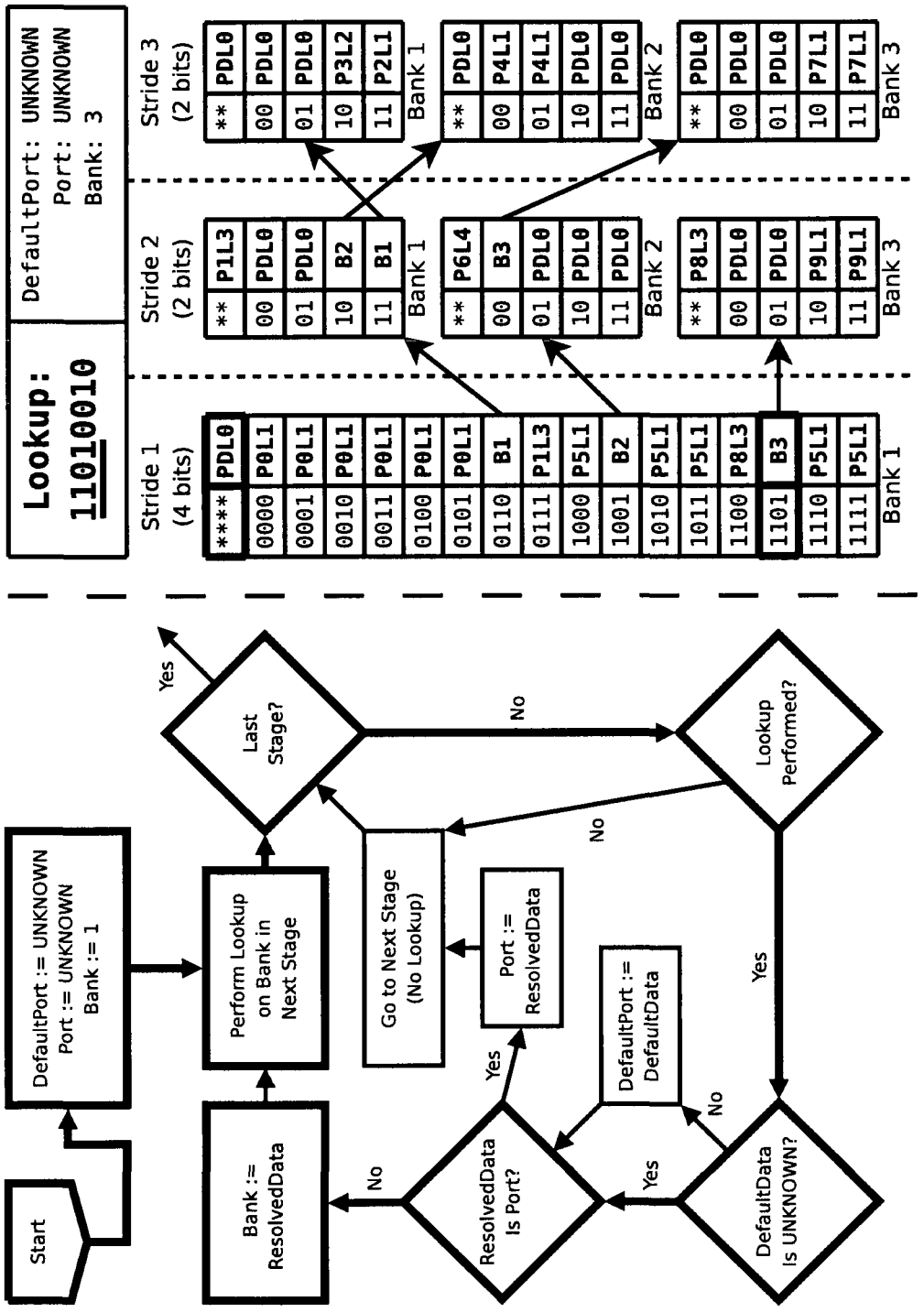


Figure B.4: Lookup Example 2: First Stage Lookup Returns A Pointer

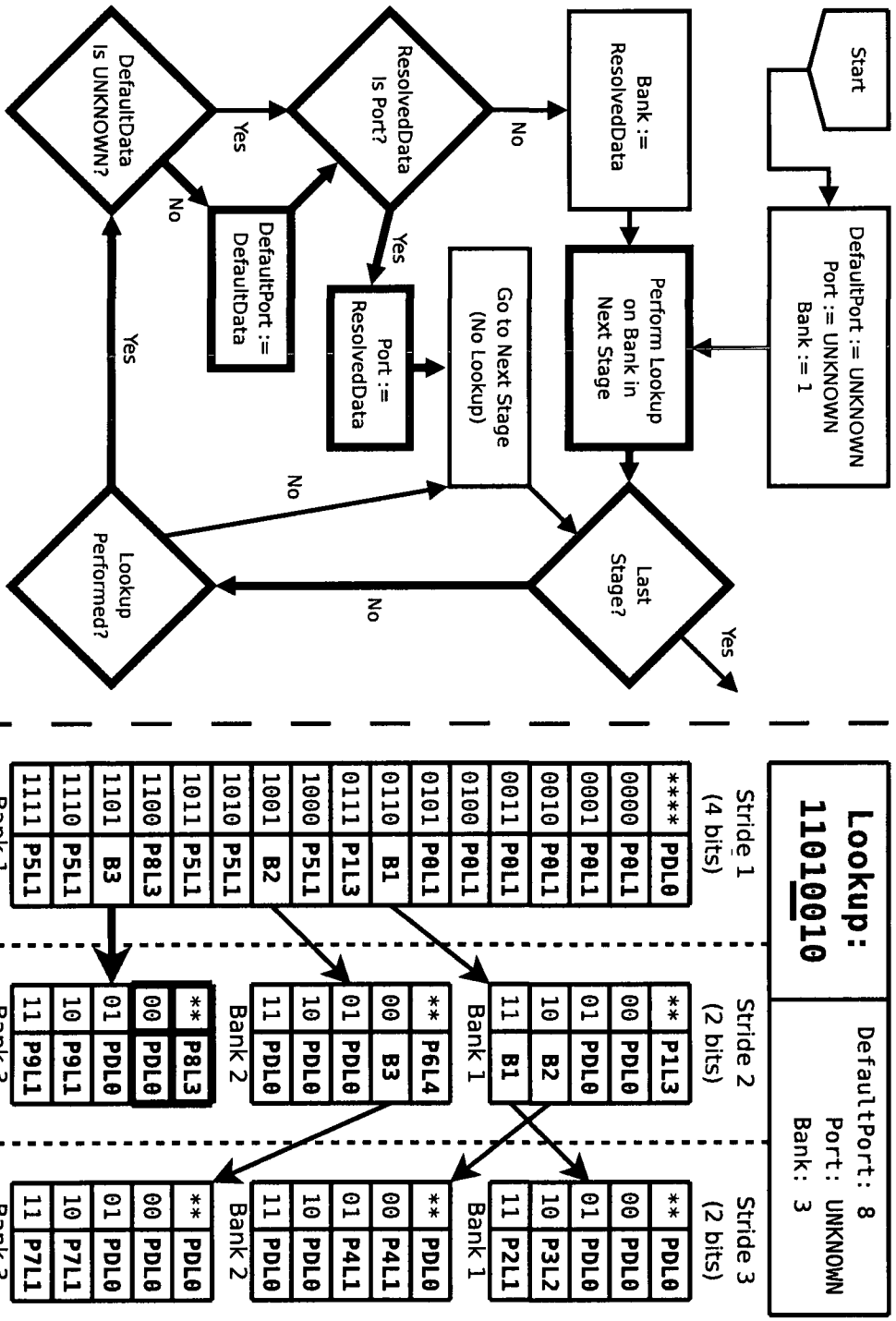


Figure B.5: Lookup Example 2: Second Stage Lookup Returns A Port Number

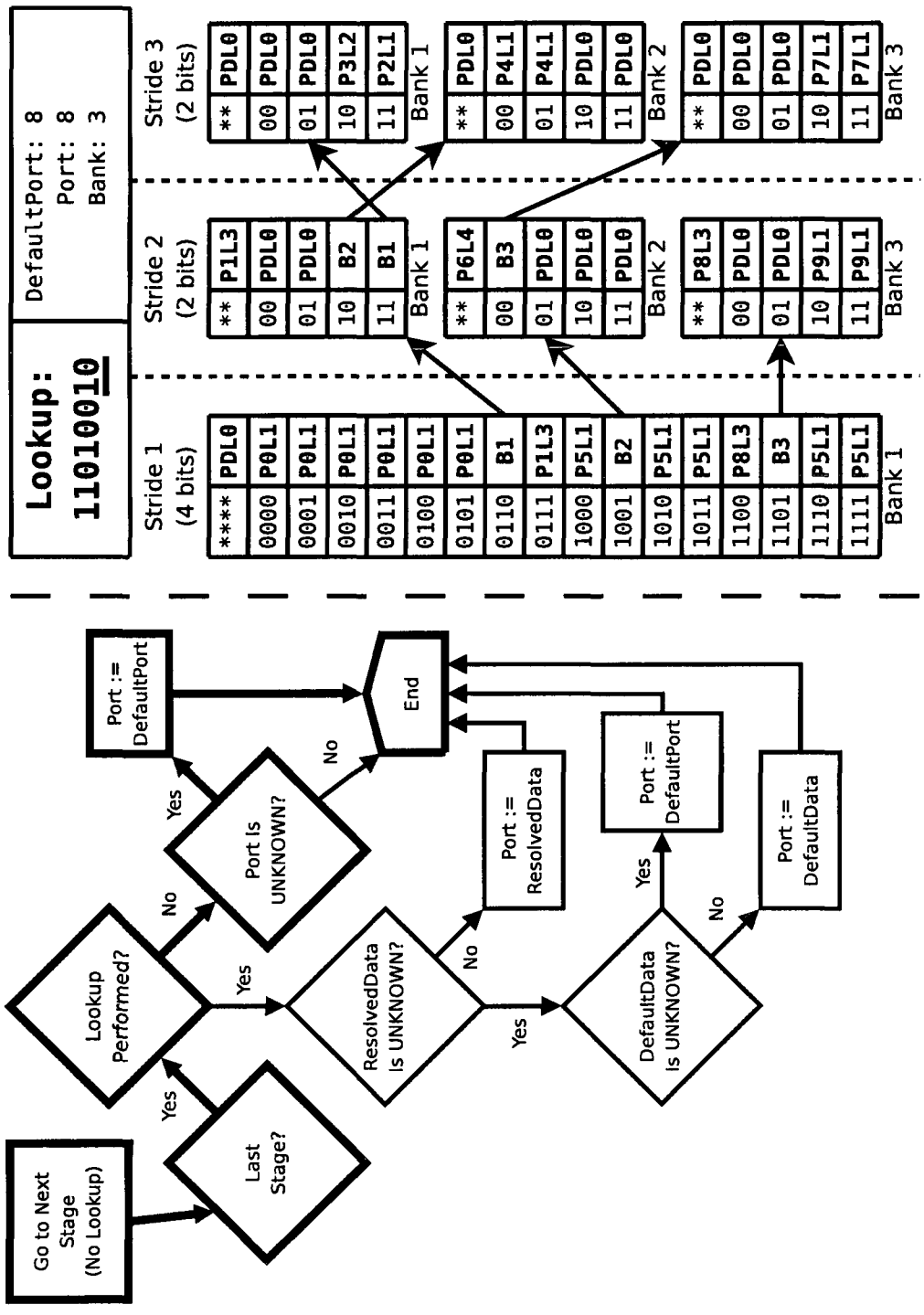


Figure B.6: Lookup Example 2: Last Stage Doesn't Perform A Lookup

B.2 Addition Process Examples

B.2.1 Addition Example 1: Prefix 01101000/7 → Port 4

In this example IP prefix 01101000/7 to port 4 is being added to the routing table. In Figure B.7 the prefix extends past the first stage (whose stride is 4) so the first four bits of the prefix, 0110, are indexed into the first bank. The result is a pointer to the first bank in the second stage which is followed. In Figure B.8, since the prefix also extends past the second stage a lookup is performed using 10 as the index to the first bank in the second stage. This yields a port number, which means a new bank must be allocated in the third stage. After the second bank in the third stage is allocated its default value is set to the port number and prefix length read from the first bank in the second stage (port *default*, length 0). In Figure B.9 the entry in the first bank of the second stage is changed to now point to the newly allocated second bank in the third stage. This pointer is then followed, arriving at the third stage, which the prefix does not extend past. In Figure B.10 and Figure B.11 the two entries in the second bank of the third stage that are covered by the prefix being added are read. Since they both contain prefix lengths of 0 they are both replaced by the prefix's entry of port 4 length 1.

B.2.2 Addition Example 2: Prefix 01100000/3 → Port 1

In this example IP prefix 01100000/3 to port 1 is being added to the routing table. In Figure B.12 the prefix does not extend past the first stage (whose stride is 4). In Figure B.13 the first entry covered by the prefix in the first bank of the first stage is read. It contains a pointer to the first bank of the second stage so that bank's default entry is analyzed. Since it contains a shorter prefix (corresponding to 00000000/1) it is replaced by the new prefix's entry of port 1 length 3. In Figure B.14 the second and last entry covered by the prefix in the first bank of the first stage is read. Since it contains a port number for a shorter prefix (corresponding to 00000000/1) it is replaced by the new prefix's entry of port 1 length 3.

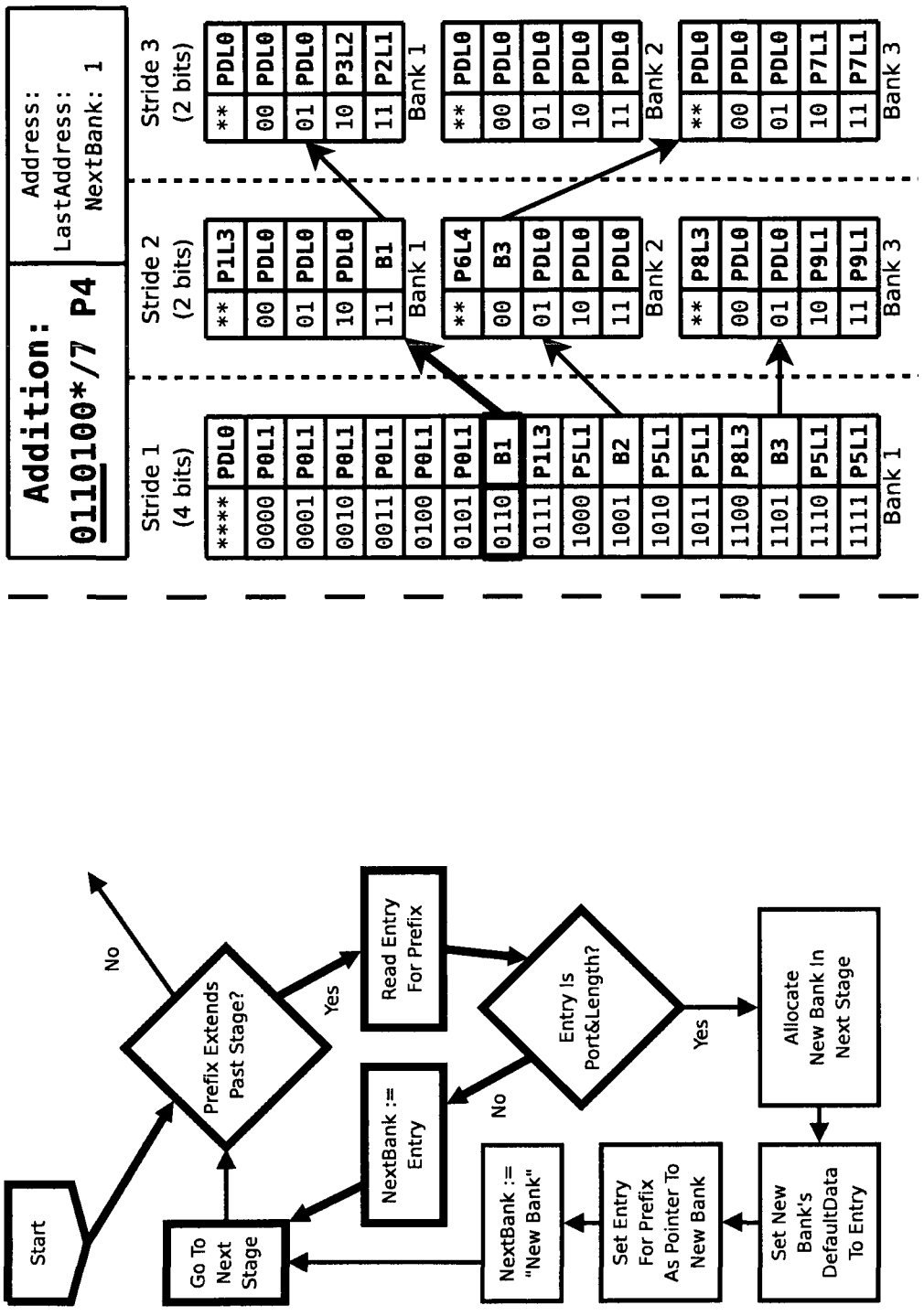


Figure B.7: Addition Example 1: First Stage Lookup Returns A Pointer

Figure B.8: Addition Example 1: Second Stage Lookup Returns A Port Number

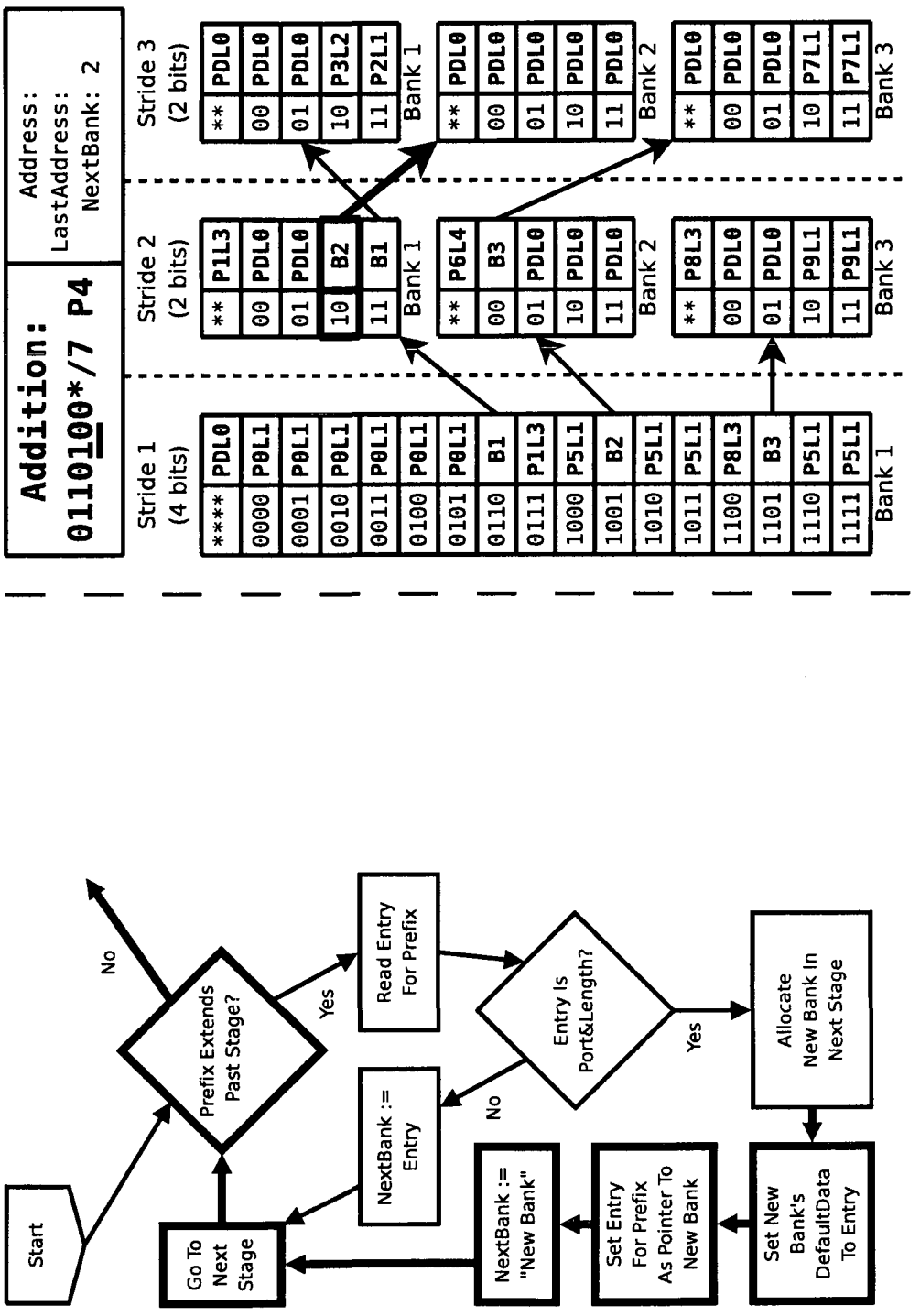


Figure B.9: Addition Example 1: Second Stage Entry Changed To Point To Newly Allocated Bank

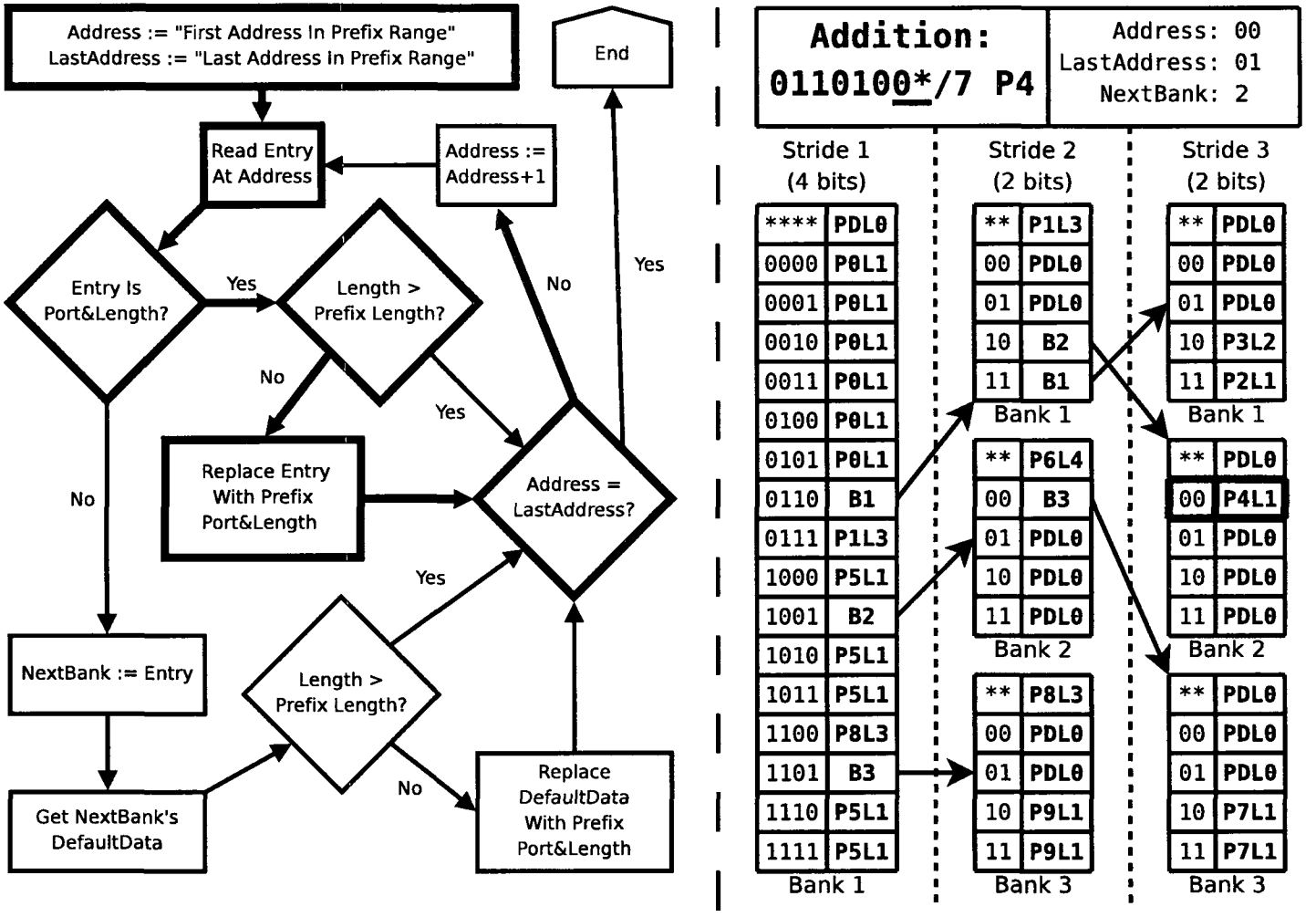


Figure B.10: Addition Example 1: Third Stage Entry Changed To New Prefix's Port Number

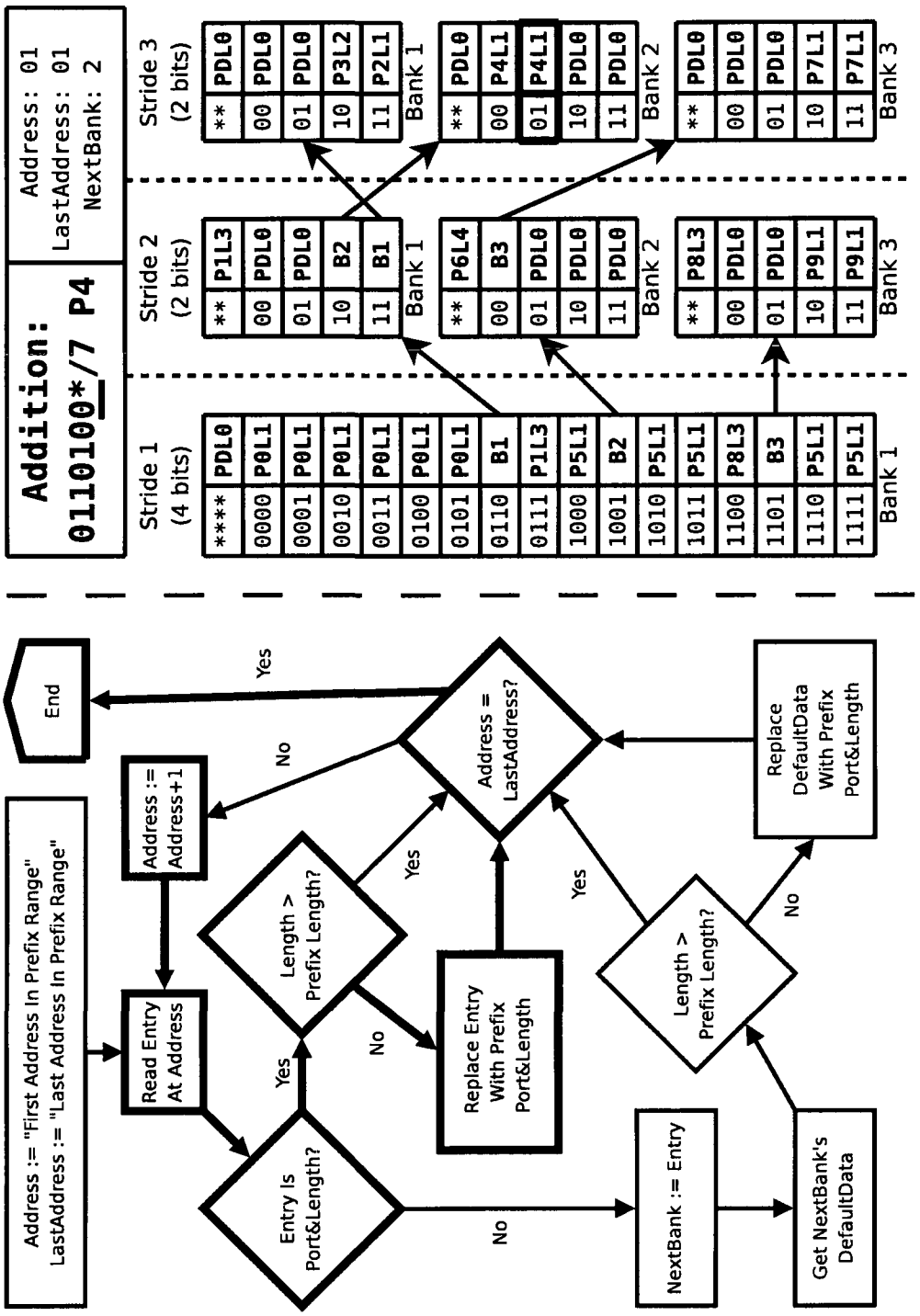


Figure B.11: Addition Example 1: Third Stage Entry Changed To New Prefix's Port Number

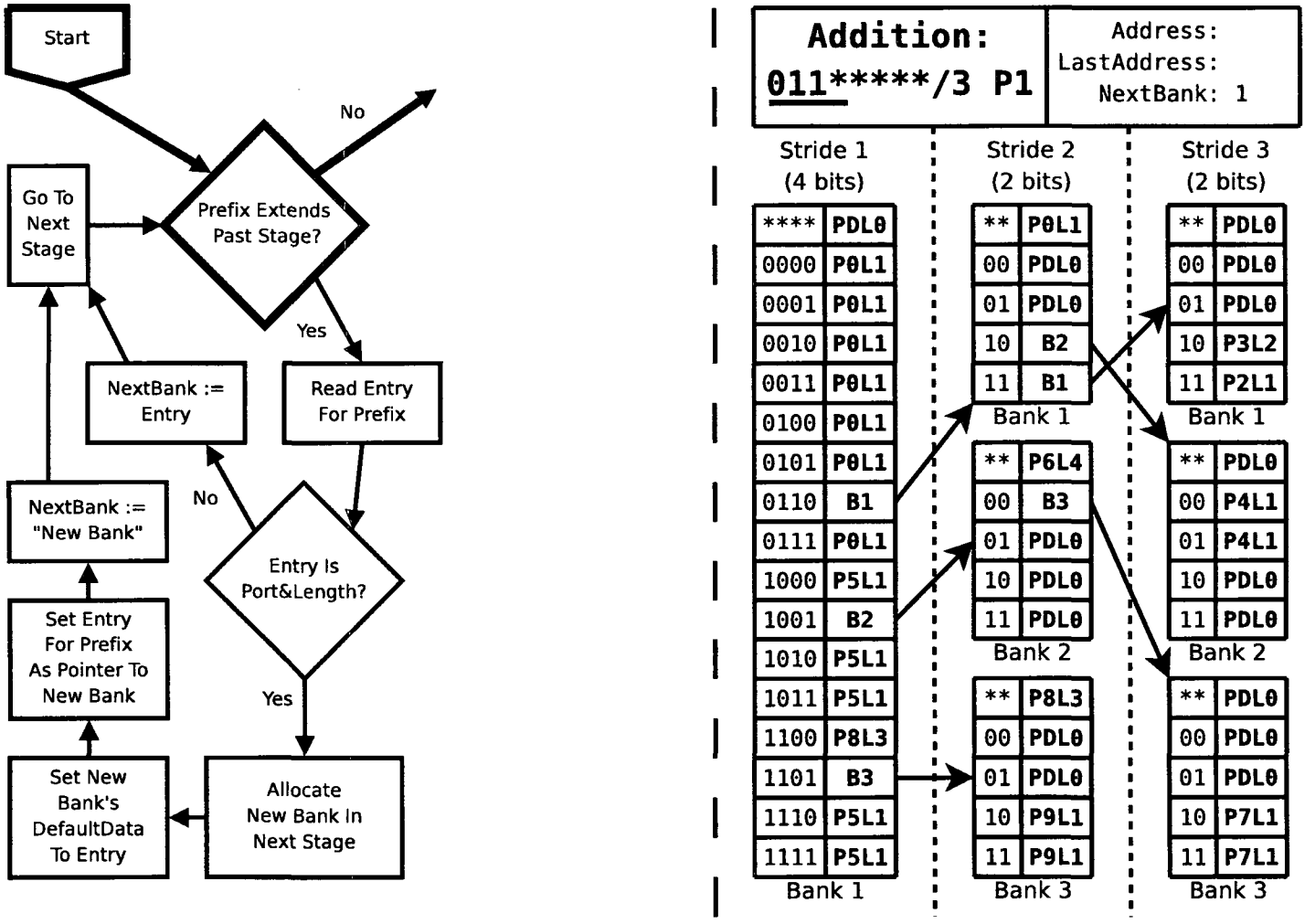


Figure B.12: Addition Example 2: Prefix Does Not Extend Past First Stage

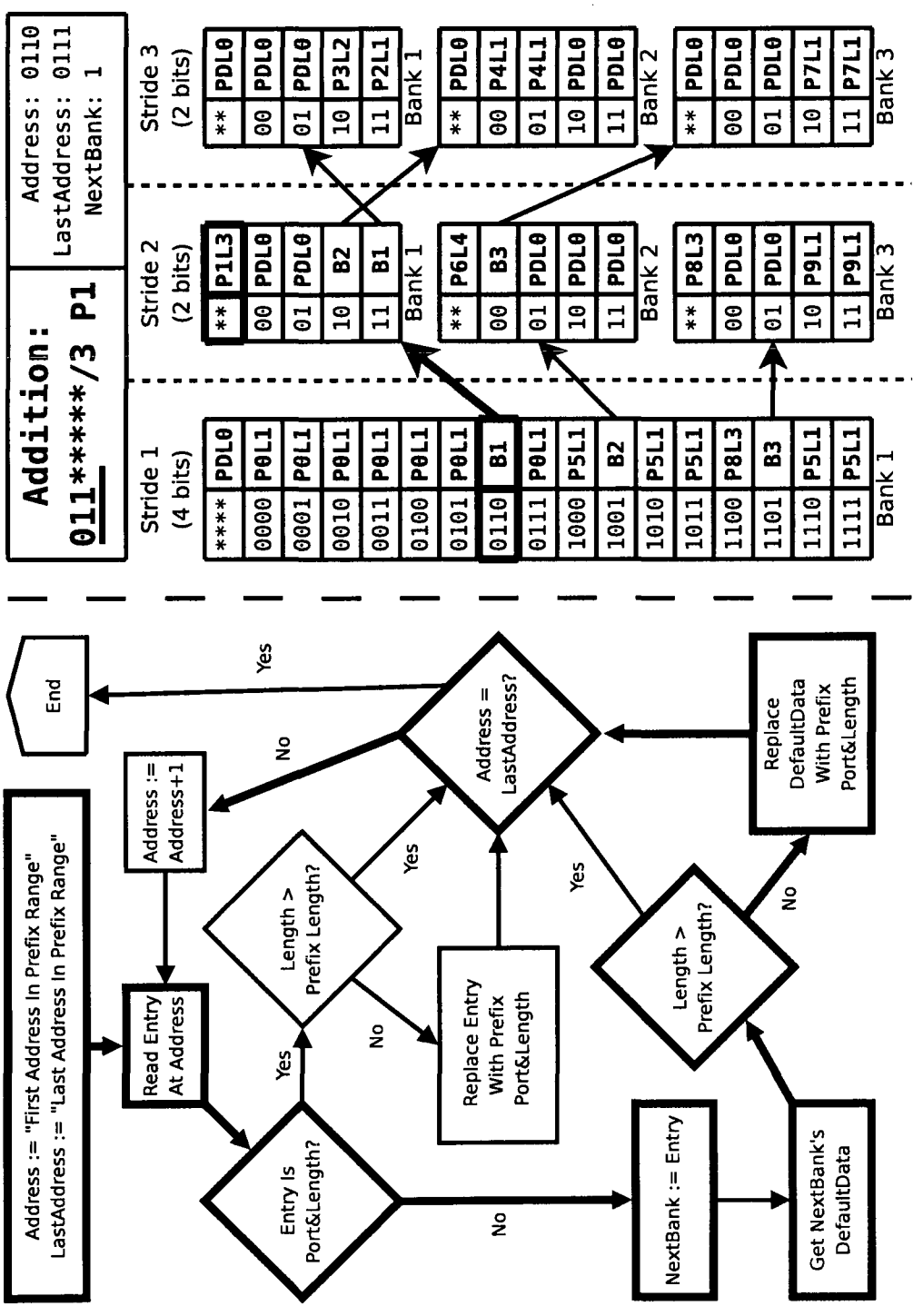


Figure B.13: Addition Example 2: First Stage Entry's Pointer Followed And Target Bank's Default Entry Changed To New Prefix's Port Number

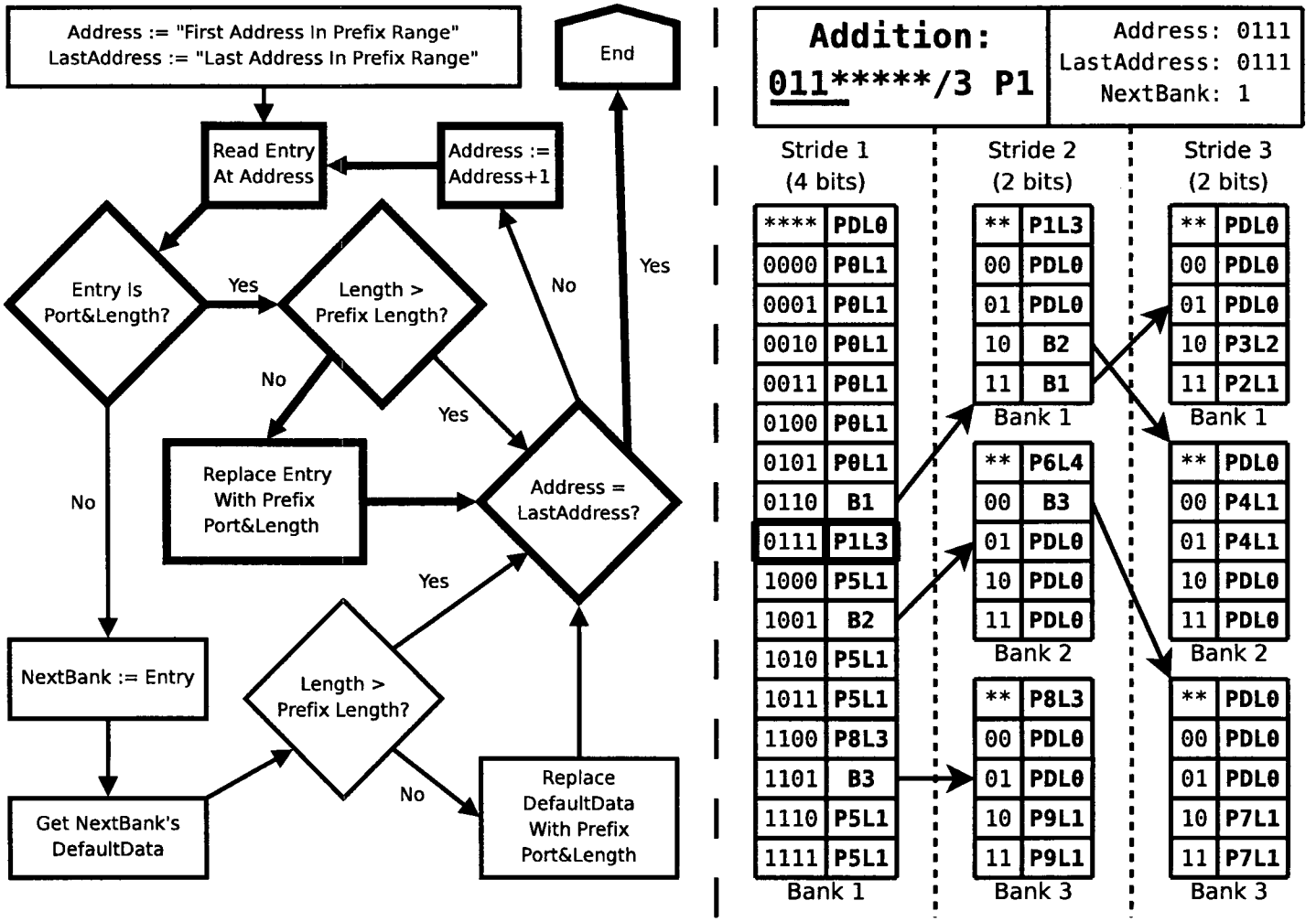


Figure B.14: Addition Example 2: First Stage Entry Changed To New Prefix's Port Number

B.3 Removal Process Examples

B.3.1 Removal Example 1: Prefix 01100000/3

In this example IP prefix 01100000/3 is being removed from the routing table. In Figure B.15 the prefix doesn't extend past the first stage, so a search for the longest prefix that encompasses the prefix to be removed is conducted. A search of entry 0100 in the first bank of the first stage yields prefix 00000000/1 → Port 0 which suffices, so it will therefore be used as a replacement for the removed prefix's entries. In Figure B.20 the first entry covered by the prefix in the first bank of the first stage is read. It contains a pointer to the first bank of the second stage so that bank's default entry is analyzed. Since it contains a match to the prefix to be removed it is replaced by port 0 length 1. In Figure B.21 the second and last entry covered by the prefix in the first bank of the first stage is read. Since it contains a match to the prefix to be removed it is replaced by port 0 length 1. In Figure B.22 since it is the first stage no bank deallocation is possible.

B.3.2 Removal Example 2: Prefix 01101000/7

In this example IP prefix 01101000/7 is being removed from the routing table. In Figure B.19 the prefix extends past the first stage (whose stride is 4) so the first four bits of the prefix, 0110, are indexed into the first bank. The result is a pointer to the first bank in the second stage which is followed. In Figure B.20, since the prefix also extends past the second stage a lookup is performed using 10 as the index to the first bank in the second stage. This yields a pointer, which is also followed. In Figure B.21, since the prefix does not extend past the current stage a search for the longest prefix that encompasses the prefix to be removed is conducted. Since such a prefix does not exist within the scope of this stage, the default entry will be used as a replacement for the removed prefix's entries. In Figure B.22 and Figure B.23 the two entries in the second bank of the third stage that are covered by the prefix being removed are read. Since they are both entries for the prefix to be removed they are both replaced by the default entry of port *default* length 0. In Figure B.24 since it is a third stage bank deallocation may be possible. In Figure B.25, Figure B.26 and Figure B.27 each entry in the second bank of the third stage is read and verified to match the default entry, which means the bank can be safely deallocated. In Figure B.28 the second bank of the third stage is deallocated and the second stage is revisited. Since it is a second stage bank deallocation may be possible. In Figure B.29, Figure B.30 and Figure B.31 the first three entries in the first bank of the second stage are read and verified to match the default entry. In Figure B.32 the last entry in the first bank of the second stage is read and it does not match the default entry, which means the bank cannot be deallocated.

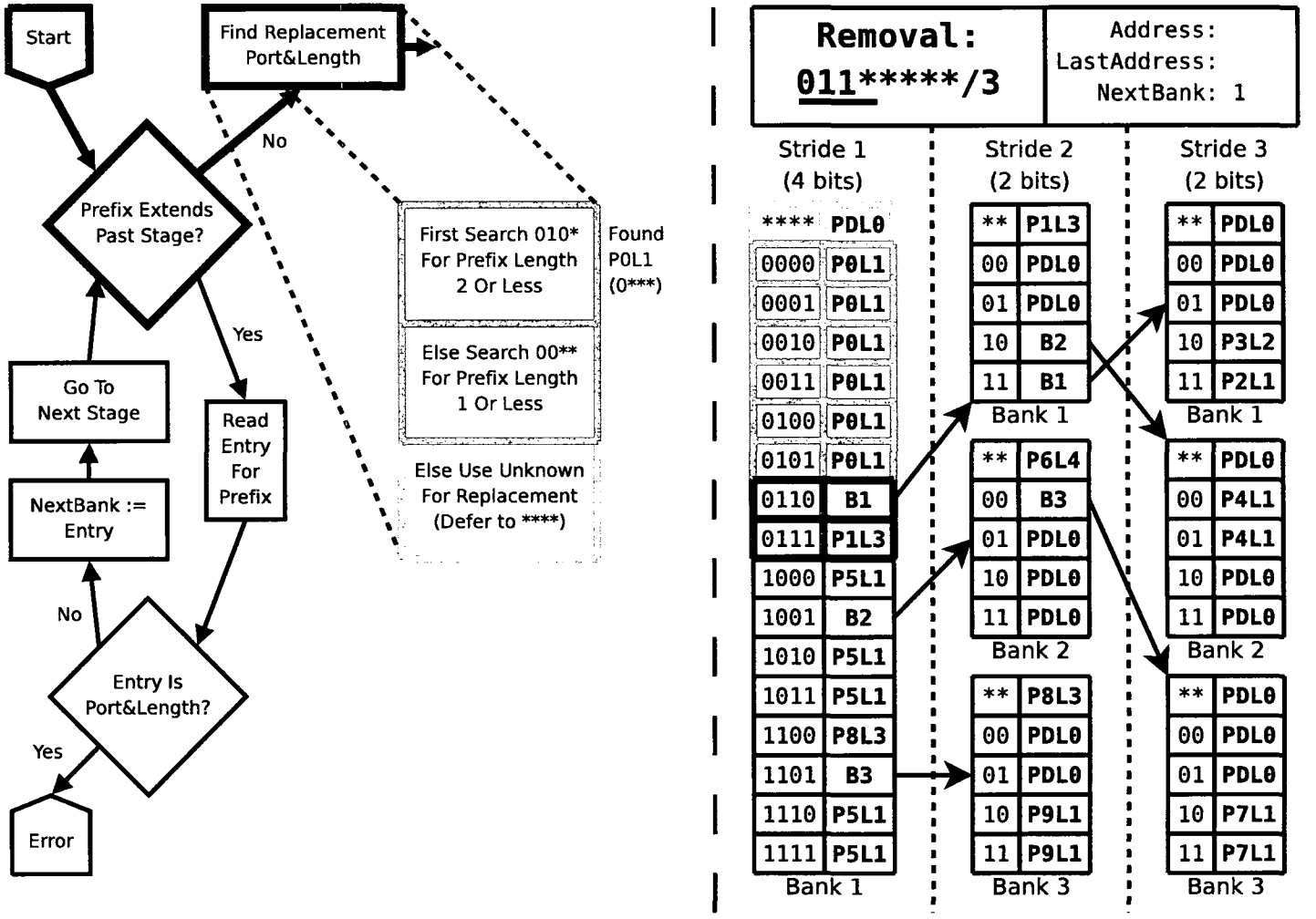


Figure B.15: Removal Example 1: Search Of First Stage For A Replacement Prefix

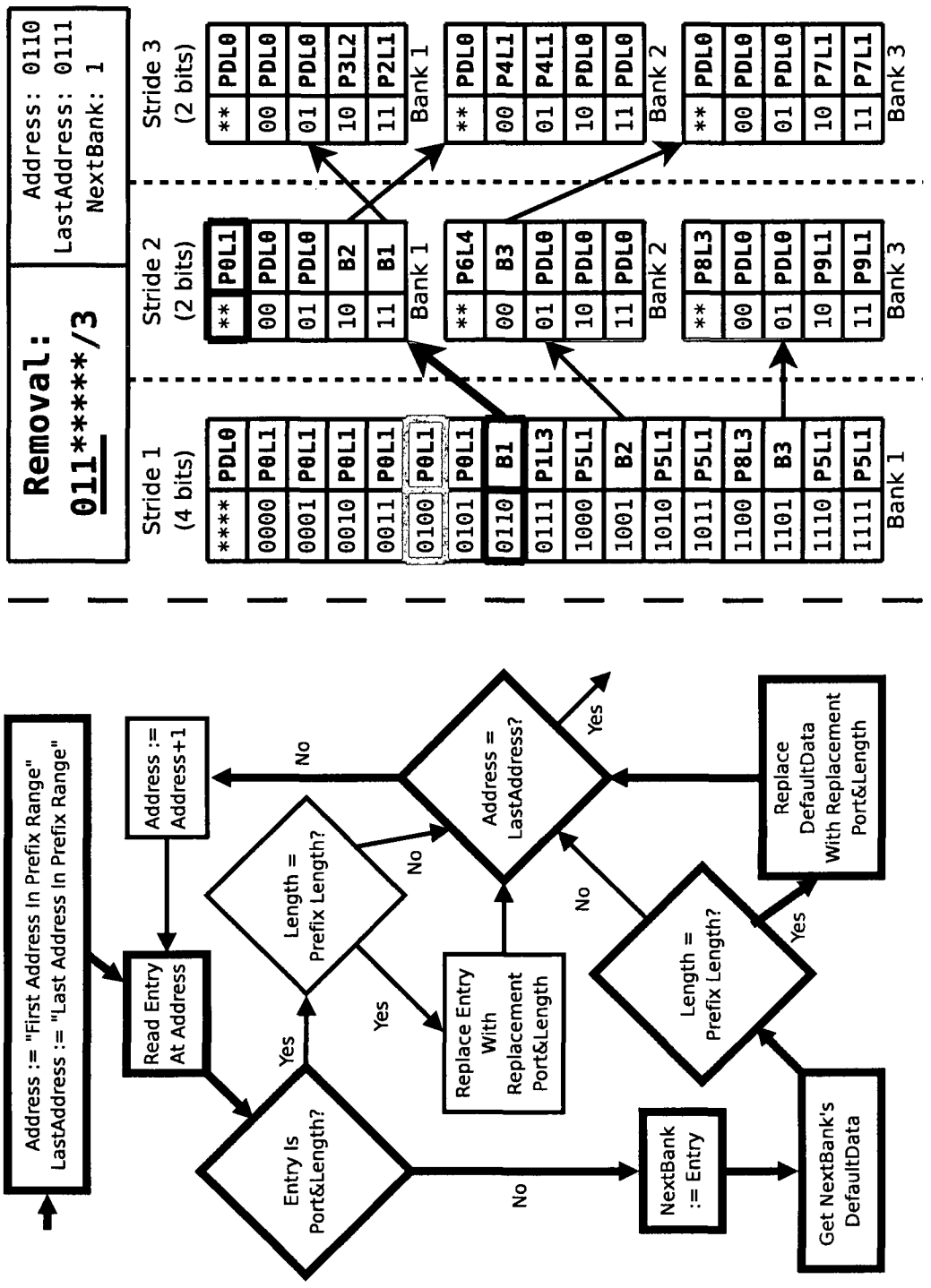


Figure B.16: Removal Example 1: First Stage Entry's Pointer Followed And Target Bank's Default Entry Removed

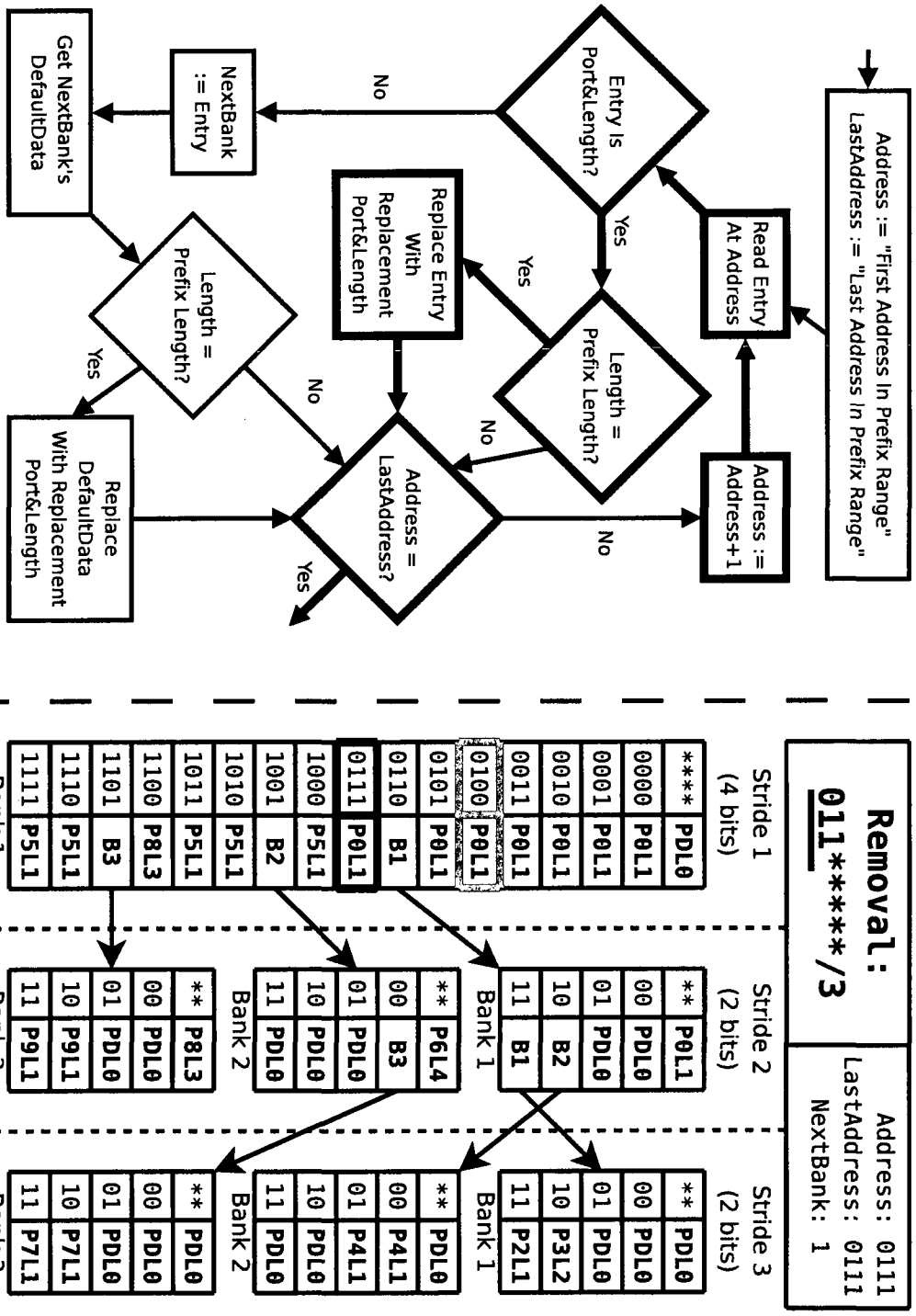


Figure B.17: Removal Example 1: First Stage Entry Removed

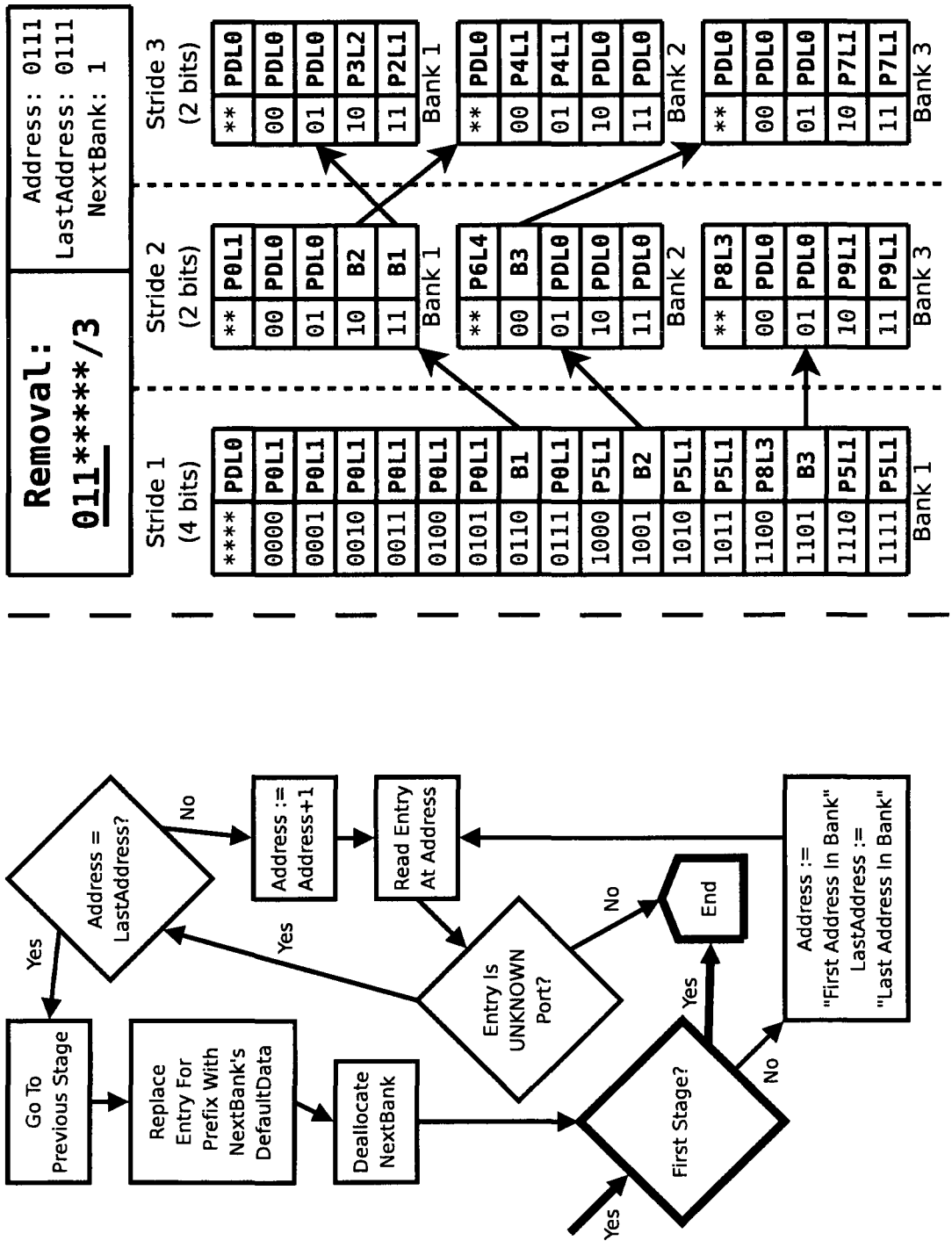


Figure B.18: Removal Example 1: No Bank Deallocation Possible

Figure B.19: Removal Example 2: First Stage Lookup Returns A Pointer

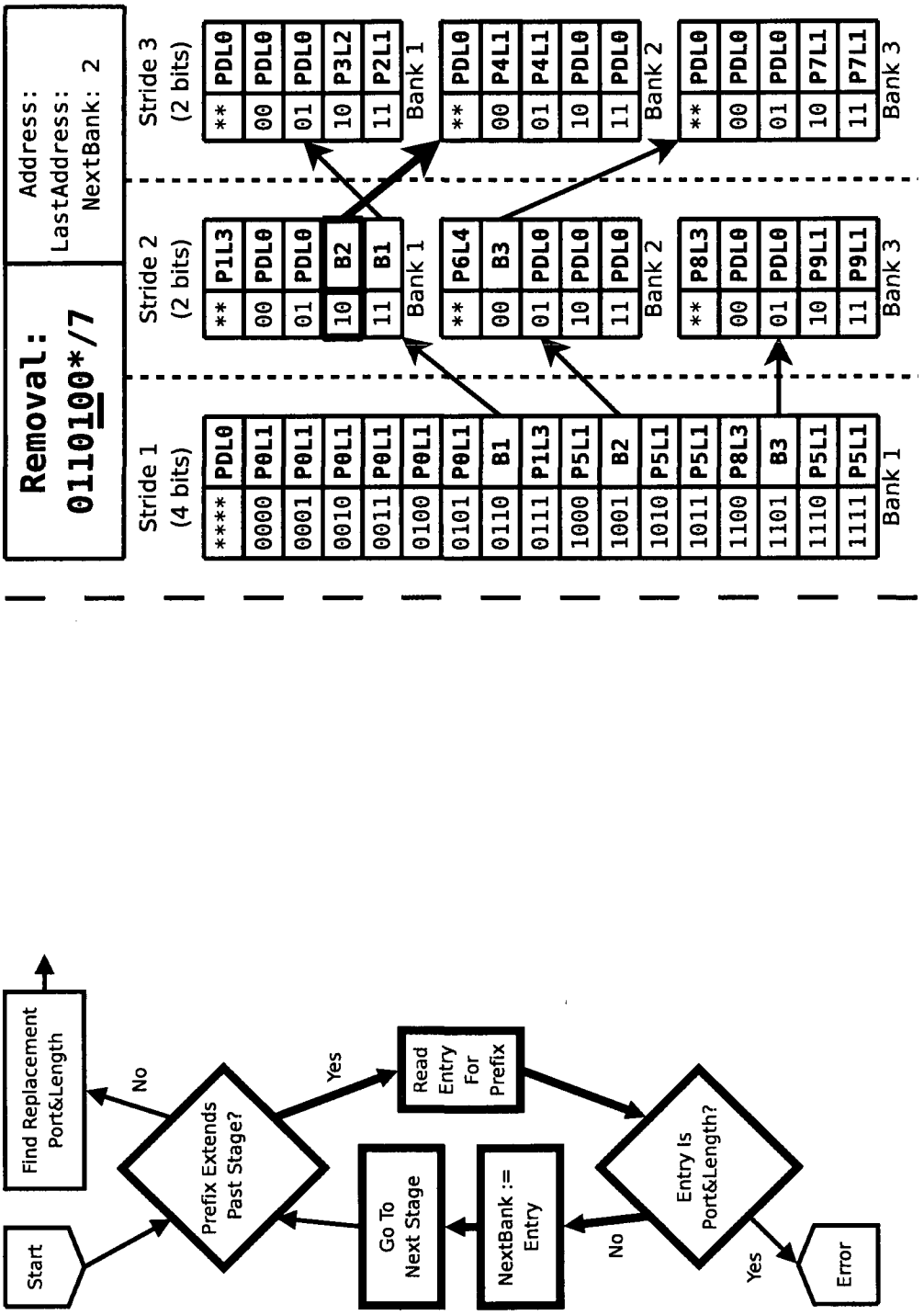


Figure B.20: Removal Example 2: Second Stage Lookup Returns A Pointer

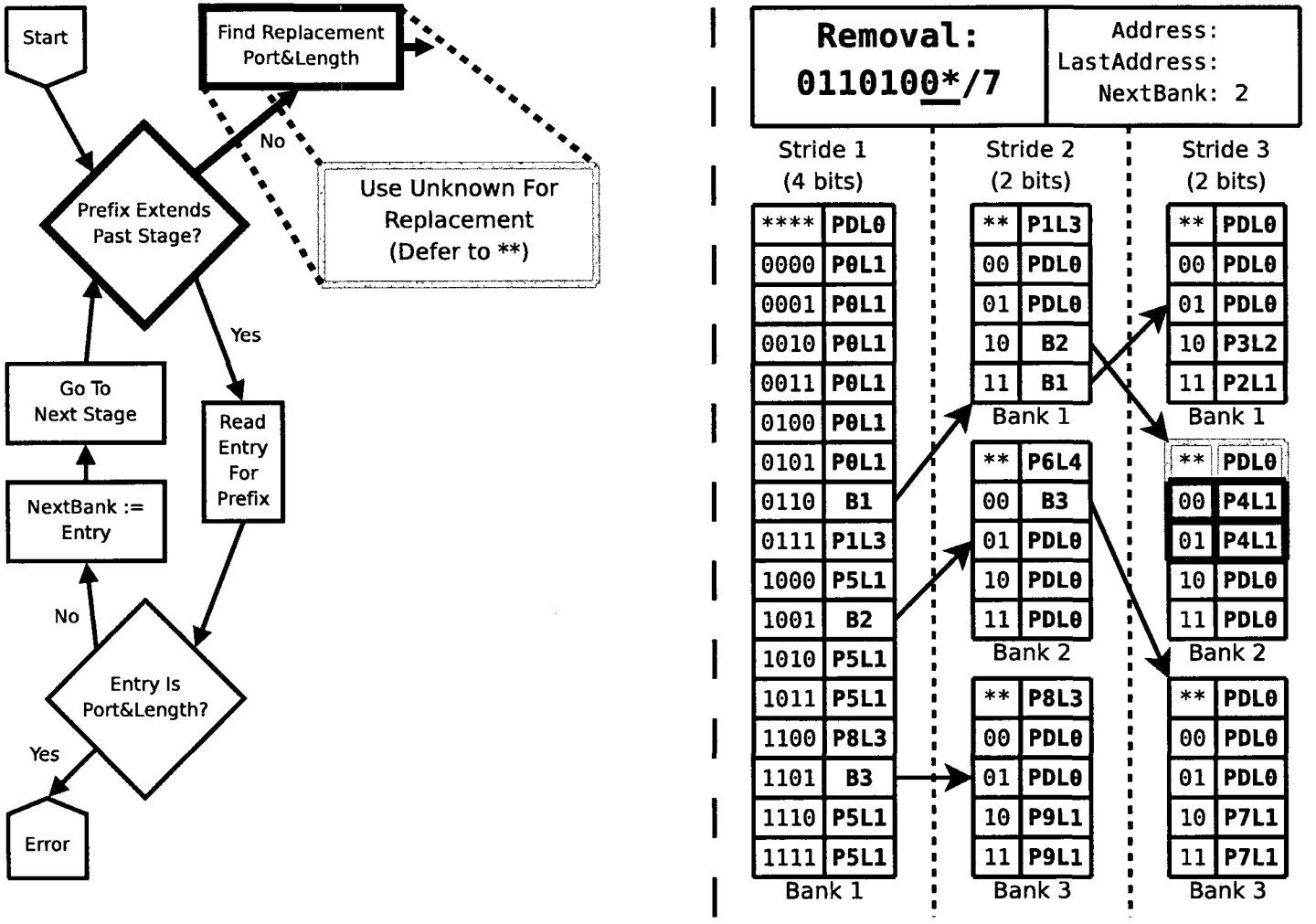


Figure B.21: Removal Example 2: Prefix Does Not Extend Past Third Stage

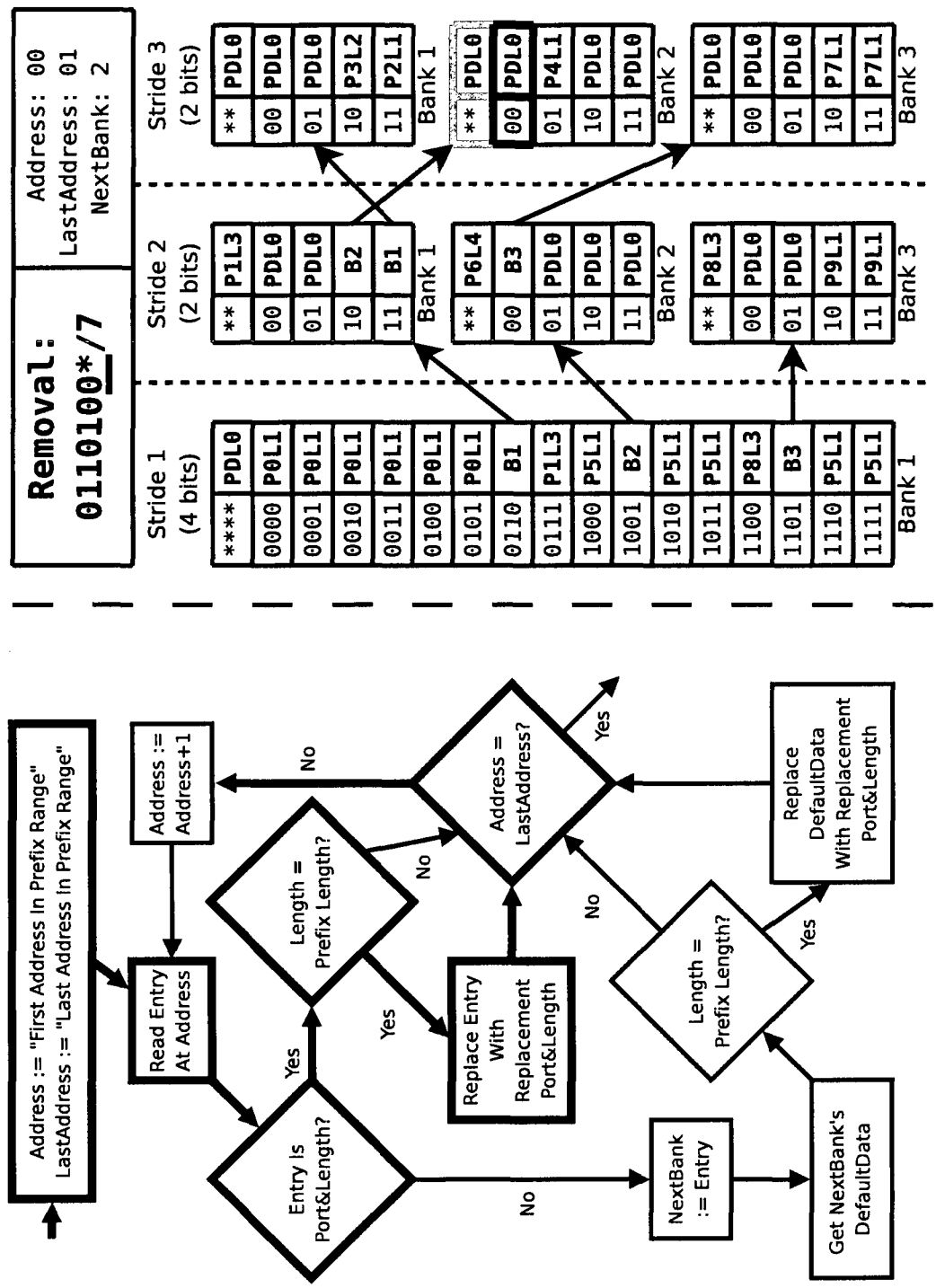


Figure B.22: Removal Example 2: Third Stage Entry Removed

Figure B.23: Removal Example 2: Third Stage Entry Removed

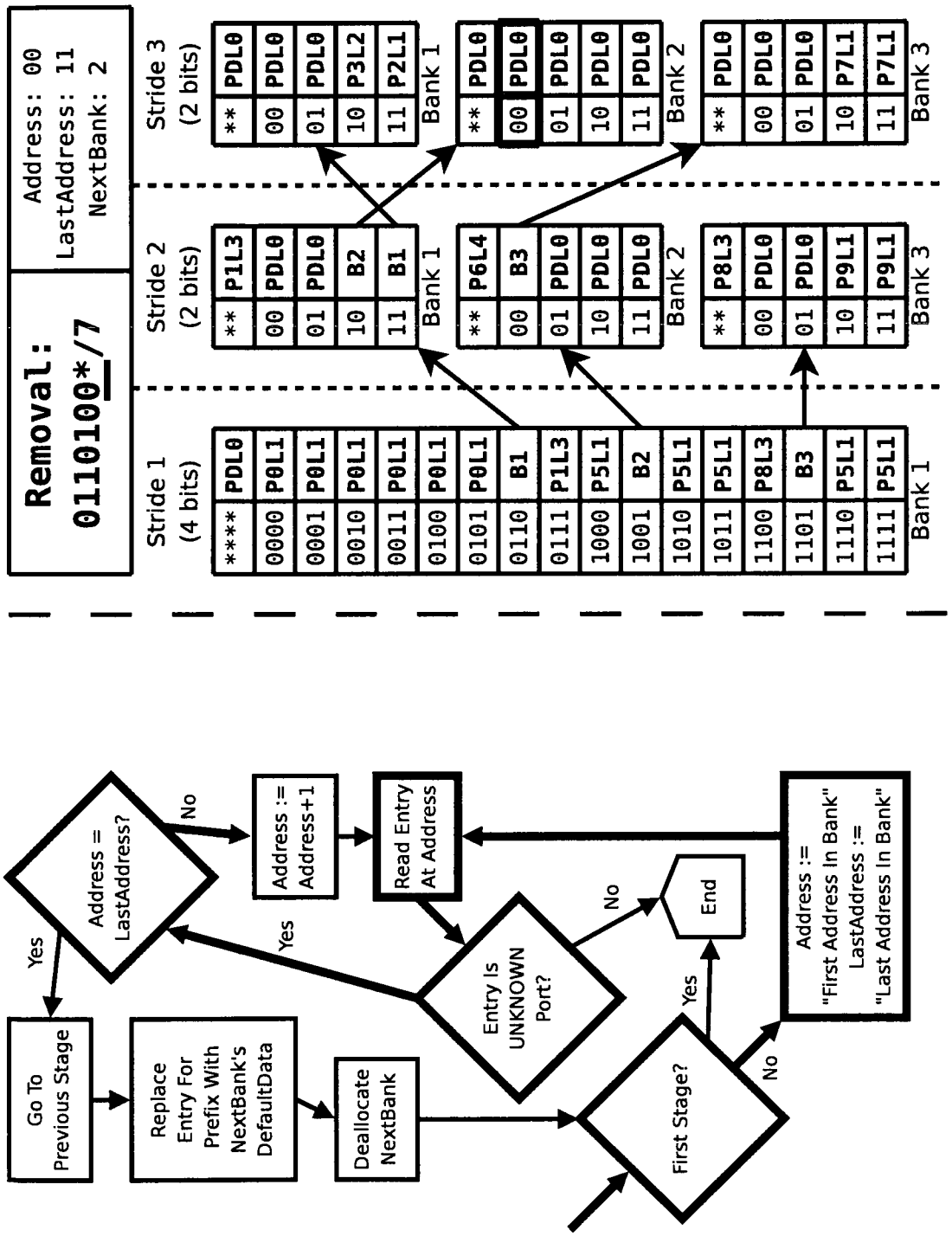


Figure B.24: Removal Example 2: Third Stage Bank Deallocation May Be Possible

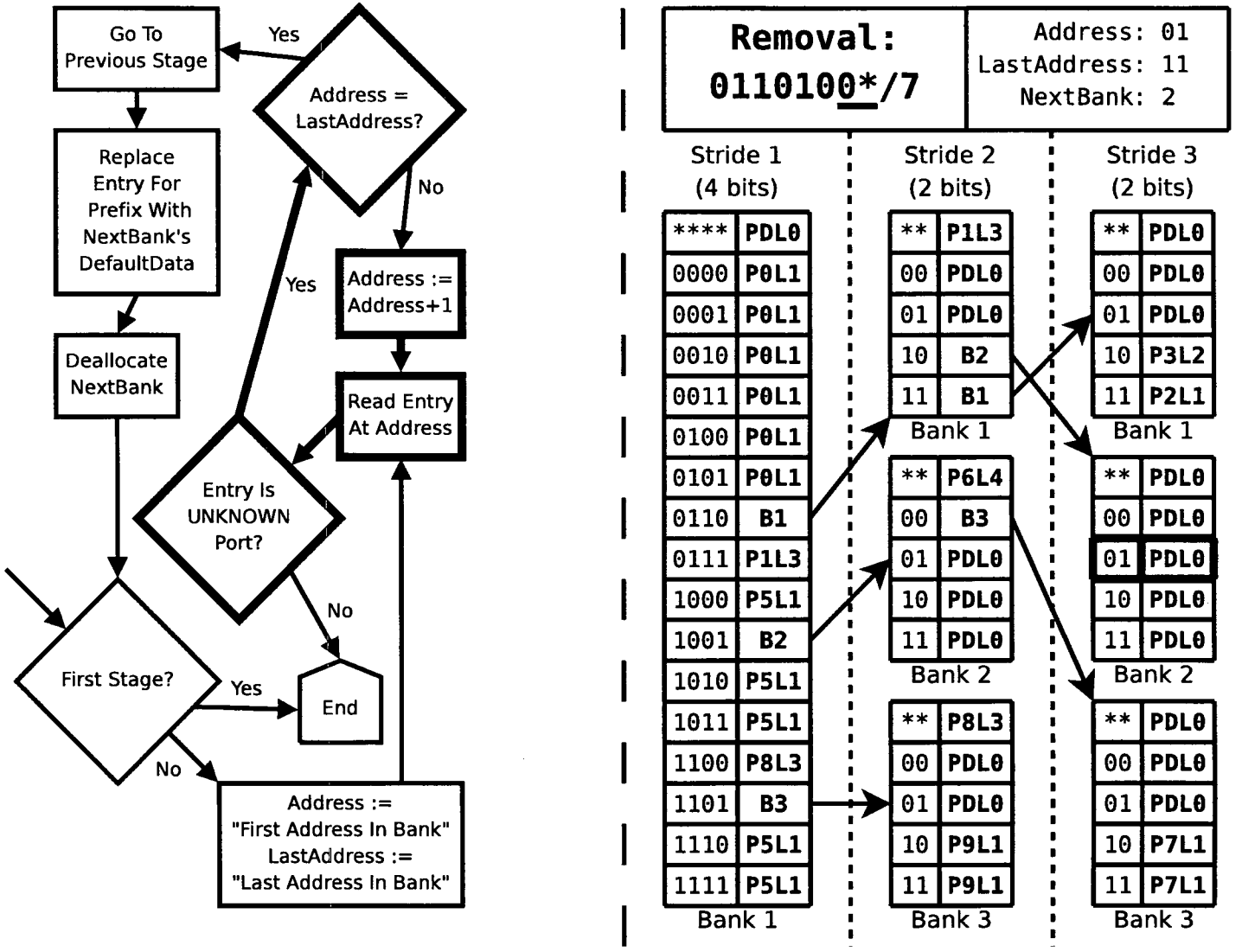


Figure B.25: Removal Example 2: Third Stage Bank Entry Matches Default Entry

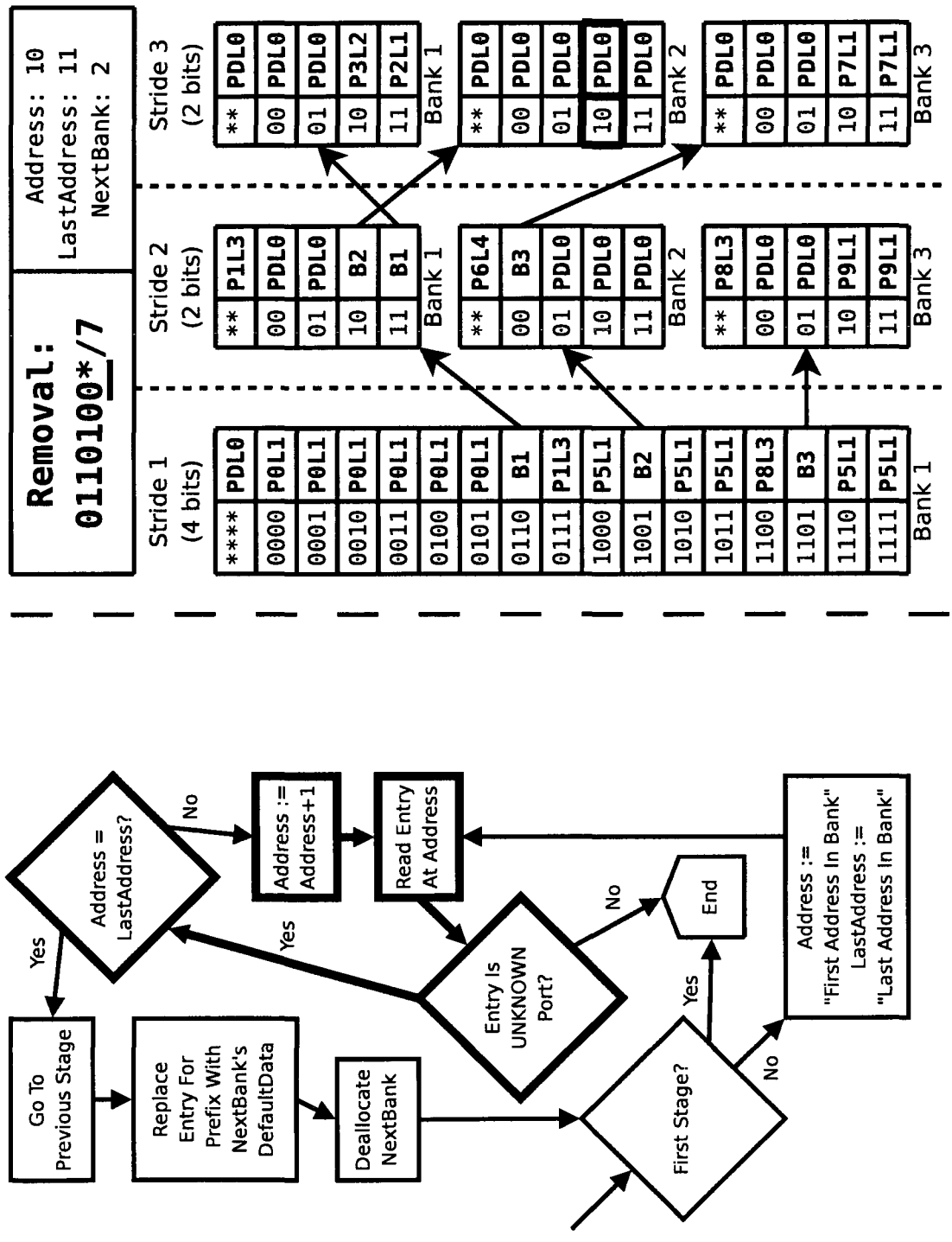


Figure B.26: Removal Example 2: Third Stage Bank Entry Matches Default Entry

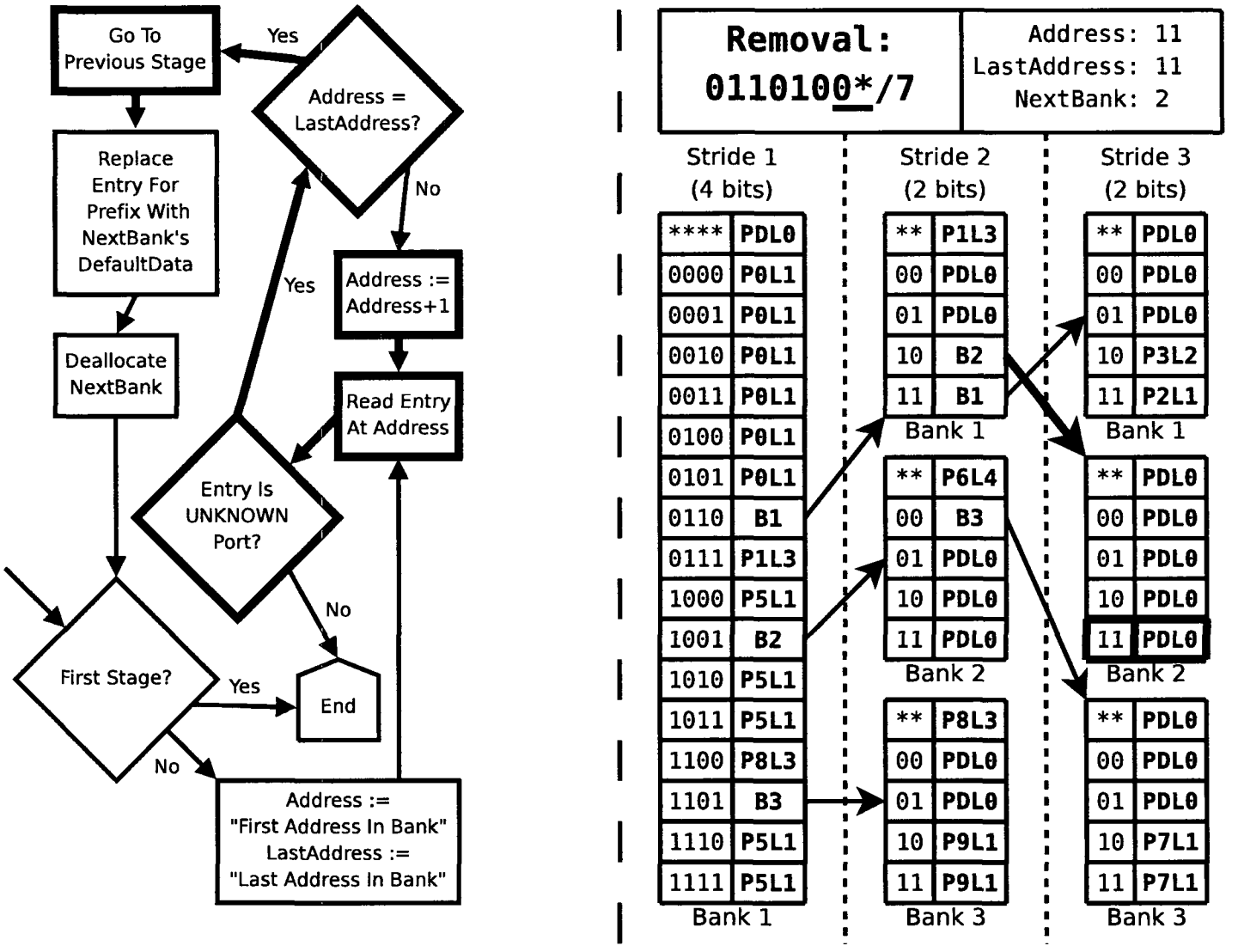


Figure B.27: Removal Example 2: Third Stage Bank Entry Matches Default Entry

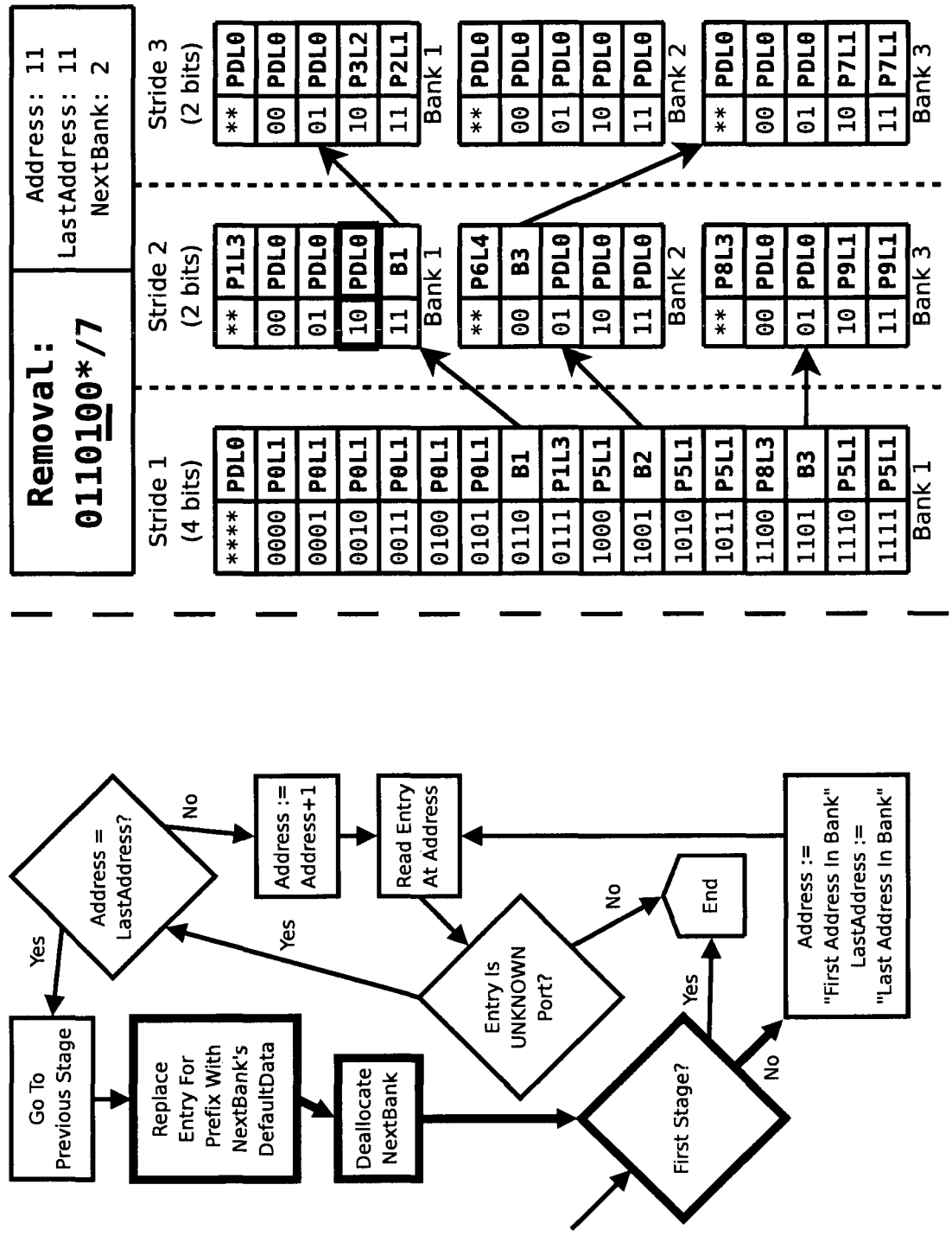


Figure B.28: Removal Example 2: Third Stage Bank Deallocated

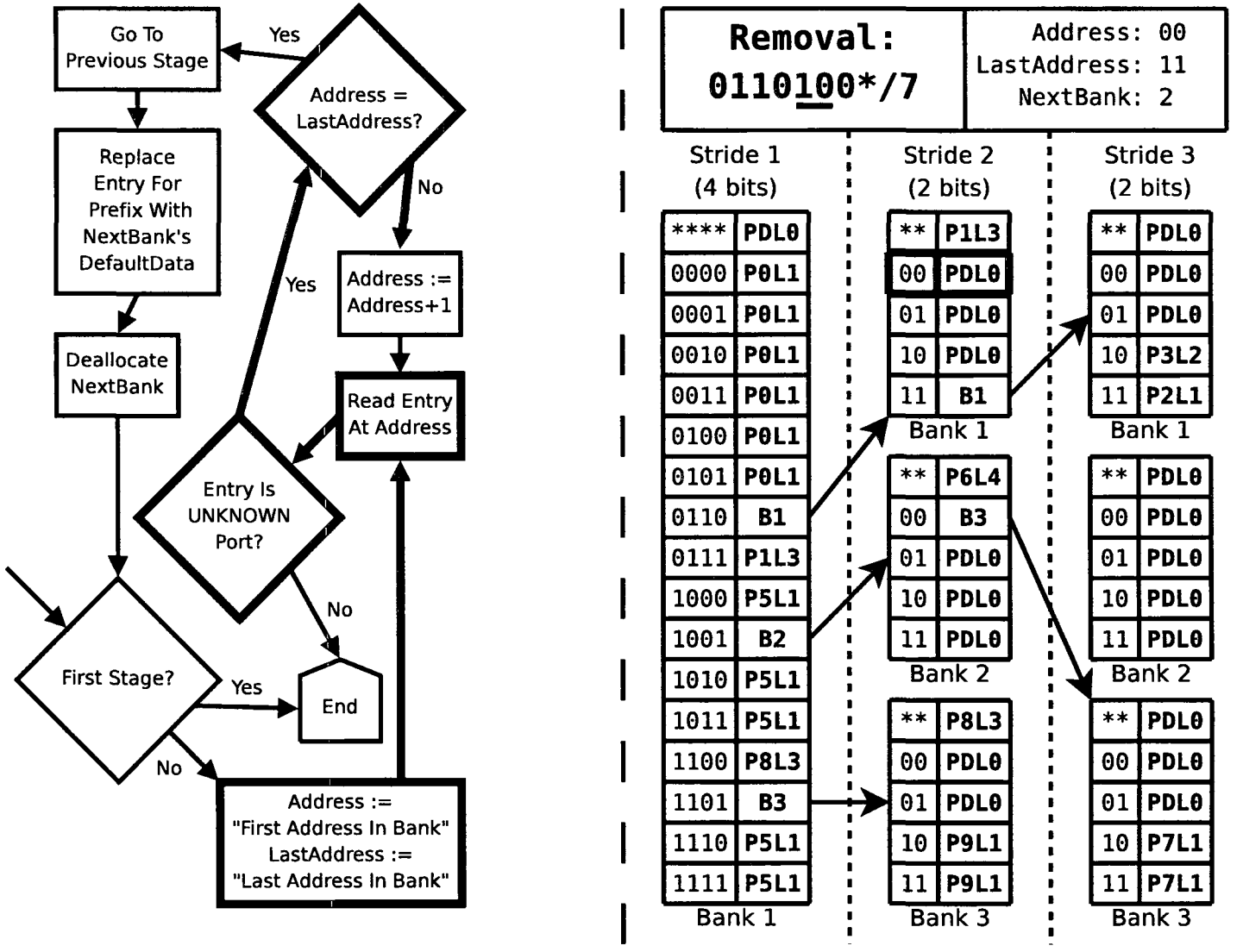


Figure B.29: Removal Example 2: Second Stage Bank Entry Matches Default Entry

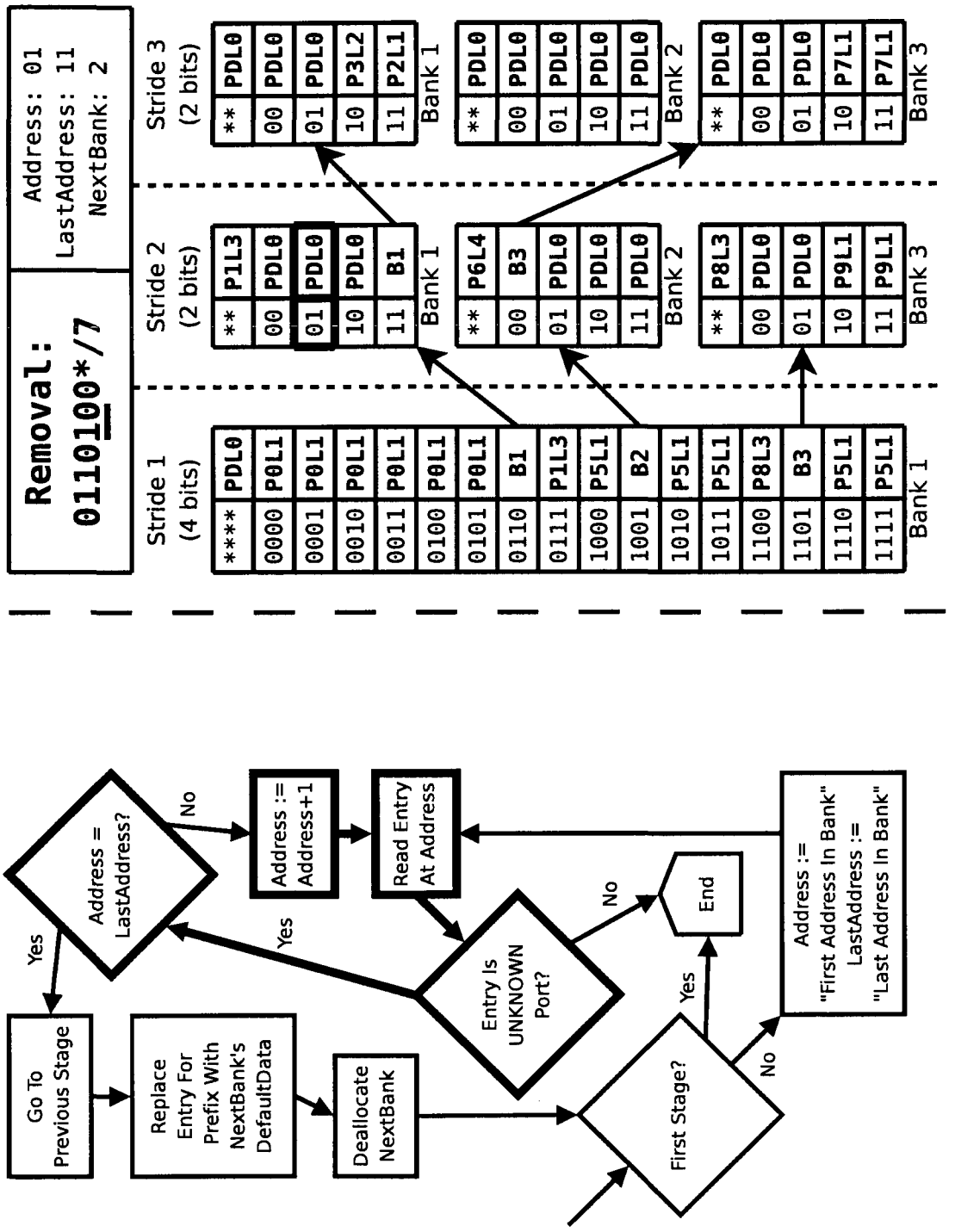


Figure B.30: Removal Example 2: Second Stage Bank Entry Matches Default Entry

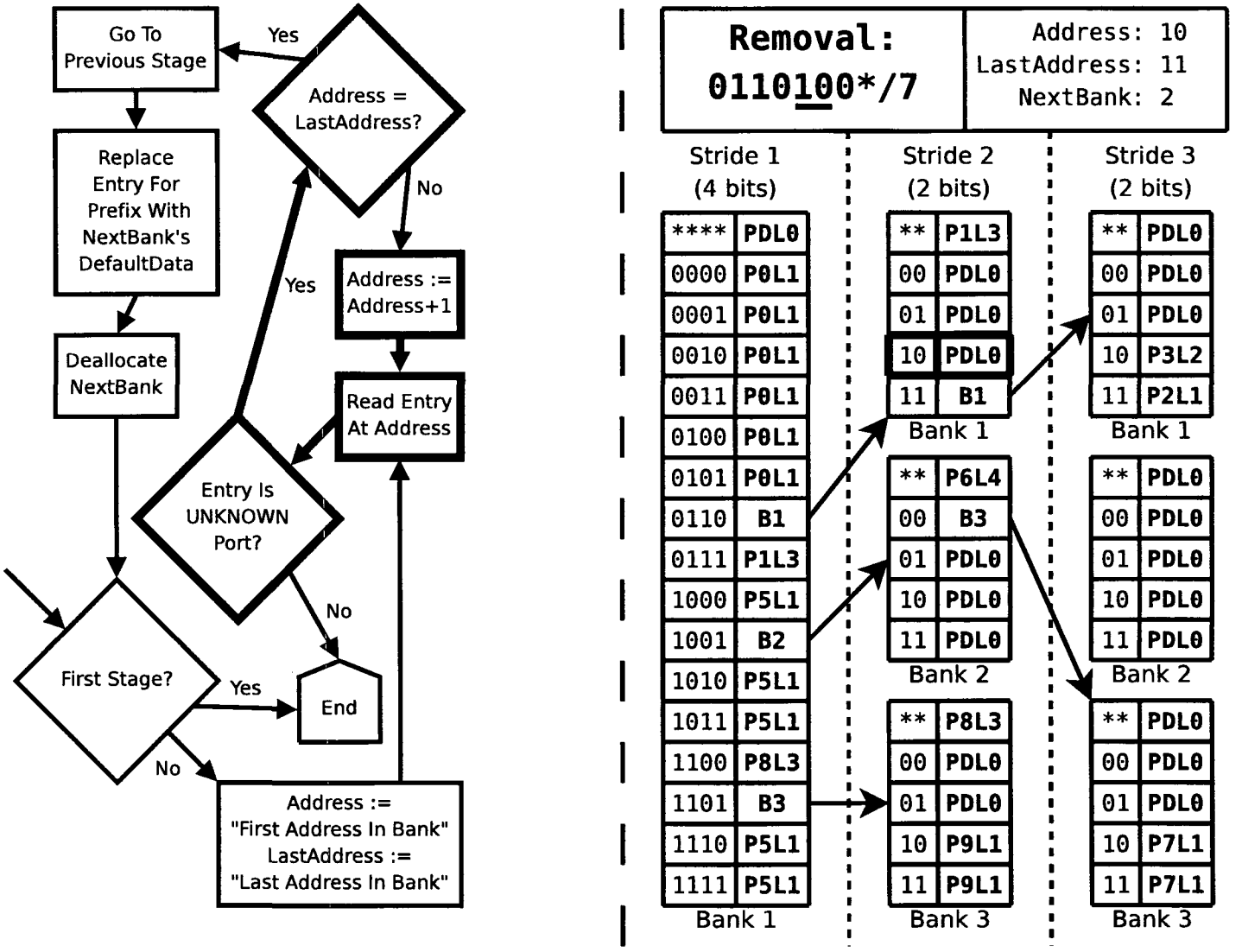


Figure B.31: Removal Example 2: Second Stage Bank Entry Matches Default Entry

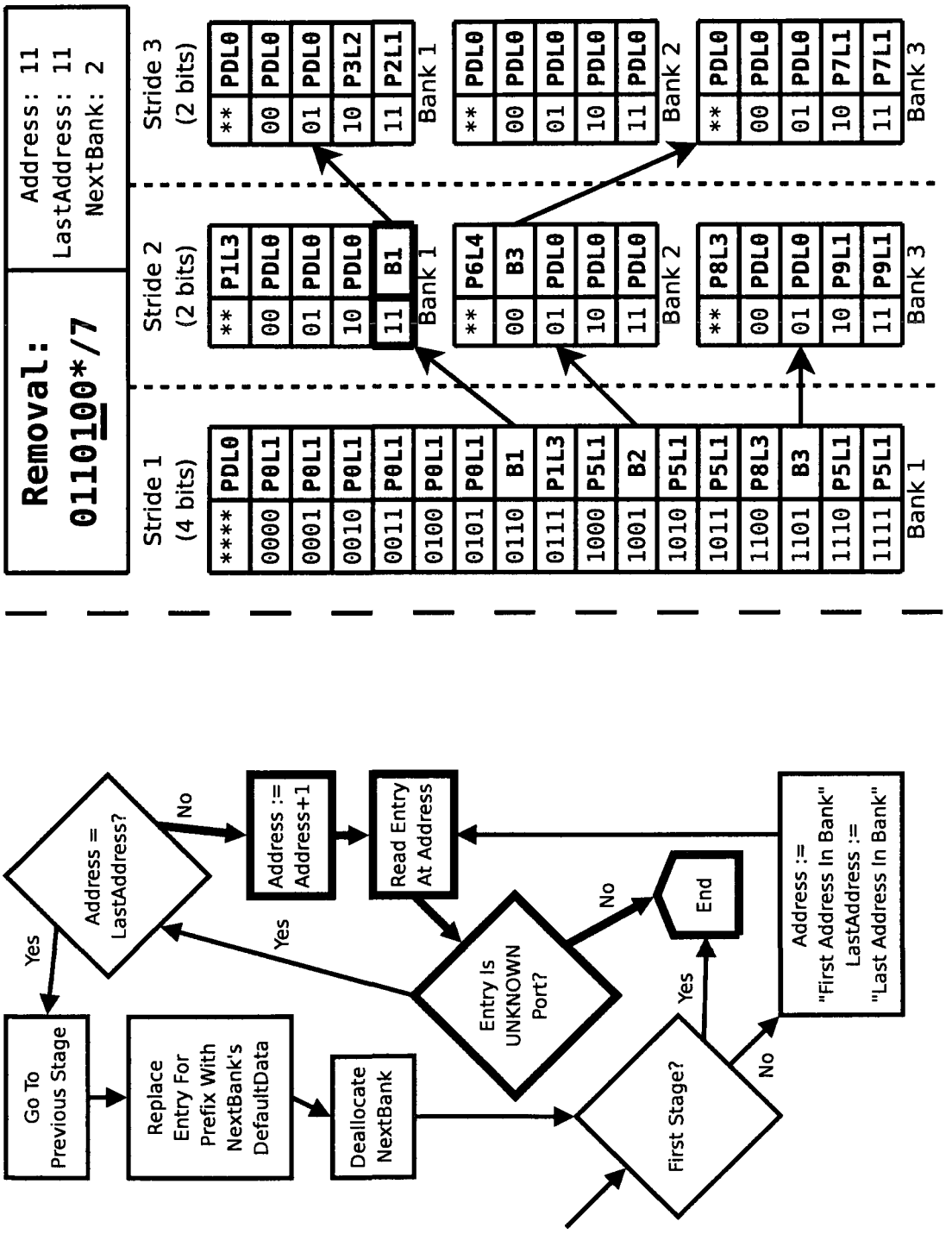


Figure B.32: Removal Example 2: Second Stage Bank Entry Mismatches Default Entry