A FAST IP LOOKUP MODULE FOR A ROUTER CHIP

PROJECT REPORT

Submitted by

RAVISHANKAR RAO **SRIHARI NARASIMHAN**

Under the Guidance of

Mr. Ramesh Kini M.
Assistant Professor
Department of E&C Engg.,
KREC, Surathkal – 574 157

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

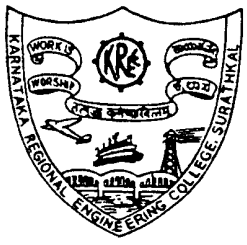
BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

OF THE

MANGALORE UNIVERSITY



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
KARNATAKA REGIONAL ENGINEERING COLLEGE, SURATHKAL,
SRINIVASNAGAR – 574 157 KARNATAKA INDIA

April 2002

Abstract

The huge increase in Internet traffic has pushed conventional architectures based on a general CPU, mostly using a single microprocessor, to the limit. High performance routers need to use specialized architectures that can implement some or all the functions of a router in hardware. The basic functionality of an IP router involves route processing, packet forwarding, and router special services. The main bottleneck for router performance has been the IP lookup part of packet forwarding. The router performs a table lookup to determine the output port onto which to direct the packet and the next hop to which to send the packet along this route. This is based on the destination IP address in the received packet and the sub-net mask(s) of the associated table entries. Traditional implementations of routing tables use a version of Patricia tries. Due to the large size of this data structure, routing tables would not fit into on-chip caches and off-chip references onto DRAMs are too slow to support gigabit routing speeds. A novel approach called the Lulea scheme promises much faster lookups as it can represent large routing tables in a very compact form and can be searched quickly using a few memory references. We have designed a Fast IP Lookup Module (FILM) that implements the above algorithm in hardware. The FILM has a worst-case lookup time of 8 clock cycles and an average of 5 clock cycles, and meets the requirement for wire-speed routing.

TABLE OF CONTENTS

List of Figures, Tables and Abbreviations Used

Figures

Tables

Abbreviations

1.	ASIC	Application Specific Integrated Circuit
2.	BGP	Border Gateway Protocol
3.	BIM	Base Index Memory
4.	CAM	Content Addressable Memory
5.	CSR	Campus Switch Router
6.	CWM	Code Word Memory
7.	DRAM	Dynamic Random Access Memory
8.	FILM	Fast IP Lookup Module
9.	Ipv4	Internet Protocol Version 4
10.	SRAM	Static Random Access Memory

1. Introduction

1.1 Problem Statement

The communication links in the Internet are being upgraded with high-speed high-bandwidth optic links. Unless Internet routers are capable of forwarding packets at gigabit rates, these links will be insufficient. The major bottleneck in high-speed routing is IP Address Lookup. Routers store addresses as prefixes rather than an entire address to avoid scaling problems, so that packets must be forwarded to the *longest matching prefix*. Finding the longest matching prefix in routing tables is a challenging algorithmic problem. To obtain a bandwidth of 10 Gbps with an average packet length of 2000 bits, a router should forward about 5 million packets per second. So, each packet must be forwarded in about 200 ns, and each lookup must be performed even faster. It is desirable to implement such critical functions of a router like packet processing and packet forwarding in hardware and do the rest in software. Hence, the requirement is of a router chip that employs a 'Fast IP Address Lookup' module, can service at least 12 ports, and is capable of forwarding 10 million packets per second (mpps).

1.2 Current Status

1.2.1 Conventional Routers

Conventional routers make use of software running on a computer system, based on an operating system like Windows / Linux / MacOS. Such routers can be set up at a relatively low cost, but they not only need a computer running to allow access to all the other computers, but are also slow. These routers were sufficient till somtime ago since the volume of data they were required to handle was small and that too, at low speeds. But today, the volumes of data flowing over the network is enormous and applications like video-conferencing and streaming video are the order of the day, which demand extremely high data rates. The conventional routers are undoubtedly incapable of handling such speeds owing to their sequential nature of operation and the operating system overheads.

Conventional IP address matching implementations are based on the PATRICIA algorithm [1], which is essentially a compressed binary tree where only bit positions in the input key that discriminate between paths are considered when traversing the tree. An example is the Radix Tree algorithm in the NetBSD kernel. As described in [2], support for longest matching prefixes is provided by means of a recursive backtracking mechanism, which worsen the search time upper bound to N^2, where N is the maximum length of the stored prefixes. PATRICIA (as well as any other search scheme based on binary prefix trees) requires O(N) memory references, where N is the maximum prefix length. With current memory and microprocessor technologies, the lookup time for both software and hardware implementations fall in the range of the microseconds.

1.2.2 Hardware Routers

When the transmission channels have become really high bandwidth compliant, the routers cannot be left behind in the wilderness. As a matter of fact, routers are essentially to a networking system since they bear the responsibility of channeling the data towards

the destination. With Optical media coming into the forefront of data communication, the slower data rates have been traced back to the "joints" i.e., the routers. Hardware routers aim at catering to this problem. Completely hardware based, they have done away with a "bulky" operating system and other overheads associated with the conventional routers. As a result, they are much faster and more efficient than the conventional routers.

Schemes exist [3] that can do lookups in 200 ns using SRAMs (with 10 ns cycle times) to store the entire routing database. But, large SRAMs are extremely expensive and are typically limited to caches in ordinary processors. Hardware solutions [4] based on content-addressable memories (CAMs) can be used to implement longest matching prefix. Again, they are very expensive when large databases have to be supported.

1.2.3 Commercial Products

Cisco's Campus Switch Router (CSR) [8] uses hardware-based routing tables on each port. Cisco`s highly optimized lookup algorithm, the FIB, is downloaded to the ASICs on each line card. This optimized algorithm, together with the CEF architecture, ensures that the route processor is free to keep the network topology consistent and to converge quickly in the event of topology changes or link failures.

We present below some of the features of the high end Cisco 10720 Internet Router.
24-port 10/100BaseTX access module

- LEDs: Cardfail, power, error (R), link/active (G), 100 Mbps
- Connectors: RJ-45

*Software***:** Cisco IOS Software 12.0(19) SP or later

The ***Cisco 10720*** Internet Router supports all unicast and multicast IP routing protocols, advanced buffering and scheduling mechanisms, and the capability to store up to 250,000 routes. It also supports a forwarding rate of approximately 2 mpps with IP services enabled.

The ***Juniper M-160*** Router architecture is presented below:

The two key components of the M160 architecture are the Packet Forwarding Engine (PFE) and the Routing Engine, which are connected via a 100-Mbps link. Control traffic passing through the 100-Mbps link is prioritized and rate limited to help protect against denial-of-service attacks.

- The PFE is responsible for packet forwarding performance. It consists of the Flexible PIC Concentrators (FPCs), PICs, SFMs, and state-of-the-art ASICs.
- The Routing Engine maintains the routing tables and controls the routing protocols. It consists of an Intel-based PCI platform running JUNOS software.

Internet Processor II ASICs
Each of the four Internet Processor II ASICs (one per SFM) supports a lookup rate of over 40 Mpps. With over one million gates, the Internet Processor II ASIC delivers high-speed forwarding performance with advanced IP services, such as filtering and sampling,

enabled. It is the largest, fastest, and most advanced ASIC ever implemented on a router platform and deployed in the Internet.

Packet Forwarding Engine
The PFE provides Layer 2 and Layer 3 packet switching, route lookups, and packet forwarding. The Internet Processor II ASIC forwards up to an aggregate 160 Mpps for all packet sizes. The aggregate throughput is 160+ Gbps.

SFM
- One Internet Processor II ASIC for 160-Mpps aggregate packet lookup (40 Mpps per SFM)
- 204.8-Gbps aggregate throughput (102.4-Gbps full duplex)
- Two Distributed Buffer Manager ASICs for coordinating pooled, single-stage buffering
- 8-MB of parity-protected SSRAM
- Processor subsystem (One PowerPC 603e processor, 256-KB of parity-protected Level 2 cache, and 64-MB of parity-protected DRAM)

Routing Engine
- 333-MHz mobile Pentium II with integrated 256-KB Level 2 cache
- SDRAM (three 168-pin DIMMs containing 768-MB ECC SDRAM)
- 80-MB compact flash drive for primary storage
- 6.4-GB IDE hard drive for secondary storage
- 110-MB PC card for tertiary storage
- 10/100 Base-T auto-sensing RJ-45 Ethernet port for out-of-band management
- Two RS-232 (DB9 connector) asynchronous serial ports for console and remote management
- Optional redundancy

1.3 Motivation for the Project

1.3.1 Need For Wire-speed Routing

Hardware routers are necessary to handle the large amount of data packets to be routed over the Internet. Hardware implementation also facilitates wire-speed routing. But why do we need wire-speed routing. The Networking arena is becoming increasingly complex owing to expanding nature of protocols. Increased complexity of traffic engineering namely QOS, Congestion, Security and Address Management also has contributed to the complexity. So, it is quite apparent that more parameters need to be incorporated in the data packet in order to manage the services. The paradox here lies in the fact that, while doing so we have to face a major trade off in the form of latency. This latency is obviously brought about by the increase in hardware required to cater to the above-mentioned services. This means the recipient of a data packet will wait for a longer time. This calls for improved Hardware design of the Router. Hence we are focusing on what we call wire-speed routing.

1.3.2 Disadvantages of Software Based Routers

There are some apparent disadvantages of Software Based Routers, namely

- ***i.*** The system is bulky because computer systems are used for S/W based routing
- ***ii.*** The energy consumption is high.
- ***iii.*** A large amount of latency is introduced owing to the sequential processing of software instructions.
- ***iv.*** Requires a lot of monitoring once the system is setup.
- ***v.*** Wire-speed processing becomes a far cry.

1.3.3 Advantage of Hardware Routers

There are some definite advantages of Hardware Routers over the conventional routers, which have made them the obvious choice in the current high-speed networking paradigm.

- ***i.*** Higher throughput than any other conventional (software + computer) router.
- ***ii.*** Much more energy efficient than a conventional router.
- ***iii.*** Does not require a computer to be running to allow access to other computers.
- ***iv.*** Reliable and runs without much, if any attention, once it is set up.
- ***v.*** Contains no files or software that can be harmed, corrupted, deleted, etc.

1.4 Organization of the Report

Chapter 1 provides an Introduction to the project. It defines the Problem Statement, gives an overview of the Current status and the motivation for the project.

Chapter 2 presents the design details and the underlying theory/principles for the Fast IP Lookup Module (FILM). A brief introduction to the IP Lookup Problem and Forwarding Tables in routers is provided, followed by a detailed explanation of the Lulea Scheme, [5] which is the lookup algorithm being implemented.

Chapter 3 describes the implementation of the FILM. The data structure extends to two levels. The Level 1 and Level 2 Architectures are described and the working of the controllers for each module is presented.

Chapter 4 summarizes the results obtained. The functional simulation results of the design using ModelSim is presented along with the implementation results for Xilinx VIRTEX FPGAs.

Chapter 5 provides a conclusion, summarizing the work done so far and hinting at possible directions for future work.

2. Design Detail/Materials and Methods/Theory/Principles

2.1 Background

2.1.1 The Internet

The Internet is a plethora of networks and this forms a huge electronic medium over which data travels to and for in the form of small packets. Any data packet transmitted from one computer connected to the Internet has to travel through many such networks to finally reach its ultimate destination. The data packet, of course, carries the address of the machine that is its destination and this helps it travel through the Internet. The Internet has several devices that enable the data packet to travel in the direction of its destination and finally directing it there.

2.1.2 Routing

Routing is an operation that is responsible for directing a data packet towards its destination. The devices specialized for facilitating Routing are Bridges, Switches, and Routers respectively.

Following are the primary objectives of Routing:

- ***i.*** To successfully deliver data packets to respective destinations
- ***ii.*** To determine the shortest path between the source and destination.
- ***iii.*** To ensure that the data packet remains error free throughout the routing process.

Every data packet carries with it the address of the destination in addition to that of the source. This address is used by the routers to direct the data towards its destination. Each router knows all the nodes that it is connected to and in turn, is aware of the networks that these nodes are linked to. The router maintains a table, which holds the addresses of all the nodes linked to it. The destination address in the data packet is compared with the entries in this table and the next hop is determined. The packet is then sent over to the node thus determined. A similar process repeats in the subsequent nodes, until the data packet reaches the destination.

Such a process of routing has one ill effect in that there is a possibility of a data packet going around the network, without reaching the destination at all. This may be due to Address Tables in routers that are not updated or a broken link or some other error. Whatever the reason, it results in an enormous wastage of resources. Such a situation is avoided by associating with each data packet a parameter called “Time to Live” so that, if the data packet does not reach the destination within a certain number of hops, it is simply discarded.

2.1.3 The Best Matching Prefix Problem

When an Internet router gets a packet from an input link interface, it uses the destination address in the packet to look up a routing database. The result of the lookup provides an

output link interface, to which the packet is forwarded. Thus, the major tasks in packet forwarding are address lookup and switching packets between link interfaces.

Address lookup can be done at high speeds if an *exact match* of the packet destination address to a corresponding address in the routing table is desired. Exact matching can be done through standard techniques like hashing or binary search. But, most routing protocols (including OSI and IP) use hierarchical addressing to avoid scaling problems. Instead of storing an entry for each possible destination IP address, the router stores address *prefixes* that represent a group of addresses that can be reached through the same interface.

But multiple prefixes may match a given address. If a packet matches multiple prefixes, it is intuitive that the packet be forwarded corresponding to the *longest matching prefix*. IPv4 prefixes are arbitrary bit strings up to 32 bits long, as shown in Table 1. To see the difference between exact matching and best-matching prefix, consider a 32-bit address *A*, whose first 8 bits are 10000111. If we searched for an exact match for *A* in the above table, a match would not be found. However, prefix matches are 100* and 1000*, of which the longest matching prefix is 1000*, whose next hop is *L5*.

2.1.4 Forwarding Tables

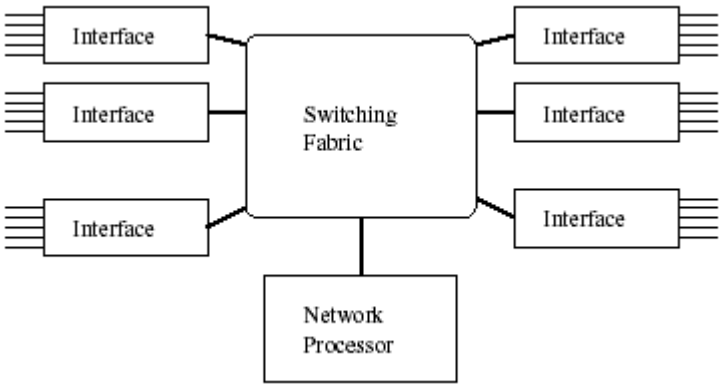


Figure 1. Router Design

A router design is schematically shown in Figure 1. A number of *network interfaces* and a *network processor* are interconnected with a *switching fabric*. The network processor ideally implements all layers of the protocol stack, (typically, only the network layer). It performs packet processing and packet forwarding, i.e. it determines which outbound interface should be used to send each packet that arrives at an inbound interface. To arrive at this decision, a *forwarding table* is used that is updated in intervals that are a few orders of time higher than that for address lookup. Routing updates can be frequent, but since routing protocols need time in the order of minutes to converge, forwarding tables need not change more than at most once per second. The network processor needs a dynamic routing table designed for fast updates and fast generation of forwarding tables. The forwarding tables, however, need to be optimized for lookup speed and need not be dynamic.

2.1.5 Design goals

The data structure used for the forwarding table is to be chosen on the basis of minimum lookup time. To achieve that, it is required to minimize both,

- the number of *memory accesses* required during lookup
- the *size* of the data structure.

Reducing the number of memory accesses required during a lookup is important as memory accesses are the bottleneck of lookup procedures. If the data structures can be made small enough, faster (but more expensive) SRAM can be used to store the entire forwarding table. This makes memory accesses orders or magnitude faster than if the data structure were to reside in memory consisting of relatively slow DRAM as in the case of Patricia Tries. It is also desirable that the data structure should avoid comparisons and bit-extraction operations to simplify implementation.

2.2 The Lulea Scheme

2.2.1 Why this Scheme

Our lookup module implements the Lulea scheme due to Degermark et al [5]. The scheme proposes a forwarding table data structure designed for quick routing lookups. This scheme was primarily aimed for implementation in software with general-purpose processors by exploiting the processor cache to store the entire forwarding table. Our implementation proposes multiple lookup modules to ensure giga-bit routing speeds and to be able to use SRAM in each of these modules; it becomes necessary to use small routing tables. The lookup time, according to [5], is a worst case of eight memory accesses, which with SRAM of cycle times of 25 ns would give us a worst-case address lookup time of 200 ns per module.

2.2.2 The Data Structure

The forwarding table is a tree with levels. To search each level, one to four memory accesses is required. But, with most routing tables, the majority of lookups need searching in only one or two levels, so the most likely number of accesses is eight or less.

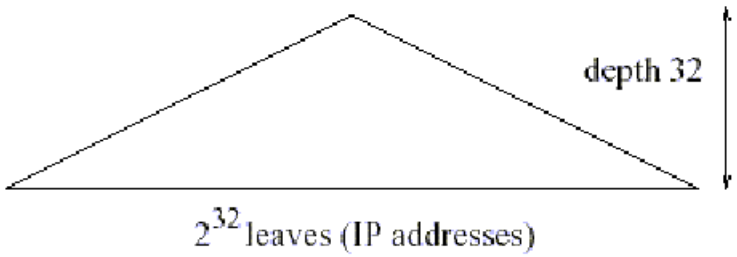


Figure 2. Binary Tree spanning the entire IP address space

Consider a binary tree spanning the entire IP address space (Figure 2). It has a depth of 32 and has 2^32 leaves, one for each possible IP address. The prefix of a routing table entry defines a path in the tree ending in some node. All the IP addresses (leaves) in the subtree rooted at that node should be routed according to that routing entry. In this manner, each entry defines a range of IP addresses with identical routing information.

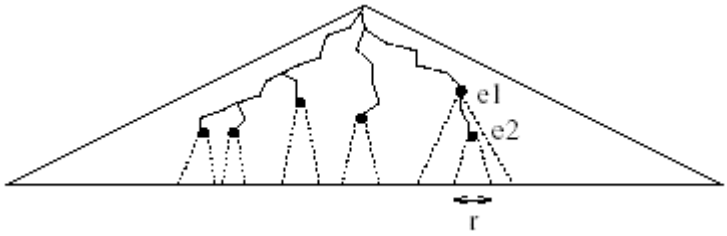


Figure 3. Routing entries defining ranges of IP addresses

If several routing entries cover the same IP address, the rule of the *longest match* is applied as illustrated in Figure 3, where the entry *e1* is hidden by *e2* for addresses in the range *r*. The forwarding table must be a *complete prefix tree* in order to build a small forwarding table - each node in the tree must have either two or no children. Nodes with a single child must be expanded to have two children; the children added this way are always leaves and their next-hop information is the same as the next-hop of the closest ancestor with next-hop information, or the "undefined" next-hop if no such ancestor exists. This is illustrated in Figure 4.

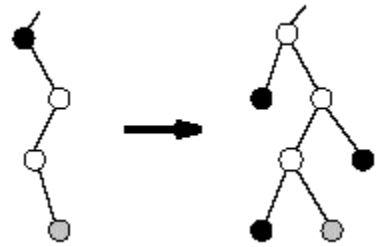


Figure 4. Expanding the prefix tree to be complete

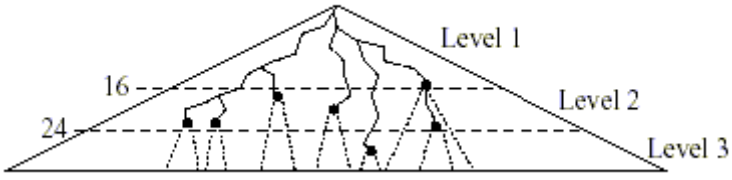


Figure 5. The three levels of the data structure

As shown in Figure 5, level one of the data structure covers the prefix tree to depth 16; level two covers depths 17 to 24 and level three, depths 25 to 32. Wherever a part of the prefix tree extends below level 16, a level-two *chunk* describes that part of the tree.

Similarly, chunks at level three describe parts of the prefix tree that are deeper than 24. The result of searching one level of the data structure is either an index into the next-hop table or an index into an array of chunks for the next level.

2.2.3 Level 1 of the data structure

The first level is a tree node with 1 to 64k children. Imagine a cut through the prefix tree at depth 16. The cut is represented by a bit-vector, with one bit per possible node at depth 16, thus requiring 8 KB.

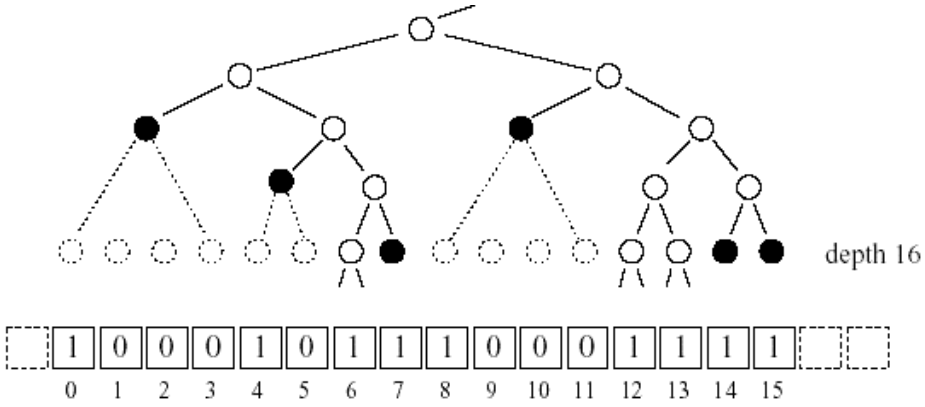


Figure 6. Part of cut with corresponding bit-vector

Heads: When there is a node in the prefix tree at a depth less than or equal to 16, the lowest bit in the interval covered by that leaf is set to one and all other bits to zero. A bit in the bit vector can thus be

- a one indicating that the prefix tree continues below the cut; a *root head* (bits 6, 12 and 13 in Figure 6), or
- a one representing a leaf at depth 16 or less; a *genuine head* (bits 0, 4, 7, 8, 14 and 15 in Figure 6), or
- a zero, indicating that this value is a *member* of a range covered by the leaf at a depth less than 16. Members have the same next-hop as the largest head smaller than the members (bits 1, 2, 3, 5, 9, 10 and 11 in Figure 6).

Head Information: For genuine heads, an index is to be stored into the next-hop table with members using the same next-hop as the largest head smaller than the members. For root heads, an index is to be stored to the level-two chunk that represents the corresponding sub-tree. This head information is encoded in 16-bit *pointers* stored consecutively in an array. Two bits of each pointer encode the pointer type.

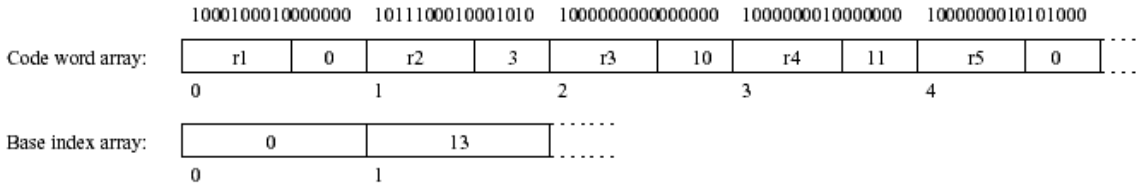


Figure 7. Bit masks vs Code Words and Base indices

Finding pointer groups: The bit-vector is divided into bit-masks of length 16. There are 2^12 = 4096 bit-masks. There are as many pointers associated with a bit-mask as its number of set bits. Figure 7 illustrates how the data structure for finding pointers corresponds to the bit-masks. It consists of an array of *code words*, as many as there are bit-masks, and also an array of base indices, one per four code words.

Each code word has a 10-bit value (r1, r2, ...) and a 6-bit offset (0, 3, 10, 11,...). The first bit-mask in Figure 7 has three set bits. The second code word thus has an offset of three because three pointers must be skipped over to find the first pointer associated with that bit-mask. The second bit-mask has 7 bits and consequently the offset in the third code word is 3 + 7 = 10.

After four code words, the offset value may become too large to represent with 6 bits. Therefore, a base index is used together with the offset to find a group of pointers. There can be at most 64K pointers to level 1 of the data structure, so the base indices need to be at most 16 bits. In Figure 7, the second base index is 13 because there are 13 set bits in the first four bit-masks. To locate a group of pointers, the first 12 bits of the IP address are used to index the proper cod word and the first 10 bits are an index into the array of base indices.

Map Table: Within the group of pointers, the correct pointer is to be found. The 10-bit value (r1, r2, ...) is an index into a table that maps bit-numbers in the IP address to pointer offsets. As bit-masks are generated from a complete prefix tree, not all combinations of the 16 bits are possible.

A non-zero bit-mask of length *2n* can be any combination of two bit-masks of length *n* or the bit-mask with value 1. Let *a(n)* be the number of possible non-zero bit-masks of length 2^n. a(n) is defined by the recurrence a(0) = 1, a(n) = 1 + a(n-1)^2. The number of possible bit-masks with length 2^4 is thus a(4) + 1 = 678, the additional one is because the bit-mask can be zero. An index into a table with an entry for each bit-mask thus requires only 10 bits.

A table called **Map Table** is maintained to map bit numbers within a bit-mask to 4-bit offsets. The offset indicates the number of pointers to skip to find the required pointer and hence equals the number of set bits smaller than the bit index. These offsets are independent of the forwarding table – **Map Table** is constant and is generated once and for all.

Searching: Figure 8 illustrates the procedure of searching level 1 of the data structure. Code words are stored in an array called *code*, base indices in *base*, and level 1 pointers in *level1_ptrs*.

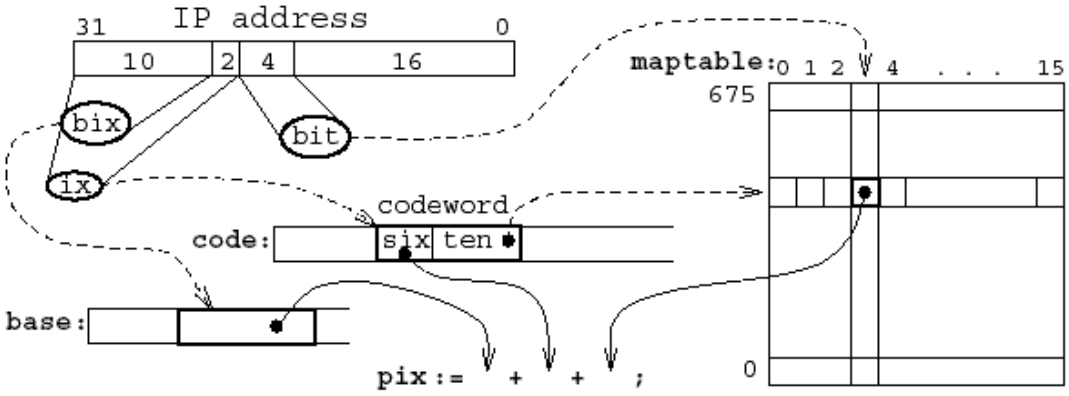


Figure 8. Finding the pointer index

cix := high 12 bits of IP address
bix := high 10 bits of IP address
mapcol := low 4 of high 16 bits of IP address
codeword := code[cix]
maprow := high 10 bits of codeword
offset := low 6 bits of codeword
pix := base[bix] + offset + maptable[maprow][mapcol]
pointer := level1_ptrs[pix]

After the pointer is retrieved, it is examined to determine if it is a next-hop (search ends) or a pointer to the next level (search continues). The code requires only a few bit extractions, array references, and additions - no multiplication or division instructions are required except for the implicit multiplication when indexing an array. This makes it feasible to implement the algorithm in hardware.

To search the first level, 7 bytes are to be accessed:

- a two byte code word
- a two byte base address
- a Map Table
- a two byte pointer

The size of the first level is

- 8KB for the code word array

- 2KB for the base index array
- 2B to 128 KB for the pointers

All three levels share the 5.3 KB required by Map Table.

2.2.4 Levels 2 and 3 of the data structure

Levels 2 and 3 of the data structure consist of chunks. A chunk covers a sub-tree of height 8 and can contain at most 256 heads. There are three kinds of chunks depending on the number of heads in the imaginary bit-vector.

- 1-8 heads, *sparse* chunk, represented by an array of the 8-bit indices of the heads, plus eight 16-bit pointers, a total of 24 bytes.
- 9-64 heads, *dense* chunk, represented analogously with level 1, except for the number of base indices. Only one base index is required for all 16 code words as 6-bit offsets can cover all 64 pointers. A total of 34 bytes are needed, plus 18 to 128 bytes for pointers.
- 65-256 heads, *very dense* chunk, represented analogously with level 1. 16 code words and 4 base indices give a total of 40 bytes. In addition the 65 to 256 pointers require 130 to 512 bytes.

Dense and very dense chunks are searched analogously with the first level. For sparse chunks, the 1 to 8 values are placed in decreasing order. To avoid a bad worst-case when searching, the fourth value is examined to determine if the desired element is among the first four or last four elements. After that, a linear scan determines the index of the desired element, and the pointer with that index can be extracted. The first element less than or equal to the search key is the desired element. At most 7 bytes need to be accessed to search a sparse chunk.

2.2 Performance Measurement

Site	Date	Year	Routing entries	Leaves	next-hops	Size (Kb)	Build time	sparse chunks	dense chunks	dense+ chunks	level 3 chunks
Mae East	Jan 9	'97	32732	58714	56	160	99 ms	1199	587	186	2
Mae East	Oct 21	'96	38141	36607	50	148	91 ms	1060	593	149	4
Sprint	Jan 1	'97	21797	43513	17	123	72 ms	988	483	98	3
PacBell	Jan 28	'97	18308	33250	2	99	49 ms	873	357	67	0
Mae West	Jan 1	'97	12049	28273	51	86	46 ms	775	312	42	3
AADS	Jan 4	'97	1109	5670	12	28	11 ms	320	38	0	2

Table 1. Forwarding Table Generation Data

3. Implementation

3.1 Overview of Router Chip

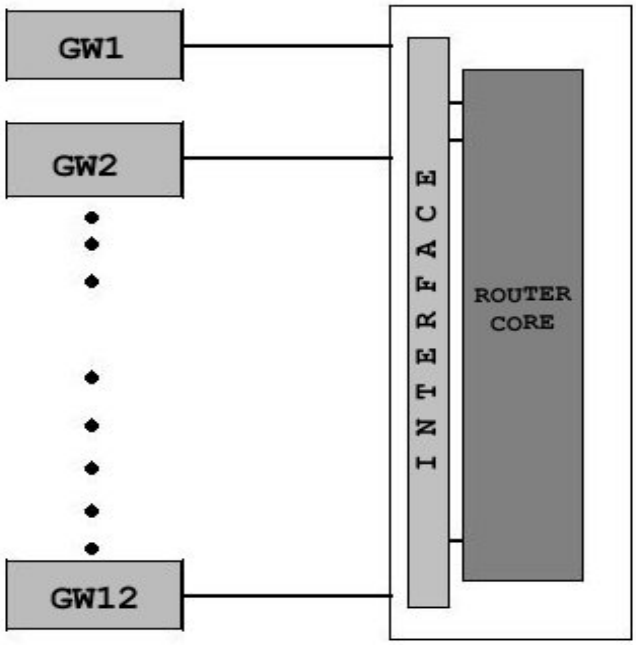


Figure 9. Router Core Interfaced to the Gateways

Figure 9 shows a simplified schematic of the router chip for which the lookup module is being designed. Three types of units handle the functions of the chip, namely:

- Serial Input/Output Unit
- Memory Management Unit and
- Address Lookup Unit

Each of the 12 ports is serviced by an I/O Unit, which processes packets before sending each one to one of the free memory slots indicated by the Memory Management Unit. The I/O unit computes the CRC for the incoming frame checks the type field in the Ethernet frame and checks the IP header checksum. The Memory Management Unit allocates a free memory slot, if available, otherwise uses an LRU-based scheme to free a slot. Though not finalized, it is proposed to have one lookup module for every four Packet Memories. The memory management unit reads the tags corresponding to each group of packet memories and presents IP addresses for packets to be looked up, by turn, to the Address Lookup Unit which then finds the next-hop corresponding to that address by accessing the compressed forwarding table.

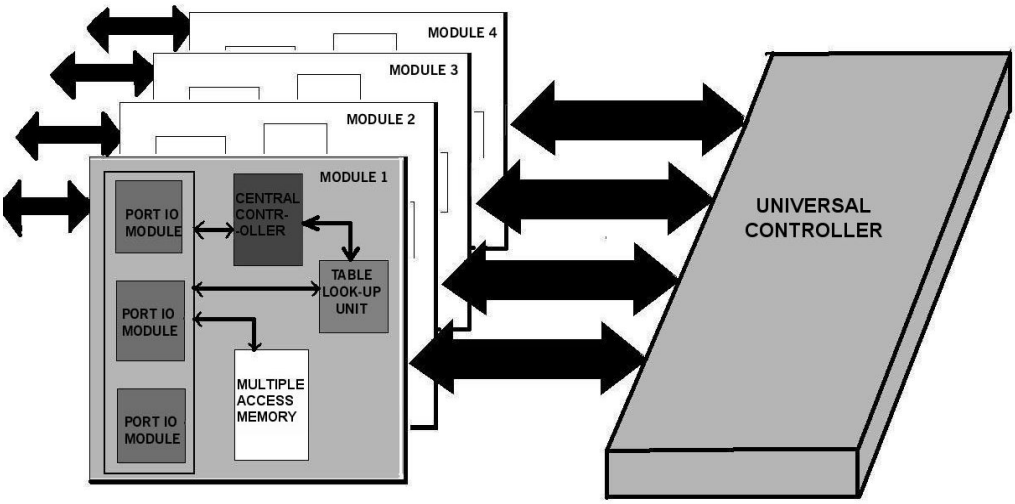


Figure 10. Top Level Architecture of the Router Chip

The 12 I/O ports are divided into four similar modules. Each module comprises of the following functional units:

i. Serial I/O Processors, which are three in number per module
ii. One Central Controller, which controls and co-ordinates the interaction between the functional units
iii. One Table Look-Up unit, which determines the next hop address for the IP packet based on the Destination address
iv. 16 Memory Blocks, each of length 2Kb.

Apart from the above, there is a Universal Controller that manages the interaction between the four modules. Figure 10 illustrates the top-level architecture of the Router Core.

3.2 Top Level and Block schematic of the Fast IP Lookup Module

Figure 11 shows the *entity* for the Fast IP Lookup Module (FILM). The inputs to the unit are the IP address to be looked up (*IPAddr*) 24-bit, clock and reset signals and *StartLookupt* (1-bit) that tells the FILM to start the lookup. The outputs are an 8-bit *NextHop* and a *NextHopValid* signal. The NextHop indicates the output interface to which the packet with the given IP address must be forwarded. The *NextHopValid* signal indicates that the lookup is finished and that the *NextHop* can be used. The router chip being designed performs routing only on the high 24 bits of the IP address. The majority of routing tables have very few entries with prefixes longer than 24 bits and this reduces the number of levels of the data structure to two, thus giving a worst case of eight memory accesses.

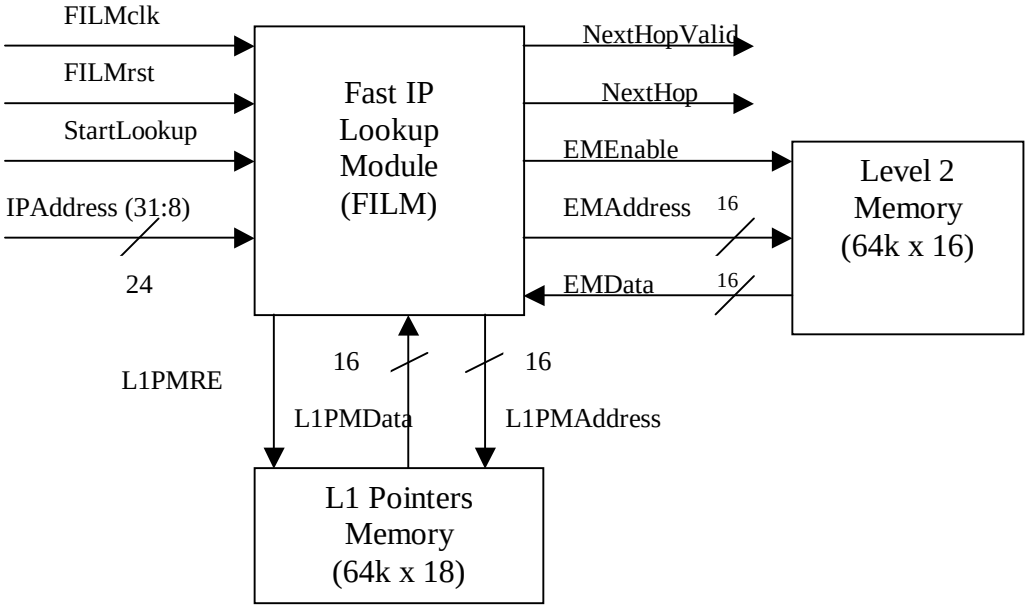


Figure 11: Entity for the Fast IP Lookup Module

The FILM also contains interfacing signals (Read Enable, Address and Data lines) to two memory modules each of 64k words – the Level 1 Pointers Memory and the Level 2 Memory.

Figure 12 shows the block schematic of the FILM. It consists of four FSM blocks – namely the Level 1, Level 2 Sparse Chunk, Level 2 Dense Chunk and the Level 2 Very Dense Chunk blocks. Three of these use the MapTable ROM. To avoid contention between these modules, a SelectController is used which generates appropriate StartLookup signals for each module and chooses the right NextHop output.

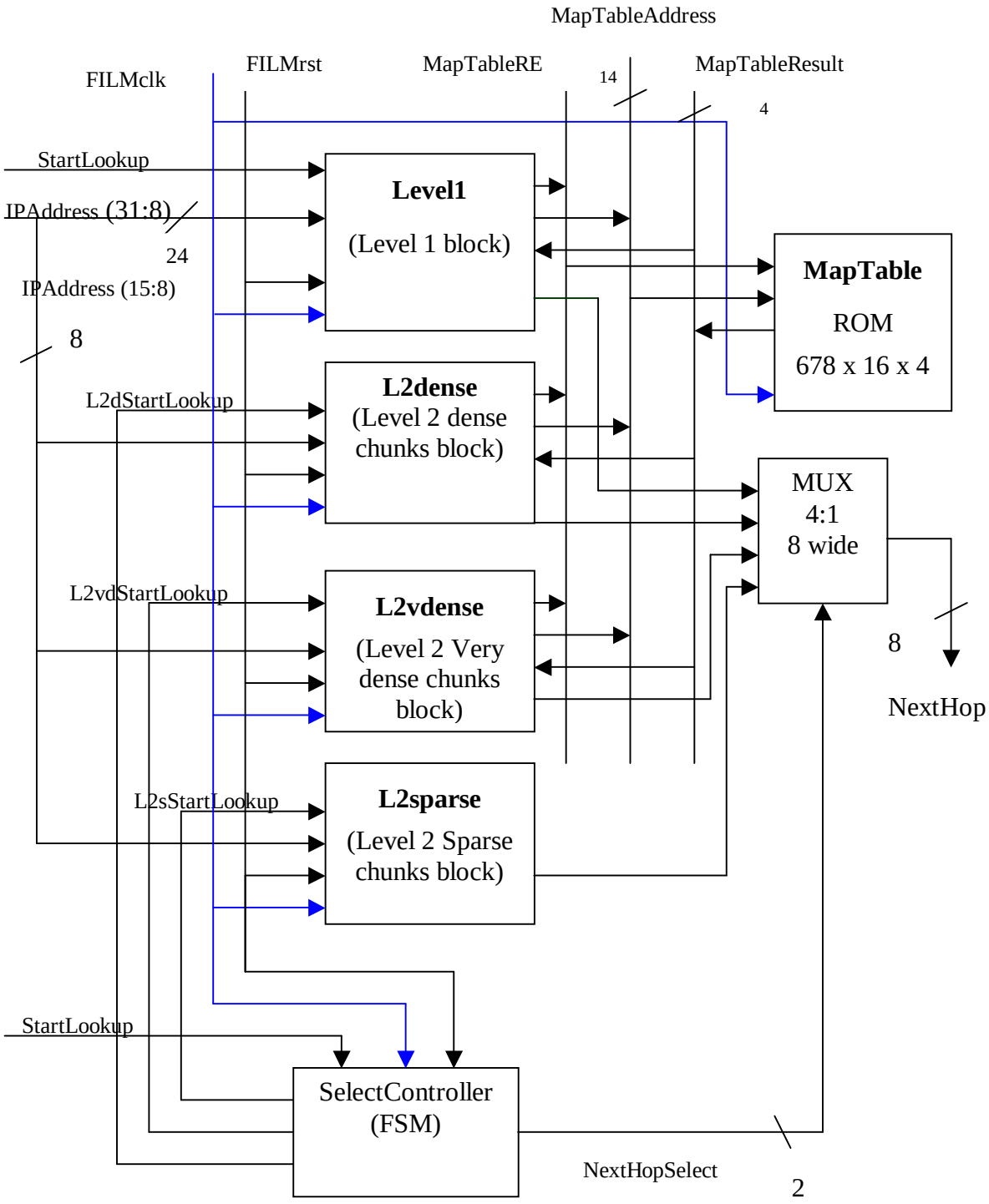


Figure 12: Overview of the Fast IP Lookup Module Architecture

3.3 Level 1 Architecture

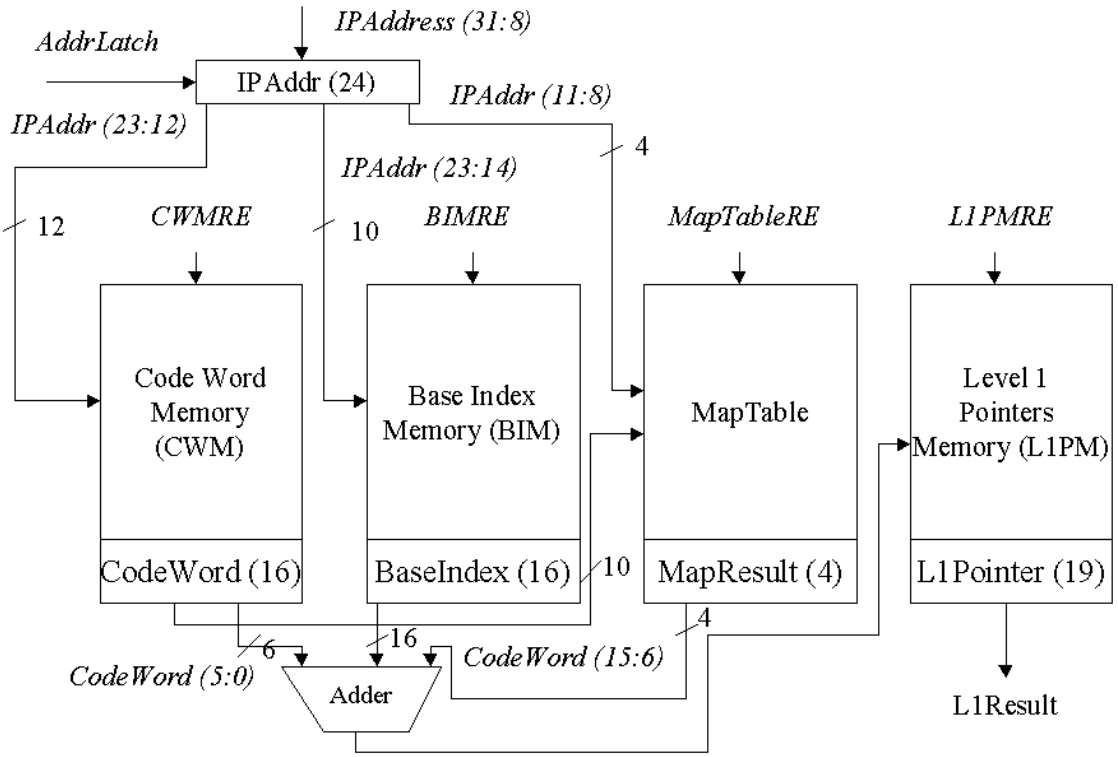


Figure 13: Level 1 Architecture

Figure 13 shows the *architecture* for Level 1 of the ALM. The *IPAddr* lines go to a 24-bit register that latches the value of the IP Address when *StartLookup* signal goes high. The high 10 bits of the IP address are used to index the Base Index Memory (BIM) to obtain *BaseIndex* (16 bit). The high 12 bits of the IP address are used to index the Code Word Memory (CWM), to obtain the *CodeWord*. These two memory accesses are done simultaneously.

MapTable ROM is implemented as a 678 x 64 memory. A 10-bit Row Address and a 4-bit Column Address are used to access first one of the 678 rows, and then, one of the 16 columns is selected to yield the 4-bit *MapTableResult*. This constitutes the second memory access at level 1. *BaseIndex* and *MapResult* are added with the lower 6 bits of *CodeWord* to obtain *L1PMAddress*, which is an index into an array of Level 1 pointers. The resulting 16-bit value is used to index the Level 1 Pointers Memory (L1PM) to obtain the *L1Result* (18-bit). This is the third memory access at level 1. The high two bits of *L1Result* are interpreted as:

- 00 - NextHop (search ends, LEnd asserted)
- 01 - Pointer to sparse chunk (search continues to level 2)
- 10 - Pointer to dense chunk (search continues to level 2)
- 11 – Pointer to very dense chunk (search continues to level 2)

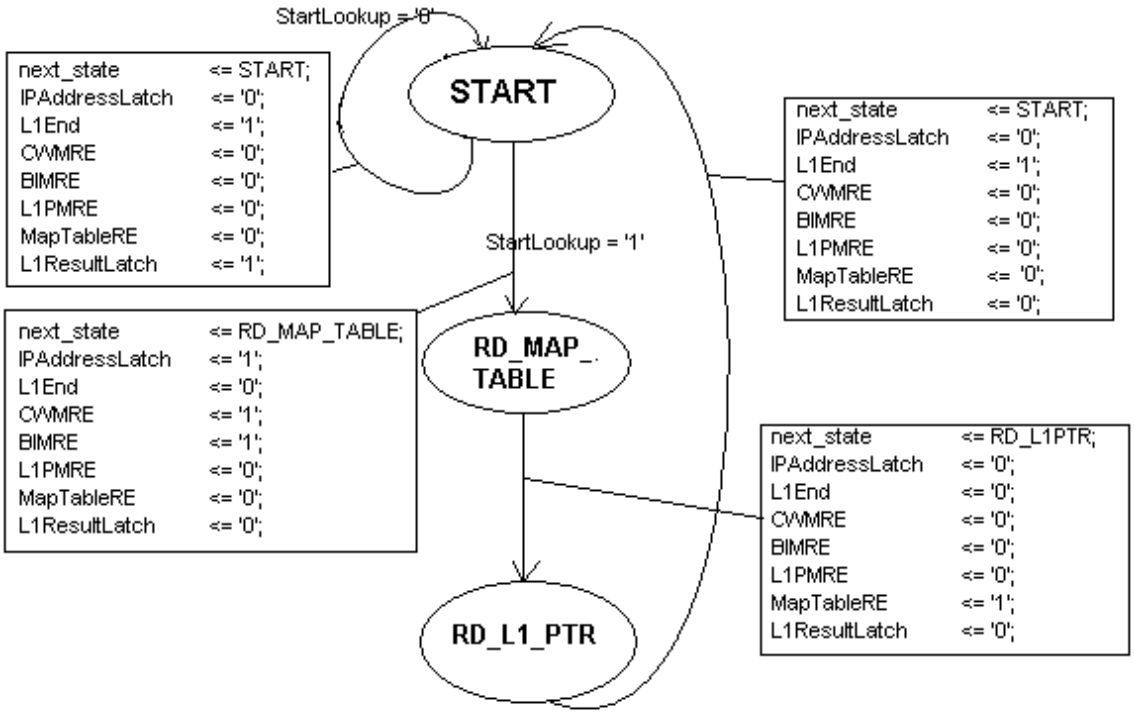


Figure 14: State Diagram for the Level 1 Controller

Figure 14 shows the state diagram for the level 1 controller. In the START state, the controller has finished the previous lookup and is awaiting the *StartLookup* signal to start a new lookup sequence. Hence, the *L1End* and *L1ResultLatch* signals are asserted.

When the *StartLookup* signal arrives, the input IP address is latched by asserting the *IPAddressLatch* signal and the Code Word Memory and the Base Index Memory are simultaneously read by asserting the *CWMRE* and the *BIMRE* signals. The FSM then moves to the Read Map Table (RD_MAP_TABLE) state.

In the RD_MAP_TABLE state, the Map Table is read by asserting the *MapTableRE* signal. The next state is the Read Level 1 Pointer (RD_L1_PTR) state.

In the RD_L1_PTR state, *L1End* is asserted before the end of this last memory access. This is done to facilitate the Select Controller to transfer control smoothly from level 1 to level 2 without wasting one clock cycle. The Select Controller, in case the search ends at Level 1, asserts the actual *NextHopValid*, only at the end of the RD_L1_PTR state. After this state, the FSM returns to the START state.

3.4 Level 2 Sparse Architecture

The level 2 sparse chunk block is used for those chunks with 1 to 8 heads. Using Base Indices, Code Words and Map Table would be cumbersome. Hence, a linear search is used among the 8 heads with the middle element being checked first to avoid a bad worst case. The key for the search here is the third octet of the latched IP address. The heads are sorted in descending order in the sparse chunk memory.

The result of the Level 1 search gives a L1Pointer that is used to store the address of the top element of the sparse chunk as well as the offset to the middle element from the top of the chunk, within the 16 bits of the L1Pointer. This is made possible by storing the L2 Sparse chunks in the initial addresses of the Level 2 Memory and keeping in mind that the L2 Sparse Chunks together amount to less than 32 K words of memory. So, only 14 bits are required for the address of the top element of the chunk. The offset to the middle element can be at-most three; hence the remaining two bits of L1Pointer are used for storing this.

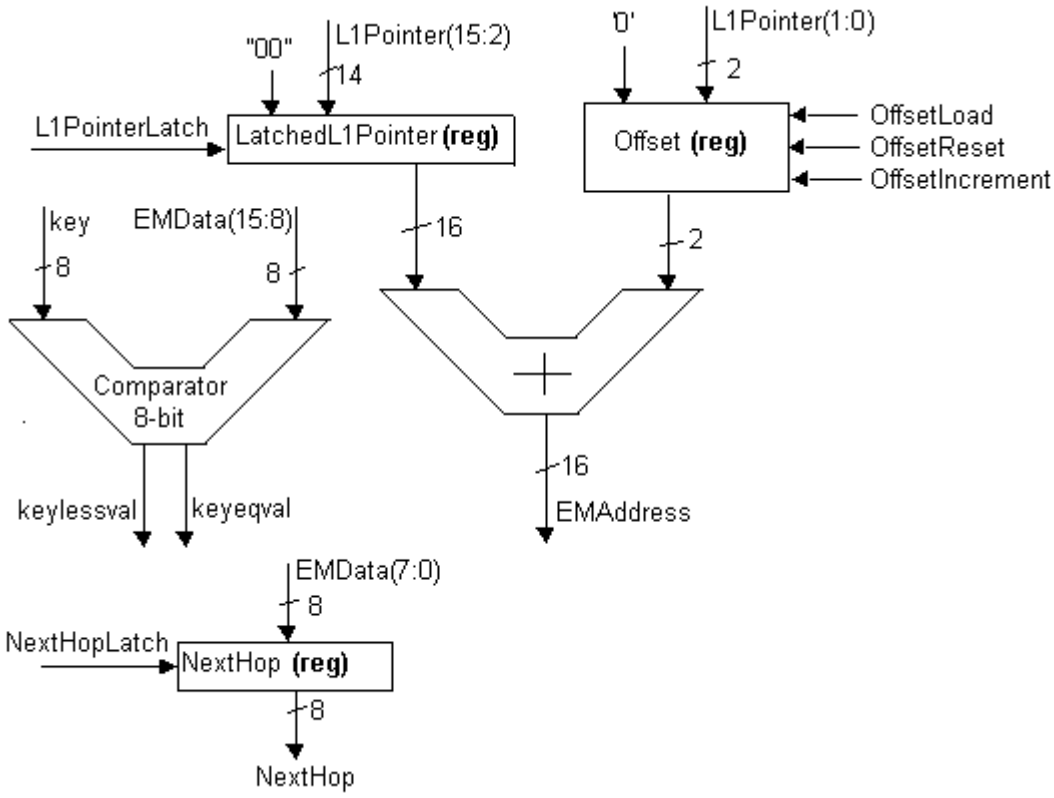
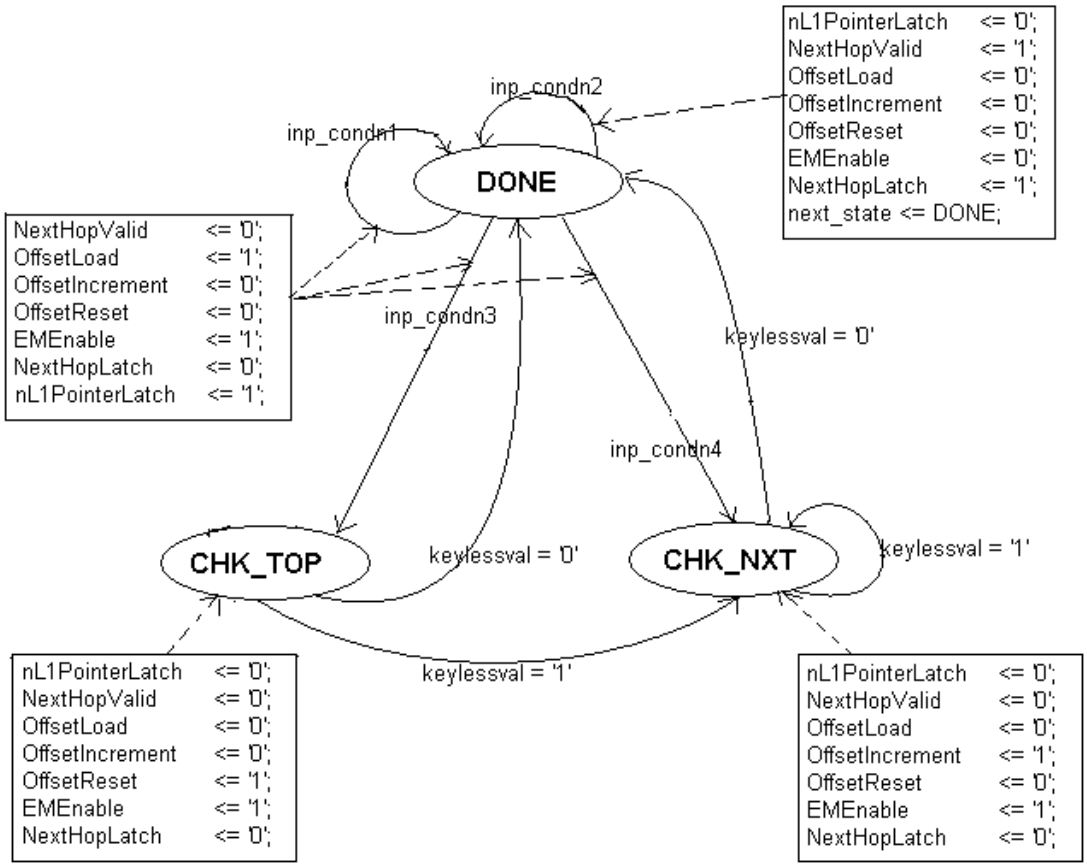


Figure 15: Level 2 Sparse Chunk Architecture

The Level 2 sparse chunk controller is initially in the DONE state and is awaiting the *StartLookup* signal to begin another lookup sequence. The *NextHopLatch* signal is asserted to latch the Next Hop and the *NextHopValid* signal is asserted to indicate that lookup is finished.



inp_condn1*: StartLookup* = '1' and *KeyEqVal* = '1'
inp_condn2: *StartLookup* = '0'
inp_condn3: *StartLookup* = '1' and *KeyEqVal* = '0' and *KeyLessVal* = '0'
inp_condn4: *StartLookup* = '1' and *KeyEqVal* = '0' and *KeyLessVal* = '1'

Figure 16: State Diagram for Level 2 Sparse Chunks Lookup Controller

When the *StartLookup* signal arrives, the *OffsetLoad* signal is asserted to load the offset of the middle element into the Offset register, from L1Pointer. This address is used to access the middle element of the sparse chunk from the Level 2 Memory. The resulting head is then compared with the key. The result of the comparison arrives in the signals *KeyEqVal* and *KeyLessVal*, which are self-explanatory. If Key equals Val, the middle element is the match, and the corresponding next hop, which is stored along with the 8-bit head as one 16-bit word is now the required next hop. The FSM, in this case, now returns to the DONE state. If the *KeyLessVal* signal is asserted after the comparison, the required head lies below the middle element and searching proceeds with the next element by moving to the CHK_NXT state. If *KeyEqVal* and *KeyLessVal*, both are deasserted, the key is greater than the middle-element and hence, searching begins from the top. The same rules continue till an element is found that is greater than or equal to key. The search cannot fail, at worst the last element will match.

3.5 Level 2 Dense Chunk Architecture

The level 2 dense chunk blocks are similar to the level 1 blocks and are used for chunks with 9 to 64 heads. As the maximum offset from the top of the chunk possible, i.e. 64, fits within the 6 bits reserved for offset within a 16-bit Code Word as in Level 1, no Base Indices are required. 16 Code Words are required to cover the 256 bit Bit-Vector. An offset of 16 from the L1Pointer is required to obtain the actual first element of the dense chunk memory. Further, as next hops are 8-bit, two of them are fit into a 16-bit word and the appropriate byte is chosen using the LSB of the *NextHopOffset1* signal. The Address Select multiplexer chooses the right offset to add to L1Pointer to obtain the Level 2 Memory Address.

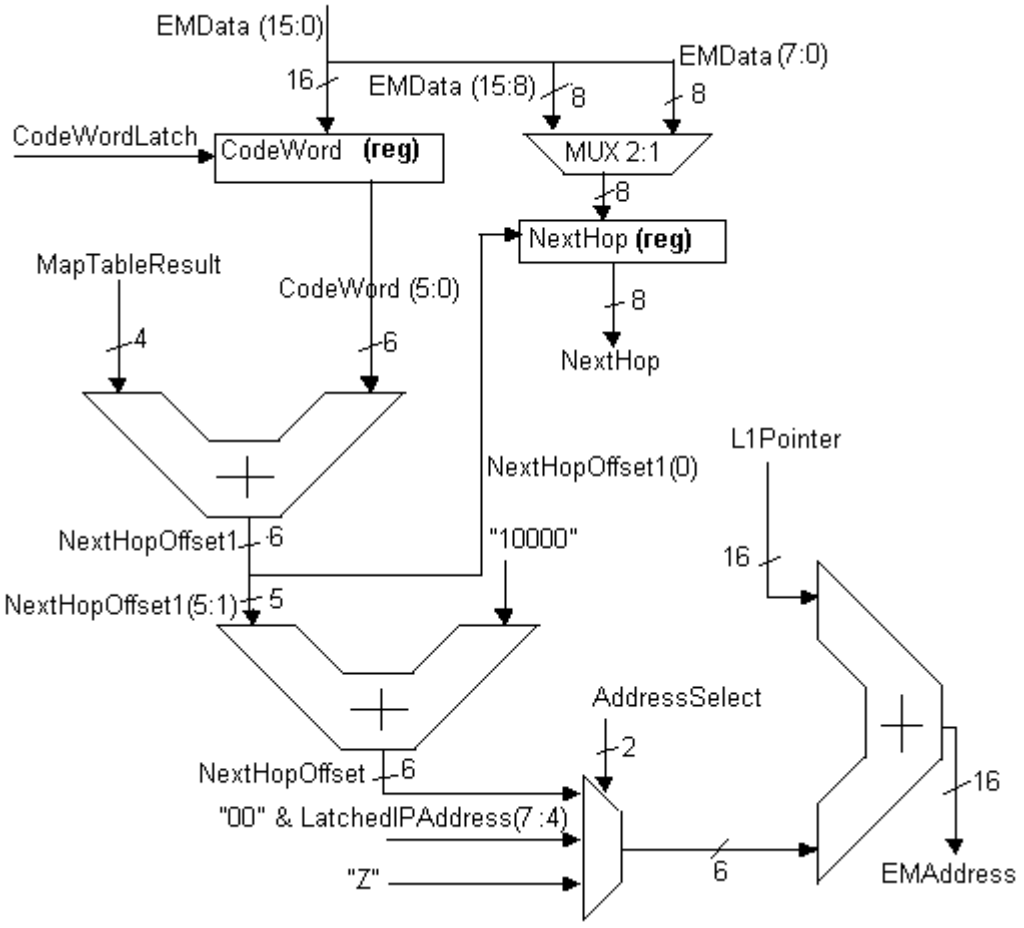


Figure 17: Level 2 Dense Chunks Architecture

In the START state, the dense lookup controller has finished lookup and asserts the *NextHopLatch* and *NextHopValid* signals. *AddressSelect* tri-states the *EMAddress* lines as no memory access is to be performed here. When the *StartLookup* signal arrives, the Level 2 Memory is read to get the *CodeWord*. *AddressSelect* chooses the high four bits of the third octet of the latched IP address as *Offset* to be added to *L1Pointer* to obtain *EMAddress*.

The FSM moves to the RD_MAP_TABLE state, where the *MapTableRE* signal is asserted to obtain the *MapTableResult*. At the beginning of this state, *CodeWordLatch* is asserted and the *CodeWord* read in the previous state is latched on the positive edge of *CodeWordLatch*. *AddressSelect* again tri-states Offset as Level 2 Memory is not accessed in this state, instead the Map Table is to be read.

The Next Hop is read from the level 2 memory in the RD_NEXT_HOP state, and the Offset for the *EMAddress* is computed as shown in figure 17. Note that analogous to the L1End signal at Level 1, the *NextHopValid* signal is asserted one state earlier, i.e. at the beginning of the RD_NEXT_HOP state. This allows the final *NextHopValid* signal to be generated by the Select Controller without wasting a clock cycle after lookup is finished.

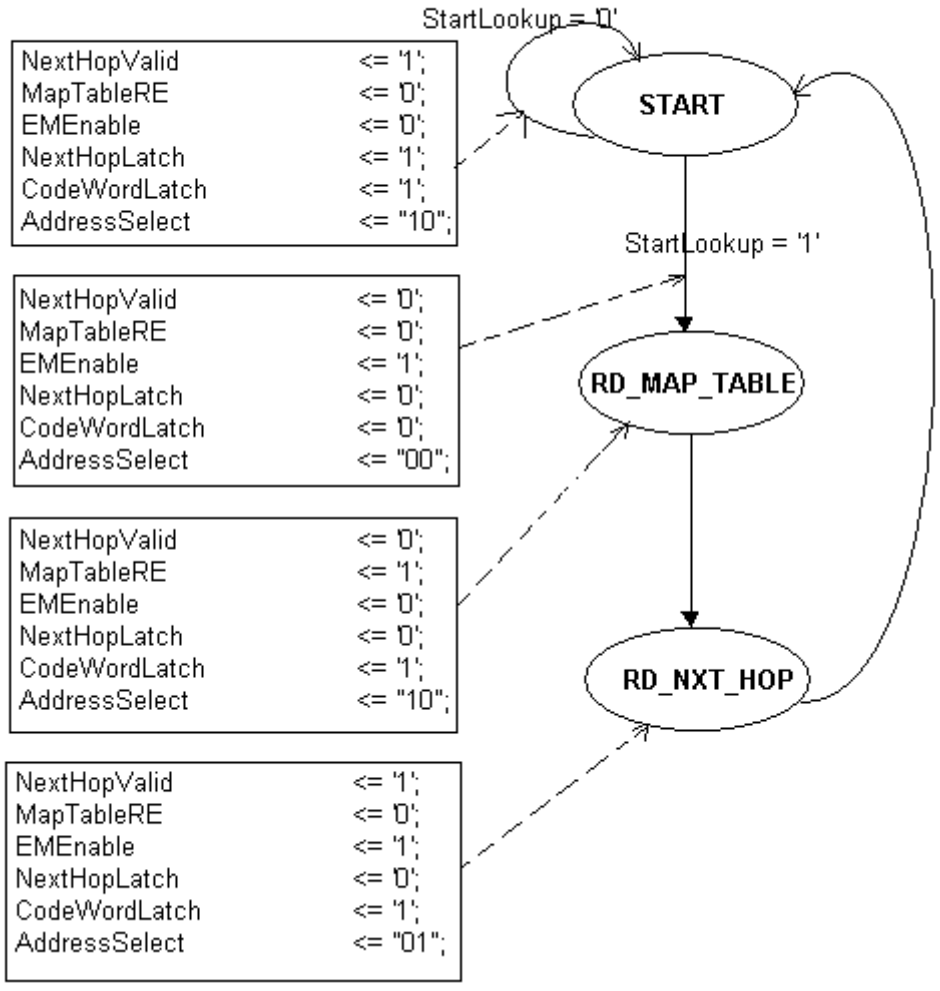


Figure 18: State Diagram for Level 2 Dense Chunk Controller

3.6 Level 2 Very Dense Chunks Architecture

The Very Dense Chunk Block consists of chunks with 65 to 256 heads. It is similar to the level 1 architecture, and uses 16 Code Words (of 16 bit) to cover the 256-bit Bit Vector. The maximum offset from the first head is 255, so, 4 Base Indices of 8-bit are required. The Address Select multiplexer chooses the right offset to add to L1Pointer to obtain the Level 2 Memory Address. An additional offset of 4 is required to access CodeWord from the Level 2 Memory (as the 4 Base Indices are stored at the top of the chunk followed by 16 CodeWords) and 20 to access the Next Hop.

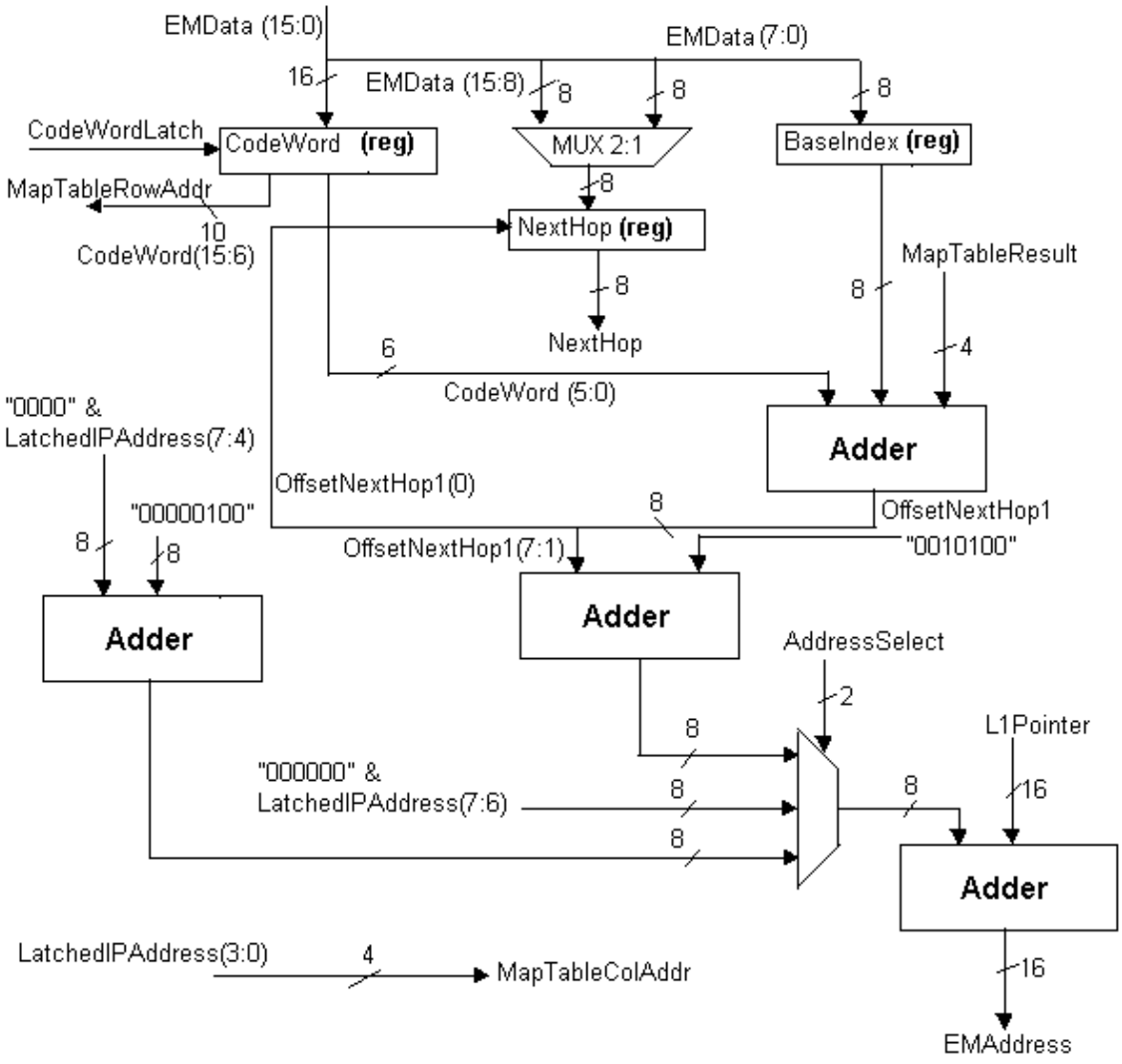


Figure 19: Level 2 Very Dense Chunks Architecture

In the START state, the very dense chunk controller has finished lookup and asserts the *NextHopLatch* and *NextHopValid* signals. *AddressSelect* tri-states the *EMAddress* lines as no memory access is to be performed here. When the *StartLookup* signal arrives, the Level 2 Memory is read to get the *BaseIndex.* AddressSelect uses the high 2 bits of the third octet of the latched IP address to obtain the offset for BaseIndex.

In the RD_CODE_WORD state the controller then reads the CodeWord using 4 + the high 4 bits of the third octet as Offset and also latches the BaseIndex. The MapTable is read in the RD_MAP_TABLE state and the EMAddress computed to access the Next Hop in the RD_NEXT_HOP state. As in the dense lookup controller, the NextHopValid is asserted one cycle early.

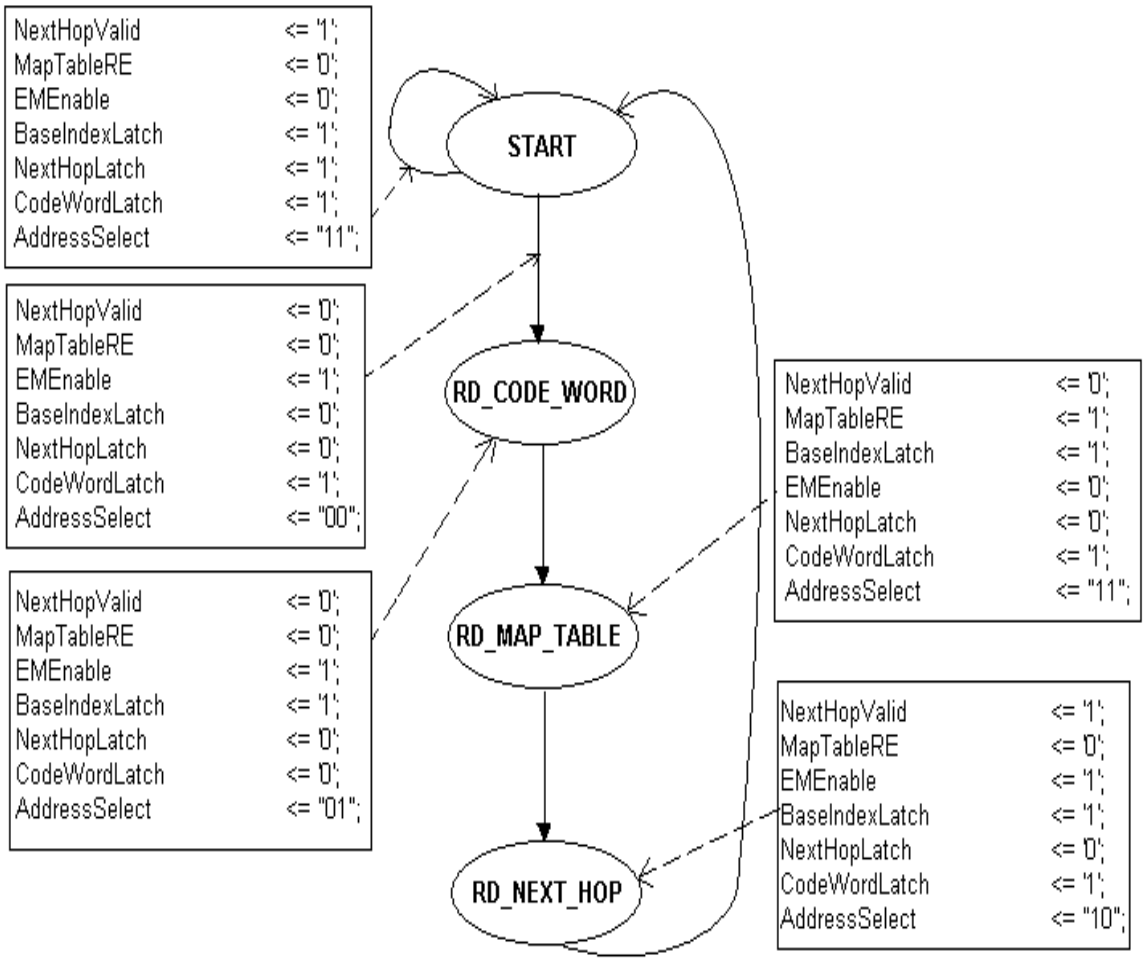


Figure 20. State Diagram for the Very Dense Chunk Controller

3.7 Select Controller

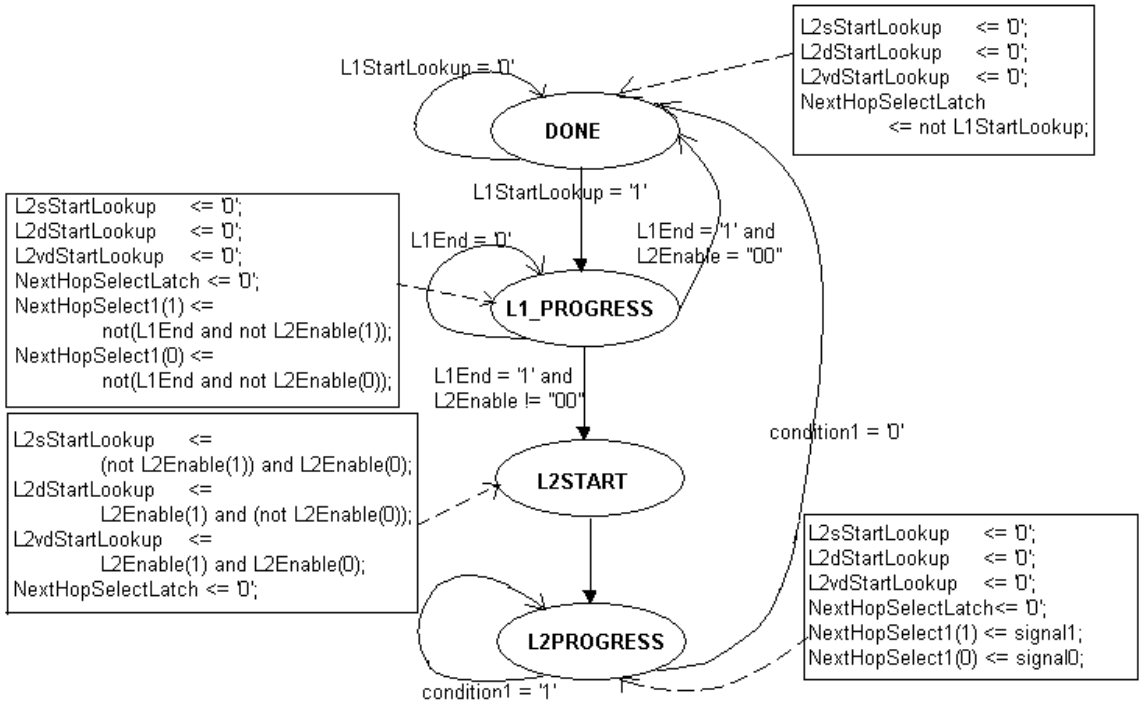


Figure 21: State Diagram for the Select Controller

The Select Controller is a 4-state FSM.

DONE: The Select Controller awaits the StartLookup signal. When the StartLookup signal arrives, the level 1 lookup controller latches the input IP address and begins the lookup. The Select Controller moves to the L1_PROGRESS state. The Level 2 controllers are disabled in this state.

L1_PROGRESS: In the L1_PROGRESS state, the Select Controller checks if the L1End signal is asserted. If not, it remains in the L1_PROGRESS state. If the L1End signal is asserted, the level 1 lookup is complete and the high two bits of the L1Result are input to the Select Controller are L2Enable. If L2Enable is “00”, the Level 1 Result is a next hop, search ends here and NextHopValid is asserted and NextHopSelect is made “00” to choose the Level 1 Next Hop. The FSM returns to DONE. Otherwise, the next state is L2START. The Level 2 controllers remain disabled in L1_PROGRESS state.

L2START: The search must now proceed to the second level. One of the three Level 2 controllers – Sparse, dense or Very dense is enabled by asserting the corresponding StartLookup signal to that controller based on the L2Enable signal. The search continues to the L2PROGRESS state.

L2PROGRESS: The controller monitors the NextHopValid signals from each of the Level 2 blocks, and if the enabled controller asserts its NextHopValid signal, it has completed lookup. The NextHopSelect now selects the NextHop from this L2 block and the controller asserts the FILM NextHopValid signal. The FSM returns to the DONE state.

4. Results and Discussion

4.1 Summary of Work Done

- The functional partitioning of the Router Chip was done. The interface or top-level for the FILM was defined.
- The lookup algorithm to be implemented was chosen – the Lulea algorithm [5] was chosen as it offered small forwarding tables and yet allowed fast lookups.
- The functional partitioning of the FILM was performed. The Level1, Level 2 Sparse Chunk, Level 2 Dense Chunk and the Level 2 Very Dense Chunk blocks form the core of the lookup module. They are interfaced to the Map Table, Select Controller, Level 1 Pointers Memory and the Level 2 Memory.
- The signals for the individual blocks were defined and the FSMs for the controllers designed.
- The entire design was coded in VHDL, a hardware description language, and the individual blocks were simulated using **Modelsim EE/Plus 5.3a_p1**. The simulation results are shown.
- Synthesis and implementation of the design was performed using **Xilinx Foundation F2.1i, Build 3.1.162** targeting ***Xilinx VIRTEX XCV-1000-6-BG560*** device. The design and timing summary are shown.

4.2 Simulation Results

4.2.1 Level 1 Lookup Controller

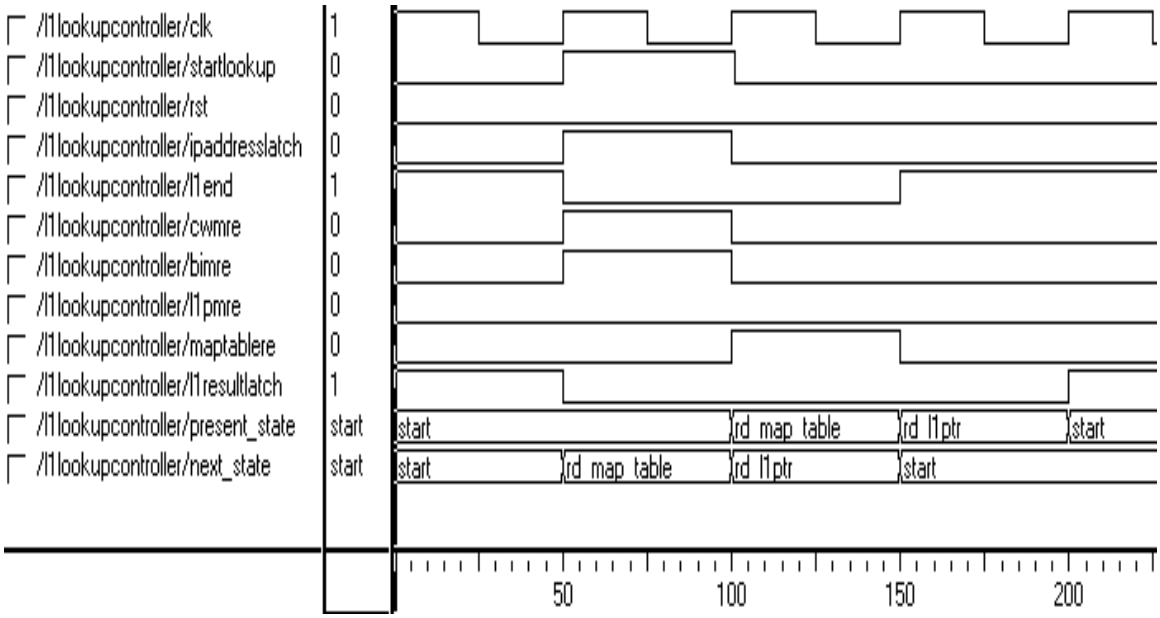


Figure 22: Simulation Waveforms for Level 1 Lookup Controller

The Level1 Lookup Controller is a 3-state FSM.

START: In the START state, the controller has finished the previous lookup and is awaiting the *StartLookup* signal to start a new lookup sequence. Hence, the *L1End* and *L1ResultLatch* signals are asserted. When the *StartLookup* signal arrives, the input IP address is latched by asserting the *IPAddressLatch* signal and the Code Word Memory and the Base Index Memory are simultaneously read by asserting the *CWMRE* and the *BIMRE* signals. The FSM then moves to the Read Map Table (RD_MAP_TABLE) state.

RD_MAP_TABLE: The Map Table is read by asserting the *MapTableRE* signal. The next state is the Read Level 1 Pointer (RD_L1_PTR) state.

RD_L1_PTR: *L1End* is asserted before the end of this last memory access. This is done to facilitate the Select Controller to transfer control smoothly from level 1 to level 2 without wasting one clock cycle. The Select Controller, in case the search ends at Level 1, asserts the actual *NextHopValid*, only at the end of the RD_L1_PTR state. After this state, the FSM returns to the START state.

4.2.2 Level 2 Sparse Lookup Controller

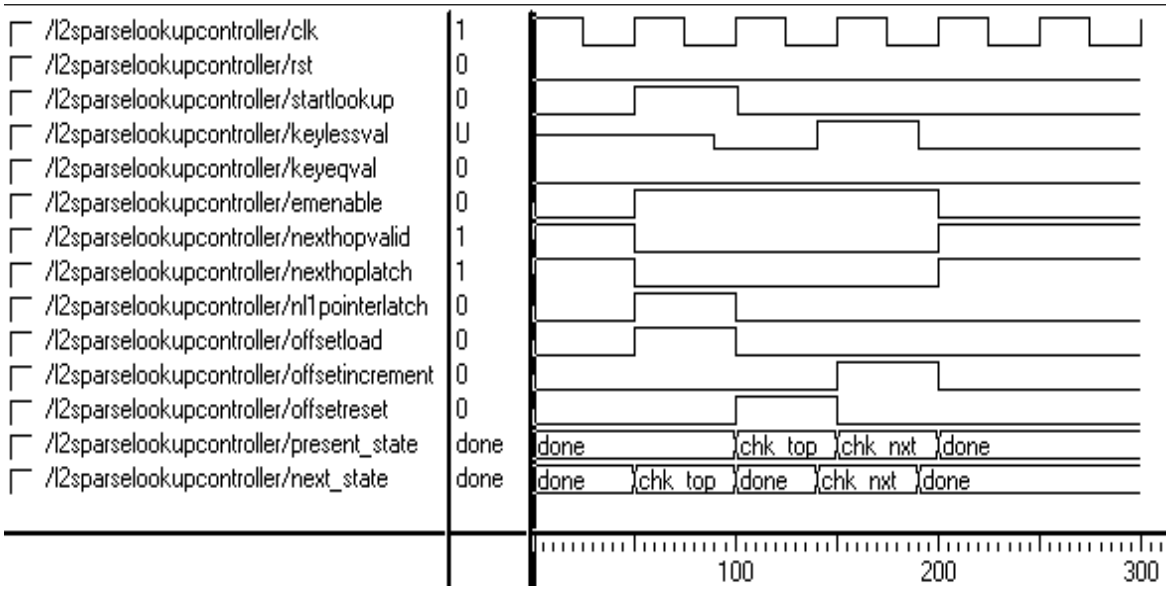


Figure 23: Simulation Waveforms for Level 2 Sparse Chunk Lookup Controller

The Level 2 Sparse Chunk Lookup Controller is a 3-state FSM.

DONE: The Level 2 sparse chunk controller is initially in the DONE state and is awaiting the *StartLookup* signal to begin another lookup sequence. The *NextHopLatch* signal is asserted to latch the Next Hop and the *NextHopValid* signal is asserted to indicate that lookup is finished. When the *StartLookup* signal arrives, the *OffsetLoad* signal is asserted to load the offset of the middle element into the Offset register, from L1Pointer. This address is used to access the middle element of the sparse chunk from the Level 2 Memory. The resulting head is then compared with the key. The result of the comparison arrives in the signals *KeyEqVal* and *KeyLessVal*. If Key equals Val, the middle element is the match, and the corresponding next hop, which is stored along with the 8-bit head as one 16-bit word is now the required next hop. The FSM, in this case, now returns to the DONE state. If the *KeyLessVal* signal is asserted after the comparison, the required head lies below the middle element and searching proceeds with the next element by moving to the CHK_NXT state. If *KeyEqVal* and *KeyLessVal*, both are deasserted, the key is greater than the middle-element and hence, searching begins from the top, FSM goes to the CHK_TOP state.

CHK_TOP: The Level 2 Memory is accessed to obtain the head and is compared with the Key. If KeyLessVal = ‘1’ and KeyEqVal = ‘0’, FSM goes to CHK_NXT to check the next element, otherwise the search has ended and it returns to the DONE state.

CHK_NXT: The Level 2 Memory is accessed to obtain the head and is compared with the Key. If KeyLessVal = ‘1’ and KeyEqVal = ‘0’, FSM returns to CHK_NXT to check the next element, otherwise the search has ended and it returns to the DONE state.

4.2.3 Level 2 Dense Chunk Controller

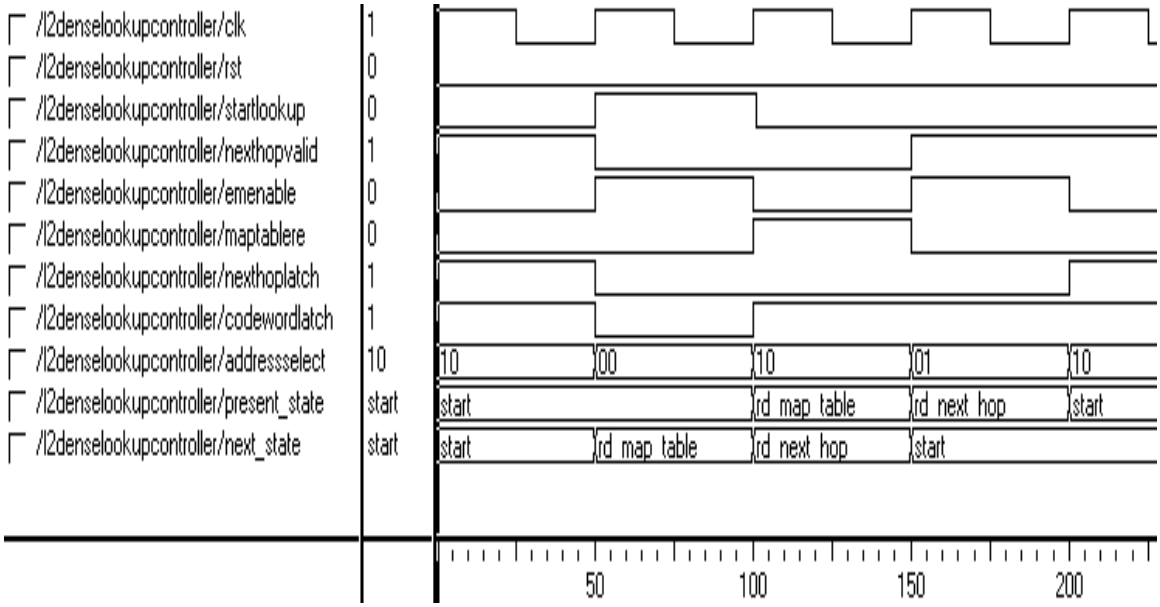


Figure 24: Simulation Waveforms for Level 2 Dense Chunk Controller

The Level 2 Dense Chunk Controller is a 3-state FSM.

START: The dense lookup controller has finished lookup and asserts the *NextHopLatch* and *NextHopValid* signals. *AddressSelect* tri-states the *EMAddress* lines as no memory access is to be performed here. When the *StartLookup* signal arrives, the Level 2 Memory is read to get the *CodeWord*. *AddressSelect* chooses the high four bits of the third octet of the latched IP address as *Offset* to be added to *L1Pointer* to obtain *EMAddress*.

RD_MAP_TABLE: The *MapTableRE* signal is asserted to obtain the *MapTableResult*. At the beginning of this state, *CodeWordLatch* is asserted and the *CodeWord* read in the previous state is latched on the positive edge of *CodeWordLatch*. *AddressSelect* again tri-states Offset as Level 2 Memory is not accessed in this state, instead the Map Table is to be read.

RD_NEXT_HOP: The Next Hop is read from the level 2 memory in the RD_NEXT_HOP state, and the Offset for the *EMAddress* is computed as shown in figure 17. Note that analogous to the L1End signal at Level 1, the *NextHopValid* signal is asserted one state earlier, i.e. at the beginning of the RD_NEXT_HOP state. This allows the final *NextHopValid* signal to be generated by the Select Controller without wasting a clock cycle after lookup is finished.

4.2.4 Level 2 Very Dense Chunk Controller

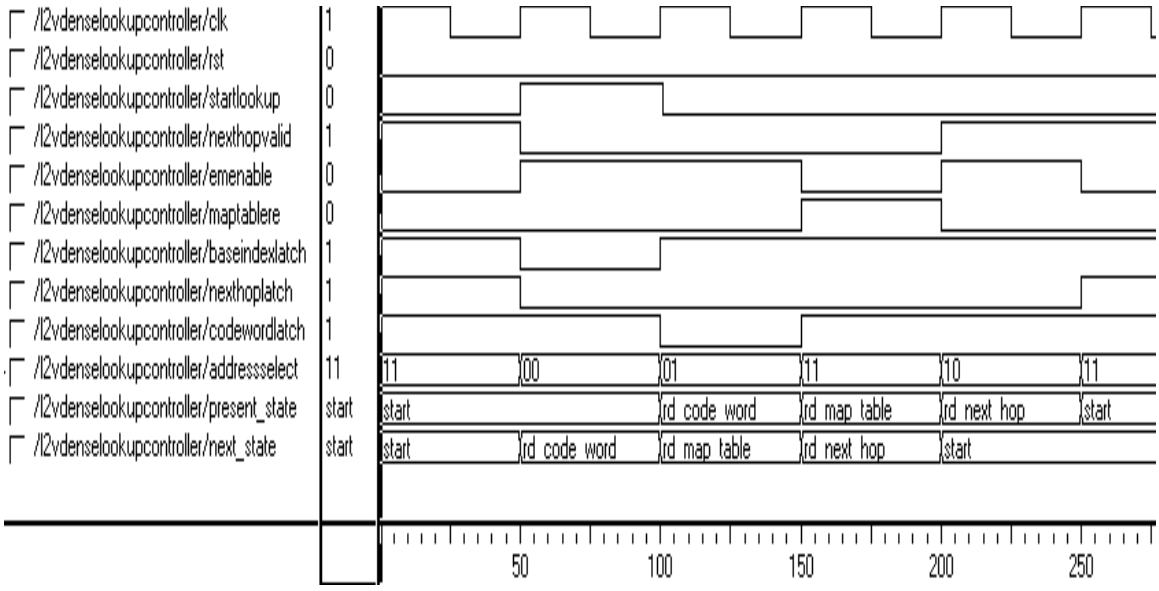


Figure 25: Simulation Waveforms for Level 2 Very Dense Chunk Controller

The Level 2 Very Dense Chunk Controller is a 4-state FSM.

START: The very dense chunk controller has finished lookup and asserts the *NextHopLatch* and *NextHopValid* signals. *AddressSelect* tri-states the *EMAddress* lines as no memory access is to be performed here. When the *StartLookup* signal arrives, the Level 2 Memory is read to get the *BaseIndex.* AddressSelect uses the high 2 bits of the third octet of the latched IP address to obtain the offset for BaseIndex.

RD_CODE_WORD: The controller asserts *EMEnable* signal to read the CodeWord using 4 + the high 4 bits of the third octet as Offset and also latches the BaseIndex.

RD_MAP_TABLE: The Map Table is read in the RD_MAP_TABLE state by asserting the MapTableRE signal.

RD_NEXT_HOP: The computed *EMAddress* is used to access the Next Hop in the RD_NEXT_HOP state and the *EMEnable* signal is asserted. As in the dense lookup controller, the *NextHopValid* is asserted one cycle early.

4.2.5 Select Controller

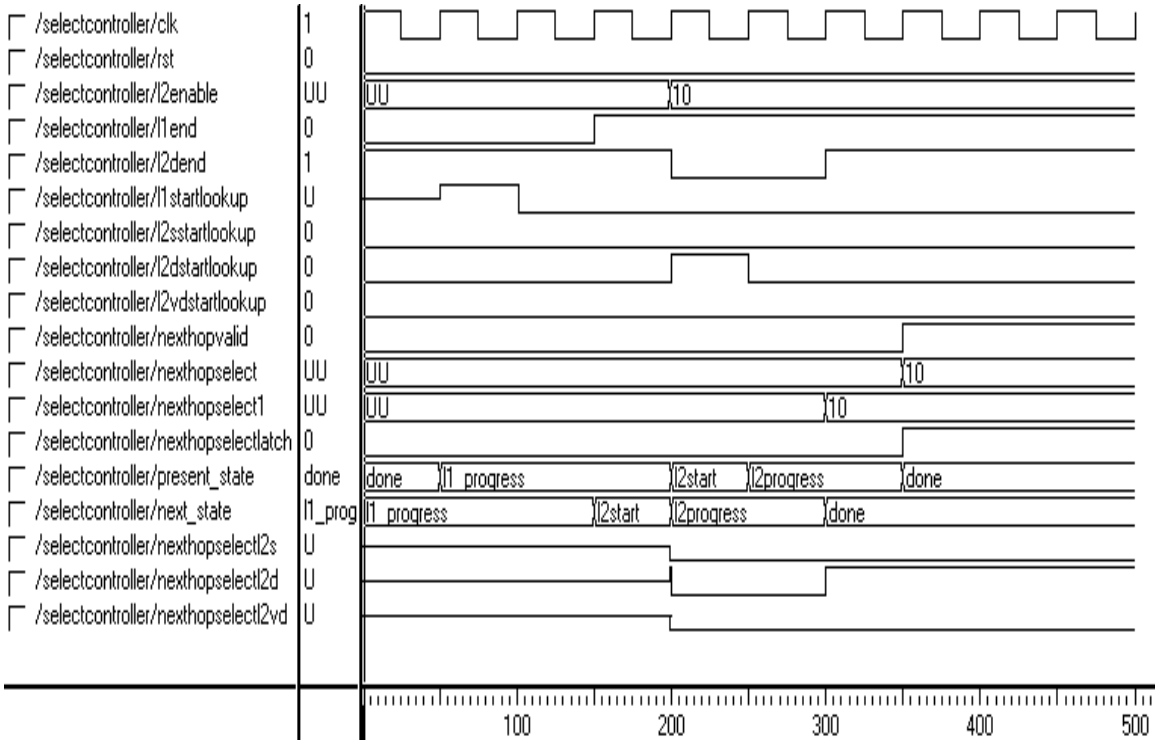


Figure 26: Simulation Waveforms for Select Controller

The Select Controller is a 4-state FSM.

DONE: The Select Controller awaits the StartLookup signal. When the StartLookup signal arrives, the level 1 lookup controller latches the input IP address and begins the lookup. The Select Controller moves to the L1_PROGRESS state. The Level 2 controllers are disabled in this state.

L1_PROGRESS: In the L1_PROGRESS state, the Select Controller checks if the L1End signal is asserted. If not, it remains in the L1_PROGRESS state. If the L1End signal is asserted, the level 1 lookup is complete and the high two bits of the L1Result are input to the Select Controller are L2Enable. If L2Enable is "00", the Level 1 Result is a next hop, search ends here and NextHopValid is asserted and NextHopSelect is made "00" to choose the Level 1 Next Hop. The FSM returns to DONE. Otherwise, the next state is L2START. The Level 2 controllers remain disabled in L1_PROGRESS state.

L2START: The search must now proceed to the second level. One of the three Level 2 controllers – Sparse, dense or Very dense is enabled by asserting the corresponding StartLookup signal to that controller based on the L2Enable signal. The search continues to the L2PROGRESS state.

L2PROGRESS: The controller monitors the NextHopValid signals from each of the Level 2 blocks, and if the enabled controller asserts its NextHopValid signal, it has completed lookup. The NextHopSelect now selects the NextHop from this L2 block and the controller asserts the FILM NextHopValid signal. The FSM returns to the DONE state.

4.3 Synthesis and Implementation Results

Synthesis and implementation of the design was performed using **Xilinx Foundation F2.1i, Build 3.1.162** targeting ***Xilinx VIRTEX XCV-1000-6-BG560*** device. The results are as follows:

4.3.1 Low Effort Option

For *Low Effort* and *Speed* optimization for Synthesis, and for *Place and Route Effort Level 2*, the following results were obtained:

Design Summary

Number of Slices:	214 out of 12,288	(1%)
Slice Flip-Flops:	139	
Slice Latches:	8	
4 input LUTs:	219 (3 used as a route-thru)	
Number of Slices containing unrelated logic:	0 out of 214	(0%)
Number of bonded IOBs:	103 out of 404	(25%)
Number of Tbufs:	584 out of 12,544	(4%)
Number of Block RAMs:	30 out of 32	(93%)
Number of GCLKs:	4 out of 4	(100%)
Number of GCLKIOBs:	1 out of 4	(25%)

Total equivalent gate count for design: 496,278
Additional JTAG gate count for IOBs: 4,992

Timing Summary

Minimum period	: 54.380ns (Maximum frequency: 18.389MHz)
Maximum combinational path delay	: 80.412ns
Maximum net delay	: 14.291ns

Though the SLICE count is low, a larger FPGA was chosen to accommodate on-chip memory of about 15 KB by using 30 out of the 32 available BLOCK RAMs. Hence, Area Optimization during synthesis was not attempted.

4.3.2 High Effort Option

For *High Effort* and *Speed* optimization for Synthesis, and for *Place and Route Effort Level 5*, the following results were obtained:

Design Summary

Number of Slices:	213 out of 12,288	(1%)
Slice Flip-Flops:	139	
Slice Latches:	8	
4 input LUTs:	215 (3 used as a route-thru)	
Number of Slices containing unrelated logic:	0 out of 214	(0%)
Number of bonded IOBs:	103 out of 404	(25%)
Number of Tbufs:	584 out of 12,544	(4%)
Number of Block RAMs:	30 out of 32	(93%)
Number of GCLKs:	4 out of 4	(100%)
Number of GCLKIOBs:	1 out of 4	(25%)

Total equivalent gate count for design: 496,254
Additional JTAG gate count for IOBs: 4,992

Timing Summary

Minimum period	: **42.767ns** (Maximum frequency: **23.383MHz**)
Maximum combinational path delay	: 67.042ns
Maximum net delay	: 14.745ns

4.4 Comparison of Results

The Lulea algorithm [5] implemented in the FILM takes a worst case lookup time of 8 memory accesses (Level 1 + Level 2 Sparse) and a best case of 3 memory accesses (Level 1 Only). Assuming that external memory can function at or above the FILM clock frequency (i.e 23.383 MHz), the worst case lookup time is now 8 clock cycles - about 340 ns and the best case is about 130 ns. As more searches are expected to finish in the first level itself, it is reasonable to take a worst case of 5 clocks or about 215 ns.

A comparison of the performance and memory requirements of various lookup algorithms is presented below. The Lulea algorithm on FILM is seen to have the smallest memory requirement and average lookup time.

Search Algorithm	**Avg. Lookup Time (ns)**	**Worst Case Lookup Time (ns)**	**Memory Requirement (KB)**
FILM using Lulea algorithm	**215**	**340**	**283**
6-way Search on Prefixes (Varghese et.al)	330	490	950
Patricia Trie	1500	2500	3262
Binary Search on Hash Tables	250	650	1500
Level Compressed Trie	1000	-	700
Expansion-Compression Method (Crescenzi et.al)	235	235	960

Table 2: Comparison of the lookup algorithms

4.4 Future Work

- The Fast IP Lookup Module can be implemented as an Application Specific Integrated Circuit (ASIC). This can give a higher performance and can use on-chip SRAM to improve memory access time.
- Integration of the lookup module with the Router I/O Unit, Universal Controller and the Packet Memory Unit to build a working router chip.
- As the FILM consists of two levels that run one after the other, a pipelining architecture could improve the speed of operation.

5. Conclusion

We have presented the work done in the design and implementation of 'A Fast IP Address Lookup Module (FILM) for a Router Chip'. The IP Address Lookup problem is one of the chief bottlenecks for wire-speed routing. The Lulea algorithm was used to store the router forwarding tables in a compact form, and at the same time allowing fast lookups. The FILM has a memory requirement of 283 Kbytes, a worst-case lookup time of 340 ns and a best case of 130 ns. This meets the requirement of wire-speed routing. The speed can be further improved by incorporating a pipelined architecture – the two level architecture of the FILM facilitates pipelining.

References

[1] K Sklower, “A tree-based routing table for Berkeley Unix”, presented at the 1991 Winter Usenix Conf., Dallas, TX

[2] G. Wright, W.R. Stevens: “TCP/IP illustrated, vol.2 -The implementation”, Addison Wesley Publishing Co., 1995

[3] P. Newman, G. Minshall, and L. Huston, “IP Switching and gigabit routers”, *IEEE Commun. Mag.*, vol. 35, pp. 64-69, Jan. 1997

[4] A. McAuley, P. Tsuchiya, and D. Wilson, “Fast Multilevel hierarchical routing table using content-addressable memory”, US Patent 034 444

[5] M. Degermark, A.Brodknik, S. Carlsson, S. Pink, “Small Forwarding Tables for Fast Routing Lookups”, *Proc. ACM SIGCOMM*, Sept. ‘97.

[6] Donald R Morrison. “PATRICIA | Practical Algorithm to Retrieve Information Coded In Alfanumeric”, *Journal of the ACM*, 15(4): 514-534, Oct ‘68.

[7] Michigan University and Merit Network. “Internet Performance Management and Analysis (IPMA) Project”. Details available at http://nic.merit.edu/~ipma/

[8] Cisco’s Catalyst 8500 CSR - Campus Switch: Product Overview, available online: http://www.cisco.com/univercd/cc/td/doc/pcat/ca8500c.htm

[9] V. Srinivasan and G. Varghese, “Fast IP Lookups using controlled prefix expansion”, *ACM TOCS*, vol. 17, pp. 1-40, Feb. 1999

[10] B. M. Ravishankar and G.N.Srihari, “Comparison of Fast IP Lookup techniques”, *Proc. Ninth Annual IEEE Conference*, Bangalore, India, Nov. 2000