CS6040
Router Architectures and Algorithms

Prof. Krishna M. Sivalingam, CSE Dept., IIT Madras

IP Address Lookup Algorithms
Oct. 9, 2024 (Wed.); Oct. 18, 2024 (Fri.) [Quiz II on Oct. 11; No Class on Oct. 16]

Overview

1. IP Addresses
2. Longest Prefix Matching
3. Trie-based software approaches
4. Helper Techniques to avoid Prefix Overlaps
5. Multi-bit Tries based LPM
6. Search based on Lengths and Values
7. TCAM-based hardware approaches

IP Addresses

Class-based IP Addresses

- Five IPv4 address classes
  - Unicast: Class A, B, C; Multicast: D, Reserved: E
- Unicast addresses:

Type	Byte 1	Byte 2	Byte 3	Byte 4
Class A	$0 +$ NetID (7 bits)	Host ID (24 bits)		
Class B	$10 +$ NetID (14 bits)		Host ID (16 bits)	
Class C	$110 +$ NetID (21 bits)			Host ID (8 bits)

- Incoming packet's destination address (DA) is extracted by router
- Based on DA's first three bits, packet's class is determined and network identifier (NetId) is obtained
  - NetId is 8 or 16 or 24 bits long
- NetId is looked up in the forwarding table to determine output port

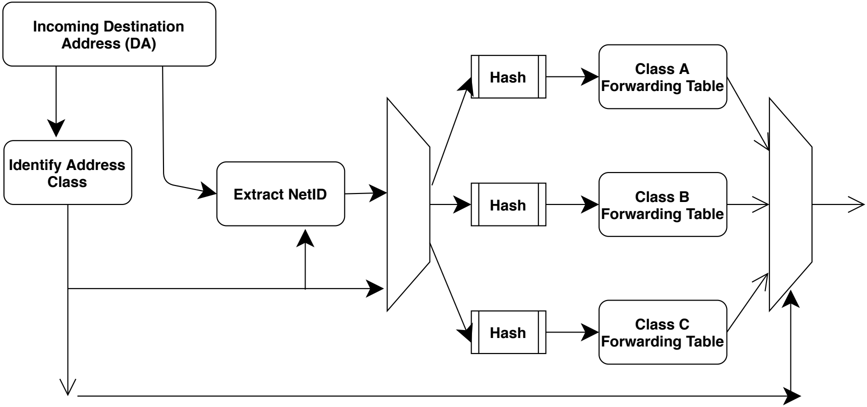
NetID Lookup in Class-based Addressing

- Each forwarding table contains a set of network prefixes and corresponding outgoing interface numbers
- Three separate forwarding tables are maintained:
  - one each for Classes A, B and C
- Input NetID is hashed and searched for in its Class' table
  - Exact network prefix match is found (if present) in forwarding table
  - Time complexity: $O(1)$

Forwarding Table (entries for all Classes shown)

Prefix	NetMask	Output Interface for next hop
11.0.0.0	255.0.0.0	2
72.0.0.0	255.0.0.0	4
128.206.0.0	255.255.0.0	3
161.13.0.0	255.255.0.0	7
195.114.73.0	255.255.255.0	1
212.97.63.0	255.255.255.0	8

NetID Lookup in Class-based Addressing, Contd.



Adapted from MR book

Problems with Class-based Addressing

- Exhaustion of IP Address Space
  - Insufficient Network Identifiers
  - Inefficient use of allocated addresses especially within Class A (4M hosts) and Class B (64K hosts) if not all addresses are used
- Exponential Growth of Routing Tables
  - Route information stored in Core IP routers grows proportional to the number of networks
  - Larger routing tables lead to increased address lookup times and higher memory requirements

Class-less Inter-domain Routing (CIDR) based Addressing

- Network identifiers can be from 8 to 32 bits
  - Network identifier length, in bits, is included in address specification
- Class A networks can be broken down into smaller networks
  - For example, 11.0.0.0 can be split into four networks: 11.0.0.0/10; 11.64.0.0/10; 11.128.0.0/10; 11.192.0.0/10
  - For example, 21.0.0.0 can be split into 16 networks: 21.0.0.0/12; 21.16.0.0/12; 21.32.0.0/12; . . . ; 21.240.0.0/12
- Class B networks can be broken down into smaller networks
  - For example, 129.74.0.0 can be split into eight networks: 129.74.0.0/19; 129.74.32.0/19; 129.74.64.0/19; . . . ; 129.74.224.0/19
- Increases availability of network identifiers, releasing any unused network addresses
- Provides address aggregation in the forwarding table, reducing the number of table entries
  - Eight entries from 192.168.40.0/24 to 192.168.47.0/24 can be merged into one entry 192.168.40.0/21

Breakup of Class A and Class B

Splitting 11.0.0.0/8 into four networks

Network	Start Address	End Address	Start	End
11.0.0.0/10	00001011.00000000.*.*	00001011.00111111.*.*	11.0.*.*	11.63.*.*
11.64.0.0/10	00001011.01000000.*.*	00001011.01111111.*.*	11.64.*.*	11.127.*.*
11.128.0.0/10	00001011.10000000.*.*	00001011.10111111.*.*	11.128.*.*	11.191.*.*
11.192.0.0/10	00001011.11000000.*.*	00001011.11111111.*.*	11.192.*.*	11.255.*.*

Splitting 129.74.0.0/16 into eight networks

Network	Start Address	End Address	Start	End
129.74.0.0/19	129.74.00000000.*.*	129.74.00011111.*.*	129.74.0.*	129.74.31.*
129.74.32.0/19	129.74.00100000.*.*	129.74.00111111.*.*	129.74.32.*	129.74.63.*
129.74.64.0/19	129.74.01000000.*.*	129.74.01011111.*.*	129.74.64.*	129.74.95.*
...	...	...	...	...
129.74.224.0/19	129.74.11100000.*.*	129.74.11111111.*.*	129.74.224.*	129.74.255.*

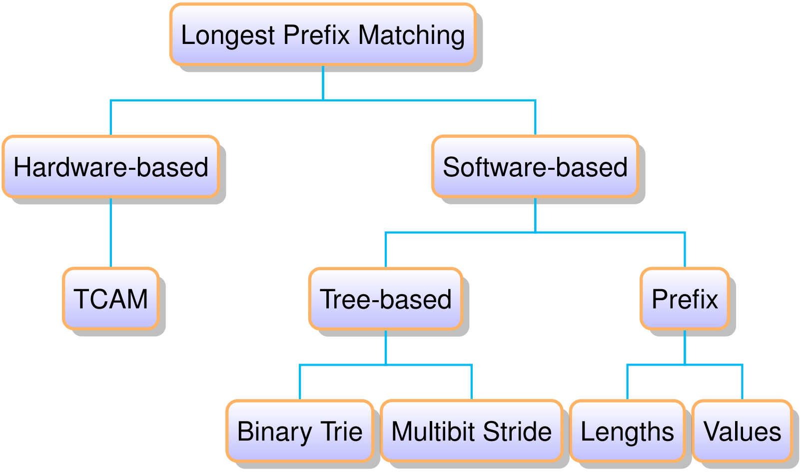
Longest Prefix Matching

Longest Prefix Matching

- Exception in Address Aggregation:
  - Consider a router R1 that connects to routers R2 (100.2.0.0/24); R3 (100.2.1.0/24); R4 (100.2.2.0/24); R5 (100.2.3.0/24)
    - Only last two bits of netid are different
  - R8 is connected to R1, with separate FT entry for each network
  - R8 can aggregate above 4 network identifiers to 100.2.0.0/22 that points to R1; reduces FT size
  - Assume that the customer with network 100.2.2.0/24 moves to another router R6, also connected to R8
  - R8 can break up 100.2.0.0/22 entry into separate network entries (OR) retain this entry and add an exception for 100.2.2.0/24
  - 100.2.2.13 will match both prefixes; need to select the longest prefix
- Input IP address can match multiple prefixes in forwarding table
- Objective: Find the longest matching prefix and use the corresponding forwarding table entry

LPM Algorithm: Design Requirements

- **Fast Lookup Speed:** Processing time is limited
  - Time available to process at 40-byte packet at 40 Gbps is 8 ns
- **Minimal Memory Usage:** Use of SRAM, DRAM and TCAM
  - Tradeoff between cost, maximum size, access latency and power
- **Power and Cost:** Should be as low as possible
- **Scalability:** Support Large forwarding tables
  - IPv4 forwarding tables in core routers can have 1M entries or higher
  - IPv6 has 64-bit network addresses!
- **Updatability:** Accomodate (frequent) routing and forwarding table updates in a fast and efficient manner


Longest Prefix Matching
Hardware-based
Software-based
TCAM
Tree-based
Prefix
Binary Trie
Multibit Stride
Lengths
Values

Trie-based software approaches

Binary Trie algorithm

- Based on a binary tree representation
- Called as “Trie” since it is used for “Retrieval” purposes
- Tree is constructed by processing one prefix at a time
- Child nodes are added as needed; Prefix nodes are labeled
- Each prefix is represented by an inner node or leaf node in the Trie

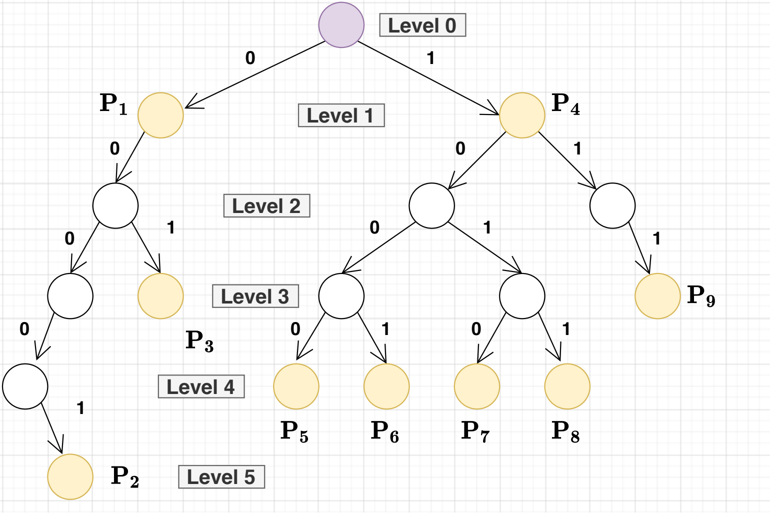
Example Prefix Table

- W: Maximum prefix length (or width)
- N: Number of prefix entries in table
- If a packet's address does not match any prefix in the table, the packet will be sent on a default output interface

Forwarding Table

Prefix Label	**Prefix ($W = 6$)**	**Next Hop**
P_1	$0*$	2
P_2	$00001*$	3
P_3	$001*$	5
P_4	$1*$	6
P_5	$1000*$	7
P_6	$1001*$	9
P_7	$1010*$	3
P_8	$1011*$	2
P_9	$111*$	6

Binary Trie Example



Binary Trie: Search for Longest Prefix

- Start traversing tree at root node (Level 0)
- Start from the Most Significant Bit (MSB) of input address
- Traverse level by level based on the corresponding bit until there is no matching bit
- Record all prefix nodes encountered along the path from root
- Search ends when no matching sub-tree is found or if a leaf node is reached
- Select the prefix node with the longest path from the root
  - If search does not encounter any prefix node along path from root, use default table entry

Binary Trie: Search, Examples

- Input Address, A = 010010 ($W = 6$): Traversal ends at Level 1; Matching Prefix: $\{P_1\}$; LMP = P_1
- Input Address, A = 000011: Traversal ends at Level 5; Matching Prefixes: $\{P_1, P_2\}$; LMP = P_2
- Input Address, A = 110011: Traversal ends at Level 2; Matching Prefixes: $\{P_4\}$; LMP = P_4
- Input Address, A = 100110: Traversal ends at Level 4; Matching Prefixes: $\{P_4, P_6\}$; LMP = P_6

Binary Trie: Complexity

- Number of nodes in tree: $O(2^W)$
- Search Time, i.e. Number of Memory Accesses: $O(W)$
- Time to create a new Trie from scratch: $O(NW)$
- Time to access
- Average Number of Memory Accesses depends on the distribution of addresses
  - Let p_i be the probability of an address matching prefix P_i
  - Let a_i be number of memory accesses for P_i (at Level i)
  - Average number of accesses = $\sum_{i=1}^{N} p_i a_i$
- Observations:
  - There are several long sequences of one-child nodes: can they be removed to result in a compressed tree?

Binary Trie: Updating

- Inserting new prefix
  - Traverse tree based on new prefix and add prefix node where search ends
  - For example, adding $P_{10} = 110*$ adds it as left sibling of P_9
  - For example, adding $P_{11} = 0110*$ creates a new right sub-tree of length 3, under node P_1
  - Time complexity: $O(W)$
- Deleting prefix
  - Traverse tree based on to-be-deleted prefix and delete prefix node where search ends; if it is an inner node, mark it is a non-prefix node
  - For example, deleting P_5 removes the corresponding leaf node
  - For example, deleting P_4 marks the inner node as a non-prefix node
  - Time complexity: $O(W)$
- Implementation: Each tree node contains left and right child pointers; prefix number, if applicable
  - Prefix number can be directly looked up in forwarding table

Binary Trie with Path Compression

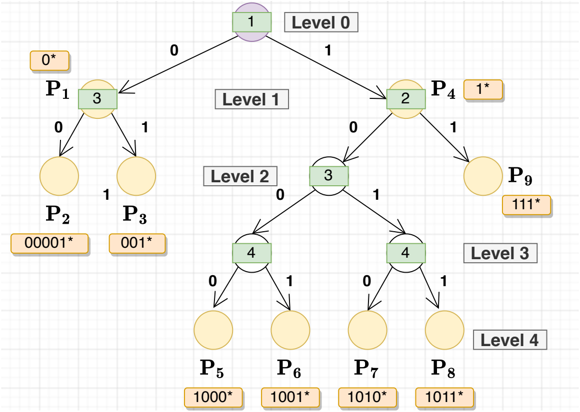
Construction of Compressed Trie

- Replace sequences of single-child nodes with a single node
- A prefix node stores the complete prefix values(s) of that node and the specific bit position to be compared at that node
  - Possible to have multiple prefixes at a node
- Reduces tree levels and average number of memory accesses
  - Degree of compression depends on set of prefixes in table

Search in Compressed Trie

- At each node, starting from root, compare bit-position stored in node to input address' corresp. bit
- If prefix node is reached, compare input address to list of prefixes and record any match(es) found
- Search ends when no matching sub-tree is found or if leaf node is reached

Binary Trie with Path Compression, Example



- From P_1, bit 3 is compared
- P_2 and P_3 moved to left and right of P_1: 2 bits
- P_9 is moved to right of P_4: 2 bits compared vs. 3
- More than one possible compressed trie may exist
- Worst-case search time complexity: $O(W)$, same as basic trie
- Update of compressed trie is more complex than basic trie

Binary Trie with Path Compression, Example, Contd.

- Traverse the tree from root node
- For each prefix node that is encountered along the path, compare the input address to the stored prefix values
  - If there is match, the prefix is recorded and the search continues
- When traversal ends, the list of matching prefixes is considered and the longest prefix is selected
- For example: 001100 will match P_1, and also match P_3
- For example: 011000 will match P_1, but will not match P_3
- For example: 010000 will match P_1, but will not match P_2
- For example: 110110 will match P_4, but will not match P_9
- Deleting P_3: If P_3 is deleted, then P_2 will be the only child of P_1; we can merge and store both P_1 and P_2 in same node
  - One less memory access; one extra prefix comparison (less expensive than access)
- We can also merge and store both P_4 and P_9 in same node

Helper Techniques to avoid Prefix Overlaps

Helper Technique 1: Prefix expansion

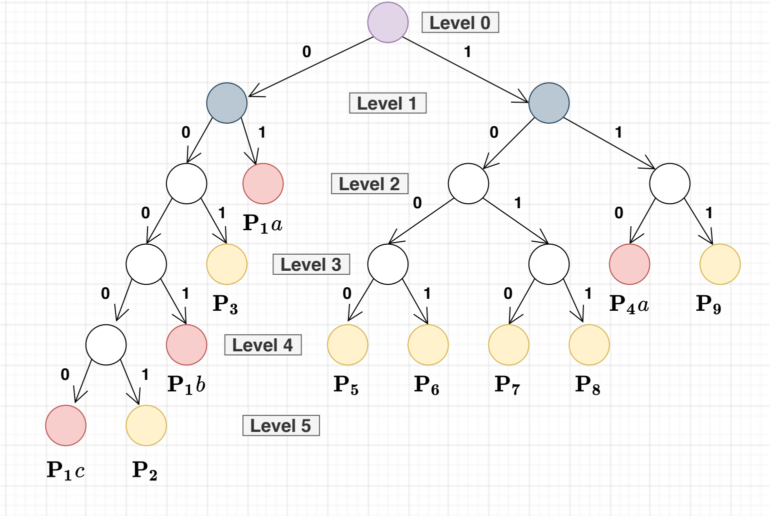
- For a given prefix in the forwarding table of length n bits, expand it to $n+k, 1 \leq k \leq W-n$ bits
  - $0*$ expanded to $\{000*, 001*, 010*, 011*\}$, for $n=1, k=2$
- Collision with existing longer prefixes is avoided
- A given set of prefixes of different lengths can be transformed into another equivalent set of fewer different lengths
  - In table below, all prefixes are of length 3 or 5, after expansion
- Increasing number of prefixes requires more table storage

Prefix Label	Prefix ($W=6$)	Expanded Prefixes
P_1	$0*$	$000*, 010*, 011*$ [$001*$ not included due to P_3]
P_2	$00001*$	$00001*$
P_3	$001*$	$001*$
P_4	$1*$	$100*, 101*, 110*$ [$111*$ not included due to P_9]
P_5	$1000*$	$10000*, 10001*$
P_6	$1001*$	$10010*, 10011*$
P_7	$1010*$	$10100*, 10101*$
P_8	$1011*$	$10110*, 10111*$
P_9	$111*$	$111*$

Helper Technique 2: Leaf Pushing / Disjoint Prefixes

- Eliminate prefix overlaps by transforming original set of prefixes into a set of disjoint prefixes
- All prefix nodes are present only at leaf nodes
  - No inner node will have a prefix
- Add leaf nodes to nodes that have only one child, with prefix information of leaf node inherited from its closest ancestor
- Exact matching can be done using this set, since overlapping prefixes do not exist
- Cost: Increasing number of prefixes increases storage needs

Leaf Pushing, Example



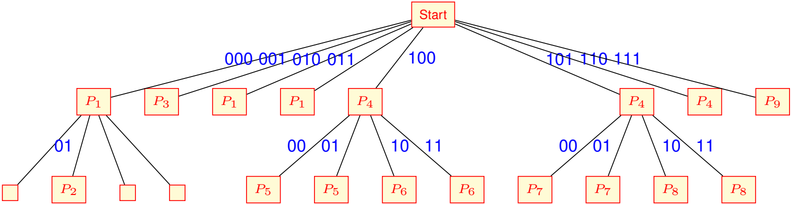
- Four leaf prefix nodes are newly added to replace inner prefix nodes (shown in gray)

Multi-bit Tries based LPM

Multi-bit Trie

- Trees are no longer binary, but n-ary
- Prefix expansion and leaf pushing can be used to construct multi-bit stride tries
- Branching decision at each node of trie is based on a set of bits
- The number of bits used for decision can be different at different trie nodes
- Fixed-Stride Multibit Trie: Nodes at the same level use the same number of bits (stride size) for traversal decision
- Variable-Stride Multibit Trie: Nodes at given level use different number of bits for traversal decision

Fixed-Stride Multibit Trie, Example

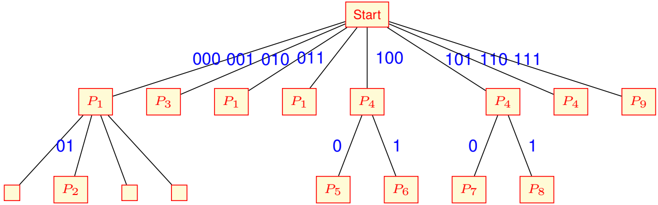


Prefix Label	Prefix ($W = 6$)	Expanded Prefixes
P_1	$0*$	$000*, 010*, 011*$
P_2	$00001*$	$00001*$
P_3	$001*$	$001*$
P_4	$1*$	$100*, 101*, 110*$ [$111*$ not included due to P_9]
P_5	$1000*$	$10000*, 10001*$
P_6	$1001*$	$10010*, 10011*$
P_7	$1010*$	$10100*, 10101*$
P_8	$1011*$	$10110*, 10111*$
P_9	$111*$	$111*$

Fixed-Stride Multibit Trie, Operations

- Comparison to Binary Trie:
  - Number of nodes in BT and FSMB: 18 and 18, respectively
  - Number of memory accesses for 00001*: 5 and 2, respectively
- Search: At each tree level, multiple bits are used to make the branching decision
  - Lower number of memory accesses compared to Binary Trie
- Inserting new prefix: Similar to Binary Trie, but might require prefix expansion and hence more complex; See Example 15.7 in book
- Deleting prefix: Similar to Binary Trie, but more complex
- Tradeoff between tree height/depth (i.e. the number of memory accesses) and the storage requirements, where more nodes may be created
  - Multiple ways to construct a Multi-bit Trie from a Binary Trie – depends on prefix set

Variable-Stride Multibit Trie, Example



Prefix Label	Prefix ($W = 6$)	Expanded Prefixes
P_1	$0*$	$000*, 010*, 011*$
P_2	$00001*$	$00001*$
P_3	$001*$	$001*$
P_4	$1*$	$100*, 101*, 110*$ [$111*$ not included due to P_9]
P_5	$1000*$	$10000*, 10001*$
P_6	$1001*$	$10010*, 10011*$
P_7	$1010*$	$10100*, 10101*$
P_8	$1011*$	$10110*, 10111*$
P_9	$111*$	$111*$

Search based on Lengths and Values

Search based on Prefix Lengths

- Let $\mathcal{L}$ denote the set of unique lengths in the prefix tree
  - For example, $\mathcal{L} = \{3, 5, 7, 8\}$ and $|\mathcal{L}| = 4$ for $W = 8$
- For each unique length value in $\mathcal{L}$, create a list of all prefixes of this length
- Linear Search, given an input address:
  - Search the prefix lists in decreasing order of length by extracting the relevant number of bits for each length value in $\mathcal{L}$
    - Report the first match found (if any) as LPM
  - Each list can be searched using a Hash Table
  - Worst-case time complexity: $O(W)$
    - Parallel search across multiple lists can reduce lookup time
- Binary Search on Prefix Lengths: Read Sec 15.7.2 (Optional)

Search based on Prefix Lengths, Example 1

Length	**List of prefixes**	Values
1	$\{0, 1\}$	$\{0, 1\}$
3	$\{001, 111\}$	$\{1, 7\}$
4	$\{1000, 1001, 1010, 1011\}$	$\{8, 9, 10, 11\}$
5	$\{00001\}$	$\{1\}$

Search Example

Input address – 101000:

- Check 10100 against $\{00001\}$: No match
- Check 1010 against $\{1000, 1001, 1010, 1011\}$: Match ($P_7$) found

Search based on Prefix Lengths, Example 2

Refer Table in Prefix expansion, with only prefixes of lengths 3 and 5

Length	**List of prefixes**	Values
3	{P1: 000, P3: 001, P1: 010, P1: 011, P4: 100, P4: 101, P4: 110, P9: 111}	Fill In
5	{P2: 00001, P5: 10000, P5: 10001, P6: 10010, P6: 10011, P7: 10100, P7: 10101, P8: 10110, P8: 10111 }	Fill In

Search based on Prefix Value Ranges

- Each prefix is converted into a range of values, i.e. an interval
  - For $W = 4$ and prefix of $10*$, the range is: $[8, 11]$
- The forwarding table is represented as a set of intervals
  - if prefixes overlap, their intervals also overlap
- Given an input address, convert it to a numerical value
  - (e.g. $1010 = 10$ will lie in the range $[8, 11]$)
- Find the set of intervals that the input value is enclosed in
  - select the prefix corresp. to closest enclosing interval
- To make the problem more manageable, the set of intervals is made non-overlapping using prefix expansion and leaf-pushing
  - Interval Tree can be used to identify the enclosing range for a given input address
  - This is formally called Enclosing Interval Searching Problem

Search based on Prefix Values, Example 1

Overlapping Intervals

Prefix Label	Prefix ($W = 6$)	Interval
P_1	$0*$	$[0 - 31]$
P_2	$00001*$	$[2 - 3]$
P_3	$001*$	$[8 - 15]$
P_4	$1*$	$[32 - 63]$
P_5	$1000*$	$[32 - 35]$
P_6	$1001*$	$[36 - 39]$
P_7	$1010*$	$[40 - 43]$
P_8	$1011*$	$[44 - 47]$
P_9	$111*$	$[56 - 63]$

Example Search

- Input: 010010 = 18; Enclosing Interval(s): $\{[0 - 31]\} \implies P_1$
- Input: 000011 = 3; Enclosing Interval(s): $\{[0 - 31], [2 - 3]\} \implies P_2$
- Input: 110011 = 51; Enclosing Interval(s): $\{[32 - 63]\} \implies P_4$
- Input: 100110 = 38; Enclosing Interval(s): $\{[32 - 63], [36 - 39]\} \implies P_6$

Search based on Prefix Values, Example 2

Non-Overlapping Intervals

Prefix Label	**Prefix ($W = 6$)**	**Interval**
P_1a	$00000*$	$[0-1]$
P_2	$00001*$	$[2-3]$
P_1b	$0001*$	$[4-7]$
P_3	$001*$	$[8-15]$
P_1c	$01*$	$[16-31]$
P_5	$1000*$	$[32-35]$
P_6	$1001*$	$[36-39]$
P_7	$1010*$	$[40-43]$
P_8	$1011*$	$[44-47]$
P_4a	$110*$	$[48-55]$
P_9	$111*$	$[56-63]$

Example Search

- Input: 010010 = 18; Enclosing Interval(s): $\{[16-31]\} \implies P_1$
- Input: 000011 = 3; Enclosing Interval(s): $\{[2-3]\} \implies P_2$
- Input: 110011 = 51; Enclosing Interval(s): $\{[48-55]\} \implies P_4$
- Input: 100110 = 38; Enclosing Interval(s): $\{[36-39]\} \implies P_6$

TCAM-based hardware approaches

Content Associative Memory (CAM)

- Given: a table of **key, value** pairs
- Each "bit" is represented using 2 states: 0, 1
- Given: an input search value, it is compared in parallel against all keys in the table
  - Set of XNOR gates to compare input and each key, bit-by-bit
- System outputs matched value (if found) or none (if no match)
- Lookup time: $O(1)$
- Circuitry requirements is high – higher cost and hence limited CAM capacity is usually available
- Used for cache memory and other applications

Key	**Value**
00011	13
01000	2
11010	4
10001	8

Ternary Content Associative Memory (TCAM)

- Each "bit" is represented using 3 states: 0, 1, X (dont-care)
- Given: a table of **key, value** pairs
- Given: an input search item, it is compared in parallel against all keys in the table
  - Set of modified XNOR gates to compare input and each key, bit-by-bit
  - If bit i in a key is X, then both 0 and 1 in bit i of input search item will match
  - Lookup time: $O(1)$

CAM: Comparison Operation

Binary CAM, Example 1

	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Input	0	1	1	1	0	0	1	0
Key	0	1	1	1	0	0	1	0
	Bit-wise XNOR ($\oplus$)							
Interim Output	1	1	1	1	1	1	1	1
	AND ($\wedge$)							
Final Value	**1**							

Binary CAM, Example 2

	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Input	0	1	0	1	0	1	1	0
Key	0	1	1	1	0	0	1	0
	Bit-wise XNOR ($\oplus$)							
Interim Output	1	1	0	1	1	0	1	1
	AND ($\wedge$)							
Final Value	**0**							

TCAM: Comparison Operation

Ternary CAM, Example 1

	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Input	0	1	1	1	0	0	1	0
Key	0	1	1	1	0	X	X	X
	Bit-wise XNOR ($\oplus$)							
Interim Output	1	1	1	1	1	1	1	1
	AND ($\wedge$)							
Final Value	**1**							

Ternary CAM, Example 2

	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8
Input	0	1	0	1	0	1	1	0
Key	0	1	1	1	0	X	X	X
	Bit-wise XNOR ($\oplus$)							
Interim Output	1	1	0	1	1	1	1	1
	AND ($\wedge$)							
Final Value	**0**							

Prefixes stored using TCAM, Example

Prefixes, arranged in order of decreasing length

$W = 6$		TCAM Contents	
Label	**Prefix**	**Value**	**Bitmask**
P_2	$00001*$	00001X	111110
P_5	$1000*$	1000XX	111100
P_6	$1001*$	1001XX	111100
P_7	$1010*$	1010XX	111100
P_8	$1011*$	1011XX	111100
P_3	$001*$	001XXX	111000
P_9	$111*$	111XXX	111000
P_1	$0*$	0XXXXX	100000
P_4	$1*$	1XXXXX	100000

TCAM Contents

- Each prefix stored as a *value* and as a *bitmask*
- For a prefix of length Y bits:
  - Most significant Y bits of *value* are from prefix
  - Remaining bits are set to X (dont-care)
  - Most significant Y bits of *bitmask* are set to 1
  - Remaining bits are set to 0
- For input address, *netmask* applied and compared against *value*, in parallel
- Priority encoder applied to select LMP

LMP using Prefix, Example

- Input: 000011 (Showing only matching prefixes operation)
  - P_2 : AND[(000011 & 111110) $\oplus$ 00001X] $\Longrightarrow$ AND[000010 $\oplus$ 00001X] = 1
  - P_1 : AND [(000011 & 100000) $\oplus$ 0XXXXX] $\Longrightarrow$ AND[(000000) $\oplus$ 0XXXXX] = 1
  - Priority encoder selects higher index entry in table and P_2 is match
- Input: 110000 (Showing only one non-matching prefix operation)
  - P_5 : AND [(110000 & 111100) $\oplus$ 1000XX] $\Longrightarrow$ AND[(110000) $\oplus$ 1000XX] = 0

TCAM, Disadvantages

- High power consumption, due to the circuitry for parallel compare
- Lower density due to higher number of transistors per bit
- Limited memory capacity (limits size of forwarding tables)
- Higher cost

Techniques to reduce power and other constraints are being actively pursued in research studies

End of Packet Lookup Algorithms