XPliant Family of Programmable Multilayer Switches

Software Theory of Operation

CNX-TOO-V3.2.4P

September 2016

PUBLISHED BY
Cavium, Inc.
2315 North First Street
San Jose, CA 95131-1010
Phone: 408-943-7100
Fax: 408-577-1992
Email: sales@cavium.com
Web: http://www.cavium.com

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

XPliant is a family of integrated, multilayer, fully programmable Ethernet switches based on the XPA™ line of chips. XPA (XPliant Packet Architecture) allows for the programming of switch packet processing elements. The software-definable implementations enable tailoring the switch forwarding pipeline and associated resources to meet various requirements in different places-in-network (PIN).

XPA enables customers to express switch “personalities”, or network profiles, enabling creation of flexible packet processing pipelines. The network profiles are typically configured into the switch at start-up. The network profiles define the forwarding pipeline, its functional behavior, and associated on-chip memory resources partitioning and entry formatting. The forwarding tables can be modified at runtime.

The XPliant Pipeline Compiler (XPC) generates the network profiles. For more details on the XPC, see the *XPliant Software Programmer and Configuration Guide*.

This document describes the XPliant Switches Theory of Operation (TOO). TOO contains a per-profile description of the supported functionality and lists the formats of the associated tables. Software APIs to be used in conjunction with the TOO are listed in the API Guide. The *XPliant Software Programmer and Configuration Guide* elaborates on the XDK components and focuses on recommendations on how to use the available APIs correctly and efficiently in order to facilitate software development and debugging.

The contents of this document are organized into the following sections:

- Chapter 1, Introduction (this chapter)

  Overview of document contents and revision history.

- Chapter 2, XPA Fundamentals

  An introduction to the XPliant ASICs architecture and the software-defined platform.

- Chapter 3, XPliant Software-Defined Forwarding Pipeline

  Overview of the pipeline elements and detailed descriptions of each pipeline feature.

- Appendices

  The document includes a set of appendices that provide reference material, including data structures and APIs:

  - Appendix A, Reason Codes
  - Appendix B, Generic Data Structures
  - Appendix C, Generic Packet Commands

- Appendix D, XP Header
- Appendix E, APIs

1.1 Conventions

The following conventions are used in this document:

Table 1–1 Conventions

Convention	Description
`Monospace font`	Commands, information the system displays, and file paths/ names appear in `monospace font`.
`Italic monospace font`	Arguments for which the user provides a value are in *`italic monospace font`*.
`Bold monospace font`	Commands that the user must enter exactly are in **`bold monospace font`**.
{} Braces	{} are used to enclose a list of pipe-delimited values from which the user must include one; for example {1|2|3}.
Bold font	Screen selections are in bold font; for example, "Select the page **Type Based Boot Priority**".

1.2 Revision History

The revision history is shown in Table 1–2.

Table 1–2 Revision History

Revision	Date	Remark
3.0	May 2015	First release.
3.1	July 2015	Added SAI, ECMP Group, ERSPAN.
3.2	February 2016	Added L3 subinterfaces and MPLS-over-GRE support; Unicast Reverse Path Forwarding (uRPF); QoS Policing and Shaping.
3.2.3	June 2016	Added CoPP.
3.2.4	September 2016	Updated copyright page.

1.3 Related Documentation

This document should be used in conjunction with the documents listed in Table 1–3: *Publications*. This document may contain information that was not previously published.

Table 1–3 Publications

Publication	Document Number
XPliant Functional Specification	
XPliant Software Programmer and Configuration Guide	CNX-PG-V3.2.3P
XPliant API Guide	CNX-API-V3.2P.html (located in XDK doc/ folder)

Chapter 2

XPA Fundamentals

This chapter provides an introduction to the XPliant Packet Architecture (XPA) and the fundamental primitives used to build the forwarding pipeline.

For more details about the XPA, refer to the Functional Specifications document.

2.1 XPliant Packet Architecture (XPA)

XPA separates and abstracts the Platform data path and the forwarding path, allowing simplification of management through software.

The *data path* incorporates blocks that operate with packet incoming and outgoing data, such as SerDes, DMAs, packet memory, etc., marked in gray in the following figure.

The *forwarding path* incorporates blocks that process the packet headers and apply forwarding policies. These tasks are performed by the *Software-Defined Engines* (SDEs) in the XPA Pipeline, marked in blue in the following figure.

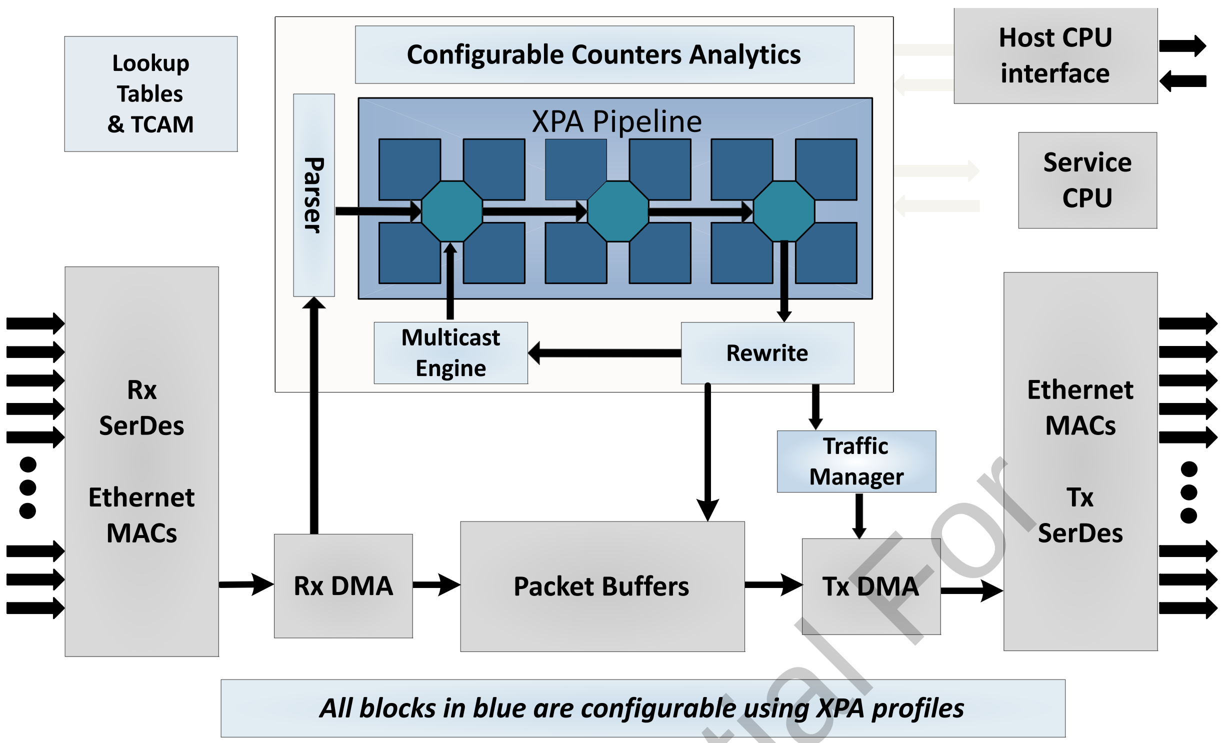


Figure 2–1 XPliant Switch Architecture

XPA enables parsing and modification of any current, future, or proprietary packet format or tagging scheme. It also supports flexible and customized tables, as well as traditional and proprietary forwarding schemes, through a flexible execution pipe.

The Device *Software-Defined Engine* (SDE) is a programmable switching core. The SDE includes the main blocks involved in packet processing and editing; it is comprised of the following elements (illustrated in Figure 2–2: *Software-Defined Engine (SDE)*:

- Fully programmable header parser (2.1.1 Programmable Parser).
- Forwarding lookup decision engines (LDEs), interconnected in clusters through an inter-switch memory element (ISME) (2.1.2 Lookup Decision Engines).
- Programmable header rewrite engine (URW) blocks (2.1.3 Update and Rewrite Blocks).
- Large block of configurable counters for policing, sampling, analytics, and debugging purposes and for searchable memory data structures (ACM).
- Multicast replication engine (MRE) (2.1.4 Multicast Replications and Modifications Block).

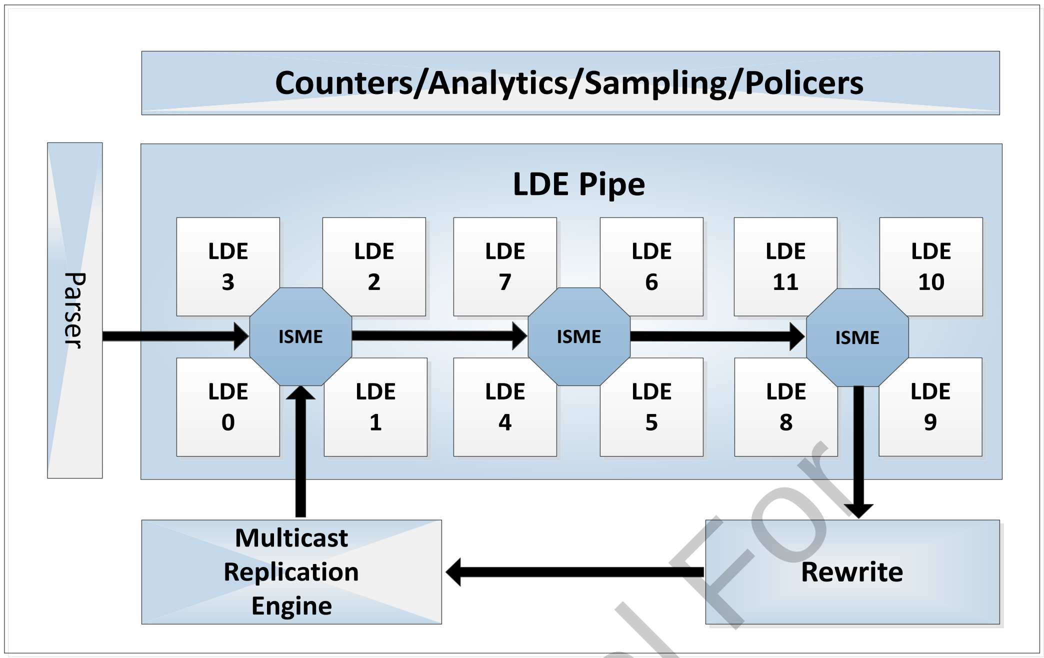


Figure 2–2 Software-Defined Engine (SDE)

The software-defined forwarding processing of an SDE starts with the parser, proceeds through the LDEs, and finishes in the update-rewrite/multicast replication blocks.

2.1.1 Programmable Parser

The programmable parser includes the following features:

- Supports parsing of any existing header, future header, or proprietary header.
- Per ingress port configurable parsed header size based on required latency and functionality:
  - 64-byte header—Low latency and high throughput mode, with up to four layers of parsing.
  - 128-byte header
  - 192-byte header
- Extracts header information into user-definable layers.
- Parses up to eight layers of packet data.
- Supports up to 64 layer types (Ethernet, IPv4, IPv6, MPLS, YOUR_LAYER_TYPE, etc.) and up to 256 concurrent packet templates.
- Collects programmable metadata during packet parsing.
- Supports XP header and E-Tag (802.1BR) header data extraction.
- Creates a token that holds packet information for forwarding decisions.

  A *token* is a data structure that carries internal information and data for the device packet processing, such as:
  - Timestamp.
  - Information to describe and identify a flow/packet.
  - Parsed packet-header layer information.
  - Control data for SDE processing.

- Information to describe and identify a flow/packet.
- Parsed packet-header layer information.
- Control data for SDE processing.
- Forwarding parameters.
- QoS parameters.
- Mirroring parameters.
- Packet modification-related information.
- 64-bit Scratchpad metadata.
- The Scratchpad is uniquely definable on input and output per Engine hop.

2.1.2 Lookup Decision Engines

Each LDE is a generic lookup and decision engine that can be programmed to perform one or more packet processing tasks. An LDE completes the following steps:

1. Inputs a token.
2. Based on the token and the programmed software-defined logic, the LDE issues a lookup request to the search engine (SE) tables.
   - A lookup is issued with a super key, which includes keys for all the lookups.
   - Up to four SE tables can be accessed in parallel.

     SE tables can be of various types, such as TCAM, Hash, LPM, or Direct Index.
   - A key for each SE table may contain different fields for each table lookup.
   - A lookup can be marked as regrettable (which may not be served in case of memory access congestion or bandwidth access limitations) or not (which halts the processing until granted).
   - Each SE result carries the following:
     - SE Sorry bit (for rejected regrettable lookups)
     - SE Hit bit
     - SE Hit address
     - SE Hit data
3. Based on the token and the programmed software-defined logic, the LDE issues a lookup request to the advanced counters module (ACM).
   - Up to four requests to the ACM can be accessed in parallel.
   - The ACM supports three modes per request: counting, sampling, and policing.
4. Based on the programmed software-defined logic, the lookup results and the input token are merged to generate an output token and define the next engine to continue the packet forwarding processing.
5. The LDE checks token time-to-live (TTL) to avoid token livelock.
6. The LDE passes the output token and the processing to the programmed next engine.

Token loopback to the same LDE for performing multiple lookups and modifications in serial is also supported; however, bandwidth and processing order considerations must be applied.

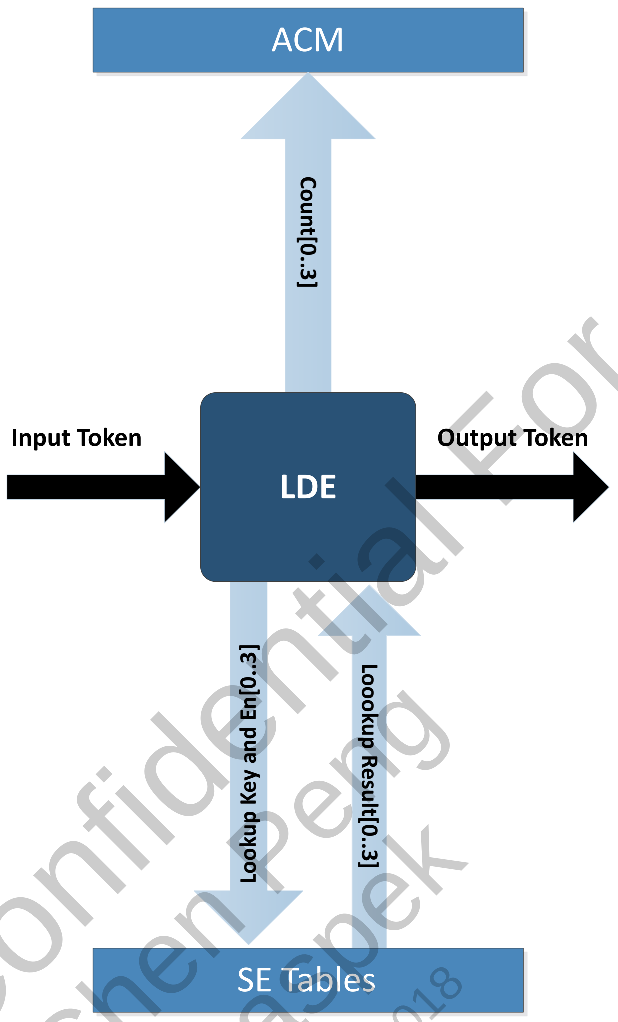


Figure 2–3 Lookup Decision Engine (LDE)

7. At the end of the LDE-forwarding pipe-packet processing, a token reaches the header Update and Rewrite Engine (URW), which resolves the packet destination, modifies the existing packet header, and adds tunneling headers, if needed. Additionally, the engine may forward packet copies to the host CPU and/or to the multicast replication engine, if required.

2.1.3 Update and Rewrite Blocks

The update and rewrite blocks perform the following functions:

- Resolve the final destination of the packet based on VIF lookups, modify the existing packet header, and add tunneling headers.
- Apply the packet drop decision based on VIF lookups, incoming token `<PacketCommand>`, and Traffic Manager queries.
- Generate copies of the header and token and forward packet copies to the host CPU and/or the ingress/egress analyzer and/or the multicast replication engine, if required.

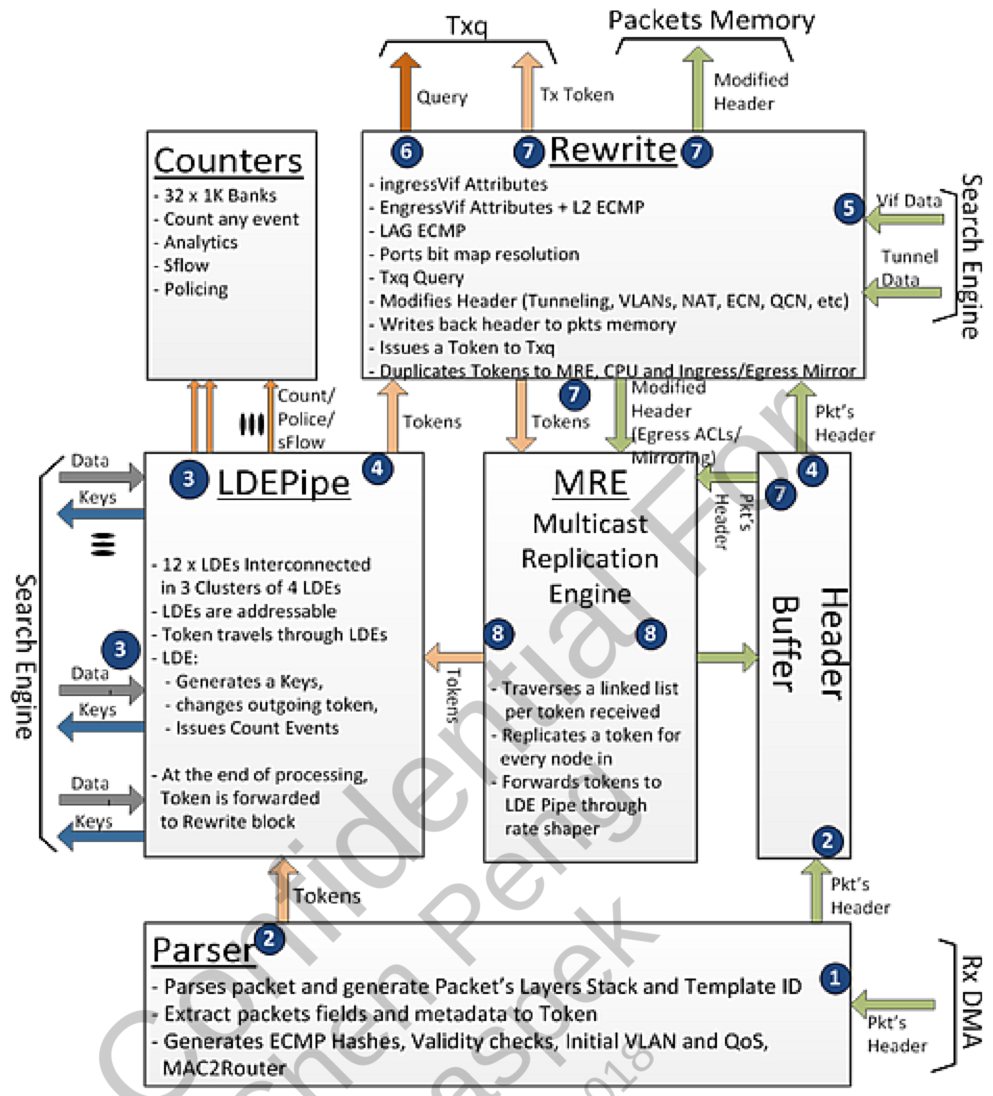
- Build packet headers, including stripping, modifying, and inserting layers within the header, as well as adding the XP headers for CPU-bound and loopback packets.
- Complete packet truncation for packets forwarded to the CPU or to an analyzer.
- Perform ECN marking.
- Perform MTU check.
- Supply source port suppression.
- Perform LAG resolution.
- Apply egress filter.

2.1.4 Multicast Replications and Modifications Block

The multicast replications and modifications block performs the following functions:

- Goes through a linked list of nodes and duplicates a token with different encapsulation properties for every node; implemented as a two-dimensional linked list.
- Supports any number of linked list nodes based on memory allocated.
- Returns packets to LDE pipe after modification (for example, tunnel termination and then lookup based on inner header).

Schematically, a packet walk-through can be represented as shown in Figure 2–4: *Packet Flow*. For more details, refer to the Functional Specifications document.



Txq
Packets Memory
Query
Tx Token
Modified Header
Counters
- 32 x 1K Banks
- Count any event
- Analytics
- Sflow
- Policing
Rewrite
- ingressVif Attributes
- EgressVif Attributes + L2 ECMP
- LAG ECMP
- Ports bit map resolution
- Txq Query
- Modifies Header (Tunneling, VLANs, NAT, ECN, QCN, etc)
- Writes back header to pkts memory
- Issues a Token to Txq
- Duplicates Tokens to MRE, CPU and Ingress/Egress Mirror
Vif Data
Tunnel Data
Search Engine
Count/ Police/ sFlow
Tokens
Tokens
Modified Header (Egress ACLs/ Mirroring)
Pkt's Header
Data
Keys
LDEPipe
- 12 x LDEs Interconnected in 3 Clusters of 4 LDEs
- LDEs are addressable
- Token travels through LDEs
- LDE:
- Generates a Keys,
- changes outgoing token,
- Issues Count Events
- At the end of processing, Token is forwarded to Rewrite block
Search Engine
Data
Keys
Data
Keys
Tokens
MRE
Multicast Replication Engine
- Traverses a linked list per token received
- Replicates a token for every node in
- Forwards tokens to LDE Pipe through rate shaper
Pkt's Header
Header Buffer
Tokens
Pkt's Header
Parser
- Parses packet and generate Packet's Layers Stack and Template ID
- Extract packets fields and metadata to Token
- Generates ECMP Hashes, Validity checks, Initial VLAN and QoS, MAC2Router
Pkt's Header
Rx DMA

Figure 2–4 Packet Flow

2.2 Forwarding Pipeline Processing Primitives

- 2.2.1 Packet Template
- 2.2.2 Bridge Domain
- 2.2.3 Virtual Interface (VIF)
- 2.2.4 Header Manipulations
- 2.2.5 XP Header

2.2.1 Packet Template

The XPliant Software-Defined Platform refers to the packet template as a stack of layers. The Platform is flexible and supports the identification of both standard and proprietary headers; for example:

- Ethernet, PBB Ethernet, Tu-Ethernet
- IPv4, IPv6, MPLS
- FCoE, TCP, UDP
- ICMP, IGMP, GRE
- ICMPv6, VXLAN, TRILL
- CNM, ARP

The Parser identifies the layer types; each layer is represented uniquely by <layerType>.

Identified headers can be combined in different ways into the layers; for example:

- Ethernet
- Ethernet + FCoE
- Ethernet + IPv4 + UDP + VXLAN + Tu-Ethernet + IPv4 + IGMP

The Parser also performs the combination of headers into layers. Each arrangement of the layers is represented by a unique <templateID>, which is used by the forwarding pipeline engines for the packet processing.

2.2.2 Bridge Domain

A bridge domain (BD) is a set of logical interface(s) configured with the same broadcast and flood behavior. A *logical interface* represents a physical interface and its encapsulation. An *encapsulation* represents the type of frames accepted or sent out of the physical interface. Table 2–1: *Encapsulation Type Enumeration* explains the correlation between the encapsulation type enumeration and the encapsulation data by method:

Table 2–1 Encapsulation Type Enumeration

Encapsulation Type	Encap Data	Ingress	Egress
802.1Q untagged	VID	Ingress classification on interface and VID.	Egress encapsulation to use VLAN-unaware header.
802.1Q tagged	VID	Ingress classification on interface and VID.	Egress encapsulation to use VLAN-aware 802.1Q header; can also be used for VLAN translation.
Q-in-Q untagged	Not applicable.	Not applicable.	Egress encapsulation to remove both S-Tag and C-Tag.
Q-in-Q C-Tag	cVID	Not applicable.	Egress encapsulation to remove sTag only; packet egresses with C-Tag only.

Table 2–1 Encapsulation Type Enumeration

Encapsulation Type	Encap Data	Ingress	Egress
Q-in-Q S-Tag	sVID	Not applicable.	Egress encapsulation to add or replace S-Tag only.
VXLAN/NVGRE/ PBB/MPLS	Tunnel ID	Not applicable.	Egress encapsulation to add tunnel-specific encapsulation to the L2 domain.

When the packet ingresses to the XPliant device, it is assigned various attributes as configured per the ingress port in the Initial Token table (ITT) (B.1 Initial Token Table (ITT), including ingressVif and VLAN-ID override (see `pvidModeAllPkt` in B.1 Initial Token Table (ITT).

A combination of this ingressVif (from the default ingress port setting) and a VLAN-ID (which may be the packet original VLAN-ID (outermost VLAN in case of tunnels or other headers encapsulations) or the VLAN-ID assigned by the Initial Token table) is used as a key to various IVIF assignment tables (B.3.1.1 Port VLAN Table), from where a bridge domain ID (BD-ID) is extracted.

The software-defined logic identifies a bridge domain using the BD-ID. Each BD-ID represents unique attributes for the domain, which can be configured using specific accessors for each attribute (B.4.1 BD Table).

Every incoming packet is inspected against a Bridge Domain table and is assigned the bridge domain attributes extracted from the table if there is a lookup match or dropped with the corresponding reason code if there is a lookup miss (see `XP_BRIDGE_BD_TABLE_MISS` in Appendix A, Reason Codes).

The BD assignment mechanism is given below:

```
if( (tunnel termination table lookup is hit)
{
    if(tunnel terminationTable.setBd == true)
         packet BD = tunnel terminationtable configured BD.
   else if (portConfigTable.setBd == true)
         packet BD = port config table configured BD.
   else
         packet ingress vlan check failed and packet is dropped with reason code =
XP_IVIF_RC_NO_BD
}
else
if((port vlan table lookup is hit)
{
    if(portVlanTable.setBd == true)
         packet BD = port vlan table configured BD.
   else if (portConfigTable.setBd == true)
         packet BD = port config table configured BD.
   else
         packet ingress vlan check failed and packet is dropped with reason code =
XP_IVIF_RC_NO_BD
}
else
```

```
{
   if (portConfigTable.setBd == true)
         packet BD = port config table configured BD.
   else
         packet ingress vlan check failed and packet is dropped with reason code =
XP_IVIF_RC_NO_BD
}
```

The BD-ID is stored within the bridge domain context (`xpL2DomainCtx_t`).

An L2 domain is a generic concept that provides the flexibility to manage the internal primitives of a bridge domain. It generalizes a VLAN instance: every interface added to an L2 domain with a VLAN-ID generates a specific port VLAN index; every port-VLAN interface created identifies unique attributes for the interface.

`xpL2DomainMgr` as a singleton class provides the associated services in the L2 domain. `xpL2DomainCtx_t` structure is an L2 domain context that abstracts a bridge domain for all XPliant devices. This is a global structure that holds globally allocated IDs for primitives specific to a bridge domain instance. After a successful L2 domain is created, services such as adding and removing interfaces, adding and removing tunnels, updating per-VLAN attributes, and updating per-interface per-VLAN attributes are available.

For more details see the *XPliant Software Programmer and Configuration Guide*.

2.2.3 Virtual Interface (VIF)

The Platform supports *virtual interface* (VIF) architecture. A VIF is a system-wide global ID representing a virtual interface.

In the XPliant Software-Defined Platform, VIFs provide a level of indirection when making forwarding decisions:

- Received packets are assigned an ingressVif identifier (ingress virtual interface or IVIF).
- Packets are forwarded to an egressVif (egress virtual interface or EVIF).

Features such as port, IP routing, IP multicast, tunnels, L2 bridging, L2/L3 ECMP, VLAN, and LAGs all rely on VIF processing for their implementation.

The ingressVif and egressVif assignments are programmable and can be based on any lookup or any packet attribute.

An IVIF can represent:

- Ingress physical port.
- Link Aggregation Group (LAG) of physical or virtual ports.
- Router interface (typically {Port,VLAN}).
- Virtual interface of any type.
- Tunnel termination point.

The IVIF is primarily used for source suppression and Layer 2 station movement detection (3.4.1 Layer 2 Bridging).

The IVIF is assigned according to the software-defined logic in the IVIF Assignment and Tunnel Termination Engine, based on the portVif/outerVLAN lookup hit *or* tunnel termination hit.

An EVIF can represent:

- Egress physical port.
- List of egress physical ports.
- Tunnel start.
- Linked list of egress virtual interfaces.

The software-defined logic can populate the EVIF by various mechanisms, including the following:

- `xpTnlIvifTbl<eVif>`in tunneling cases (3.7 Tunnel).
- `xpFdbTable<vif>` (B.3.1.8 FDB Table) in case of MAC DA lookup for Layer 2 forwarding (3.4.1 Layer 2 Bridging).
- `xpIpvxMcBridgeTable<vif>` (B.3.1.9 IPv4 Multicast Bridge Table) in case of multicast groups for the IP multicast bridging (3.4.2 IP-Based Multicast Bridging).
- `xpIpvxUcHostTable<vif>`or xpIpv4UcRouteTable`<vif>` (B.3.1.13 IPv4 Host Table) in case of DIP lookup for routing (3.5.1.3 Route Table Lookup).
- `MDT<egressVif>` (B.2 Multicast Distribution Table) in case of a packet replication (3.12 Multicast Replication Engine (MRE) / Multicast Forwarding).

The EVIF is primarily used for extracting a destination port and/or destination port mask and instruction modifications and insertions.

The assigned VIF is carried in the token to the Update and Rewrite Engine, which uses it to issue lookups in the VIF SE table to obtain the actual destination in order to resolve the following:

- Loops prevention by avoiding packets forwarding back to the source port or any of the source ports that belong to the same LAG (IVIF lookup).
- Egress LAGs membership (EVIF lookup) so that a packet is sent only to a single selected LAG member.
- Cases where a packet is to be replicated (identical packet to be sent to multiple ports to the Traffic Manager) or replicated and modified (packets with a different/ modified header to be sent to the Multicast Replication Engine).
- Mirroring (mirrorMask) information is incorporated in the VIF response).

NOTE: The VIF lookups will not be issued in the following cases:

- Token<PktCmd> == DROP because the destination resolution is not VIF based but rather extracted from the Reason Code table (Appendix A, Reason Codes).
- Token<PktCmd> == TRAP because the packet to be dropped and the destination resolution is not required.

The IVIF and EVIF entries contain a port bit map.

To resolve LAG and ECMP paths, the IVIF and the EVIF lookup results are bitwise ANDed with the Trunk Resolution table (TRT) (3.4.5 Layer 2 LAGs).

`xpVifMgr` is a singleton class that provides the association with VIF management services.

The VIF Manager belongs to the XP Primitive Layer, providing an interface to the XP Feature Managers to allocate and interact with VIFs in the platform. For instance, the XP Feature Layer L2 Domain Manager may operate in independent instances on the following VIF types:

- Port-VIF—Configured with a port ID; reused across multiple VLANs.
- Port-VLAN-VIF—Configured with a port ID and VLAN-specific information.
- LAG-VIF—Configured with LAG-specific ports; reused across multiple VLANs.
- LAG-VLAN-VIF—Configured with LAG-specific ports.
- Tunnel-VIFs—Configured with tunnel-specific data by the tunnel manager.

For more details, see the *XPliant Software Programmer and Configuration Guide*.

2.2.4 Header Manipulations

The XPliant Software-Defined Platform processing may result in either modification of the existing header or insertion of a new header. The header modification manager provides an interface to program the Platform on how to modify or insert new headers.

The header manipulations are applied in the Update and Rewrite Engine, which modifies packet headers before they are transmitted out. The rewrite commands are derived based in the lookups; the commands are received on a per layer basis.

The engine receives a token, which contains packet-based data as well as forwarding-pipe processed information, such as packet command, layer data, rewrite pointers, insert pointers, etc., and performs various lookups in its local memories in order to extract and apply its decisions.

The following memories are accessed

- SE—Indexed based on the incoming token EVIF and IVIF fields.
- Trunk Resolution table—Indexed by the LAG hash, where each bit represents a logical port. It is used for the trunk member selection resolution.
- Egress Filtering table—Indexed by the incoming token <filter_grp_num>, where each bit represents a logical port. It is used for the egress ports selection resolution.
- TemplateID table—Contains the layer information for each of the packet header layers stack-up.
- Reason Code table—Indexed by the incoming token <reasonCode>; used to apply reason-code related attributes (Appendix A, Reason Codes).

Upon getting the lookup results, the engine resolves the destination bitmap and determines whether the packet is uni-destination or to be replicated.

In addition, it updates and rebuilds the packet header, based on operations such as the following:

- Strip outer (first –N) headers layers (if any).
- Modify the header based on the rewritePointers.
- Insert the new headers based on the insertPointers or XP Header insertion.

- Format the final header.

After this operation completes, the Update and Rewrite Engine passes the modified final header accompanied with a token to the next engine or to the Multicast Replications Engine (MRE).

Some features, such as tunneling, are implemented based on using insertion pointers that contain insert instructions and header data to be inserted in the network order.

A VIF may contain pointers to the header insertion table retrieved from the header modification manager. These pointers can be set as ingress or egress attributes.

A VIF may contain header data that is used by modify or insert instructions retrieved by their respective pointers as a source. This data can be set as ingress or egress attributes. The header data must be stored in network order.

For more details, see the *XPliant Software Programmer and Configuration Guide*.

2.2.5 XP Header

The XP Header is an XPliant proprietary header (Appendix D, XP Header) that is a fundamental block for communicating with the host CPU. The XP Header is prepended to the packet and is typically used for internal communication between the host CPU and the Platform or for internal communication within the Platform (for looped-back traffic) communication.

The main purposes of the XP Header are:

- Carry additional data (such as Reason Code, Src Port, etc.) for packets forwarded to the host CPU. The data is currently using the output metadata scratchpad from either the last LDE or MME to the CPU.
- Provide a means for sending traffic by the host CPU through the Platform when some decisions are CPU-controlled and to assume forwarding pipe-specific processing, such as skipping certain software-defined forwarding pipeline engines or carrying forwarding decisions resolved by the host CPU (such as ingressVif).
- Carry timestamp information.
- Be useful when some frames are looped back.

The Software-Defined Platform uses an XP Header with a fixed size of 24 bytes.

NOTE: The Platform supports configuration of the XP Header size for future expansions

The Parser terminates a packet that arrives with an XP header on ingress after the Parser extracts the corresponding data, including the expected metadata scratchpad information. The packet is optionally added by the Update and Rewrite Engine on egress or removed by the TxDMA if the packet is not destined to an XP Header-enabled port.

The Parser, Update and Rewrite Engine, and TxDMA processing is described in more detail below.

Parser

- May receive packets with an XPH from Loopback port or from CPU port (it will not receive a TO CPU XPH).
    - When it does, all packets received on that port include an XPH.
    - Configuration: Per Source Port – XPHExists.
    - if XPHExists<Port[n]> is SET all packets received on this port include an XPH and set token.XPHExists to 1.
- Parsing
    - XPH fields are extracted into the packet Token and Context.
        - Normal parsing skips XPH.
    - Configuration: Per source port – Initial Offset.
    - For ports receiving packets with XPH, set Initial Offset to 24 bytes.
- First Layer offset is not zero.
    - XPH is not a layer.
    - First Layer offset for packets with XPH is 24bytes.
        - Parser forwards Layer0 Start offset to Rewrite.

Update and Rewrite Engine

- Removes XPH:
    - If token.XPHExists = 1, that means that Layer0 Offset is 24 bytes and not zero.
    - When assembling the header, an existing XPH must be removed.
    - A new XPH may be added.
- XPH is added *only* to packets forwarded to SDMA; the header byte count includes the 24bytes of XPH.
- Adding an XPH Header:
    - When an XPH is added, set TxqToken.XPHExists to 1.
    - If packet is forwarded to CPU, add a TO_CPU XPH.
    - If packet is forwarded to LOOPBACK port or QMirrorEn is set, add a LOOPBACK XPH.
    - In case of multicast, add XPH if one of the ports requires XPH.
- Configurations:
    - TO_CPU_XPHEn – enable sending packets with XPH to CPU.
        - When set, all packets sent to CPU are prepended with a TO CPU XPH.
    - port<n>LOOPBACKXPHEn
        - When set, all packets forwarded to this target port are prepended with a LOOPBACK XPH.
    - QMirrorEn – Configuration from egressVif entry.
        - When set, all packets forwarded to this egressVif are prepended with a LOOPBACK XPH.

TxDMA

- Configuration per port:
  - Port<n>RemoveXPH
  - Default configuration for CPU port and Loopback port – do not remove.
  - Default configuration for Network Ports – Remove.
- If (txToken<XPHExists> == 1){
  If (Port<n>RemoveXPH == 1) remove XPH}.

Chapter 3

XPliant Software-Defined Forwarding Pipeline

3.1 Profile Pipeline

3.1.1 Default Profile Pipelines

The Device Software-Defined Engine (SDE) is a programmable switching core. To process a bandwidth of up to 3.2Tbps, XPliant integrates two SDEs, each processing half of this bandwidth. These two SDEs can be programmed identically or can be programmed differently, based on system requirements.

SDEs incorporate a programmable pipeline. The pipeline may be comprised of up to 12 Lookup and Decision Engines (LDEs). LDEs are addressable, and packet flow through LDEs is programmable through the profiling.

A profile represents "personality" that is pushed to the configurable elements, including the parser, lookup decision engines (LDEs), MME, and the update and rewrite engine. The profiles are typically loaded at start-up as XPC (XPliant Pipeline Compiler) output of metadata configurations,for various forwarding processing pipeline blocks. Forwarding tables can be modified in runtime. Profiles enable the implementation of different forwarding processing pipeline schemes on the same hardware, tailored to the Place-In-Network (PIN) deployment requirements.

NOTE:

- The default profiles listed in the following sections incorporate soft-coded functionality for some of the typical implementations they describe. Being software-definable, this functionality, as well as resource partitioning, can be further optimized for specific customer deployments.
- Not all logical blocks are relevant to all profiles. For example, NAT is not included in either the 2.4 Bpps profile or OpenFlow profile.

3.1.1.1 Generic Profile Pipeline

The forwarding processing pipeline consists of engines, which are defined as a set of conditional logic that makes forwarding decisions based on a series of lookups using information derived from an ingressing packet. Figure 3–1: *Forward Processing Pipeline* outlines the schematic pipeline representation (”Profile_pipeline_LDEs”).

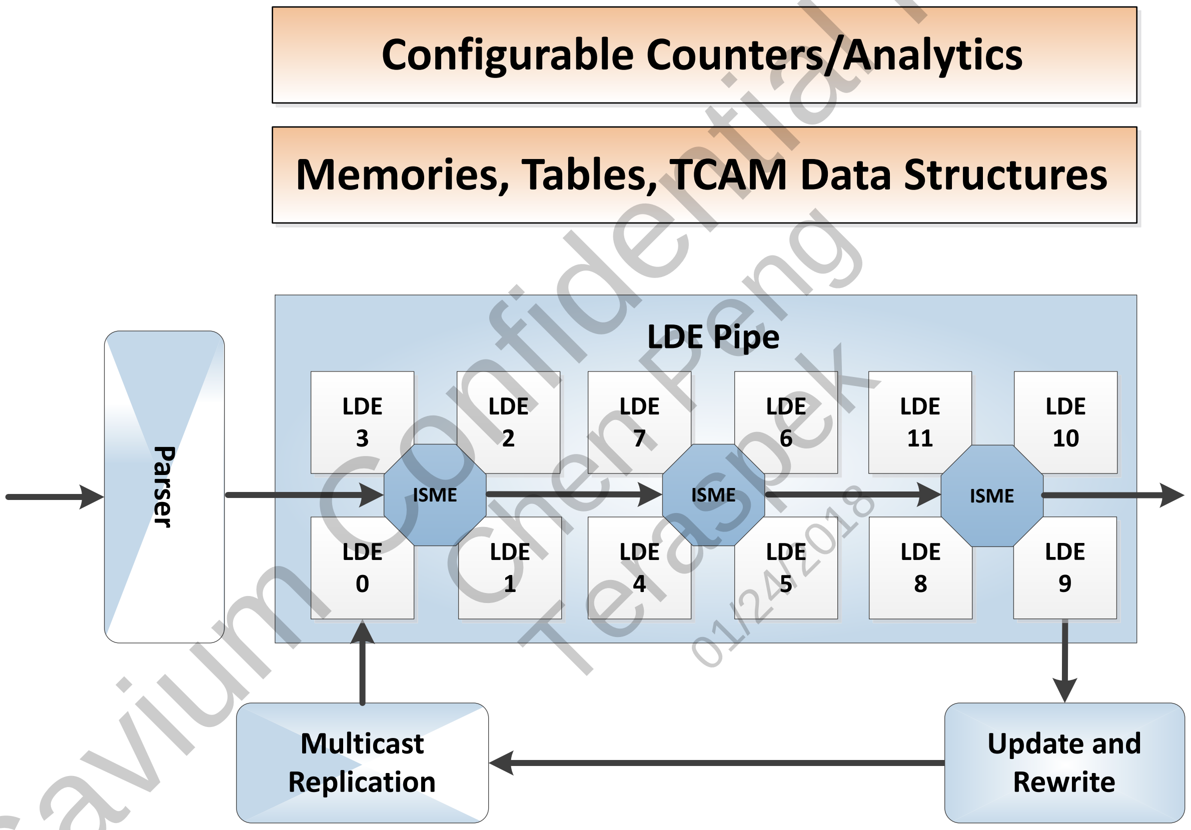


Figure 3–1 Forward Processing Pipeline

NOTE: The default profiles listed in the following sections incorporate soft-coded functionality for some of the typical implementations they describe. Being software-definable, this functionality, as well as resource partitioning, can be further optimized for specific customer deployments.

3.1.1.2 1.2Bpps Default Profile Pipeline

The 1.2Bpps default profile represents a typical functionality implementation at a processing rate up to 1.2Bpps packets per second.

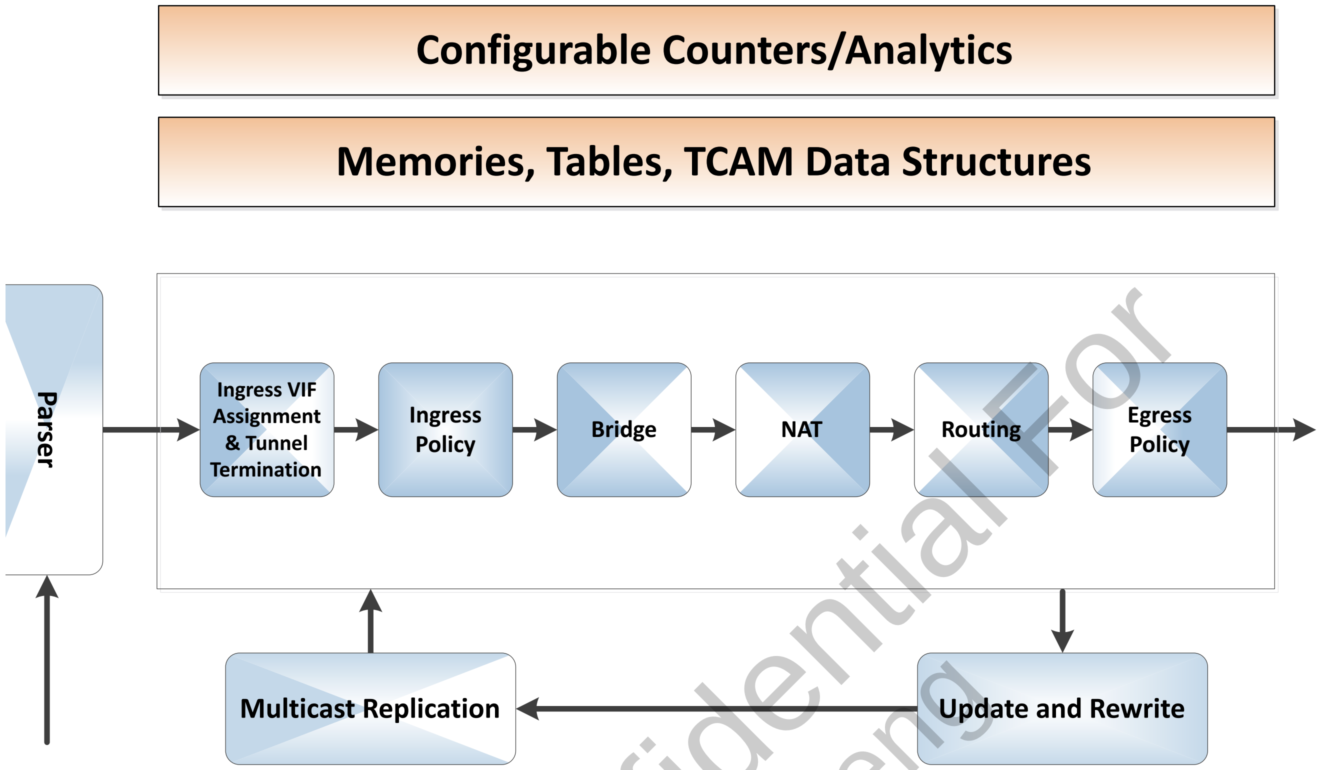

Configurable Counters/Analytics
Memories, Tables, TCAM Data Structures
Parser
Ingress VIF Assignment & Tunnel Termination
Ingress Policy
Bridge
NAT
Routing
Egress Policy
Multicast Replication
Update and Rewrite

Figure 3–2 1.2Bpps Default Profile Pipeline

3.1.1.3 2.4Bpps Default Profile Pipeline

The 2.4Bpps profile represents a typical functionality implementation at processing rate up to 2.4Bpps packets per second.

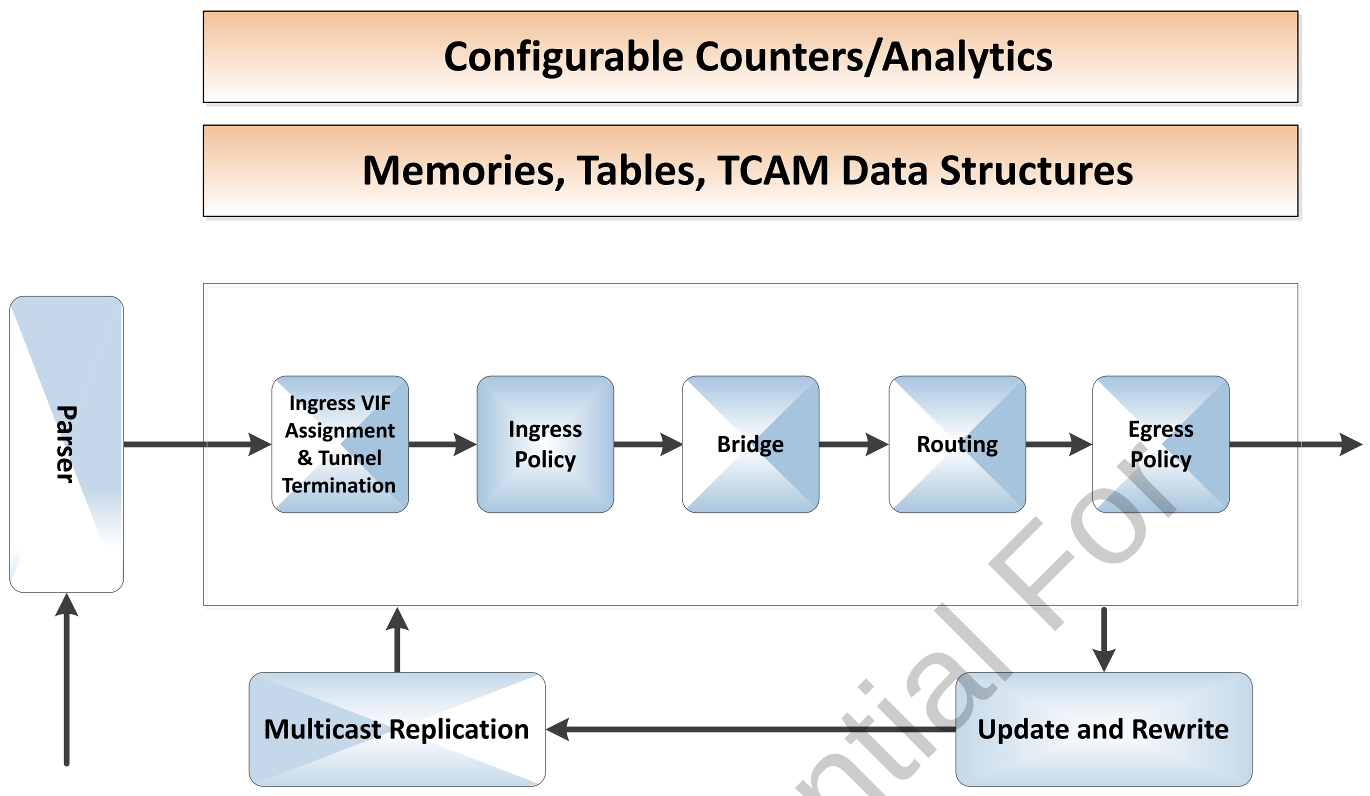

Configurable Counters/Analytics
Memories, Tables, TCAM Data Structures
Parser
Ingress VIF Assignment & Tunnel Termination
Ingress Policy
Bridge
Routing
Egress Policy
Multicast Replication
Update and Rewrite

Figure 3–3 2.4Bpps Profile Pipeline

3.1.1.4 OpenFlow Hybrid Profile Pipeline

The OpenFlow hybrid profile represents a typical functionality implementation of OpenFlow and standard switching processing together.

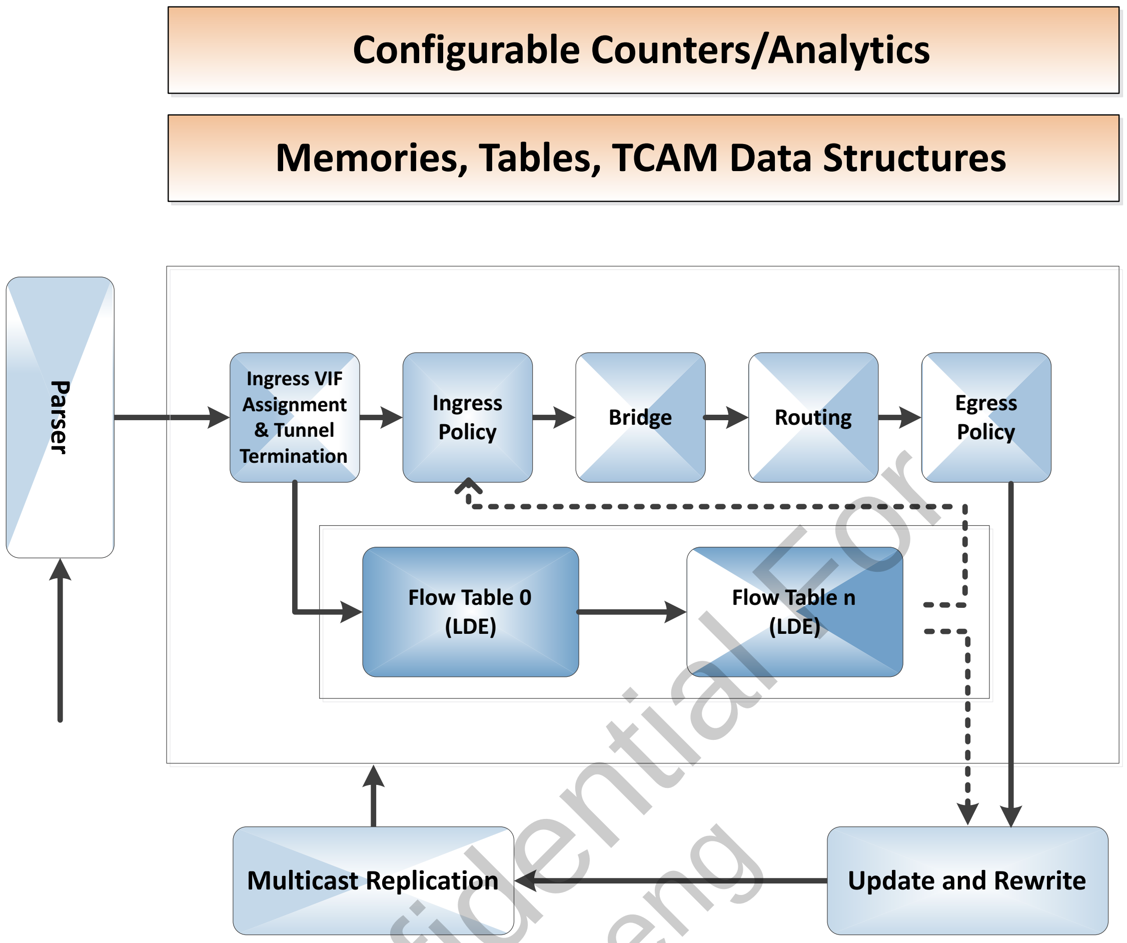


Figure 3–4 OpenFlow Hybrid Pipeline

3.1.2 Programmable Parser

- Initial VLAN assignment (802.1Q based, port based, protocol based, or costumer specific).
- Initial QoS assignment (port based, 802.1p based, IP TOS/DSCP based, MPLS EXP based, or customer specific).
- Programmable packet hashes for ECMP.
- Validity checks for well-known protocols:
  - Layer 2—MAC checks, etc.
  - IP—IP header validity checks, checksum, TTL, TTL-1, etc.
- Initial port-based IVIF and EVIF attributes.
- Support for XP header and E-Tag header data extraction.

3.1.3 Ingress VIF Assignment and Tunnel Termination Engine

- Tunnel Termination (PBB 802.1ah, VxLAN, NvGRE, Geneve, IP Tunnels, MPLS Tunnels).
- Initial port-based IVIF and EVIF attributes.

3.1.4 Ingress Policy Engine

Ingress ACL packet filtering and classification for IPv4 and/or IPv6 packets:

- Port, Bridge, and Route-based ACL.
- Action, Egress interface, Policer ID, Traffic Class.

3.1.5 Bridge Engine

- Regrettable Mac SA / Mac DA FDB lookup.
- Bridge, Bridge and IP Bridge Multicast lookup.

3.1.6 NAT Engine

Src-NAT, Dest-NAT, Double NAT, M-NAT

3.1.7 Routing Engine

- IPv4/IPv6 routing.
- IP Unicast routing.
- IP Host lookup.
- PIM Bi-directional.
- IP Multicast lookup.

3.1.8 Egress Policy Engine

- IPv4 / IPv6 Egress ACL lookups.
- Egress Bridge Domain lookup.

3.1.9 Update and Rewrite Block

- Programmable header modification with up to eight modification commands per packet.
- Programmable tunnel termination.
- Programmable header insertion (tunnel start).
    - Supports any tunneling header format.
    - Tunneling header size of up to 64 bytes
- IP headers and TCP headers checksum generation.
- QCN packet generation based on target queue congestion level.
- ECN marking (for DCTCP) of IP headers.
- Layer 2 and LAG ECMP with up to 1 KB paths.
- Group ID-based filtering, with up to 256 filtering groups (can be used for MSTP of source-based filtering).

3.1.10 Multicast Replications and Modifications Block

- IP multicast bridging and routing to outgoing interfaces.
- Flooding of L2 packet to a remote and local site/station.
- Mirroring.

3.2 Features Summary

The Search Tables associations per Engine are listed below

Ingress VIF Assignment and Tunnel Termination Engine

Lookup types supported and tables accessed:

- Port/VLAN and PBB Lookup
  - Lookup 0: Port VLAN Table
  - Lookup 1: PBB Tunnel Termination Table
- Port/VLAN and Virtual Tunnel Lookup (VxLAN, NvGRE, Geneve)
  - Lookup 0: Port VLAN Table
  - Lookup 1: Local VTEP Table
  - Lookup 2: Remote VTEP Table
  - Lookup 3: Instance ID Table
- Port/Vlan and IPvX Tunnels Lookup
  - Lookup 0: Port VLAN Table
  - Lookup 1: X Over Ipv4 Tunnel Termination Table
- MPLS Tunnel Lookup
  - Lookup 0: Port VLAN Table
  - Lookup 1: X Over MPLS Tunnel Termination Table

Ingress Policy1 Engine

Lookup types supported and tables accessed:

- IPv4 Lookup
  - Lookup 0: Port ACL Table
  - Lookup 1: Bridge ACL Table
- IPv6 Lookup
  - Lookup 0: Port ACL Table
  - Lookup 1: Bridge ACL Table
- MPLSv4 Lookup
  - Lookup 0: Port ACL Table
  - Lookup 1: Bridge ACL Table

Bridge Engine

Lookup types supported and tables accessed:

- Bridge Lookup
  - Lookup 0: FDB Table MAC SA
  - Lookup 1: FDB Table MAC DA
- Bridge MC lookup
  - Lookup 1: FDB Table MAC DA
- Bridge IP MC Lookup
  - Lookup 0: Bridge IP MC Table

Ingress Policy2 Engine & NAT Engine

Lookup types supported and tables accessed:

- IPv4 Lookup
  - Lookup 0: NAT Table
  - Lookup 1: Route ACL Table
- IPv6 Lookup
  - Lookup 0: Route ACL Table
- MPLSv4 Lookup
  - Lookup 0: Route ACL Table

Routing Engine

Lookup types supported and tables accessed:

- IPv4 Unicast Routing Lookup
  - Lookup 0: IPv4 UC Route Table
  - Lookup 1: Host Table
- IPv6 Unicast Routing Lookup
  - Lookup 0: IPv6 UC Route Table
  - Lookup 1: Host Table
- IPv4 Multicast Routing Lookup
- Lookup 0: PIM Bi-Directional Table
  - Lookup 1: Multicast Route Table
- IPv6 Multicast Routing Lookup
  - Lookup 0: PIM Bi-Directional Table
  - Lookup 1: Multicast Route Table

Egress Policy Engine

Lookup types supported and tables accessed:

- IPv4 Lookup
  - Lookup 0: EACL IPv4 Table
  - Lookup 1: Egress Bridge Domain Table
- IPv6 Lookup
  - Lookup 0: EACL IPv6 Table
  - Lookup 1: Egress Bridge Domain Table

3.3 ACL

3.3.1 Description

Ingress and Egress ACL are comprised of TCAM ACL lookups in the Ingress and Egress Policy LDEs. An ACL provides a list of rules that are applied to the inbound flow of packets from an interface and to packets that are send on an outbound interface. The rules that are applied on inbound are referred to as Ingress Access Control Lists (IACL) and those on the outbound as Egress Access Control Lists

(EACLs). IACLs typically apply a list of rules on the ingress interface attributes, while EACLs provides packet filtering and/or classification based on various packet fields at the egress interface.

Rules are implemented in TCAM and resulting actions in SRAM. Each TCAM rule consists of two parts: a key and a mask. The mask is a bitmap that defines which bits of the key are to be compared to the flow: a value of 0 in the mask means to compare, a value of 1 means “do not care”. There is corresponding programmable action data for a key/mask rule.

A match occurs when the flow matches to the derived key bits that are defined as “to compare bits” by the mask. There is corresponding programmable action data for a key/mask rule. The data action is applied on the flow if there is a TCAM match. Various matching rules and actions are supported in the XPliant Software-Defined Platform.

3.3.2 Ingress ACLs

3.3.2.1 Software-Defined Implementation

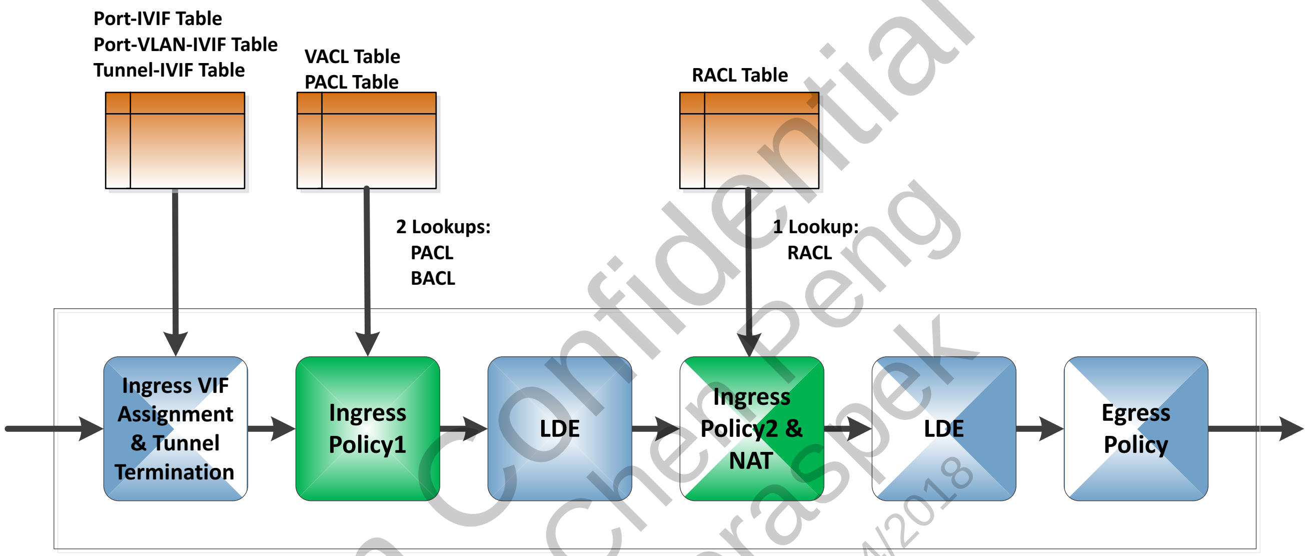


Figure 3–5 Default Pipeline with IACL and EACL

Resource Usage

Table 3–1 IACL Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
IACL Engine	1. PACL Lookup 2. BACL Lookup	PACL Table VACL Table	IACL Hit Data result	L2/IPv4 or IPv6 or MLPSv4 Key Type
IACL Engine2 & NAT Engine	1. RACL Lookup	IACL Table	IACL Hit Data result	L2/IPv4 or IPv6 or MLPSv4 Key Type

The Ingress Access Control Lists (IACL) engine provides packet filtering and/or classification based on various packet fields. For each rule entry, the action or data are returned on a match. IACL data is applied on a token when it matches a rule. If no rules match for a list of all rules, the packet is forwarded to the next processing engine; for example, the Bridge Engine.

As shown in the pipeline, Port ACL and VLAN ACL are launched from IACL Engine1, and ACL for only routed flows is launched from IACL Engine2.

Any field combinations in these keys can be used to match on packets. Each method requires key, mask and data to be passed to ensure correct rule programming.

IACL Key

IACL key formats are flexible and are defined at runtime. The key fields are selected from a predefined field list, which provides the flexibility to configure the format and key size based on the deployment scenario. The key fields can be user-defined as long as they fit within the predefined key size, such as 64B, 128B, 192B, and 384B.

Flexible table creation and flexible key definition allow better scalability and efficient resource management. They also provide the ability to change the IACL tables and key formats dynamically as requirements and deployments change.

There are three key types supported by default: one for IPv4/L2, one for IPv6, and one for MPLS V4.

IACL Lookup

There are three IACL lookups supported: PACL, BACL, and RACL. All three IACLs can be applied simultaneously to match the flow of a packet, if enabled.

Each lookup has a designated table and may be individually enabled or disabled through configuration. The PACL and BACL lookups, if enabled, are launched in parallel by the Ingress Policy engine. The RACL lookups, if enabled, are launched by the Ingress Policy2 and NAT Engine. The key for the lookup can be formed based on the type of traffic to be matched and the interface type. Currently, the lookup key to each IACL table is based on the key format set for each lookup type plus the ACL ID.

Each of the lookups is referred to by ACL ID and should have a unique value for the lookup type. ACL ID provides the capability to group rules together, and thus provides the ability to scale down the number of IACL rules programmed. The ID is obtained from the previous physical/virtual interface table hit before the Ingress Policy engine. The ACL ID is applied on all interface types, such as port, LAG, etc. The index values are order dependent; the first index hit will be returned. If counters are enabled, then counters are incremented to count all hits until `isTerminal` entry result is hit.

The maintenance of the ACL ID and index number and order must is performed by the host process. Refer to the API guide on the passing of these values.

The interface types on which the three IACL lookups can be enabled are one of the following:

Port—When enabled on port, the port ACL table lookup is triggered, and the ACL ID obtained from the port table lookup is used as part of the key to the ACL table.

VLAN—When enabled on VLAN, the VLAN ACL table lookup is triggered, and the ACL ID obtained from the VLAN table lookup is used as part of the key to the ACL table.

Routing or tunnel interface—When enabled on the routing or tunnel interface, the router ACL table lookup is triggered, and the ACL ID obtained from the routing or VTEP/table is used as part of the key to the ACL table.

IACL Key Size

The key size is flexible and based on the fields picked from the supported key. The key size is determined to be one of the predefined length. The predefined key size is 64 bits, 128 bits, 192 bits and 384 bits.

IACL Data

There is a predefined IACL data. IACL data defines a set of attributes that are applied on the flow in case of a successful match on a corresponding rule.

The result of the IACL Lookup table when enabled returns the IACL data entries. Each table returns a hit offset that is used as an index to the IACL data table. Each IACL data has a dedicated action table in SRAM. When all IACL lookups are enabled, three IACL data results are returned, one each for table, and data resolution must be performed.

IACL Data Resolution

Based on the enable bit for the Lookup Type of the table, the result of that table is returned for IACL data resolution. The result contains action, egress interface, policer ID, Traffic Class, etc., if they are enabled. If there is more than one hit from the three-table lookup and that table is enabled, then the `isTerminal` flag is considered. The `isTerminal` flag is the user-defined priority (high or low) in the result entry.

When multiple tables are enabled the priority of order will be PACL, BACL, and RACL in decreasing priority. The counter will count all lookup hits until `isTerminal` or highest priority result is hit. The IACL has the ability to OR mirror sessions from all hits until `isTerminal` or highest priority result.

The Data Resolution table provides the logic when multiple IACL are enabled.

Table 3–2 Data Resolution Table on Lookup Type in IACL

NO	PACL: IsTerm	BACL:IsTerm	RACL:IsTerm	Data Resolution
1	1	0	0	PACL lookup result applied.
2	0	1	0	BACL lookup result applied.
3	0	0	1	RACL lookup result applied.
4	1	1	0	PACL lookup result applied.
5	0	1	1	BACL lookup result applied.
6	1	1	1	PACL lookup result applied.

*Assumes all three lookups enabled and hit in SE lookup.

Mirroring and packet command resolution is merged across a hit on all three lookups.

Packet command resolution between the IACL Policy1 engine and the IACL Policy2 engine is as follows:

Table 3–3 IACL Packet Command Resolution

NO	PktCmd	PktCmd	Merged PktCmd
1	Drop	X	Drop.
2	Fwd & Copy	!Drop	Fwd & Copy.
3	Trap	Trap	Trap.
4	Fwd	Fwd	Fwd.

3.3.2.2 Decision Logic

Ingress Policy Engine

The Ingress Policy1Engine and Ingress Policy2 Engine receive the token from the previous engine. The engine will pick the fields to be looked into the search engine based on the configuration for the interface.

Lookup: Build the key based on IPv4, IPv6, or MPLSv4 packet template. Set the Key Type to IPv4, IPv6, or MPLSv4.

Extract the fields for the Lookup-Type that is enabled. The logic of extraction is as below:

- If packet is not Layer3 terminating, copy Layer 3 info from packet Layer 3 like DIP, SIP, DSCP. Also, BD comes from Scratchpad.
- If packet has Layer 4, then copy Source and Destination Port and TCP Flags.
- If packet is an ICMP then copy the ICMP Message Type.
- If packet is a tunnel-terminated packet or Layer 3 is terminated, then copy L2 fields like MAC DA, SA, VLAN.
- If the packet is an IPv4 MPLS packet, copy the MPLS fields like label, exp.
- Copy PACL-ID, BACL-ID, and RACL-ID, along with their enable bit.
- Build the lookup key, then launch search on Search Engine. The result of the lookup is explained in the following Token section.

Token:

The result of the three lookups of PACL, BACL, and RACL are returned from SE and one of them is picked based on enable bit of the lookup and `isTerminal` bit set in the lookup.

The resolution is performed based on:

- Entry user-defined priority (high and low).
- Lookup-based prioritization (PACL, VACL, and RACL)
- Ability to count all hits until terminal (high priority) result.
- Ability to OR mirror sessions from all hits until terminal (high priority) result.

The IACL engine resolves data results. The merge is performed based on the following two criteria:

- Entry priority

- Lookup priority

If the priorities of all entries are the same, then lookup priority is applied. Lookup priority is not configurable: IACL0 is the highest and IACL2 is the lowest lookup priority. For example, if there is a hit on both IACL1 and IACL2 and both data has the same entry priority, then IACL1 data is applied.

```
If isTerminal == TRUE on any hit
    (apply results from this hit) AND
    (count all previous hits (if CNT_EN == TRUE)) AND
    (OR all mirror sessions from previous hits (if MIRROR_SSN_UPD ==
    TRUE))
else
    resolve based on lookup priority:
    PACL = high priority
    VACL = medium priority
    RACL = low priority
```

For example, for PACL results to be applied, the PACL should be enabled. Similarly for all lookups. One of the results is picked based on the Data Resolution table when multiple `isTerminal` values are set.

The following are picked from one of the three lookup results:

If the result of the ACL action like DROP or Trap or Redirect, packet is send directly to URW. Also update rewrite pointers based on tag or untag packet type that needs to be sending out.

If the lookup result has redirect to Evif is TRUE, then get virtual interface and update the packet TunnelId.

Update the egressVif from lookup results to packet and reason code.

Also, if policer set, then get policer ID.

If QoS Traffic Class enabled, then TC is updated in token.

Similarly, QOS parameter of DSCP and PCP are updated in token.

The counting and mirroring are picked from the previous hierarchy like:

Counting on the previous hierarchy is also done. Mirroring when configured for update will be done for all previous hierarchy.

Example in Table for No 3 where RACL `IsTerminal` is only set; the result of RACL are applied. Counting from PACL, BACL, and RACL is also applied. If Mirroring session is configured on any previous hierarchy like BACL or PACL, then they will be also be applied along with RACL.

3.3.3 Egress ACLs

3.3.3.1 Software-Defined Implementation

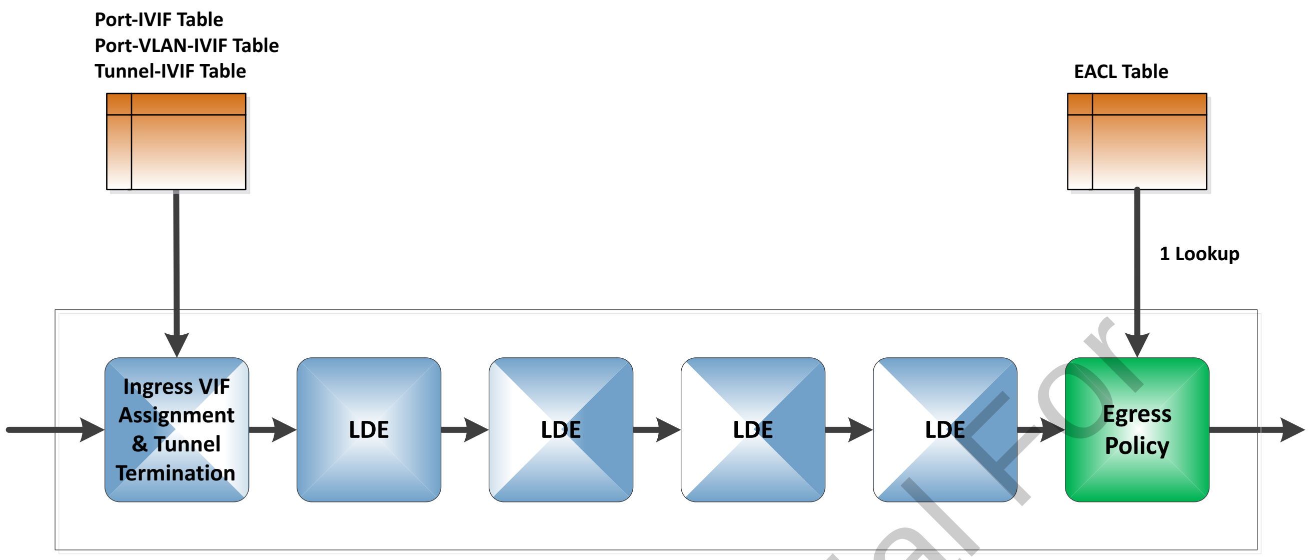


Figure 3–6 Egress ACLs Pipeline View

Resource Usage

Table 3–4 Egress ACLs Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
EACL Engine	1. EACL lookup	EACL for L2 *or* IPv4 Table	EACL Hit Data result	IPv4 Key Type IPv6 Key Type

The Egress Access Control Lists (EACL) engine provides packet filtering and/or classification based on various packet fields after the bridging and routing decisions have been made. EACL lookup rules that are supported are defined below. For each rule entry, the action or data are returned on a match. EACL data is applied on a token when it matches a rule. If no rules match for a list of all rules, the packet is forwarded to the next processing engine; for example, the URW.

The EACL table is flexible and is created at runtime, which provides the flexibility to the user to dynamically define the number of tables and per table size for each table based on their deployment scenario. If the user decides to keep egress filtering disabled, then the EACL table does not need to be created.

EACL Key

The EACL key formats are flexible and are defined at runtime. The key fields are selected from a predefined field list. thus providing the flexibility to configure the format and key size based on the deployment scenario.

Flexible table creation and flexible key definition allows better scalability and efficient resource management. This also provides the ability to change the EACL table and key format dynamically as requirements and deployments change.

There are two key types supported, one for IPv4 and one for IPv6.

EACL Lookup

In the pipeline the packet is always forwarded to the Egress Policy engine; the decision to look up the EACL table is based on whether the EACL lookup is enabled/ disabled. If egress traffic filtering is not required, then disabling the EACL table lookup reduces the latency in the pipeline.

EACL supports a single lookup type, which is applied on all interface types by default. The key for the lookup can be formed based on the type of traffic to be matched.

The rules in the ACL entries are referred to by index. The index values are order dependent; the first index hit will be returned.

EACL Data

There are a predefined EACL data. EACL data defines a set of attributes that are applied on the flow in case of a successful match on a corresponding rule. Examples of fields are like DSCP, PCP, and EXP fields.

3.3.3.2 Decision Logic

Egress Policy Engine

The Egress Policy Engine receives the token from any of the previous engines. The tokens are marked while processing the packet from the previous engine. The Key fields to be looked at are formed and the lookup launched into the Search Engine. The Key builds and result data analysis are the main functions of this engine.

Lookup: build the KEY based on IPv4 or IPv6 packet template. There is only one lookup type in EACL

For regular IPv4 packet, extract from packet to search the following fields:

SIP, DIP, Protocol, DSCP, PCP, EXP, Tunnel ID, and egress BD, as well as eVIF and reason code from token.

If packet is marked as Layer 2 packet, extract MAC SA,MAC DA, and VLAN.
If packet is marked as Layer 3 packet, extract MAC DA only.
If packet has TCP layer, then extract TCP Source and Destination port.
If packet has UDP layer, then extract UDP Source and Destination port.
If packet is ICMP, then extract ICMP message type.

If packet is marked as Ethernet packet, then extract egress BD.
Perform SE lookup.

The same operation is done for an IPv6 packet.

Token:

The Egress ACL action performed is based on which flags were set, and copy those fields.

The fields that are copied are DSCP, PCP and EXP. For each of the fields, the rewrite pointers are updated to the correct offset.

The above logic of operation is done for IPv6 packet handling as well, with lookup on IPv6 Lookup Table.

For Unicast routed packet where the next engine is not URW, then this is re-injected packet. Extract Source MAC and Egress BD. If NAT is enabled, then get NAT-ed SIP.

Update the rewrite pointers.

If Egress BD Table is not hit or if debug flag is set, the packet is set to TRAP to CPU. If debug flag is not set, the packet is simple DROPPED. The reason code is updated to BD Table miss.

3.4 Layer 2

Conceptually, an L2 domain (bridging domain) is a set of logical interface(s), configured with the same broadcast and flood behavior. A logical interface represents a physical interface and its encapsulation. An encapsulation represents the type of frames accepted or sent out of the physical interface.

The Layer 2 domain processing uses the outermost packet VLAN ID assignment done by the Parser block prior to the Bridge Engine. For instance, in the case of tunnel-terminated packets, the L2 domain will use the passenger VLAN ID as the VLAN ID. and in the case of tunnel-started packets, the tunnel header VLAN ID.

3.4.1 Layer 2 Bridging

3.4.1.1 Description

The XPliant Software-Defined Platform maintains a forwarding table that stores MAC addresses with associated attributes (B.3.1.8 FDB Table).

For unicast bridging purposes, the Bridge Engine performs up to two lookups in parallel to the same FDB table (B.3.1.8 FDB Table): MAC SA lookup and MAC DA lookup.

NOTE: For tunnel-terminated packets, the lookup and the checks are performed on the inner passenger packet.

- **MAC SA Lookup**

  MAC SA lookup is primarily performed to track the IVIF at which the packet arrived; i.e., MAC SA learning.

  MAC SA lookup can be enabled or disabled per bridge domain.

  Unknown MAC SA packet commands can be configured per bridge domain to one of the supported packet commands listed below (FDB Lookup Packet Commands). Packets arriving with unknown MAC SA addresses may optionally be learned.

  MAC SA learning is performed by opening a mirroring session, delivering the packet to the host CPU with the addition of a special XPHeader (Appendix D, XP Header) to carry the required information so the application can analyze the MAC SA of the IVIF and re-install it to the FDB.

- **MAC DA Lookup**

  MAC DA lookup is primarily performed to identify the corresponding EVIF to determine the egress destination of the packet with that particular MAC DA.

For unknown or to-be-flooded flows, various packet commands can be assigned per bridge domain per forwarding packet type:

- Unregistered multicast DA (MAC DA miss and MAC DA type is multicast).
- Unknown unicast DA (MAC DA miss and MAC DA type is unicast).
- Broadcast DA.

NOTE: In the Profile Software-Defined logic, in the case of a MAC DA lookup miss, a packet command assignment for all the mentioned above packet types is collapsed into the < broadcastCmd> field of the Bridge Domain table (B.4.1 BD Table).

Control MAC addresses can be marked with a flag in the FDB table for special processing of various control protocols with these special MAC DA addresses; for example, BPDU frames.

The Platform Software-Defined Logic is able to identify some well-known control protocols, such as ARP, ICMP, or IGMP, allowing their subsequent actions to be configured by bridge packet command assignment.

In addition to the forwarding decision, the Bridge Engine can trigger counting, policing, or sampling per Bridge Domain per MAC DA and/or open a mirroring session.

FDB Lookup Packet Commands

If there is no match for the FDB source or destination lookup, the implicit command is *Forward as per other lookup results*; however, the Platform supports the following configurable packet commands:

- Drop
- Trap the packet to CPU
- Forward as per other lookup results
- Forward as per other lookup results and send a copy to CPU

Resolution of the final command based on the source and destination commands is according to the command resolution rules.

3.4.1.2 Software-Defined Implementation

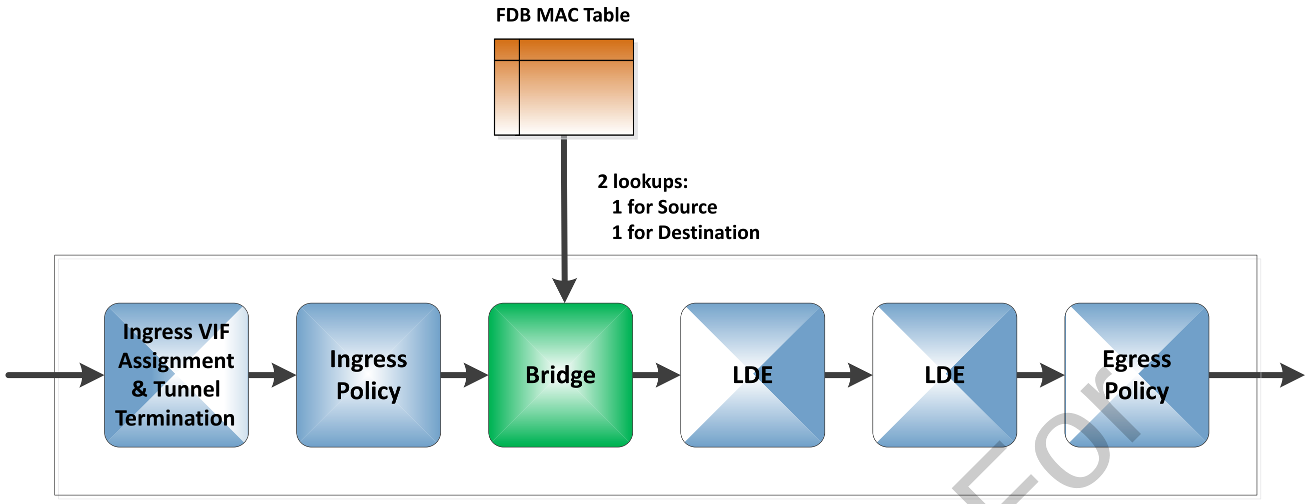


Figure 3–7 Layer 2 Bridging Pipeline View

Resources Usage

Table 3–5 Layer 2 Bridging Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Bridge	2: Shared with the FDB MAC SA and DA lookups	FDB Table	FDB entry attributes	

3.4.1.3 Decision Logic

Bridge Engine—Receives the packet-related information and the corresponding bridge domain attributes as an input from previous engines and implements the unicast bridging logic.

This engine will, for valid and eligible-for-processing packets:

Lookup—For valid packets, FDB lookups can be performed as described below, per an existing and valid bridge domain, which enables isolation between hosts belonging to different bridge domains (i.e., MAC addresses must be unique within a given bridge domain but can be the same across different bridge domains).

- Always perform a non-regrettable **MAC DA lookup** to the FDB table with the key {MAC DA, BD} for
  - valid IPv4 packets with DIP as multicast and Bridge Domain table<`Ipv4McBridgeMode`> configured to BRIDGE_MC_MODE_FDB *or*
  - valid Ethernet packets that are not enabled/eligible for the IP Multicast Bridging (3.4.2 IP-Based Multicast Bridging).
- Perform **MAC SA lookup** to the FDB table with the key {MAC SA, BD} only for valid Ethernet packets that are not enabled/eligible for IP multicast bridging (3.4.2 IP-Based Multicast Bridging) and when the lookup is enabled in the Bridge Domain table <MacSAMode>.

- If the Bridge Domain table <MacSaMissCmd> is configured to Trap or to Drop, the MAC SA lookup will be non-regrettable; i.e., guaranteed to be performed. Otherwise,
- Regrettable; i.e., can be skipped in case of oversubscribed bandwidth to access the table.

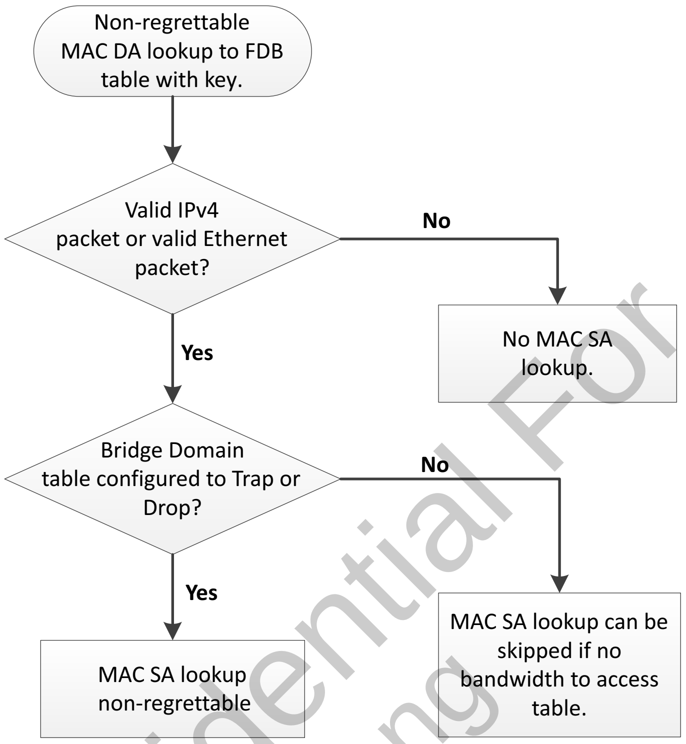

Non-regrettable MAC DA lookup to FDB table with key.
Valid IPv4 packet or valid Ethernet packet?
No
No MAC SA lookup.
Yes
Bridge Domain table configured to Trap or Drop?
No
Yes
MAC SA lookup non-regrettable
MAC SA lookup can be skipped if no bandwidth to access table.

Figure 3–8 Layer 2 Bridging Lookup Logic

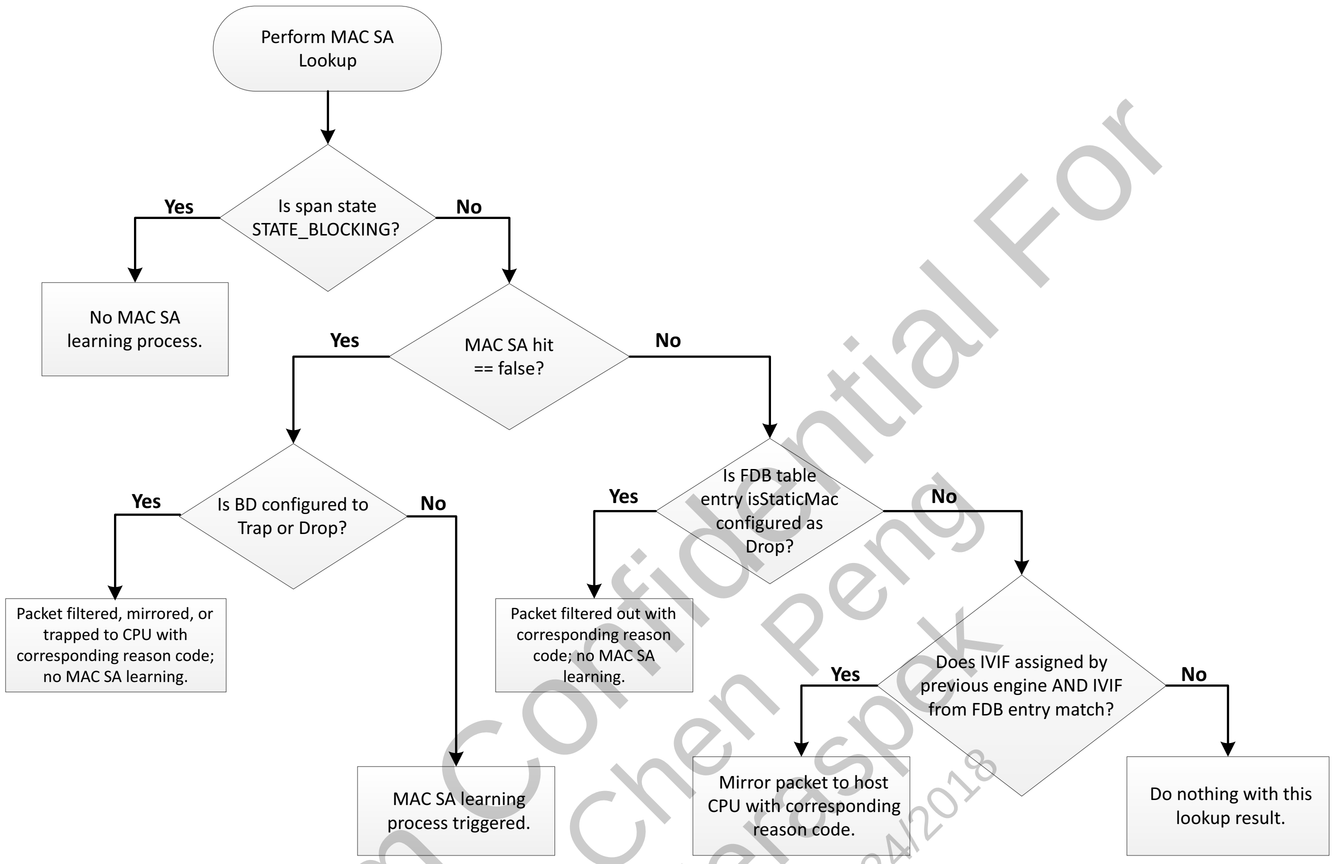
Token—When a lookup has been performed, and a packet is Layer 2 bridged non-routed or a packet with MAC DA that is not Route2Me (for more details refer to 3.5.1.3 Route Table Lookup and 3.6 NAT) and also is not L2 tunneled (L2_MAC_OVER_PORT, L2_MAC_OVER_MPLS_TUNNEL:

- MAC SA Lookup
  - If the span state is STATE_BLOCKING (3.4.6 Spanning Tree), there would be no MAC SA learning process triggered, else
  - If there is a miss:
    - → If the Bridge Domain <MacSaMissCmd> is configured to Trap or to Drop, the packet can be optionally filtered out, mirrored, or trapped to the CPU with corresponding reason code (<XP_BRIDGE_RC_IVIF_SA_MISS> in A.1 Default Profile Reason Codes). In this case no MAC SA learning process will be triggered. Else
    - → MAC SA learning process is triggered: the packet will be mirrored to the host CPU with the corresponding reason code (<XP_BRIDGE_MAC_SA_NEW> in A.1 Default Profile Reason Codes).
  - If there is a hit:
    - → If the FDB table static entry <isStaticMac> is configured with the <pktCmd> as Drop, the packet will be filtered out with corresponding reason code (<XP_BRIDGE_RC_FDB_SA_CMD> in A.1 Default Profile

Reason Codes). In this case, no MAC SA learning process will be triggered. Else

→ If the incoming tokeniVif assigned by the previous engine (3.1.3 Ingress VIF Assignment and Tunnel Termination Engine) and the IVIF extracted from the non-static FDB entry `<vif>` do not match, it indicates a detection of a station movement. This triggers a notification to the application. The packet will be mirrored to the host CPU with the corresponding reason code (`<XP_BRIDGE_MAC_SA_MOVE>` in A.1 Default Profile Reason Codes).

– Else do nothing with this lookup result.



Perform MAC SA Lookup
Is span state STATE_BLOCKING?
Yes
No
No MAC SA learning process.
MAC SA hit == false?
Yes
No
Is BD configured to Trap or Drop?
Yes
No
Packet filtered, mirrored, or trapped to CPU with corresponding reason code; no MAC SA learning.
MAC SA learning process triggered.
Is FDB table entry isStaticMac configured as Drop?
Yes
No
Packet filtered out with corresponding reason code; no MAC SA learning.
Does IVIF assigned by previous engine AND IVIF from FDB entry match?
Yes
No
Mirror packet to host CPU with corresponding reason code.
Do nothing with this lookup result.

Figure 3–9 Layer 2 Bridging MAC SA Lookup Logic

❍ MAC DA Lookup

– If there is a hit AND the FDB table entry `<macDAIsControl>` is configured as Control, while the span state is NOT `STATE_DISABLED`, the packet would be trapped to the host CPU as per the Spanning Tree protocol logic (3.4.6 Spanning Tree) with the corresponding reason code (`<XP_IVIF_RC_BPDU>` in A.1 Default Profile Reason Codes), else

– If the span state is `STATE_LEARNING` or `STATE_BLOCKING`, the packet will be dropped with the corresponding reason code (`<XP_BRIDGE_RC_IVIF_SPAN_BLOCKED>` in A.1 Default Profile Reason Codes), else

❍ If there is a hit, then

→ If routerMac, jump to the Route Engine; otherwise, jump to the Egress Policy Engine (since the packet is not L3 Routable or NAT enabled, see the note below),

→ extract from the FDB Entry `<pktCmd>`, `<VIF>`, and `<ecmpSize>` and

→ assign the Egress Filter ID `<EgressFilterID >`

– If the FDB entry is configured with the `<pktCmd>` as Trap or Drop, the packet can optionally be trapped or filtered out with corresponding reason code (`<XP_BRIDGE_RC_FDB_DA_CMD>` in A.1 Default Profile Reason Codes).

❍ Else when there is a miss, then

→ jump to the next functional for these packets engine, Egress Policy engine.

→ Assign Egress VIF with the Bridge Domain table `<FloodVif>`

→ Packet Command with the Bridge Domain table `<EgressBcCmd>` and if it is Trap or Drop, the packet can be optionally trapped or filtered out with corresponding reason code (`<XP_BRIDGE_RC_FLOOD_CMD>` in A.1 Default Profile Reason Codes).

→ assign the Egress Filter ID with the Bridge Domain Table `<EgressFilterID>`

NOTE:

- Please refer to the L2-tunneled packet bridging processing in the corresponding sections (3.7 Tunnel).
- Please refer to the to-be-routed or NAT-eligible-packet bridging processing in the corresponding sections (3.5.1.3 Route Table Lookup and 3.6 NAT) and also is-not-L2-tunneled (L2_MAC_OVER_PORT, L2_MAC_OVER_MPLS_TUNNEL).

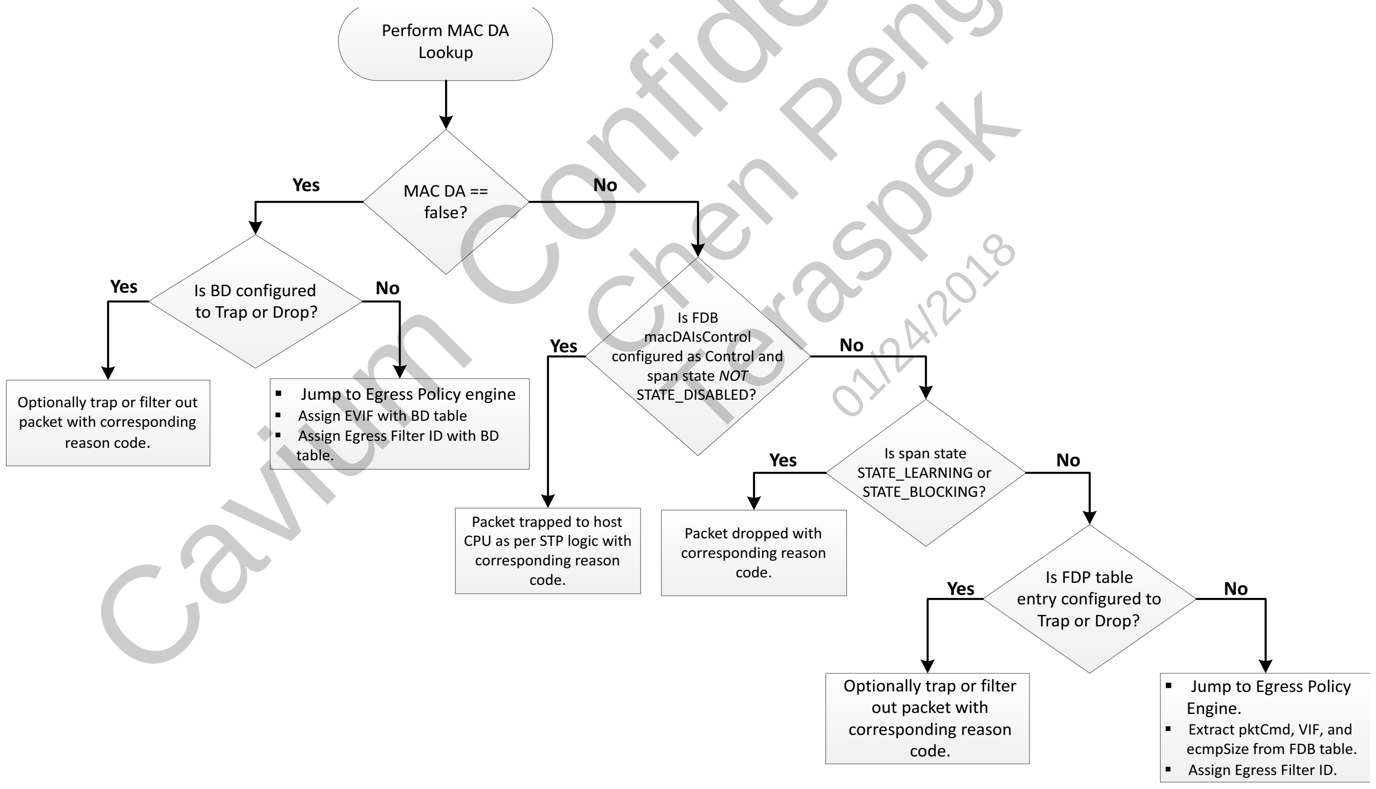

Perform MAC DA Lookup
MAC DA == false?
Yes
No
Is BD configured to Trap or Drop?
Yes
No
Optionally trap or filter out packet with corresponding reason code.
Jump to Egress Policy engine
Assign EVIF with BD table
Assign Egress Filter ID with BD table.
Is FDB macDAIsControl configured as Control and span state NOT STATE_DISABLED?
Yes
No
Packet trapped to host CPU as per STP logic with corresponding reason code.
Is span state STATE_LEARNING or STATE_BLOCKING?
Yes
No
Packet dropped with corresponding reason code.
Is FDP table entry configured to Trap or Drop?
Yes
No
Optionally trap or filter out packet with corresponding reason code.
Jump to Egress Policy Engine.
Extract pktCmd, VIF, and ecmpSize from FDB table.
Assign Egress Filter ID.

Figure 3–10 Layer 2 Bridging MAC DA Lookup Logic

3.4.2 IP-Based Multicast Bridging

3.4.2.1 Description

The XPliant Software-Defined Platform maintains a forwarding table that stores multicast addresses with associated attributes (B.3.1.9 IPv4 Multicast Bridge Table).

The IP multicast entries contain an IP multicast address that is associated with a multicast group.

For multicast bridging purposes, the Bridge Engine performs a single lookup in the same Multicast Bridging table (B.3.1.9 IPv4 Multicast Bridge Table) to get the EVIF used to extract the forwarding destination.

The SA lookup is not performed in this mode, since typically the multicast entries are installed to the Multicast Bridging table (B.3.1.9 IPv4 Multicast Bridge Table) by the host CPU and are not subject to learning or aging (i.e., static entries)

NOTE: For tunnel-terminated packets, the lookup and the checks are performed on the inner passenger packet.

3.4.2.2 Software-Defined Implementation

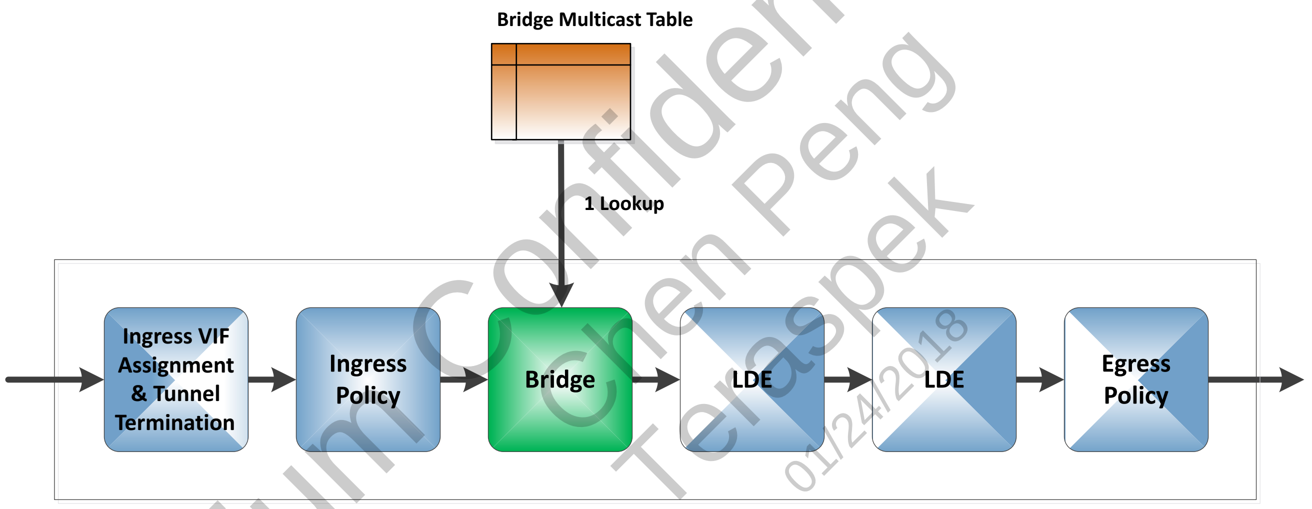


Figure 3–11 IP Multicast Bridging Pipeline View

Resources Usage

Table 3–6 IP Multicast Bridging Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Bridge	1	Multicast Bridging table	Multicast bridging entry attributes	

3.4.2.3 Decision Logic

Bridge Engine—Receives the packet-related information and the corresponding bridge domain attributes as an input from previous engines and implements the multicast bridging logic.

This engine will, for valid and eligible for processing packets,

Lookup—For valid IPvX packets with DIP as multicast that belong to a valid and existing bridge domain, if the Bridge Domain table<Ipv4McBridgeMode> is configured to (BRIDGE_MC_MODE_S_G or to BRIDGE_MC_MODE_0_G), then a non-regrettable (i.e., guaranteed) lookup to the IP Multicast Bridging table (B.3.1.9 IPv4 Multicast Bridge Table) can be performed.

Token—When a lookup has been performed on a valid IPvX multicast packet:

- If there is a hit, then
  - extract from the Multicast Bridging entry <`pktCmd`> and <`VIF`> and
  - set the reason code to <`XP_BRIDGE_RC_MC_BRIDGE_CMD`> (A.1 Default Profile Reason Codes)
- Else when there is a miss, then
  - extract from the Bridge Domain Entry <`EgressUnRegMcCmd`> and <`FloodVif`> and
  - set the reason code to <`XP_BRIDGE_RC_MC_UNREG_CMD`> (A.1 Default Profile Reason Codes)

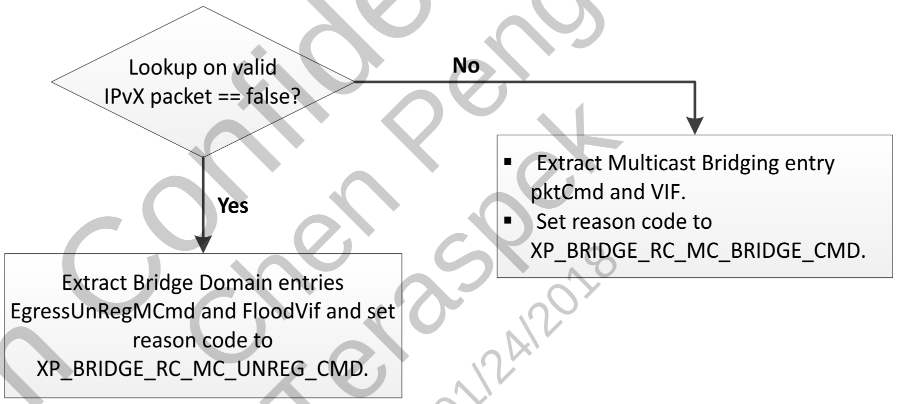

Lookup on valid IPvX packet == false?
No
Yes
Extract Multicast Bridging entry pktCmd and VIF.
Set reason code to XP_BRIDGE_RC_MC_BRIDGE_CMD.
Extract Bridge Domain entries EgressUnRegMCmd and FloodVif and set reason code to XP_BRIDGE_RC_MC_UNREG_CMD.

Figure 3–12 IP Multicast Bridging Decision Logic

NOTE: If multicast bridging is disabled, then

- extract from the Bridge Domain entry <`EgressBcCmd`> and <`FloodVif`> and
- set the reason code to <`XP_BRIDGE_RC_FLOOD_CMD` > (A.1 Default Profile Reason Codes)

3.4.3 Layer 2 FDB MAC Management

3.4.3.1 Description

The XPliant Software-Defined Bridge Engine supports automatic or controlled learning of new or moved MAC source addresses. The Bridge Engine supports FDB MAC movement and/or MAC SA learning automatically or controlled by the CPU, or both, while an application can select the desirable option.

In addition, a host CPU can add, delete, or modify any entry in the FDB table.

MAC SA Learning and MAC Movement

When MAC learning is enabled and a new packet with an unknown MAC SA arrives or a MAC movement event occurs, the software-defined Bridge Engine will create a special packet to be sent to the host CPU with all information relevant to the event in an XPH header (Appendix D, XP Header).

Reason codes are assigned to each CPU-destined packet and can be configured in a reason code table (Appendix A, Reason Codes). The table provides the flexibility of mapping a reason code to a specific traffic class, which in turn can be mapped to one of the CPU DMA queues, which can then be individually throttled or shaped.

To perform CPU-controlled MAC learning or MAC movement, the host CPU, upon processing of the received frame, may issue an update to the FDB table with the new MAC address, the outgoing interface information, and other associated MAC attributes.

MAC Aging

MAC addresses stored in the FDB are subject to an aging mechanism. Aging is a mechanism where FDB entries are examined based on configurable criteria, which are invoked periodically according to a configurable time interval. If an address remains inactive a specific time or longer, it is removed from the FDB table. Aging can be controlled on a per entry basis.

MAC addresses can also be set in the FDB to static mode, in which case they will not be subject to the removal process. However, static entries can optionally be subject to the FDB delete and transplant mechanisms.

3.4.4 Other FDB Functionality

A MAC entry includes some additional attributes that are used in various features beyond L2 processing. These attributes are described in other sections, such as <ECMP Size> for Layer 3 traffic <isTunnelSpoke> used for the source suppression.

3.4.5 Layer 2 LAGs

3.4.5.1 Description

A LAG (Link Aggregation Group) is a method to aggregate traffic over multiple physical Ethernet links into a single, logical link. A LAG is defined by the IEEE 802.1AX-2008 standard.

In the XPliant Software-Defined Platform, a LAG is represented by a VIF of a special LAG type.

All member ports of the LAG must have identical feature attributes, including, but not limited to, VLAN membership, 802.1Q encapsulation type, tunnel memberships, etc. A LAG load-balances the traffic across its physical links, effectively providing improved throughput. In addition to increased bandwidth, a LAG provides improved resiliency. When a port member of a LAG has a link failure, the LAG bandwidth may be reduced, but traffic continues to pass through the remaining port members of the LAG.

The XPliant Software-Defined Platform performs a LAG hash computation on each packet, which is used to select one of the egress LAG members for packet transmit. The distribution of LAG members is represented in the Platform through the programming of the Trunk Resolution table (TRT). This table contains a bit map of all ports on a device (there are multiple tables in a system). The TRT is used to resolve LAG and ECMP paths, while the IVIF and the EVIF lookup results are bitwise ANDed with the TRT selected by the hash entry. Each TRT entry contains only a single bit unset per LAG for the selected entry; ports that are not part of the trunk have their corresponding bit set. For a given trunk of size N, there will be N lines in the table for that trunk, each with only one bit set for the trunk ports.

The LAG member selection is based on a module function. For instance, in case of two ports LAG, the table is programmed as follows:

```
*    - 1 0 1 1 1 1 1 ... 1
*    - 0 1 1 1 1 1 1 ... 1
*    - 1 0 1 1 1 1 1 ... 1
*    - 0 1 1 1 1 1 1 ... 1
*    - etc...
```

3.4.6 Spanning Tree

3.4.6.1 Description

The XPliant Software-Defined Platform supports enforcement of the Spanning Tree Protocol (STP), conforming to IEEE 802.1q, on a per port-VLAN basis.

On the ingress path, STP state is configured as a property of an ingress virtual interface to be one of the following states: Disabled, Learning, Forwarding or Blocking.

According to the software-defined logic of the profile, the STP logic decisions are based on the packet parsed type and the configured corresponding STP state:

- Disabled—The feature is not activated.
- Blocking—No user data is forwarded and/or learned; the control DA traffic is trapped to host CPU.
- Learning—No user data is forwarded, but MAC addresses can be populated for learning; the control DA traffic is trapped to host CPU.
- Forwarding—User data can be forwarded and/or learned; the control DA traffic is trapped to host CPU.

On the egress path, the STP state of blocking/non-blocking is achieved through Group ID-based filtering. For the data traffic, the application is expected to reflect the STP state of egress physical ports in the ITT <egressPortFilterID> if controlled on the port level or in the Egress BD Table <egressPortFilterID> if controlled on the bridge domain level (B.3.1.21 Egress Bridge Domain Table). For control packets generated by the CPU that are not to be blocked due to a span state, it is expected from the Application to set the <EgressPortFilter> field in the XPH Header (Appendix D, XP Header) accordingly.

NOTE: The recommended implementation for the transparent control traffic that is expected to traverse through the Platform unblocked by the Span state: it is recommended to trap these control frames to the host CPU, analyze, validate and re-inject them back to the Platform with the corresponding distribution list reflected through the XPH Header metadata field <egressFilterId> (Appendix D, XP Header).

3.4.6.2 Software-Defined Implementation

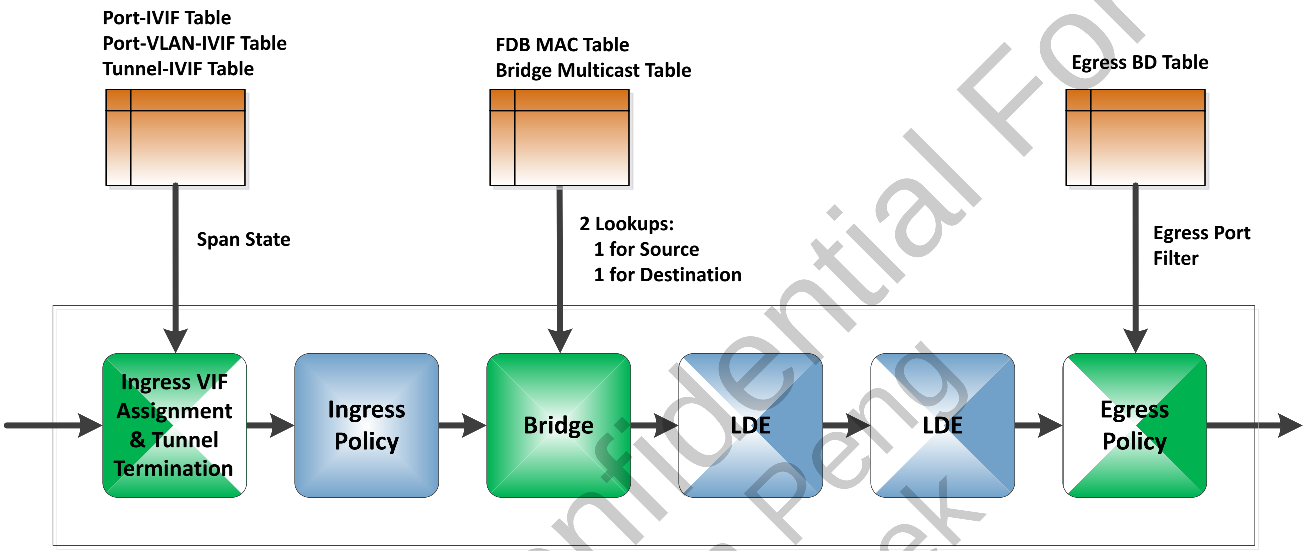


Figure 3–13 Spanning Tree Pipeline View

Resource Usage

Table 3–7 Spanning Tree Pipeline Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Ingress VIF Assignment and Tunnel Termination	1. Shared with the general IVIF lookup	Port-IVIF *or* Port-VLAN-IVIF *or* Tunnel-IVIF	Get the span state	
Bridge	2. Shared with the FDB MAC SA and DA lookups	Both lookups to the FDB table	MAC entry attributes	
Egress Policy	1	Egress Bridge Domain	Get the egress port filter ID.	

3.4.6.3 Decision Logic

Ingress VIF Assignment and Tunnel Termination—Fetches the span state from a corresponding table.

This engine will, for valid and eligible for processing packets:

Lookup—The lookup is not dedicated to the feature, but always performed in this engine with the key and to the tables in accordance with the IVIF assignment SE logic.

Token—Copy the span state received from a lookup to the outgoing token.

Bridge Engine—Receives the span state as an input from previous engines and implements the spanning tree logic.

This engine will, for valid and eligible for processing packets:

Lookup—There are two lookups not dedicated to the feature that are performed in this engine with the key to the tables in accordance with the bridge SE logic.

Token—

1. **Lookup0:** *MAC SA Lookup results-based processing* is performed for MAC SA learning purposes.
   - If the packet is eligible for MAC SA learning (see 3.4.1 Layer 2 Bridging for more details) and the span state is NOT set to blocking, its MAC SA may be learned.
   - Else do not perform MAC SA learning.
2. **Lookup 1:** *FDB MAC DA Lookup* is performed for resolving the forwarding destination egress interface.
   - If the span state of the IVIF is set to Disabled, then proceed with the normal operation.
   - Else,
     - → if there is a hit on the MAC DA in the FDB table and the MAC DA has been configured as of the control type, then mark the packet to be trapped with the corresponding reason code and skip the forwarding processing by jumping to the Update-Rewrite engine at the end of the forwarding processing pipe
     - → else if the span state of the IVIF is set to Blocking or to Learning, then mark the packet to be dropped with the corresponding reason code and skip the forwarding processing by jumping to the update-rewrite engine at the end of the forwarding processing pipe
     - → else proceed with the normal operation

Egress Policy Engine—Receives the Egress Bridge Domain as an input from previous engines and implements the spanning tree logic.

This engine will, for valid and eligible for processing packets:

Lookup—The lookup is not dedicated to the feature, but performed in this engine with the Egress BD key and to the Egress BD table in accordance with the SE logic.

Token—If there is a hit on Egress BD, the egressPortFilterId is programmed into the token. If mirrorEn is set, the token.mirrorBitMask is set to the specific mirrorId.

3.4.7 802.1Q/D VLAN-IDs Bridging

3.4.7.1 Description

The XPliant Software-Defined Platform is capable of supporting standard IEEE 802.1Q/D bridging features, including VLAN-unaware, VLAN-aware, and priority-tagged VLAN, as well as proprietary software-defined extensions.

The IVIF and bridge domain concepts are instrumental for implementing 802.1Q/D VLAN IDs bridging.

Per the IEEE 802.1Q specification, the XPliant Platform can support the following ingress checking mechanisms.

- Acceptable-frame-type setting per interface, with the following options:
  - Admit only VLAN-tagged frames.
  - Admit only untagged and priority-tagged frams.
  - Admit all frames.

  Ingress VLAN port acceptance and membership functionality is checked through a hit in the Bridge Domain table (B.4.1 BD Table), based on the combination of a port default IVIF and the VLAN ID (which may be the packet original VLAN ID (outermost VLAN in case of tunnels or other encapsulations)). If there is a miss in the Bridge Domain table (B.4.1 BD Table), it implies compliance with the port-VLAN acceptance condition and will be discarded with the corresponding reason code <XP_BRIDGE_BD_TABLE_MISS>.

- VLAN-aware and VLAN-unaware modes.

  In a VLAN-unaware mode, no VLAN modifications or checks are performed within the processing device for either ingress or egress packets. This means packets received tagged are transmitted tagged; packets received untagged are transmitted untagged.

  While operating in this mode, it is recommended to use the Initial Token Table configuration (B.1 Initial Token Table (ITT)), which allows either selecting between preserving the incoming original VLAN ID or always overriding it (<pvidModeAllPkt> in B.1 Initial Token Table (ITT)). Overriding the incoming VLAN IDs for all traffic effectively ignores the ingress VLAN if it existed in the incoming packet header. Maintaining the same VLAN state within a given bridge domain allows an application to implement the VLAN-unaware mode, where the VLAN tag will remain (i.e., not be added, removed, or modified) from the egress outgoing VLAN perspective as well.

  In a VLAN-aware mode, VLAN attributes can be manipulated to achieve Layer 2 bridging domain functionality, based on virtual LAN standard mechanisms and best practices to partition a single Layer 2 bridging domain into multiple, separate, independent bridging domains. This is achieved through the normal operation of the device. Every interface added to an L2 domain with a VLAN ID generates a specific port VLAN index. Every created port-VLAN interface identifies unique attributes for the interface, which can be configured using specific accessors for each attribute.

3.4.8 Provider Bridge 802.1ad (Q-in-Q)

3.4.8.1 Description

The XPliant Software-Defined Platform supports provider bridge 802.1ad functionality.

802.1ad, informally known as 802.1Q tunneling or 802.1QinQ, expands the VLAN space by using a VLAN-in-VLAN hierarchy and adding an 802.1Q tag to existing Ethernet frames. Ingress Ethernet frames may or may not have the first 802.1Q tag, and in some cases may loosely refer to VLAN stacking.

The XPliant Platform implements the functionality as described below.

1. A port configured to support 802.1Q tunnel can be a tagged port or untagged port dedicated as a tunnel port.
   - Can be a customer-edge port.
   - Can be a service-edge port.
   - Can be a core (transient) port .
2. Assign the native VLAN (which becomes the S-VLAN) on the tunnel port which is dedicated for tunneling.
3. This port can be configured to receive both untagged and tagged frames per the Application setting.
4. Each customer-edge device is connected to a tunnel port and is configured to use the S-VLAN (Service VLAN).
5. A link between a customer device and a service provider device is an asymmetric link because VLAN configured on one end is the customer's VLAN and the VLAN configured on the other end is a service provider's VLAN.

The XPliant Software-Defined Platform may support, for instance, the following mappings:

- One C-VLAN to one S-VLAN.
- One C-VLAN to multiple S-VLAN.
- Limit a set of customer tags to a range of tags or to discrete values.
- All customer-edge interfaces are access interfaces and can accept tagged and untagged packets.

3.4.8.2 Software-Defined Implementation

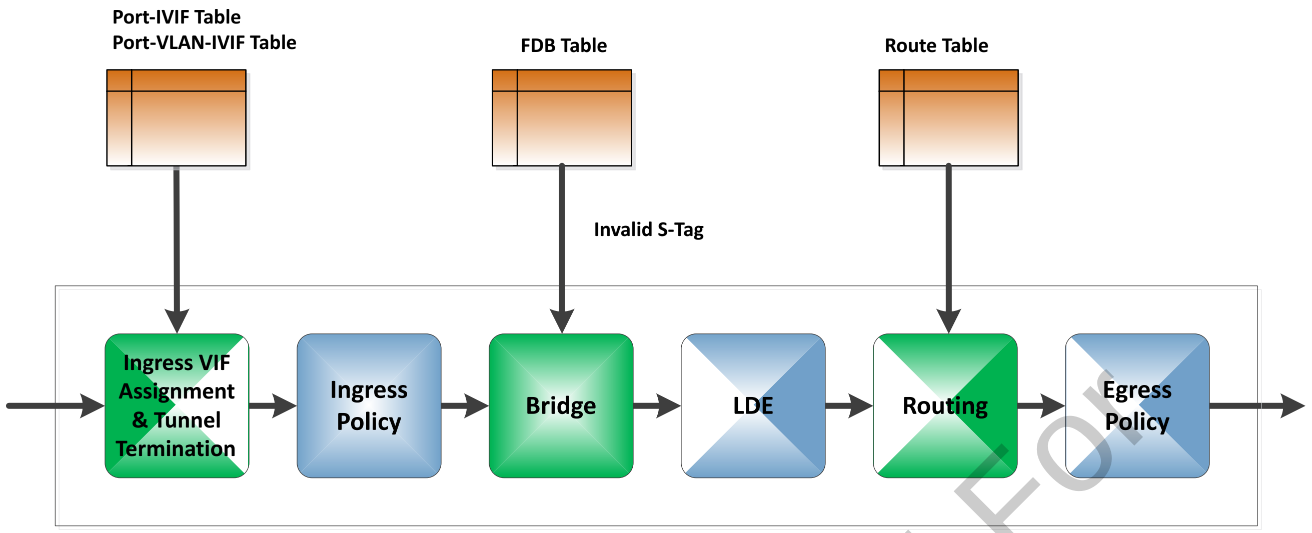


Figure 3–14 Q-in-Q Pipeline View

Resources Usage

Table 3–8 Q-in-Q Tunnel Termination Resources Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Ingress VIF Assignment and Tunnel Termination	**1.** portVlan	PortVlanIvifTable		
Bridge	**2.** Shared with the FDB MAC SA and DA lookups	Both lookups to the FDB table	MAC entry attributes	
Routing	Routing lookup			

3.4.8.3 Decision Logic

- In general, Q-in-Q tunnel origination is a port configuration or can be a port VLAN configuration (ingressVif-port-vlan).
- If all C-VLANs are mapped to S-VLANs, the ingress ports (tunnel ports) should be classified as native S-VLAN interfaces, and the ingressVif-port table can provide BD ID and a Q-in-Q configuration bit/status. When the port-VLAN lookup fails in the IVIF LDE portConfig tables, the BD is selected.
- For the ingressVif tunnel table, the Q-in-Q lookup key is S-VLAN, C-VLAN, and IVIF. The application is expected to configure all C-VLAN and S-VLAN configurations for all Q-in-Q tunnels.

Tunnel Origination

- Based on the ingress port (being configured as a tunnel port), all packets should be accepted on this port (i.e., untagged and tagged) and the ITT table (B.1 Initial Token Table (ITT)) provides a configuration to override the BD ID and configure this interface for a Q-in-Q tunnel. This will facilitate the application to configure a dedicated port to perform an S-VLAN based Q-in-Q tunnel.

- If the application configures any ingress port to be a 1Q tunnel port for more than one S-VLAN, it uses a configuration in the portVlan IVIF, with the range of C-VLANs mapping to the S-VLAN required when there is a many (C-VLAN) mapping to one (S-VLAN). BD ID is looked up by the portVlan IVIF lookup and the S-VLAN for egress encapsulation is provided from the FDB entry. The FDB entry incorporates an encapsulation type as QinQ_Encap when it adds an S-VLAN when the packet is egressing out. The MAC learning for the FDB resolves the egress port to be provider-port or the PB (provider bridge) port configured, and the encapsulation type adds the 1Q tag.

Tunnel Termination

- FDB lookup is performed on the S-VLAN of the ingress packet and associated BD.
- FDB control data carries a modification pointer to the encapType "QinQ_Decap", which invalidates the S-TAG in the layer data and sends the packet out with only the C-VLAN header. The MAC learning for the FDB resolves the egress port to be a Q-in-Q-tunnel port, and the application can program appropriate encapsulation in the header.

Switching and Routing

- When performing a BD lookup, the BD ID is based only on the S-VLAN; the C-VLAN is not used. The C-VLAN is typically used for PB configurations, whereas only the S-VLAN is used for MAC lookups. Based on the packetType being QinQ in the port+Vlan Ingress VIF table (B.3.1.1 Port VLAN Table), the S-VLAN and IVIF lookup are performed, proceeding with the bridging as normal on the S-VLAN BD.
- In the switching case, there is no change in VLAN tags and only the destination port is looked up from the MAC table.
- In the routing case, the Next Hop entryprovides the S-VLAN to be inserted based on the portVif table (B.3.1.1 Port VLAN Table.

3.5 Layer3

The Layer 3 feature is implemented in the Routing Engine. Some profiles, such as the 2.4 Bpps Default Pipeline, use more than one LDE for better load balancing of the traffic eligible for routing.

Layer 3 supports routing of IPv4, IPv6, MPLS, and Multicast for outer IP and inner IP for tunnel-terminated packets. It is VRF aware and supports L3 ECMP. The Parser Engine, based on the Control MAC table lookup for the router MAC, will qualify the packet to be routed. Routing must be enabled on the bridge domain of the ingress interface or tunnel interface.

This section describes only IPv4 and IPv6 routing. Both host route lookup and IP route lookup are supported. Lookups are launched in parallel; when both have a hit, the host route result is preferred over the IP route lookup.

Although Multicast and MPLS are handled in the same Routing Engine, they are described in detail in separate sections.

3.5.1 Description

3.5.1.1 L3 Interfaces and L3 Subinterfaces

L3 interfaces are used to route traffic between VLAN isolated domains.

A Layer 3 subinterface is a logical division of a VLAN-based interface that is set to receive and forward 802.1Q VLAN tags.

Layer 3 subinterfaces enable sharing common interfaces while each VLAN ID is bound to the logical partitions; i.e., it supports partitioning into up to 4094 different L3 subinterfaces, one for each VLAN. In this way Layer 3 subinterfaces can be used to route traffic among multiple VLANs along a single trunk line that connects two Layer 2 switches, using only one physical connection.

An L3 subinterface is implemented by enabling VLAN-tagging on the interface and extracting egress VLAN ID information from the Next Hop entry.

NOTE:

- Flooding to VLAN of an L3 subinterface is not allowed.
- Sending unregistered multicast, unknown unicast, or broadcast packets to the VLAN of L3 subinterface is not allowed with the exception of an ARP broadcast that is unconditionally trapped to the CPU.

3.5.1.2 Host Table Lookup

The Host table is used for directly connected routes and implemented as a hash table. The Host table lookup is performed with the key of VRF ID that is configured on the bridge domain and destination IP address. The hit result is the next hop IP address. There is no L3 ECMP support for the next hop address; however, an L2 ECMP can be performed in the Update and Rewrite Engine.

3.5.1.3 Route Table Lookup

The host process using any routing protocol or default route set must perform the Route table population. The Route table is implemented as a longest prefix match (LPM) table. Both IPv4 and IPv6 prefixes are supported on separate LPM tables. The Route table lookup is performed with the key of VRF ID and Destination IP address. The longest prefix length is selected for match. This is achieved through LPM Tables represented as a trie data structure divided into different levels of sub-tries. Each sub-trie contains a set of nodes representing valid prefixes and VRF IDs called leaves. Each leaf has an identifier, and the level of nodes within the sub-trie is termed as the stride.

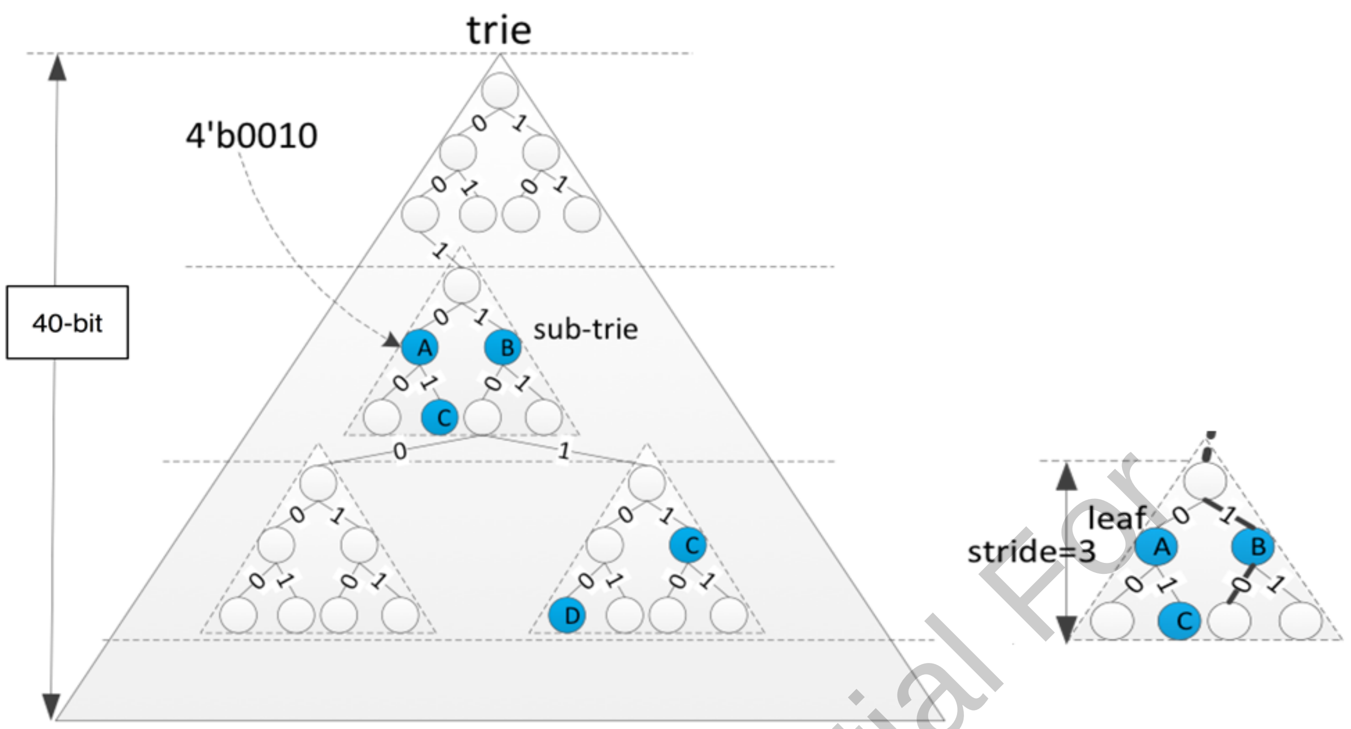


Figure 3–15 IPv4 LPM Trie Structure

This representation is represented in hardware as an entry in one of the 32 prefix tables accessed in parallel.

The Route table hit points to a Next Hop Indirection table. This table stores the next hop index and ECMP size. Based on ECMP, the next hop index is calculated and read from the Next Hop table. Through the Next Hop Indirection and Next Hop tables, L3 ECMP is supported.

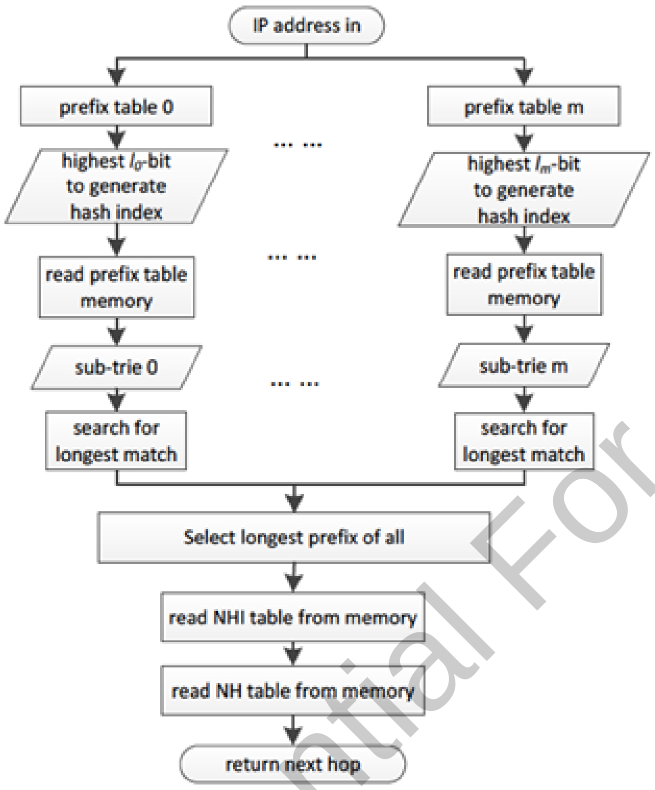


Figure 3–16 IPv4 LPM Flow

3.5.2 Software-Defined Implementation

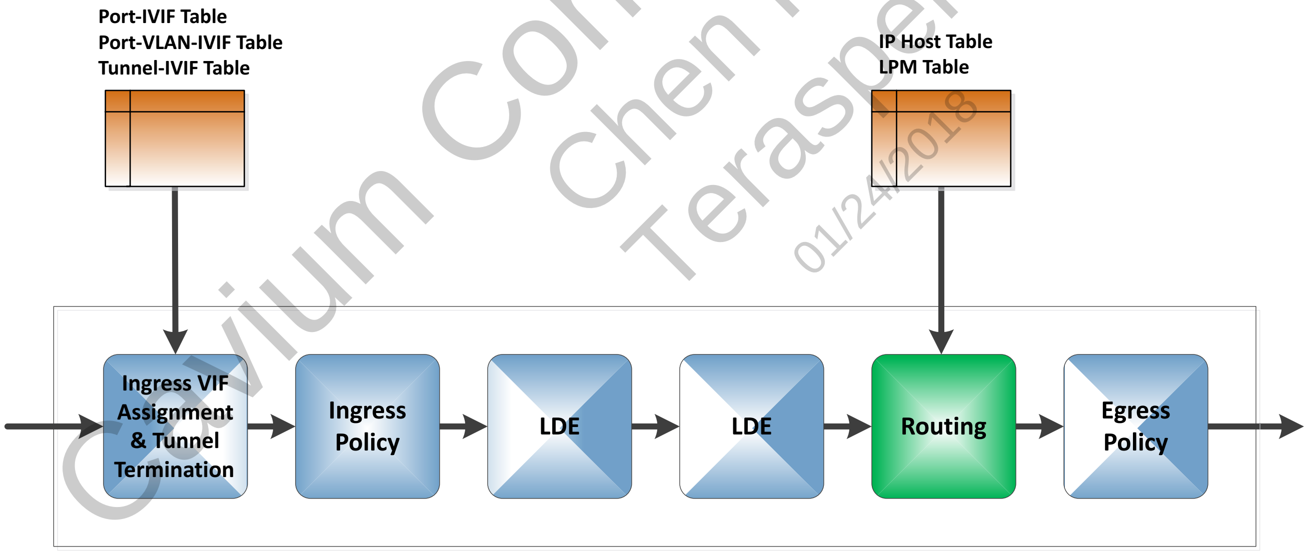


Figure 3–17 Layer 3 Pipeline View

Resource Usage

Table 3–9 Layer 3 Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Routing	1. VRF ID, Dest IP 2. VRF ID, Dest IP	IP HostTable LPM Route table	IP Host entries IP Route entries	No L3 ECMP for IP Host Entries only for Route entries

3.5.3 Decision Logic

Ingress VIF Assignment and Tunnel Termination—Receives packet information from parser and for valid packets:

Lookup—For valid packets, perform lookup to PortVlanIvif table with the key {PORT VIF, OUT VLANID}

- Get Port VIF from ingressVif, which is from the port configuration table from the parser.
- Get the VLAN ID from the ethLayer.

Route Engine—Key build for lookup.

Format the super key based on the packet template ecmpHash = L3_ECMP.

Hash_A

```
if (Tunnel.Terminated and InnerIpv4) OR (NOT Tunnel.Terminated) { Get BD,
    macDA, macSAif ( NOT Tunnel.Terminated and IpV4) OR Tunnel.Terminated and
    InnerIpv4) {
    if Not MultiCastMode and NOT Multicast Packet
       if (Sa.Miss is to send a TRAP or DROP)
       set SEARCH_PROFILE = UcBridgeRouteSaNonRegretSearchProfile
       else
       set SEARCH_PROFILE = UcBridgeRouteSearchProfile
If packet DA is router MAC which is determined by Parser TCAM Lookup
  Perform LPM Lookup with Key = IPV4_UC_HOST_KEY
else
    Perform FDB Lookup with Key = BRIDGE_FDB_KEY_TYPE
```

The above logic is also performed on IPv6 packets.

Token—Unicast process follows.

```
If UnicastHostTable is hit OR UnicastRouteTable is
  hit If (ttl is zero)
    DROP packet with reason code
    nextEngine = URW
if UnicastHostTable is hit
  If packet cmd is TRAP or DROP : set reason code
  Else {
    Copy relevant field into token like evif, ecmp_size , macDA
    Set ether_type = 0x0800

    }
For Both Route Hit or MPLS Transit hit {
based on the encapType and for MPLS
        EtherType = 0x8847
Else    EtherTupe = 0x0800
}
if no Hit TRAP
```

3.5.4 Unicast Reverse Path Forwarding (uRPF)

Unicast Reverse Path Forwarding is supported for both IPv4 and IPv6. uRPF eliminates a malformed or spoofed SIP and discards it if the uRPF check fails. This also limits a router to verify the reachability of the source address in packets being forwarded.

The Unicast RPF implementation supports two different modes: strict mode and loose mode. In strict mode, the packet must be received on the interface that the router uses to forward the return packet. In loose mode, the source address must appear in the routing table.

The following diagram depicts the uRPF flow. The BD table is modified to enable uRPF. If URPF is enabled, then both VRF, DIP and VRF, SIP are performed.

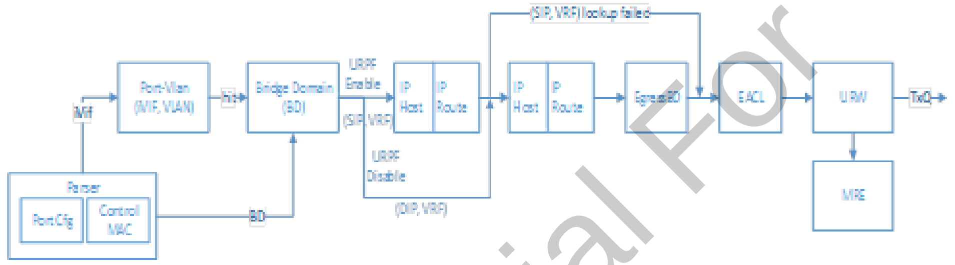


Figure 3–18 uRPF Flow

NOTE:	The uRPF check is not performed if the DIP is not a unicast address.

3.5.5 Reason Code Changes

3.5.5.1 Description

In XDK 3.2, one of the key features implemented is the Reason Code support for L3. The Next Hop Table (NH-Table) has sub-Reason Code support. Correspondingly, the Next Hop entry can now provide a Reason Code. When L3 Route lookup happens, the Reason Code is also one of the fields obtained. L3 packets trapped or dropped will carry this Reason Code.

Summary of changes required for implementation:

- Two of the reserved bits in the NH table data structure are now used for `reasonCode`.
- Updated PL for NH table: added reasonCode variable.
- Subreason code set and get from NH manager.
- Added reason code in `xpReasonCodeTable.h`, `xpReasonCodeTable.cpp`.

Relevant code changes in XDK:

- Created additional reason codes:

  In `xp/dm/devices/include/xpReasonCodeTable.h`:

```
#define XP_ROUTE_RC_NH_SUB_REASON_2            62
#define XP_ROUTE_RC_NH_SUB_REASON_3            63
```

- Added the following reason codes to the default reason code enum/list:

  ```
  xp/dm/devices/xp80/profiles/urw/xpDefaultReasonCodeTable.cpp
  static xpReasonCodeEntryCfg
  reasonCodeDefEntCfg[XP_REASON_CODE_DEFAULT_ENTRIES]
  ```

- In XPS layer, added the reason code to be picked up from the Next Hop table:

  ```
  xdk/xps/xpsL3.c
  static XP_STATUS xpsL3GetNextHopEntry(xpsDevice_t devId,
  xpsL3NextHopEntry_t
  *nhEntry, xpNh_t *nextHopT)
  {
  ….
      nhEntry->reasonCode = nextHopT->reasonCode;
  ..
  }
  ```

Profile changes:

- New reason codes added:

  ```
  Engines/xpReasonCodes.t
  define XP_ROUTE_RC_NH_TABLE_HIT 60
  define XP_ROUTE_RC_NH_SUB_REASON_1 61
  define XP_ROUTE_RC_NH_SUB_REASON_2 62
  define XP_ROUTE_RC_NH_SUB_REASON_3 63
  ```

- IPv4 and IPv6 UC Route table/Next Hop table:

  ```
  Engines/xpTables.t
  FIELD   available1             :2;
  FIELD   rsnCode               :2;
  ```

- Addition of `subReason` code to `baseReason` code is done at Route Engine LDE.

  ```
  Engines/xpIPvxUcastAndMplsRouteEngine.xpc
  ```

  Old code:

  ```
  token.reasonCode = XP_ROUTE_RC_ROUTE_TABLE_HIT;
  ```

  New code:

  ```
  token.reasonCode = ipvxUcRouteTable.rsnCode |
  XP_ROUTE_RC_NH_TABLE_HIT;
  ```

Tests added in AppTest for regression:

Four additional tests have been added in AppTest for both reference and testing purposes:

- `'ipv4rNhRCTc1': 'IPv4NHTable`: TC to verify that the incoming packet gets trapped with appropriate reasonCode | reasonCode : 60.'
- `'ipv4rNhRCTc2': 'IPv4NHTable`: TC to verify that the incoming packet gets trapped with appropriate reasonCode | reasonCode : 61.'
- `'ipv4rNhRCTc3': 'IPv4NHTable`: TC to verify that the incoming packet gets trapped with appropriate reasonCode | reasonCode : 62.'
- `'ipv4rNhRCTc3': 'IPv4NHTable`: TC to verify that the incoming packet gets trapped with appropriate reasonCode | reasonCode : 63.'

3.6 NAT

3.6.1 Description

Network Address Translation (NAT) is a way to map an entire network (or networks) to a single IP address. Typically, NAT operates on a routing device, usually connecting two networks together translating the private address in the internal network into public addresses before packets are forwarded, thus providing IP address conservation and security functions.

NAT is designed to modify or translate network IP address information in packet headers. Either or both source and destination IP addresses in a packet are translated. The XPliant Software-Defined Platform may optionally support PAT (Port Address Translation) as NAT overloading functionality, where as an extension of the standard NAT, the port numbers are translated along with the IP addresses.

In the XPliant Software-Defined implementation, NAT information is stored in the NAT table (B.3.1.17 Table).

NOTE: The NAT feature is only applicable to the 1.2 Bpps profile.

3.6.2 Software-Defined Implementation

3.6.2.1 General Network Address Translation (NAT)

The NAT function is applied on frames going from the internal to the external or external to internal network. It supports translating the source and destination IP and port before the frame transitions to the opposing network.

All NAT modes support the following modes:

- XP_NAT_IP_ONLY—Translates the IP address only.
- XP_NAT_IP_AND_PORT—Translates the IP address and source port.

NAT can be enabled per ingress Bridge Domain.

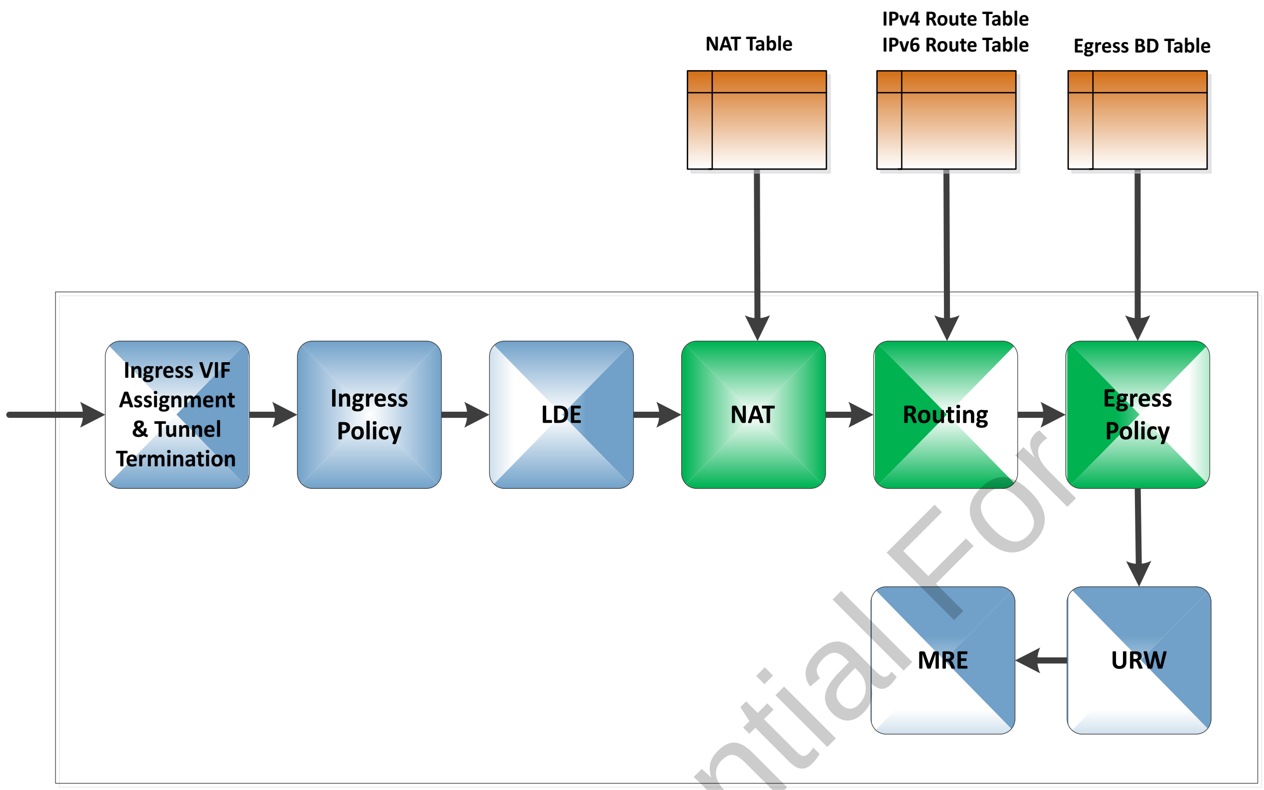


Figure 3–19 Full NAT Pipeline View

Resource Usage

Table 3–10 NAT Resources Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
NAT	1	NAT table	NAT Table entry attributes	
Route	1	Ipv4UcRoute, Ipv6UcRouteTable	Ipv4UcRoute, Ipv6UcRouteTable entry	
Egress Policy	1	Egress BD Table	Egress BD Table entry	

NAT Key Type There is one Key Type supported for all NAT types. The TCAM mask will determine which specific fields to match upon.

NAT Key Size The predefined Key Size is 192 bits.

NAT Data There is a predefined NAT data that determines the IP address and port rewrites to occur in the rewrite engine.

The result of the NAT Lookup table when enabled returns the NAT data entries. Each table returns a hit offset that is used as an index to the NAT data table.

3.6.2.2 Decision Logic

NAT Engine—Receives packet-related information from the Bridge Engine and implements the NAT logic. NAT is performed if the Multicast Bridge FDB, IPv4, or Ipv6 table gives a NAT lookup.

Lookup/Token—Will be described in subsequent sections for each type of NAT.

Routing Engine—Receives packet-related information from the NAT Engine and implements the route logic. Only applies to DIP translation

Egress Policy Engine—If the egress VIF matches the VIF, Egress BD lookup will enable SIP translation if NAT engine sets that feature

Update and Rewrite Engine—Receives packet-related information from the Egress BD Engine and modifies the header as per rewrite pointer and metadata in the token.

3.6.2.3 Source Network Address Translation (Src-NAT)

The Source NAT function is applied on frames going from the internal to the external network when the source IP and source port are translated before the frame goes to the external network.

Source NAT supports the following modes:

- XP_NAT_IP_ONLY—Translates the SIP only.
- XP_NAT_IP_AND_PORT—Translates the SIP address and source port.

NAT can be enabled per ingress Virtual Interface.

Decision Logic

NAT Engine

Lookup—For valid packets, lookups can be performed as described below,

- If the NAT mode is IP and network scope is internal, for a tunnel terminated packet that has an inner IPv4 header OR a non-tunnel terminated packet with an IPv4 header, the source IP address is used for lookup and the protocol is IPv4. Key = {SIP}.
- If the NAT mode is port and network scope is internal, for a tunnel-terminated packet that has an inner IPv4 header OR a non-tunnel terminated packet with an IPv4 header, the source IP address and source port number of TCP or UDP are used for lookup and the protocol is IPv4. Key ={SIP, Port}.

Token—Result = {Translated SIP, Translated Port, VLAN}

- If the NAT mode is IP and lookup is a hit and not a trap to CPU packet, perform source IP translation and forward the token to the route engine. Next Engine field of the token is set to Route.
- If the NAT mode is port and lookup is a hit and not a trap to CPU packet, perform source IP translation and source port translation and forward the token to the route engine. The Next Engine field of the token is set to Route.

Route Engine—Described in General NAT Decision Logic.

Egress Policy Engine—If the NAT config is enabled in the EgressBD lookup result, then SIP is translated. Consequently, the token rewrite ptr is set to modification pointer in the rewrite instruction table such that it copies the NAT SIP into the ipvxlayer SIP and/or NAT port into TCP/UDP layer port

Update and Rewrite Engine—In the specific case of Src NAT, the translated Src IP is copied into the Src IP field of the IPvX packet. For PAT mode, the translated Src port is also copied into the Src port field of the TCP/UDP packet.

3.6.2.4 Destination Network Address Translation (Dest-NAT)

Destination NAT is applied on frames going from the external to the internal network when the destination IP (DIP) and destination port must be translated before the frame gets routed inside the internal network.

Destination NAT supports the following modes:

- XP_NAT_IP_ONLY—Translates the DIP address only.
- XP_NAT_IP_AND_PORT—Translates the DIP address and destination port.

Decision Logic

NAT Engine

Lookup—For valid packets, lookups can be performed as described below.

- If the NAT mode is IP and network scope is internal, for a tunnel-terminated packet that has an inner IPv4 header or a non-tunnel-terminated packet with an IPv4 header, the destination IP address is used for lookup and the protocol is IPv4. Key = {DIP, BD}.
- If the NAT mode is Port and network scope is internal, for a tunnel-terminated packet that has an inner IPv4 header or a non-tunnel-terminated packet with an IPv4 header, the destination IP address and destination port number of TCP or UDP is used for lookup and the protocol is IPv4. Key = {DIP, Port, BD}.

Token—Result = {Translated DIP, Translated Port}

- If the NAT mode is IP and lookup is a hit and not a trap to CPU packet, perform destination IP translation and forward the token to the Route Engine. The Next Engine field of the token is set to Route.
- If the NAT mode is Port and lookup is a hit and not a trap to CPU packet, perform destination IP translation and destination Port translation and forward the token to the Route Engine. The Next Engine field of the token is set to Route.

Route Engine —Receives packet related information from NAT Engine and implements the Route logic.

Lookup—For valid packets, route lookup is performed as follows:

- The resulting translated DIP, port from the NAT table is used to look up against the Ipv4 Unicast Route Table

Token—For valid lookups, the metadata of the token is modified as follows:

- If hit, the packet header replaces DIP and port with modified DIP and /or modified port (for TCP/UDP). Consequently, the token rewrite pointer is set to modification pointer in the Rewrite Instruction table such that it copies the NAT DIP into the ipvxlayer DIP and/or NAT port into UDP/TCP layer port. The Next Engine field of the token is set to Egress Policy Engine.
- If miss, no re-write should be done to the packet and the token metadata is not modified. Rewrite pointers are set to 0. The Next Engine field of the token is set to Rewrite Engine.

Egress Policy Engine—No impact for DIP translation

Update and Rewrite Engine—In the case of Dest NAT, the translated Dest IP is copied into the Dest IP field of the IPvX packet. For PAT mode, the translated Dest port is also copied into the Dest port field of the TCP/UDP packet.

3.6.2.5 Source and Destination Network Address Translation (Src-Dest-NAT or Double NAT)

The flow for Source and Destination NAT is the combination of unicast source NAT and destination NAT. The instructions are set in the same engines as for unicast, but there are two lookups performed using source IP and destination IP, respectively.

Decision Logic

NAT Engine

Lookup—For valid packets, lookups can be performed as described below.

If the NAT mode is SIP and DIP, for a tunnel-terminated packet that has an inner IPv4 header or a non-tunnel-terminated packet with an IPv4 header, the source IP address, destination IP address. and BD is used for lookup and the protocol is IPv4. The following two lookups are performed:

- For Src Lookup, Key = {SIP, Port}
- For Dest Lookup, Key = {DIP, Port, BD}

Token—If the NAT mode is SIP and DIP and lookup is a hit and not a trap to CPU packet, perform source IP translation and destination IP translation and forward the token to the Route Engine.

- Result of Src Lookup = {Translated SIP, Translated Port, VLAN}
- Result of Dest Lookup = {Translated DIP, Translated Port}

Route Engine

Lookup—For valid packets, route lookup is performed as follows:

- The resulting translated SIP, DIP, port from the NAT table is used to look up against the IpvXroute table.

Token—For valid lookups, the metadata of the token is modified as follows:

- If hit, the packet header replaces SIP, DIP, and port with modified SIP, DIP, and /or modified port (for TCP/UDP). Consequently, the token rewrite ptr is set to modification pointer in the Rewrite Instruction table such that it copies the NAT SIP, and DIP into the ipvxlayer SIP, DIP, and/or NAT port into the UDP/TCP layer port. The Next Engine field of the token is set to Egress Policy Engine.
- If miss, no rewrite should be done to the packet and the token metadata is not modified. Rewrite pointers are set to 0. The Next Engine field of the token is set to Rewrite Engine.

Egress Policy Engine—If the NAT config is enabled in the EgressBd lookup result, then SIP is translated. Consequently, the token rewrite ptr is set to modification pointer in the rewrite instruction table such that it copies the NAT SIP into the ipvxlayer SIP and/or NAT port into TCP/UDP layer port.

Update and Rewrite Engine—In the case of Src and Dest NAT, the translated SrcIP, Dest IP is copied into the Src IP and Dest IP field of the IPvX packet, respectively. For PAT mode, translated Src, Dest port is also copied into the Src, Dest port field of the TCP/ UDP packet, respectively.

3.6.2.6 Multicast Network Address Translation (M-NAT)

The flow for multicast NAT is same as for unicast source NAT and destination NAT. The instructions are set in the same engines as for unicast NAT. The configuration on whether to translate or not is driven by the NAT configuration per egress VLAN, which is represented as a node in the multicast replication engine.

The per node NAT configuration can be programmed and, if the ignoreNAT is set, the instruction pointers are cleared and translation is skipped for that egress VLAN.

Decision Logic

NAT Engine

Lookup—For valid packets, lookups can be performed as described below.

If the NAT mode is SIP and DIP for a tunnel-terminated packet that has an inner IPv4 header or a non-tunnel-terminated packet with an IPv4 header, the source IP address, destination IP address, and BD are used for lookup and the protocol is IPv4. The following two lookups are performed:

- For Src Lookup, Key = {SIP, Port}
- For Dest Lookup, Key = {DIP, Port, BD}

Token—If the NAT mode is SIP and DIP and lookup is a hit and not a trap to the CPU packet, perform source IP translation and destination IP translation and forward the token to the Route Engine.

- Result of Src Lookup = {Translated SIP, Translated Port, VLAN}
- Result of Dest Lookup = {Translated DIP, Translated Port}

Route Engine—Receives packet-related information from NAT Engine and implements the route logic.

Lookup—For valid packets, route lookup is performed as follows:

- The resulting translated DIP, and port from the NAT table is used to look up against the Route table.

Token—For valid lookups, the metadata of the token is modified as follows:

- If hit, the packet header replace DIP, and Port with modified SIP, DIP, and /or modified Port (for TCP/UDP). Consequently, the token rewrite ptr is set to modification pointer in the rewrite instruction table such that it copies the NAT, and DIP into the ipvxlayer SIP, DIP, and/or NAT port into the UDP/TCP layer port. The Next Engine field of the token is set to Egress Policy Engine.
- If miss, no rewrite is done to the packet and the token metadata is not modified. Rewrite pointers are set to 0. The Next Engine field of the token is set to Rewrite Engine.

Egress Policy Engine—If the NAT configuration is enabled in the EgressBd lookup result, then SIP is translated. Consequently, the token rewrite ptr is set to modification pointer in the rewrite instruction table such that it copies the NAT SIP into the ipvxlayer SIP and/or NAT port into TCP/UDP layer port.

Update and Rewrite Engine—In the case of Src and Dest NAT, the translated SrcIP, Dest IP is copied into the Src IP and Dest IP field of the IPvX packet, respectively. For PAT mode, translated Src, Dest port is also copied into the Src, Dest port field of the TCP/ UDP packet, respectively.

3.7 Tunnel

The XPliant Software-Defined Platform supports various tunneling overlay schemes, allowing proprietary tunneling mechanisms and headers along with the standardized Layer 2, Layer 3, and MPLS tunnels. A few examples of the known standard tunnels supported are described in the sections below.

- 3.7.1 Layer 2 Overlay Tunnels (VXLAN, NVGRE, Geneve)
- 3.7.2 Layer 3 Overlay Tunnels (IP-over-GRE, IP-over-IPv4, MPLS-over-GRE)
- 3.7.3 MPLS Overlay Tunnels

3.7.1 Layer 2 Overlay Tunnels (VXLAN, NVGRE, Geneve)

3.7.1.1 Description

- Virtual Extensible LAN (VXLAN) is a network virtualization technology that attempts to ameliorate the scalability problems associated with large cloud computing deployments. The XPliant Software-Defined Platform supports VXLAN tunnel detection, termination, and origination, conforming to RFC 7348.
- NVGRE (Network Virtualization using Generic Routing Encapsulation) is a network virtualization technology that attempts to alleviate the scalability problems associated with large cloud computing deployments. It uses generic routing encapsulation (GRE) to tunnel layer 2 packets over layer 3 networks. The XPliant Software-Defined Platform supports NVGRE tunnel detection, termination, and origination, conforming to RFC 2890.
- Geneve (Generic Network Virtualization Encapsulation) is an evolving encapsulation protocol that aims to support a variety of network virtualization use cases. The XPliant Software-Defined Platform supports Geneve tunnels detection, termination, and origination.

3.7.1.2 Software-Defined Implementation

Tunnel Termination

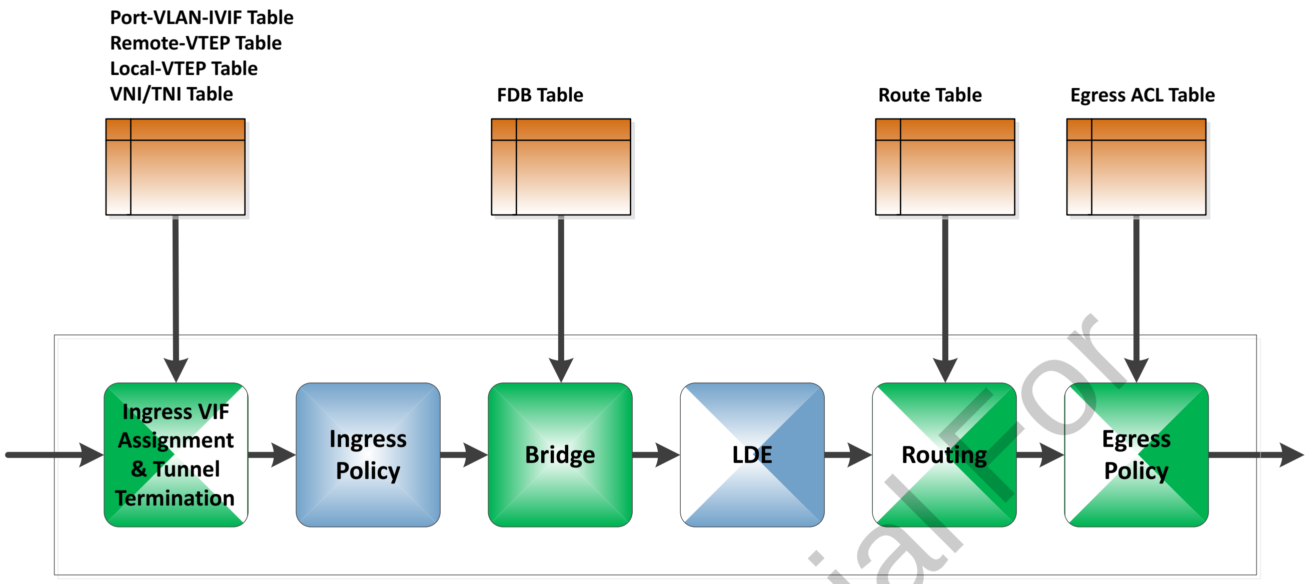


Figure 3–20 VXLAN/NVGRE/Geneve Tunnel Termination Pipeline View

Resource Usage

Table 3–11 VXLAN/NVGRE/Geneve Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Ingress VIF Assignment and Tunnel Termination	**1.** portVlan **2.** Local VTEP **3.** Remote VTEP **4.** VNI /TNI	PortVlanIvifTable IpVirtualTnlLocalVtepTable IpVirtualTnlIvifTable IpVirtualTnlIdTable	Get the Tunnel VIF attributes.	Use the same physical table: IpVirtualTnlIdTable and different tnlType key for VXLAN/NVGRE/Geneve.
Bridge	2: Shared with the FDB MAC SA and DA lookups.	Both lookups to the FDB table.	MAC entry attributes.	Lookups are based on inner packet headers.
Routing	Routing Lookup	Routing Lookup table.		Lookups are based on inner packet headers.
Egress Policy	EACL Lookup	EACL Lookup table.		Lookups are based on inner packet headers.

3.7.1.3 Decision Logic

Ingress VIF Assignment and Tunnel Termination—Receives the packet-related information from the parser as an input and implements the VXLAN/NVGRE/Geneve termination logic.

This engine will, for valid and eligible-for-processing packets:

Lookup—For valid packets, IVIF assignment and tunnel termination lookups can be performed as described below.

- Always perform a port VLAN lookup to the PortVlanIvif table with the key {PORT VIF, OUT VLANID}

   - Get the value of port VIF from ingressVif, which is from the port configuration table from the parser.
   - Get the VLAN ID from the ethLayer.
- Perform local VTEP lookup to the IpVirtualTnlLocalVtep table with the key {tunnel type, local ip} only for valid Ethernet packets, which are recognized as a VXLAN/NVGRE/Geneve packet with no IP header error and TTL value is not zero.
   - Set the value of tunnel type entryFormatLocalVtep = 1.
   - Get the local IP from IPv4 layer destination IP.
- Perform remote RFC 7348 lookup to the IpVirtualTnlIvif Table with the key {tunnel type, remote ip}, only for valid Ethernet packets, which are recognized as VXLAN/NVGRE/Geneve packet with no IP header error and TTL value is not zero.
   - Set the value of ipvxKey XP_TUNNEL_TYPE_IP_VIRTUAL.
   - Get the remote IP from IPv4 layer source IP.
- Perform VNI/TNI lookup to the IpVirtualTnlId table with the key {tunnel type, VNI/TNI}, only for valid Ethernet packets, which are recognized as VxLan/NVGRE/Geneve packet wjth no IP header error and TTL value is not zero.
   - Set the value of tunnel type XP_IPVX_TNL_TYPE_VXLAN for Vxlan packet; set the value of tunnel type XP_IPVX_TNL_TYPE_NVGRE for NVGRE packet; set the value of tunnel type XP_IPVX_TNL_TYPE_GENEVE for Geneve packet.
   - Get the VNI from VXLAN layer VNI for VXLAN packet; get the TNI from NVGRE layer TNI for NVGRE packet; get the VNI from Geneve layer VNI for Geneve packet.
   - Set the value of entryFormatTnlId 2.

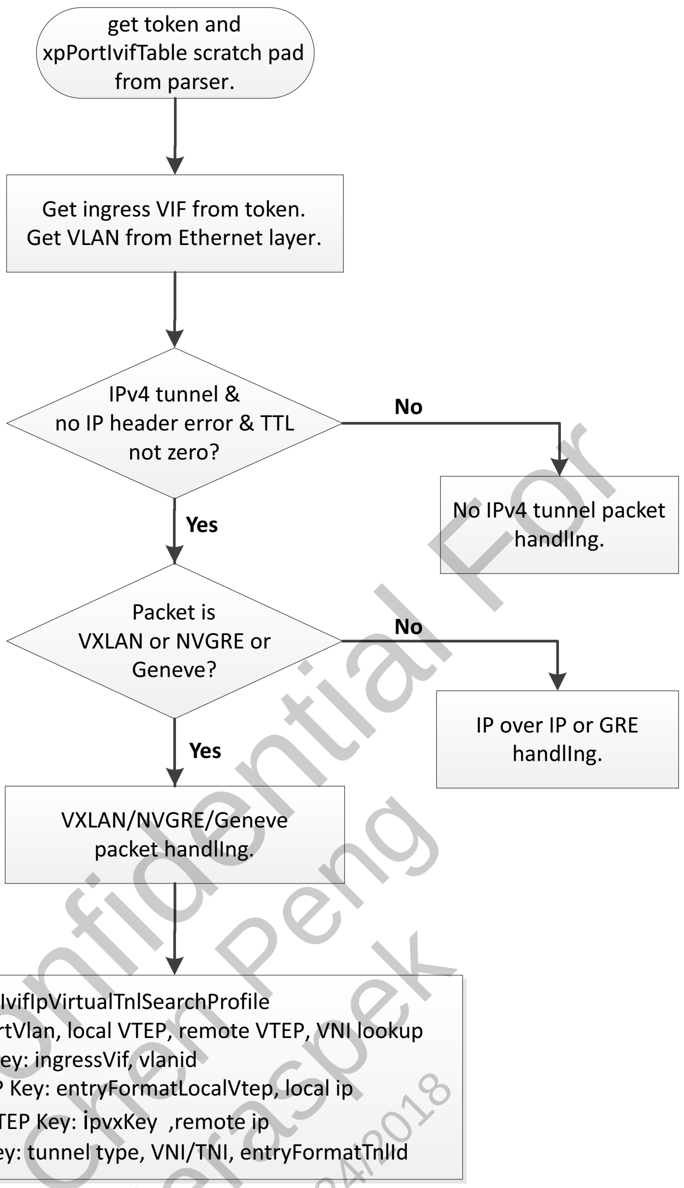

get token and xpPortIvifTable scratch pad from parser.
Get ingress VIF from token. Get VLAN from Ethernet layer.
IPv4 tunnel & no IP header error & TTL not zero?
No
No IPv4 tunnel packet handlIng.
Yes
Packet is VXLAN or NVGRE or Geneve?
No
IP over IP or GRE handlIng.
Yes
VXLAN/NVGRE/Geneve packet handlIng.
Choose xpIvifIpVirtualTnlSearchProfile
Enable portVlan, local VTEP, remote VTEP, VNI lookup
portVlan key: ingressVif, vlanid
Local VTEP Key: entryFormatLocalVtep, local ip
Remote VTEP Key: ipvxKey ,remote ip
VNI/TNI key: tunnel type, VNI/TNI, entryFormatTnlId

Figure 3–21 VXLAN/NVGRE/Geneve Tunnel Termination Key Selection Logic

Token—When a lookup has been performed and packet is VXLAN/NVGRE/ Geneve, then terminate the packet and update the token.

- If there is a miss in the Port VLAN table, invalid ingress VLAN handling (3.4.1 Layer 2 Bridging), else
- If local VTEP hit and VNI/TNI table miss, VXLAN/NVGRE/Geneve packet destined “TO ME” has invalid VNI/TNI TRAP to CPU, else
- If local VTEP hit and Remote VTEP table miss, VXLAN/NVGRE/Geneve packet destined “TO ME” has unknown remote VTEP IP, new tunnel interface lean, TRAP to CPU, else
- If local VTEP hit and VNI/TNI table hit and Remote VTEP hit, VXLAN/ NVGRE/Geneve packet destined “TO ME” and known remote IP and VNI/ TNI, then terminate the VXLAN/NVGRE/Geneve tunnel, else

- If the VNI/TNI table setBd is enable, assign the BD from the VNI table, else assign the BD from the port Ivif table;
- Assign the ACL ID from the VNI/TNI table
- Assign the interface attributes from the Remote VTEP table, including ingressVif, acm mirror_mask nataclConfig portVlanSpanState.
- Update the firstVldLayer to 3, the pointer to the inner header, and terminate the VXLAN/NVGRE/Geneve tunnel.
- Assign token tunnel type true.
- Forward to Bridge Engine process.

❍ VXLAN/NVGRE/Geneve packet is not destined “TO ME” (no match on local VTEP IP), do not terminate and take port VLAN result and proceed (3.4.1 Layer 2 Bridging).

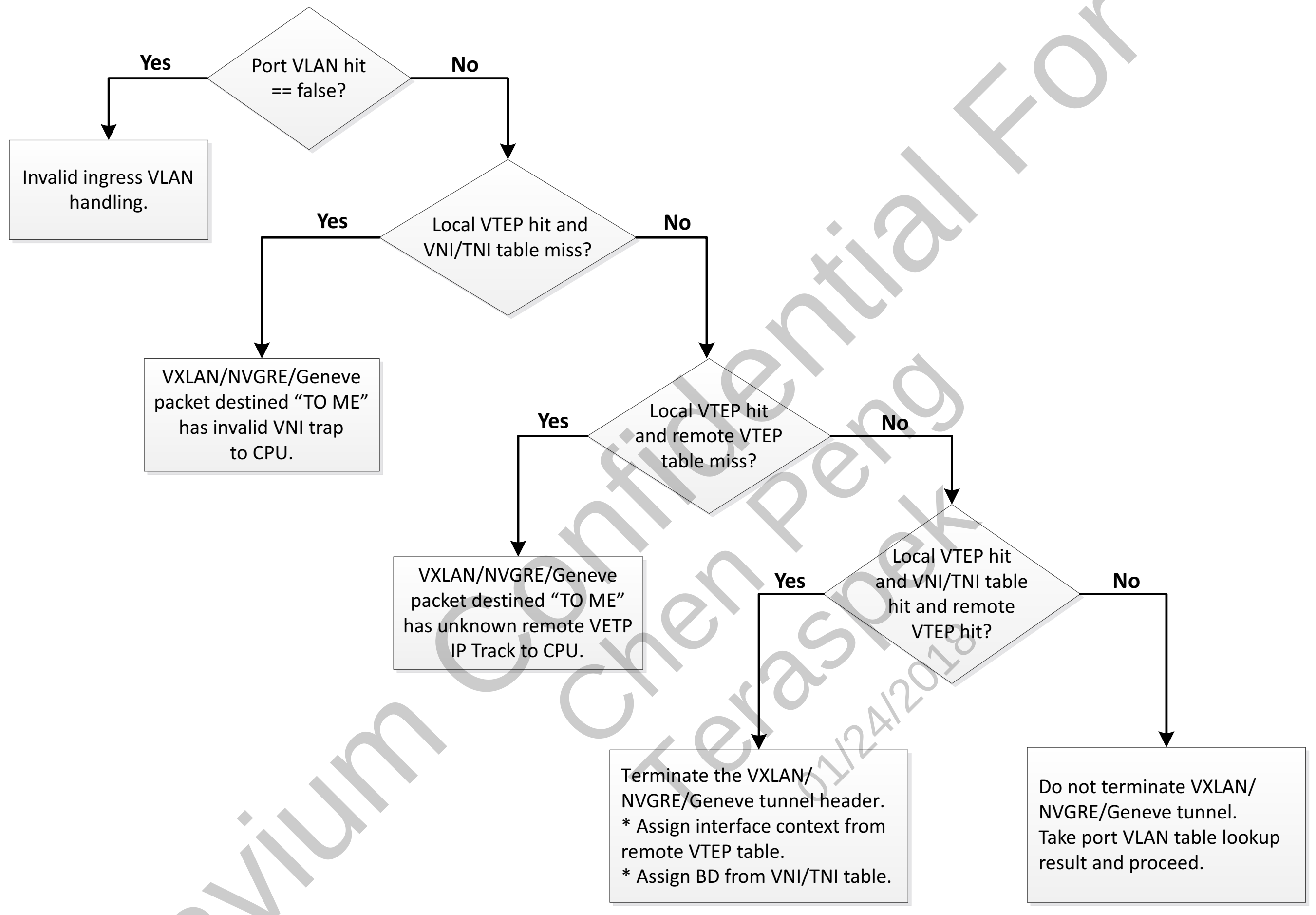

Port VLAN hit == false?
Yes
No
Invalid ingress VLAN handling.
Local VTEP hit and VNI/TNI table miss?
Yes
No
VXLAN/NVGRE/Geneve packet destined “TO ME” has invalid VNI trap to CPU.
Local VTEP hit and remote VTEP table miss?
Yes
No
VXLAN/NVGRE/Geneve packet destined “TO ME” has unknown remote VETP IP Track to CPU.
Local VTEP hit and VNI/TNI table hit and remote VTEP hit?
Yes
No
Terminate the VXLAN/ NVGRE/Geneve tunnel header.
* Assign interface context from remote VTEP table.
* Assign BD from VNI/TNI table.
Do not terminate VXLAN/ NVGRE/Geneve tunnel.
Take port VLAN table lookup result and proceed.

Figure 3–22 VXLAN/NVGRE/Geneve Tunnel Termination Lookup Logic

Bridge—Receives the packet-related information from the Tunnel Termination Engine as an input and implements the bridge logic. The firstVidLayer points to the inner header of VXLAN/NVGRE/Geneve packet, the BD and ACL ID are taken from the VNI/TNI table, and the interface attributes are from the Remote VTEP table.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups with inner header can be performed as described below.

- If the inner MAC DA is identified as a router MAC, then the next engine routing will process.
- Else it will L2 bridge (3.4.1 Layer 2 Bridging).

Routing—Receives the packet-related information from the Bridge Engine as an input and implements the route logic. The firstVidLayer pointer to the inner header of VXLAN/NVGRE/Geneve packet, vrfId routeCtrlBits from the VNI/TNI BD table.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, routing lookups can be performed as described below.

- For VXLAN/NVGRE/Geneve tunnel packet, use L3_ecmp_hash_b, RouteEn flag is from the VNI/TNI BD table, key from the inner layer and detail.

EACL—Receives the packet-related information from the Route Engine as an input and implements the EACL logic. The firstVidLayer pointer is to the inner header of VXLAN/NVGRE/Geneve packet.

Tunnel Origination

IACL—Receives the packet-related information from the Ingress VIF Assignment and Tunnel Termination Engine as an input and implements the IACL logic. IACL can originate VXLAN/NVGRE/Geneve packet by redirect to VXLAN/NVGRE/Geneve tunnel interface (EVIF).

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, IACL lookups can be performed as described below.

- IACL table Hit and the attribute enRedirectToEvif is true of this entry else
  - Assign the VXLAN/NVGRE/Geneve tunnel eVif to token egressVif.
  - Assign the VNI/TNI from the IACL table to the Ethernet layer.
  - Jump to the Update and Rewrite Engine processing.
- No redirect ingress process

Outgoing L2 and L3 insertion pointers and data are predefined and bound to a Tunnel eVIF.

- Create the tunnel eVIF.
  - Assign the l2HdrInsrtId to the insPtr0 of this tunnel eVIF entry the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE, and set insert data macDA, saLabs, sVid, cVid, and l2EncapType
- Assign the l3HdrInsrtId to the insPtr1 of this tunnel eVIF entry.

  VXLAN tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP address and Remote EP IP address

  NVGRE tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP addr and Remote EP Ip addr

Geneve tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP addr and Remote EP Ip addr

- Assign the l4HdrInsrtId to the insPtr2 of this tunnel EVIF entry.

  VXLAN: the l4HdrInsrtId use XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destination Port and VxLan reserved data, pick VNI from Ethernet layer

  NVGRE: The l4HdrInsrtId use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destination Port and NVGRE reserved data, pick TNI from Ethernet layer

  Geneve: The l4HdrInsrtId use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destination Port and GENEVE reserved data, pick VNI from Ethernet layer

- No redirect; continue with the normal ingress processing.

Bridge—Receives the packet-related information from previous engine as an input and implements the bridge logic. The bridge can originate VXLAN/NVGRE/Geneve packet if the egressVif of the FDB DA table entry pointer to VXLAN/NVGRE/Geneve tunnel interface (EVIF).

This engine will, for valid and eligible-for-processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- FDB DA table hit and EVIF of the entry pointer to the VxLan/NVGRE/ Geneve tunnel EVIF else
  - If encapType of the FDB entry is XP_L2_UNTAGGED_ENCAP_TYPE, then set token.rewritePtr0 with XP_VLAN_UNTAGGED_HDR_PROFILE for URW to strip the TAG;IF encapType of the FDB DA entry is XP_L2_TAGGED_ENCAP_TYPE, then assign 0x0800 to etherType, and if L2OverIPTunnel use insert XP_INSERT_ORIGINAL_ETH_HDR
  - Assign fdbVirtualId of FDB table to VNI/TNI
  - Forward to EACL process, the nextEngine is EACL
- FDB DA table miss or FDB entry evif is not tunnel EVIF.

Out L2 and L3 insertion pointers and data are predefined and bound to a tunnel eVIF.

- Create the tunnel eVIF
  - Assign the l2HdrInsrtId to the insPtr0 of this tunnel eVif entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and set insert data macDA, saLabs, sVid, cVid, and l2EncapType.
  - Assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry,

VXLAN tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP address and Remote EP IP address

NVGRE tunnel: the insert data corresponding data with this l3HdrInsrtId use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP addr and Remote EP IP addr

Geneve tunnel the insert data corresponding data with this l3HdrInsrtId use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP addr and Remote EP IP addr

- Assign the l4HdrInsrtId to the insPtr2 of this tunnel EVIF entry.

  VXLAN: The l4HdrInsrtId use XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destination Port and VXLan reserved data, pick VNI from Ethernet layer.

  NVGRE: The l4HdrInsrtId use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destination Port and NVGRE reserved data, pick TNI from Ethernet layer.

  Geneve: The l4HdrInsrtId use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destination Port and GENEVE reserved data, pick VNI from Ethernet layer.

Routing—Receives the packet-related information from previous engine as an input and implements the routing logic. Routing can originate VXLAN/NVGRE/Geneve packet if the egressVif of the Next Hop table entry points to VXLAN/NVGRE/Geneve tunnel interface (EVIF).

This engine will, for valid and eligible for processing packets,

Lookup—For valid packets, routing lookups can be performed as described below.

- LPM table hit and EVIF of the entry pointer to the VXLAN tunnel eVIF else
  - Strip the L2 header, token.firstVldLayer + 1.
  - Assign 1 to rewritePtr6IsInsert of token to use pseudo layer for inner L2 insertion by Update and Rewrite Engine and assign the enCapType from the Next Hop table.
  - Assign the MacDa, cVid, and VNI/TNI etc to the pseudo layer to carry to the Update and Rewrite Engine
- LPM table miss or the EVIF is not tunnel eVIF

Out L2 and L3 insertion pointers and data are predefined and bound to a Tunnel eVIF.

- Create the Tunnel eVIF
  - Assign the l2HdrInsrtId to the insPtr0 of this tunnel eVIF entry , the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and set insert data macDA,saLabs,sVid,cVid and l2EncapType.

- Assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry.

  VXLAN tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP address and Remote EP IP address.

  NVGRE tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP addr and Remote EP Ip addr.

  Geneve tunnel: The insert data corresponding data with this l3HdrInsrtId use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[0] and the insert data including local EP IP addr and Remote EP Ip addr.

- Assign the l4HdrInsrtId to the insPtr2 of this tunnel EVIF entry.

  VXLAN: The l4HdrInsrtId use XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destionation Port and VXLAN-reserved data, pick VNI from Ethernet layer.

  NVGRE: The l4HdrInsrtId use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate destination port and NVGRE reserved data, pick TNI from Ethernet layer.

  Geneve: The l4HdrInsrtId use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[1] and the insert data corresponding data with this including const data to populate Destionation Port and GENEVE reserved data, pick VNI from Ethernet layer.

• Bond port to list to tunnel VIF, set the corresponding Next Hop table EVIF pointer to Tunnel VIF.

MRE—Receives the packet-related information from previous engine as an input and implements the Multicast logic. MRE can originate a VXLAN/NVGRE/Geneve packet if the egressVif of the MDT table entry pointer to the tunnel interface (eVIF).

This engine will, for valid and eligible for processing packets,

Lookup—For valid packets, bridge lookups can be performed as described below.

❍ Egress VIF of the MDT entry pointer to the VxLan tunnel eVIF else
  - If encapType of the FDB entry is XP_L2_UNTAGGED_ENCAP_TYPE, then set token.rewritePtr0 with XP_VLAN_UNTAGGED_HDR_PROFILE for URW to strip the TAG;IF encapType of the FDB DA entry is XP_L2_TAGGED_ENCAP_TYPE, then assign 0x0800 to etherType , and if L2OverIPTunnel use insert XP_INSERT_ORIGINAL_ETH_HDR
  - Assign tunnel data of MDT table to VNI/TNI.

❍ Egress VIF of MDT is not tunnel eVIF.

Out L2 and L3 insertion pointers and data are predefined and bound to a Tunnel eVIF.

❍ Create the Tunnel eVIF.

- Assign the l2HdrInsrtId to the insPtr0 of this tunnel eVIF entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and macDA,saLabs,sVid,cVid and l2EncapType
- Assign the l3HdrInsrtId to the insPtr1 of this tunnel eVIF entry

  VXLAN: The insert data corresponding data with this l3HdrInsrtId including local EP IP addr and Remote EP Ip addrXP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[0]

  NVGRE: the insert data corresponding data with this l3HdrInsrtId including local EP IP addr and Remote EP Ip addrand use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[0].

  Geneve: x the insert data corresponding data with this l3HdrInsrtId including local EP IP addr and Remote EP Ip addr and use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[0].
- Assign the l4HdrInsrtId to the insPtr2 of this tunnel eVif entry.

  VXLAN: The insert data corresponding data with this l4HdrInsrtId including constant data to populate destination port and VXLAN reserved data, pick VNI from Ethernet layer XP_VXLAN_INSERT_HDR_PROFILE insrtPtrs[1].

  NVGRE: The insert data corresponding data with this l3HdrInsrtId including local EP IP addr and Remote EP Ip addrand use XP_NVGRE_INSERT_HDR_PROFILE insrtPtrs[1].

  Geneve: x the insert data corresponding data with this l3HdrInsrtId including local EP IP addr and Remote EP Ip addr and use XP_GENEVE_INSERT_HDR_PROFILE insrtPtrs[1];

- Bond port to list to tunnel VIF, set the corresponding MDT table EVIF pointer to tunnel VIF.

Update and Rewrite—Receives the packet-related information from previous engine as an input and implements the update and rewrite header logic. If the EVIF of token is of a tunnel VIF, then the engine will encapsulate the tunnel header.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- Update and Rewrite searches the EVIF table and if the EVIF is tunnel EVIF, else
  - Outer L2 Header insertion: use insPtr0 of the tunnel EVIF entry
    - → Pick Ether type from Ethernet Layer /Pseudo Layer
    - → Pick SA LSBs from SE Insertion Data
    - → Pick DA from SE Insertion Data
    - → Use constant data to populate TPID if required
  - Outer L3 Header insertion: use insPtr1 of the tunnel EVIF entry
    - → Use Insert Constant data to add IP version, reserved fields, flags, and UDP protocol number.

→ DIP from SE Insertion Data.

→ SIP from SE Insertion Data.

→ TTL decrement from IPv4 Layer Data.

→ DSCP from Token Common Data.

– UDP and VXLAN header insertion for VXLAN: use insPtr2 of tunnel eVIF entry.

→ Use constant data to populate Destination Port and VXLAN reserved data.

→ VXLAN VNI from Pseudo Layer/Ethernet Layer.

→ UDP Source port entropy via L3 inner packet hash.

– UDP and NVGRE header insertion for NVGRE tunnel: use insPtr2 of tunnel EVIF entry.

→ Use constant data to populate Destination Port and NVGRE reserved data.

→ NVGRE TNI from Pseudo Layer/Ethernet Layer.

→ UDP Source port entropy via L3 inner packet hash.

– UDP and GENEVE header insertion: use insPtr2 of tunnel EVIF entry.

→ Use constant data to populate Destination Port and GENEVE reserved data.

→ GENEVE VNI from Pseudo Layer/Ethernet Layer.

→ UDP Source port entropy via L3 inner packet hash.

– In the case of RIOT:

→ Inner L2 instruction is the same as the IP Next Hop instruction

❍ EVIF is not tunnel eVIF process, continue with the normal forwarding process.

3.7.2 Layer 3 Overlay Tunnels (IP-over-GRE, IP-over-IPv4, MPLS-over-GRE)

3.7.2.1 Description

- Generic Routing Encapsulation (GRE) is a tunneling protocol capable of encapsulating a wide variety of network layer protocols inside virtual point-to-point links over an Internet Protocol internetwork.

  The XPliant Software-Defined Platform supports GRE tunnel detection, termination and origination, conforming to RFC 2784, including the IP-over-GRE case.

- IP in IP is an IP tunneling protocol that encapsulates one IP packet in another IP packet. To encapsulate an IP packet in another IP packet, an outer header is added with SourceIP, the entry point of the tunnel, and the destination point, the exit point of the tunnel. While doing this, the inner packet is unmodified (except the TTL field, which is decremented).

  The XPliant Software-Defined Platform supports IP-over-IPv4 tunnel detection, termination, and origination, conforming to RFC 2003, including the IP-over-IPv4 case.

- The MPLS-over-GRE feature provides a mechanism for tunneling Multiprotocol Label Switching (MPLS) packets over a non-MPLS network. This feature uses MPLS over generic routing encapsulation (MPLSoGRE) to encapsulate MPLS packets inside IP tunnels. The encapsulation of MPLS packets inside IP tunnels creates a virtual point-to-point link across non-MPLS networks.

NOTE: An MPLS-over-GRE tunnel can only originate from the route table (LPM); it cannot originate from the host table.

3.7.2.2 Software-Defined Implementation

Tunnel Termination

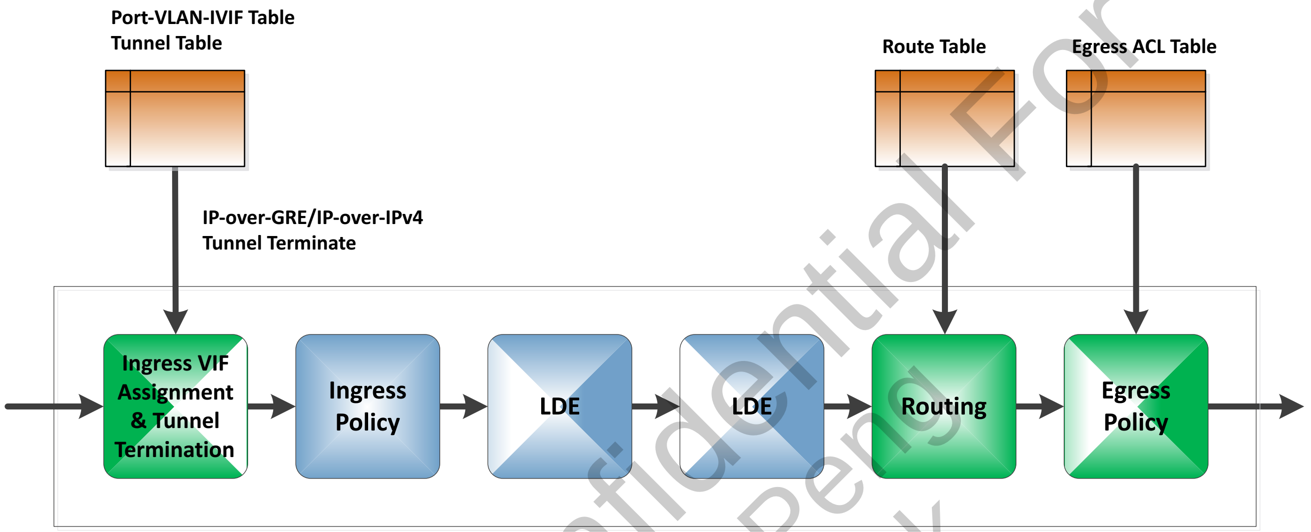


Figure 3–23 IP-over-GRE/IP-over-IPv4/MPLS-over-GRE Tunnel Terminate Pipeline View

Resource Usage

Table 3–12 IP-over-GRE/IP-over-IPv4/MPLS-over-GRE Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Ingress VIF Assignment and Tunnel Termination	1. portVlan 2. IpvxTnlIvif	PortVlanIvifTable xpIpvxTnlIvifTable	Get the tunnel VIF attributes	
Routing	Routing lookup	Routing Lookup table		Lookups are based on inner packet headers
Egress ACL	EACL lookup	EACL Lookup table		Lookups are based on inner packet headers

3.7.2.3 Decision Logic

Ingress VIF Assignment and Tunnel Termination—Receives the packet-related information from parser as an input and implements the IP-over-GRE/IP-over-IPv4/MPLS-over-GRE termination logic.

This engine will, for valid and eligible for processing packets,

Lookup—For valid packets, Ingress VIF Assignment and Tunnel Termination lookups can be performed as described below

- Always perform a Port-VLAN lookup to the PortVlanIvif Table with the key {PORT VIF, OUT VLANID}
  - Get the value of PORT VIF from ingressVif which is from the port config table from parser.
  - Get the VLAN ID from the ethLayer.
- Perform remote IpvxTnlIvif lookup to the IpvxTnlIvif Table with the key {tunnel type,ingress vif, route mac,source ip, destination ip, outer vlan}, only for valid Ethernet packets, which are recognized as GRE /IP-over-IPv4/ MPLS-over-GRE packet and no IP header error and TTL value is not zero
  - Set the value of tunnel type XP_IPVX_TNL_TYPE_GRE for GRE, set XP_IPVX_TNL_TYPE_IPV4 for IP-over-IPv4, set XP_TUNNEL_TYPE_LOOSE_GRE_MPLS or XP_TUNNEL_TYPE_GRE_MPLS for MPLS-over-GRE packet based on MPLS Tunnel table hit or not.
  - Get the source IP and destination IP from IPv4.
  - Get the mac and routerMac from ethLayer.
  - Set the value of IVIF token.ingressVif, which is from port config table.

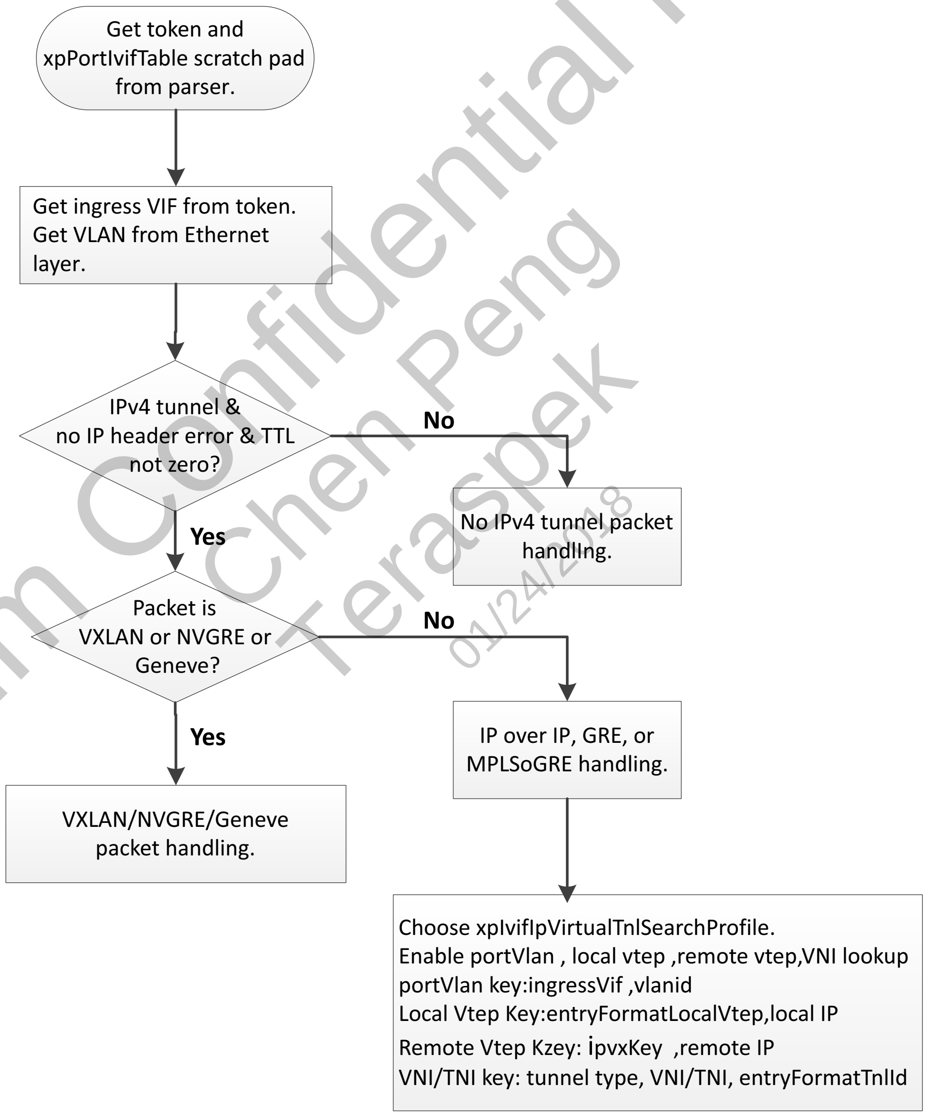

Get token and xpPortIvifTable scratch pad from parser.
Get ingress VIF from token. Get VLAN from Ethernet layer.
IPv4 tunnel & no IP header error & TTL not zero?
No
No IPv4 tunnel packet handlIng.
Yes
Packet is VXLAN or NVGRE or Geneve?
No
IP over IP, GRE, or MPLSoGRE handling.
Yes
VXLAN/NVGRE/Geneve packet handling.
Choose xpIvifIpVirtualTnlSearchProfile.
Enable portVlan , local vtep ,remote vtep,VNI lookup
portVlan key:ingressVif ,vlanid
Local Vtep Key:entryFormatLocalVtep,local IP
Remote Vtep Kzey: ipvxKey ,remote IP
VNI/TNI key: tunnel type, VNI/TNI, entryFormatTnlId

Figure 3–24 IP over GRE/IP-over-IPv4 tunnel termination KFIT logic

Token—When a lookup has been performed and a packet is IP-over- GRE/IP-over-IPv4/MPLS-over-GRE, then terminate it and update the token

- If there is a miss in Port VLAN table, invalid ingress VLAN handing (3.4.1 Layer 2 Bridging) else
- If local IpvxTnlIvif table hit, GRE/IP-over-IPv4/MPLS-over-GRE packet destined "TO ME", then terminate the GRE/IP-over-IPvV4/MPLS-over-GRE tunnel, else
  - If the IpvxTnlIvif table setBd is enable, assign the BD from the IpvxTnlIvif table, else assign the BD from the port Ivif table or assign the BD from the MPLS tunnel table.
  - Assign the aclid from the IpvxTnlIvif table.
  - Assign the interface attributes from the remote IpvxTnlIvif table, including acm mirror_mask nataclConfig portVlanSpanState; if setIngressVif is enable from IpvxTnlIvif table, assign ingressVif from IpvxTnlIvif<ingressVif>.
  - Update the firstVldLayer to 2, pointer to the inner header, terminate the GRE tunnel; update the firstVldLayer = firstVldLayer + 1, for IP-over-IPv4 packet, and also update the firstVldLayer for the MPLSoGRE packet.
  - Assign token tunnel type true and enable table search.
  - Forward to IACL Engine process.
- GRE packet is not destined "TO ME", do not terminate and take port VLAN result to proceed (3.4.1 Layer 2 Bridging).

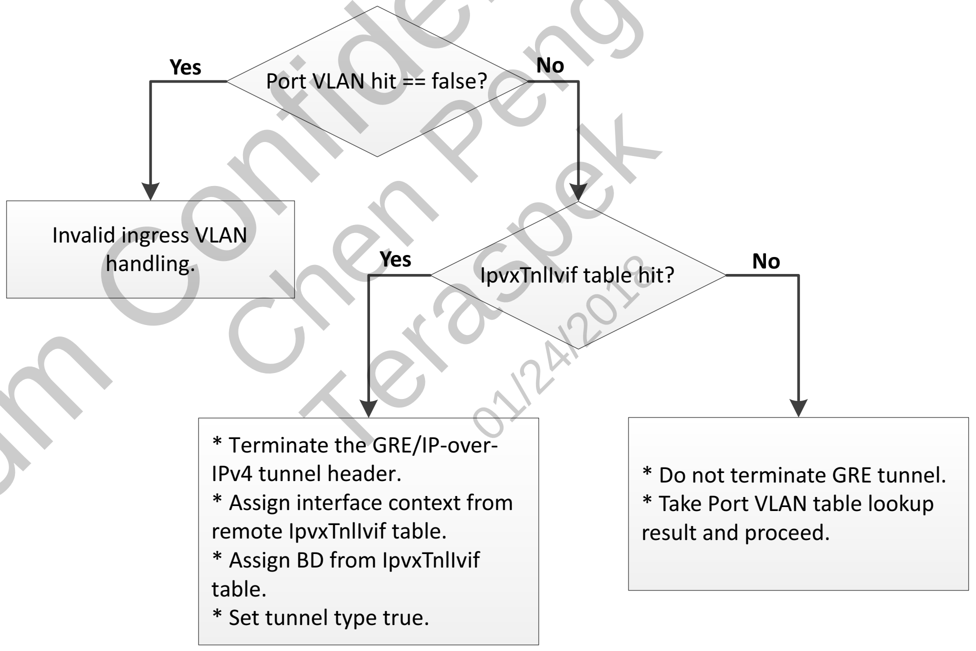

Port VLAN hit == false?
Yes
No
Invalid ingress VLAN handling.
IpvxTnlIvif table hit?
Yes
No
* Terminate the GRE/IP-over-IPv4 tunnel header.
* Assign interface context from remote IpvxTnlIvif table.
* Assign BD from IpvxTnlIvif table.
* Set tunnel type true.
* Do not terminate GRE tunnel.
* Take Port VLAN table lookup result and proceed.

Figure 3–25 IP over GRE/IP-over-IPv4/MPLS-over-GRE Tunnel Termination OFIT Logic

Routing—Receives the packet-related information from the Bridge Engine as an input and implements the route logic. The firstVidLayer pointer is to the inner header of the GRE/IP-over-IPv4/MPLS-over-GRE packet; vrfId routeCtrlBits are taken from the IpvxTnlIvif BD table.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, routing lookups can be performed as described below.

- For GRE/IP-over-IPv4/MPLSoGRE tunnel packet, use ecmpHash use L3_ecmp_hash_b, RouteEn flag is from the IpvxTnlIvif bd table, key from the inner layer, for MPLSoGRE loose mode the BD from the MPLS tunnel table.

EACL—Receives the packet-related information from the Route Engine as an input and implements the EACL logic. The firstVidLayer pointer is to the inner header of GRE/IP-over-IPv4/MPLSoGRE packet.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- For GRE/IP-over-IPv4/MPLSoGRE tunnel packet, search key from the inner layer and detail.

Tunnel Origination

IACL—Receives the packet-related information from Ingress VIF Assignment and Tunnel Termination Engine as an input and implements the IACL logic. IACL can originate GRE/IP-over-IPv4 packet by redirect to tunnel interface (eVIF).

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- IACL table Hit and the attribute enRedirectToEvif is true of this entry else
  - Assign the tunnel eVIF to token egressVif.
  - Jump to the Update and Rewrite Engine for further processing.
- No Redirect Ingress process.

Out L2 and L3 insertion pointers and data are predefined and bound to a Tunnel EVIF.

- Create the tunnel eVIF.
  - Assign the l2HdrInsrtId to the insPtr0 of this tunnel EVIF entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and macDA, saLabs, sVid,cVid, and l2EncapType.
  - Assign the l3HdrInsrtId to the insPtr1 of this tunnel eVIF entry, the insert data corresponding data with this l3HdrInsrtId XP_GRE_INSERT_HDR_PROFILE insrtPtrs[0], insert data including source ip and destination ip, DSCP from token. For IP-over-IPV4 tunnel: assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry, the insert data use XP_IP_OVER_IP_INSERT_HDR_PROFILE, set DIP and SIP to the insert data.
  - Assign the l4HdrInsrtId to the insPtr2 of this GRE tunnel eVIF entry, the insert data corresponding data with this l4HdrInsrtId XP_GRE_INSERT_HDR_PROFILE insrtPtrs[1] for GRE header insertion,
- No redirect, continue with the normal ingress processing.

Routing—Receives the packet-related information from the previous engine as an input and implements the routing logic. The routing can originate a GRE packet if the EVIF of the Next Hop table entry points to tunnel interface (eVIF).

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, routing lookups can be performed as described below.

- LPM table hit and EVIF of the entry pointer to the GRE/IP-over-IPv4/ MPLSoGRE tunnel EVIF, else
  - Strip the L2 header, token.firstVldLayer + 1.
  - Assign 1 to rewritePtr6IsInsert of token to use Pseudo Layer for inner L2 insertion by Update and Rewrite Engine and assign the enCapType from the Next Hop table.
  - Assign the MacDa,and cVid etc to the pseudo layer to carry to Update and Rewrite Engine.
- LPM table miss or the EVIF is not tunnel eVIF.

Out L2 and L3 insertion pointers and data are predefined and bound to a Tunnel EVIF.

- Create the tunnel eVIF.
  - Assign the l2HdrInsrtId to the insPtr0 of this tunnel EVIF entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and macDA, saLabs, sVid, cVid, and l2EncapType.
  - Assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry, the insert data corresponding data with this l3HdrInsrtId XP_GRE_INSERT_HDR_PROFILE insrtPtrs[0], insert data including SIP and DIP, DSCP from token; for IP-over-IPV4 tunnel, assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry, the insert data use XP_IP_OVER_IP_INSERT_HDR_PROFILE, set DIP and SIP to the insert data.
  - Assign the l4HdrInsrtId to the insPtr2 of this GRE tunnel eVif entry, the insert data corresponding data with this l4HdrInsrtId XP_GRE_INSERT_HDR_PROFILE insrtPtrs[1] for GRE header insertion, and for MPLSoGRE, use XP_GRE_VPN_INSERT_HDR_PROFLE.
- Bond port to list to tunnel VIF, set the corresponding Next Hop table EVIF pointer to tunnel VIF.

MRE—Receives the packet-related information from te previous engine as an input and implements the multicast logic. MRE can originate IP-over-GRE/IP-over-IPv4 packet, if the egressVif of the MDT table entry pointer to tunnel interface (eVIF).

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- Egress VIF of the MDT entry pointer to the IP-over-GRE/IP-over-IPv4 tunnel EVIF, else

    - If encapType of the FDB DA entry is XP_L2_MAC_OVER_IP_OR_PBB_TUNNEL, then assign 0x0800 to etherType and use insert XP_INSERT_ORIGINAL_ETH_HDR.
- The EVIF of MDT is not tunnel EVIF.

Out L2 and L3 insertion pointers and data are predefined and bound to a Tunnel EVIF.

- Create the tunnel eVIF.
    - Assign the l2HdrInsrtId to the insPtr0 of this tunnel EVIF entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and macDA,saLabs,sVid,cVid and l2EncapType.
    - Assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry, the insert data corresponding data with this l3HdrInsrtId XP_GRE_INSERT_HDR_PROFILE insrtPtrs[0], insert data including SIP and DIP, DSCP from token; For IP-over-IPV4 tunnel, assign the l3HdrInsrtId to the insPtr1 of this tunnel EVIF entry, the insert data use XP_IP_OVER_IP_INSERT_HDR_PROFILE, set DIP and SIP to the insert data.
    - Assign the l4HdrInsrtId to the insPtr2 of this GRE tunnel eVIF entry, the insert data corresponding data with this l4HdrInsrtId XP_GRE_INSERT_HDR_PROFILE insrtPtrs[1] for GRE header insertion.
- Bond port to list to tunnel VIF, set the corresponding MDT table EVIF pointer to tunnel VIF.

Update And Rewrite —Receives the packet-related information from the previous engine as an input and implements the update and rewrite header logic. If the EVIF of token if a tunnel VIF, then the Update and Rewrite Engine will encapsulate the tunnel header.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- The engine will search the EVIF table and if the EVIF is tunnel eVIF, else
    - Outer L2 Header insertion: use insPtr0 of the tunnel eVif entry or Pseudo layer
        - → Pick Ether type from Ethernet Layer/Pseudo Layer
        - → Pick SA LSBs from SE Insertion Data
        - → Pick DA from SE Insertion Data
        - → Use constant data to populate TPID if required
    - Outer L3 Header insertion: use insPtr1 of the tunnel EVIF entry
        - → DIP from SE Insertion Data
        - → SIP from SE Insertion Data
        - → DSCP from Token Common Data
    - GRE header insertion: use insPtr2 of tunnel EVIF entry
    - MPLSoGRE header insertion:
        - → MPLS label from the token

- EVIF is not eligible for the GRE tunnel eVIF process, continue with the normal processing.

3.7.3 MPLS Overlay Tunnels

3.7.3.1 Ethernet-over-MPLS

Description

Ethernet services are carried over IP/MPLS networks making use of a wide range of IP-related protocols (see IETF pseudowire standards; e.g., RFC 3985 and RFC 4448). Ethernet links are transported as pseudowires using MPLS label-switched paths (LSPs) inside an outer MPLS tunnel. This strategy can support both point-to-point (Virtual Private Wire Service - VPWS) and multipoint (Virtual Private LAN service - VPLS) services, and has recently achieved significant deployment in routed networks.

The XPliant Software-Defined Platform supports Ethernet-over-MPLS tunnel detection, termination, and origination. For MPLS tunnel, the following types are supported:

- XP_MPLS_SINGLE_LABEL_P2P_TUNNEL
- XP_MPLS_TWO_LABEL_P2P_TUNNEL
- XP_MPLS_SINGLE_LABEL_P2MP_TUNNEL
- XP_MPLS_TWO_LABEL_P2MP_TUNNEL
- XP_MPLS_SINGLE_LABEL_VPN_TUNNEL

Tunnel Termination

Software-Defined Implementation

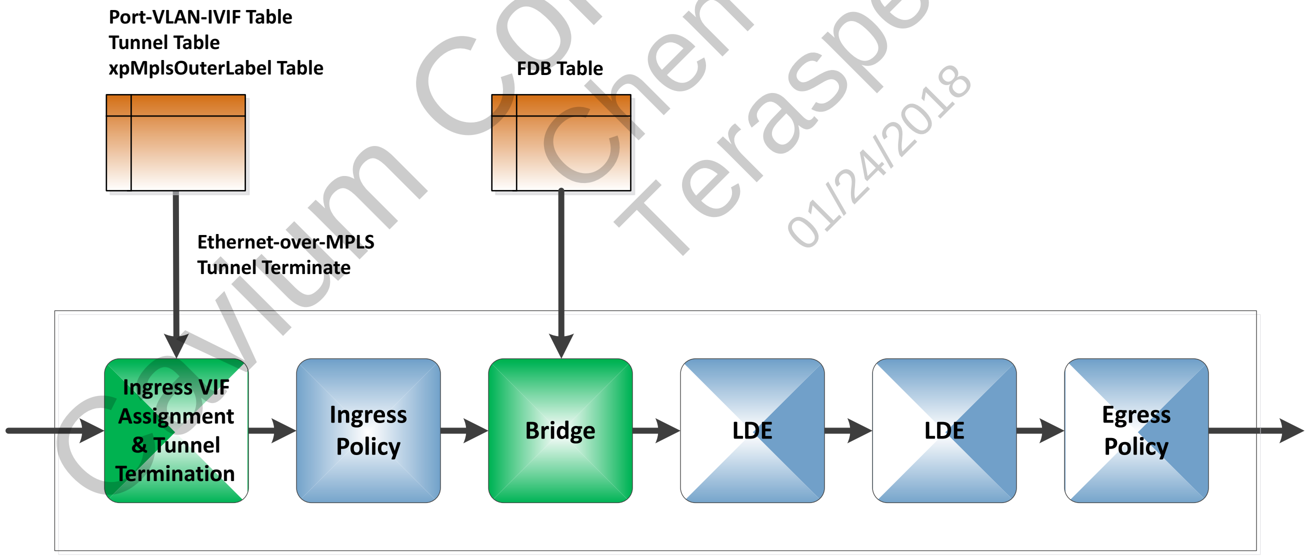


Figure 3–26 Ethernet-over-MPLS Tunnel Terminate Pipeline View

Resource Usage

Table 3–13 IP-over-GRE/IP-over-IPv4 Resource Usage

Engine	Lookups	SE Tables Accessed	SE Tables Result	Comments
Ingress VIF Assignment and Tunnel Termination	1. portVlan 2. xpMplsTnlIvif	PortVlanIvifTable xpMplsTnlIvifTable	Get the Tunnel VIF attributes or PortVlan attributes.	Get xpMplsOuterLabelTable search results from ISME.
Bridge				Lookups are based on inner packet headers.

Decision Logic

Ingress VIF Assignment and Tunnel Termination—Receives the packet-related information from the parser and xpMplsOuterLabelTable search result as an input and implements the Ethernet-over-MPLS termination logic.

This engine will, for valid and eligible for processing packets:

Lookup: For valid packets, Ingress VIF Assignment and Tunnel Termination lookups can be performed as described below.

- Always perform a port VLAN lookup to the PortVlanIvif Table with the key {PORT VIF, OUT VLANID}
  - Get the value of port VIF from ingressVif, which is from the port configuration table from parser.
  - Get the VLAN ID from the ethLayer.
- Perform xpMplsTnlIvif lookup to the xpMplsTnlIvifTable with the key {label0, label1, mplsKey}, only for valid Ethernet packets, which are recognized as MPLS packet and no MPLS header error and TTLl value is not zero and fits in the following conditions:
  - Choose search profile: xpIvifMplsTnlSearchProfile, set mplsKey = XP_TUNNEL_TYPE_MPLS.
  - For single-label MPLS packet and xpMplsOuterLabelTable not hit, get key label0 value from mplsLayer.mplsOuterLblStack.label0.
  - For two-label MPLS packet and xpMplsOuterLabelTable hit out label, get key label0 value from mplsLayer.label1.
  - For two-label MPLS packet and xpMplsOuterLabelTable not hit out label, get key label0 value from mplsLayer.mplsOuterLblStack.label0 and key label1 from mplsLayer.label1.
  - For three-label MPLS packet and xpMplsOuterLabelTable hit out label and xpMplsOuterLabelTable result show this is ThirdLabelVpn, get key label0 value from mplsLayer.label1.

Token—After a lookup has been performed and a packet is IP-over-IP, then terminate it and update the token.

Lookup—The lookup process is as follows.

- If there is a miss in the Port Vlan Table (B.3.1.1 Port VLAN Table), the packet is considered as carrying an invalid ingress vlan (3.4.1 Layer 2 Bridging) for the error handing, else

- If there is a hit in the xpMplsTnlIvifTable table AND PortVlanIvifTable mpls lookup is enable and is VPN label from xpMplsTnlIvifTable search results, then Ethernet-over-MPLS VPN packet is considered as destined "TO ME", then terminate the Ethernet-over-MPLS tunnel else follow the Bridge Domain assignment logic (2.2.2 Bridge Domain).
- Follow the IVIF assignment logicto extract the acm mirror_mask nataclConfig, and portVlanSpanState from the xpMplsTnlIvifTable.
- Assign the aclid from the xpMplsTnlIvifTable table.
- Update the firstVldLayer = firstVldLayer + 1, skip the outer Ethernet layer header.
- Assign token tunnel type to true.
- Forward to the Bridge Engine process.

❍ If MPLS packet is not MPLS, PW Control word and PortVlanIvifTable MPLS lookup is enable else

- If MPLS is single-label packet and xpMplsOuterLabelTable hit or xpMplsTnlIvifTable hit (means two labels tunnel), skip outer Ethernet layer, set tunnel type true, and set innerLabel false.
- If xpMplsOuterLabelTable hit and MPLS is MultipleLabel, set innerLabel true.
- Get attributes from PortVlanIvifTable.
- Forward to the Bridge or Ingress Policy Engine process.

❍ If MPLS packet is MPLS, PW Control word:

- Set isInnerLabel false.
- Follow the bridge domain assignment logic (2.2.2 Bridge Domain).
- Follow the IVIF assignment logicto extract the acm mirror_mask nataclConfig, and portVlanSpanState from the PortVlanIvifTable (B.3.1.1 Port VLAN Table).
- Forward to the Bridge or Ingress Policy Engine process.

Bridge—Receives the packet-related information from the tunnel termination engine as an input and implements the bridge logic. The firstVidLayer pointer is to the inner header of packet.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

For MPLS VPN, MPLS one-label tunnel. and MPLS two-label Ethernet-over-MPLS tunnel, tunnel type is true inner is Ethernet type and inner Ethernet layer is not routerMac, it will use inner Ethernet layer for bridging.

Tunnel Origination

Bridge—Receives the packet related information from previous engine as an input and implements the Bridge logic. Bridge can originate Ethernet-over-MPLS packet if the egressVif of the FDB DA table entry pointer to tunnel interface (EVIF).

This engine will, for valid and eligible for processing packets:

Lookup (VPLS)—For valid packets and VPLS config, Bridge lookups can be performed as described below.

- FDB DA table hit and FDB entry encaptype is L2_OVER_MPLS _TUNNEL else
  - SET etherType = 0x8847 and use XP_INSERT_VPLS_LABEL and evif use fdbTableDa.vif.
  - Forward to EACL process, the nextEngine is EACL.
- FDB DA table miss or FDB entry eVIF is not tunnel eVIF.

Out L2 and MPLS insertion pointers and data are predefined and bound to a tunnel eVIF.

- Create the tunnel eVIF.
  - Assign the l2HdrInsrtId to the insPtr0 of this tunnel eVIF entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and set insert data macDA, saLabs, sVid, cVid and l2EncapType.
  - Assign the l3HdrInsrtId to the insPtr1 of this tunnel eVIF entry, the insert data corresponding data with this l3HdrInsrtId use XP_MPLS_ONE_LABEL_INSERT_HDR_PORIFLE or XP_MPLS_TWO_LABEL_INSERT_HDR_PROFILE and the insert data including firstLabel and secondLabel

Lookup (VLL)—For valid packets and VLL config, bridge lookups can be performed as described below.

Port VLAN config VLL case, isP2PInLif = 1 (B.3.1.1 Port VLAN Table)

- SET etherType = 0x8847 and use XP_INSERT_VPLS_LABEL and eVIF use bdFloodVif.
- Set mplsLable from xpBdTable.bdMplsLabel.
- Forward to EACL process, the nextEngine is URW.

MRE—Receives the packet-related information from the previous engine as an input and implements the multicast logic. MRE can originate Ethernet-over-MPLS packet if the egressVif of the MDT table entry pointer to the tunnel interface (eVIF).

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

- For L2 multicast forwarding, xpMdtBridgeEntry encapType == MDT_ENCAP_MPLS
  - Use insert instruction XP_INSERT_VPLS_LABEL.
  - Set Ethernet type 0x8847 and get egressVif from xpMdtBridgeEntry.
  - Assign tunnelData of MDT table to the EVIF of MDT is not tunnel EVIF.

Out L2 and MPLS insertion pointers and data are predefined and bound to a Tunnel eVif.

- Create the Tunnel eVif

- Assign the l2HdrInsrtId to the insPtr0 of this tunnel eVif entry, the insert data corresponding data with this l2HdrInsrtId use XP_VLAN_UNTAGGED_HDR_PROFILE or XP_VLAN_TAGGED_HDR_PROFILE or XP_VLAN_Q_IN_Q_STAGGED_HDR_PROFILE and set insert data macDA, sVid, cVid, and l2EncapType.
- Assign the l3HdrInsrtId to the insPtr1 of this tunnel eVif entry , the insert data corresponding data with this l3HdrInsrtId use XP_MPLS_ONE_LABEL_INSERT_HDR_PORIFLE or XP_MPLS_TWO_LABEL_INSERT_HDR_PROFILE and the insert data including firstLabel and secondLabel.

❍ Bond port to list to tunnel VIF – Set the corresponding MDT table EVIF pointer to tunnel VIF.

Update and Rewrite —Receives the packet-related information from the previous engine as an input and implements the update and rewrite header logic. If the EVIF of token if a tunnel VIF, then the engine will encapsulate the tunnel header.

This engine will, for valid and eligible for processing packets:

Lookup—For valid packets, bridge lookups can be performed as described below.

❍ URW search the EVIF table and if the EVIF is tunnel EVIF, else
  - Outer L2 Header insertion:
    - → DA from SE Insert Data
    - → SA LSBs from SE Insert Data
    - → EtherType from Eth Layer
    - → IF C-TAG ,C-TAG C-VID from SE Insert Data
  - Outer MPLS Header insertion: use insPtr1 of the tunnel EVIF entry:
    - → MPLS Label 1 from SE Insert Data
    - → MPLS EXP from Token QoS Property
    - → MPLS BOS Bit from Token Common Dynamic ScratchPad
    - → TTL from Token Common Dynamic ScratchPad
    - → MPLS Label 0 from SE Insert Data
    - → MPLS EXP from Token QoS Property for label0
    - → MPLS BOS Bit from Token Common Dynamic ScratchPad for label0
    - → TTL from Token Common Dynamic ScratchPad for label0

❍ EVIF is not Ethernet-over-MPLS tunnel and is not eligible for the eVIF Ethernet-over-MPLS processing.

3.7.3.2 IP-over-MPLS

Description

An MPLS domain consists of two or more Label Edge Routers (LERs) connected by multiple Label Switched Routers (LSR) and LERs connected by four LSRs.

MPLS is used in the service provider network for traffic engineering, bandwidth control, and traffic placement, as well as to achieve fast network resiliency. This is accomplished through IP-over-MPLS features. XDK supports the following feature:

- MPLS label processing for LER and LSR
  - Encapsulation
  - PUSH, POP, SWAP
  - ADD/Delete
  - LSR Label operation

Decision Logic

The following diagram depicts the MPLS packet processing pipeline. The Parser parses the MPLS packet and assigns a template type. Based on configured profile, the LDEs process the MPLS.

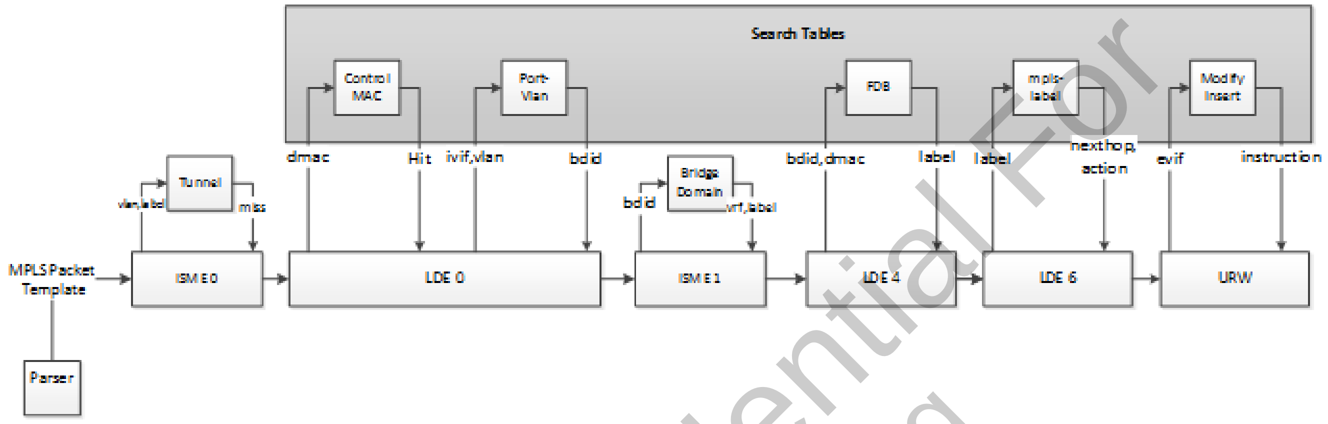


Figure 3–27 IP-over-MPLS Processing Pipeline

Ingress LER packet processing:

1. Ingress LER received unlabeled IP packet over Ethernet.
2. ISME tunnel table lookup determines between MPLS tunnel and MPLS VPN processing
3. To trigger MPLS label lookup, the outer DA MAC should be Router MAC and MPLS Routing must be enabled at the given logical IP interface.
4. Based on following table lookup (FEC), MPLS label is assigned.
5. LER adds MPLS label to the packet based on FDB lookup.

LSR and Egress LER Packet Processing:

1. MPLS label lookup is a part of Route Engine and lookup is done for outermost label only.
2. To trigger MPLS label lookup, the Outer DA MAC should be Router MAC and MPLS routing must be enabled at the given logical IP interface.
3. The following operations are supported as part of MPLS label routing through xpMplsRouteMgr:
   - SWAP—Swap the incoming label with the Next Hop label at a LSR.
   - POP—Pop the incoming outermost label at LER. For PHP (Penultimate Hop Popping), the pop operation is done at the PHP-supported penultimate node (LSR).

   - PUSH—MPLS label Next Hop points to MPLS tunnel VIF that can PUSH the tunnel label (label stacking).
   - SWAP and PUSH—Swap the incoming label with the Next Hop label.
4. The Next Hop can point to the MPLS tunnel VIF that can PUSH the tunnel label.

3.8 Mirroring and Trapping

3.8.1 Description

Well-known control traffic, as well as application-targeted flows, can be mirrored to an analyzer and/or forwarded or trapped to the host CPU.

Each of the packets carries a unique corresponding reason code, which identifies the reason for each type of control traffic and allows the appropriate set of attributes to be assigned according to the Reason Code table (A.1 Default Profile Reason Codes).

A packet can be subject to two types of mirroring:

- *Ingress mirroring*—An originally ingressed packet copy is generated by the Update and Rewrite Engine, which passes it to the Multicast Replication Engine, which resolves the destination. A mirrored copy of the packet will be sent out even if the original packet is to be dropped.
- *Egress mirroring*—A modified packet similar to the egressed packet copy is generated by the Update and Rewrite Engine, which passes it to the Multicast Replication Engine, which resolves the destination. A mirrored copy of the packet will *not* be sent if the original packet is to be dropped.

MAC SA learning and MAC SA movement use mirroring sessions to generate notifications (see 3.4.3 Layer 2 FDB MAC Management to L2 FDB Bridging).

For each packet to be mirrored, at the end of the forwarding processing pipe, the Update and Rewrite Engine will generate a mirror vector based on the:

- Ingress mirroring—Configurable Ingress Mirror Mask AND FinalMirrorMask
- Egress mirroring—Configurable Egress Mirror Mask AND FinalMirrorMask

where FinalMirrorMask is a bitwise OR operation between

- token<mirrorBitMask> is set by any of the software-defined engines
- Configurable Ingress VIF Mirror Mask AND VIF Table <IngressMirrorEn>
- Configurable Egress VIF Mirror Mask AND (VIF Table <EgressMirrorEn> OR (Egress Port Distribution List AND Egress Mirror Port Enable configuration))

In the case of a non-empty mirror vector, a packet will be sent for copy generation to the Multicast Replication Engine with corresponding pointers, token<mirrorMask>, and the packet header.

For example, the Platform Software-Defined logic identifies ARP, ICMP, or IGMP packet types in the Parser Engine and, if the corresponding Bridge Domain <packet command> is set to TRAP, the Bridge Engine will stop the packet processing, jumping to the Update and Rewrite Engine and marking the packet as to be trapped with the corresponding reason code (XP_BRIDGE_RC_IVIF_ARP_CMD /

XP_BRIDGE_RC_IVIF_IGMP_CMD / XP_BRIDGE_RC_IVIF_ICMP6_CMD in A.1 Default Profile Reason Codes). The Update and Rewrite Engine will pass the packet to the destined host CPU.

3.8.2 Encapsulated Remote SPAN (ERSPAN)

The ERSPAN brings generic routing encapsulation (GRE) for all captured traffic and allows it to be extended across Layer 3 domains. The ERSPAN source sessions copy traffic from the source ports or source VLANs and forward the traffic using routable GRE-encapsulated packets to the ERSPAN destination session. The ERSPAN destination session switches the traffic to the destination ports.

The packet processing of ERSPAN in the packet pipeline is same as in the Mirroring and Trapping section. To process the ERSPAN, the packet pipeline creates an outgoing scratch pad in MME and copies the ERSPAN ID and the source port from the MDT entry to the scratch pad in MME and sends them to the URW. The URW further fills in the ERSPAN header before sending the packet out.

3.9 Control Plane Policing (CoPP)

3.9.1 Description

Control plane policing (CoPP) is designed to allow users to manage the flow of traffic handled by the CPU of their network devices. The feature increases security on the switch by protecting the control and management planes from unnecessary overwhelming or DoS traffic and giving priority to important control plane and management traffic. CoPP helps to ensure control plan stability and forwarding plan reachability and packet delivery.

CoPP uses dedicated control plane configurations to provide protection through applying filtering, rate limiting, and prioritization mechanisms on the classified control traffic destined to the CPU. In addition, combining the policer, reason code, and queue consumption-level counters provides visibility into the real-time behavior of the classified control traffic.

CoPP involves rate limiting and prioritizing classified control traffic destined to the CPU. The main components are:

- Ingress Traffic Protocol classification—Performed as part of Parser and LDEs logic.
- Ingress Traffic Policing and Rate-Limiting—Performed based on user configuration through the interface to the ACM block from the EACL Engine LDE.
- Egress Traffic Prioritization into different CPU queues—Performed based on the unique reason codes (rc) assigned to classified control protocols to categorize the traffic into CPU queues.

NOTE: XDK also supports reason code remapping and traffic mirroring; see details below.

Packet Flow Across LDEs

Protocol Classification (Reason Code)	—>	`SourcePort` + Reason Code Table Lookup (`PolicerId`/`PktCmd`/ `mirrorSessionId`/) (`ReasonCode`)	—>	Prioritizing packets onto different TxQ CPU queues based on Reason Codes
Logic in Parser and Bridge/ Route Engine		Logic in EACL Engine Policing done by ACM block		Default reason code to Q map. Scheduling priority configuration at H1/H2 levels for CPU Q's

Source Port/Reason Code Table Lookup

This table lookup is issued only if the previous engines set the control bit in the scratchpad, indicating it is a control packet.

A shared hash table of one SRAM Tile (i.e., 32K x 64-bit wide entries) is allocated for this lookup. It contains a 32-bit key and 32-bit data.

Key: Source Port (8 bits)
Reason Code (10 bits)

Data: `EnPolicer` (1 bit)

- Control bit to enable policing or not.

`PolicerId` (10 bits)

- Used for rate-limiting of the control traffic destined to CPU.
- The PolicerId must be programmed accordingly indexing an ACM bank.

  Note: Two ACM banks are allocated for policing; i.e., 2*512 entries.

`UpdatePktCmd` (1 bit)

- Control bit to enable packet command update.

`PktCmd` (2 bits)

- Used for rate-limiting of the control traffic destined to CPU.
- Used to control the packet forwarding behavior.

`UpdateMirrorSessionId`

- Control bit to enable mirroring.

`MirrorSessionId` (2 bits)

- Used to support mirroring of the control traffic destined to the CPU to an Analyzer.

`UpdateReasonCode` (1 bit)

- Control bit to enable reason code remapping.

`ReasonCode` (10 bits)

- Reason Code to be assigned in case of a match for entries with UpdateReasonCode = TRUE.

3.9.2 Packet Prioritization

Packet prioritization can be achieved by assigning scheduling priorities (DWRR weights or strict priority) to queues at H1/H2 level in the DQ tree.

NOTE:

- The current implementation maps 1024 Reason Codes into 128 CPU queues; i.e. there is 8:1 ratio of mapping the Reason Codes into a single CPU queue. By default, XDK configures the same DWRR weights for all CPU queues, which results in round robin type of behavior among the CPU queues. In addition, today only one H2 node and four H1 nodes are used for this purpose.

  We are considering enhancing this functionality by adding more flexibility through separating Q groups across multiple H1s (8 H1s), which will allow mapping unique group of Reason Codes to a unique Q groups and assign priorities to queues at H1/H2 nodes. The current Reason Code numbering can be changed, in order to grouping them into different Reason Code groups/ buckets. Reason code groups can be 1-1 mapped to Q groups.

  This change is under discussion and has not yet been committed.

- For CPU trapped packets and BUM traffic that enables CoPP lookup, only BD and CoPP table lookup will be done as part of the EACL Engine. EACL and Egress Qos Map table lookup will be skipped.
- Reason Codes to CPU queue mapping are as below for 128 CPU queues. There is a 8:1 ratio of mapping the Reason Codes into a single CPU queue.

  RC 0-7 Qnum 0
  RC 8-15 Qnum 1
  RC 16-23 Qnum 2 and so on…
- The number of CPU queues (64/128) depends on the SKU mode. For 64 queues, the mapping changes correspondingly. There is a 16:1 ratio of mapping the Reason Codes into a single CPU queue.
- The number of mirror session IDs will be reduced from 4 to 3 as part of fitting CoPP feature.
- The feature uses a shared hash table with Egress QoS Map feature.

3.9.3 BUM Traffic Policing

BUM traffic policing can be achieved using the COPP table by enabling the policer and assigning a `policerId`. Per Source Port knob is provided to enable BUM policing using the COPP table. A bit `bumPolicerEn` must be set as part of `portConfig` configuration.

3.10 QoS

The device integrates an advanced traffic manager with four levels of scheduling hierarchy, active congestion management, and flexible queue mapping, enabling fine grain QoS support as well as per tenant queuing.

The flexible queue mapping allows output queues, input queues, or virtual output queues architectures. The flexible traffic manager configuration supports flow-based, VLAN/tenant-based, and priority-based queuing scheme implementations.

3.10.1 Active Queue Management (AQM)

The Traffic Manager uses a notion of AQM Q profiles, which dictates how Active Queue Management takes place on a queue. AQM is a series of methods used to manage resources and congestion at egress queues. It is the fundamental method of allocating packet memory to queues and controlling congestion by virtue of this allocation.

Active Queue Management incorporates the following functionality:

- Dynamic queue memory allocation with shared memory pools.
- Tail drop queuing.
- Random early discard queuing.
- Random Early Marking of ECN field on IP header of TCP packets.
- DCTCP (Data Center TCP) marking of the ECN field on IP header of TCP packets.
- Phantom-queues-based ECN marking.

3.10.1.1 Dynamic Queue Memory Allocation

Along with allocating dedicated resources to a queue, the SDK introduces the concept of a queue shared-resource pool. This shared-resource pool accounts for pages used by a series of queues associated with it, effectively increasing the burst capacity of a single queue without tying up pages for that queue.

There are a fixed number of pages and tokens on a device that can be used at the same time. Usually, the number of pages adds up to the size of the total packet memory. When implementing QoS, a panic level may be required for a case of a storm of traffic, which eats up tokens. To prevent the device from hanging, global thresholds for packets and pages can be used, limiting the total number of tokens that can be used to implement that panic level so that there will be tokens available on a device for ingressing packets.

Similarly, additional thresholds are supported to protect the proper operation and performance of the device:

1. Multicast packet threshold—Limits the number of multicast pages in the system at a given time. It can be useful to limit the impact of multicast traffic with respect to overall system performance.

3.10.1.2 Tail Drop

Tail drop is the fundamental method of resource allocation used in switches today. It relies on instantaneous feedback of queue utilization to dictate whether or not a packet should be enqueued. Unlike WRED, which uses historical data of queuing, tail drop uses only instantaneous information.

For example:

If a tail drop threshold is set to 120 pages and the queue at time T currently has 115 pages enqueued, at time T+1 a packet of size 6 pages comes in and requests whether or not it can be enqueued, the tail drop threshold will be crossed for this packet and AQM should not allow for it to be enqueued

The SDK supports implementing tail drop thresholds in pages (which is a byte resolution) as well as in packets. Both thresholds can co-exist simultaneously, and either one can decide to drop a packet. The SDK supports tail-dropping thresholds on per port and per queue levels. The port tail-drop mechanism compares the aggregate queue utilization against the port tail drop threshold.

3.10.1.3 WRED

The device supports three distinct WRED modes:

- Tail drop only/Disable WRED
- WRED drop
- WRED Mark ECN

Weighted Random Early Detection (WRED) is an Active Queue Management scheme that is used to smooth out congestion due to a bursty flow over a period of time. This smoothening happens by virtue of detecting congestion “early”. In other words once a queue starts to build, the Queuing system will start to randomly drop or mark packets at a well-known rate. This rate grows linearly with the average queue length. It means that as the queue builds up over time, the likelihood of dropping or marking a packet increases up until crosses a max threshold. After this max threshold is crossed, a packet should be either dropped or marked with ECN.

WRED is different from regular RED in that a weight can be applied on the calculation of the average queue length at the time of AQM.

If the weight is a large number, it biases the new average q length calculation in favor of the old length, and vice versa.

The calculation of the average queue length formula, as well as the WRED profile curve, are explained in the Functional Specification document.

3.10.1.4 Data Center TCP

Data Center TCP (DSTCP) is a feature used by TCP to indicate congestion while minimizing latency in a switch. DCTCP allows for shallow queue depth and marks ECN when queue depth builds, sending ECN to the sender or receiver of a TCP session and notifying the TCP stack to shrink or expand the TCP congestion window according to queue utilization on a switch in the network.

Because ECN marking is always forward facing, DCTCP institutes a protocol to notify the sender to shrink its congestion window, thereby reducing the amount of traffic in the system. This is useful for reducing overall packet latency within a network because the sender and receiver have a much better picture of congestion. For instance, when a low latency queue is starting to build, which would add average latency to a TCP session, the sender will be notified by the received implementing DCTCP to shrink its window and, by virtue of a smaller window, reduces the number of packets sent.

ECN is used for real-time notification status of a switch in the network congestion. The SDK implementation of DCTCP marking allows for a user to configure a low mark threshold for a queue independent of the queue resource allocation. If a queue builds up past its DCTCP mark threshold, ECN will be marked on subsequent packets.

The SDK supports DCTCP marking and allocation on a queue and at port level. If a port detects congestion based off of its DCTCP mark threshold, all packets egressing from that queue will be marked with ECN.

3.10.1.5 Phantom Queue Marking

Phantom queuing is another method used to prioritize lower latency traffic in the system and maintain shallow queue depths. Its high-level implementation involves a "phantom" or "shadow" queue apart from the regular queue.

Phantom queuing uses the notion of link utilization instead of buffer occupancy to detect congestion. When a link is being heavily used, phantom queuing will mark ECN. This is done effectively by treating the phantom queue as a lower speed queue (link) than the real queue.

NOTE: The SDK implementation of phantom queuing requires a queue shaper set at a lower speed than the desired link occupancy, which will trigger marking when the throughput exceeds the shaping rate.

3.10.2 Policing and Shaping

Shaping implies the existence of a queue and of sufficient memory to buffer delayed packets, while policing does not. Queueing is an outbound concept: packets going out an interface get queued and can be shaped. Only policing can be applied to inbound traffic on an interface.

Ensure there is sufficient memory when enabling shaping. In addition, shaping requires a scheduling function for later transmission of any delayed packets. This scheduling function allows you to organize the shaping queue into different queues. Examples of scheduling functions are Class-Based Weighted Fair Queuing (CBWFQ) and Low-Latency Queuing (LLQ).

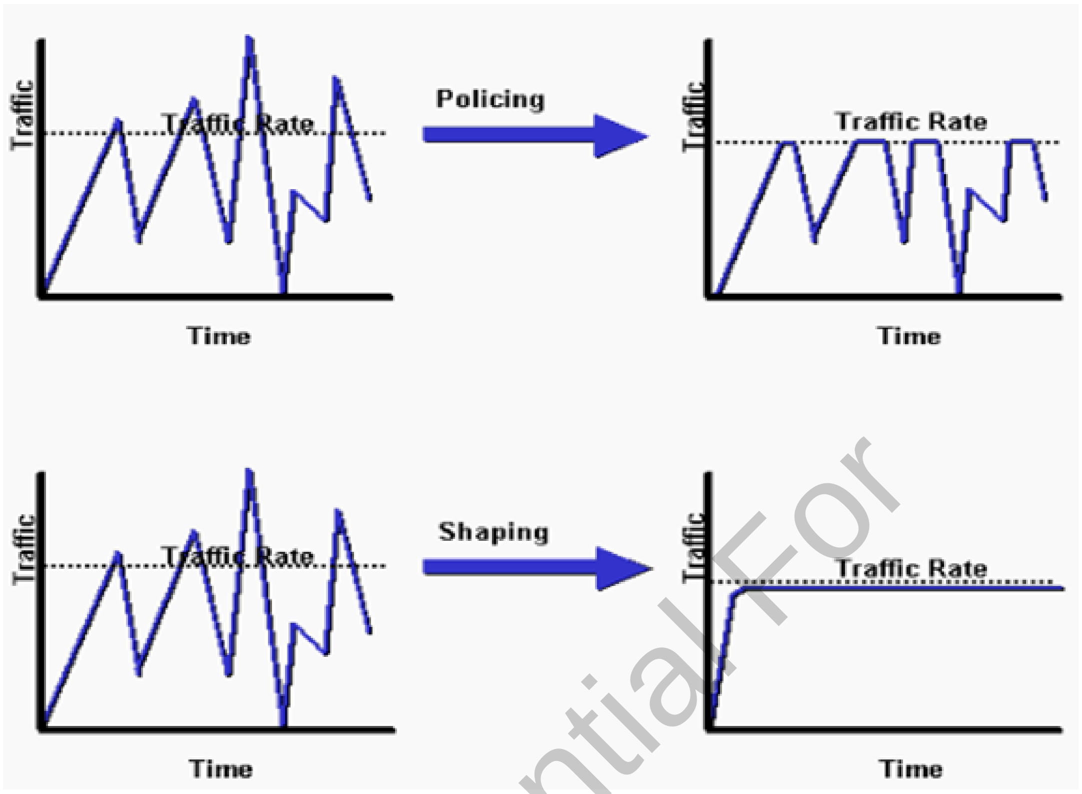


Figure 3–28 Policing and Shaping

Traffic shapers are used to condition traffic patterns to well-known behaviors. A shaper enforces two principles on traffic:

- Egress traffic rate will be well known.
- A burst will be smoothed over a period of time.

These two principles behind shapers guarantee a deterministic traffic pattern inside a network.

The following figure demonstrates the shaper impact on an incoming traffic flow:

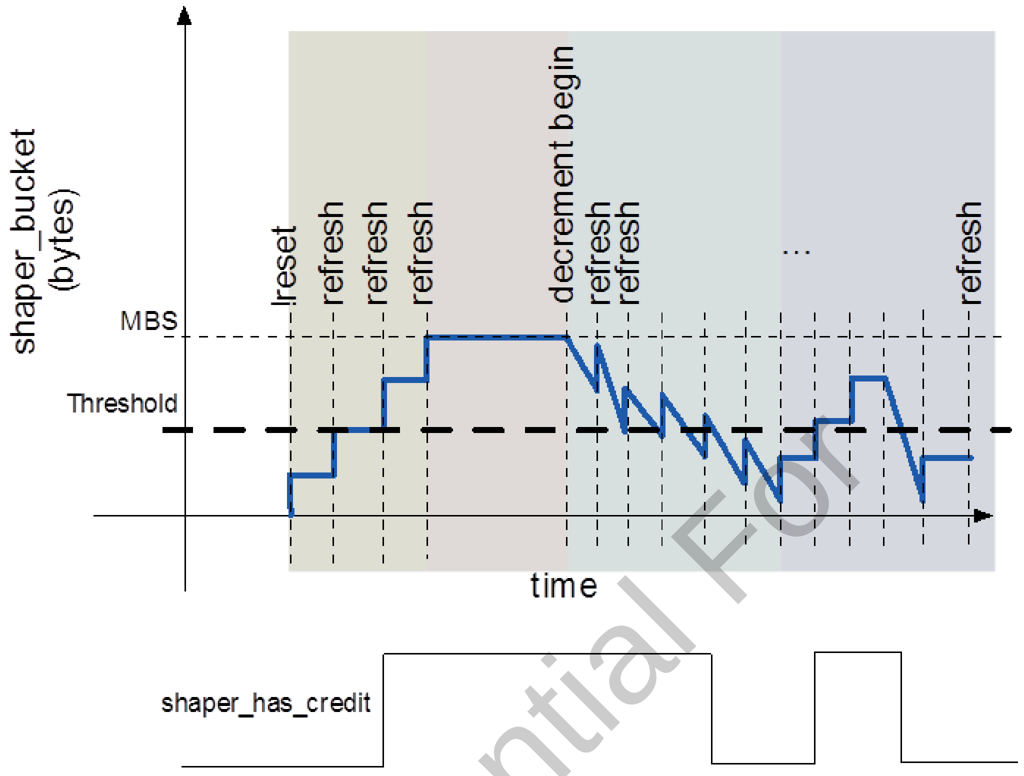


Figure 3–29 Shaper Impact on Incoming Traffic Flow

Token Refresh Rate

A key difference between shaping and policing is the rate at which tokens are replenished. This section reviews the difference.

Simply stated, both shaping and policing use the token bucket metaphor. A token bucket itself has no discard or priority policy. Consider how the token bucket metaphor works:

- Tokens are put into the bucket at a certain rate.
- Each token is permission for the source to send a certain number of bits into the network.
- To send a packet, the traffic regulator must be able to remove a number of tokens from the bucket equal in representation to the packet size.
- If not enough tokens are in the bucket to send a packet, the packet either waits until the bucket has enough tokens (in the case of a shaper) or the packet is discarded or marked down (in the case of a policer).
- The bucket itself has a specified capacity. If the bucket fills to capacity, newly arriving tokens are discarded and are not available to future packets. Thus, at any time, the largest burst a source can send into the network is roughly proportional to the size of the bucket. A token bucket permits burstiness, but bounds it.

With the token bucket metaphor in mind, look at how shaping and policing add tokens to the bucket.

Shaping increments the token bucket at timed intervals using a bits per second (bps) value. A shaper uses the following formula:

```
Tc = Bc/CIR (in seconds)
```

3.10.3 Scheduling

The device supports Traffic Manager Scheduling in DWRR mode (Deficit Weight Round Robin) and/or SP (Strict Priority) modes.

The arbitration is done in two stages:

Stage 1 – DWRR

DWRR weight must be assigned if DWRR is enabled on the queue. A strict priority also must be assigned for queues as this is required for the next stage.

Stage 2 – Strict Priority (SP)

Queues only assigned with SP only participate in stage 2.

Queues with the same Strict Priority assigned will have their packets scheduled in a round robin fashion.

DWRR arbitration is done based on the weights assigned to queues. The DWRR result is fed to the SP arbiter, which chooses the queue with the highest priority.

The following diagrams illustrate how the two-stage arbitration behaves over a period of time.

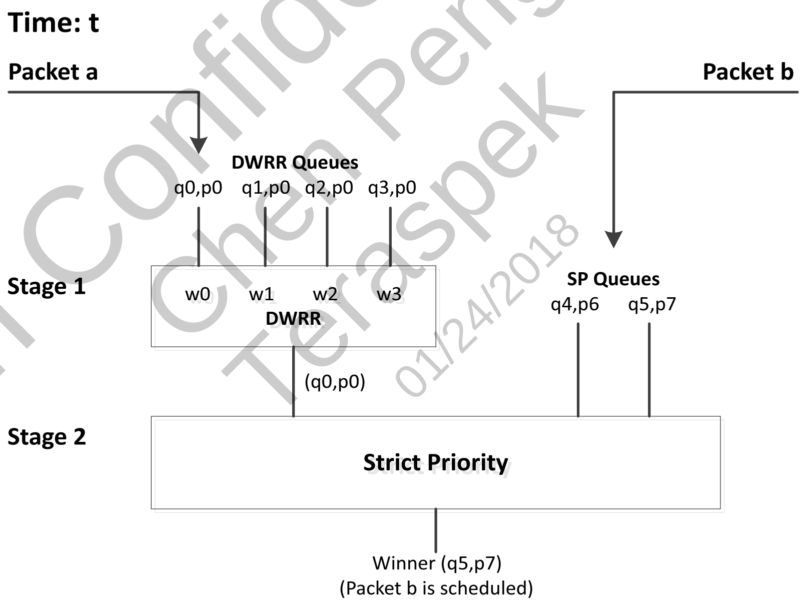


Figure 3–30 DWWR Scheduling (Time: t)

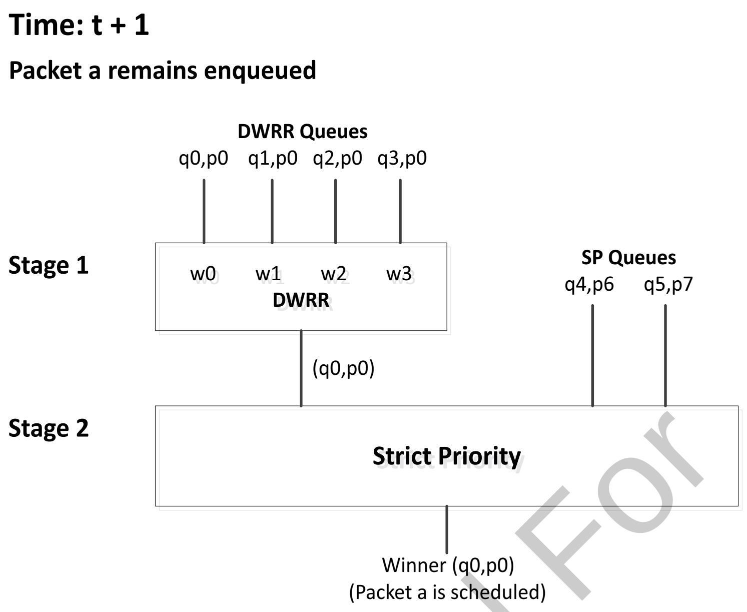


Figure 3–31 DWWR Scheduling (Time: t+1)

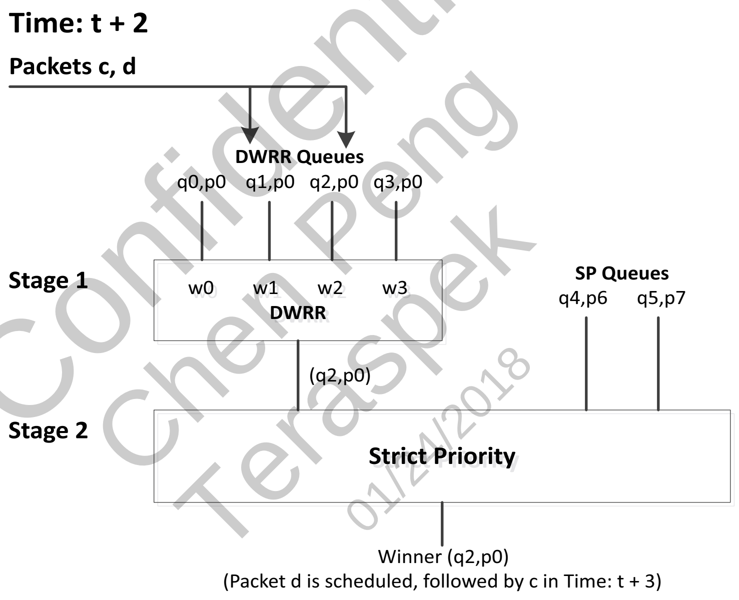


Figure 3–32 DWWR Scheduling (Time: t+2)

After an arbitrary length of time, time x is reached and the Strict Priority of q0 changes to 7:

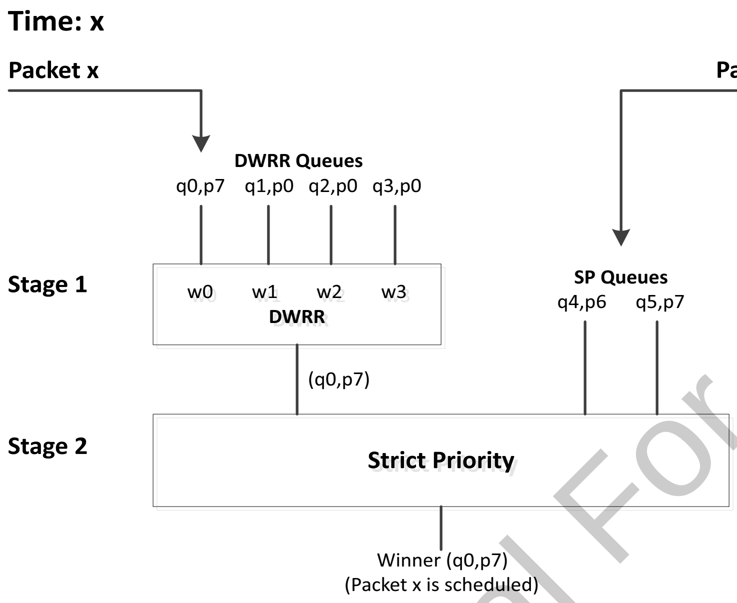


Figure 3–33 DWWR Scheduling (Time: x)

NOTE: The above figures assume the following:
- Weights: w0 < w1 < w2 < w3
- Priorities: p0 < p1 < p2 ... < p7

3.10.4 Shaping

Traffic shapers are used to condition traffic patterns to well-known behaviors.

A shaper enforces two principles on traffic:

- Egress traffic rate will be well known.
- A burst will be smoothed over a period of time.

These two principles behind shapers guarantee a deterministic traffic pattern inside a network.

The following figure demonstrates the shaper impact on an incoming traffic flow:

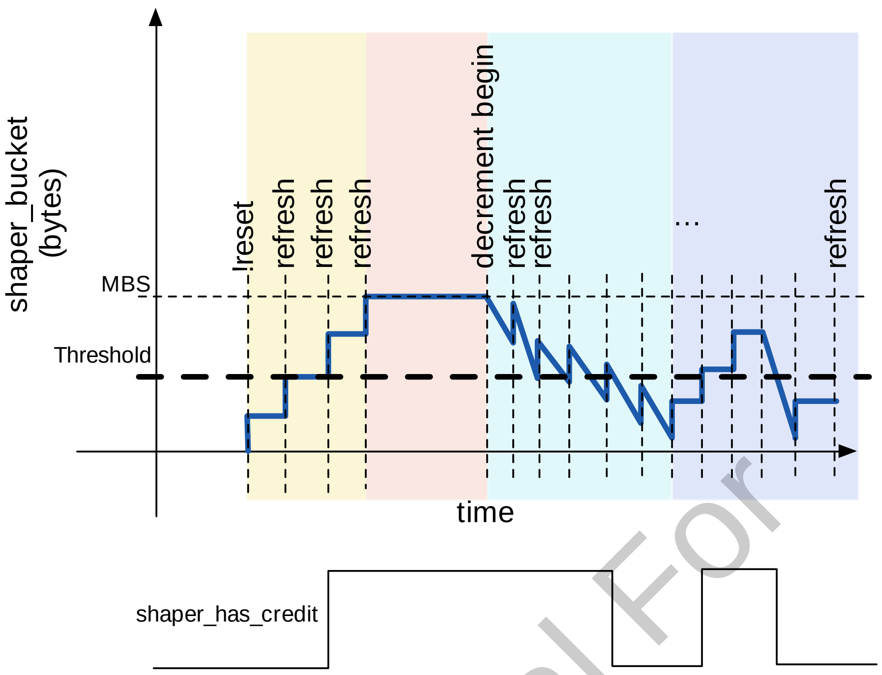


Figure 3–34 Shaper Impact on Incoming Traffic Flow

As shown in the example above, at time t0 a max burst was received, with shaping disabled,

The burst would be propagated within the network. The graph trends downwards after time t0 to illustrate the smoothing out of the burst over a time delta. At time t1 another mini burst was received and also subsequently smoothed out.

At time t2 the traffic pattern hits the expected shaper rate, which will be maintained on average throughout.

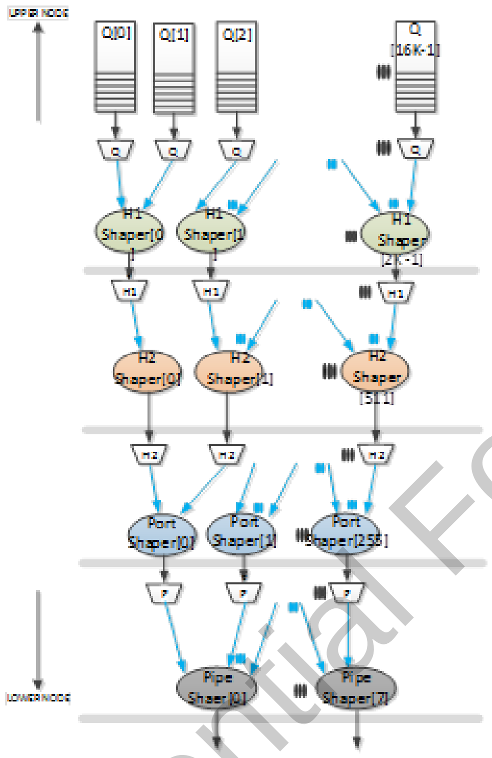


Figure 3–35 Q Shapers—Fast vs. Slow

TxQ supports traffic shaping for Q and Port. The following are the shaping parameters:

- Max Burst Multiplier (8b) - (in terms of 2^ShaperMTUCfg (4b))
- Update Rate (3b)
- BytesToAdd (14b)
- Threshold (24b)
- Qslow desired rate bps = (8 * BytesToAdd * clockFreq) / (512 * updateRate)
- Qfast desired rate bps = (8 * BytesToAdd * clockFreq) / (40 * updateRate)
- Port desired rate bps = (8 * BytesToAdd * clockFreq) / (32 * updateRate)

Table 3–14 Shaping Parameters

Shaper	Min Rate	Max Rate	Comments
Qslow (upd_rate=7)	1.5Mbps	21Gbps	Use upd_rate=7 for <1Gbps rates
Qslow (upd_rate=1)	12Mbps	75Gbps	<30Gbps recommended
Qfast (upd_rate=5)	20Mbps	100Gbps	160 fast Qs
Port (upd_rate=7)	25Mbps	100Gbps	

Egress Traffic Shaping—The Token Bucket Scheme

Egress traffic shaping is different from policing in terms of their implementation.

A shaper implies that there is a queue or FIFO that can allow for traffic attached to an arbiter or scheduler. A shaper will then play a role in the scheduler decision-making process to see if that queue or FIFO can dequeue a packet. The SDK uses a token bucket scheme for the shaper implementation, in contrast to policing, which uses a "leaky token bucket" scheme.

A token bucket scheme is a way of limiting the rate of dequeue of a FIFO. This method relies on a crediting scheme: each token corresponds to a credit, which is the equivalent to a byte of a packet. When a packet is ready to be dequeued and shaping is enabled, the scheduler will first determine how many tokens are required to dequeue this particular packet. It will then compare the number of tokens required against the number of remaining tokens currently in the token bucket. If there are enough tokens in the bucket without crossing the minimum threshold, this packet will pass the shaper check and will be scheduled out. Because every packet to be dequeued will require the bucket to contain enough tokens to account for the packet size, and the token bucket itself is of a fixed size, there must be a method for refreshing the token bucket at a defined interval. Since a shaper also meters outgoing traffic to a fixed rate, simply filling the token bucket all the way every refresh cycle is not adequate. To ensure that traffic is metered correctly, the shaper will need to be configured to add a fixed number of tokens per refresh cycle that corresponds with the desired output rate of traffic.

A snapshot of a token bucket at two particular times is shown below:

```
* Packet x: 500 bytes
* ---------------
*               |
*               |
*              \/
* saturate - |________________|   Key:
*            |                |   +:        10 Tokens
*            |+++++++++       |   saturate: Num tokens when
*            |++++++++++++++++|             bucket is full
*            |++++++++++++++++|   min:      Minimum tokens
*            |++++++++++++++++|             required to
*            |++++++++++++++++|             schedule
* min -      |________________|
*             \++++++++++++++/
*              \____________/
*                     |
*                     | Packet x Scheduled
*                    \/
*
* Time: t + 1
* Packet y: 500 bytes
* ---------------
*               |
*               |
*              \/
* saturate - |________________|   Key:
*            |                |   +:        10 Tokens
*            |                |   saturate: Num tokens when
*            |                |             bucket is full
*            |                |   min:      Minimum tokens
*            |+++++++++++++++ |             required to
*            |++++++++++++++++|             schedule
* min -      |________________|
*             \++++++++++++++/
*              \____________/
*                     X
*                     X  Packet y Remains in the Queue
*
* Time: t + UR
* saturate - |________________| /\ Key:
*            |++++++++        | |  +:        10 Tokens
*            |++++++++++++++++| |  saturate: Num tokens when
*            |++++++++++++++++| |            bucket is full
*            |++++++++++++++++| |  min:      Minimum tokens
*            |++++++++++++++++| |            required to
*            |++++++++++++++++| |            schedule
* min -      |________________| |
*             \++++++++++++++/  |
*              \____________/   | Token Update +570 Tokens
*                     X
*                     X  Packet y Remains in the Queue
*
* Time: t + UR + 1
* saturate - |________________| /\ Key:
*            |                | |  +:        10 Tokens
*            |                | |  saturate: Num tokens when
*            |                | |            bucket is full
*            |++++++          | |  min:      Minimum tokens
*            |++++++++++++++++| |            required to
*            |++++++++++++++++| |            schedule
* min -      |________________| |
*             \++++++++++++++/  |
*              \____________/   | Token Update
*                     |
*                     | Packet y Scheduled -500 Tokens
*                    \/
```

Figure 3–36 Token Bucket Snapshot

The above diagram illustrates the following example:

- The token bucket size is 960.
- There are 730 tokens available at time t.
- A 500 byte packet x arrives at time t.
- Packet x will be scheduled because it can take 500 tokens without going below min.
- At time t + 1, packet y arrives; it is also 500 bytes.
- Packet y will not be scheduled because the token bucket will only have 230 tokens at time t + 1. It will remain in the queue.
- At time t + UR (where UR is the update rate), a token refresh occurs and adds 570 tokens to the bucket.
- At time t + UR + 1 packet, y will be scheduled because the new token bucket size is 230 + 570 = 800.
- The final token bucket size is 300.

This example demonstrates the impact of three major configurations for every token bucket

- Token bucket size.
- Token bucket refresh rate.
- Token bucket tokens to add every refresh period.

The token bucket size reflects the maximum burst size that can be absorbed and smoothed over a period of time.

The tokens to add every refresh period, along with the token bucket refresh rate, are used to fix the average dequeue rate.

Fast and Slow Shapers

Each node that can be scheduled by the Traffic Manager can also have its dequeue rate shaped, which will guarantee a fixed, well-known throughput and smoothing of a burst. To achieve this, the device uses memory dedicated for its token bucket implementation at every scheduling hierarchy. There are two types of shaper memories at most nodes:

- Fast
- Slow

The SDK also implements a global flag to enable the shaper feature that must be turned on for queues and ports independently. If this flag is not set, shaping will not take effect.

In addition, Maximum Transmission Unit (MTU) configuration is used to set the minimum token bucket size.

Traffic shapers are used to condition traffic patterns to well known behaviors. A shaper enforces two principles on traffic:

- Egress traffic rate will be well known.
- A burst will be smoothed over a period of time.

These two principles behind shapers guarantee a deterministic traffic pattern inside a network.

The following figure demonstrates the shaper impact on an incoming traffic flow:

```
* Rate     |
* ======   |
* Initial  |- - -___  - - - - - - -
* Burst    |    /   \
*          |   /     \  /\
* Shaper   |- /- - - -\/- \_______
* rate     | /
*         _|/____|_______|_|_______
*          |    t0      t1 t2  time
```

Figure 3–37 Shaper Impact on Incoming Traffic Flow

As shown in the example above, at time t0 a max burst was received, with shaping disabled, The burst would be propagated within the network.

The graph trends downwards after time t0 to illustrate the smoothing out of the burst over a time delta.

At time t1 another mini burst was received and also subsequently smoothed out.

At time t2 the traffic pattern hits the expected shaper rate, which will be maintained on average throughout.

3.11 Counting, Policing, and Sampling

The Analytics Complex Module (ACM) is used for counting, policing, and sampling of the flows on the device. The SDK provides full set of APIs for retrieving or configuring the counter statistics, policing, and sampling entries.

An ACM is incorporated in each SDE. The module contains a large amount of generic purpose counters blocks, which can be fused for independent and parallel counting of events, for analytics purposes, sampling, policing, or debug events.

NOTE: ACM architecture restricts the policing configuration to only the first lane of each LDE, which is also enforced during initialization.

There are four ACM lanes per LDE. Each lane can be used for sampling, policing, or counting purposes. The configuration and bank sizing / partitioning are software defined and loaded at boot time. The client interfaces from LDEs to ACM are evenly distributed. Each counting, policing, or sampling entry of a given flow is uniquely accessible for different tables with a unique entry index into corresponding ACM bank entries. The same configuration is replicated for multiple pipelines on different LDE.

The default ACM configuration is lane 0 is for policing, lanes 1 and 2 are for counting, and lane 3 is for sampling. There is one ACM lane for the MME. There are 32 banks in the ACM that can be divided between each analytic engine. Currently, policing is bank 0-10, sampling is bank 11-21, counting is bank 22-30, and bank 31 is for the MME.

The default ACM configuration is described in the table below:

- LDE/MME Lane: specifies the ACM lane associated with the LDE
- configMode: defines the configuration mode (policing / counting / sampling)

- Bank Start/End: The banks used for the configuration mode
- Dual Bank: Designates the next bank of policer to be used for associated counting
- Count Offset: Count to be added for incoming counter value (for CRC)
- Wrap Around: Decides whether to remain at max value or wrap around to zero
- Clear on Read: Resets the counter after it is read
- Bank Mode: Sets the associated banks for each lane to one of the five count modes, or the police or sample mode

`XP_ANA_BANK_MODE_A` :	*counter mode A (136b x 1k entries)*
`XP_ANA_BANK_MODE_B` :	*counter mode B (68b x 2k entries)*
`XP_ANA_BANK_MODE_C` :	*counter mode C (34b x 4k entries)*
`XP_ANA_BANK_MODE_D` :	*counter mode D (17b x 8K entries)*
`XP_ANA_BANK_MODE_P` :	*police without billing counter (272b x 512 entries*
`XP_ANA_BANK_MODE_PC` :	*police with billing counter (270b x 512 entries)*
`XP_ANA_BANK_MODE_S` :	*sampling (272b x 512 entries)*

Table 3–15 Default ACM Configurations for an LDE/MME

LDE/ MME Lane	configMode	Bank Start	Bank End	Dual Bank	Count Offset	Wrap Around	Clear On Read	Bank Mode	out Order	LDE PipeLine 0
0	XP_ACM_POLICING	0	10	0	4	1	1	XP_ANA_BANK_MODE_P	1	LDE 0
1	XP_ACM_COUNTING	22	30	0	4	1	1	XP_ANA_BANK_MODE_P	1	LDE 0
2	XP_ACM_COUNTING	22	30	0	4	1	1	XP_ANA_BANK_MODE_B	1	LDE 0
3	XP_ACM_SAMPLING	11	21	0	4	1	1	XP_ANA_BANK_MODE_S	1	LDE 0
48	XP_ACM_POLICING	31	31	0	4	1	1	XP_ANA_BANK_MODE_P	1	MME 0

For policing, the device supports two-rate, three-color (trTCM) policing in color-aware or color- blind mode. The following settings can be configured:

- Color aware or color blind.
- Peak information rate (PIR).
- Commited information rate (CIR).
- Time granularity.
- Commited burst size.
- Mark red, yellow, and green packet.
- Drop red, yellow, and green packet.
- Enable counter based on color.

For counting, the following methods are supported:

- Per port ingress policing.
- Per port ingress sampler.
- Ingress BD counter.
- VLAN BUM traffic policer.

- Ingress ACL Policer; common for PACL, BACL, and RACL.
- Ingress PACL counter.
- Ingress BACL counter.
- Ingress RACL counter.
- Egress ACL counter.
- Drop reason code counter.
- Port VLAN counter.
- Tunnel termination counter.
- Egress BD counter.

3.12 Multicast Replication Engine (MRE) / Multicast Forwarding

3.12.1 Description

There are two levels of replication. Unique packet replications are performed by the programmable Multicast Replication Engine (MRE), which goes through a linked list of nodes and duplicates a packet per every node. Packets with the same encapsulation are replicated in the Traffic Manager.

A packet may be subject to Multicast Replication Engine processing for various reasons, such as:

- Ingress mirroring copy.
- Egress mirroring copy.
- Packet must be modified and duplicated to more than one destination port.

Packet replications are required for implementing features such as:

- Layer 2 replications (multicasts or broadcasts).
- Layer 3 multicast forwarding to outgoing interfaces.
- VPLS multicasts.
- Forwarding tunneled packets to remote site or station (VXLAN Multicast, etc.).
- Mirroring packets to multiple analyzers.

The replication is achieved using a two-dimensional linked list and is based on the Multi-Distribution table (MDT) (B.2 Multicast Distribution Table).

MDT is a software-defined table that is used by the MRE. During system initialization this table must be created by the primitive layer through the table manager.

MDT implements the multicast egress destination. The mVif index to this table is stored in various other forwarding table entries, such as IPv4 bridge or route entry, corresponding to a multi-port destination.

A multicast Outgoing Interface (OIF) list is implemented as a two-directional linked list within this table using the Next L3Ptr and Next L2Ptr; all entries within this table that belong to the same multicast distribution are linked to each other through these pointers. The MRE makes a copy of the incoming packet for each MDT node.

The horizontal list (L2Ptr) represents the different multicast VIFs for the same L3 interface and the vertical list (L3Ptr) represents different L3 interfaces. The L3Ptr is valid only for the first node in each row.

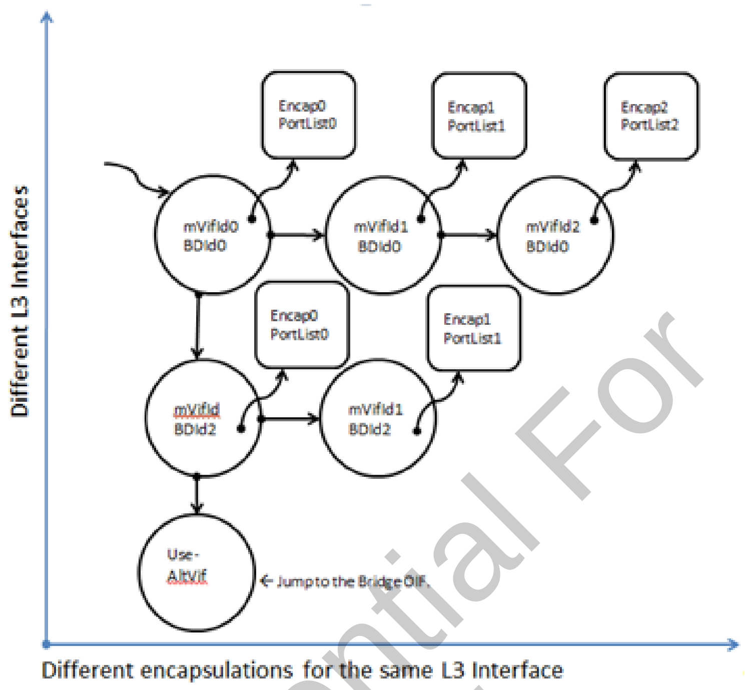


Figure 3–38 MDT Routing OIF List

Each multicast VIF essentially represents a pair (outgoing encapsulation type, port-list). In addition, the multicast VIF can point to an MDT node to provide indirections and hierarchies. This architecture allows for maximum flexibility in programming and managing of resources through their sharing across multiple flows.

Each node in the list points to an EVIF, which must be created prior to setting this entry.

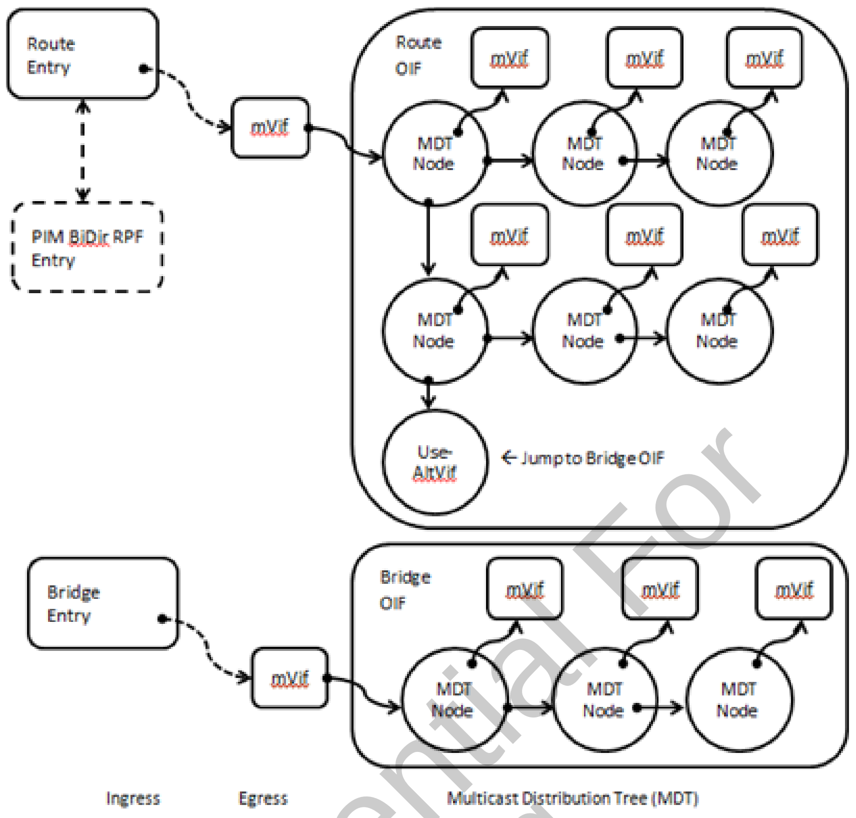
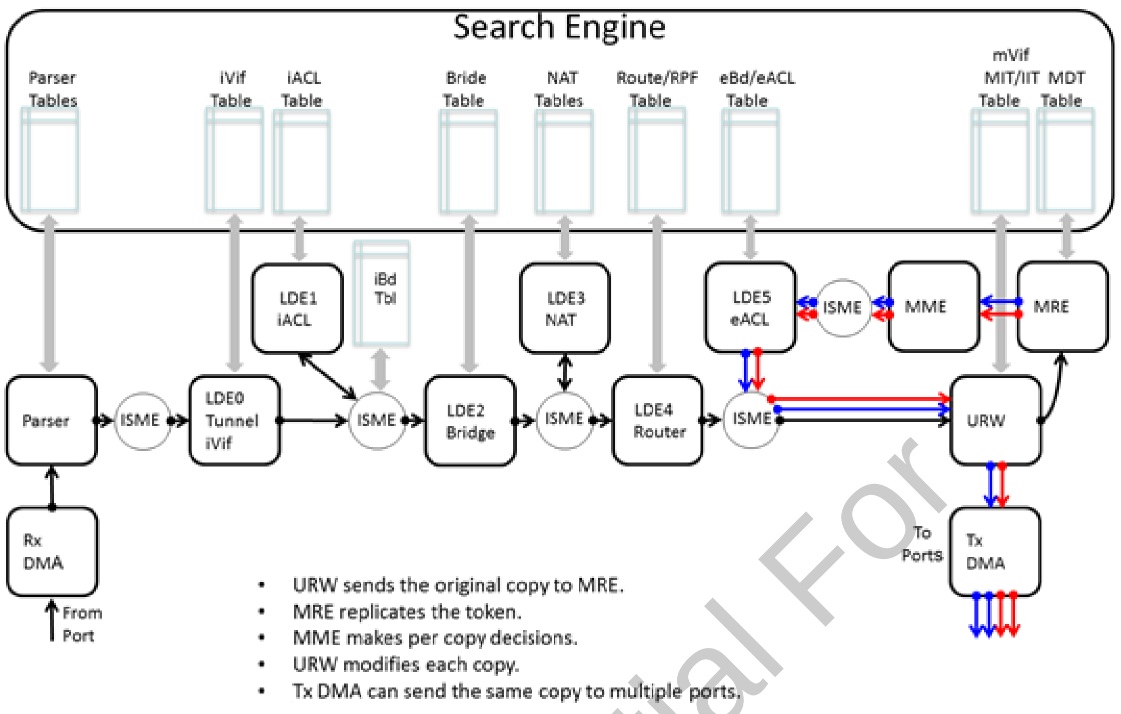


Figure 3–39 Multicast Flow for Bridging and Routing

3.12.2 Multicast Forwarding

3.12.2.1 IPv4 Multicast

The following packet flow represents the path taken by a multicast packet through various blocks inside the XP80.



Legend:

- Black flow represents the original copy.
- Blue and Red flows represent replicated copies.

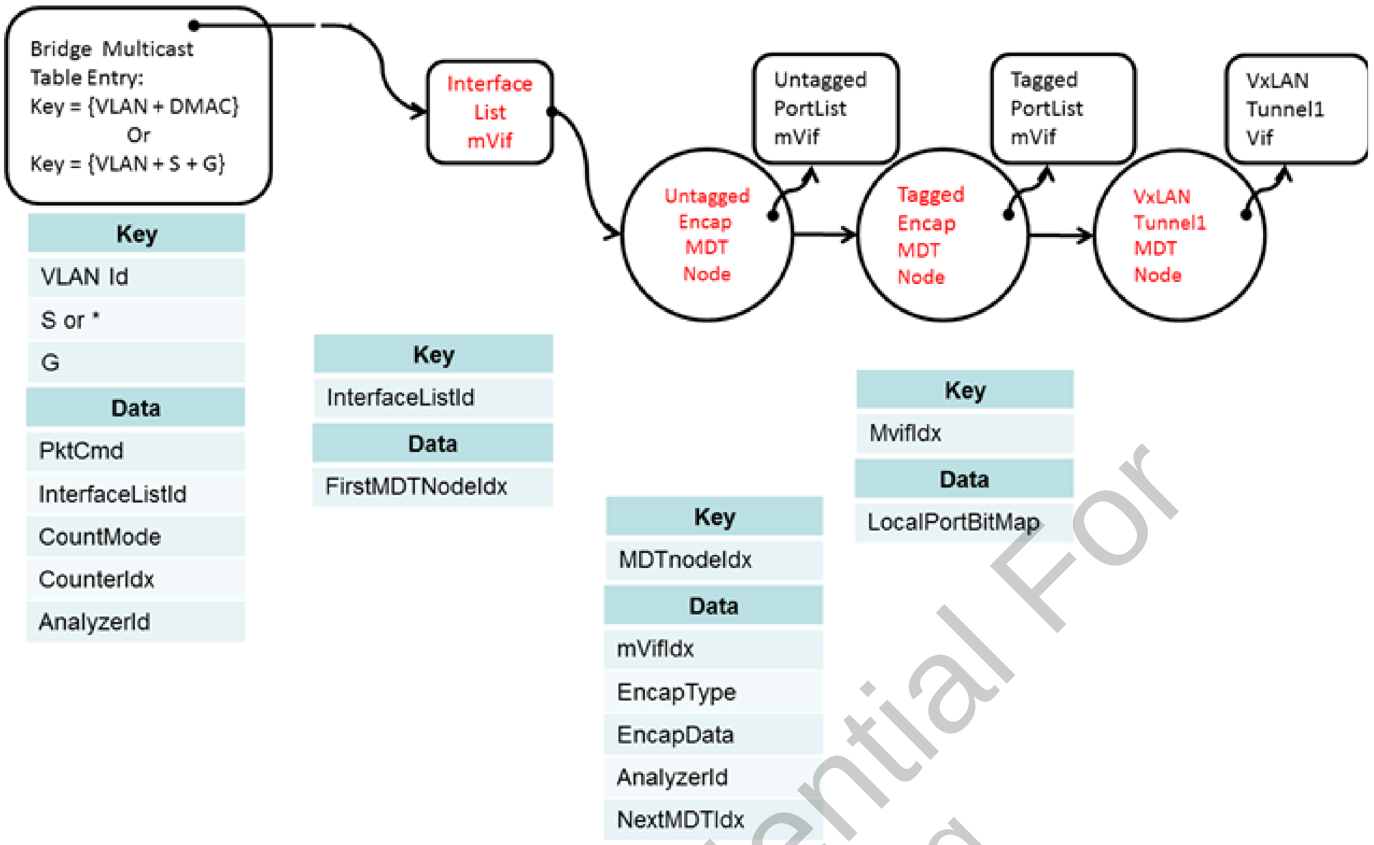
Figure 3–40 Block-level Schematic of Multicast Forwarding

Resource Allocation For a Multicast Flow

The following steps are required to set up a multicast flow.

1. Create a Multicast Tree (MDT) that represent the Outgoing Interface List (OIF) of the flow.
2. Create a virtual interface (mVif) that points to the root node of the MDT.
3. Create a Multicast Ingress entry (IPv4 or IPv6 Multicast Bridge or Multicast route entry) that points to mVif.

The following diagram shows the Multicast Bridge entry.



Key
VLAN Id
S or *
G
Data
PktCmd
InterfaceListId
CountMode
CounterIdx
AnalyzerId

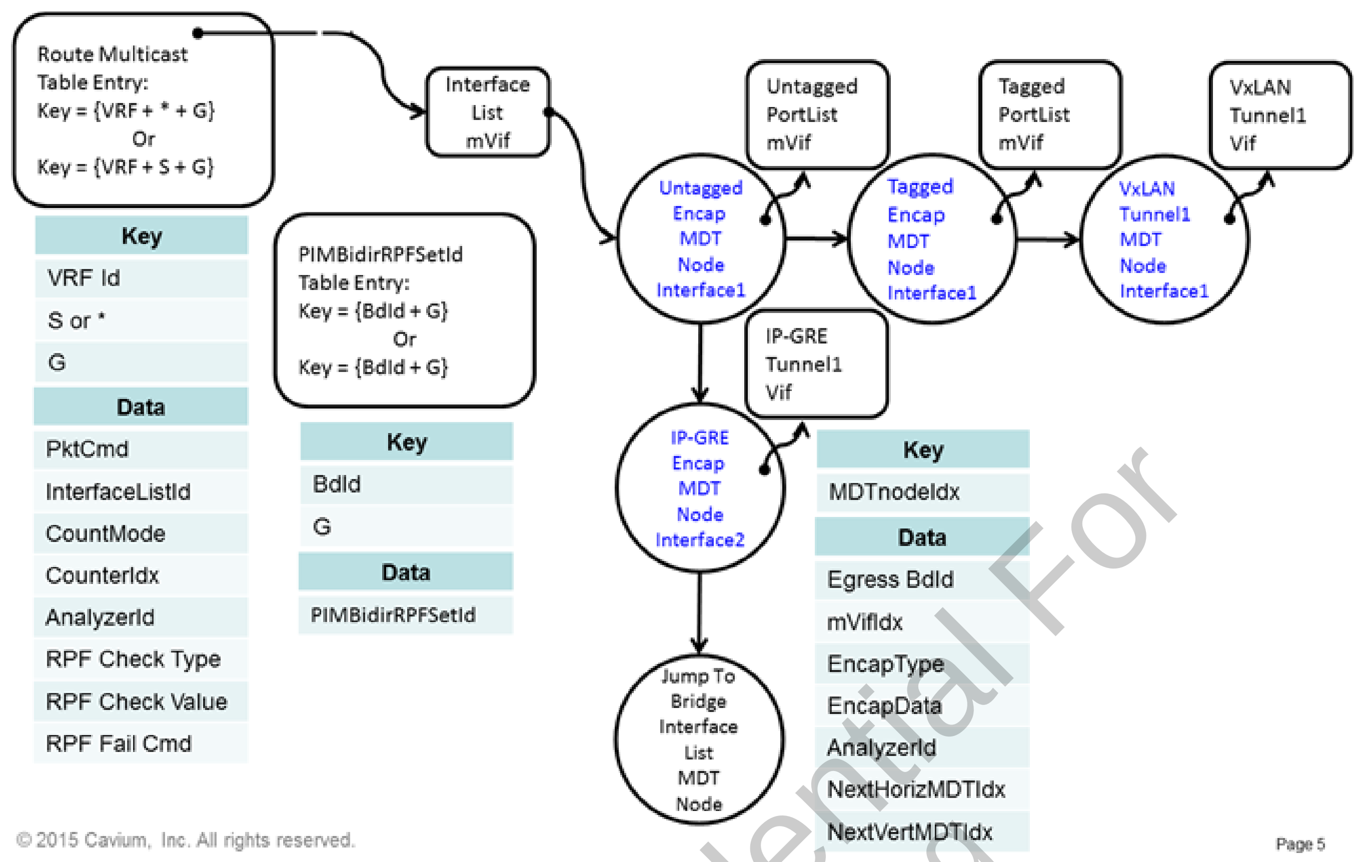
Key
InterfaceListId
Data
FirstMDTNodeIdx

Key
MDTnodeIdx
Data
mVifIdx
EncapType
EncapData
AnalyzerId
NextMDTIdx

Key
MvifIdx
Data
LocalPortBitMap

Figure 3–41 Resource Architecture for Multicast Bridge

The following diagram shows the Multicast Route entry.



Key
VRF Id
S or *
G
Data
PktCmd
InterfaceListId
CountMode
CounterIdx
AnalyzerId
RPF Check Type
RPF Check Value
RPF Fail Cmd

Key
BdId
G
Data
PIMBidirRPFSetId

Key
MDTnodeIdx
Data
Egress BdId
mVifIdx
EncapType
EncapData
AnalyzerId
NextHorizMDTIdx
NextVertMDTIdx

Figure 3–42 Multicast Route Resource Architecture

Implementation of Outgoing Interface Lists using MDT

The Outgoing Interface List of a Multicast Flow is built as a linked-list of MDT Nodes. The MRE makes a copy of the incoming packet for each MDT Node. As shown in figure 4 (Resource Architecture for Multicast Route)the horizontal list (L2Ptr) represents the different multicast VIFs for the same L3 interface and vertical list(L3Ptr) represent different L3 interfaces. The L3Ptr is valid only for the first node in each row. Each multicast VIF essentially represents a pair (outgoing encapsulation type, port-list). In addition, the multicast VIF can also point to MDT Node to provide indirections and hierarchies. This architecture allows for maximum flexibility in programming and managing of resources through their sharing across multiple flows.

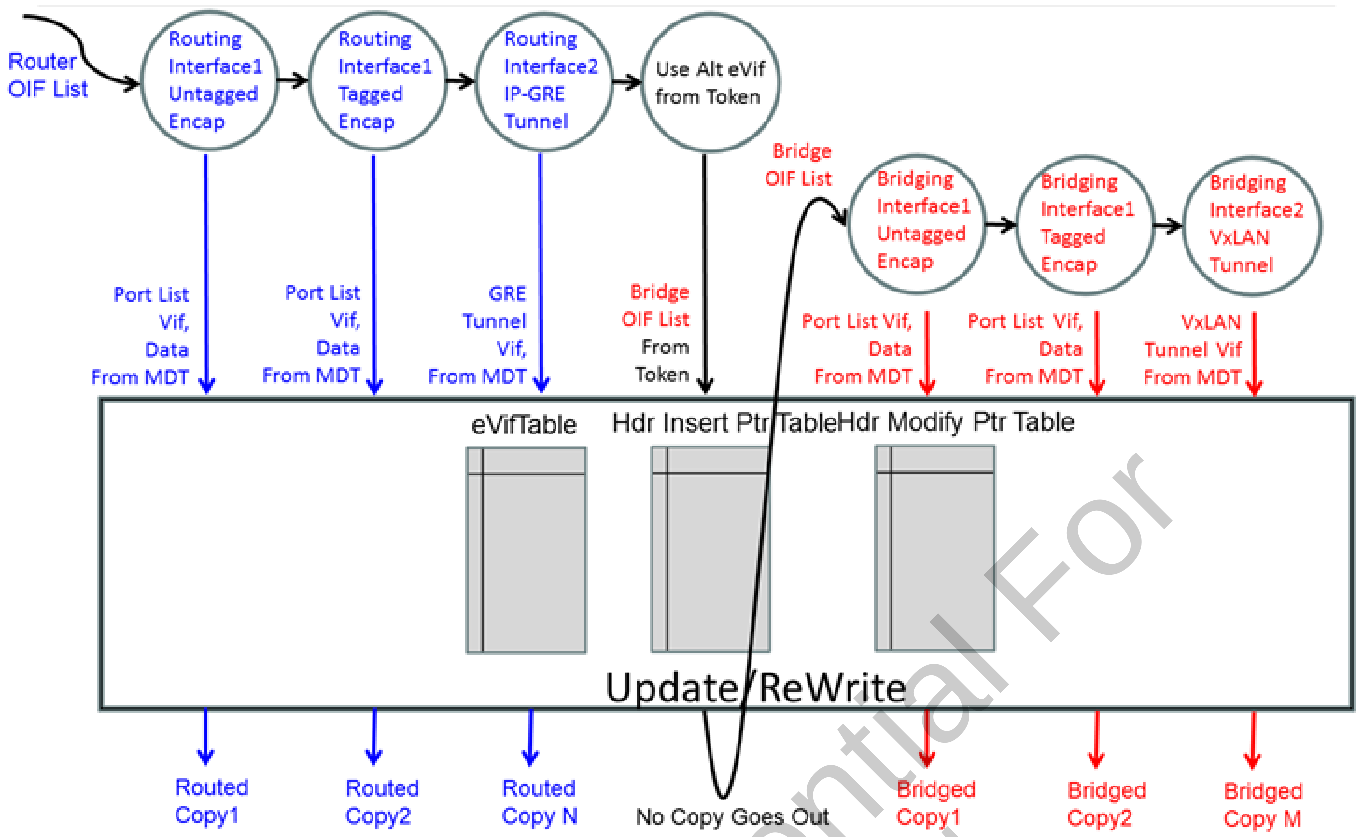

Router OIF List
Routing Interface1 Untagged Encap
Routing Interface1 Tagged Encap
Routing Interface2 IP-GRE Tunnel
Use Alt eVif from Token
Bridge OIF List
Bridging Interface1 Untagged Encap
Bridging Interface1 Tagged Encap
Bridging Interface2 VxLAN Tunnel
Port List Vif, Data From MDT
Port List Vif, Data From MDT
GRE Tunnel Vif, From MDT
Bridge OIF List From Token
Port List Vif, Data From MDT
Port List Vif, Data From MDT
VxLAN Tunnel Vif From MDT
eVifTable
Hdr Insert Ptr Table
Hdr Modify Ptr Table
Update/ReWrite
Routed Copy1
Routed Copy2
Routed Copy N
No Copy Goes Out
Bridged Copy1
Bridged Copy2
Bridged Copy M

Figure 3–43 MDT Configuration for Hierarchical Multicast

Figure 3–43 illustrates the interaction of Multicast Route and Multicast Bridging tables when a multicast packet is sent out from the created router OIF list and Bridge OIF list.

When a multicast packet must be sent out, the router OIF list is traversed from left to right, and the packet is sent out on an interface one by one. At the end of the router OIF list, the bridge OIF list is taken from AlteVif and traversed to send the multicast packet out.

3.13 Bridge Port Extension (802.1BR)

802.1BR is the IEEE standard that specifies the devices, protocols, procedures, and managed objects necessary to extend a bridge and its management beyond its physical enclosure using 802 LAN technologies.

The 802.1BR architecture provides the ability to extend the bridge (switch) interface to downstream devices and associates the Logical Interface (LIF) to a Virtual Interface (VIF), which addresses the increasing demands of server virtualization for abstraction and flexibility to manage and configure virtual ports.

The IEEE is defining a tag, referred to as E-Tag, under the IEEE 802.1BR working group (Bridge Port Extension). E-Tag definition effort was under the IEEE 802.1Qbh working group, and was moved to the IEEE 802.1BR.

The following diagram depicts a port extension deployment scenario where an external and internal port extension (PE) connected to a controlling bridge and extended bridge is formed by a controlling bridge plus attached port extensions. Each port of a port extension is a virtual port of the controlling bridge. All traffic is relayed by the controlling bridge to the external network (including to network management).

Refer to 802.1BR specification (http://www.ieee802.org/1/pages/802.1br.html) for additional details.

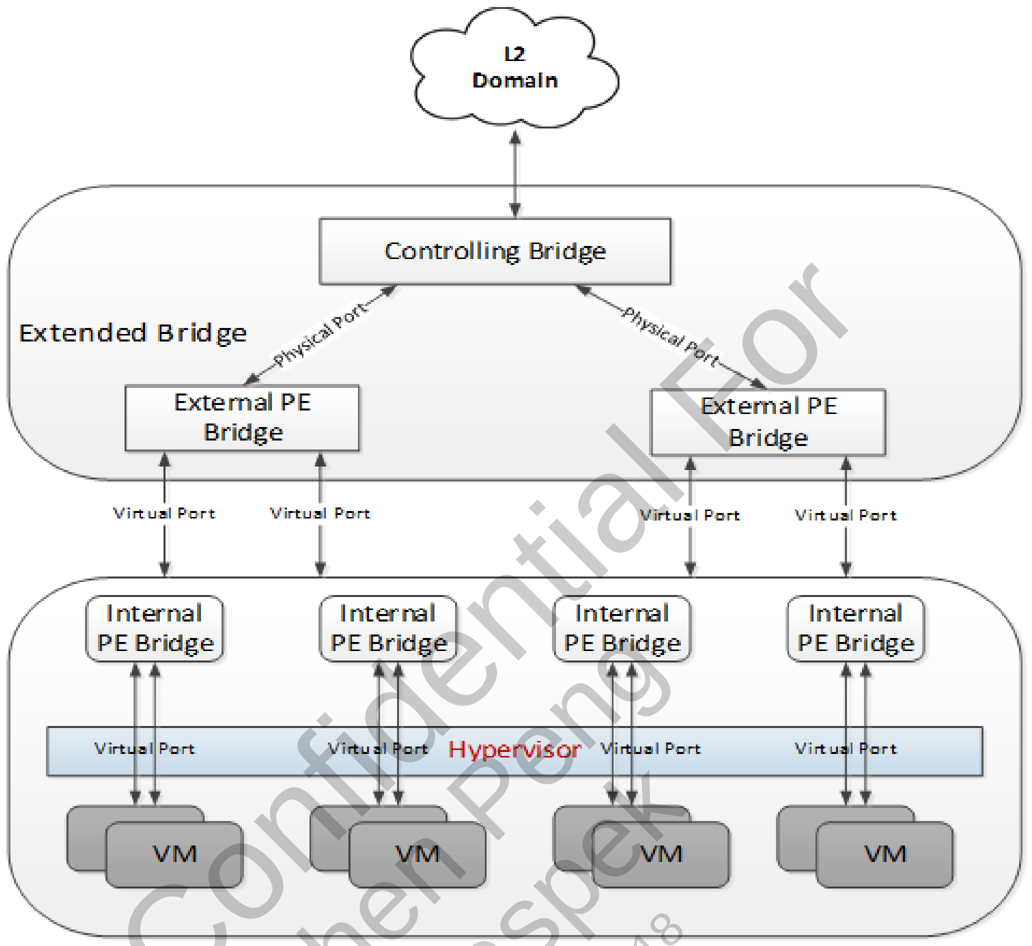


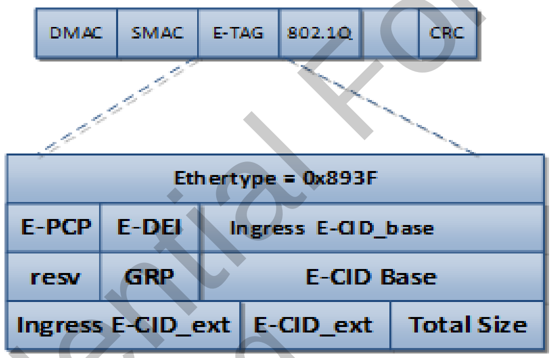
Figure 3–44 Port Extender Deployment

The port extension includes the following elements:

Table 3–16 Port Extender Elements

Bridge Port Extender	A device used to extend the MAC service of a C-VLAN component to form a controlling bridge and to extend the MAC service of a controlling bridge to form an extended bridge.
Base Port Extender	A bridge port extension that supports a subset of the E-channel Identifier (E-CID) space.
Aggregating Port Extender	A bridge port extension that supports the full E-channel Identifier (E-CID) space and is capable of aggregating base port extensions.
External Bridge Port Extender	A Bridge Port Extender that is not physically part of a Controlling Bridge but is controlled by the Controlling Bridge.
Controlling Bridge	A Bridge that supports one or more Bridge Port Extenders, i.e. xp80
E-channel	An instance of the MAC service supported by a set of two E-paths forming a bidirectional service. An E-channel is point-to-point or point-to-multipoint.

Table 3–16 Port Extender Elements

Upstream Port	A port on a bridge port extension that connects to a cascade port. In the case of the connection between two Bridge Port Extenders, the Upstream Port is the Port furthest from the Controlling Bridge.
E-channel Identifier (E-CID)	A value conveyed in a E-Tag that identifies an E-channel.
E-path	A configured unidirectional connectivity path between an internal extended port and one or more external extended ports and/or upstream ports. E-paths originating from the internal bridge port extension can be point-to-point or point-to-multipoint. E-paths originating from an external bridge port extension can be point-to-point or multipoint-to-point.
E-Tag	A tag header with a tag protocol identification value allocated for IEEE 802.1BR E-Tag type.  DMAC SMAC E-TAG 802.1Q CRC Ethertype = 0x893F E-PCP E-DEI Ingress E-CID_base resv GRP E-CID Base Ingress E-CID_ext E-CID_ext Total Size
Extended Bridge	A controlling bridge and at least one bridge port extension under the controlling bridge control.
Extended Port	A port of a bridge port extension that is not operating as a cascade port or upstream port. This includes the ports of a bridge port extension connected through internal LANs to the port of a C-VLAN component within a controlling bridge.
External Aggregating Port Extender	An aggregating bridge port extension that is not physically part of a controlling bridge but is controlled by the controlling bridge.

NOTE: When *port extension* is used in this document, it means external aggregating port extension.

3.13.1 Traffic Path

Multicast and unicast traffic from the VM gets forwarded to the CB, which gets identified by SMAC and VLAN and forwarded with DMAC and VLAN. The controlling bridge identifies a VM based on E-Tag CID. The following diagram depicts data and control messages.

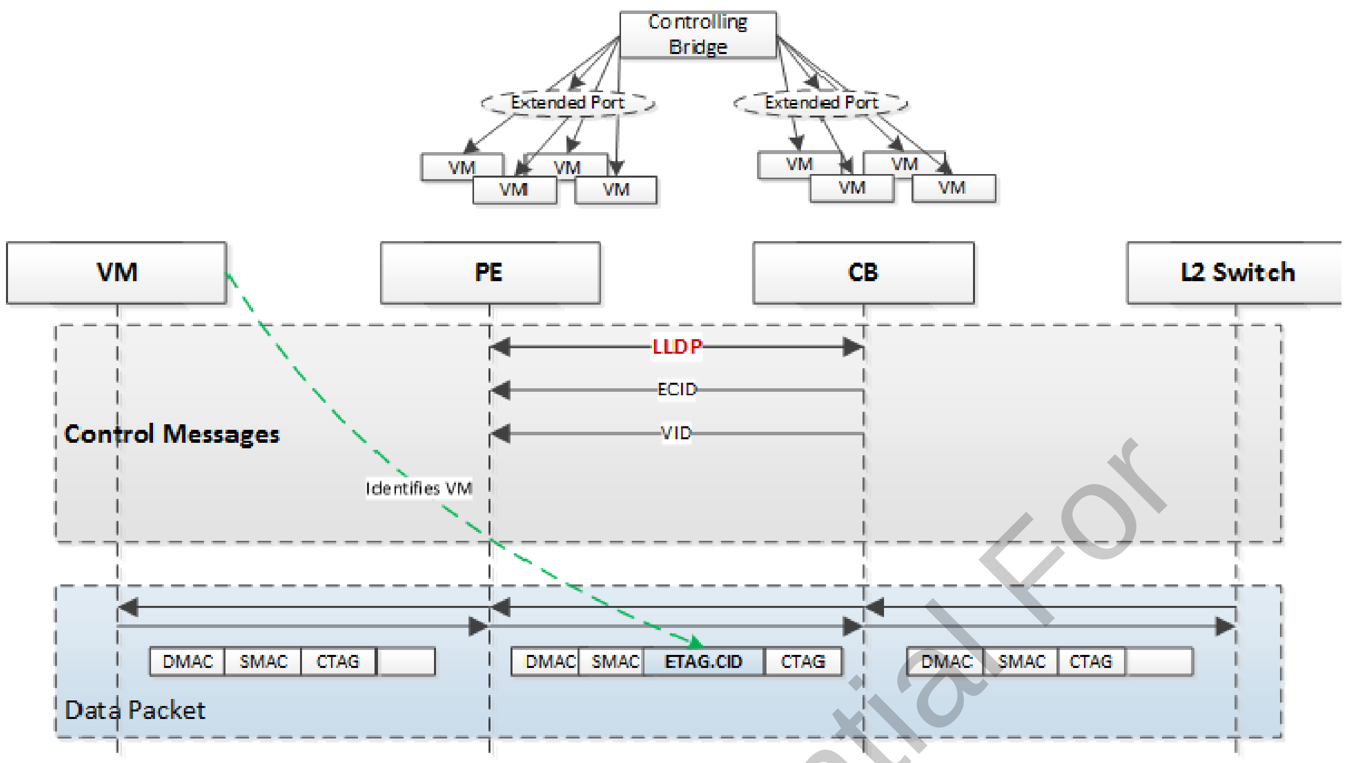


Figure 3–45 Port Extender Connection Steps

Port extension connection steps for control bridge:

1. Physical connection.
2. LLDP with application.
3. Application asks SDK to create BR group for <portsNum>.
4. SDK finds and allocates <portsNum> free VIF_IDs above 1024 from the XP_PORT range.
5. Application asks SDK to create interface from the BR group and bind it to the cascade ports.
6. SDK allocates ID from group and initializes port.
7. SDK configures physical cascade ports to add / receive E-Tags.
8. Application stores information about group ids and mapping of extended ports to the physical cascade ports.
9. SDK stores information about allocated VIFs inside group.

Ingress packet on extended port handling:

1. Cascade port must be configured to expect E-Tags.
2. All packets with E-Tags on cascade ports are stripped of tags and treated as ingress from VIF that corresponds to source ECID in tag.
3. Packets are bridged regularly.

Egress packet on extended port handling:

1. Cascade ports must be configured to add E-Tag.
2. E-Tag is added.
3. Source ECID equals VIF from where the packet was received.
4. Target ECID equals VIF of extended port.
5. Cascade port is selected using load balancing algorithm (if more than one is present).
6. Packet is sent from selected cascade port.

Table 3–17 E-Tag Parsing

Field Name	Value	W(b)	Description	Setting of Token by Parser Receiving a pkt with E-Tag
TPID	0x893F	16	E-Tag TPID	N/A
E-PCP	0-7	3	Priority Code Point	Use E-PCP and E-DEI as the L2 QoS in QoS assignment Alg
E-DEI	0-1	1	Drop Eligibilty bit	
Ingress_E-CID_base	0-0xFFF	12	Ingress Channel ID	Token.ingressVif[19:0]<= {Ingress_E-CID_ext[7:0],Ingress_E-CID_base[11:0]}
Reserved	0x0	2	Reserved	N/A
GRP	0-3	2	Extension for E CID	N/A
E-CID_Base	0-0xFFF	12	Egress Channel ID	Token.egressVif[19:0]<= {E-CID_ext[7:0],E-CID_base[11:0]}
Ingress_E-CID_ext	0-0xFF	8	Ingress Channel ID	See Ingress_E-CID_Base
E-CID_ext	0-0xFF	8		See E-CID_Base

- E-Tagged packets modified based on global configuration.
- E-Tag is added to header based on per port configuration.

Table 3–18 E-Tag at URW

Field Name	Value	W(b)	Description	Setting of E-Tag by Rewrite
TPID	0x893F	16	E-Tag TPID	From a Global Configuration Register E-Tag-TPIDConfig[15:0] Default Value: 0x893F
E-PCP	0-7	3	Priority Code Point	<= Token.PCP
E-DEI	0-1	1	Drop Eligibility bit	<= Token.DEI
Ingress_E-CID_base	0-0xFFF	12	Ingress Channel ID	<= ingressVif[11:0]
Reserved	0x0	2	Reserved	<= 2'b00
GRP	0-3	2	Extension for E CID	<= 3'b000

E-CID_Base	0-0xFFF	12	Egress Channel ID	If token.SetEtagEgressVifToMREPtr == 0 {token.egressVif[11:0]}
				Else {token.scratchpad[11:0]}

Ingress_E-CID_ext	0-0xFF	8	Ingress Channel ID	<= ingressVif[19:12]

E-CID_ext	0-0xFF	8		If token.SetEtagEgressVifToMREPtr == 0 {token.egressVif[19:12]}
				Else {token.scratchpad[19:12]}

3.14 OpenFlow

The following diagram depicts the OpenFlow Manager support in the SDK:

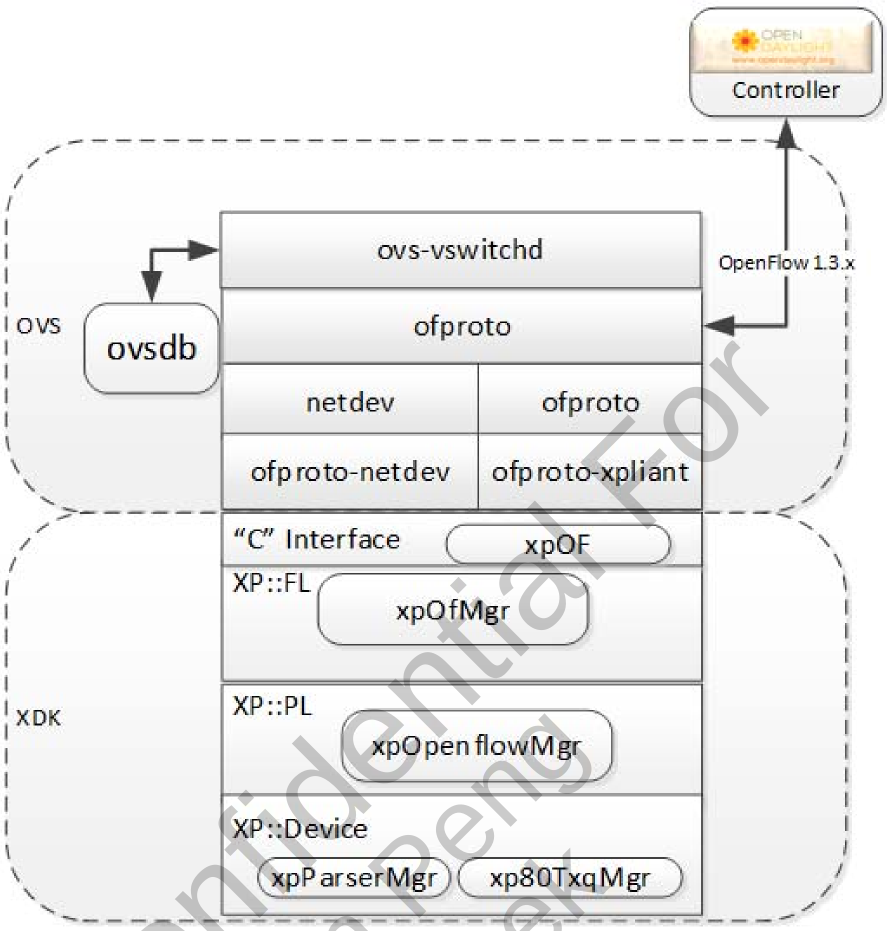


Figure 3–46 OpenFlow/XDK Implementation

3.14.1 OpenFlow Manager

The OpenFlow Manager supports OpenFlow-specific APIs. These modules expose required OpenFlow-specific table creation, configuration, and debugging information. There are OpenFlow-specific software modules within the XPS layer, Feature layer, and Primitive layers. Refer to Figure 3–46: *OpenFlow/XDK Implementation*.

The OpenFlow Manager API can be of the following types:

- Flow table—Allows table creation, table deletion, and add/delete/modify flow entries. These routines take as input the flow entry type. Flow entry type is present as a C data structure that contains all the necessary attributes of the specific flow (e.g., port number, port mask, etc.).
- OpenFlow Group-related API—Add support for Group Table (add/remove/ update).
- OpenFlow Statistics API—Add support for retrieving supported statistics for flows, tables, ports, groups, queues, and meters.
- Meter API.
- Asynchronous messages API.

3.14.2 Flow Table

Flow tables are the fundamental data structures for SDN. The device uses flow tables to evaluate incoming packets and take the appropriate action based on the contents of the packet.

Flow tables consist of a prioritized flow entry, which typically consists of match fields and actions.

- *Match fields* are used to compare against incoming packets. An incoming packet is compared against the match fields in priority order, and the first complete match is selected.
- *Actions* are the instructions that the network device perform if an incoming packet matches the match fields specified for that flow entry.

The flow table is implemented in both TCAM and SRAM.

- The TCAM table contains all flow entries that include a wildcard field. A table miss entry with a wildcard field is installed in the TCAM. The entries are also ordered by priority
- The SRAM tables contain entries where all the match fields have a specific value.

Match Fields	Priority	Counters	Instructions

  - Match Fields—To match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
  - Priority—Matching precedence of the flow entry.
  - Counters—Updated when packets are matched.
  - Instructions—To modify the action set or pipeline processing.

Table 3–19 Flow Table: Match

Match Field (520 bits)	Description
OXM_OF_IN_PORT	Required Ingress port. This may be a physical or switch-defined logical port.
OXM_OF_ETH_DST	Required Ethernet destination address. Can use arbitrary bitmask.
OXM_OF_ETH_SRC	Required Ethernet source address. Can use arbitrary bitmask.
OXM_OF_ETH_TYPE	Required Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_IP_PROTO	Required IPv4 or IPv6 protocol number.
OXM_OF_IPV4_SRC	Required IPv4 source address. Can use subnet mask or arbitrary bitmask.
OXM_OF_IPV4_DST	Required IPv4 destination address. Can use subnet mask or arbitrary bitmask.
OXM_OF_TCP_SRC	Required TCP source port.
OXM_OF_TCP_DST	Required TCP destination port.
OXM_OF_UDP_SRC	Required UDP source port.
OXM_OF_UDP_DST	Required UDP destination port.

The Flow table provides the following result:

1. Egress VIF (could be outgoing port vif or mvif if action is to forward to a group).
2. VLAN ID: If action is to push VLAN ID, then the value of VLAN ID will be obtained from this field.
3. Next Engine: If action is goto table ID, then this value contains the next engine where that table is located.

4. TTL: If action is to set TTL, then this field contains the TTL value.
5. Flags to indicate which action is set:
    - ActionOutputPort
    - ActionOutputGroup
    - ActionPopVlan
    - ActionPushVlan
    - ActionGotoTable
    - ActionSetTTL
    - ActionDecrTTL
    - ActionSetField

Table 3–20 Flow Table: Action

Action	Description
Pop	Apply all tag pop actions to the packet.
Set TTL	Set TTL value to the packet.
Decrement TTL	Apply decrement TTL action to the packet.
Set	Apply all set-field actions to the packet.
Group	If a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list.
Output	If no group action is specified, forward the packet on the port specified by the output action.

Table 3–21 Flow Table: Set Field

Set Field	Description
OPFXMT_OFB_ETH_DST	Ethernet destination address. Can use arbitrary bitmask
OPFXMT_OFB_ETH_SRC	Ethernet source address. Use lower 8 bits
OPFXMT_OFB_IPV4_SRC	IPv4 source address. Can use subnet mask or arbitrary bitmask
OPFXMT_OFB_IPV4_DST	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OPFXMT_OFB_TCP_SRC	TCP source port
OPFXMT_OFB_TCP_DST	TCP destination port
OPFXMT_OFB_UDP_SRC	UDP source port
OPFXMT_OFB_UDP_DST	UDP destination port
OPFXMT_OFB_VLAN_VID	Vlan

3.14.3 Group Table

A group table in XP can be implemented using MDT nodes. For every group an mVIF is allotted. The flow table result yields a mVIF. When the packet is being processed by the MME, for every MDT node it hits, it replicates the packet and executes the actions in the action bucket.

A group table consists of group entries. A flow entry may point to a group to represent additional methods of forwarding. Each group entry is identified by its group identifier and contains:

- Group identifier—A 32-bit unsigned integer uniquely identifying the group.
- Group type—Determines group semantics.
- Counters—Updated when packets are processed by a group.
- Action Buckets—An ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters.

Group Identifier	Group Type	Counters	Action Buckets

Table 3–22 Group Table: Action

Action	Description
Set TTL	Set TTL value to the packet.
Decrement TTL	Apply decrement TTL action to the packet.
Set	Apply all set-field actions to the packet.
Output	If no group action is specified, forward the packet on the port specified by the output action.

Table 3–23 Group Table: Set

Set Field	Description
OPFXMT_OFB_ETH_DST	Ethernet destination address. Can use arbitrary bitmask.
OPFXMT_OFB_IPV4_SRC	IPv4 source address. Can use subnet mask or arbitrary bitmask.
OPFXMT_OFB_IPV4_DST	IPv4 destination address. Can use subnet mask or arbitrary bitmask.
OPFXMT_OFB_TCP_SRC	TCP source port.
OPFXMT_OFB_TCP_DST	TCP destination port.
OPFXMT_OFB_UDP_SRC	UDP source port.
OPFXMT_OFB_UDP_DST	UDP destination port.
OPFXMT_OFB_VLAN_VID	VLAN.
OPFXMT_OFB_VLAN_PCP	PCP.

The following table summarizes specific behavior for Action List, Action Set, and Action Bucket in the Group Table:

Table 3–24 Group Table Behaviors

#	Action List (either in Apply-Actions instruction or Packet-out message)	Action Set (either a set associated with a packet or a set within Action Bucket)
1	Actions are executed in the order specified by the list.	Actions are executed in the predefined order specified below.
2	Multiple actions of the same type are allowed.	Maximum of one action of each type is allowed.
3	On output action, a copy of the packet is forwarded in its current state to the desired port.	On output action, the packet is forwarded on the port.
4	On group action, a copy of the packet is processed in its current state by the group buckets.	On group action, the packet is processed by the group buckets.
5	After the execution of the action list, pipeline execution continues on the modified packet according to Goto-Table instruction.	If no output action and no group action were specified, the packet is dropped.

3.14.4 Data Path

Given flexibility in search and packet processing, the device supports traditional, OpenFlow only, and hybrid pipeline.

The following diagram depicts a data path for OpenFlow. Based on the port VLAN, a packet can be treated by an OpenFlow pipeline or a traditional pipeline.

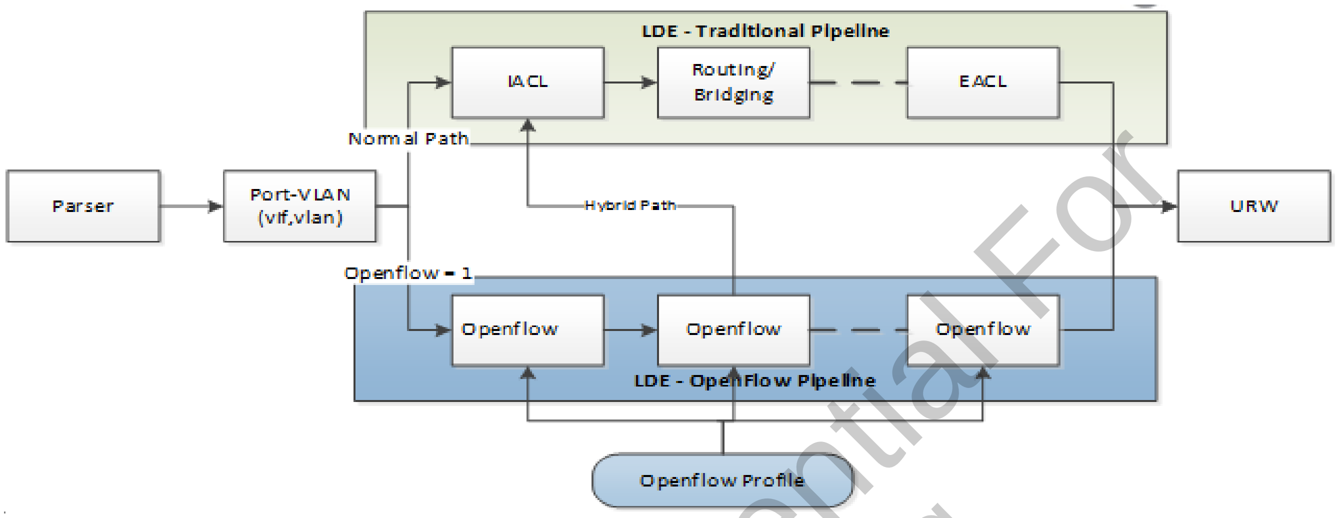


Figure 3–47 OpenFlow Data Path

The data path through an OpenFlow pipeline has the following characteristics:

- The flow tables of an OpenFlow switch are sequentially numbered.
- Pipeline processing always starts at the first flow table.
- These instructions may explicitly direct the packet to another flow table (Goto).
- A flow entry can only direct a packet to a flow table number that is greater than its own flow table number; in other words, pipeline processing can only go forward and not backward.
- The flow entries of the last table of the pipeline cannot include the Goto instruction.
- If the matching flow entry does not direct packets to another flow table, pipeline processing stops at this table.
- When pipeline processing stops, the packet is processed with its associated action set and usually forwarded.
- If a packet does not match a flow entry in a flow table, this is a table miss.
- A table-miss flow entry (the flow entry that wildcards all fields—all fields omitted—and has priority equal to 0) in the flow table can specify how to process unmatched packets. Options include:
  - ❍ Dropping packets;
  - ❍ Passing packets to another table;
  - ❍ Ending packets to the controller over the control channel through packet-in messages.
- Matches can be performed against: packet headers, ingress port, and metadata fields. The following generic matching rules are applicable:

- The packet is matched against the table and only the highest priority flow entry that matches the packet must be selected.
- The counters associated with the selected flow entry must be updated.
- The instruction set included in the selected flow entry must be applied.
- If there are multiple matching flow entries with the same highest priority, the selected flow entry is explicitly undefined (use OFPFF_CHECK_OVERLAP bit to avoid this).

As per the OpenFlow specification, a flow modification request may include an ID of a buffered packet. In such case, it removes the corresponding packet from the buffer and processes it through the entire OpenFlow pipeline after the flow is inserted, starting at the first flow table. This is effectively equivalent to sending a two-message sequence of a flow modification and a packet-out forwarding to the OFPP_TABLE logical port, with the requirement that the switch must fully process the flow modification before the packet out.

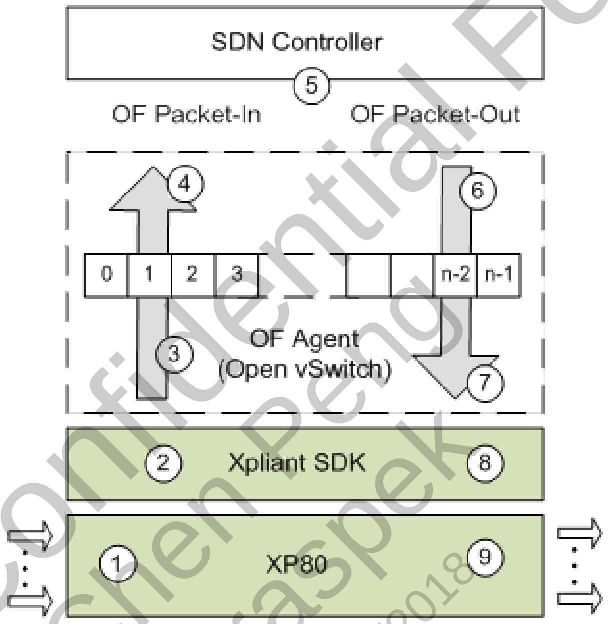


Figure 3–48 OpenFlow Manager Packet Handling

1. XP80 incoming packet triggers packet-in event on OF table miss.
2. OF Manager handles packet-in event and executes user-defined callback. See `registerPacketInHandler()` API.
3. User-defined callback places whole incoming packet into the packet buffer.
4. OF Agent generates OF Packet-In message that includes preconfigured amount of bytes from incoming packet and appropriate buffer_id, which holds entire packet.
5. SDN Controller makes forwarding decision on incoming packet and generates OF Packet-Out message with appropriate buffer_id and Action List, which must be applied to the buffered packet.
6. OF Agent retrieves packet from the buffer based on buffer_id provided in OF Packet-Out message.

7. OF Agent passes whole packet and Action List to XDK. See `applyActs()` API.
8. XDK passes OF Packet-Out request to XP80.
9. XP80 applies actions from Action List to the packet.

Appendix A

Reason Codes

A.1 Default Profile Reason Codes

The reason codes are software-defined per the following table:

Table A–1 Profile Reason Codes

Reason	Code	Description
XP_IVIF_PORT_VLAN_MISS	10	
XP_BRIDGE_MAC_SA_NEW	11	
XP_BRIDGE_MAC_SA_MOVE	12	
XP_BRIDGE_MAC_SA_HIT	13	
XP_IVIF_RC_INVALID_HDR	101	
XP_IVIF_RC_PORT_TAG_POLICY	102	
XP_IVIF_RC_BPDU	103	
XP_IVIF_RC_PKT_TMPLT_UNKNOWN	104	
XP_IVIF_RC_MARTIAN_IPADDR	105	
XP_IVIF_RC_NO_BD	106	
XP_TNL_ID_MISS	107	
XP_TNL_RVTEP_NEW	108	
XP_BRIDGE_RC_FLOOD_CMD	202	`iVif` level trap/mirror/drop for BC packets.
XP_BRIDGE_RC_FRAME_TYPE_INVALID	203	Received frame is unacceptable as per `iVif` config.
XP_BRIDGE_RC_FDB_DA_CMD	204	Packet trap/mirror due to FDB `DaCmd`.
XP_BRIDGE_RC_FDB_STATIC_SA_VIOLATION	205	Packet drop due to static SA appearing on `iVif` different from FDB entry.
XP_BRIDGE_RC_IVIF_ARP_CMD	206	ARP broadcast trap/mirror due to `iVif` control.
XP_BRIDGE_RC_IVIF_IGMP_CMD	207	IGMP broadcast trap/mirror due to `iVif` control.
XP_BRIDGE_RC_IVIF_ICMP6_CMD	208	ICMP broadcast trap/mirror due to `iVif` control.
XP_BRIDGE_RC_IVIF_MAC_SAMODE	209	Packet trapped/forwarded to CPU due to `macSAMode` configuration.
XP_BRIDGE_RC_IVIF_SPAN_BLOCKED	210	Packet dropped due to `iVif` state BLOCKED.
XP_BRIDGE_RC_IVIF_SA_MISS	211	Packet dropped due to SA Miss command in IVIF.
XP_BRIDGE_RC_MC_BRIDGE_CMD	212	Packet trap/mirror due to FDB `DaCmd`.
XP_RC_PBB_REDIRECT	213	
XP_BRIDGE_RC_UNKNOWN_L2_PACKET	214	`iVif` level trap/mirror/drop for unknown UC packets.
XP_BRIDGE_BD_TABLE_MISS	215	
XP_BRIDGE_RC_MC_UNREG_CMD	216	Bridge domain `unregisteredMcCmd`.
XP_BRIDGE_RC_FDB_SA_CMD	217	

Table A–1 Profile Reason Codes

Reason	Code	Description
XP_BRIDGE_RC_SPLIT_HORIZON_RULE	218	
XP_ROUTE_RC_TTL0	301	
XP_ROUTE_RC_HOST_TABLE_HIT	302	
XP_ROUTE_RC_ROUTE_TABLE_HIT	303	
XP_ROUTE_RC_MC_BRDG_TABLE_HIT	304	
XP_ROUTE_RC_MC_ROUTE_TABLE_HIT	305	
XP_ROUTE_RC_MC_ROUTE_MISS	306	
XP_ROUTE_RC_UC_TABLE_MISS	307	
XP_ROUTE_RC_PKT_TYPE_UNKNOWN	308	
XP_ROUTE_RC_MC_INVALID_DA	309	
XP_ROUTE_RC_IP_OPTIONS	310	
XP_ROUTE_RC_MC_RPF_FAIL	311	
XP_ROUTE_RC_ROUTE_NOT_POSSIBLE	312	
XP_IACL_ACTION	401	
XP__TABLE_MISS	512	
XP__TRAP_TO_CPU	513	
XP_SFLOW_RC_SAMPLE	601	
XP_POLICING_PKT_DROP	602	
XP_EGRESS_BD_TABLE_MISS	701	
XP_EACL_PKT_CMD	702	

A.2 Reason Code Table (RCT) Contents

Table A–2 Reason Code Table (RCT) Contents

Field	Width (bits)	Description
Port#	8	CPU port number (no EVIF lookups are done for these).
Traffic Class	4	This overrides the traffic class in the token (as well as the Traffic Manager.
Truncation	1	This bit is sent to truncate the packet part to be sent to the CPU.
Format	1	If 1, egress modifications are done as per token rewrite instructions and modified insert instructions. If 0, packet is not to be modified and hence insert and modify instructions will be masked and only InsertPointer insertion will be done.

Appendix B

Generic Data Structures

B.1 Initial Token Table (ITT)

Table Name: Initial Token Table (ITT)
Table Type: Local Table, Direct Access

Table B–1 Initial Token – Table Entry

Field Name	Size (b)	Type	Description
`etagExists`	1	Control	Indicates if packets received on this port has eTag
`l2QosProfileIdx`	3	Control	Index to L2 QoS Profile table
`ipQosProfileIdx`	3	Control	Index to IP QoS Profile table
`mplsQosProfileIdx`	3	Control	Index to MPLS QoS Profile table
`pvid`	12	Control	Port Default VLAN ID
`pvidModeAllPkt`	1	Control	0: disabled (default) 1: force PVID as VLAN for ALL packets
`ingressVif`	20	Control	Port default ingress VIF
`mirrorBitMask`	16	Control	Port default mirror bitMask
`scratchPad`	64	Control	portIVIF configuration
`egressFilterId`	8	Control	Port default egress Port Filter Group ID
`layer2QosEn`	1	Control	0: Layer 2 QoS assignment disabled (default) 1: Layer 2 Qos assignment enabled QoS parameters are determined based on Layer2 header of ingress packet
`lPQosEn`	1	Control	0: IP header-based QoS assignment disabled(default) 1: IP header-based Qos assignment enabled QoS parameters are determined based on IPV4 or v6 DSCP of ingress packet
`mplsQosEn`	1	Control	0: MPLS header-based QoS assignment disabled (default) 1: MPLS header-based Qos assignment enabled QoS parameters are determined based on MPLS header EXP field of ingress packet
`portDefaultTC`	4	Control	Default traffic class for this port.
`portDefaultDP`	2	Control	Default traffic class for this port.
`portDefaultPCP`	3	Control	Default traffic class for this port.
`portDefaultDEI`	1	Control	Default traffic class for this port.
`portDefaultDSCP`	6	Control	Default traffic class for this port.
`portDefaultEXP`	3	Control	Default traffic class for this port.

Table B–2 Initial Token – Table Key

Field Name	Size (b)	Type	Description
portNum	8	Key	Ingress port number

B.2 Multicast Distribution Table

Table type: Direct Access

Table B–3 Bridge Format

Name	Size	Description
hdrModPtr1	4	Header Modification Pointer 1.
hdrModPtr2	4	Header Modification Pointer 2.
encapType	2	Encap type for L2 switching.
isVPLSHub	1	Only useful for VPLS Case. Needed for Split Horizon run for VPLS/H-VPLS where packet coming on MPLS spoke tunnel should not go back onto spoke tunnels.
isL2OverIPTunnel	1	Packet going over L2 IP tunnels VXLAN/ NvGRE.
reserved0	4	Reserved.
nextEngine	2	Next Engine code.
mirrorSessionId	4	Mirror Session ID.
sMac0to8	8	8 LSB bits of the source MAC.
cVlan	12	Customer VLAN ID.
egressVif	16	Egress VIF to be used for this copy.
entryFormat	2	Entry format = 00; // Bridge/L2 Format
l2HdrInsPtr	4	L2 Header Insert Pointer.
tunnelData	24	Data for tunneling.
reserved1	2	Reserved.
lastCopy	1	Indicates this is the last copy of replication.
nextL3Ptr	18	L3 pointer to be followed. 0 -> End of List.
nextL2Ptr	18	L2 pointer to be followed. 0 -> Last entry of the L2 branch.
useAltVif	1	If set, use altere EVIF for this copy (from scratchpad).

Table B–4 Route Format

Name	Size	Description
BdId	16	Egress BD ID to be used for multicast RPF failure check.
nextEngine	2	Next Engine code.

Table B–4 Route Format

Name	Size	Description
mirrorSessionId	4	Mirror Session ID.
tunnelType	2	
sMac0to8	8	8 LSB bits of the source MAC.
cVlan	12	The Customer VLAN ID.
egressVif	16	EVIF to be used for this copy.
entryFormat	2	Entry format = 10; // Route/L3 format.
l2HdrInsPtr	4	L2 header insert pointer.
tunnelData	24	Data for tunneling.
ignore	1	
reserved1	1	Reserved.
lastCopy	1	Indicates this is the last copy of replication.
nextL3Ptr	18	L3 Pointer to be followed. 0 -> End of List.
nextL2Ptr	18	L2 Pointer to be followed. 0 -> Last entry of the L2 branch.
useAltVif	1	If set, use altere EVIF for this copy (from scratchpad)

Table B–5 Mirror Format

Name	Size	Description
reserved0	54	Reserved.
mirrorSessionId	4	Mirror Session ID for Mirror Fn.
entryFormat	2	Entry format = 10; // Mirror Format.
pktCmd	2	Packet command to be copied to token. Refer to Appendix B, Generic Data Structures.
egressVif	20	EVIF to be used for this copy.
analyzerId	8	
lastCopy	1	Indicates this is the last copy of replication.
nextL3Ptr	18	L3 pointer to be followed. 0 -> End of List.
nextL2Ptr	18	L2 pointer to be followed. 0 -> Last entry of the L2 branch.
useAltVif	1	If set, use altere EVIF for this copy (from scratchpad).

B.3 Default Profile Tables Entry Formats

B.3.1 LDE Tables

B.3.1.1 Port VLAN Table

Type: Hash
Key Size: 32b
Data Size: 96b

Table B–6 Port VLAN Table

Field Name	Sizze (b)	Description
Table Entry		
`setIngressVif`	1	Enable setting the ingress VIF attributes.
`setBridgeDomain`	1	Enable setting the bridge domain attributes.
`spanState`	2	Bridge attributes: Spanning-tree relevant for MSTP, for this {port,VLAN} 00 - DISABLED (data packets are forwarded, BPDU is flooded) 01 - LEARNING (data packets are dropped, but mac learning is triggered) 10 - FORWARDING (data packets are forwarded, mac learning is triggered) 11 - BLOCKING (data packets are dropped, no mac learning, BPDU is trapped to CPU)
`bypassTunnelVif`	1	Bypass tunnel VIFlLookup.
`bypassAclsPBR`	1	Bypass ACLs Engine.
`bypassBridgeRouter`	1	Bypass BridgeRouter Engine.
`enableMirror`	1	0: Mirroring is not enabled 1: Mirroring is enabled and token.MirrorMask |= (1 << missorSessionId)
`BridgeACLEn`	1	Used in Ingress Policy Engine to enable Bridge ACLs TCAM lookup.
`RouterACLEn`	1	Used in Ingress Policy Engine to enable Router ACLs TCAM lookup.
`Mode`	2	Router attributes: 0 - Disabled 1 - IP Only: Only IP addresses, either SIP or DIP depending on Direction, is replaced 2 - IP and Port: {Both SIP and UDP/TCP SrcPort} OR {Both DIP and UDP/TCP DstPort} are replaced
`networkScope`	1	Router attributes: Used for 0 - Internal network 1 - External network
`countMode`	3	Port mask: b0: counting enabled b1: policing enabled b2: sampling enabled
`QinQLookupEn`	1	Enable lookup for QinQ.
`PBBLookupEn`	1	Enable lookup for PBB tunnels.
`ipv4TunnelLookupEn`	1	Enable lookup for IP Tunnels
`mplsTunnelLookupEn`	1	Enable lookup for MPLS tunnels.
`isV11`	1	0 - VPLS Case (P2MP) 1 - Vll Case (P2P)
`isHub`	1	0 - Attachment Circuit is Spoke 1 - Attachment Circuit is Hub
`reserved / Metadata`	2	Reserved for future use

Table B–6 Port VLAN Table

Field Name	Sizze (b)	Description
bridgeAclId	8	Bridge ACL ID. Valid only if iacl1En == true.
routeAclId	8	Route ACL ID. Valid only if iacl2En == true.
bridgeDomain	16	Packet bridge domain attribute if <setBd>=1, set token.bridgeDomain <= <bd>
ingressVif	16	Packet ingress virtual interface if <setIngressVif>=1, set token.ingressVif <= <ingressVif>
mirrorSessionId	3	Mirroring session ID. Valid only if enableMirror == 1
reserved / Metadata	5	Reserved for future use
counterIndex	16	Counter ID as configured in the counter block (table hit address by default/not specified here).
Table Key		
ingressVif	16	Based on ingressVIF from port setting.
outerVlan	16	BD is mapped based on Port VLAN assignment algorithm (this is actually the 4b0'+12bVID).

B.3.1.2 X over IPv4 Tunnel Termination Table

Table Type: Hash
Key Size: 128b
Data Size: 128b

Table B–7 X over IPv4 Tunnel Termination Table

Field Name	Size (b)	Description
Table Entry		
setIngressVif	1	Enable setting the ingress VIF attributes.
setBridgeDomain	1	Enable setting the bridge domain attributes.
spanState	2	Bridge attributes: Spanning-tree relevant for MSTP, for this {port,VLAN} 00 - DISABLED (data packets are forwarded, BPDU is flooded) 01 - LEARNING (data packets are dropped, but mac learning is triggered) 10 - FORWARDING (data packets are forwarded, mac learning is triggered) 11 - BLOCKING (data packets are dropped, no mac learning, BPDU is trapped to CPU)
setPktCmd	1	Enable packet command.
pktCmd	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands.
enableMirror	1	0: Mirroring is not enabled 1: Mirroring is enabled and token.MirrorMask |= (1 << missorSessionId)
BridgeACL	1	Used in Ingress Policy Engine to enable Bridge ACLs TCAM lookup.
PACL	1	Used in Ingress Policy Engine to enable Port ACLs TCAM lookup.
RouterACLEn	1	Used in Ingress Policy Engine to enable Router ACLs TCAM lookup.
Mode	2	Router attributes: 0 - is disabled 1 - IP Only: Only IP addresses, either SIP or DIP depending on Direction, is replaced 2 - IP and Port: {Both SIP and UDP/TCP SrcPort} OR {Both DIP and UDP/TCP DstPort} are replaced

Table B–7 X over IPv4 Tunnel Termination Table

Field Name	Size (b)	Description
`networkScope`	1	Router attributes: Used for 0 - Internal network 1 - External network
`countMode`	3	Port mask: b0: counting enabled b1: policing enabled b2: sampling enabled
`firstValidLayerTT`	4	When <firstValidLayerTT> != 0, the packet is tunnel terminated, token.firstValidLayer <= token.firstValidLayer + <firstValidLayerTT> **NOTE:** Packet is tunnel terminated only when the end packet command is forward.
`copyTTLfromTunnelHeader`	1	Copy Tunnel Header TTL into the inner packet. Relevant when inner packet is routed.
`mirrorToAnalyzerMask`	3	Mirroring session ID. Valid only if enableMirror == 1
`bridgeDomaind`	16	Packet bridge domain attribute. if <setBd>=1, set token.bridgeDomain <= <bd>
`iVif`	16	Packet ingress virtual interface. if <setIngressVif>=1, set token.ingressVif <= <ingressVif>
`reserved / Metadata`	8	Reserved for future use.
`counterIndex`	16	Counter ID as configured in the counter block (table hit sddress by default/not specified here).
`isV11`	1	0 - VPLS Case (P2MP) 1 - Vll Case (P2P)
`isHub`	1	0 - Attachment Circuit is Spoke 1 - Attachment Circuit is Hub
`IsVpn`	1	1 - VPN Lookup Result 0 - Tunnel Lookup Result
`isP2MPBudNode`	1	1 - P2MP Bud Node 0 - Non Bud Node
`reserved`	4	Reserved for future use.
`bridgeAclId`	8	Bridge ACL ID. Valid only if BridgeACLEn == true.
`pAclId`	8	Port ACL ID. Valid only if PACLEn == true.
`routeAclId`	8	Route ACL ID. Valid only if RouterACLEn == true.
`egressVif`	16	VIF to forward when setPktCmd == 1 AND pktCmd == FORWARD or FORWARD_AND_MIRROR
`ecmpSize`	8	L2/3 ECMP when setPktCmd == 1 AND pktCmd == FORWARD or FORWARD_AND_MIRROR
Table Key		
Entry Format	4	X - Over IPv4.
RouterMAC	1	Indicates that the packet is forwarded to this router and may be tunnel terminated.
Reserved / Metadata	3	Reserved for future use.
VxLAN - VNI / NVGRE-TNI	24	Tunneled packet identifier.
SIP	32	
DIP	32	

Table B–7 X over IPv4 Tunnel Termination Table

Field Name	Size (b)	Description
ingressVif	16	ingressVIF based on port.
Outer VLAN	12	
Tunnel Type	4	3 – IPv4

B.3.1.3 PBB Tunnel Termination Table

Type: Hash Key
Size: 128b
Data Size: 128b

Table B–8 PBB Tunnel Termination Table

Field Name	Size (b)	Description
Table Entry		
`setIngressVif`	1	Enable setting the ingress VIF attributes.
`setBridgeDomain`	1	Enable setting the bridge domain attributes.
`spanState`	2	Bridge attributes: Spanning-tree relevant for MSTP, for this {port,VLAN} 00 - DISABLED (data packets are forwarded, BPDU is flooded) 01 - LEARNING (data packets are dropped, but mac learning is triggered) 10 - FORWARDING (data packets are forwarded, mac learning is triggered) 11 - BLOCKING (data packets are dropped, no mac learning, BPDU is trapped to CPU)
`setPktCmd`	1	Enable packet command.
`pktCmd`	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands.
`enableMirror`	1	0: Mirroring is not enabled 1: Mirroring is enabled and token.MirrorMask |= (1 << missorSessionId)
`BridgeACLEn`	1	Used in Ingress Policy Engine to enable Bridge ACLs TCAM lookup.
`PACLEn`	1	Used in Ingress Policy Engine to enable Port ACLs TCAM lookup.
`RouterACLEn`	1	Used in Ingress Policy Engine to enable Router ACLs TCAM lookup.
`Mode`	2	Router attributes: 0 - is disabled 1 - IP Only: Only IP addresses, either SIP or DIP depending on Direction, is replaced 2 - IP and Port: {Both SIP and UDP/TCP SrcPort} OR {Both DIP and UDP/TCP DstPort} are replaced
`networkScope`	1	Router attributes: Used for 0 - Internal network 1 - External network
`countMode`	3	Port mask: b0: counting enabled b1: policing enabled b2: sampling enabled
`firstValidLayerTT`	4	When <firstValidLayerTT> != 0, the packet is tunnel terminated, token.firstValidLayer <= token.firstValidLayer + <firstValidLayerTT> **NOTE:** Packet is tunnel terminated only when the end packet command is forward.
`copyTTLfromTunnelHeader`	1	Copy Tunnel Header TTL into the inner packet. Relevant when inner packet is routed.

Table B–8 PBB Tunnel Termination Table

Field Name	Size (b)	Description
mirrorToAnalyzerMask	3	Mirroring session ID. Valid only if enableMirror == 1.
bridgeDomain	16	Packet bridge domain attribute. if <setBd>=1, set token.bridgeDomain <= <bd>
ingressVif	16	Packet ingress virtual interface. if <setIngressVif>=1, set token.ingressVif <= <ingressVif>
reserved / Metadata	8	Reserved for future use.
counterIndex	16	Counter ID as configured in the counter block (table hit address by default/not specified here).
isV11	1	0 - VPLS Case (P2MP) 1 - Vll Case (P2P)
isHub	1	0 - Attachment Circuit is Spoke 1 - Attachment Circuit is Hub
IsVpn	1	1 - VPN Lookup Result 0 - Tunnel Lookup Result
isP2MPBudNode	1	1 - P2MP Bud Node 0 - Non Bud Node
reserved	4	Reserved for future use.
bridgeAclId	8	Bridge ACL ID. Valid only if BridgeACLEn == true.
pAclId	8	Port ACL ID. Valid only if PACLEn == true.
routeAclId	8	Route ACL ID. Valid only if RouterACLEn ==true.
egressVif	16	VIF to forward when setPktCmd == 1 AND pktCmd == FORWARD or FWD_MIRROR
ecmpSize	8	L2/3 ECMP when setPktCmd == 1 AND pktCmd == FORWARD or FWD_MIRROR
Table Key		
Entry Format	4	PBB (802.1ah).
Reserved	4	Reserved for future use.
I-SID	24	
Reserved	64	
ingressVif	16	ingressVIF based on port.
B-Tag	12	
Reserved	4	

B.3.1.4 Remote VTEP Table

Type: Hash
Key Size: 128b
Data Size: 128b

Table B–9 Remote VTEP Table

Field Name	Size (b)	Description
Table Entry		
setIngressVif	1	Enable setting the ingress VIF attributes.
setBridgeDomain	1	Enable setting the bridge domain attributes.

Table B–9 Remote VTEP Table

Field Name	Size (b)	Description
`spanState`	2	Bridge attributes: Spanning-tree relevant for MSTP, for this {port,VLAN} 3 - DISABLED (data packets are forwarded, BPDU is flooded) 4 - LEARNING (data packets are dropped, but mac learning is triggered) 10 - FORWARDING (data packets are forwarded, mac learning is triggered) 11 - BLOCKING (data packets are dropped, no mac learning, BPDU is trapped to CPU)
`setPktCmd`	1	Enable packet command.
`pktCmd`	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands.
`enableMirror`	1	0: Mirroring is not enabled 1: Mirroring is enabled and token.MirrorMask |= (1 << missorSessionId)
`BridgeACLEn`	1	Used in Ingress Policy Engine to enable Bridge ACLs TCAM lookup.
`PACLEn`	1	Used in Ingress Policy Engine to enable Port ACLs TCAM lookup.
`RouterACLEn`	1	Used in Ingress Policy Engine to enable Router ACLs TCAM lookup.
`Mode`	2	Router attributes: 0 - Disabled 1 - IP Only: Only IP addresses, either SIP or DIP depending on Direction, is replaced 2 - IP and Port: {Both SIP and UDP/TCP SrcPort} OR {Both DIP and UDP/TCP DstPort} are replaced
`networkScope`	1	Router attributes: Used for 0 - Internal network 1 - External network
`countMode`	3	Port mask: b0: counting enabled b1: policing enabled b2: sampling enabled
`firstValidLayerTT`	4	When <firstValidLayerTT> != 0, the packet is tunnel terminated, token.firstValidLayer <= token.firstValidLayer + <firstValidLayerTT> **NOTE:** Packet is tunnel terminated only when the end packet command is forward.
`copyTTLfromTunnelHeader`	1	Copy Tunnel Header TTL into the inner packet. Relevant when inner packet is routed.
`mirrorToAnalyzerMask`	3	Mirroring session ID. Valid only if enableMirror == 1.
`bridgeDomain`	16	Packet bridge domain attribute. if <setBd>=1, set token.bridgeDomain <= <bd>
`ingressVif`	16	Packet ingress virtual interface. if <setIngressVif>=1, set token.ingressVif <= <ingressVif>
`reserved / Metadata`	8	Reserved for future use.
`counterIndex`	16	Counter ID as configured in the counter block (table hit Address by default/not specified here).
`isV11`	1	0 - VPLS Case (P2MP) 1 - Vll Case (P2P)
`isHub`	1	0 - Attachment Circuit is Spoke 1 - Attachment Circuit is Hub
`IsVpn`	1	1 - VPN Lookup Result 0 - Tunnel Lookup Result

Table B–9 Remote VTEP Table

Field Name	Size (b)	Description
`isP2MPBudNode`	1	1 - P2MP Bud Node 0 - Non Bud Node
`reserved / Metadata`	4	Reserved for future use.
`bridgeAclId`	8	Bridge ACL ID. Valid only if BridgeACLEn == true.
`pAclId`	8	Port ACL ID. Valid only if PACLEn ==true.
`routeAclId`	8	Route ACL ID. Valid only if RouterACLEn == true.
`egressVif`	16	VIF to forward when setPktCmd == 1 AND pktCmd == FORWARD or FWD_MIRROR.
`ecmpSize`	8	L2/3 ECMP when setPktCmd == 1 AND pktCmd == FORWARD or FWD_MIRROR.
Table Key		
Entry Format	4	Virtual tunnel.
reserved	27	
SIP	32	Remote VTEP address.
reserved	60	

B.3.1.5 X Over MPLS Tunnel Termination Table (LER and L2/L3 VPN Only)

Type: Hash
Key Size: 128b
Data Size: 128b

Table B–10 X Over MPLS Tunnel Termination Table (LER and L2/L3 VPN Only)

Field Name	Size (b)	Description
Table Entry		
`setIngressVif`	1	Enable setting the ingress VIF attributes.
`setBridgeDomain`	1	Enable setting the bridge domain attributes.
`spanState`	2	Bridge attributes: Spanning-tree Relevant for MSTP, for this {port,VLAN} 1 - DISABLED (data packets are forwarded, BPDU is flooded) 2 - LEARNING (data packets are dropped, but mac learning is triggered) 10 - FORWARDING (data packets are forwarded, mac learning is triggered) 11 - BLOCKING (data packets are dropped, no mac learning, BPDU is trapped to CPU)
`setPktCmd`	1	Enable packet command.
`pktCmd`	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands.
`enableMirror`	1	0: Mirroring is not enabled 1: Mirroring is enabled and token.MirrorMask |= (1 << missorSessionId)
`BridgeACLEn`	1	Used in Ingress Policy Engine to enable Bridge ACLs TCAM lookup
`PACLEn`	1	Used in Ingress Policy Engine to enable Port ACLs TCAM lookup.
`RouterACLEn`	1	Used in Ingress Policy Engine to enable Router ACLs TCAM lookup

Table B–10 X Over MPLS Tunnel Termination Table (LER and L2/L3 VPN Only)

Field Name	Size (b)	Description
`Mode`	2	Router attributes: 0 - Disabled. 1 - IP Only: Only IP addresses, either SIP or DIP depending on direction, are replaced. 2 - IP and Port: {Both SIP and UDP/TCP SrcPort} OR {Both DIP and UDP/TCP DstPort} are replaced.
`networkScope`	1	Router attributes: Used for 0 - Internal network 1 - External network
`countMode`	3	Port mask: b0: counting enabled b1: policing enabled b2: sampling enabled
`firstValidLayerTT`	4	When <firstValidLayerTT> != 0, the packet is tunnel terminated, token.firstValidLayer <= token.firstValidLayer + <firstValidLayerTT> **NOTE:** Packet is tunnel terminated only when the end packet command is forward.
`copyTTLfromTunnelHeader`	1	Copy tunnel header TTL into the inner packet. Relevant when inner packet is routed.
`mirrorToAnalyzerMask`	3	Mirroring session ID. Valid only if enableMirror == 1.
`bridgeDomain`	16	Packet bridge domain attribute if <setBd>=1, set token.bridgeDomain <= <bd>
`ingressVif`	16	Packet ingress virtual interface if <setIngressVif>=1, set token.ingressVif <= <ingressVif>
`reserved / Metadata`	8	Reserved for future use.
`counterIndex`	16	Counter ID as configured in the counter block (table hit address by default/not specified here).
`isV11`	1	0 - VPLS Case (P2MP) 1 - Vll Case (P2P)
`isHub`	1	0 - Attachment Circuit is Spoke 1 - Attachment Circuit is Hub
`IsVpn`	1	1 - VPN Lookup Result 0 - Tunnel Lookup Result
`isP2MPBudNode`	1	1 - P2MP Bud Node 0 - Non Bud Node
`reserved / Metadata`	4	Reserved for future use.
`bridgeAclId`	8	Bridge ACL ID. Valid only if BridgeACLEn == true.
`pAclId`	8	Port ACL ID. Valid only if PACLEn == true.
`routeAclId`	8	Route ACL ID. Valid only if RouterACLEn == true.
`egressVif`	16	VIF to forward when setPktCmd == 1 AND pktCmd == FORWARD or FWD_MIRROR.
`ecmpSize`	8	L2/3 ECMP when setPktCmd == 1 AND pktCmd == FORWARD or FWD_MIRROR.
Table Key		
Entry Format	4	X - Over IPv4
reserved	8	Reserved.
Label	20	Label (VC Label)

Table B–10 X Over MPLS Tunnel Termination Table (LER and L2/L3 VPN Only)

Field Name	Size (b)	Description
reserved	4	
Label	20	Label (Virtual Circuit Label)
reserved	40	
ingressVif	16	Ingress VIF based on port.
outerVLAN	12	
reserved	4	

B.3.1.6 Local VTEP Table

Type: Hash
Key Size: 64b
Data Size: 64b

Table B–11 Local VTEP Table

Field Name	Size (b)	Description
Table Entry		
reserved / Metadata	64	Application specific
Table Key		
`tnlType`	4	Type of tunnel.
`IVtepIp`	32	IP address of my VTEP.
Reserved	26	Reserved for future use
`entryFormat`	2	Entry Format: 0x01 - Local VTEP table, 0x10 - TunnelId table

B.3.1.7 Tunnel ID Table

Type: Hash
Key Size: 64b
Data Size: 64b

Table B–12 Instance ID Table

Field Name	Size (b)	Description
Table Entry		
`setBricgeDomain`	1	Enable setting the bridge domain attributes.
`bridgeAclEn`	1	
`routerAclEn`	1	
`reserved / Metadata`	5	Reserved for future use.
`BridgeDomain`	16	Packet bridge domain attribute if <setBd>=1, set token.bridgeDomain <= <bd>
`bridgeAclId`	8	
`routeAclId`	8	
`firstValidLayerTT`	4	When <firstValidLayerTT> != 0, the packet is tunnel terminated, token.firstValidLayer <= token.firstValidLayer + <firstValidLayerTT> **NOTE:** Packet is tunnel terminated only when the end packet command is forward.
`reserved / Metadata`	20	Reserved for future use.

Table B–12 Instance ID Table

Field Name	Size (b)	Description
Table Key		
TunnelType	4	
TunnelId	24	
Reserved	34	Reserved for future use.
`entryFormat`	2	

B.3.1.8 FDB Table

Type: Hash
Key Size: 64b
Data Size: 64b

Table B–13 FDB Table

Field Name	Size (b)	Description
Table Entry		
`pktCmd`	2	Packet command for MAC DA lookup to be copied to token. Refer to Appendix C, Generic Packet Commands.
`mirrorMask`	3	Mirror Entry, Up to 2 Copies
`macDAIsControl`	1	Relevant for MAC DA lookup only: When set, this MAC DA is considered as control and it bypasses port ingress span state.
`routerMac`	1	Relevant for DS lookup only: When set, classified the MAC DA of packet MAC as router MAC /MAC to Me.
`countMode`	3	b0: counting enabled b1: policing enabled b2: sampling enabled
`ecmpSize`	3	Relevant for MAC DA lookup only: This the ECMP block size, Actual egressVif = <egressVif> + token.PacketHash % <ECMPSize>
`encapType`	2	0 - Untagged/Single Tagged Encap Type 1 - Double Tagged 2 - EoMPLS (VPLS Case) 3 - EoIPTinnl (VxLAN/NvGRE Cases)
`isHub`	1	Only useful for VPLS case. Needed for Split Horizon run for VPLS/H-VPLS where packet coming on MPLS spoke tunnel should not go back on to spoke tunnels.
`isStaticMac`	1	Specifies static MAC. Useful in MAC SA learning to prevent station movement.
`isL2OverIPTunnel`	1	Specifies MAC is learned over L2 IP tunnels. VxLAN/NvGRE. relevant for DA lookups only.
`reserved`	6	Reserved for future use.
`tnnlId`	12	12 bits of egress interface Tunnel ID. The next fields (reserved / Metadata, tnnlIdOrModPtr0, and tnnlIdOrModPtr1) contain the other 12 bits of the tunnel ID to form a 24-bit value.
`reserved / Metadata`	4	Reserved for future use.
`tnnlIdOrModPtr0`	4	Modification pointer.
`tnnlIdOrModPtr1`	4	Modification pointer.

Table B–13 FDB Table

Field Name	Size (b)	Description
vif	16	Virtual interface associated with this MAC. MAC DA lookup: - set this egressVif to token.egressVif - ingressVif filtering is done in Rewrite based on per ingressVif localEn config MAC SA lookup - compare token.ingressVif to this <ingressVif>; if they do not match, issue a moved address (NA)
Table Key		
MAC Address	48	MAC DA or MAC SA.
BridgeDomain	14	Packet bridge domain.
Entry/Key Type	2	0-MAC entry

B.3.1.9 IPv4 Multicast Bridge Table

Type: Hash
Key Size: 128b
Data Size: 128b

Table B–14 IPv4 Multicast Bridge Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command for multicast group to be copied to token. Refer to Appendix C, Generic Packet Commands.
mirrorMask	3	Mirror entry; up to two copies.
controlAddress	1	The IP address is control; packet is trapped to CPU.
staticMAC	1	Static multicast group; entry is not subjected to aging.
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
Reserved	22	Reserved for future use.
ingressOrEgressVif	16	Virtual interface associated with this multicast group: - set this egressVif to token.egressVif - ingressVif filtering is done in Rewrite based on per ingressVif localEn config
counterIndex	16	counterId for <countMode>
ECMPSize	8	Relevant for MAC DA lookup only: This the ECMP block size, Actual egressVif = <egressVif> + token.PacketHash % <ECMPSize>
Reserved	56	Reserved for future use.
Table Key		
bd	16	Interface bridge domain.
GroupAddress (DIP)	32	IPv4 multicast destination address.
Reserved	16	Reserved; must be set to 0.

Table B–14 IPv4 Multicast Bridge Table

Field Name	Size (b)	Description
SIP	32	IPv4 source address: if ingressVif.IPv4BridgingMode is {S,G,V}, SIP, Else: 32'h0
Reserved	28	Reserved; must be set to 0.
Entry/Key Type	4	Uniquely identify the key/entry.

B.3.1.10 IPv4 Multicast Route Table

Type: Hash
Key Size: 128b
Data Size: 128b

Table B–15 IPv4 Multicast Route Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command to be copied to token for DIP lookup. Refer to Appendix C, Generic Packet Commands.
mirrorMask	3	Mirror entry; up to two copies.
decTTL	1	Relevant for DIP lookup only: Decrement TTL of IP header (in place of IP TTL, place TTL-1).
networkScope	1	Relevant for DIP lookup only: Used for 0 - Internal network 1 - External network
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
RPFCheckType	2	This field specifies the type of RPF check to be performed: 0: Compare RPFCheckValue with the ingress port. 1: Compare RPFCheckValue with the ingress VIF. 2: Compare RPFCheckValue with the ingress BD. 3: Compare RPFCheckValue with PIMBiDirTable.RPFCheckValue.
Reserved	4	Reserved for future use.
RPFCheckValue	16	Value for the RPF check.
RPFCheckFailCmd	2	Packet command to be copied to token in the event of RPF check failure. Refer to Appendix C, Generic Packet Commands.
reserved	6	Reserved for future use.
egressVif	16	This is the egress VIF representing the multicast OIF list for this entry.
reserved	72	Reserved for future use.
Table Key		
GroupAddress (DIP)	32	IPv4 multicast destination address
Reserved	16	Reserved; must be set to 0.
SIP	32	IPv4 source address: if ingressVif.IPv4BridgingMode is {S,G,V}, SIP, Else: 32'h0
reserved	16	Reserved; must be set to 0.

Table B–15 IPv4 Multicast Route Table

Field Name	Size (b)	Description
VRF	16	Virtual router interface.
Reserved	12	Reserved; must be set to 0.
Entry/Key Type	4	Unique key identifier.

B.3.1.11 PIM Bi-Directional Table

Type: Hash
Key Size: 64b
Data Size: 64b

Table B–16 PIM Bi-Directional Table

Field Name	Size (b)	Description
Table Entry		
RPFCheckValue	16	Value for the RPF check.
Reserved	40	Reserved for future use.
Table Key		
BdId	32	Packet bridge domain.
DIP	16	IPv4 destination address.
VRF	16	Virtual router interface.

B.3.1.12 IPv6 Multicast Bridge Table

Type: Hash
Key Size: 256b
Data Size: 256b

Table B–17 IPv6 Multicast Bridge Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command to be copied to token for multicast group. Refer to Appendix C, Generic Packet Commands.
mirrorMask	3	Mirror entry; up to two copies.
controlAddress	1	The IP address is control, packet is trapped to CPU
staticMAC	1	Static multicast group; entry is not subjected to aging.
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
Reserved	22	reserved for future use
ingressOrEgressVif	16	Virtual interface associated with this multicast group: - set this egressVif to token.egressVif - ingressVif filtering is done in Rewrite based on per ingressVif localEn config
counterIndex	16	counterId for <countMode>.
ECMPSize	8	Relevant for MAC DA lookup only: This the ECMP block size, Actual egressVif = <egressVif> + token.PacketHash % <ECMPSize>

Table B–17 IPv6 Multicast Bridge Table

Field Name	Size (b)	Description
reserved	184	Reserved for future use.
Table Key		
ipv6SIP	128	SIP
ipv6DIP	120	DIP
BD	8	Interface bridge domain.

B.3.1.13 IPv4 Host Table

Type: Hash
Key Size: 64b
Data Size: 192b

Table B–18 IPv4 Host Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands..
mirrorMask	2	Mirror entry; up to two copies.
propTTL	1	Relevant for DIP lookup only: Decrement TTL of IP Header (in place of IP TTL, place TTL-1).
isTunnelVif	2	If destination is tunnel.
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
L2HdrINsertPtrOrEncapType	4	Insertion template pointer or encapType.
ecmpSize	4	Size of ecmp (l2 ecmp)
egressBd	14	Egress bridge domain ID
virtualId	24	Virtual Identifier for Different Tunnels VC-Label for MPLS Tunnels VNI for VxLAN and Geneve Tunnels TNI for NvGRE Tunnels ISID for PBB Tunnels S-VLAN for Q-in-Q Tunnels
macDa	48	Next Hop MAC DA
egressVif	16	Relevant for DIP lookup only: Virtual interface associated with this next hop. This egressVIF may represent a physical port/s or a tunnel interface.
reserved	6	
Entry/Key Type	2	Unique key/entry identifier: 1 - IPv4 Host Entry 2 - IPv6 Host Entry
reserved	64	
Table Key		
iPAddress	32	DIP or SIP (for uRPF).

Table B–18 IPv4 Host Table

Field Name	Size (b)	Description
VRF	16	Virtual router ID.
reserved	14	Reserved; must be set to 0.
Entry/Key Type	2	Unique key/entry identifier: 1 - IPv4 Host Entry

B.3.1.14 IPv6 Host Table

Type: Hash
Key Size: 256b
Data Size: 256b

Table B–19 IPv6 Host Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands..
mirrorMask	2	Mirror entry; up to two copies.
propTTL	1	Relevant for DIP lookup only: Decrement TTL of IP Header (in place of IP TTL, place TTL-1)
isTunnelVif	2	If destination is tunnel.
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
L2HdrINsertPtrOrEncapType	4	Insertion template pointer or encapType.
ecmpSize	4	Size of ecmp (l2 ecmp)
egressBd	14	Egress bridge domain ID.
virtualId	24	mplsBos - 1 bit mplsExp - 3 bits tnnlId - 20 bits
macDa	48	Next Hop MAC DA
egressVif	16	Relevant for DIP lookup only: Virtual interface associated with this next hop. This egressVif may represent a physical port/s or a tunnel interface.
Entry/Key Type	2	Unique key/entry identifier 1 - IPv4 Host Entry 2 - IPv6 Host Entry
Table Key		
ipv6AddressLsb	64	IPv6 address LSB.
reserved	60	Reserved; must be set to 0.
Entry/Key Type	4	Unique key/entry identifier: 2- IPv6 Host Entry
ipv6AddressMsb	64	IPv6 address MSB.

Table B–19 IPv6 Host Table

Field Name	Size (b)	Description
vrfId	16	Virtual router ID.
Reserved	44	Reserved; must be set to 0.
Entry/Key Type	4	Unique key/entry identifier: 2- IPv6 Host Entry

B.3.1.15 IPv4 Unicast Route Table

Type: Hash
Key Size: 48b
Data Size: 128b

Table B–20 IPv4 Unicast Route Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands..
mirrorToAnalyzerMask	2	Mirror entry; up to two copies.
propTTL	1	Relevant for DIP lookup only: Decrement TTL of IP Header (in place of IP TTL, place TTL-1)
isTunnelVif	2	
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
ecmpSize	6	Size of ecmp (l2 ecmp).
macDa	48	Next Hop MAC DA
encapType	4	Insertion template pointer or encapType.
reserved	4	
vif	16	Relevant for DIP lookup only: Virtual interface associated with this next hop. This egressVif may represent a physical port/s or a tunnel interface.
nhVirtualId	24	mplsBos - 1 bit mplsExp - 3 bits tnnlId - 20 bits
egressBd	14	Egress bridge domain ID.
Entry/Key Type	2	Unique key/entry identifier: 1 - IPv4 Host Entry 2 - IPv6 Host Entry
Table Key		
ipAddress	32	DIP or SIP (for uRPF).
vrfId	16	Virtual Router ID.

B.3.1.16 IPv6 Unicast Route Table

Type: Hash
Key Size: 144b
Data Size: 128b

Table B–21 IPv6 Unicast Route Table

Field Name	Size (b)	Description
Table Entry		
`pktCmd`	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands..
`mirrorToAnalyzerMask`	2	Mirror entry; up to two copies
`propTTL`	1	Relevant for DIP lookup only: Decrement TTL of IP Header (in place of IP TTL, place TTL-1)
`isTunnelVif`	2	
`countMode`	3	b0: counting enabled b1: policing enabled b2: sampling enabled
`ecmpSize`	6	Size of ecmp (l2 ecmp).
`macDa`	48	Next Hop MAC DA
`encapType`	4	Insertion template pointer or encapType.
`reserved`	4	
`vif`	16	Relevant for DIP lookup only: Virtual interface associated with this next hop. This egressVif may represent a physical port/s or a tunnel interface.
`nhVirtualId`	24	mplsBos - 1 bit mplsExp - 3 bits tnnlId - 20 bits
`egressBd`	14	Egress bridge domain ID.
`Entry/Key Type`	2	Unique key/entry identifier: 1 - IPv4 Host Entry 2 - IPv6 Host Entry
Table Key		
`compIpv6Addr`	128	DIP or SIP (for uRPF).
`vrfId`	16	Virtual Router ID.

B.3.1.17 Table

Type: TCAM
Key Size: 192b
Data Size: 128b

Table B–22 Table

Field Name	Size (b)	Description
Table Entry		
`trap`	1	
`srcAddress`	32	Source IP address.
`srcPort`	16	TCP/UDP Port.

Table B–22 Table

Field Name	Size (b)	Description
destAddress	32	Destination IP address.
dstPort	16	Relevant for DIP lookup only: Used for 0 - Internal network 1 - External network
eVlan	16	b0: counting enabled b1: policing enabled b2: sampling enabled
reserved / Meta data	14	Reserved for future control bits.
Table Key		
Type	1	
SrcAddress	32	Source IP address.
SrcPort	16	TCP/UDP port.
DestAddress	32	Destination IP address.
DestPort	16	Relevant for DIP lookup only: Used for 0 - Internal network 1 - External network
Bd	16	Packet bridge domain attribute.
Flag	9	
Protocol	8	
reserved	62	Reserved for future use.

B.3.1.18 MPLS Label Table

Type: Hash
Key Size: 32b
Data Size: 153b

Table B–23 MPLS Label Table

Field Name	Size (b)	Description
Table Entry		
pktCmd	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands.
mirrorMask	2	Mirror entry: up to two copies.
propTTL	1	Relevant for DIP lookup only: Decrement TTL of IP Header (in place of IP TTL, place TTL-1)
isTunnelVif	2	
countMode	3	b0: counting enabled b1: policing enabled b2: sampling enabled
ecmpSize	6	Size of ecmp (l2 ecmp).
macDa	48	Next Hop MAC DA
l2HdrInsrtPtr	4	Insertion template pointer or encapType.
available	4	

Table B–23 MPLS Label Table

Field Name	Size (b)	Description
egressVif	16	Relevant for DIP lookup only: Virtual interface associated with this next hop. This egressVif may represent a physical port/s or a tunnel interface.
virtualId	24	mplsBos - 1 bit mplsExp - 3 bits tnnlId - 20 bits
egressBd	14	Egress bridge domain ID
Entry/Key Type	2	Unique key/entry identifier: 1 - IPv4 Host Entry 2 - IPv6 Host Entry
isMplsPopOper	1	
mplsSwapLabel	24	
Table Key		
Reserved	4	Reserved; must be set to 0.
Label	20	Outermost MPLS tunnel label.
Reserved	6	Reserved; must be set to 0.
Entry/Key Type	2	Unique key/entry identifier.

B.3.1.19 PACL, BACL, RACL (Ingress ACL Tables)

Type: TCAM
Key Size: flexible (64b, 128b, 192b, 384b)
Default profile IPv4 and IPv6 table keys are both 384b.
Data Size: 128b

Table B–24 PACL, BACL, RACL Tables

Field Name	Size (b)	Description
Table Entry		
isTerminal	1	Sets entry priority.
enPktCmdUpd	1	Enable packet command update.
enRedirectToEvif	1	Enable redirect to egress VIF.
enRsnCodeUpd	1	Enable reason code update.
enPolicer	1	Enable policer.
enCnt	1	Enable count.
enMirrorSsnUpd	1	Enable mirror session update.
remarkTcp	1	Enable TCP.
remarkDscp	1	Enable DSCP marking.
remarkPcp	1	Enable PCP marking.
pktCmd	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands..
TC	4	QoS Traffic Class value.
mirrorSessionId	2	Mirror session ID.
encapType	2	Encap type.
egrVifId	16	Egress VIF ID.
policerId	16	Policer ID.

Table B–24 PACL, BACL, RACL Tables

Field Name	Size (b)	Description
rsnCode	10	Reason code.
PCO	3	PCP value.
DSCP	6	DSCP value.
instanceId	24	Instance ID.
Reserved	149	Reserved.
IPv4 Table Key		
keyType	2	0 - L2/IPv4
iAclId	8	iAclId allows grouping of rule/application of rules on a set of interfaces. Assigned based on ingress interface (together with IACL enable).
macDA	48	MAC DA
macSA	48	MAC SA
EtherType	16	
CTAG (VID, DEI, PCP)	16	
STAG (VID, DEI, PCP)	16	
DIP	32	Destination IP.
SIP	32	Source IP.
L4DestPort	16	L4 Destination port number.
L4SourcePort	16	L4 Source Port number.
iVif	16	Ingress VIF.
icmpMessageType	8	ICMP message type.
Protocol	8	IPv4 protocol number.
Dscp + isCTagged + isStagged	8	DSCP value,i tagged meta data.
Bd	16	Bridge domain.
ipv4DfSet + ipv4MfSet	2	Fragment.
IPv6 Table Key		
keyType	2	1 - IPv6
iAclId	8	iAclId allows grouping of rule/application of rules on a set of interfaces. Assigned based on ingress interface (together with IACL enable).
DIP	128	Destination IP.
SIP	128	Source IP.
Bd	16	Bridge domain.
Next Header	8	
L4DestPort	16	L4 Destination port number.
L4SourcePort	16	L4 Source Port number.
icmpMessageType	8	ICMP message type.
Hop Limit	8	
EtherType	16	
Router Mac + isTcp + isUdp	3	

B.3.1.20 EACL

Type: TCAM
Key Size: flexible (64b, 128b, 192b, 384b)
Default profile IPv4 and IPv6 tablekeys are both 384b.
Data Size: 128b

Table B–25 EACL Table

Field Name	Size (b)	Description
Table Entry		
`enPktCmdUpd`	1	Enable packet command.
`enRsnCodeUpd`	1	Enable reason code.
`pktCmd`	2	Packet command to be copied to token. Refer to Appendix C, Generic Packet Commands.
`rsnCode`	10	Reason code value.
`remarkDscp`	1	Update flag DSCP.
`remarkExp`	1	Update flag EXP.
`remarkPcp`	1	Update flag PCP.
`mirrorSsnUpd`	1	Update flag Mirror Session.
`mirrorSessionId`	3	Mirror session.
`enCnt`	1	Update count.
`enPolicer`	1	Update policer.
`dscpVal`	6	DSCP value.
`expVal`	3	EXP value.
`pcpVal`	3	PCP value.
`counterId`	16	Counter ID.
`policerId`	16	Policer ID.
IPv4 Table Key		
`keyType`	8	0 - L2/IPv4.
`macDA`	48	MAC DA.
`macSA`	48	MAC SA.
`DIP`	32	Destination IP.
`SIP`	32	Source IP.
`L4DestPort`	16	L4 destination port number.
`L4SourcePort`	16	L4 source port number.
`EtherType`	16	
`icmpMessageType`	8	ICMP message type.
`Protocol`	8	IPv4 protocol number.
`Egress Bd`	16	
`Egress VIF`	16	
`InstanceId`	24	VNI/TNI/Egress VLAN.
`DSCP + PCP + EXP`	16	QoS.
`TCP Flags + Reason Code`	19	
IPv6 Table Key		

Table B–25 EACL Table

Field Name	Size (b)	Description
keyType	8	1 - IPv6.
DIP	128	Destination IP.
SIP	128	Source IP.
L4DestPort	16	L4 destination port number.
L4SourcePort	16	L4 source port number.
EtherType	16	
icmpMessageType	8	ICMP message type.
Protocol	8	IPv4 protocol number.
Egress Bd	16	
Egress VIF	16	
Reason Code + DSCP	16	
PCP + EXP	6	

B.3.1.21 Egress Bridge Domain Table

Type: Direct Access
Key Size: 16b
Data Size: 128b

Table B–26 Egress Bridge Domain Table

Field Name	Size (b)	Description
Table Entry		
ignore	1	Check if enabled on interface.
mirrorId	3	Mirror Session ID.
mirrorEn	1	Enable Mirroring.
Reserved	3	Reserved.
egressPortFilterId	8	Egress Port Filter ID.
macSA	8	Lower 8 bits of MAC SA.
instanceId	24	Service Instance / Tunnel ID.
Reserved	80	Reserved.
Table Key		
egressBd	16	Egress Bridge Domain

B.4 ISME Tables

B.4.1 BD Table

Type: Hash
Key Size: 16b
Data Size: 128b

Table B–27 BD Table

Field Name	Size (b)	Description
Table Entry		
macSAmode	1	Bridge attributes: 0: lookup is disabled 1: lookup is enabled
setEgressPortFilter	1	Set to 0 - Parser assigns Token.egressPortFilgerGroup <= PortCfg.egressPortFilterID
macSAmissCmd	2	Packet command to be copied to token when FDB lookup for SA results in a miss. Refer to Appendix C, Generic Packet Commands.
ipv4MulticastBridgeMode	2	Bridge attributes: 00: {MAC_DA, BD} - Lookup in FDB 01: (*, G, BD) {0_G_MODE} - Lookup in Bridged IPM Table 10: (S, G, BD) {S_G_MODE} - Lookup in Bridged IPM Table 11: reserved
ipv6MulticastBridgeMode	2	Bridge attributes: 00: {MAC_DA, BD} - Lookup in FDB 01: (*, G, BD) {0_G_MODE} - Lookup in Bridged IPM Table 10: (S, G, BD) {S_G_MODE} - Lookup in Bridged IPM Table 11: reserved
unregisteredMulticastCmd	2	Packet command to be copied to token when FDB lookup for Multicast or broadcast MAC DA results in a miss. Refer to Appendix C, Generic Packet Commands.
broadcastCmd	2	Packet command to be copied to token for broadcast/flooded unicast/multicast traffic. Refer to Appendix C, Generic Packet Commands.
unknownUnicastCmd	2	Packet command to be copied to token when FDB lookup for unicast MAC DA results in a miss. Refer to Appendix C, Generic Packet Commands.
ipv4ARPBCCmd	2	Packet command for IPv4 ARP Broadcast control packets. Refer to Appendix C, Generic Packet Commands. Reason Code if not Forward = XP_BRIDGE_RC_IVIF_ARP_CMD.
ipv4IgmpCmd	2	Packet command for IPv4 IGMP packets. Refer to Appendix C, Generic Packet Commands. Reason Code if not Forward - XP_BRIDGE_RC_IVIF_IGMP_CMD
ipv6IcmpCmd	2	Packet command for IPv6 ICMP packets. Refer to Appendix C, Generic Packet Commands. Reason Code if not Forward = XP_BRIDGE_RC_IVIF_ICMP6_CMD.
setVRF	1	
ipv4UnicastRouteEn	1	Router attributes: 0: IPv4 Unicast Routing is disabled, packets sent to router MAC are dropped 1: IPv4 Unicast Routing is enabled, packets sent to router MAC are routed
ipv6UnicastRouteEn	1	Router attributes: 0: IPv6 Unicast Routing is disabled, packets sent to router MAC are dropped 1: IPv6 Unicast Routing is Enabled, packets sent to router MAC are routed

Table B–27 BD Table

Field Name	Size (b)	Description
`mplsRouteEn`	1	MPLS router attributes: 0: MPLS Routing Disable 1: MPLS Routing Enable
`floodVif`	16	
`VRF`	16	Router attributes: Packet bridge domain if <setBridgeDomain>=1, set token.bridgeDomain <= <bridgeDomain>
`egressPortFilterID`	8	
`ipv4MulticastRouteEn`	1	Router attributes: 0 - IPv4 Multicast Routing is disabled, IPv4 Multicast packets are bridged ONLY 1 - IPv4 Multicast Routing is Enabled, IPv4 Multicast packets are bridged and routed
`ipv6MulticastRouteEn`	1	Router attributes: 0 - IPv6 Multicast Routing is disabled, IPv4 Multicast packets are bridged ONLY 1 - IPv6 Multicast Routing is Enabled, IPv4 Multicast packets are bridged and routed

`ipv4MulticastRouteMode`	1	Router attributes:
		0 - {*, G, V}
		1 - (S, G, V}

`ipv6MulticastRouteMode`	1	Router attributes:
		0 - {*, G, V}
		1 - (S, G, V}

`policyBasedRoutingEn`	1	Router attributes: Enable policy-based routing for this interface. Policy-based routing lookup is done in ACLs Engine. The policy-based routing entry is forwarded to the Bridging/Routing Engine for resolution.
`mirrorToAnalyzerMask`	3	set to 0 - Parser already assigns token.mirroMask <= PortCfg.mirrorMask.
`countMode`	3	b0: counting enabled b1: policing enabled b2: sampling enabled
`mirrorEnable`	1	Mirror packet.
`Mode`	1	
`Scope`	1	
`reserved / Metadata`	9	Reserved for future control bits.
`counterIndex`	16	Counter ID for <countMode>.
`vllVpnLabel`	24	MPLS Label. Applicable for VLL case.
Table Key		
`bridgeDomain`	16	Based on ingressVif assigned.

B.4.2 MPLS Tunnel Table

Type: Hash
Key Size: 32b
Data Size: 128b

Table B–28 MPLS Tunnel Table

Field Name	Size (b)	Description
Table Entry		
`isP2MPBudNode`	1	Specifies P2MP Bud Node case.
`innerTermieLabelPos`	2	Label position to use for inner label lookup.
`countMode`	3	b0: counting enabled b1: policing enabled b2: sampling enabled
`copyTTLfromTunnelHeader`	1	Copy Tunnel Header TTL into the inner packet. Relevant when inner packet is routed
`reserved`	17	Reserved for future use.
`floodVif`	16	Packet ingress virtual interface: if <setVif>=1, set token.ingressVif <= <ingressVif>
`counterIndex`	16	Counter ID as configured in the counter block (table hit address by default/not specified here).
`reserved`	72	Reserved for future use.
Table Key		
Outer VLAN Id	12	Outer VLAN ID.
MPLS Label	20	Outermost MPLS tunnel label.

B.5 Update Rewrite Engine Tables

B.5.1 VIF (Virtual Interface) Table

Type: Direct Access
Data Size: 256b

Table B–29 Update Rewrite Engine Tables

Field Name	Size (b)	Description
Table Entry		
`EntryType`	2	3 - 256b- Ingress LAG Filtering, egress Multicast + MRE.
`truncate`	1	Truncate indication. When set, only packet header of up to 256B is transmitted.
`MTUProfile`	3	Relevant for egress only: This MTU Profile points to one of eight MTUs. If final packet BC (after all modifications and insertions) is greater than MTU, packet is treated based on MTUExceedCmd Associated with this MTU Profile. MTUExceedCmd can be: FWD, FWD+COPY2CPU (with reason code, MTUExceeded), DROP, TRAP to CPU (with reason code, MTUExceeded).
`MirrorEn`	1	Ingress: Mirror to ingress VIF ingress session (based on config). Egress: Mirror to egress VIF egress session (based on config).
`qMirrorEn`	1	Relevant for egress only: When set, it means that target queue for this packet may be mirrored, set RefCntValid to 1.

Table B–29 Update Rewrite Engine Tables

Field Name	Size (b)	Description
portsBitmap[7:0]	8	Ingress: Used to filter packets from being forwarded back to their source LAG. To disable filtering, set this to all zeros. Egress: This is the packet target physical port bitmap on this device. To disable packet forwarding, set this to all zeros
rewritePtr0	8	Rewrite pointer for header modification; invalid Ptr is 0xFF.
rewritePtr1	8	Rewrite pointer for header modification; invalid Ptr is 0xFF.
rewriteData	32	Rewrite data.
insertPtr0	16	Insertion pointer for pre-pending a header to the packet; invalid Ptr is 0xFFFF.
insertPtr1	16	Insertion pointer for pre-pending a header to the packet; invalid Ptr is 0xFFFF.
insertPtr2	16	Insertion pointer for pre-pending a header to the packet; invalid Ptr is 0xFFFF.
MREPtr	16	Pointer to MRE for replicating packets through a linked list of VIFs; invalid Ptr is 0xFFFF.
portsBitmap[135:8]	128	See: portsBitmap[7:0].
Table Key		
Directly indexed by IVIF/EVIF ID, offset by well known value coming from SE VIF RSP TABLE.		

Note: also accounts for L2 ECMP by EVIF ID + (L2 ECMP HASH A/B % L2 ECMP SIZE) prior to offsetting

B.5.2 Trunk Resolving Table

Type: Direct Access
Data Size: 136b

Table B–30 Trunk Resolving Table

Field Name	Size (b)	Description
Table Entry		
PortMask	128	A port mask considering all physical port in the device. This chosen mask is then AND with the ports derived from egressVIF to determine the final set of egress ports.
Reserved	20	Reserved for future use.
Table Key		
Directly indexed by LAG HASH A/B % 8 (A/B determined by TOKEN.TT BIT)		

B.5.3 Egress Filtering Table

Type: Direct Access
Data Size: 136b

Table B–31 Egress Filtering Table

Field Name	Size (b)	Description
Table Entry		
EfltData	128	A port-filtering mask considering all physical ports in the device.
Reserved	8	Reserved for future use.

Table B–31 Egress Filtering Table

Field Name	Size (b)	Description
Table Key		
Directly indexed by TOKEN.FILTER_GRP_NUM.		

B.5.4 CPU Reason Code Table

Type: Direct Access
Data Size: 16b

Table B–32 CPU Reason Code Table

Field Name	Size (b)	Description
Table Entry		
`DestPort`	8	Destination port for CPU packets with this reason code.
`Tc`	4	Traffic class for CPU packets with this reason code.
`Truncation`	1	When set, only packet header of up to 256B is transmitted.
`Format`	1	If 1, egress modifications will be done as per token rewrite instructions and modified insert instructions (based on insert pointer from this table). If 0, egress modifications will NOT be performed. Only insert corresponding to insert pointer from this table will be performed.
Table Key		
Directly indexed by TOKEN.REASON_CODE[9..0].		

B.5.5 Template Lookup Table

Type: Direct Access
Data Size: 136b

Table B–33 Template Lookup Table

Field Name	Size (b)	Description
Table Entry		
`Layer0 Info`	17	● 8-bit offset of physical layer0 in the token's Layers portion ● 6-bit type of physical layer0 ● 3-bit checksum info of physical layer0 **ID** **Checksum Info** **Description** 0 OTHER Other layers 1 IPv4 IPv4 layer 2 IPV6 IPv6 layer 3 TCP TCP layer 4 UDP UDP layer
`Layer1 Info`	17	● 8-bit offset of physical layer1 in the token layers portion ● 6-bit type of physical layer1 ● 3-bit checksum info of physical layer1
`Layer2 Info`	17	● 8-bit offset of physical layer2 in the token layers portion ● 6-bit type of physical layer2 ● 3-bit checksum info of physical layer2

Table B–33 Template Lookup Table

Field Name	Size (b)	Description
`Layer3 Info`	17	• 8-bit offset of physical layer3 in the token layers portion • 6-bit type of physical layer3 • 3-bit checksum info of physical layer3
`Layer4 Info`	17	• 8-bit offset of physical layer4 in the token layers portion • 6-bit type of physical layer4 • 3-bit checksum info of physical layer4
`Layer5 Info`	17	• 8-bit offset of physical layer5 in the token layers portion • 6-bit type of physical layer5 • 3-bit checksum info of physical layer5
`Layer6 Info`	17	• 8-bit offset of physical layer6 in the token layers portion • 6-bit type of physical layer6 • 3-bit checksum info of physical layer6
`Layer7 Info`	17	• 8-bit offset of physical layer7 in the token layers portion • 6-bit type of physical layer7 • 3-bit checksum info of physical layer7
Table Key		
Directly indexed by Template ID.		

B.5.6 Insertion Data Table

Type: Direct Access
Data Size: 256b

Table B–34 Insertion Data Table (INS0, INS1, INS2)

Field Name	Size (b)	Description
Table Entry		
`EntryType`	2	3 – 256b mode
`Reserved`	6	
`insertTemplatePtr`	8	Insert template pointer; invalid Ptr is 0xFF.
`InsertData`	240	Insert data.
Table Key		
Directly indexed by InsertProfileKey.		

B.5.7 Modify Instruction Table

Type: Direct Access
Data Size: 108b

Table B–35 Modify Instruction Table

Field Name	Size (b)	Description
Table Entry		
`CmdNum`	2	Instruction enumeration.
`Order`	1	Indicates in which internal modify stage of the URW block the instruction will be applied.
`Layer`	2	Indicates which of the first four layers the instruction operates on. The value is offset by the FirstValidLayer.
`Instruction`	100	If the Cmd is Delete: ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion, in (Size + 1)B, maximum is 8B ● 91b Reserved, for future use If the Cmd is Copy or Move: ● 4b Source, starting location in the Token Layer to copy or move, from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction manipulation (Size + 1), maximum is 8B ● 6b Target, target location within the current Layer to copy or move, to ● 9b bitmask (LSB is set to 1 when applies mask on target byte else (target+size-1)th byte) ● 8b bitmap, each bit represent offset relative to target. If 1, copy or move a constant, else copy or move from source ● 64b ConstData (relevant for Copy command only), the data to be copied or moved. The data is in the network order, i.e. byte 0 => 63:56, byte 7 => 7:0 ● 91b Reserved, for future use
`Cmd`	3	0 – NOP 1 – DEL 2 – COPY 3 – MOVE
Table Key		
Directly indexed by the Modify Instruction Pointer, first 8 valid are used.		

B.5.8 Insert Instruction

Type: Direct Access
Data Size: 248b

Table B–36 Insert Instruction

Field Name	Size (b)	Description
Table Entry		
`overlappingSourceLayer`	3	If set to 1, first 32 bytes of token layers will be taken as source layer and sourceLayer field below will be ignored.
`sourceLayer`	2	Token Layer data (0 => 1st valid layer). This field not applied if overlappingSourceLayer == 1.

Table B–36 Insert Instruction

Field Name	Size (b)	Description
`totalSize`	1	Total size of inserted data: 0 => No bytes inserted, 1=> 1 byte inserted. Max value is 32.
`mdfyCmd0`	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
`mdfyCmd1`	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
`mdfyCmd2`	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
`mdfyCmd3`	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
`mdfyCmd4`	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
`mdfyCmd5`	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)

Table B–36 Insert Instruction

Field Name	Size (b)	Description
mdfyCmd6	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the current layer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
mdfyCmd7	27	● 1b Cmd (0-NOP, 1-COPY) ● 3b Source, starting location in the token layer to insert from ● 6b Start, offset within the current layer ● 3b Size, data size of the instruction insertion (Size + 1), maximum is 8B ● 5b Target, target location within the currentlLayer to insert to ● 9b bitmask (LSB applies mask on (target+size-1)th byte)
metadata	10	● 1b lyrChksumInfo ● 7b LengthAdjust, to be added to the modified packet length to derive the Header.total_length of the layer and written back to the header-layer ● 2b Reserved for future use
Table Key		
Directly indexed by the insert template ID, within the range 0-25.		

B.5.9 Insertion Constant Data Table

Type: Direct Access
Data Size: 288b

Table B–37 Insertion Constant Data Table

Field Name	Size (b)	Description
Table Entry		
ConstData	256	Eight x 32-byte of constant data.
BitMap	32	Each bit set tells the corresponding constant byte is valid.
Table Key		
Directly indexed by InsertTemplatePointers.		

Note: The data is programmed in little endian.

B.6 Port Tables

B.6.1 Port Configuration Table

Type: Direct Access
Data Size: 128b

Table B–38 Port Configuration Table

Field Name	Size (b)	Description
Table Entry		
`setIngressVif`	1	0: no action 1: port default ingress VIF overrides other IVIF assignments
`setBridgeDomain`	1	0: no action 1: set token.bridgeDomain <== bridgeDomain
`acceptedFrameType`	2	Bridge attributes: 00: Accept All Frames 01: Accept only UNTAGGED and PRIORITY_TAGGED frames 10: Accept only TAGGED frames 11: reserved
`bypassTunnelVif`	1	0: Tunnel Termination Engine is enabled 1: Tunnel Termination Engine is disabled
`bypassACLsPBR`	1	0: Policy Engine is enabled 1: Policy Engine is disabled
`samplerEn`	1	
`policerEn`	1	
`portState`	2	Bridge attributes: 00 - DISABLED (data packets are forwarded, BPDU is flooded) 01 - LEARNING (data packets are dropped, but mac learning is triggered) 10 - FORWARDING (data packets are forwarded, mac learning is triggered) 11 - BLOCKING (data packets are dropped, no mac learning, BPDU is trapped to CPU)
`setEgressPortFilter`	1	0: no action 1: do not override the port egress filterID
`macSAmissCmd`	2	Packet command when FDB lookup for SA results in a miss. Refer to Appendix C, Generic Packet Commands.
`bypassBridgeRouter`	1	0: Bridging and Routing engine is enabled 1: Bridging and Routing engine is disabled
`portACLEn`	1	In Ingress Policy Engine, enable Port ACLs TCAM lookup.
`portDebugEn`	1	
`portAclId`	8	Port ACL ID. Valid only if portACLEn == true.
`bridgeDomain (Reserved for Port)`	16	Bridge attributes: Packet bridge domain if <setBridgeDomain>=1, set token.bridgeDomain <= <bridgeDomain>
`Reserved`	24	Reserved.
`mirrorBitMask`	16	
`ingressVif`	20	
`pvidModeAllPkt`	1	
`Pvid`	12	
`mplsQosProfileIdx`	3	
`ipQosProfileIdx`	3	

Table B–38 Port Configuration Table

Field Name	Size (b)	Description
`L2QosProfileIdx`	3	
`etagExists`	1	
Table Key		
Directly indexed by Port ID.		

Appendix C

Generic Packet Commands

Table C–1 Generic Packet Commands

Packet Command	Actions
DROP(0)	Drop the packet. Some of the reasons that may cause a packet drop are: ● Token<PktCmd> = Drop. ● Traffic Manager Query gives a drop response. ● destBitMap is all 0s. ● VIF response is MRE, but MRE is flow controlling. ● MTU error happens on the packet, but MRE is flow controlling.
FWD(1)	Forward to the egress destination, based on the destination VIF, packet header, and the token.
TRAP(2)	Trap the packet to the host CPU. Encapsulate the packet header into the XP header.
FWD+MIRROR(3)	Generate a TRAP copy to the host CPU in addition to the original packet. **NOTE:** The copy to the CPU is not eligible for mirroring.

Appendix D

XP Header

The following table layouts assume big-endian ordering.

D.1 XPH Template

Table D–1 XPH Template

Bit	Field Name	Size (b)	Description)
[63:0]	`pktReceiveTimeStamp`	64	Received Packet Timestamp
[67:64]	`headerType`	4	The type of header: 0 - TO CPU 1 - FROM CPU 2 - LOOPBACK 3 - 15 - Reserved
[71:68]	`HeaderSize`	4	Size of header in resolution of 8 bytes: 0..2 - Reserved 3 - 24 Bytes 4..15 - Reserved
[191:72]	`Header Data`	120	Header Data, based on <headerType>

D.2 XPH TO CPU

Table D–2 XPH TO CPU

Bit	Field Name	Size (b)	Description)
[63:0]	`pktReceiveTimeStamp`	64	Received Packet Timestamp
[67:64]	`headerType`	4	The type of header: 0 - TO CPU
[71:68]	`HeaderSize`	4	3 - 24 Bytes
[79:72]	`ingressPortNum`	8	Device original ingress port number. **Note:** If packet is recirculated, this is the port number at which the packet was first received.
[99:80]	`ingressVif`	20	The ingressVif assigned to the packet.
[103:100]	Reserved	4	Reserved. must be set to all zeros.
[113:104]	`reasonCode`	10	Packet's reasonCode.
[115:114]	Reserved	2	Reserved; must be set to all zeros.
[116:116]	`Truncated`	1	Indicates that the received packet is truncated and only its header is sent to CPU.

Table D–2 XPH TO CPU

Bit	Field Name	Size (b)	Description)
[119:117]	Reserved	3	Reserved; must be set to all zeros.
[127:120]	Reserved	8	Reserved; must be set to all zeros.
[191:128]	`metadata`	64	Metadata from token (token.scratchPad[63:0]) output from either last LDE or MME.

D.3 XPH FROM CPU

Table D–3 XPH FROM CPU

Bit	Field Name	Size (b)	Description)
[63:0]	`pktReceiveTimeStamp`	64	Received packet timestamp.
[67:64]	`headerType`	4	The type of header: 1 - FROM CPU
[71:68]	`HeaderSize`	4	3- 24 Bytes.
[72:72]	`Reserved`	1	Reserved; must be set to all zeros.
[73:73]	`useXPHTimeStamp`	1	For packet timestamp use <pktReceiveTimeStamp> and not the timestamp at which this packet was received.
[77:74]	`TC`	4	Packet traffic class.
[79:78]	`DP`	2	Packet drop precedence.
[80:80]	`txSample`	1	Packet is to be sampled when transmitted by the MAC and its timestamp is to be logged on one of four registers based on <txSampleID[1:0]> .
[82:81]	`txSampleID`	2	
[87:83]	`Reserved`	5	Reserved; must be set to all zeros.
[107:88]	`ingressVif`	20	Packet ingressVif.
[127:108]	`egressVif`	20	Packet egressVif.
[135:128]	`nextEngine`	8	Next engine for this packet.
[191:136]	`metadata`	56	Metadata set to token.scratchpad[55:0] into the parser.

D.4 XPH LOOPBACK

Table D–4 XPH Loopback

Bit	Field Name	Size (b)	Description)
[63:0]	`pktReceiveTimeStamp`	64	Received packet timestamp.
[67:64]	`headerType`	4	The type of header: 2 - LOOPBACK
[71:68]	`headerSize`	4	3- 24 Bytes
[72:72]	`Reserved`	1	Reserved; must be set to all zeros.
[73:73]	`useXPHTimeStamp`	1	For packet timestamp use <pktReceiveTimeStamp> and not the timestamp at which this packet was received.

Table D–4 XPH Loopback

Bit	Field Name	Size (b)	Description)
[77:74]	`TC`	4	Packet traffic class.
[79:78]	`DP`	2	Packet drop precedence.
[87:80]	`Reserved`	8	Reserved; must be set to all zeros.
[107:88]	`ingressVif`	20	Packet ingressVif.
[127:108]	`egressVif`	20	Packet egressVif.
[191:128]	`metadata`	64	Packet metadata set to token scratchpad[55:0]. **NOTE:** When this XPH is generated for a loopback port, metadata[63:56] MUST carry nextEngine to be used in the second round.

Appendix E

APIs

The documentation for the XPliant Software Development Kit is available in the XDK release package, under xdk\doc\html. The XDK documentation has been generated using Doxyfile 1.8.5 and it describes the following components:

```
* - \ref featureLayer
* - \ref primitiveLayer
* - \ref tableManager
```