DesignWare® Cores Compute Express Link (CXL) Controller

Databook

CXL Device
CXL Host
CXL Dual Mode
CXL Switch
Product Codes

Version 6.00a
June 2022

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.

www.synopsys.com

Contents

Revision History

Revision History for Compute Express Link (CXL) controller Databook:

Version	Date	Description
6.00a	June 2022	■ A full list of functional (RTL) changes is in the “Release Notes: PCI Express Cores” document. ■ A change-tracked version of this databook is available at https://www.synopsys.com/dw/doc.php/iip/DWC_pcie/6.00a/doc/DWC_pcie_ctl_cxl_databook_change.pdf ■ The change-tracked version does not indicate minor changes that are made for readability, formatting, and so on. It only indicates changes to functionality, new features, and major rewrites.

Preface

The DesignWare® Cores Compute Express Link (CXL) controller implements the port logic required to build a CXL Device, Host, or Switch.

You can also configure the CXL controller for dual-mode applications. For dual-mode applications, an input pin at power-up determines the CXL Dual Mode controller operating mode as a Device or Host.

For Switch applications, an input pin at power-up determines the CXL Switch controller operating mode as an Upstream Port or Downstream Port.

Synopsys provides a single coreKit for the CXL controller ports. Configure the CXL controller as a Device, Host, Switch, or Dual Mode port by setting the parameters according to Table 1 in the coreConsultant GUI.

Table 1 CXL Controller Configuration

Parameter	Parameter Setting for CXL Controller Configuration			
	CXL Device Port	**CXL Host Port**	**CXL Dual Mode Port**	**CXL Switch Port**
Device Type	CC_DEVICE_TYPE =0	CC_DEVICE_TYPE =1	CC_DEVICE_TYPE=2[a]	CC_DEVICE_TYPE =3[b]
Support Compute Express Link	CX_CXL_ENABLE =1			
CXL Version	CX_CXL_VERSION =1 for CXL 1.1 Port[c] CX_CXL_VERSION =2 for CXL 2.0 Port			
Operating Speed	Gen 5.0 with 32 GT/s x16 link width			

a. The Device_type input pin determines the CXL Dual Mode controller operating mode as a Device or Host at power-up.
b. The Device_type input pin determines the CXL Switch controller operating mode as an Upstream Port or Downstream Port.
c. The CXL 1.1 Port version is not applicable for CC_DEVICE_TYPE=3, that is, CXL Switch Port Controller configuration.

After the CXL controller is configured as a Device, Host, Dual Mode, or Switch Port, depending upon the features licensed, you can identify the CXL controller as:

- **CXL Controller Without AMBA Bridge**: The CXL controller which has its own non-standard, proprietary, dedicated bus interface to the CXL.cache, CXL.mem, and CXL.io logic of your application.
- **CXL Controller With AMBA Bridge**: The CXL controller which has its own non-standard, proprietary, dedicated bus interface to the CXL.cache and CXL.mem logic of your application, but uses the

AMBA bridge to interface with the CXL.io logic of your application. To enable the AMBA bridge, set the parameter `AMBA_INTERFACE` to a value greater than '1'.

Product Codes

Table 2 lists the product codes and product names for DesignWare Cores Compute Express Link (CXL) controller ports.

Table 2 CXL Controller Product Codes

Product Code	Product Name
E354-0	DWC CXL Premium
E355-0	DWC CXL Premium AMBA
F484-0	DWC CXL 2.0 Premium
F485-0	DWC CXL 2.0 Premium AMBA
F486-0	DWC CXL 2.0 Premium AMBA HPC
F487-0	DWC CXL 2.0 Security Interface Add-on[a]

a. This product requires DWC CXL 2.0 IDE Security Module (F910-0) license.

Reference Documentation

After installing the controller, the DWC PCI Express documents can be found under: `$DESIGNWARE_HOME/iip/DWC_pcie_ctl/latest/doc.`

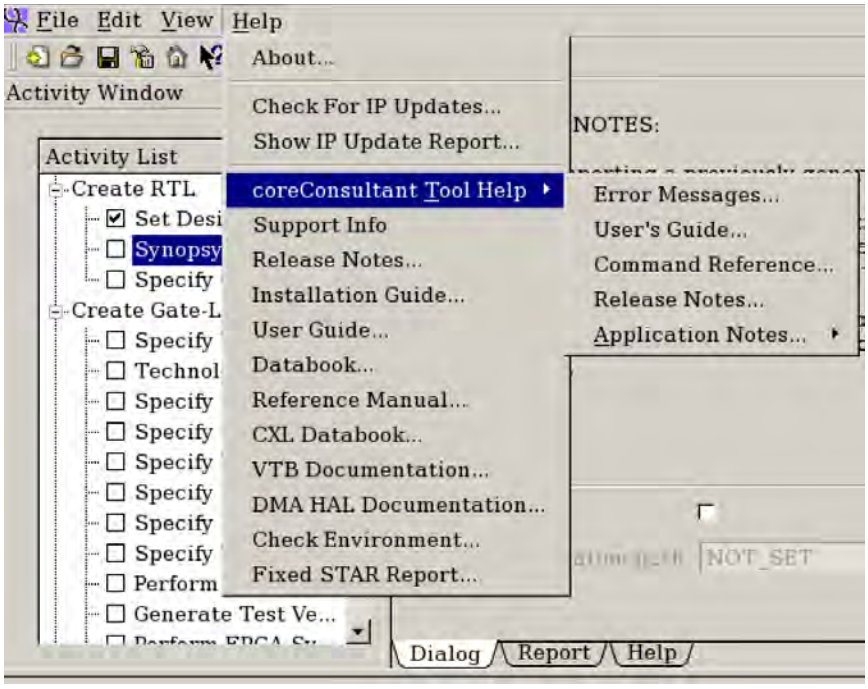
When you create a workspace in coreConsultant,

- Pre-configuration: The DWC PCI Express documents are available in `<workspace>/doc`.
- Post-configuration: Only the documents corresponding to CXL controller are available in `<workspace>/doc`.

When viewing documents from coreConsultant GUI (as shown in Figure 1),

- Pre-configuration: The coreConsultant GUI points to the DM port documents.
- Post-configuration: The coreConsultant GUI points to CXL controller Databook.

Figure 1 Viewing Documents From coreConsultant GUI



Web Resources

- DesignWare IP product information: https://www.synopsys.com/designware-ip.html
- Your custom DesignWare IP page: https://www.synopsys.com/dw/mydesignware.php
- Documentation through SolvNet: https://solvnetplus.synopsys.com (Synopsys password required)
- Synopsys Common Licensing (SCL): http://www.synopsys.com/keys

Terms and Abbreviations

The following terms are used throughout this document.

Table 3 Terms and Descriptions

Term Acronym	Definition
ALMP	ARB/MUX Link Management Packet
ASPM	Active State Power Management
CXL	Compute Express Link
CXL.cache	CXL cache coherency protocol
CXL.mem	CXL memory access protocol
CXL.io	CXL I/O Protocol
D2H	Device to Host

Term Acronym	Definition
DVSEC	Designated Vendor-Specific Extended Capability
DWC	Design Ware Cores
DP	Downstream Port
H2D	Host to Device
IDE	Integrity and Data Encryption
M2S	Master to Subordinate
MEMBAR0	A memory region hosting configuration registers.
PM	Power Management
RCiEP	Root Complex Integrated Endpoint
RCRB	Root Complex Register Block
RX	Receiver
S2M	Subordinate to Master
TX	Transmitter
UP	Upstream Port
vLSM	Virtual Link State Machines

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Customer Support

To obtain support for your product, prepare the required files and contact the support center using one of the methods described:

- Prepare the following debug information, if applicable:
  - For environment set-up problems or failures with configuration, simulation, or synthesis that occur within coreConsultant or coreAssembler, select the following menu:

- **File > Build Debug Tar-file**

  Check all the boxes in the dialog box that apply to your issue. This option gathers all the Synopsys product data needed to begin debugging an issue and writes it to the <*core tool startup directory*>/debug.tar.gz file.

- For simulation issues outside of coreConsultant or coreAssembler:
  - Create a waveforms file (such as VPD or VCD).
  - Identify the hierarchy path to the DesignWare instance.
  - Identify the timestamp of any signals or locations in the waveforms that are not understood.

- *For the fastest response*, enter a case through SolvNetPlus:

a. https://solvnetplus.synopsys.com

Note SolvNetPlus does not support Internet Explorer. Use a supported browser such as Microsoft Edge, Google Chrome, Mozilla Firefox, or Apple Safari.

b. Click the **Cases** menu and then click **Create a New Case** (below the list of cases).

c. Complete the mandatory fields that are marked with an asterisk and click **Save**.

   Make sure to include the following:

   - **Product L1:** DesignWare Cores
   - **Product L2:** PCI Express

d. After creating the case, attach any debug files you created.

   For more information about general usage information, refer to the following article in SolvNetPlus:

   https://solvnetplus.synopsys.com/s/article/SolvNetPlus-Usage-Help-Resources

- Or, send an e-mail message to support_center@synopsys.com (your email will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):
  - Include the Product L1 and Product L2 names, and Version number in your e-mail so it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood
  - Attach any debug files you created.
- Or, telephone your local support center:
  - North America:

    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:

    https://www.synopsys.com/support/global-support-centers.html

1

Product Overview

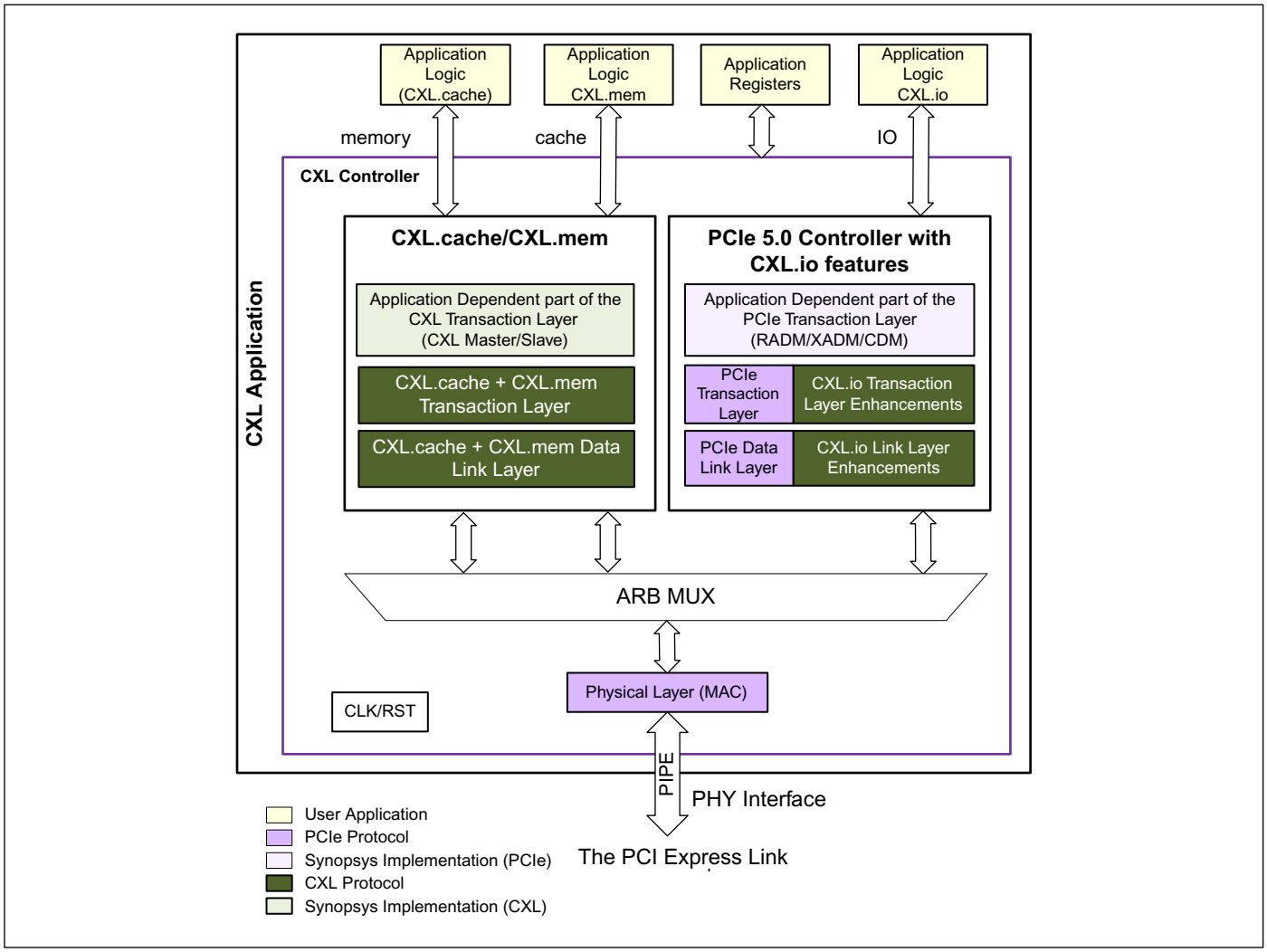
This chapter provides an overview of the DesignWare Cores Compute Express Link (CXL) controller. The following topics are discussed:

1.1 General Product Description

The DesignWare Cores Compute Express Link (CXL) controller provides a solution to implement a CXL Host, Device, or Switch which supports the CXL.io, CXL.cache, and CXL.mem protocols. The current release of the CXL controller is compliant with all the required CXL features as defined in the *Compute Express Link Specification, Revision 3.0, Version 0.7*as well as *PCI Express 5.0, 4.0, and 3.1 specifications*. You can connect the CXL controller to a DesignWare 32 GT/s PHY through the built-in PIPE 5.1.1 interface.

A CXL 1.1 Device is exposed to the Host as a Root Complex Integrated Endpoint (RCiEP). An RCiEP is a PCIe Endpoint logically located inside the Root Complex. The CXL Host and Device implement Root Complex Register Block (RCRB), which exposes link specific registers mapped to the CXL controller's register configuration space.

Figure 1-1 CXL Application



The CXL controller uses the PCIe 5.0 PHY and electricals. You can plug the CXL controller into a system where you can use PCIe through a Flex Bus. Flex Bus is a flexible high-speed port that you can statically configure to support either PCIe or CXL. The CXL controller is configurable for a 16-lane, 8-lane, and 4-lane links. It also supports operation with narrower link partners, for example, configured for x16 but connected

to link partner through 8-lanes. In keeping with the nomenclature of the CXL specification, any link operating at less than 32GT/s or with fewer than 4 lanes is considered to be operating in a degraded mode.

1.2 Applications

Typical applications for a CXL component built with the CXL controller include:

- Device (for example, an accelerator) with cache, requiring cache coherency with Host memory.
- Device (for example, an accelerator) with attached memory (for example, DDR/NVRAM). In this type of system, CXL provides a means for the Host to read and write the Device memory.
- Device (for example, a memory expander) with Host managed memory attached. The Host uses CXL.mem only to send requests to the memory expander.

1.3 Features and Limitations

The CXL controller has the following features and limitations.

1.3.1 Features

The CXL controller supports the following features:

- All the required CXL features defined in the *Compute Express Link Specification, Revision 3.0, Version 0.7*.
- CXL.io, CXL.cache, and CXL.mem protocols
- All the four defined CXL controller configurations, Device, Host, Dual Mode, and Switch as shown in Table 1, for maximum design flexibility
- Dual Mode capability for CXL Host and Device
- PCIe/CXL.io application interface with the Synopsys native interface or the optional ARM AMBA AXI4 and AXI3
- High Speed CXL.cache/CXL.mem interfaces for minimum latency
- Multiple CXL.cache Device to Host Request and Response interfaces to maximize throughput
- Multiple CXL.mem Subordinate to Master Response/No Response channels for enhanced throughput
- Precision Time Management (PTM) support when the controller operates in PCIe mode

Note The controller does not support PTM when the link operates in CXL 1.1 or CXL 2.0 mode.

- Transaction Layer Features
  - Optional PCIe Features required for CXL
    - Data Poisoning by Transmitter
    - Address Translation Services (ATS)
    - PCIe Advanced Error Reporting (AER)
  - Deferrable writes
- Link Layer Features
  - Highly efficient flit-packing algorithm
  - CXL containment feature (“Viral”)
- Physical Layer Features
  - PCIe Alternate Protocol
- PCIe Port and Device Designated Vendor-Specific Extended Capabilities (DVSEC) for CXL
- CXL 1.1 Features
  - CXL Downstream/Upstream Port RCRB support with registers from PCIe Downstream /Upstream Ports, configuration header, and PCIe capabilities (CXL 1.1)
  - CXL MEMBAR0 (CXL 1.1)
  - SR-IOV support for RCiEP and FLR support for CXL.io

- CXL 2.0 Features
  - Limited Multi-Logical Device (MLD) support
  - CXL Reset (Enhanced FLR)
  - CXL 2.0 Component Specific Register
  - Device Disable Caching
  - Device Cache Writeback and Invalidate Cache
  - Device PM Initialization Completion Reporting
  - DevLoad Field support in S2M NDR Message
  - Viral LD-ID Field support in Retry.Ack Payload
  - Backward Compatibility with CXL 1.1 (Option to linkup in CXL 1.1 mode)
- ELBI support for CXL Host


Note Unlike PCIe RP, the CXL Host supports the ELBI interface.

- Industry-leading Reliability, Availability, and Serviceability (RAS) features extended to support new CXL features
- ECAM support for Root Port in PCIe mode and CXL 2.0 mode.


Note
- CXL 1.1 configuration does not support ECAM
- CXL 2.0 Root Port does not support ECAM access to alternate buses

- All the PCIe features defined as required in the *PCI Express Base Specification, Revision 6.0, Version 1.0*
  - Full transaction layer, data link layer, and physical layer
  - Four, Eight, or Sixteen lanes at 32.0, 16.0, 8.0, 5.0, and 2.5 GT/s


Note CXL operation can take place on links running at 8.0, 16.0, or 32.0 GT/s only.

  - 128-bit, 256-bit, and 512-bit datapath width
  - Optional embedded DMA controller with up to 8 read and 8 write channels for high throughput with minimal SoC resource overhead
  - Selected optional ECNs
- PIPE 5.1 or later interface for connection to 32G PCIe 5.0 PHYs using SerDes architecture
- x4, x8, and x16 configurations, supporting operation in x8 (when x16 configuration) and x4 (when x8 or x16 configuration) when connected to either narrower link partners or fewer PHY lanes.
- Supports IDE for CXL controllers with Native interface
- CXS.B Interface

1.3.2 Limitations

- The current release of the CXL controller is not fully compliant with MLD functionality as defined in the *CXL specification*.

  Function 0 is only visible to fabric Manager (`LD_ID =0xFFFF`). All other LDs are assigned a unique function number outside of Function 0. Only one function is supported per LD in case of an MLD.

  The Fabric manager is responsible to map function numbers to LD-ID.
- Power Gating using the UPF flow cannot be enabled when `CX_CXL_ENABLE =1.`
- L1 sub state is not supported when `CX_CXL_ENABLE =1`.
- Only supports IDE for CXL controllers with Native interface.

1.4 Frequency, Speed, and Width Support

For the CXL controller, you must set the frequency, datapath width, and speed modes of the underlying PCIe controller as follows:

- Max Link Speed to 32 GT/s (`CX_MAX_PCIE_SPEED =Gen5(32.0GT/s)`)
- PCIe controller Gen3, Gen4, and Gen5 Freq/Width (`CX_MAC_SMODE_GEN3, CX_MAC_SMODE_GEN4, and CX_MAC_SMODE_GEN5`) to 4s/32-bit per-lane
- PHY Gen1, Gen2, Gen3, Gen4, and Gen5 Freq/Width (`CX_PHY_SMODE_GEN1, CX_PHY_SMODE_-GEN2, CX_PHY_SMODE_GEN3, CX_PHY_SMODE_GEN4, and CX_PHY_SMODE_GEN5`) to 4s/32-bit per-lane
- Maximum Link Width to 16 lanes (`CX_NL =x16`)

Table 1-1 lists PCIe controller configurations for CXL controller.

Table 1-2 provides a list of supported CXL controller and PHY combinations.

Table 1-1 PCIe Configurations Supported for the CXL Controller

core_clk @ Gen{1,2,3,4,5} speed (in MHz)		Gen1 Speed Mode	Gen2 Speed Mode	Gen3 Speed Mode	Gen4 Speed Mode	Gen5 Speed Mode	Max Link Width				
							32-bit	64-bit	128-bit	256-bit	512-bit
Gen5 Configuration	62.5, 125, 250, 500, 1000	g1_4s	g2_4s_df	g3_4s_df	g4_4s_df	g5_4s_df	N/A	N/A	x4	x8	x16
	125, 250, 250, 500, 1000	g1_2s	g2_2s_df	g3_4s_dw	g4_4s_df	g5_4s_df	N/A	N/A	x4	x8	x16
	62.5, 125, 125, 250, 500	g1_4s	g2_4s_df	g3_8s_dw	g4_8s_df	g5_8s_df	N/A	N/A	N/A	x4	x8

Table 1-2 CXL Controller-PHY Combinations Supported

	PHY		
CONTROLLER	**g1_2s g2_2s_df g3_4s_dw g4_4s_df g5_4s_df**	**g1_4s_df g2_4s_df g3_4s_df g4_4s_df g5_4s_df**	**g1_4s g2_4s_df g3_8s_dw g4_8s_df g5_8s_df**
g1_2s g2_2s_df g3_4s_dw g4_4s_df g5_4s_df	Yes	No	No
g1_4s g2_4s_df g3_4s_df g4_4s_df g5_4s_df	No	Yes	No
g1_4s g2_4s_df g3_8s_dw g4_8s_df g5_8s_df	Yes	Yes	Yes

1.4.1 Synthesis for 1 GHz/2 GHz Frequency

The CXL controller operates at a frequency of 1 GHz with options to run the controller interfaces at 2 GHz for reduced latency.

Timing closure at 1 GHz/2 GHz frequency requires a recent technology node with high-speed cells, a significant percentage of low and ultra-low VT cells (LVT, ULVT). Timing closure with extensive feature sets and minimum latency may require overdrive voltages depending on the specific technology and library in use.

Timing closure at 2 GHz frequency requires the usage of fast RAMs or custom banks of registers.

1.5 Deliverables

- Synthesizable RTL source code
- Synopsys coreConsultant tool for automated configuration, synthesis, and simulation
- Synopsys PHY simulation model
- A Verilog Testbench (VTB) for regression of your controller configuration

1.6 Standards

The PCIe controller implements the following standards:

- *Compute Express Link Specification, Revision 3.0, Version 0.7*
  Download site:
  https://www.computeexpresslink.org/download-the-specification
- *PCI Express Base Specification, Revision 6.0, Version 1.0*
  Access to this specification requires membership in PCI-SIG.
  Download site:
  http://pcisig.com/specifications
- *DesignWare Cores, PCI Express DM Controller Databook, Version 5.80a*

2

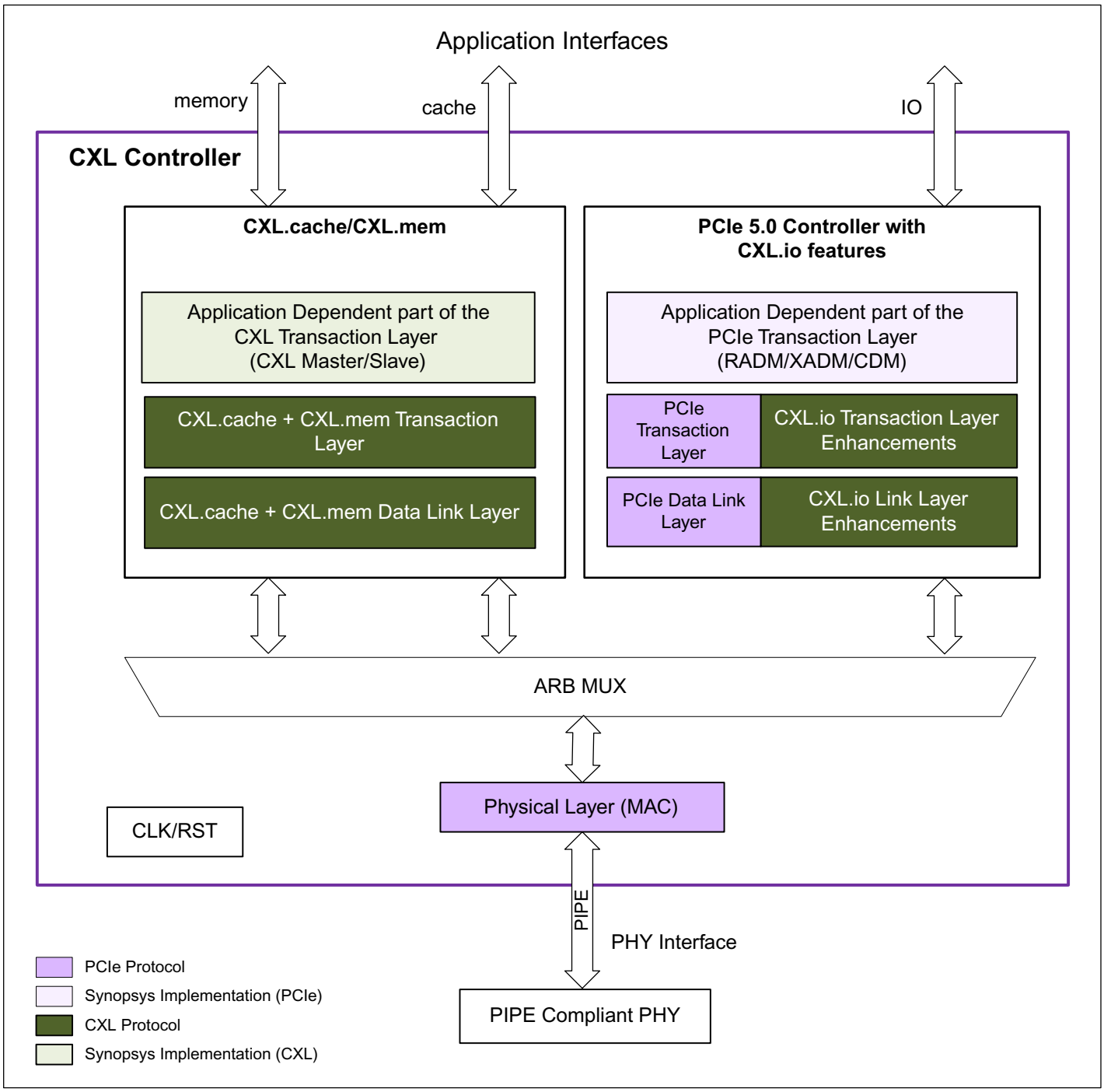
Architecture

This chapter describes the architecture of the DesignWare Cores Compute Express Link (CXL) controller. The following topics are discussed:

2.1 Architecture Overview

The *Compute Express Link Specification, Revision 3.0, Version 0.7* defines CXL.io, CXL.cache, and CXL.mem protocols that are dynamically multiplexed together before being transported through a standard PCIe 5.0 PHY at 32 GT/s.

Figure 2-1 CXL Architecture Overview



- **CXL.cache/CXL.mem Link and Transaction Layer**

- Link Layer implements Flit Packing/Unpacking, integrity checking, ACK protocol and retry.
- Transaction Layer implements Tx/Rx Channel interfaces for each message channel defined by the CXL protocol, and Flow control mechanism.

- **CXL.io Link and Transaction Layer**

  The PCIe link and transaction layer is enhanced to handle the CXL.io protocol messages.

- **ARB/MUX**

  The ARB/MUX dynamically multiplexes the CXL.io and CXL.cache/CXL.mem inbound and outbound traffic. It interfaces with the PCIe physical layer. For outbound traffic, the ARB/MUX arbitrates between requests from the CXL.io and CXL.cache/CXL.mem link layers and multiplexes the data to forward to the physical layer. For inbound traffic, the ARB/MUX determines the protocol ID and forwards the packets to the appropriate link layer. It implements a vLSM for CXL.io and CXL.cache/CXL.mem, which negotiates the virtual link state with a link partner vLSM.

- **Physical Layer (MAC)**

  Physical layer is common to CXL.io and CXL.cache/CXL.mem. It negotiates with the physical link.

Figure 2-2 CXL Device Block Diagram

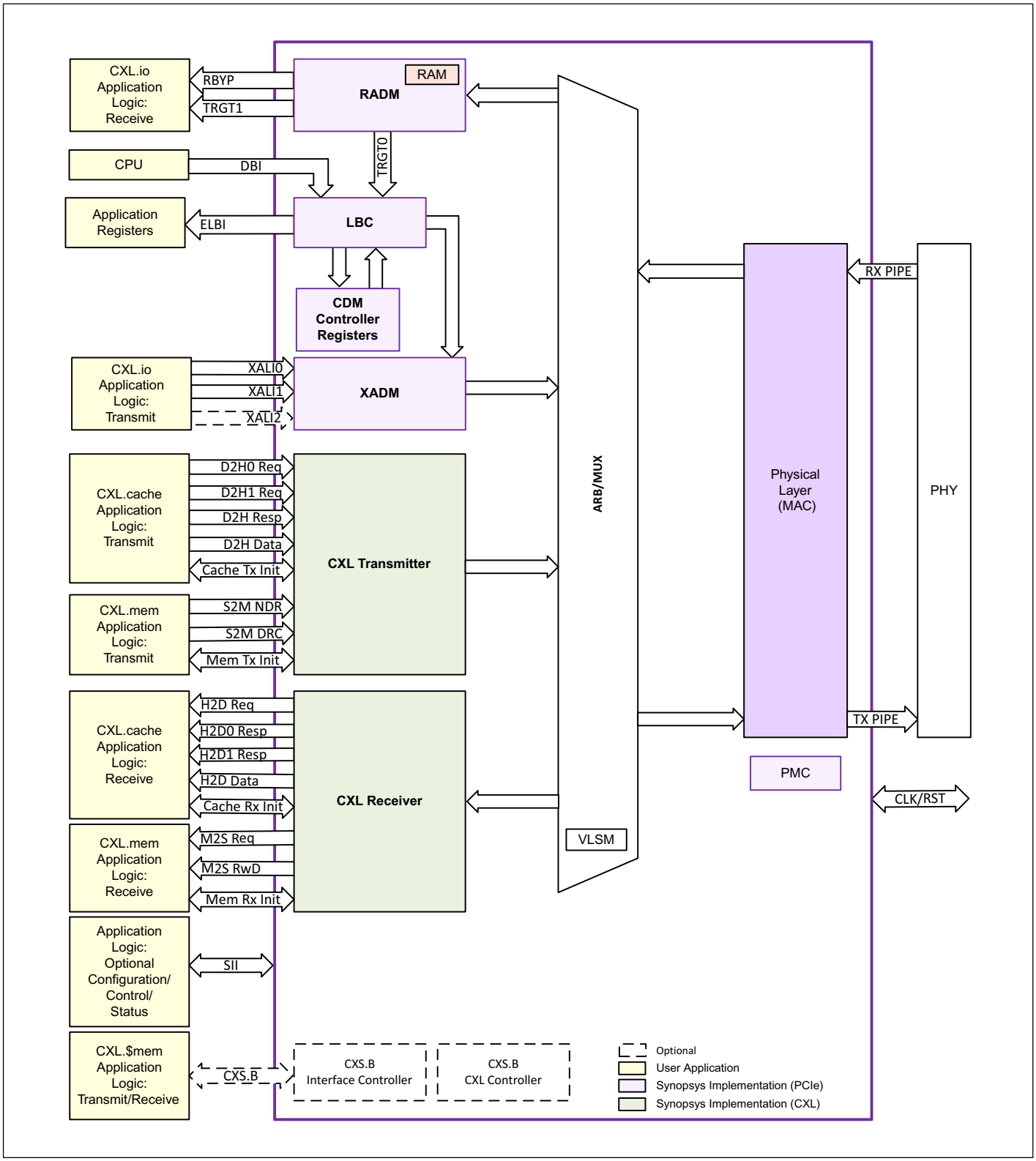


Figure 2-3 CXL Host Block Diagram

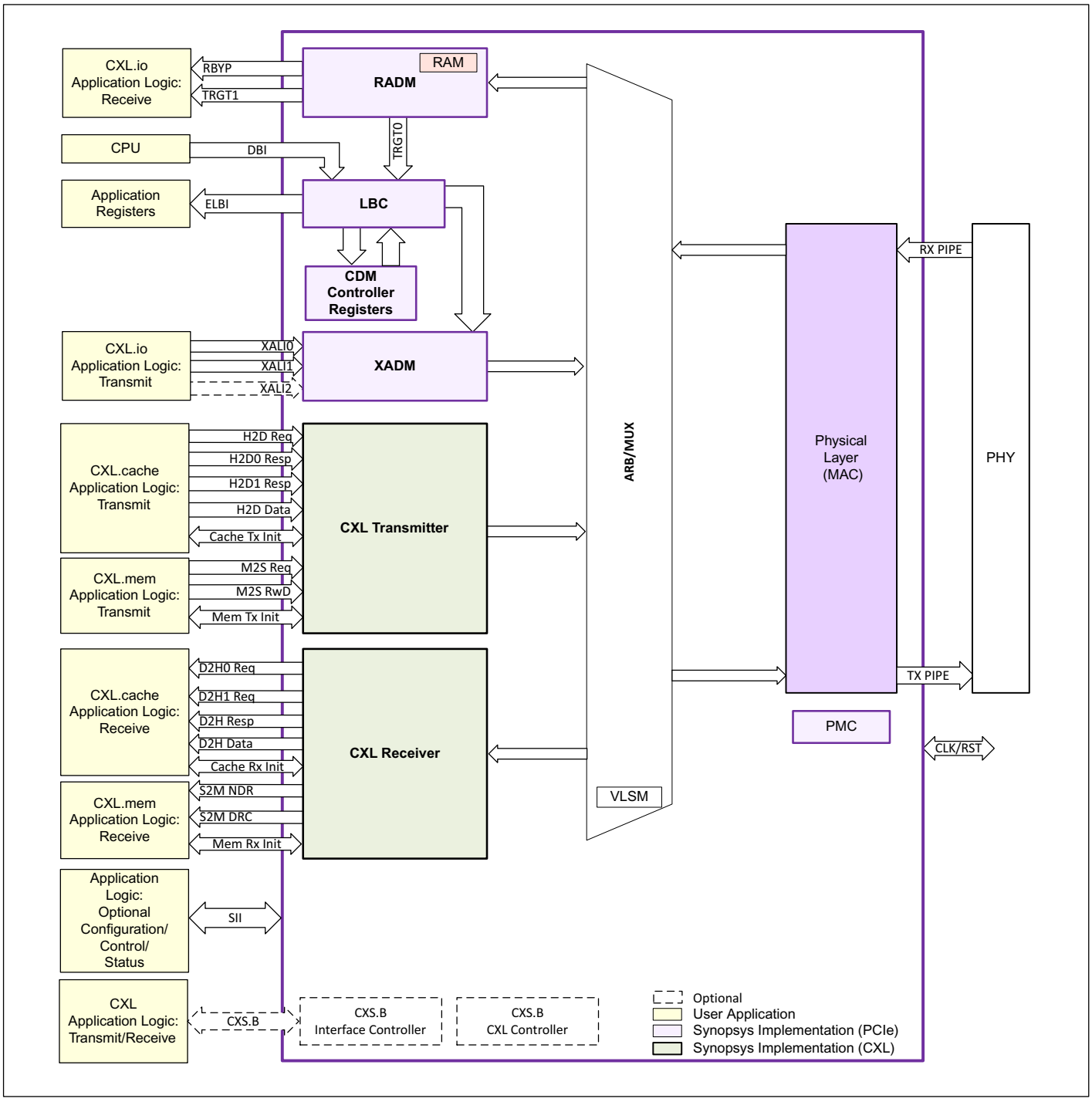


Table 2-1 provides a summary of native CXL controller interfaces. The interfaces to the CXL.io protocol are the same as those used in the PCIe controller. For a complete list of CXL controller interfaces, see the PCIe Controller Databook.

Table 2-1 Native CXL Interface Summary

Interface Type	CXL Device	CXL Host
CXL.cache Interfaces		
D2H0 Req/D2H1 Req Device to Host Request Channel Interfaces	(Tx) D2H request channel carries outbound requests targeting memory attached to the Host. This channel interface comprises of cxl_slv_d2h*_req, cxl_slv_d2h*_req_chk, and cxl_slv_d2h*_req_crd signals.	(Rx) D2H request channel carries inbound requests targeting memory attached to the Host. This channel interface comprises of cxl_mst_d2h*_req, cxl_mst_d2h*_req_chk, and cxl_mst_d2h*_req_crd.
D2H Resp Device to Host Response Channel Interface	(Tx) The D2H response channels carry the response sent by a Device for a H2D request received. This channel interface comprises of cxl_slv_d2h_rsp, cxl_slv_d2h_rsp_chk, and cxl_slv_d2h_rsp_crd signals.	(Rx) The D2H response channels carry the response received by the Host for a H2D request sent. This channel interface comprises of cxl_mst_d2h_rsp, cxl_mst_d2h_rsp_chk, and cxl_mst_d2h_rsp_crd signals
D2H Data Device to Host Data Channel Interface	(Tx) The D2H data channel carries 17-bits D2H Data Header, 64-bits for D2H Byte Enables and 512-bits D2H Data. This channel interface comprises of cxl_slv_d2h_data, cxl_slv_d2h_data_chk, and cxl_slv_d2h_data_crd signals.	(Rx)The D2H data channel carry 17-bits D2H Data Header, 64-bits for D2H Byte Enables and 512-bits D2H Data. This channel interface comprises of cxl_mst_d2h_data, cxl_mst_d2h_data_chk, and cxl_mst_d2h_data_crd signals.
H2D Req Host to Device Request Channel Interface	(Rx) The H2D request channel carries inbound requests from the Host to the Device. Typically, these are snoops to maintain coherency in the Host memory. This channel interface comprises of cxl_mst_h2d_req, cxl_mst_h2d_req_chk, and cxl_mst_h2d_req_crd signals.	(Tx) The H2D request channel carries outbound requests from the Host to the Device. Typically, these are snoops to maintain coherency in the Host memory. This channel interface comprises of cxl_slv_h2d_req, cxl_slv_h2d_req_chk, and cxl_slv_h2d_req_crd signals.
H2D0 Resp/H2D1 Resp Host to Device Response Channel Interfaces	(Rx) The H2D response channel carries inbound response from the Host to the Device. This channel interface comprises of cxl_mst_h2d*_rsp, cxl_mst_h2d*_rsp_chk, and cxl_mst_h2d*_rsp_crd signals.	(Tx) The H2D response channel carries outbound response from the Host to the Device. This channel interface comprises of cxl_slv_h2d*_rsp, cxl_slv_h2d*_rsp_chk, and cxl_slv_h2d*_rsp_crd signals.
H2D Data Host to Device Data Channel Interface	(Rx) The H2D data channel carries 24-bits H2D Data Header, 64-bits H2D Byte Enables and 512-bits H2D Data. This channel interface comprises of cxl_mst_h2d_data, cxl_mst_h2d_data_chk, and cxl_mst_h2d_data_crd signals.	(Tx) The H2D data channel carries 24-bits H2D Data Header, 64-bits H2D Byte Enables and 512-bits H2D Data. This channel interface comprises of cxl_slv_h2d_data, cxl_slv_h2d_data_chk, and cxl_slv_h2d_data_crd signals.
CXL.mem Interfaces		

Interface Type	CXL Device	CXL Host
S2M NDR Subordinate to Master No Data Response Channel Interface	(Tx) The S2M NDR channel carries message response without data. This channel interface comprises of cxl_slv_s2m_ndr, cxl_slv_s2m_ndr_chk, and cxl_slv_s2m_ndr_crd signals.	(Rx) The S2M NDR channel carries message response without data. This channel interface comprises of cxl_mst_s2m_ndr, cxl_mst_s2m_ndr_chk, and cxl_mst_s2m_ndr_crd signals.
S2M DRC Subordinate to Master Data Response Channel Interface	(Tx) The S2M DRC channel carries 64-bits S2M Byte Enables and 512-bits S2M Data. This channel interface comprises of cxl_slv_s2m_drc, cxl_slv_s2m_drc_chk, and cxl_slv_s2m_drc_crd signals.	(Rx) The S2M DRC channel carries 64-bits S2M Byte Enables and 512-bits S2M Data. This channel interface comprises of cxl_mst_s2m_drc, cxl_mst_s2m_drc_chk, and cxl_mst_s2m_drc_crd signals.
M2S Req Master to Subordinate Request Channel Interface	(Rx) The M2S request channel carries requests from Host to Device. This channel interface comprises of cxl_mst_m2s_req, cxl_mst_m2s_req_chk, and cxl_mst_m2s_req_crd signals.	(Tx) The M2S request channel carries requests from Host to Device. This channel interface comprises of cxl_slv_m2s_req, cxl_slv_m2s_req_chk, and cxl_slv_m2s_req_crd signals.
M2S RwD Master to Subordinate Request with Data Channel Interface	(Rx) The M2S RwD channel carries requests with data from Host to Device. The request contains 64-bits M2S Byte Enables and 512-bits M2S data. This channel interface comprises of cxl_mst_m2s_rwd, cxl_mst_m2s_rwd_chk, and cxl_mst_m2s_rwd_crd signals.	(Tx) The M2S RwD channel carries requests with data from Host to Device. The request contains 64-bits M2S Byte Enables and 512-bits M2S Data. This channel interface comprises of cxl_slv_m2s_rwd, cxl_slv_m2s_rwd_chk, and cxl_slv_m2s_rwd_crd signals.
Tx/Rx Initialization/Deinitialization Interfaces		
Cache/Mem Tx Init Interface	To transmit CXL.cache messages on D2H0 Req/D2H1 Req/D2H Resp/D2H Data channel interfaces and CXL.mem messages on S2M NDR/S2M DRC channel interfaces a handshake must be performed between your application's master and CXL controller's slave to synchronize the state of the channel interface. This interface comprises of cxl_<prot>_slv_req and cxl_<prot>_slv_ack signals (<prot> = mem or cache).	To transmit CXL.cache messages on H2D0 Req/H2D1 Req/H2D Resp/H2D Data channel interfaces and CXL.mem messages on M2S NDR/M2S DRC channel interfaces a handshake must be performed between your application's master and CXL controller's slave to synchronize the state of the channel interface. This interface comprises of cxl_<prot>_slv_req and cxl_<prot>_slv_ack signals (<prot> = mem or cache).

Interface Type	CXL Device	CXL Host
Cache/Mem Rx Init Interface	A handshake is performed between CXL controller's master and your application's slave to synchronize the state of the channel interface, so that your application can receive CXL.cache messages on H2D Req/H2D0 Resp/H2D1 Resp/H2D Data channel interfaces and CXL.mem messages on M2S Req/M2S RwD channel interfaces. This interface comprises of cxl_<prot>_mst_req and cxl_<prot>_mst_ack signals (<prot> = mem or cache).	A handshake is performed between CXL controller's master and your application's slave to synchronize the state of the channel interface, so that your application can receive CXL.cache messages on D2H0 Req/D2H1 Req/D2HResp/D2H Data channel interfaces and CXL.mem messages on M2S Req/M2S RwD channel interfaces. This interface comprises of cxl_<prot>_mst_req and cxl_<prot>_mst_ack signals (<prot> = mem or cache).
Configuration/Control/Status Interfaces		
SII: CXL Configuration/Control/Status Signals	This interface is used for exchanging CXL configuration information, CXL.mem memory ranges, cache/mem enable, and so on.	This interface is used for exchanging CXL configuration information, CXL.mem memory ranges, cache/mem enable, and so on.
SII: CXL CTS Configuration/Control/Status Signals	This interface is used for CXL Compliance DVSEC signaling.	This interface is used for CXL Compliance DVSEC signaling.
CXL Link Status Interfaces		
SII: Reset/Status Signals	The signal smlh_cxl_enabled provides CXL PHY Logical Layer LinkUp information.	The signal smlh_cxl_enabled provides CXL PHY Logical Layer LinkUp information.
	The signal vlsm_mc_state provides the CXL.cache/CXL.mem vLSM state.	The signal vlsm_mc_state provides the CXL.cache/CXL.mem vLSM state.
	The signal vlsm_io_state provides the CXL.io vLSM state.	The signal vlsm_io_state provides the CXL.io vLSM state.
CXL Application Viral Interface		
General Core Control Signals	The signal cxl_viral_status provides the CXL Viral Mode status information.	The signal cxl_viral_status provides the CXL Viral Mode status information.
	You can use the input signal cxl_viral_request to trigger Viral Mode	You can use the input signal cxl_viral_request to trigger Viral Mode
CXL RAS Interface		

Interface Type	CXL Device	CXL Host
CXL Controller RAS Interface	You can use cxl_ras_stat_rx_corr signal to set the CXL RAS Correctable Error Status register. You can use cxl_ras_stat_rx_uncorr signal to set the CXL RAS Uncorrectable Error Status register.	You can use cxl_ras_stat_rx_corr signal to set the CXL RAS Correctable Error Status register. You can use cxl_ras_stat_rx_uncorr signal to set the CXL RAS Uncorrectable Error Status register.

2.2 RAM Requirements

This section describes the CXL-specific RAMs required by the CXL controller for retry and receive buffering over and above the RAMs required by a corresponding PCIe controller.

The widths and depths presented in Table 2-2 are for reference only. The exact RAM requirement varies according to the CXL controller configuration.

Table 2-2 RAM Requirements

RAM	Width	Depth	Address	Instances	Clock Domain	RAM Type	Instance Name
CXL Transmitter							
CXL Transmitter H2D/D2H Data RAM Interface	536–592	6	3	1	cxl_clk	2p1c	u_cxl_slv_hd_data_ram
CXL Transmitter M2S/S2M Data RAM Interface	663–552	5	3	1	cxl_clk	2p1c	u_cxl_slv_ms_data_ram
CXL Receiver							
CXL Receiver H2D Request/D2H Response RAM Interface	64–20	256	8	2	core_clk	2p1c	u_cxl_mst_hreq_drsp_ram
CXL Receiver H2D Response/D2H Request RAM Interface	32–79	128	7	4	core_clk	2p1c	u_cxl_mst_hrsp_dreq_ram
CXL Receiver H2D/D2H Data RAM Interface	536–592	512	9	1	core_clk	2p1c	u_cxl_mst_hd_data_ram
CXL Receiver M2S/S2M NDR RAM Interface	87–28	256	8	2	core_clk	2p1c	u_cxl_mst_ms_ndr_ram
CXL Receiver M2S/S2M Data RAM Interface	663–552	512	9	1	core_clk	2p1c	u_cxl_mst_ms_data_ram
Miscellaneous							
CXL Retry Buffer RAM Instance	528	256	8	1	core_clk	1p1c	u_ram_1p_cxl_rbuf

2.3 Clock Requirements

The CXL controller has the same clock requirements as the PCIe controller. In addition to that, the CXL controller provides `cxl_clk` input. To increase the CXL channel interface bandwidth, you can drive `cxl_clk` at double the frequency of the `core_clk` using the PIPE PCLK Mode parameter.

The value of the PIPE PCLK Mode parameter (`CX_PIPE_PCLK_MODE`) governs the `cxl_clk` to the `core_clk` ratio. When `CX_PIPE_PCLK_MODE =3`, the `cxl_clk` to `core_clk` ratio is 2:1, otherwise the `cxl_clk` to `core_clk` ratio is 1:1.

2.4 Reset Requirements

The CXL controller has the same reset requirements as the PCIe controller.

2.5 Receive Queues

When setting the receive queue for posted TLPs in bypass mode, the application logic must loop any posted access to memory-mapped internal registers back through DBI. The application logic must also handle the ordering rules of transactions, as non-posted TLPs can still access internal registers through the TRGT0 interface.

3

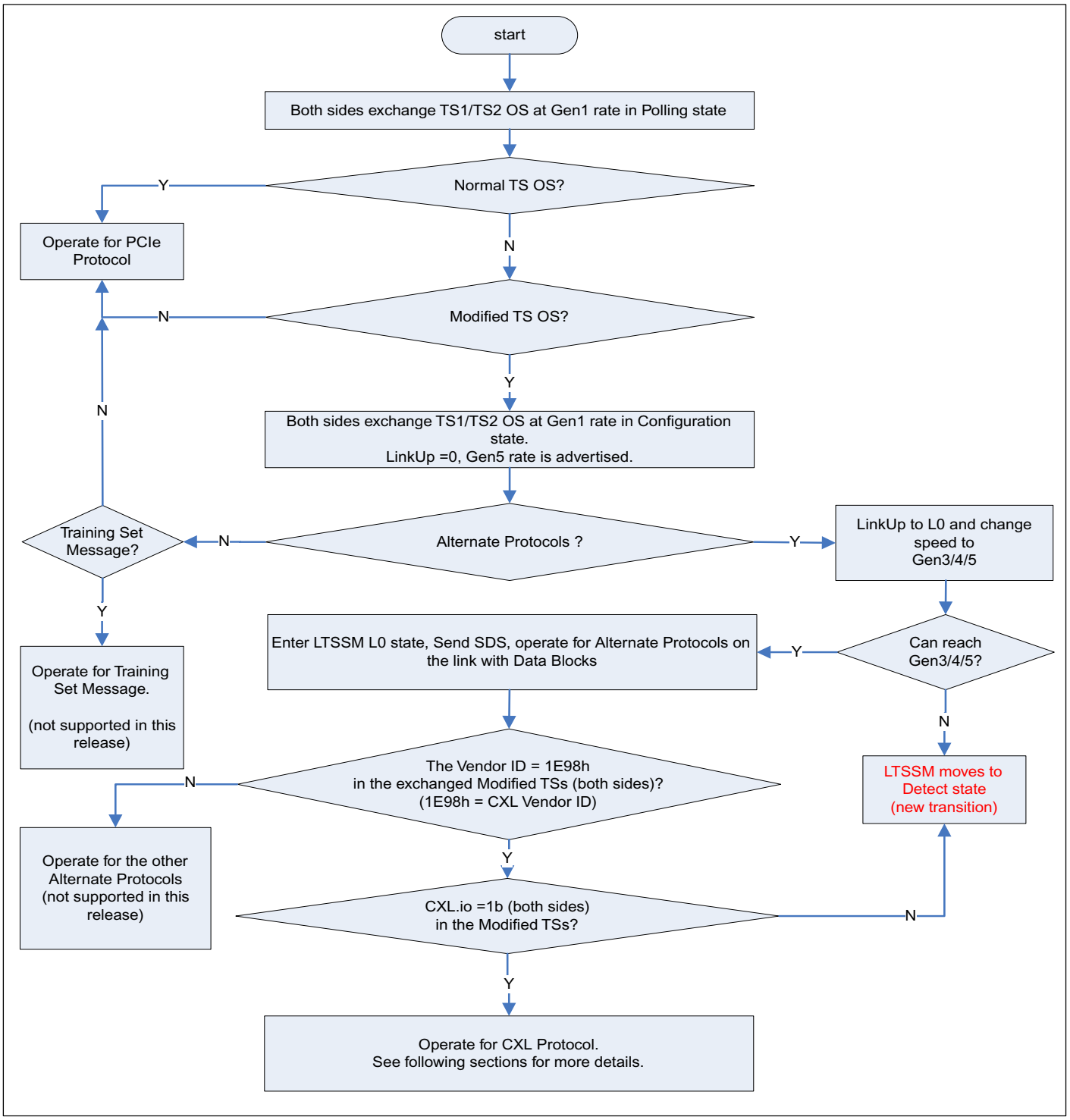
Controller Operations

This chapter describes the operations of the DesignWare Cores Compute Express Link (CXL) controller. The following topics are discussed:

3.1 Alternate Protocol Negotiation

The CXL controller autonomously negotiates CXL by sending Modified TS Ordered Sets advertising Alternate Protocol with the CXL Vendor ID during configuration.

Figure 3-1 CXL Link Training and Operation Flowchart



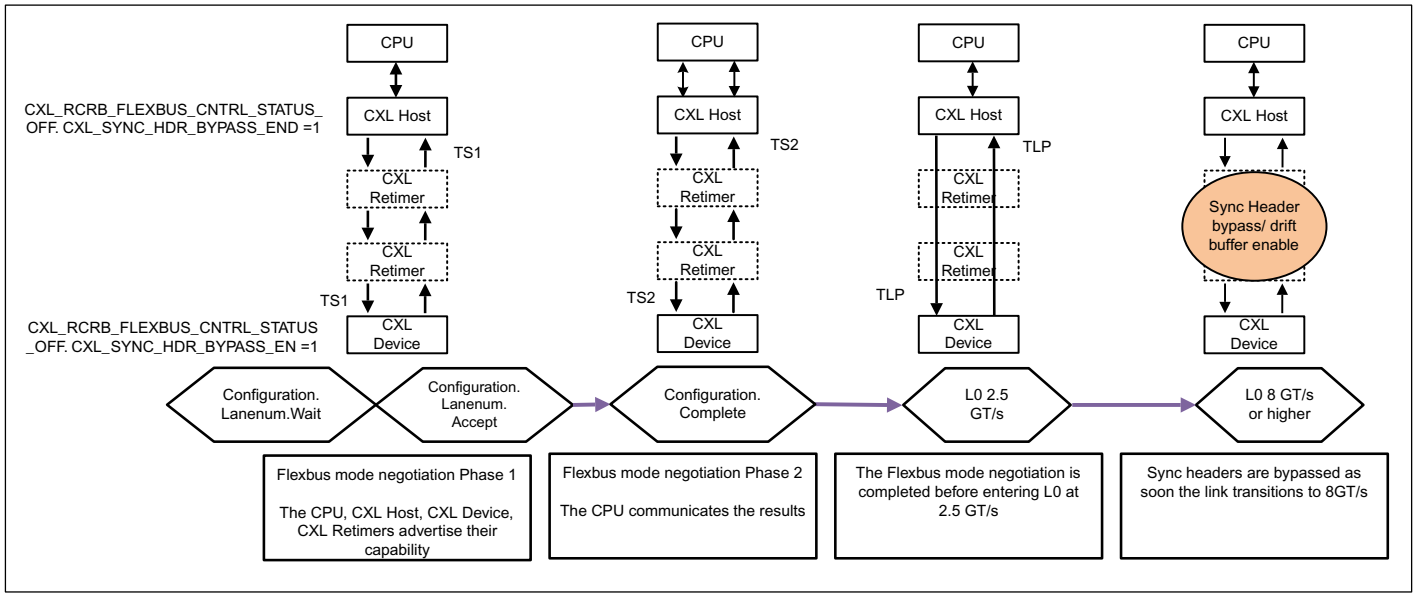
3.2 CXL Physical Layer Low Latency Features

The CXL controller supports the following CXL low latency features to optimize latency according to the section, “Retimers and Low Latency Mode” of the *Compute Express Link Specification, Revision 3.0, Version 0.7*.

- Drift Buffer
- Sync Header Bypass
- Lane-to-lane de-skew in Drift Buffer

Enable `CX_DRIFT_BUFFER_SUPPORT` to implement Drift Buffer feature.

Figure 3-2 Drift Buffer and Sync Header Bypass Negotiation



3.2.1 Sync Header Bypass Latency Considerations

The CXL controller sets the synthesis constraints around the PIPE interface based on the assumption that PHY inputs are pipelined in the PHY, bifurcation multiplexing logic, and buffering are minimal (20% of a cycle).

If your bifurcation multiplexing logic cannot register between the CXL controller and PHY, or if it is necessary to insert many buffering stages between the controller and the PHY on the PIPE interface, you can enable the pipeline stage on the PIPE interface by setting the parameter `CX_SERDES_ARCH_RX_SH_BYPASS_REGIN =1`.

When `CX_SERDES_ARCH_RX_SH_BYPASS_REGIN =1`, the CXL controller inserts a pipeline at the `phy_mac_rxdata` and `phy_mac_rxvalid` signals in the receive path, providing a trade-off of latency and gates for ease of timing closure. The latency in the receive path increases by 1 ns.

The controller does not support Sync Header Bypass, if you enable SRIS Mode `(app_sris_mode =1b).`

3.3 CXL ARB/MUX and vLSMs

ARB/MUX multiplexes and De-multiplexes CXL.io, CXL.cache/mem, and CXL ARB/MUX Link Management Packets (ALMPs).

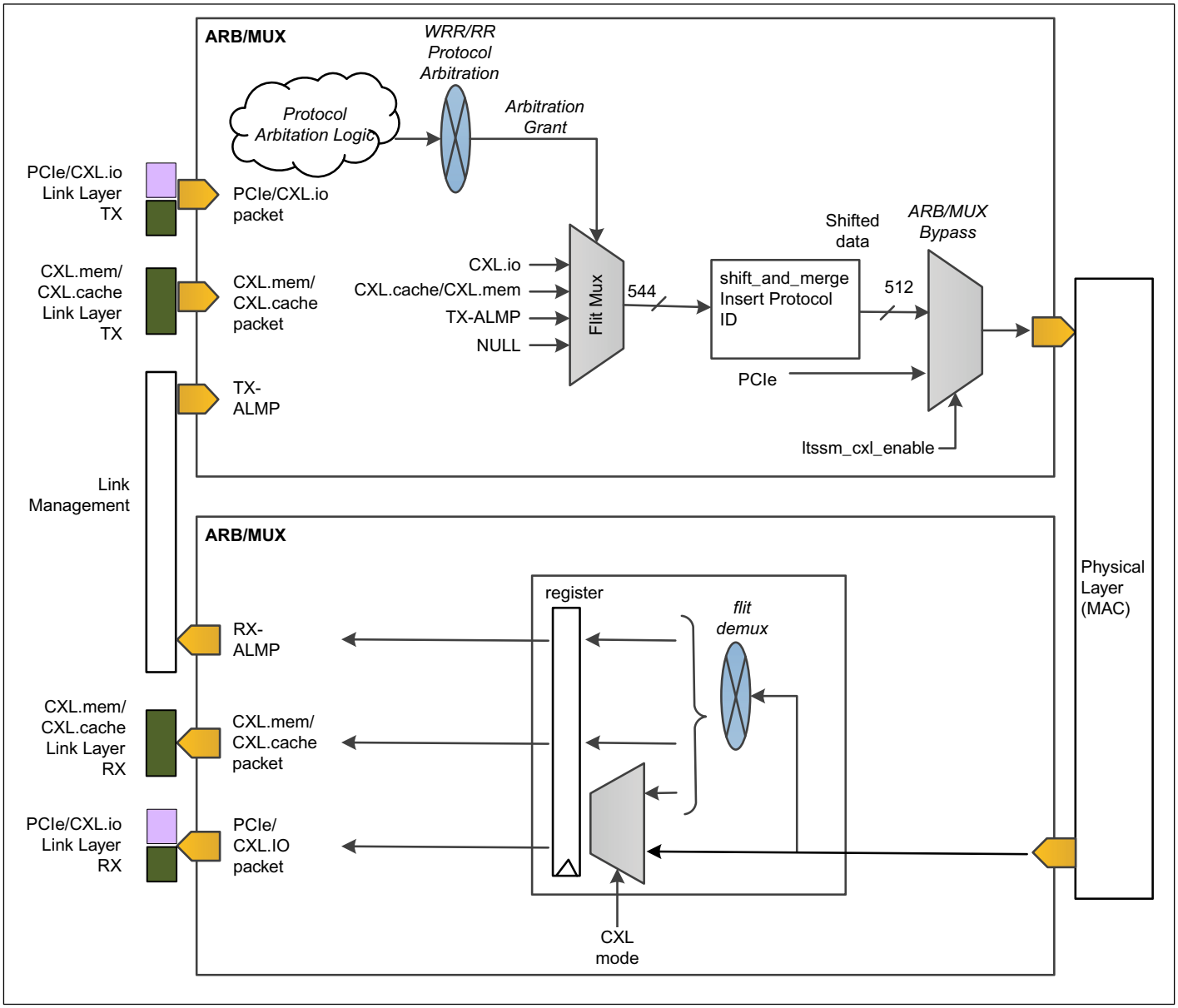
3.3.1 Transmit and Receive Protocol Multiplexing

For outbound packets, the ARB/MUX arbitrates between requests from the CXL link layers and multiplexes the data based on the arbitration results. The protocol arbitration implements a weighted round robin arbitration scheme. You can set the weight for CXL.io protocol using `CXL_RCRB_MEMBAR0_ARB_MUX_CXL_IO_RND_ROBIN` field of the `MEMBAR0_ARB_MUX_CXL_IO_REG_OFF` register, and the weight for CXL.cache/CXL.mem protocol using `CXL_RCRB_MEMBAR0_ARB_MUX_CXL_CACHE_MEM_RND_ROBIN` field of the `MEMBAR0_ARB_MUX_CXL_CACHE_MEM_REG_OFF` register. If the CXL.mem/CXL.cache link layer is performing a retry sequence the arbitration switches to a simple round robin scheme to allow a retry flit to be transmitted in every other clock cycle. The result of the arbitration is passed to the shift and merge block, where the protocol ID is inserted before transferring 512-bits of data to the physical layer for transmission.

The ARB/MUX disables generation of ALMPs when there is no dynamic multiplexing of CXL.io with other CXL protocols, that is, when the controller is operating in PCIe mode or when only CXL.io protocol is enabled. The bypass condition is determined during hwinit or during link training.

For inbound packets, ARB/MUX decodes the protocol ID of the received packet and forwards the packet to the appropriate link layer. The ALMP packets used by the link management protocol, they are not forwarded to the link layer.

Figure 3-3 Transmit and Receive Protocol Multiplexing



3.3.2 Virtual Link State Machine (vLSM)

The ARB/MUX maintains vLSM state for each CXL link layer it interfaces with – CXL.io and CXL.$mem. It receives requests from Power Management Controller (PMC), local link layer, and the remote ARB/MUX on behalf of a remote link layer to resolve a single state request to forward to the physical layer.

vLSM comprises of the following components:

- CXL.io Link vLSM
- CXL.io Link vLSM ALMP Generator
- CXL.$mem Link vLSM
- CXL.$mem Link vLSM ALMP Generator

- VLSM Resolution Table

The VLSM resolution table looks at the status of each vLSM to resolve to a single state request as specified in Table 3-1.

Table 3-1 ARB/MUX vLSM Resolution Table

vLSM.io/vLSM.$mem	RESET	ACTIVE	L1	L2
RESET	RESET	ACTIVE	L1	L2
ACTIVE	ACTIVE	ACTIVE	ACTIVE	ACTIVE
L1	L1	ACTIVE	L1	L1
L2	L2	ACTIVE	L1	L2

Most of the vLSM state transitions discussed in the following sections, require state synchronization with remote ARB/MUX by initiating certain ALMP protocol. Each vLSM requests associated ALMP generator to perform certain handshake to complete a state transition. ALMP generators support the following handshakes:

- **ALMP TX Request-Status:** Send State Request ALMP and wait for State Status ALMP. Performed on transitions from Reset to Active or from Active to L1/L2.
- **ALMP RX Request-Status:** If State Request ALMP is received, respond with a current vLSM State Status ALMP. Performed on transitions from Reset to Active or from Active to L1/L2.
- **ALMP Status Exchange:** Synchronize vLSM state between local and remote ARB/MUX by sending and receiving State ALMP Status. Performed on exit from Recovery.

The CXL controller flags any violation to the expected sequence as an unexpected ALMP which triggers a link recovery. The following ALMP error conditions are considered:

- When in the Synchronization portion, on exit from Retrain, any ALMP other than State Status ALMP is considered as an unexpected ALMP and will trigger recovery.
- When an Active Request ALMP has been sent, any ALMP other than an Active State Status ALMP that is sent in response, is considered as an unexpected ALMP and will trigger recovery.
- Status ALMP is considered an unexpected ALMP and will trigger recovery, if it is received without first sending a Request ALMP or it is received outside of the Synchronization portion on exit from Retrain.

3.3.2.1 Link Activation Flow

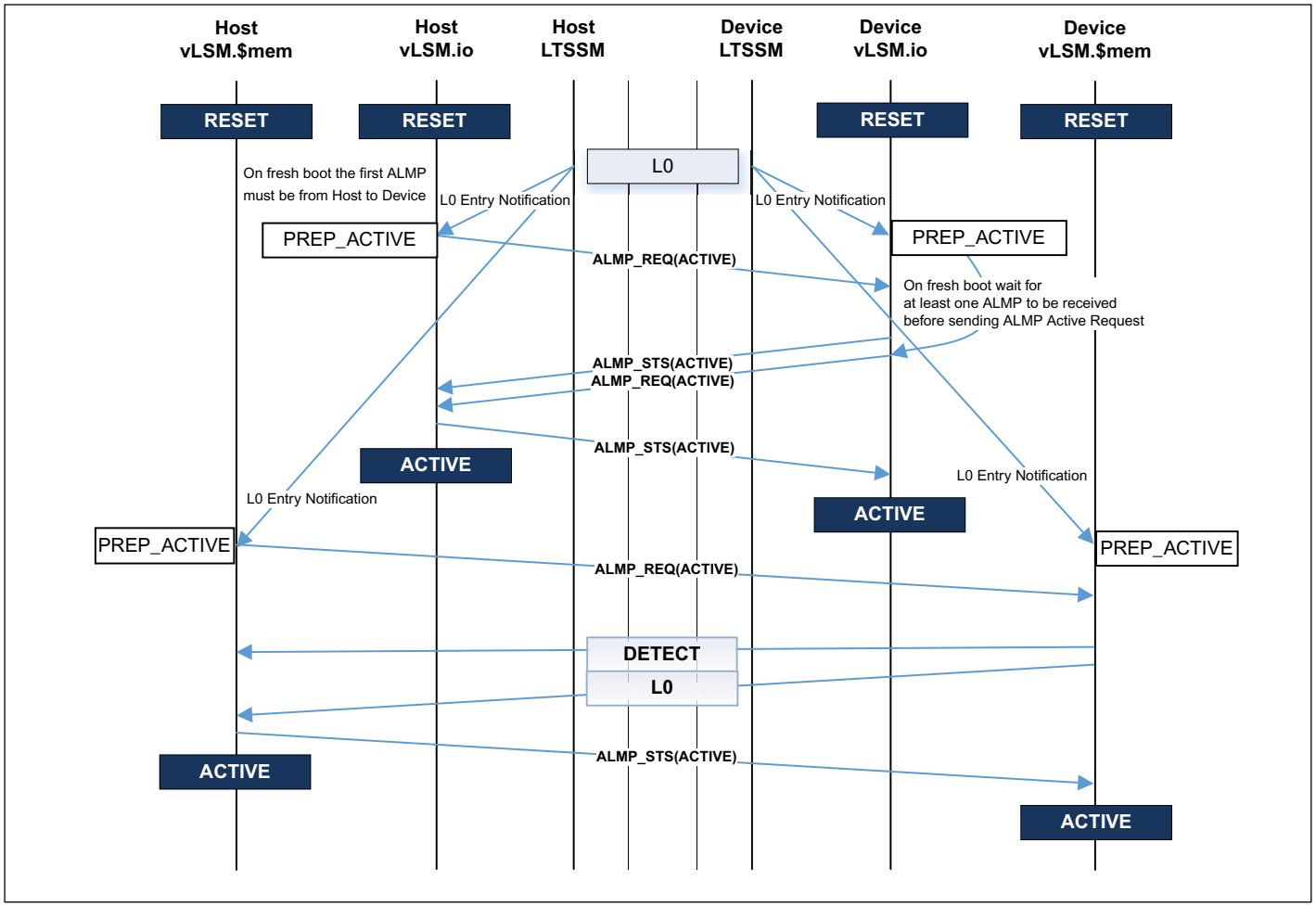
On the fresh boot and after the link reset, CXL controller Link vLSMs are expected to transit from RESET to ACTIVE state. LTSSM transition triggers it to L0 state if the link is negotiated in Alternate Protocol mode and CXL.io and/or CXL.$mem protocols are enabled.

To complete vLSM transition to ACTIVE state, four ALMP exchange protocol is required between the local and remote vLSMs as shown in Figure 3-4. The following ALMP protocol rules must be followed during four ALMP exchange protocol:

- On the fresh boot, first ALMP - Active State Request ALMP must be sent by the Host.
- On the fresh boot, the order of ALMPs with Link vLSM is essential and must be followed.

- Any ALMP other than Active Status ALMP received in response to the Active Request ALMP is considered as unexpected ALMP and will trigger a recovery.

Figure 3-4 RESET to ACTIVE State Transition Example



The following two examples shown in Figure 3-5 and Figure 3-6, cover the corner case which may occur when linking up with a link partner exposing the first entry into recovery on a fresh boot to ARB/MUX Layer. In this case, the link partner must exit from RESET to ACTIVE state through a RETRAIN state, initiating Status Exchange handshake in response to Active Request handshake by CXL controller. In both the cases, when a link partner is Host or Device not hiding a recovery, this conflict is resolved by the rules allowing Host and Device ARB/MUX to sync up by forcing an extra recovery in case of mismatched implementations. For simplicity, all handshakes are shown for only vLSM, however, they same rules are applicable for both CXL.io and CXl.$mem vLSMs.

Figure 3-5 RESET to ACTIVE State Transition Example (Device is Not Hiding Recovery)

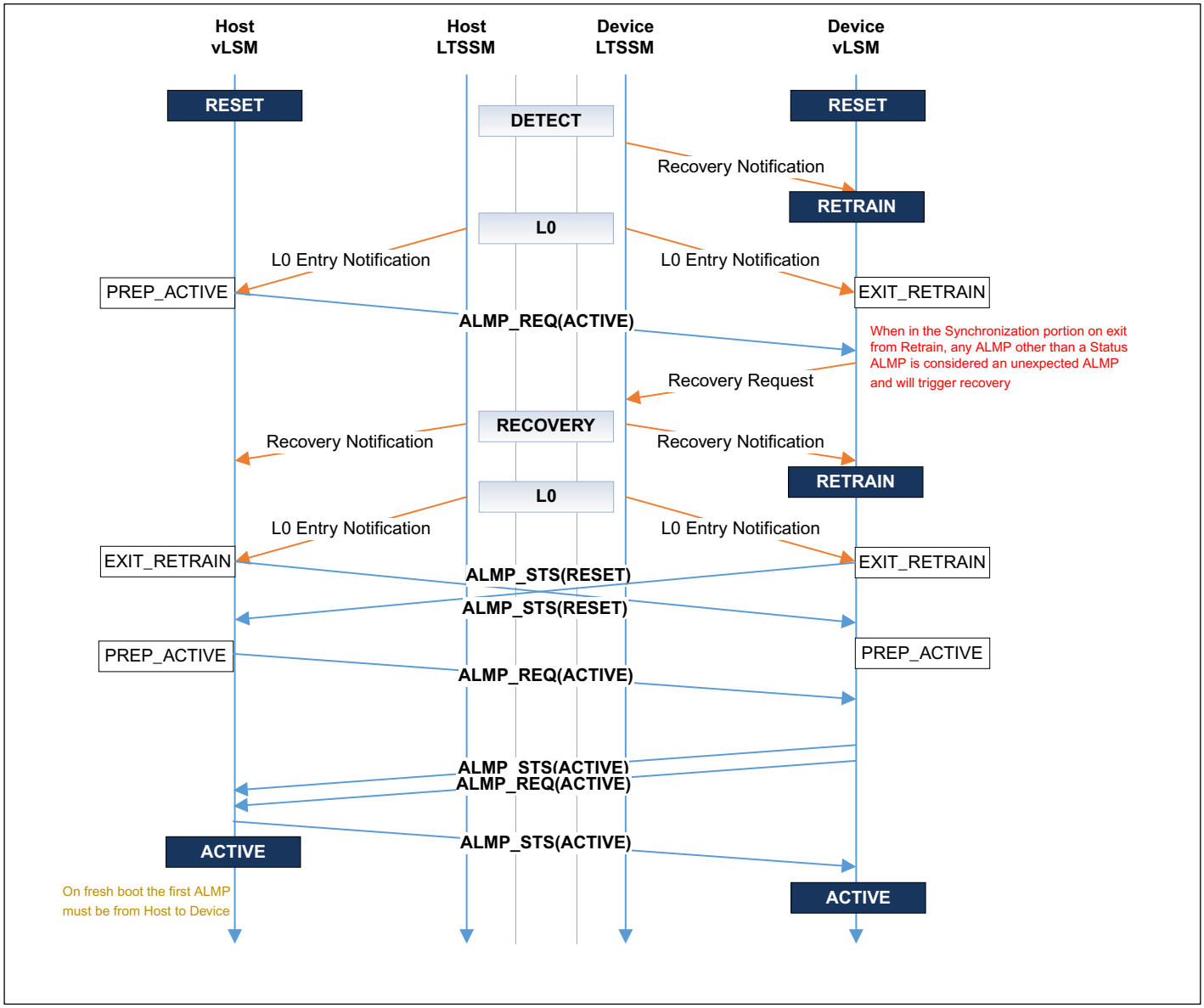
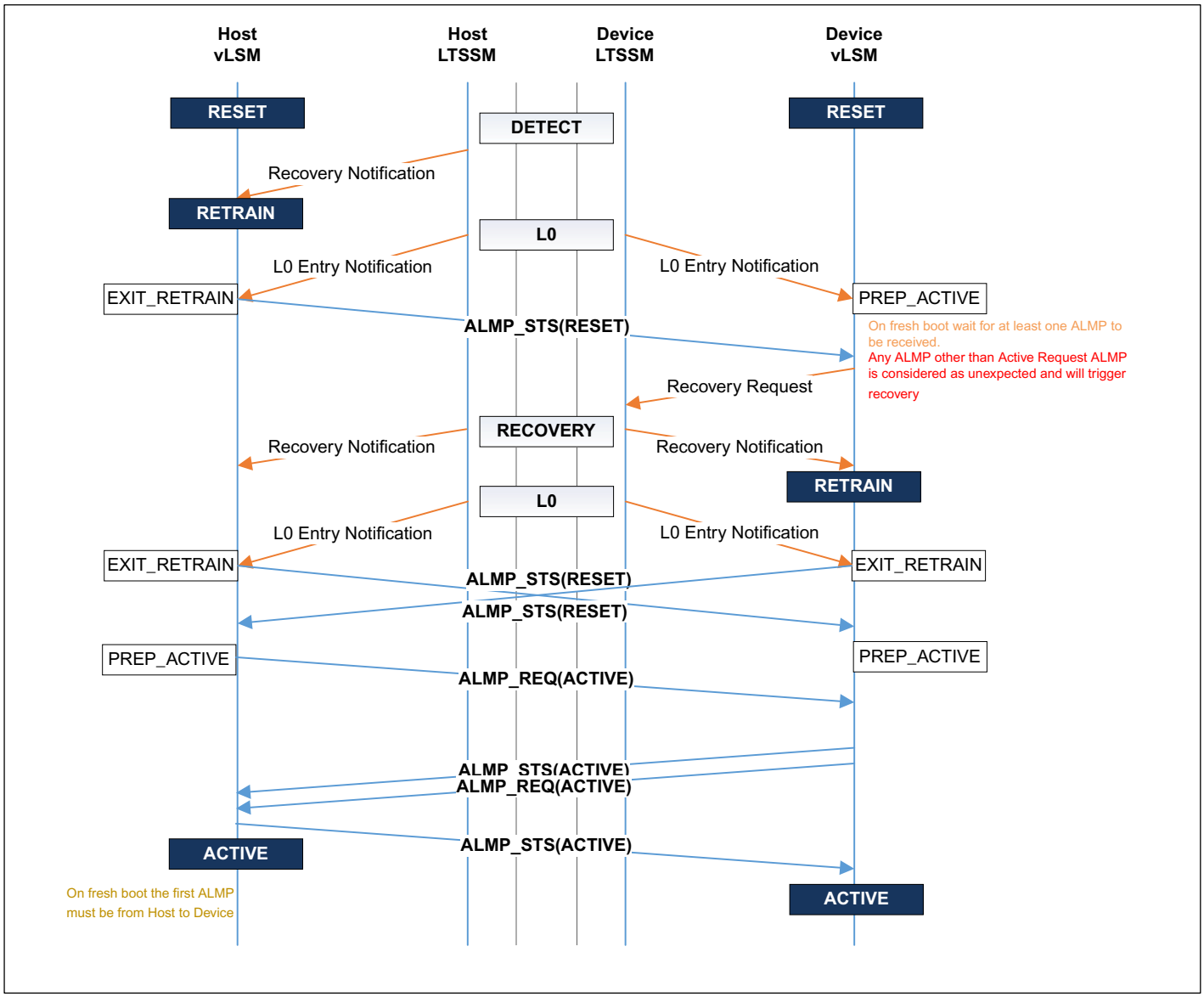


Figure 3-6 RESET to ACTIVE State Transition Example (Host is Not Hiding Recovery)



3.3.2.2 Link Recovery Flow

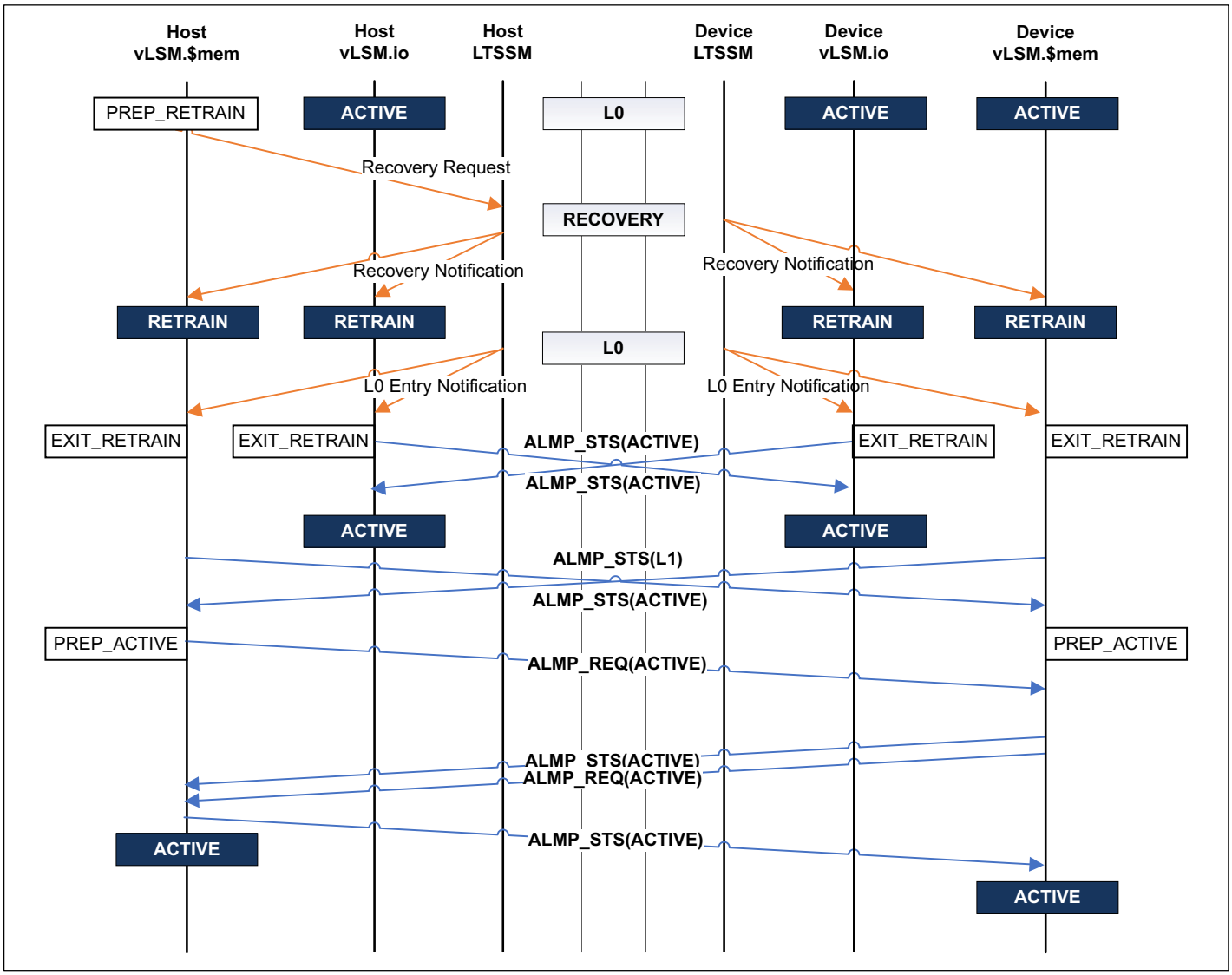
Link vLSM may enter RETRAIN state in the following scenarios:

- On exit from Low Power states;
- In Active State when LRSM is in RETRY_PHY_REINIT state
- When unexpected ALMP is received during ALMP handshake.

All the situations above are resolved by Link vLSM requesting local LTSSM to enter recovery. Once Recovery Notification is observed from LTSSM, Link vLSM waits for Exit Recovery Notification from LTSSM. On exit from recovery, Link vLSM follows link state synchronization protocol by exchanging Status ALMPs.

The Status ALMP provides the state of the vLSM prior to it entering Retrain. The Status ALMPs for synchronization may trigger a State Request ALMP if the provided state and the requested state are not the same, as seen in Figure 3-7 for vLSM.$cache. The ALMP for synchronization may trigger a re-entry to recovery if the vLSMs on either side of the channel are not in the same state out of Retrain, as seen in Figure 3-5 and Figure 3-6. If the provided states from both vLSMs are the same as the requested state prior to the Recovery, the vLSMs are considered synchronized and will continue normal operation, see vLSM.io on Figure 3-7.

Figure 3-7 Example of Link Recovery Flow



3.3.2.3 Link Reset Handling

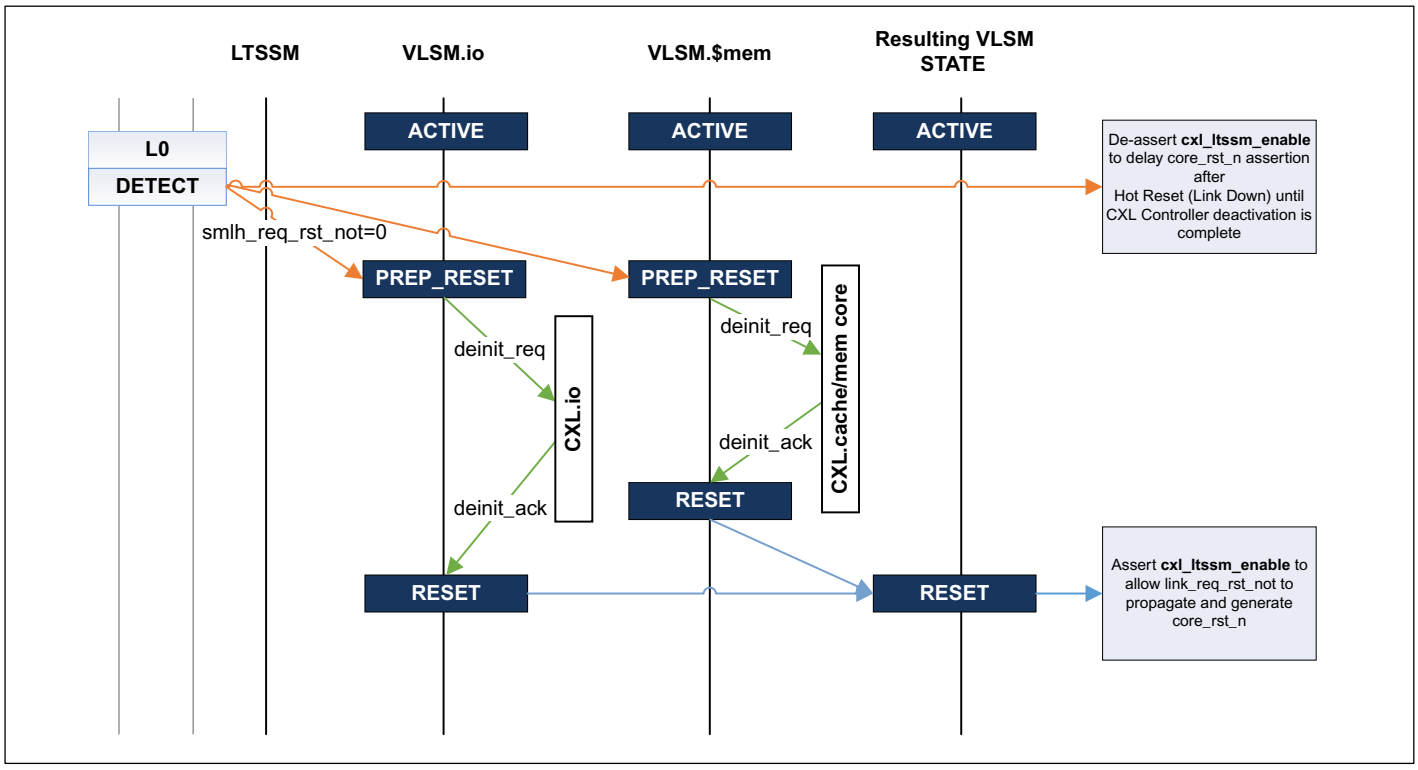
When the Physical Layer enters Hot-Reset or LinkDisable state, that state is communicated to all link layers as LinkReset or LinkDisable. CXL controller ARB/MUX unconditionally moves Link vLSM to

LINK_VLSM_RESET state from any other state when `LTSSM asserts ltssm_link_rst_req` (early version of `core_rst_n`).

vLSM transits through LINK_VLSM_PREP_RESET transitional state to LINK_VLSM_RESET state.Expecting a graceful reset here, the Link vLSM awaits for Link Reset acknowledgment from CXL controller

Once all ongoing transactions are complete and receive buffers are flushed, CXL controller acknowledges the link reset by asserting `cxl_lttsm_enable` and allowing propagation of `ltssm_link_rst_req` to `core_rst_n`.

Figure 3-8 Link Reset Handling



3.3.2.4 Link Down Handling

When link down is detected, Link vLSM performs unconditional transition to RESET state as there is no expectation of a graceful Link Down. A Link Down condition may result in the transactions not making progress from Host to Device and from Device to Host.

3.3.2.5 ARB/MUX Bypass

CXL controller ARB/MUX disables generation of ALMPs, when the link is operating in PCIe mode. Determination of the bypass condition is performed as a result Alternate Protocol negotiation.

3.3.2.6 ARB/MUX Control and Status PortLogic Register

You can control the behavior of ARB/MUX VLSM using the ARB/MUX VLSM Control and Status Register (CXL_VLSM_CSR). The CXL_VLSM_CSR register also provides the vLSM state machine status.

Table 3-2 ARB/MUX VLSM Control and Status Register (CXL_VLSM_CSR)

Bits	Name	Attributes	Reset Value	Description
3:0	VLSM_IO_STATE	RO	0x0	CXL.io Link VSLM State
7:4	VLSM_MC_STATE	RO	0x0	CXL.$mem Link VLSM State
10:8	VLSM_STATE	RO	0x0	VLSM State based on resolution table
11	VLSM_FORCE2ACTIVE	RWS	0x0	Force VLSM to ACTVE State
12	VLSM_LP_ALMP_ENABLE	RWS	0x1	Enables ALMP Handshake on entry to Low Power states
31:13	RSVD	RO	0x0	Reserved

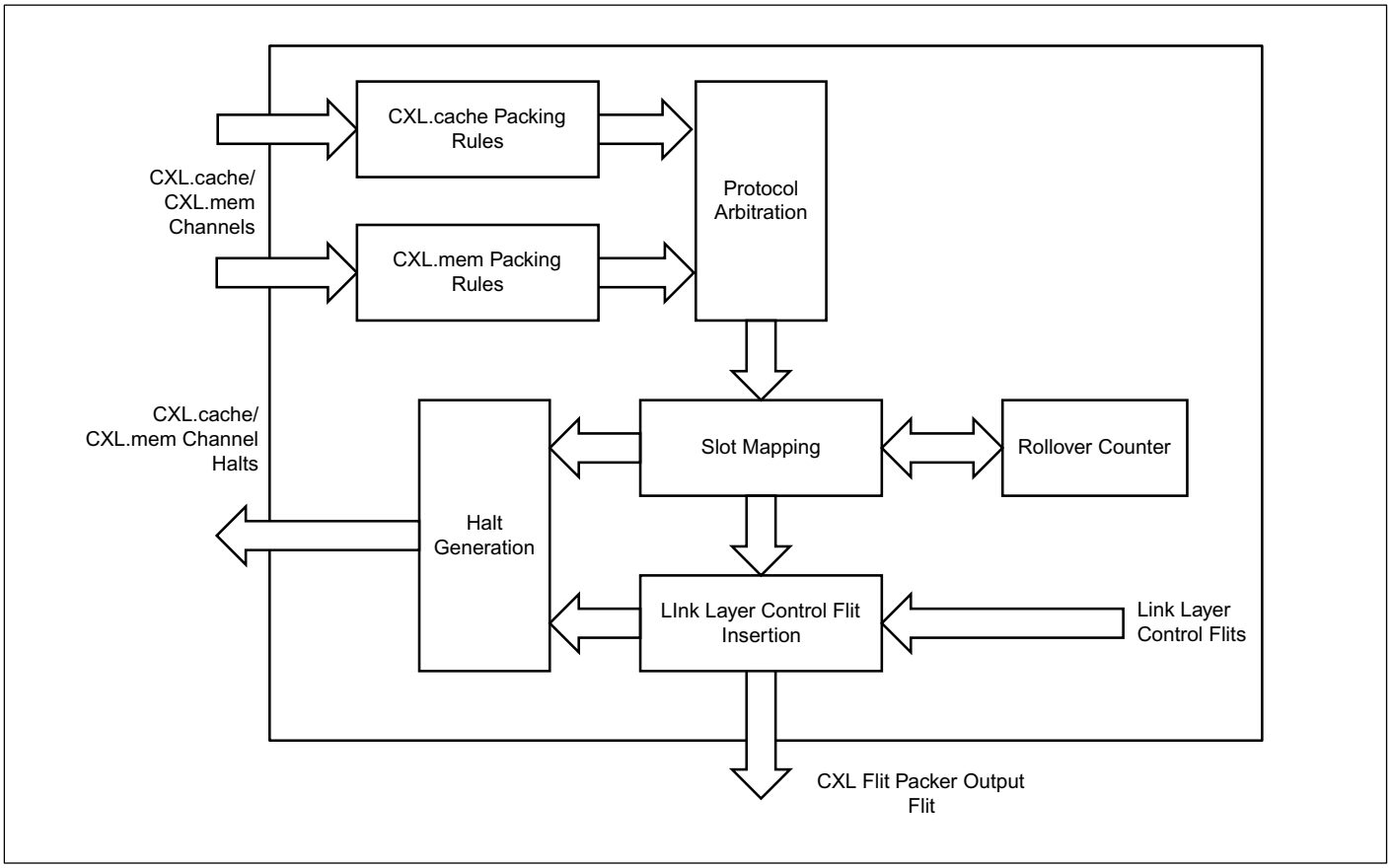
3.4 CXL.cache/CXL.mem Link Layer

The CXL.cache/CXL.mem link and transaction layers implement the flit packing/unpacking, ack and retry protocol when the native CXL interface is used.

3.4.1 Flit Packing and Unpacking

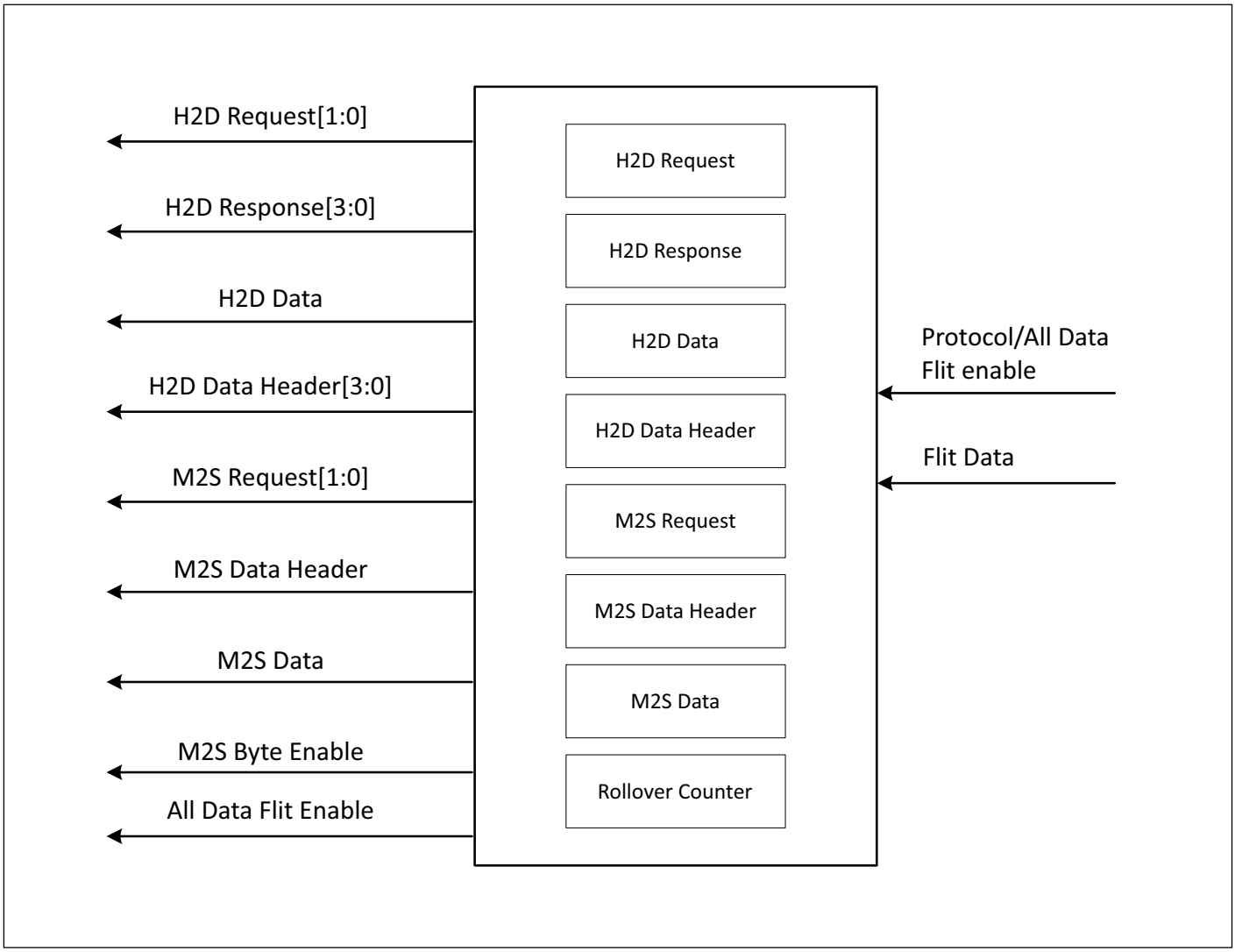
The flit packer implements a simple round-robin arbiter to prioritize CXL.cache or CXL.mem transactions in alternating flit cycles. The flit packer packs the high priority protocol messages in a flit first. The lower priority protocol messages fill the remaining slots available in a flit.

Figure 3-9 CXL.cache/CXL.mem Link Layer Flit Packing



The flit unpacker decodes the flit header, unpacks each message, and transmits the relevant information to the corresponding message channel.

Figure 3-10 CXL.cache/CXL.mem Link Layer Flit Unpacking

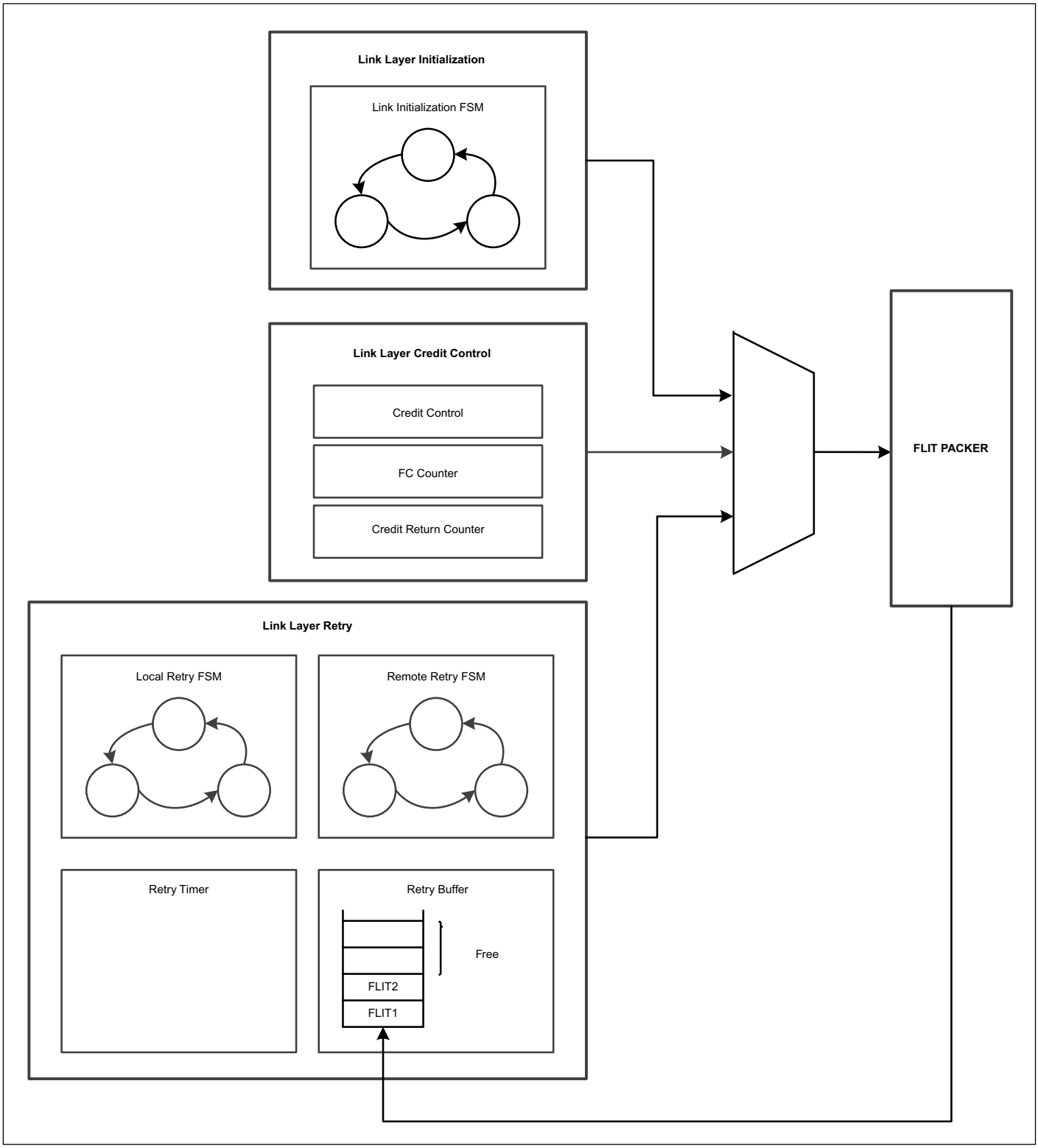


Note The CXL controller does not support the transmission of data at half cache line granularity.

3.4.2 CXL.cache/CXL.mem Link Layer Control

The CXL.cache/CXL.mem Link Layer control is responsible for link initialization and retry.

Figure 3-11 CXL.cache/CXL.mem Link Layer Control



3.4.3 Link Layer Control and Status Register

You can control the operation of the link layer and observe its status using the CXL Link Layer Control (`CXL_LL_CTRL_CR_REG_OFF`) register and CXL LRSM Control and Status (`CXL_LRSM_CSR_REG_OFF`) register.

For more information on these registers, see the *PCIe DM Reference Manual*.

3.5 CXL Transaction Layer

3.5.1 Channel Interface Configuration

CXL Channel interfaces can be configured to meet various latency and data protection requirements.

Table 3-3 CXL Channel Interface Configuration Parameters

Parameter	Values	Description
CX_CXL_ CHI_MAXCREDITS[a]	2–8 (Default: 2)	Maximum number of credits for CXL channel interfaces. This parameter is common for all the channel interfaces.
CX_CXL_CHI_DATACHK	None (0) Parity (1) ECC (2)	Data Check support on Message Channels: ■ Odd parity, 1 bit per byte ■ SECDED: ECC on a 64-bit granularity (zero extended to 64 bits)

a. These credits apply to the Native CXL interface only and should not be confused with CXL link- layer credits.

When assembling a system, ensure that the CXL channel interfaces are compatible in terms of the number of channel credits. For every channel, CXL receiver interface must be able to track the maximum allowed number of interface credits which is 8. The same compatibility requirement is applied to Data Protection as well.

3.5.2 Tx/Rx Initialization/Deinitialization

As shown in Figure 3-12, you can send outbound messages on CXL.cache/CXL.mem channel interfaces only after a transmit initialization between your application and the CXL transmitter. In outbound direction, deinitialization can be initiated either by the CXL transmitter or by your application.

Similarly, as shown in Figure 3-13, you can receive inbound messages on CXL.cache/CXL.mem channel interfaces only after a receive initialization between the CXL receiver and your application. In inbound direction, deinitialization can be initiated either by the CXL receiver or by your application.

This four phase (IDLE, INIT, RUN, DEINIT), request and acknowledge transmit/receive initialization mechanism synchronizes the state of the channel interface between your application and CXL transmitter/receiver.

Figure 3-12 Transmit Initialization (Transmitter Initiated Deinitialization)

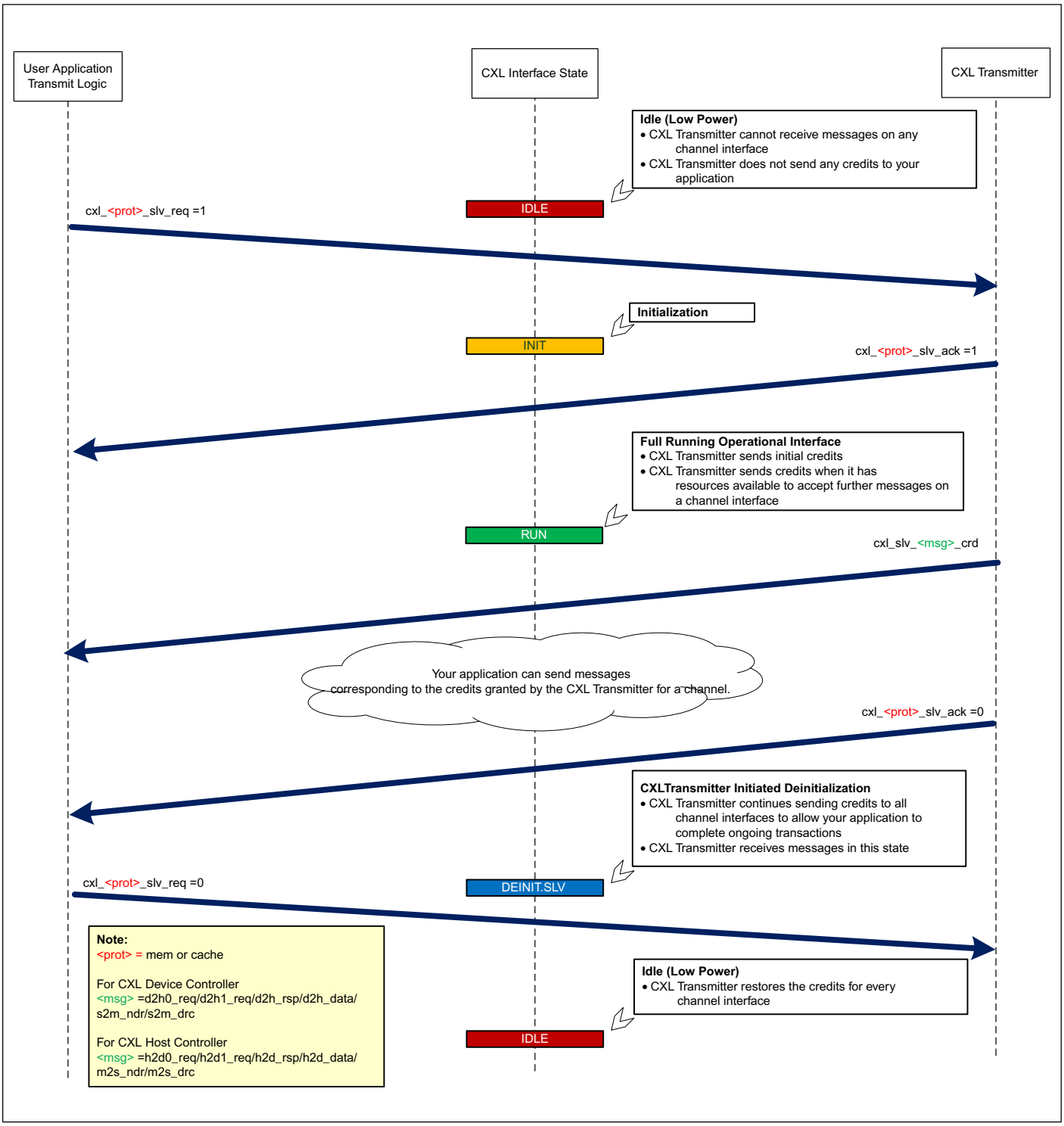
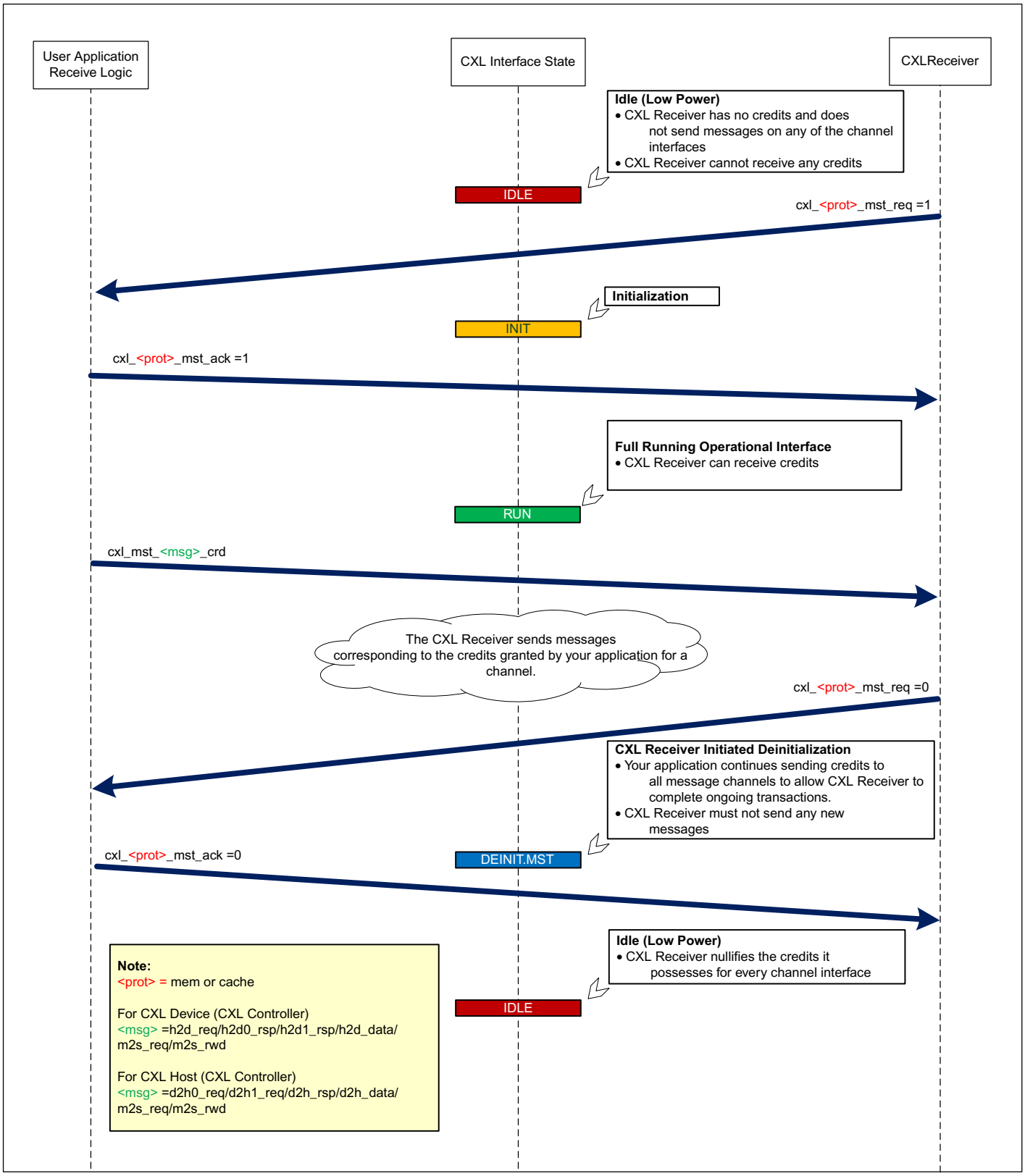


Figure 3-13 Receive Initialization (Receiver Initiated Deinitialization)



3.5.2.1 Tx/Rx Initialization

The CXL controller provides the Tx/Rx initialization mechanism for the CXL channel interfaces of a given protocol to move between a full running operational state and a low power state.

The CXL controller provides a four phase, request and acknowledge handshake mechanism for both CXL.cache and CXL.mem protocols to synchronize the state of the channel interfaces between your application and the CXL Transmitter/Receiver. The request and acknowledge handshake uses `cxl_cache_*_req` and `cxl_cache_*_ack` signals for CXL.cache channel interface initialization, and `cxl_mem_*_req` and `cxl_mem_*_ack` signals for CXL.mem channel interface initialization.

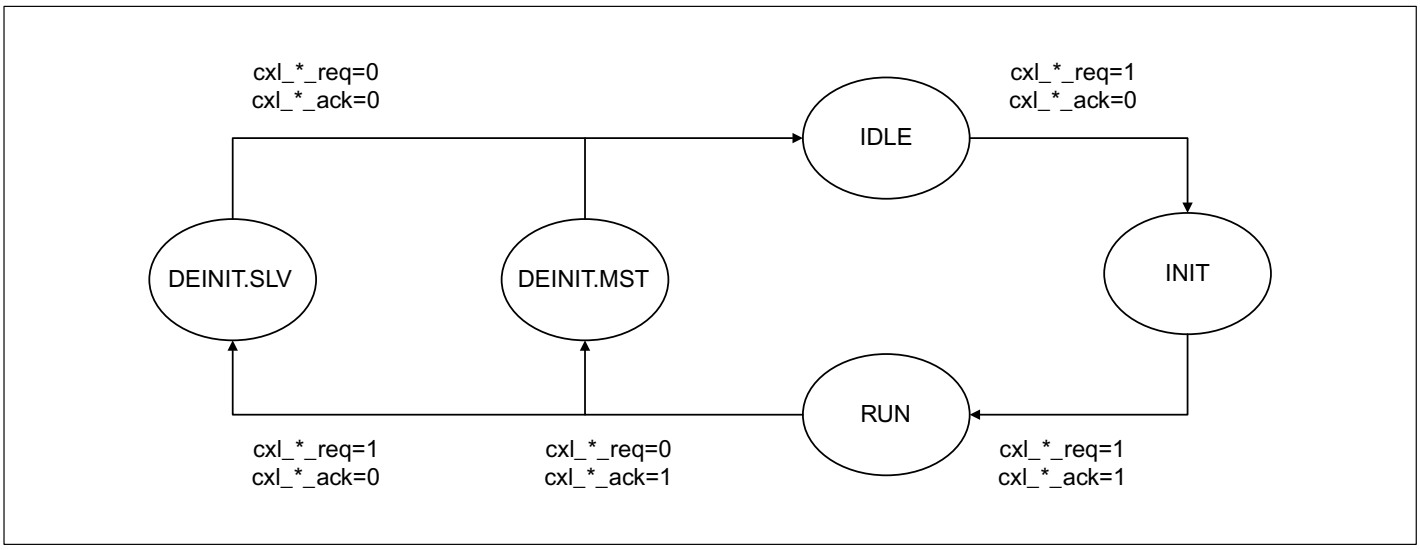
Four states of a channel interface are as follows:

- IDLE: The channel interface is in a low power state and it is not operational. All credits granted on the channel interface are nullified at the CXL Receiver and restored on the CXL Transmitter channel interfaces.
- INIT: This is a transient state. The channel interface is in INIT state when it is moving from the IDLE state to the RUN state.
- RUN: In this state there is an ongoing exchange of messages and credits on channel interfaces.
- DEINIT: This is a transient state. The channel interface is in INIT state when it is moving from the RUN state to the IDLE state.

IDLE and RUN are stable states. A channel interface can remain in one of these states for an indefinite period. INIT and DEINIT are transient states. A channel interface moves to the next stable state in a relatively short period of time after enters INIT or DEINIT state.

The sequence of `cxl_*_req` and `cxl_*_ack` signals determine the state of a channel interface. Figure 3-14 and Table 3-4 shows the relationship between the `cxl_*_req/cxl_*_ack` signals and the channel interface states.

Figure 3-14 CXL Link Activation Request and Acknowledge Handshake[1]



1. When both cxl_*_req =0 as well as cxl_*_ack =0; then, cxl_*_req =0 is at a higher priority.

Table 3-4 CXL Channel Interface State Transition

State	cxl_*_req	cxl_*_ack
IDLE	0	0
INIT	1	0
RUN	1	1
DEINIT.MST[a]	0	1
DEINIT.SLV[b]	1	0

a. If the link deactivation is initiated by the CXL Receiver Interface
b. If the link deactivation is initiated by the CXL Transmitter Interface

On exit from reset or when moving to a full running operational state, the CXL Channel interface is in IDLE state. The transfer of CXL.cache/CXL.mem protocol messages starts only when the CXL transmitter grants credits to your application, or your application grants credits to the CXL receiver for a given channel interface. Credits are granted only after a channel interface is initialized and is in RUN state.

Table 3-5 describes the behavior of CXL Receiver and Transmitter interfaces of a single protocol link for each of the four states.

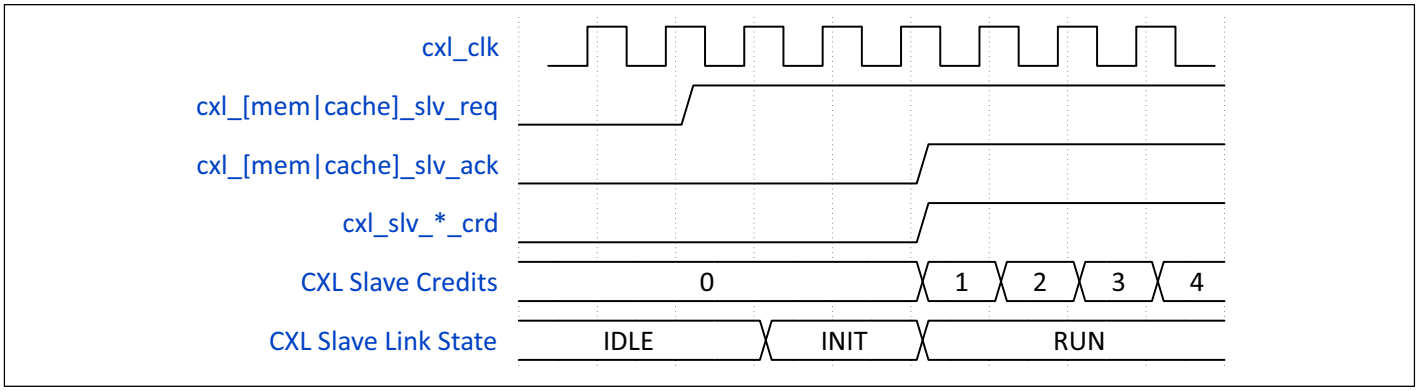
Table 3-5 Behavior for Request and Acknowledge States

State	CXL Receiver	CXL Transmitter
IDLE	■ CXL Receiver has no credits and does not send messages on any of the channel interfaces. ■ CXL Receiver cannot receive any credits. ■ CXL Receiver must assert cxl_<prot>_mst_req to move to the INIT state if it has messages to send.	■ CXL Transmitter cannot receive messages on any channel interface. ■ CXL Transmitter does not send any credits to your application.
INIT	■ CXL Receiver does not send messages on any of the channel interfaces. ■ CXL Receiver remains in the INIT state while it is waiting for your application to acknowledge the link initialization request before moving to the RUN state.	■ CXL Transmitter cannot receive messages on any channel interface. ■ CXL Transmitter does not send credits on any channel interface. ■ The INIT state is a transient state and CXL transmitter controls the move to the RUN state by asserting cxl_<prot>_slv_ack. ■ CXL Transmitter must assert cxl_<prot>_slv_ack and move to the RUN state before sending credits.

State	CXL Receiver	CXL Transmitter
RUN	■ CXL Receiver can receive credits. ■ CXL Receiver can send message on the channel interface when it has credits available for that channel. ■ CXL Receiver must remain in the RUN state until it observes the de-assertion of cxl_<prot>_mst_ack. If cxl_<prot>_mst_ack is de-asserted, CXL Receiver moves to DEINIT.SLV state ■ CXL Receiver may deassert cxl_<prot>_mst_req and move to DEINIT.MST state to request link deinitialization.	■ CXL Transmitter can receive messages corresponding to the credits granted for a channel interface. ■ CXL Transmitter sends credits when it has resources available to accept further messages on a channel interface. ■ CXL Transmitter deasserts cxl_<prot>_slv_ack to exit from this state to DEINIT.SLV state if it wants to move to a low power state. ■ CXL Transmitter must move to the DEINIT.MST state and prepare for link deinitialization when your application deasserts cxl_<prot>_slv_req.
DEINIT.MST	■ CXL Receiver can receive credits. ■ CXL Receiver must not send new messages. ■ CXL Receiver waits for your application to deassert cxl_<prot>_mst_ack before entering the IDLE state. ■ When moving to the IDLE state, the CXL Receiver nullifies the credits it possesses for every channel interface.	■ CXL Transmitter stops sending credits on all channel interfaces. ■ CXL Transmitter receives messages in this state. ■ CXL Transmitter completes all ongoing transactions before entering the IDLE state. ■ CXL Transmitter deasserts cxl_<prot>_slv_ack before entering the IDLE state. ■ When moving to the IDLE state, CXL Transmitter restores the credits for every channel interface.
DEINIT.SLV	■ CXL Receiver can receive credits. ■ CXL Receiver completes ongoing transactions on the channel interfaces and stops sending new messages. Once completed, it must deassert cxl_<prot>_mst_req and move to the IDLE state. ■ When moving to the IDLE state CXL Receiver must nullify all the credits it possesses for every channel interface.	■ In this state the CXL Transmitter continues sending credits to all channel interfaces to allow your application to complete the ongoing transactions. ■ CXL Transmitter receives messages in this state. ■ CXL Transmitter waits for your application to deassert cxl_<prot>_slv_req before entering the IDLE state. ■ When moving to the IDLE state, the CXL Transmitter restores the credits for every channel interface.

Figure 3-15 illustrates an example of the CXL Tx initialization.

Figure 3-15 CXL Tx Initialization Example[1]



1. The example illustrates CXL Tx initialization from CXL Transmitter perspective.

3.5.2.2 Tx/Rx Deinitialization

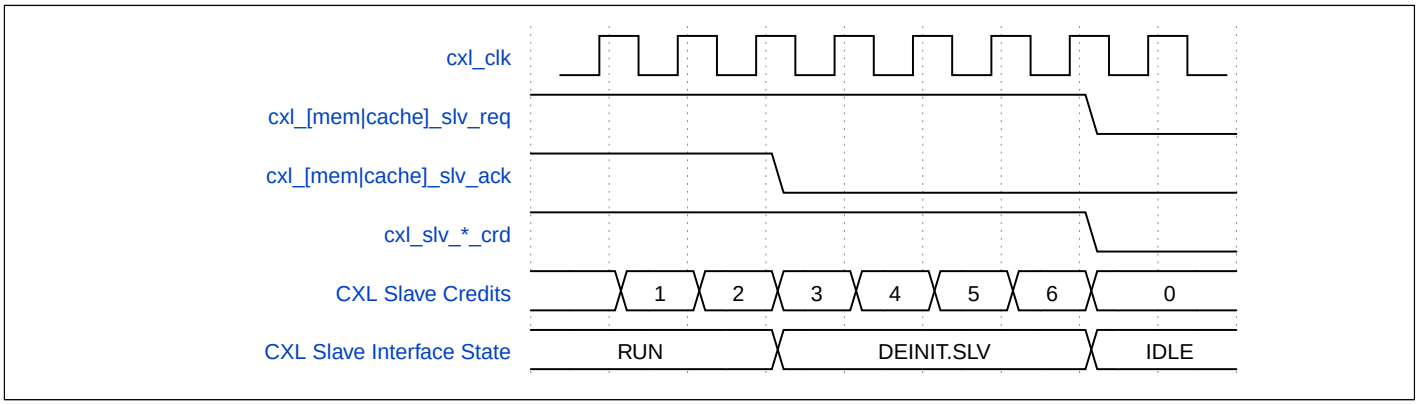
Tx Deinitialization

Tx deinitialization can be initiated by CXL Transmitter or by your application's transmit logic.

- **Tx deinitialization initiated by CXL Transmitter:** The CXL Transmitter requests deinitialization to move a channel interface from full operational state to the low power state. The CXL Transmitter deasserts `cxl_<prot>_slv_ack` signal and moves to the DEINIT.SLV state where it waits for your application to deassert `cxl_<prot>_slv_req` signal in response, before entering the IDLE state. In this scenario, the CXL Transmitter continues granting credits in the DEINIT.SLV state to allow your application to complete any ongoing transactions.
- **Tx deinitialization initiated by your application:** Your application can initiate Tx deinitialization by de-asserting `cxl_<prot>_slv_req` signal. When the CXL Transmitter observes de-assertion of `cxl_<prot>_slv_req` signal by your application, it moves to the DEINIT.MST state. The CXL Transmitter waits for all ongoing transaction on the channel interfaces to complete, then deasserts `cxl_<prot>_slv_ack` signal to indicate to your application that it is ready to enter the IDLE state, and subsequently, makes a transition to the IDLE state

Figure 3-16 illustrates the link deinitialization scenario initiated by CXL Transmitter Interface

Figure 3-16 CXL Tx Deinitialization Example Initiated by the Transmitter[1]



Rx Deinitialization

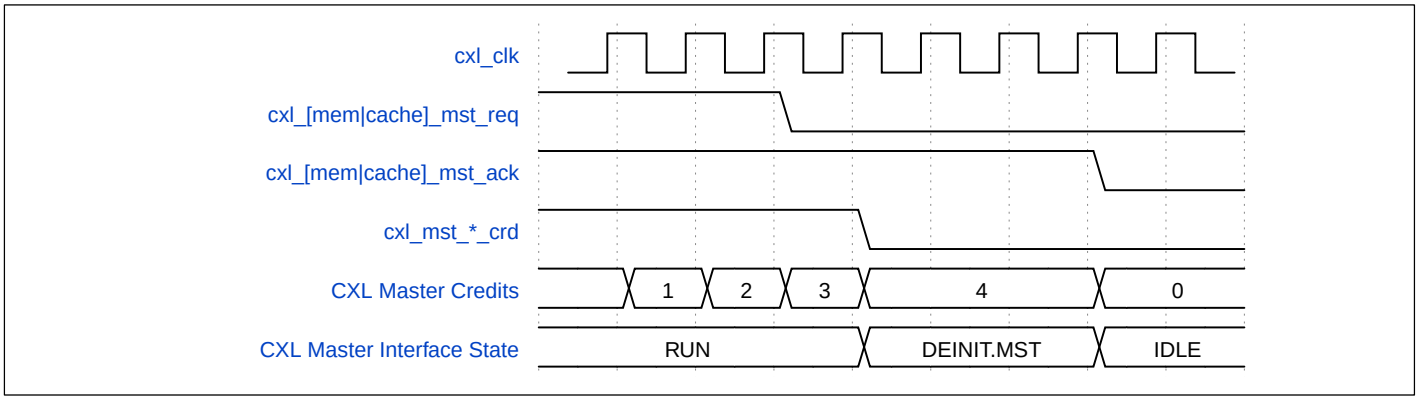
Rx deinitialization can be initiated either by the CXL Receiver or by your application's receive logic.

- **Rx deinitialization initiated by CXL Receiver:** The CXL Receiver requests the link deinitialization to move into low power state. CXL Receiver deasserts `cxl_<prot>_mst_req` signal and moves to the DEINIT.MST state where it waits for your application to deassert `cxl_<prot>_mst_ack` signal in response, before entering the IDLE state.
- **Rx deinitialization initiated by your application:** Your application can initiate Rx deinitialization by de-asserting `cxl_<prot>_mst_ack` signal. When the CXL Receiver observes de-assertion of `cxl_<prot>_mst_ack` signal in the RUN state, it moves to the DEINIT.SLV state, completes all ongoing transactions on the channel interfaces, then deasserts `cxl_<prot>_mst_req` signal to indicate to your application that it is ready to enter the IDLE state, and subsequently, makes a transition to the IDLE state.

Figure 3-17 illustrates the link deinitialization scenario initiated by CXL Receiver Interface.

1. The example illustrates CXL Link Deinitialization from CXL Transmitter Interface perspective.

Figure 3-17 CXL Rx Deinitialization Example Initiated by the CXL Receiver[1]



3.5.3 Channel Interface Message Format

The CXL Channel interface is closely aligned with the message formats defined in the *Compute Express Link Specification, Revision 3.0, Version 0.7*, each channel has a valid bit in the LSB. When this bit of the interface is asserted, it indicates a transmit/received message in that cycle.

Each message transfer single beat occurring in a single cycle.

Since the interfaces are credited, there is no halt mechanism, and a transaction is accepted on the same cycle in which it is presented.

Note The message header format and bit-order on the Channel Message Interface must replicate the Slot format definitions as defined in the CXL Specification.

Table 3-6 D2H Request Message Format

Bits	Name	Description
0	Valid	D2H Request Message Valid bit
78:1	Message	D2H Request Message Fields

Table 3-7 D2H Response Message Format

Bits	Name	Description
0	Valid	D2H Response Message Valid bit
19:1	Message	D2H Response Message Fields

Table 3-8 D2H Data Message Format

Bits	Name	Description
0	Valid	D2H Data Message Valid bit

1. The example illustrates CXL Link Deinitialization from CXL Receiver Interface perspective.

Bits	Name	Description
16:1	Header	D2H Data Message Header Fields
80:17	Byte Enable	D2H Byte Enable (64 bits)
592:81	Data	D2H Data (512 bits)

Table 3-9 S2M No Data Response (NDR) Message Format

Bits	Name	Description
0	Valid	S2M NDR Message Valid bit
29:1	Message	S2M NDR Message Fields

Table 3-10 S2M Data Response Message Format

Bits	Name	Description
0	Valid	S2M Data Message Valid bit
39:1	Header	S2M Data Message Header Fields
551:40	Data	S2M Data (512 bits)

Table 3-11 H2D Request Message Format

Bits	Name	Description
0	Valid	H2D Request Message Valid bit
63:1	Message	H2D Request Message Fields

Table 3-12 H2D Response Message Format

Bits	Name	Description
0	Valid	H2D Response Message Valid bit
31:1	Message	H2D Response Message Fields

Table 3-13 H2D Data Message Format

Bits	Name	Description
0	Valid	H2D Data Message Valid bit
23:1	Header	H2D Data Message Header Fields
535:24	Data	H2D Data (512 bits)

Table 3-14 M2S Request Message Format

Bits	Name	Description
0	Valid	M2S Request Message Valid bit
86:1	Message	M2S Request Message Fields

Table 3-15 M2S Request With Data (RwD) Message Format

Bits	Name	Description
0	Valid	M2S RwD Message Valid bit
86:1	M2S Data Request	M2S RwD Message Header Fields
150:87	M2S Byte Enables	M2S Byte Enable (64 bits)
662:151	M2S Data	M2S Data (512 bits)

Note Based on the mode of CXL link negotiation, your application must set or ignore the reserved bits in the message.

Further details on the Message fields can be found in *Compute Express Link Specification, Revision 3.0, Version 0.7*, however the bit of most interest to the controller is the Valid bit which maps to the bit position '0' in the Message format. The CXL controller monitors Valid bit to determine when the transfer is valid.

Byte Enable bits in D2H and M2S Data Messages determine the variable size of the data messages. CXL controller assumes that generally all bytes are enabled for most of data messages. When all bytes are enabled, the link layer does not transmit the byte enable bits.

3.5.4 Channel Interface Flow Control

The CXL controller implements a credit exchange mechanism for flow control. When the CXL protocol link is IDLE, CXL Receiver does not have credits, and therefore it cannot send messages across the channel interfaces. CXL Transmitter grants credits to CXL Receiver by asserting `cxl_slv_<msg>_crd` signal.

One credit allows CXL Receiver to send a single message across the channel interface. Each cycle in which `cxl_mst_<msg>_crd` signals is asserted grants a single credit to the corresponding CXL Receiver Channel Interface. Each cycle in which CXL Receiver Channel Interface sends one valid message across the channel interface, implicitly returns one credit to the given CXL Transmitter Channel Interface.

You can use `CX_CXL_ CHI_MAXCREDITS` parameter to configure the maximum number of credits that CXL Transmitter Channel Interface grants to CXL Receiver Channel Interface. CXL Transmitter Interface channel tracks the credits it has granted at any given time.

CXL Receiver Interface cannot use a credit to send a message over channel until the cycle after the credit grant signal is asserted.

Figure 3-18 shows an example of basic credit exchange for a H2D Request Channel. The CXL controller grants a single credit to H2D Request channel and gets a single message in return. CXL controller then grants three credits and receives three messages on H2D Request channel interface.

Figure 3-18 CXL Channel Interface Basic Credit Exchange Example

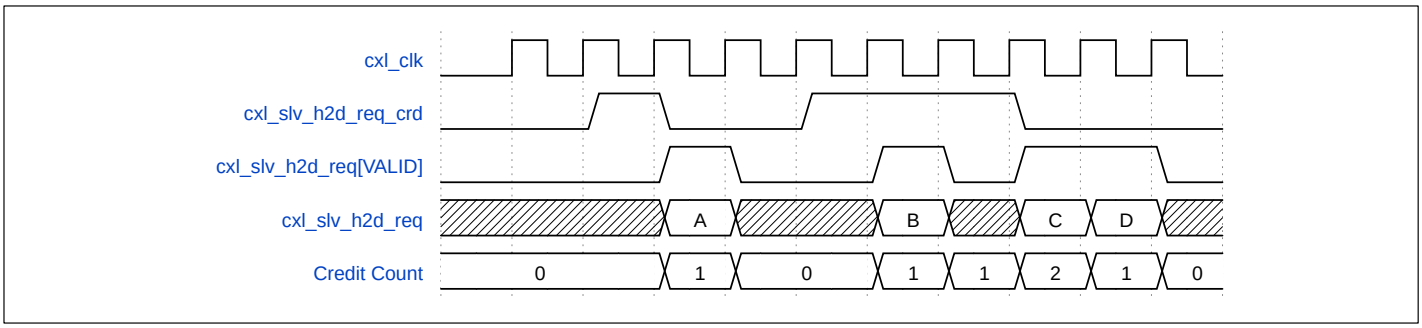
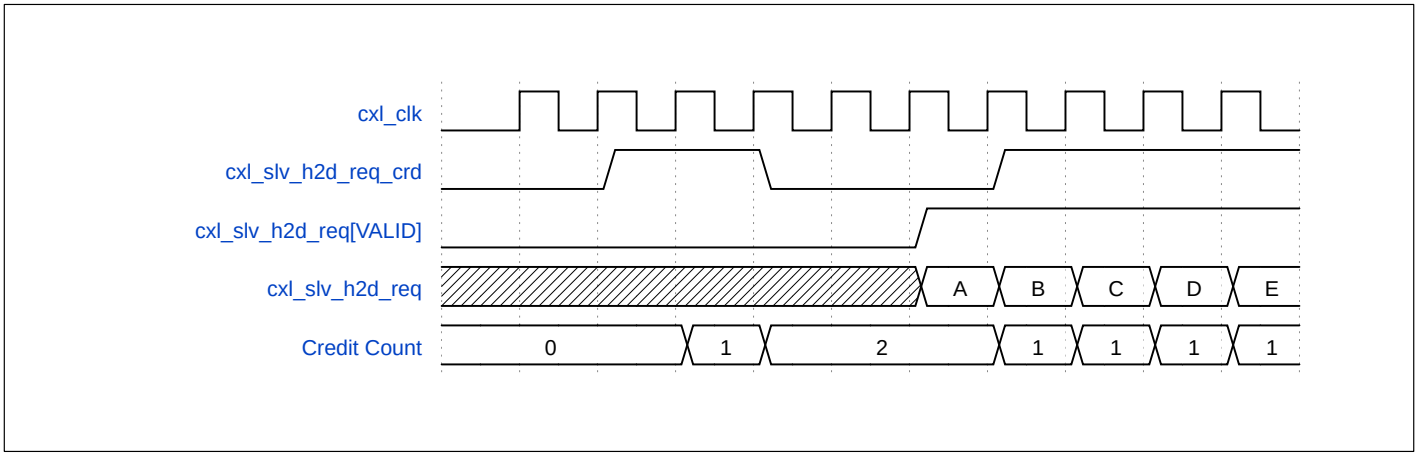


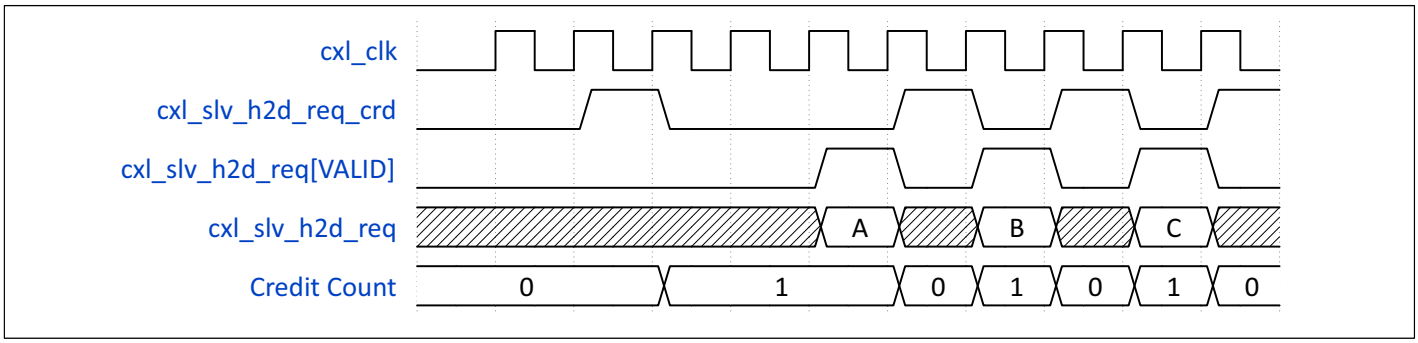
Figure 3-19 shows a steady state credit exchange example. In this example, the credit loop is two cycles: the steady state delay between sending a message on the channel, receiving back the credit that message consumes, and sending another message targeting the same channel buffer resource. In this case, message "A" is transmitted, the credit is returned one cycle later, and message "C" is sent one cycle later using the returned credit. Because this is a two-cycle loop, to keep the interface flowing at full bandwidth, CXL controller must introduce at least two credits for H2D Request Channel and therefore have a storage for two H2D Request messages.

Figure 3-19 CXL Message Channel Interface Steady Credit Exchange Example



In this example, if CXL controller has a buffer for only one H2D Request message and hence only one credit for H2D Request Channel, the performance would be lower, as shown in Figure 3-20.

Figure 3-20 CXL Message Channel Interface Low Credit Exchange Example



3.5.5 CXL Controller Channel Interface Credit Requirement

Total credit loop cycles is the sum of controllers credit cycles requirement and application credit cycles requirement.

When a message is received on the CXL TX channel interface, corresponding credit is available in 2 cycles. So controller has it's requirement of 2 cycles towards total credit loop. To achieve full bandwidth, configure CXL TX channel interface credits to generate buffer size that supports total credit loop cycles

When a credit is available on CXL RX Channel interface, corresponding valid message is available in 3 cycles. So controller has it's requirement of 3 cycles towards total credit loop. To achieve full bandwidth, application must implement a buffer that supports the total credit loop cycles.

3.5.6 Ordering Rules

To comply with the *Compute Express Link Specification, Revision 3.0, Version 0.7* ordering rules, the following ordering rules are implemented for both Tx and Rx in the CXL Transaction Layer:

- H2D Request messages may not pass H2D Response messages if they are put on the Channel Message interface on the same or later cycle than H2D Response message. This ensures that the controller meets the requirement that GO messages that have been sent over CXL.cache in a given cycle cannot be passed by snoops sent in a later cycle.
- M2S Request messages may not pass M2S Request messages put on the Channel Message interface on the same or later cycle. This ensures that the controller meets the requirement that MemRd*/MemInv* messages must not pass prior to MemFwd* messages to the same cache line address.
- If there are more than one message channel for a given message type, for example D2H Request or H2D Response messages, the message order is not preserved between the channels.
- The order within a single message channel is always preserved.

3.6 CXL Register Module

The Compute Express Link (CXL) Device control and status registers are mapped into separate spaces:

- Configuration space registers: These are accessed using configuration reads and configuration writes.
- Memory mapped registers: These are accessed using memory reads and memory writes.

3.6.1 Configuration Space Registers

The CXL configuration space registers are implemented by the CXL 1.1 RCiEP, CXL 2.0 Devices, CXL 2.0 Root Ports, and CXL 2.0 Switch Ports. The CXL 1.1 Downstream Ports do not map any registers into configuration space.

Figure 3-21 CXL 1.1 RCiEP

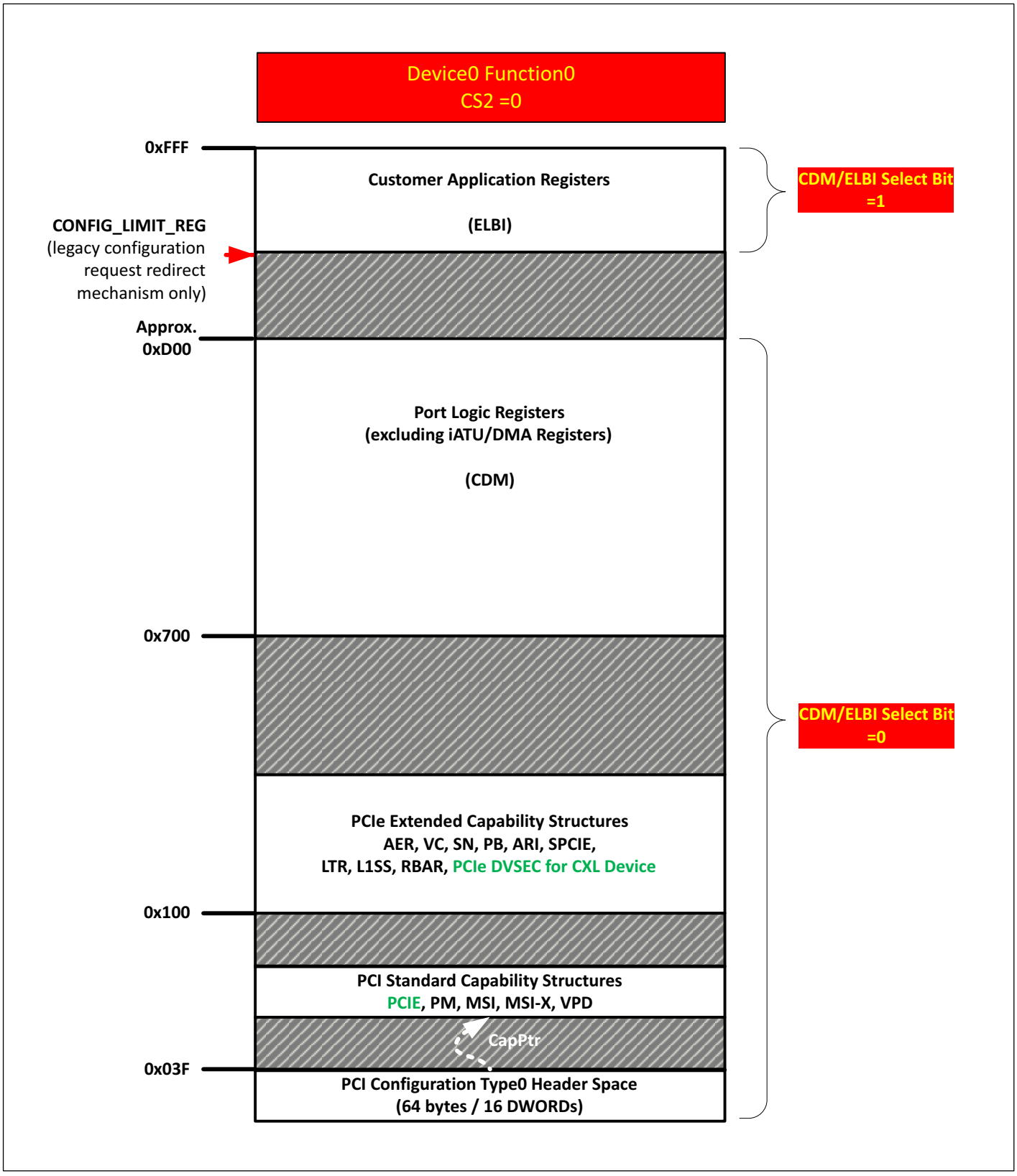


Figure 3-22 PCIe Configuration Space for CXL 2.0 Device

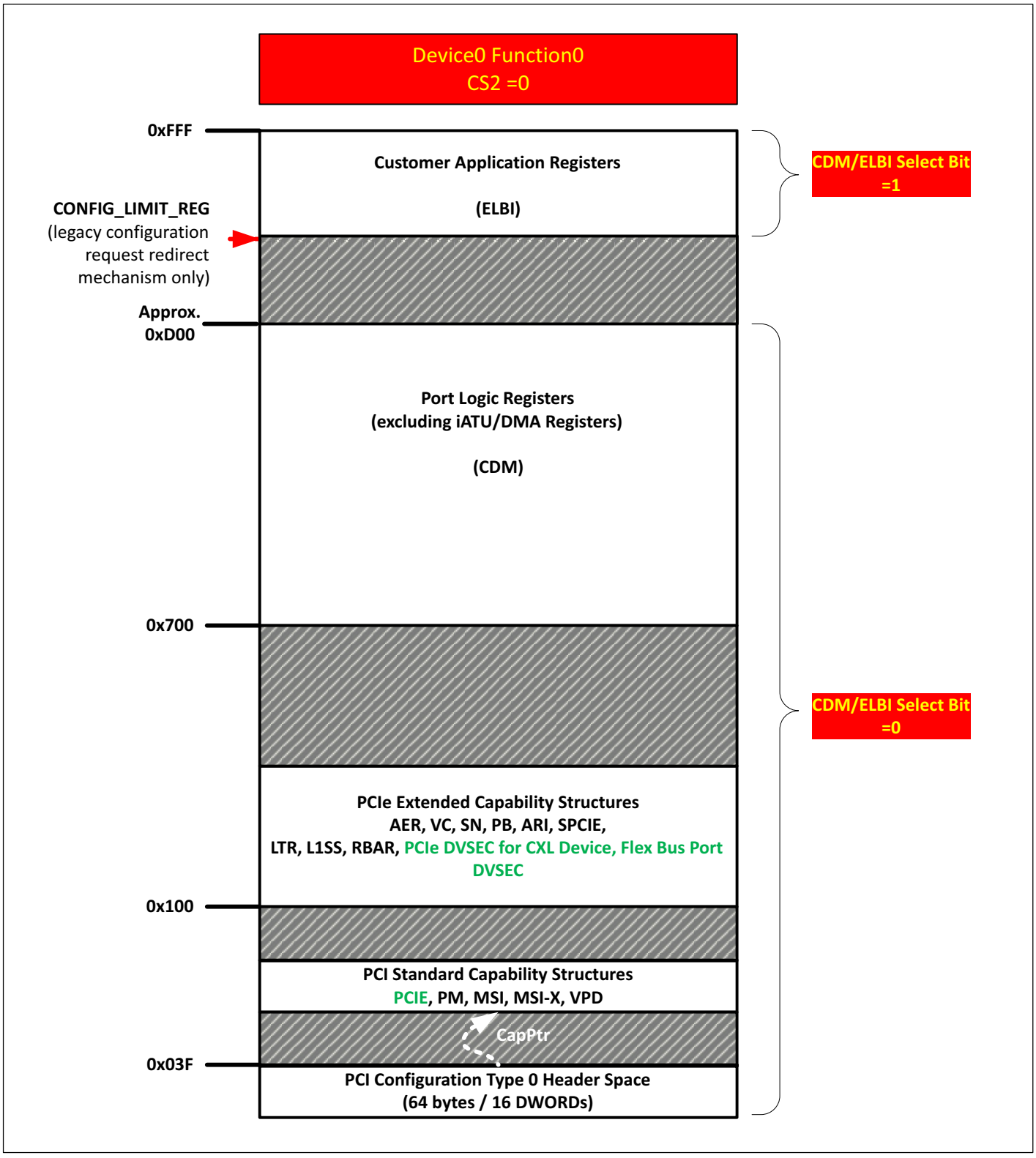
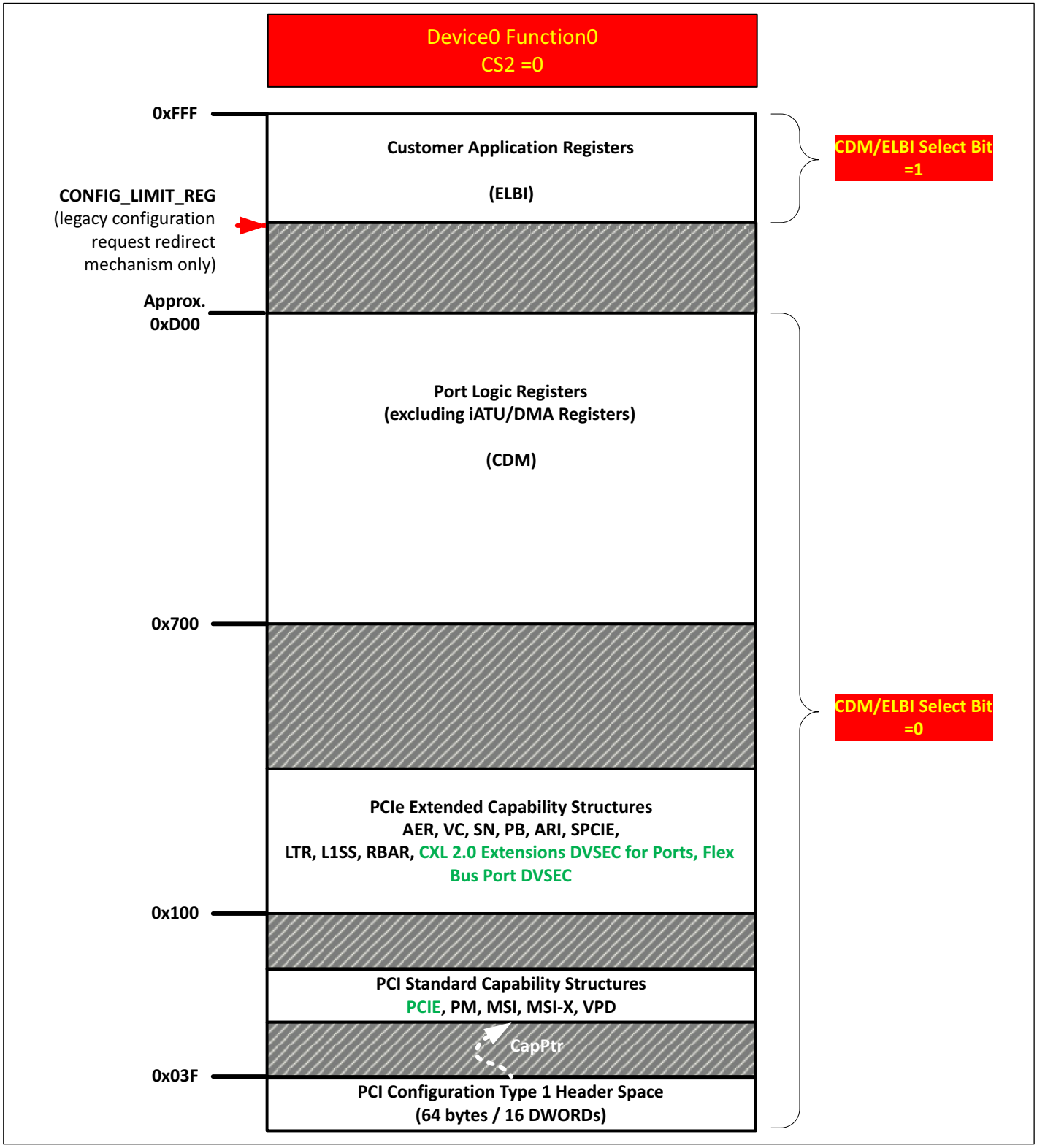


Figure 3-23 PCIe Configuration Space for CXL 2.0 Root Port



Note CONFIG_LIMIT_REG is not valid for a CXL 2.0 Root Port that does not have ECAM enabled. Application must take care of the partitioning between CDM and ELBI by implementing a CONFIG_LIMIT_REG register outside the controller. For more information, see Controller Operations chapter in the *PCIe Controller Databook.*

3.6.1.1 PCIe DVSEC for CXL Device

The CXL 1.1 Device is exposed as Root Complex integrated Endpoint. The PCIe configuration of Function 0 includes the CXL PCI Express Designated Vendor-Specific Extended Capability (DVSEC).

A CXL 2.0 Device is enumerated like a standard PCIe Endpoint and appears below a CXL 2.0 Root Port. The PCIe configuration space of Device 0, Function 0 includes the CXL PCI Express Designated Vendor-Specific Extended Capability (DVSEC).

3.6.1.2 CXL 2.0 Extensions DVSEC for Ports

The PCIe configuration space of CXL 2.0 Root Ports and CXL 2.0 Switch Ports implements CXL 2.0 Extensions DVSEC for ports. For more information, see `CXL_2_0_EXT_*` registers in the "Register Descriptions" section of the *Reference Manual*.

3.6.1.3 CXL 2.0 Configuration Space Mapped Registers (Implemented External to the Controller)

Your application must implement the following CXL 2.0 configuration space registers/capabilities. The controller accesses these registers through the ELBI interface.

- Non-CXL Function Map DVSEC (only for CXL 2.0 Endpoint)
- MLD DVSEC (only for CXL 2.0 Endpoint)
- DoE

Note By default, GPF DVSEC for CXL Ports (`PF0_CXL_GPF_PORT_CAP`), GPF DVSEC for CXL Device (PF0_CXL_GPF_DEVICE_CAP), and Register Locator DVSEC (PF0_CXL_REG_LOCATOR_CAP)are internally implemented inside the controller.

Note Unlike PCIe RC, the CXL Host (CXL Root Port) supports the ELBI interface.

3.6.2 Memory Mapped Space

This section describes the CXL Device and CXL Host memory mapping.

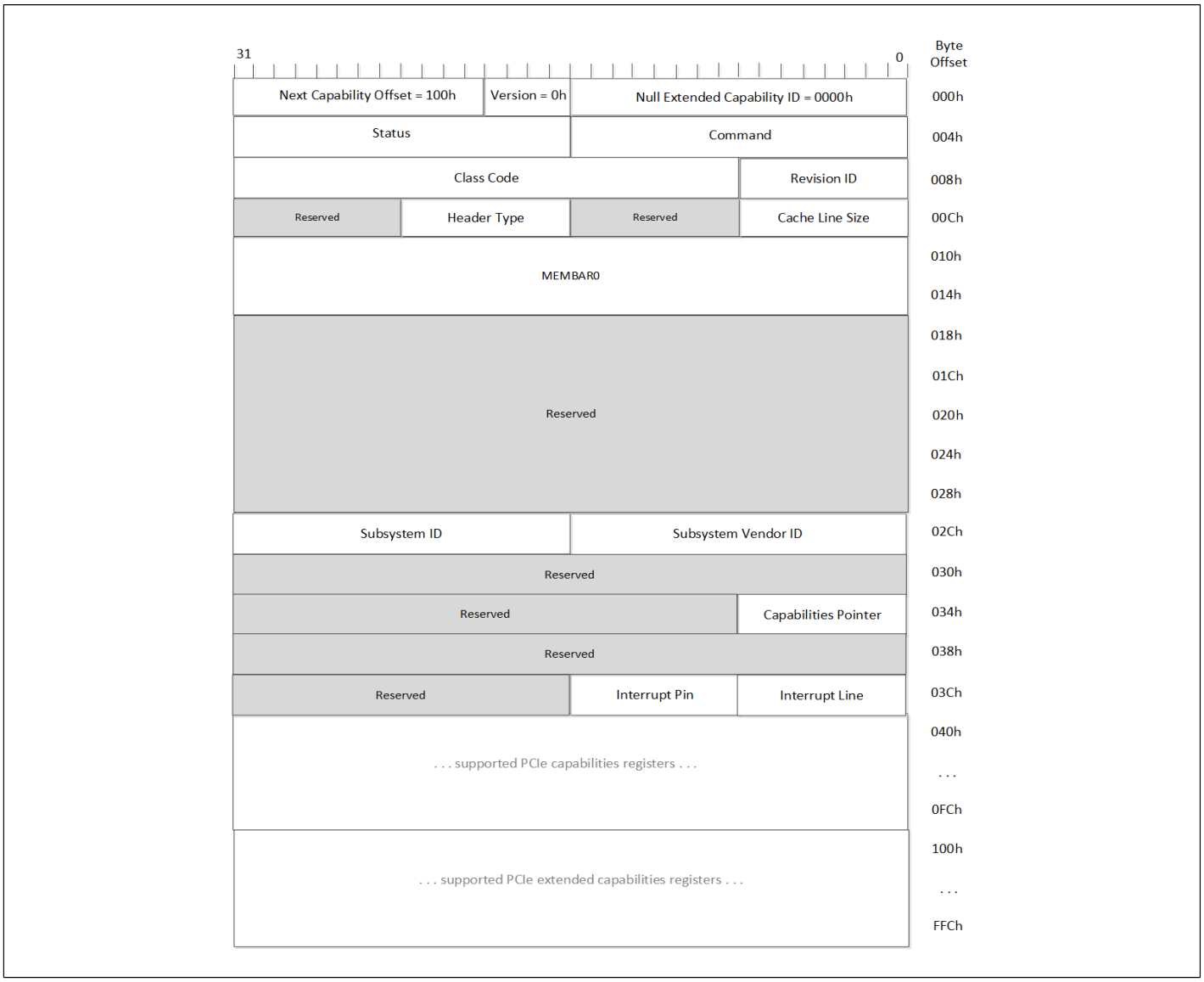
3.6.2.1 CXL 1.1 Upstream Port RCRB

The CXL Device (Upstream Port) controller is not discoverable through the PCIe configuration space. The CXL Device controller implements the PCIe Root Complex Registers Block (RCRBs). Additionally, the CXL Device controller also implements a MEMBAR0 region to Host registers for configuring the CXL subsystem components associated with the Upstream Port. The RCRB regions in the CXL Device controller are:

1. CXL Upstream Port RCRB
2. CXL Upstream Port MEMBAR0

The CXL Device controller RCRB is a 4K memory region. The address of this region is defined by the first memory read access received after the link initialization. The CXL Device controller snoops this access. The base address of CXL Device controller RCRB is defined by the upper address bits [63:12] of the first memory read access.

Figure 3-24 CXL Device Controller RCRB

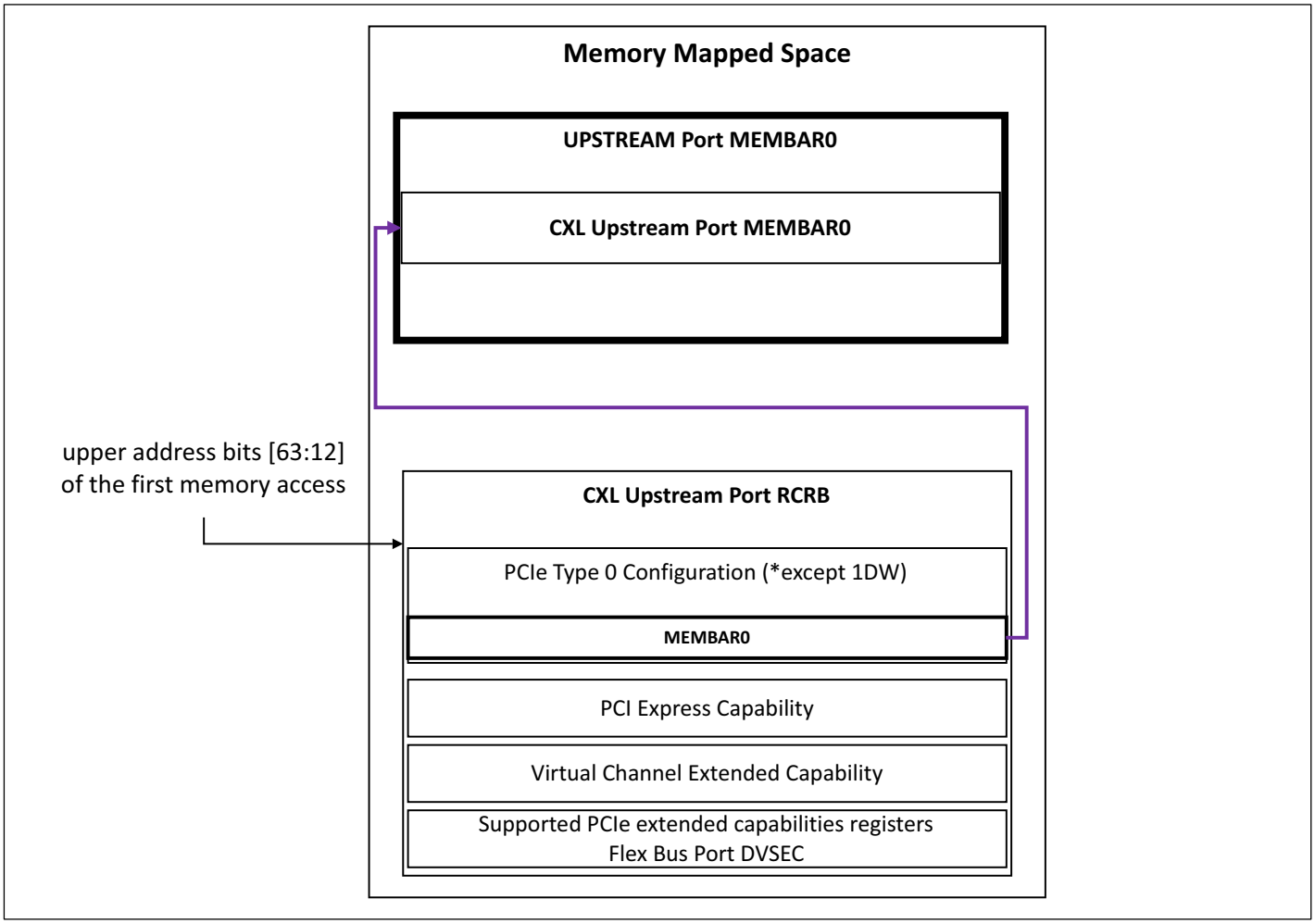


Except for the first DW, the first 64 bytes of the CXL Device controller RCRB implement the registers from a PCIe Type0 Configuration Header. The first DW contains a NULL Extended Capability ID with a Version of 0h and a Next Capability Offset pointer.

A 64-bit MEMBAR0 is implemented at offset 10h and 14h; this points to a memory region that hosts registers for configuring Upstream Port subsystem CXL.mem.

The supported PCIe capabilities and extended capabilities are discovered by following the linked lists of pointers. The supported PCIe capabilities are mapped into the offset range from 040h to 0FFh. The supported PCIe extended capabilities are mapped into the offset range from 100h to FFFh.

Figure 3-25 CXL Device Controller RCRB and MEMBAR



The CXL Device controller supported PCIe capabilities and extended capabilities are listed in Table 3-16:

Table 3-16 PCIe Capabilities and Extended Capabilities Supported by CXL Device Controller

Supported PCIe Capabilities and Extended Capabilities
PCI Express Capability
PCI Power Management Capability
MSI Capability
Advanced Error Reporting Extended Capability
Virtual Channel Extended Capability
Designated Vendor-Specific Extended Capability (DVSEC)
Data Link Feature Extended Capability
Secondary PCI Express Extended Capability
Physical Layer 16.0 GT/s Extended Capability
Physical Layer 32.0 GT/s Extended Capability
Lane Margining at the Receiver Extended Capability

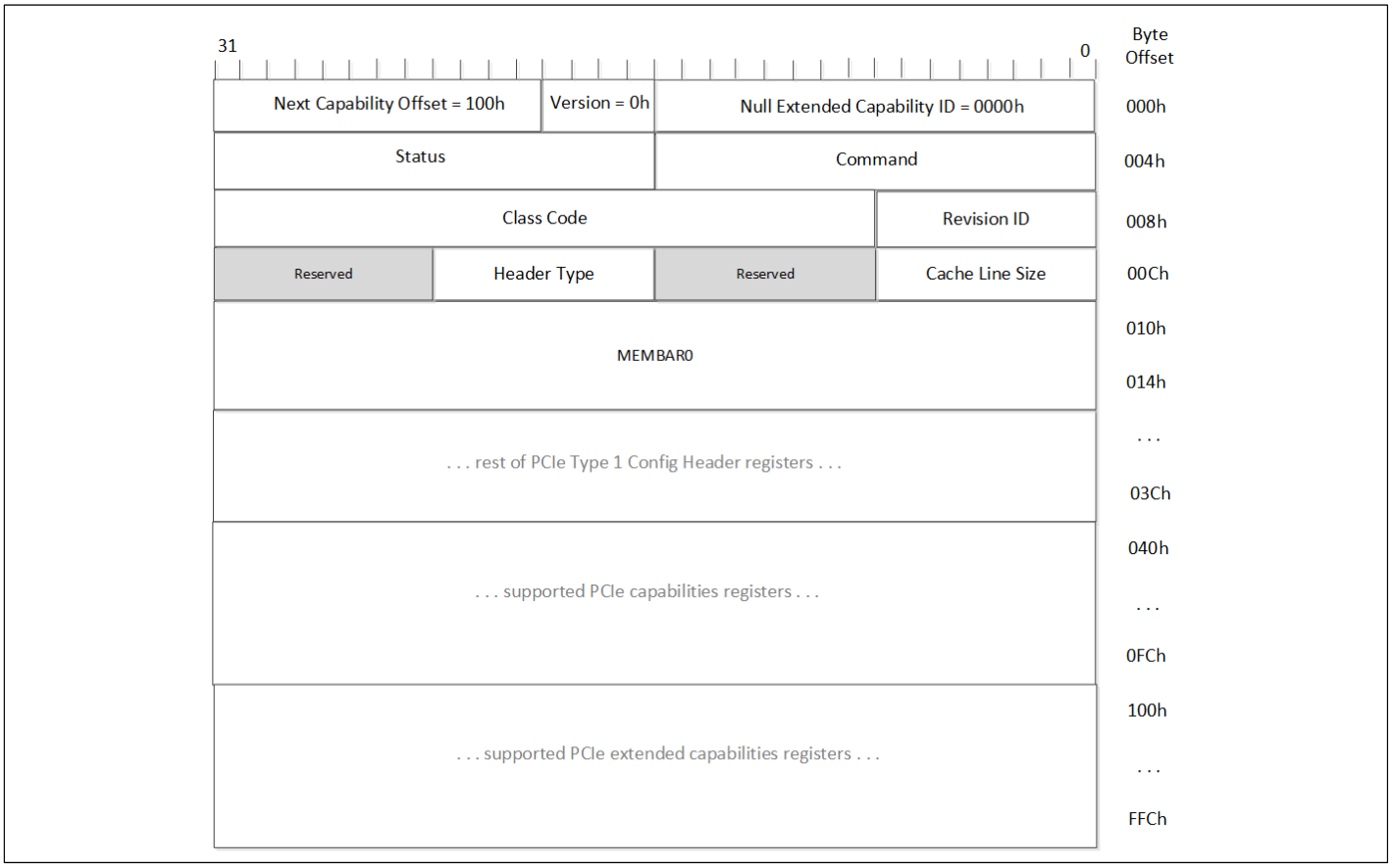
3.6.2.2 CXL 1.1 Downstream Port RCRB

The CXL Host (Downstream Port) controller is not discoverable through the PCIe configuration space. The CXL Host controller implements the PCIe root complex registers block (RCRBs). Additionally, the CXL Host controller also implements a MEMBAR0 region to Host registers for configuring the CXL subsystem components associated with the Downstream Port. The RCRB regions in the CXL Host controller are:

1. CXL Downstream Port RCRB
2. CXL Downstream Port MEMBAR0

The CXL Host controller RCRB is a 4K region with registers based upon PCIe defined registers for a Root Port with some exceptions. The CXL Host controller includes registers from PCIe Type 1 Configuration Header and PCIe capabilities and extended capabilities. In the PCIe Type1 Configuration Header there is a MEMBAR0 address that indicates the address of CXL Downstream Port MEMBAR0.

Figure 3-26 CXL Downstream Port RCRB



The first DW of the RCRB contains a NULL Extended Capability ID with a Version of 0h and a Next Capability Offset pointer.

Table 3-17 lists the PCIe capabilities and extended capabilities supported by the CXL Host controller.

Table 3-17 PCIe Capabilities and Extended Capabilities Supported by CXL Host Controller

Supported PCIe Capabilities and Extended Capabilities
PCI Express Capability[a]
PCI Power Management Capability
MSI Capability
Advanced Error Reporting Extended Capability
ACS Extended Capability
Downstream Port Containment Extended Capability
Data Link Feature Extended Capability
Virtual Channel Extended Capability
Designated Vendor-Specific Extended Capability (DVSEC)
Secondary PCI Express Extended Capability
Physical Layer 16.0 GT/s Extended Capability
Physical Layer 32.0 GT/s Extended Capability
Lane Margining at the Receiver Extended Capability

a. Slot Capabilities, Slot Control, Slot Status, Slot Capabilities 2, Slot Control 2, and Slot Status 2 registers are not applicable.

3.6.2.3 Flex Bus Port DVSEC

The CXL Upstream and Downstream Ports implement a Flex Bus Port DVSEC, which is distinct from that implemented by a CXL Device in the configuration space. This DVSEC is in the RCRBs of the 1.1 Upstream and 1.1 Downstream Ports.

This DVSEC is in the primary function (Device 0, Function 0) of the CXL 2.0 Upstream Port and the configuration space of the CXL 2.0 Downstream Port.

3.6.2.4 CXL2.0 Component Registers

The controller maps the 64K CXL 2.0 Port Specific Component Registers to PCIe Function0, BAR0. The mapping starts at the offset stored in the Register Block 1 entry of the Register Locator DVSEC.

Note

The default value for the MEM_FUNC0_BAR0_TARGET_MAP parameter is ‘1’ (RTRGT1).

In CXL versions prior to 5.97a, it was stipulated that you must set the MEM_FUNC0_BAR0_TARGET_MAP parameter to '0' (RTRGT0) as CXL2.0 Port Specific Component Registers need to be mapped to RTRGT0.

From the current release, the CXL2.0 Port Specific Component registers are always mapped to RTRGT0 independently from the MEM_FUNC0_BAR0_TARGET_MAP setting. This allows a more flexible use of PCIe Function0, BAR0 resources.

Table 3-18 CXL 2.0 Component Specific Register Space Implementation

Range	Size	Destination	Implementation
0000_0000h to 0000_0FFFh	4k	CXL.io registers	External (ELBI)[a]
0000_1000h to 0000_100Fh	4K	CXL.cache and CXL.mem registers	Internal
0000_1010h to 0000_101Fh[b]			External (ELBI)
0000_1020h to 0000_11FFh			Internal
0000_1200h to 0000_1FFFh[c]			External (ELBI)
0000_2000h to 0000_DFFFh	48K	Implementation specific registers	External (ELBI)
0000_E000h to 0000_E3FFh	1K	CXL ARB/MUX registers	Internal
0000_E400h to 0000_FFFFh	7K	Reserved	External (ELBI)

a. The output signal lbc_ext_cxl_mbar0_access indicates accesses to the ELBI mapped regions of the CXL 2.0 Port Specific Component Registers. You can use signal lbc_ext_cxl_mbar0_access along with lbc_ext_addr[15:0] to decode accesses to ELBI mapped CXL 2.0 Port Specific Component Registers region.
b. You must implement this address range on ELBI for CXL_HDM_Capability_Header, CXL_Extended_Security_Capability_Header, CXL_IDE_Capability_Header, and CXL_Snoop_Filter_Capability_Header registers.
c. You must implement this address range on ELBI for CXL_HDM_Capability, CXL_Extended_Security_Capability, CXL_IDE_Capability, and CXL_Snoop_Filter_Capability registers.

3.6.2.5 CXL 2.0 Memory Mapped Registers (Implemented External to the Controller)

Your application must implement the following CXL 2.0 memory mapped registers/capabilities. The controller does not implement these registers. These CXL 2.0 memory mapped registers/capabilities can be accessed through the ELBI interface.

- CXL HDM Decoder Capability
- CXL Extended Security Capability
- CXL IDE Capability
- CXL Snoop Filter Capability
- BAR Virtualization ACL Register Block
- Memory mapped CXL Device Register

3.6.2.6 DBI Access Mechanism for the RCRB, RCRB MEMBAR0, and ELBI2 Regions

The DBI access for the RCiEP is the same as that for the PCIe Config and Extended Config register space. You can access the RCRB and RCRB MEMBAR0 using the DBI access mechanism shown in Table 3-19 for the native controller configuration.

Table 3-19 Native Controller DBI Address Bus Layout (dbi_cs2 =1 and cs =1)

Access Type	31-20	21	20	19	18-17	16 - 9	15 - 12	11 - 9	8	7-2	1	0
RCRB	Not Used	1	0	Reserved				1K DWORD RCRB Address			0	1
RCRB MEMBAR0	Not Used	1	1	Reserved			16K DWORD RCRB MEMBAR Address				0	1

Note

For CX_SRIOV_ENABLE =1 with native DBI interface,

- The DBI access for the CXL RCIEP is the same as that for PCIe.
- The dbi_func_num input port must be driven to 0 when accessing the RCRB or the RCRB MEMBAR0.

For CXL controller, the parameter DBI_MULTI_FUNC_BAR_EN =1.

CXL 1.1 AXI DBI Access (CX_CXL_VERSION =1 and (DBI_4SLAVE_POPULATED =1 or SHARED_DBI_ENABLED =1))

You can access the RCRB and RCRB MEMBAR0 using the DBI access mechanism shown in Table 3-20 and Table 3-21 for the AXI bridge configuration.

Table 3-20 AXI DBI Address Bus Layout (dbi_cs2 =1 and cs =1) (CX_SRIOV_ENABLE =0 and CX_ARI_ENABLE =0)

Access Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RCRB	0								1	0	1	1	0	Rsvd							1K-DWORD RCRB Addr										0	0
RCRB MEMBAR0	0								1	1	1	1	0	Rsvd			16K-DWORD RCRB MEMBAR Addr														0	0

Table 3-21 AXI DBI Address Bus Layout (dbi_cs2 =1 and cs =1) (CX_SRIOV_ENABLE =1 or CX_ARI_ENABLE =1)

Access Type[a]	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RCRB	1	1	0	0								0	1	0	Rsvd							1K-DWORD RCRB Addr										0	0
RCRB MEMBAR0	1	1	0	0								1	0	0	Rsvd			16K-DWORD RCRB MEMBAR Addr																0	0

a. In releases previous to 5.95a-lca01, the controller used bits 22 and 23 for RCRB or RCRB MEMBAR0 region access.

Table 3-22 describes the RCRB MEMBAR0 implementation.

Table 3-22 RCRB MEMBAR0 Implementation

Range	Size	Destination	Implementation
0000_0000h to 0000_0FFFh	4k	CXL.io registers	External (ELBI)
0000_1000h to 0000_100Fh	4K	CXL.cache and CXL.mem registers	Internal
0000_1010h to 0000_101Fh			External (ELBI)
0000_1020h to 0000_11FFh			Internal
0000_1200h to 0000_1FFFh			External (ELBI)
0000_2000h to 0000_DFFFh	48K	Implementation specific registers	External (ELBI)
0000_E000h to 0000_E3FFh	1K	CXL ARB/MUX registers	Internal
0000_E400h to 0000_FFFFh	7K	Reserved	External (ELBI)

CXL 2.0 AXI DBI Access (CX_CXL_VERSION =2 and (DBI_4SLAVE_POPULATED =1 or SHARED_DBI_ENABLED =1))

You can access the CXL 2.0 Component Specific Registers using the CXL 1.1 RCRB MEMBAR0 mechanism shown in Table 3-20 and Table 3-21.

In addition, the controller provides an ELBI2 DBI mechanism to access the 512K region of any BAR mapped to RTRGT0. You can map the ACL Register Block and the memory-mapped Device Register Block any of the initial 512k region of PCIe BARs mapped to RTRGT0 and use the ELBI2 DBI access mechanism shown in Table 3-23 and Table 3-24 to access these register blocks.

Table 3-23 AXI DBI Address Bus Layout (dbi_cs2 =1 and cs =1) (CX_SRIOV_ENABLE =0 and CX_ARI_ENABLE =0)

Access Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ELBI2	0		Func			BAR			0	1	1	1	0	128 K-DWORD Register Addr																		0	0

Table 3-24 AXI DBI Address Bus Layout (dbi_cs2 =1 and cs =1) (CX_SRIOV_ENABLE =1 or CX_ARI_ENABLE =1)

Access Type	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ELBI2	1	1	0	Func					BAR			0	0	0	128 K-DWORD Register Addr																	0	0

Note Table 3-25 and Figure 3-26 show the Address Bus layout for accessing the other regions for a CXL controller.

Table 3-25 axi dbi address bus layout (cs =1) (cx_sriov_enable =0 and cx_ari_enable =0)

Access Type	31	30	29-22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CDM	0			0	CS2	0	Func			0				1K-DWORD Register Addr										0	0
ELBI	0			1	0	ROM	Func			BAR			I/O	1K-DWORD Register Addr										0	0
iATU	0			1	1	0	iATU Address																	0	0
DMA	0			1	1	1	DMA Address																	0	0
MSI-X Table	0	1	0 0	1	1	0	Func			0	8K-DWORD MSI-X Table Addr													0	0
MSI-X PBA	0	1		1	1	0	Func			1	8K-DWORD MSI-X PBA Addr													0	0

Table 3-26 AXI DBI Address Bus Layout (cs =1) (CX_SRIOV_ENABLE =1 or CX_ARI_ENABLE =1)

Access Type[a]	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CDM	0	CS2	0	VF#								VF Active	PF#					0				1-K DWORD Register Addr										0	0
ELBI	1	0	ROM	VF#								VF Active	PF#					BAR			I/O	1 K- DWORD Register Addr										0	0
iATU	1	1	0	0								1	1	0	iATU Address																	0	0
DMA	1	1	0	0								1	1	1	DMA Address																	0	0
MSI-X Table	1	1	1	VF#								VF Active	PF#					0	8K-DWORD MSI-X Table Addr													0	0
MSI-X PBA	1	1	1	VF#								VF Active	PF#					1	8K-DWORD MSI-X Table Addr													0	0

a. In releases previous to 5.95a-lca01, the controller used bits 22 and 23 for RCRB or RCRB MEMBAR0 region access.

3.6.3 Support for 1DW Unaligned ELBI access (when CX_LBC_NW =2)

All CXL configurations have a 64 bit ELBI databus. For these configurations, the controller supports 1DW unaligned access (accesses to offsets such as 0x4 or 0xC and so on) on the ELBI interface.

3.6.3.1 1DW Unaligned ELBI Read for CX_LBC_NW =2 Configurations

Whenever there is a 1DW unaligned ELBI read, the application must return the data on the upper 32 bits of the ELBI databus.

3.6.3.2 1DW Unaligned ELBI Write for CX_LBC_NW =2 Configurations

Whenever there is a 1DW unaligned ELBI write. The controller presents the data in such a way on the ELBI interface that the upper 32 bits of the ELBI databus are valid.

Note Follow the same procedure to perform 1 DW Unaligned ELBI access through the DBI.

Table 3-27 lbc_ext_rd Behavior

Access Type	Type of Alignment	Value of lbc_ext_rd
1 DW Read Access	Unaligned to 8 bytes but Aligned to 4 bytes such as 0x4, 0xC	0xf0
	Aligned to 8 bytes such as 0x8, 0xF	0x0f
2 DW Read Access	Aligned to 8 bytes such as 0x8, 0xF	0xff
Write Access	Any	0x00

3.6.4 Generating Register Map

There are multiple maps for the controller because some modes are determined by an input pin at reset which adds or removes capability registers from the linked list.

Attention

Generating Register Map Documentation

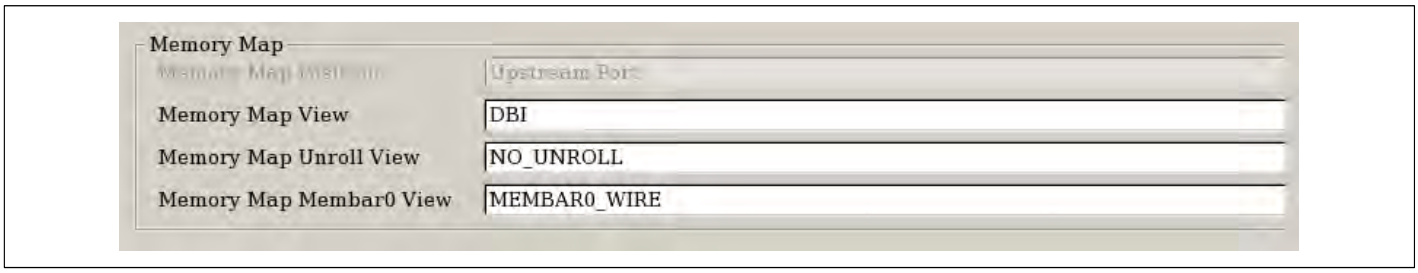
When you configure the controller in coreConsultant, you can access the register descriptions for your actual configuration at *workspace*/report/ComponentRegisters.html using the process described in the “Creating Optional Views and Reports” section in the *User Guide*. This report comes from the exact same source as the Databook but removes all the registers that are not in your actual configuration.

Table 3-28 describes the parameters to consider for generating DocBook XML and HTML Register Reports.

Table 3-28 Parameters to Consider for Generating DocBook XML and HTML Register Reports

Parameter Name	Values	Notes
CX_MEMORY_MAP_POSTION	0: Upstream Port 1: Downstream Port	The memory map of an Upstream Port is different to that of a Downstream Port. Using the CX_MEMORY_MAP_POSTION parameter you can generate the memory map either for the Upstream Port or the Downstream Port.
CX_MEMORY_MAP_VIEW	0: DBI 1: WIRE 2: DBI2	The registers for each port type, Upstream Port/Downstream port can be accessed by the remote link partner through the wire or by your application through DBI (DBI or DBI2). The wire view of the registers is different to the DBI view. To choose the type of register view, use CX_MEMORY_MAP_VIEW parameter.
CX_UNROLL_VIEW	0: NO_UNROLL 1: UNROLL	You can generate the memory map view containing only ■ iATU registers ■ DMA registers
CX_MEMBAR0_VIEW	0: NO_MEMBAR0_WIRE 1: MEMBAR0_WIRE	You can generate the memory map view containing only MEMBAR0 wire view registers.

Figure 3-27 Memory Map Configuration Parameters in coreConsultant



Register Default Values

The register descriptions in this chapter indicate the default of the register after a cold reset, either as a specific numerical value or as the name of the design configuration parameter that sets the default. You can inspect the values of the configuration parameters in two different locations:

- In the Component Configuration report that is described in the Creating Optional Views and Reports section in the *User Guide*
- In `<workspace>/src/DWC_pcie_ctl_cc_constants.svh`

Rebuilding Capability Linked Lists (PFs only)

When your configuration software requires any set of standard PCI capability registers to be removed, it can overwrite the next capability pointers through the DBI and rebuild the linked list structure.

Note

- Even though there is a unique standard capabilities linked lists provided per function, specific capabilities cannot be included/excluded on a per-function basis. For example, MSI-X capability is included/excluded for all functions (PF) at the same time through the coreConsultant GUI.
- Even though there is a unique extended capabilities linked list provided per function, specific capabilities cannot be included/excluded on a per-function basis (in particular for LTR and VC, this is a non-fatal violation of a strict reading of the PCIe specification).

  This is also true for the SR-IOV capability structure when a DM controller is in RP mode.
- Each function can have a different setup of a capability structure after it is included, although some features/settings are common across all functions.
- All VFs have the same linked list of capabilities.

3.7 CXL Controller Reliability, Availability, and Serviceability (RAS)

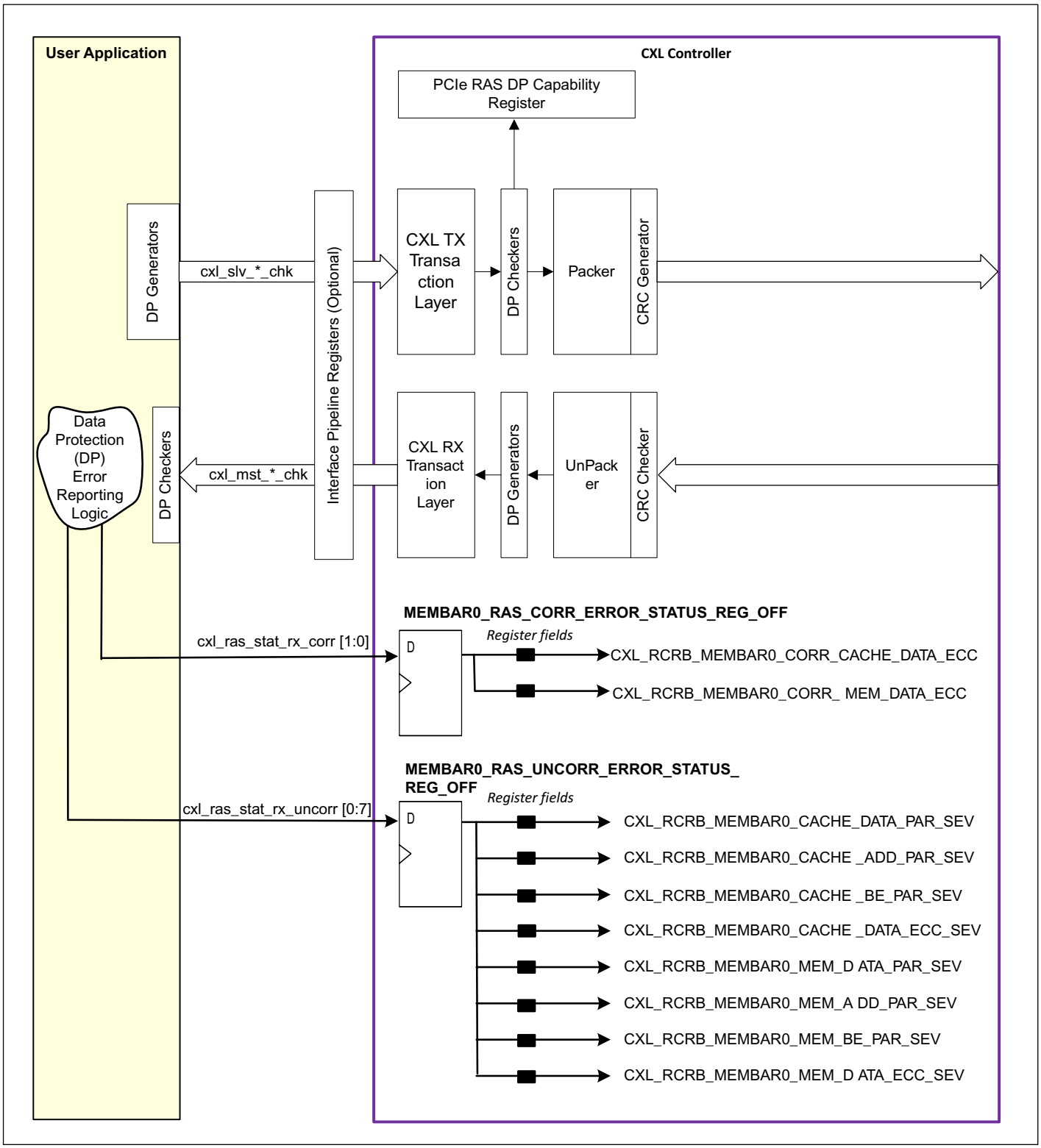
CXL controller Reliability, Availability, and Serviceability (RAS) ensures that failures in the underlying processes and hardware components do not cause any interruptions in the overall system operation. CXL controller RAS consists of following fundamental components:

- Internal device-level datapath and RAM protection (RAS DP) of data, header, and RAM control signals from message channel interfaces to CRC extraction/generation points.
- Debug, Error Injection, and Statistic gathering (RAS DES)
- CXL Link CRC and Retry
- CXL Link Retraining and Recovery
- CXL RAS Correctable and Uncorrectable error detection and reporting
- CXL controller internal device-level fatal error reporting
- CXL controller Viral handling
- CXL controller Data Poisoning

3.7.1 CXL Controller RAS Data Protection

For the parts of the controller outside of CRC protection, the CXL controller provides an optional data protection over the message channel interfaces, internal datapath, and associated RAMs. You can enable RAS DP feature using the parameter `CX_CXL_CHI_DATACHK`.

Figure 3-28 RAS DP Architecture



When CXL Channel interface data protection is enabled, the signals mentioned in Table 3-29 are used.

Table 3-29 Data Protection Signals

Signals[a]	Type of signal	Path	Description
cxl_slv_*_chk[b]	Input Signal	Application to CXL Transmitter	Carries Data protection codes for the corresponding channel messages
cxl_mst_*_chk[c]	Output Signal	CXL Receiver to Application	Carries data protection codes, the CXL controller generates for every CXL.cache/CXL.mem message unpacked from the received flits
cxl_ras_stat_rx_uncorr/ cxl_ras_stat_rx_corr[d]	Input Signals[e]	Application to CXL controller	To report the detected errors to the CXL controller

a. Figures 3-29, 3-30, 3-31, and 3-32 show the mapping between cxl_slv_*_data and cxl_slv_*_data_chk, cxl_mst_*_data and cxl_mst_*_data_chk, and protection codes mapping to individually report an error on the header, byte enable, and data fields of the Data Message.
b. The cxl_slv_*_chk bits must cover the entire bus, zero-extended on the LEAST SIGNIFICANT bit side to 64-bit boundary. So, the CXL controller separately identifies the protection for data, byte enables, and header/address/message.
c. The cxl_mst_*_chk bits are generated for the entire message bus including header, byte enable and data fields for data messages. Data protection codes are generated by aligning the message to 64-bit boundary, zero-extended on the LEAST SIGNIFICANT bit side; so that, the CXL controller separately identifies protection for data, byte enables, and header/address/message.
d. Your application must implement the corresponding data protection checkers for every message channel to detect and report errors.
e. These input signals set the corresponding fields of the RAS Correctable/ Uncorrectable Error Status Register, as described in Table 3-31.

The CXL transmitter does not provide a mechanism to communicate poisoned data information through CXL.cache/CXL.mem Data Header. The CXL controller reports the correctable/uncorrectable errors detected on a message channel transmit datapath using the PCIe RAS DP capability registers.

Figure 3-29 D2H Data Message and Data Protection Code Mapping

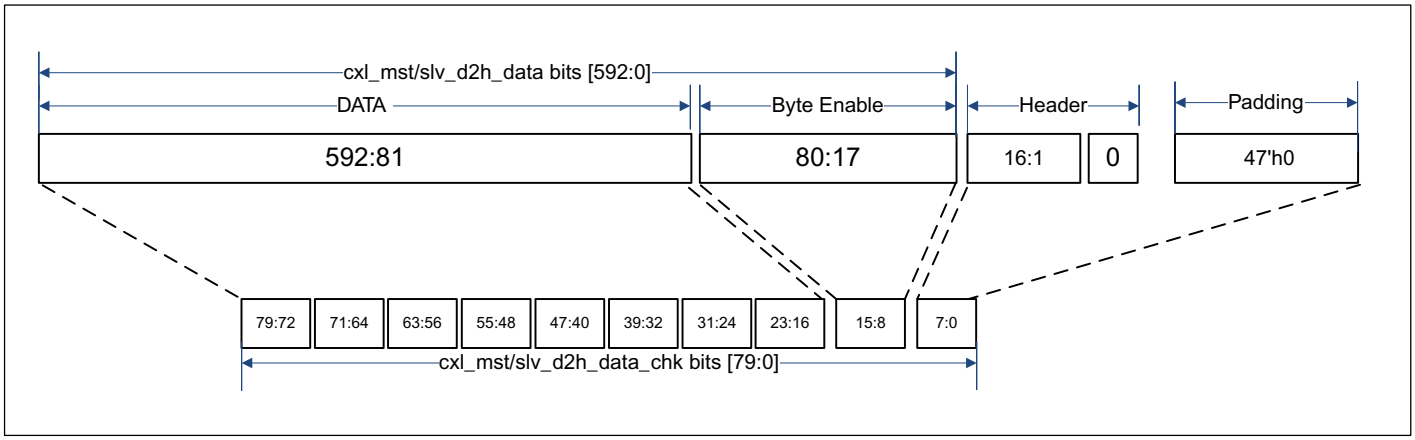


Figure 3-30 S2M Data Message and Data Protection Code Mapping

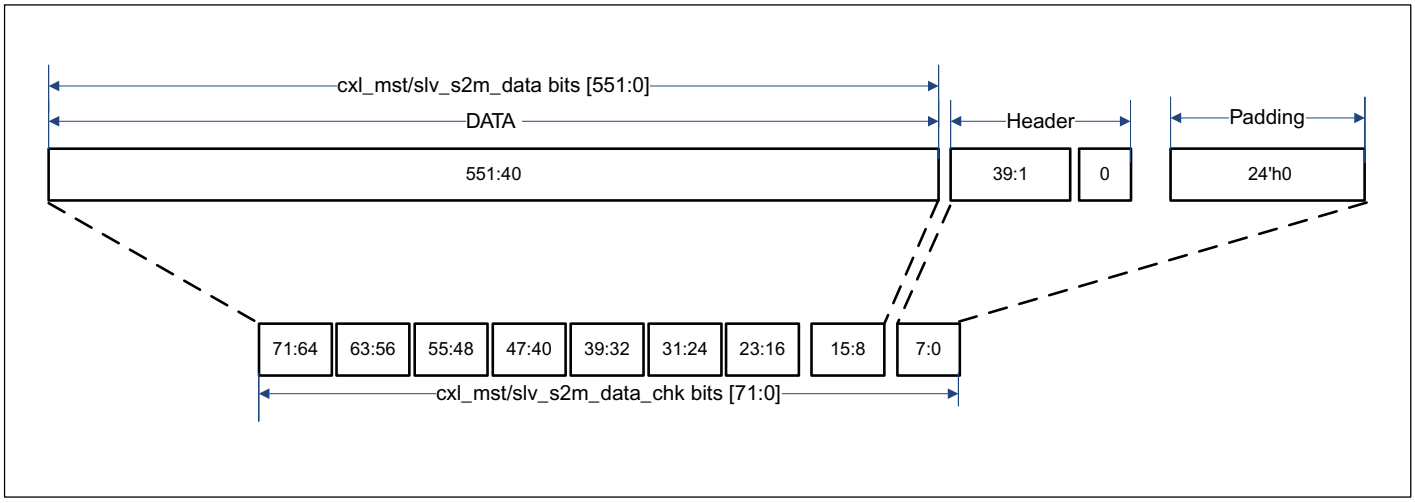


Figure 3-31 H2D Data Message and Data Protection Code Mapping

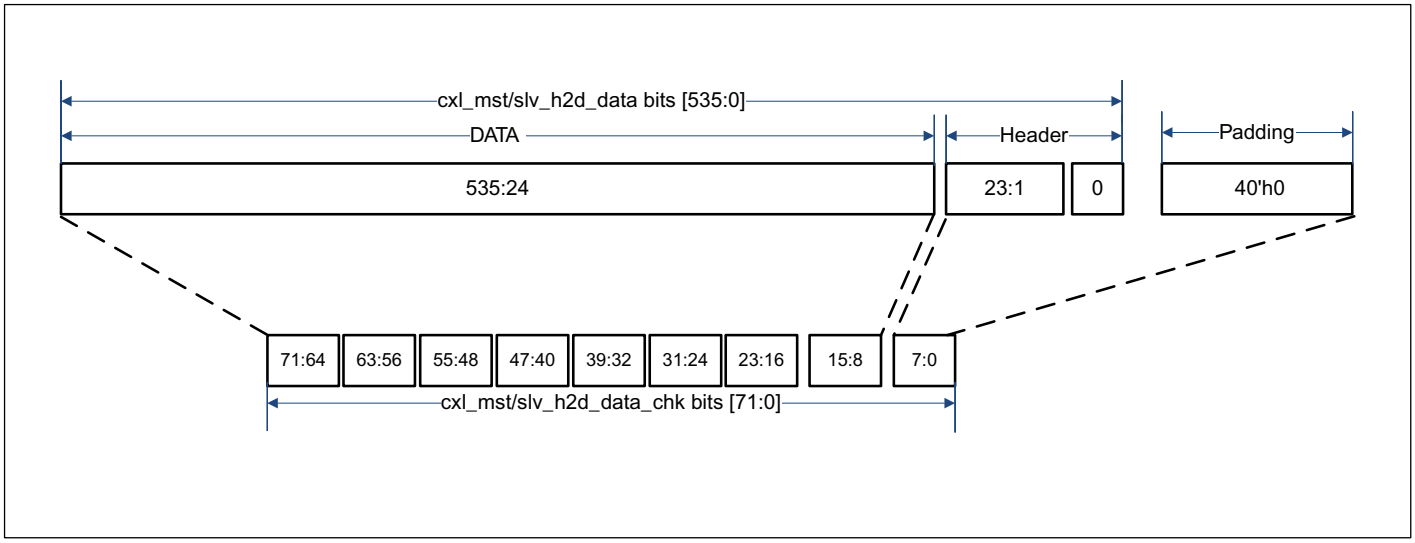
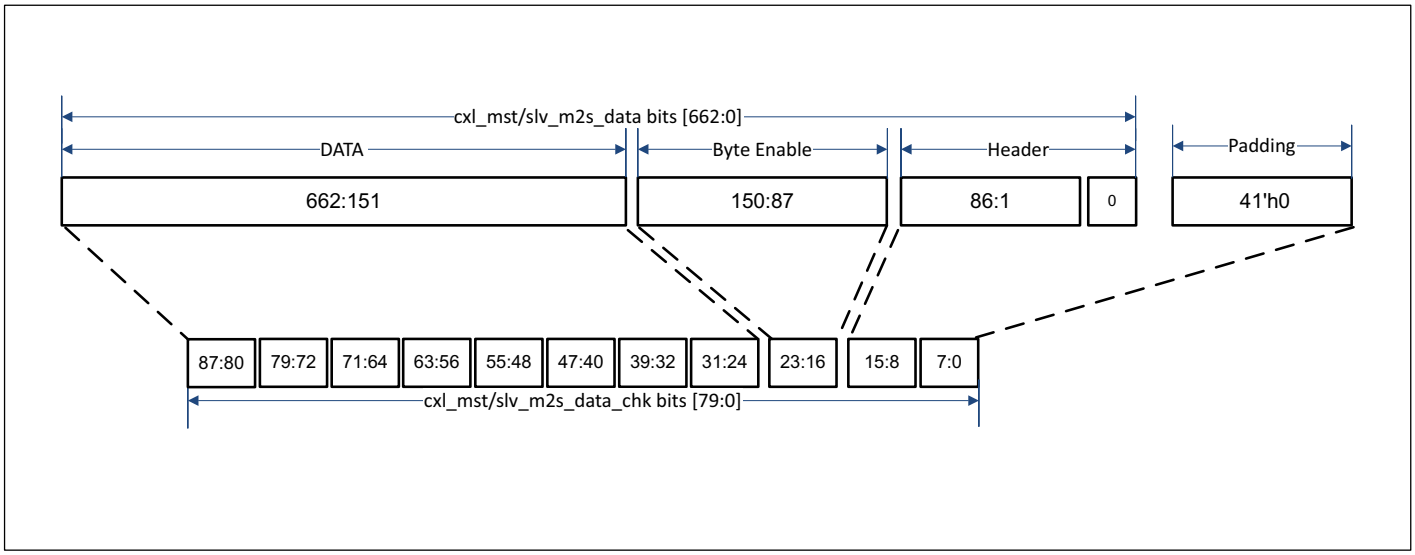


Figure 3-32 M2S Data Message and Data Protection Code Mapping



The Figure 3-33 and Figure 3-34 illustrate an example of bus protection code generator and bus protection code checkers implementation for D2H Data Messages in RASDP Glue Logic and their mapping to CXL controller RAS Interface signals - `cxl_ras_stat_rx_uncorr/cxl_ras_stat_rx_corr`. The similar approach must be used for all data channels.

Figure 3-33 D2H Data Message Protection Code Generator for Slave Channel

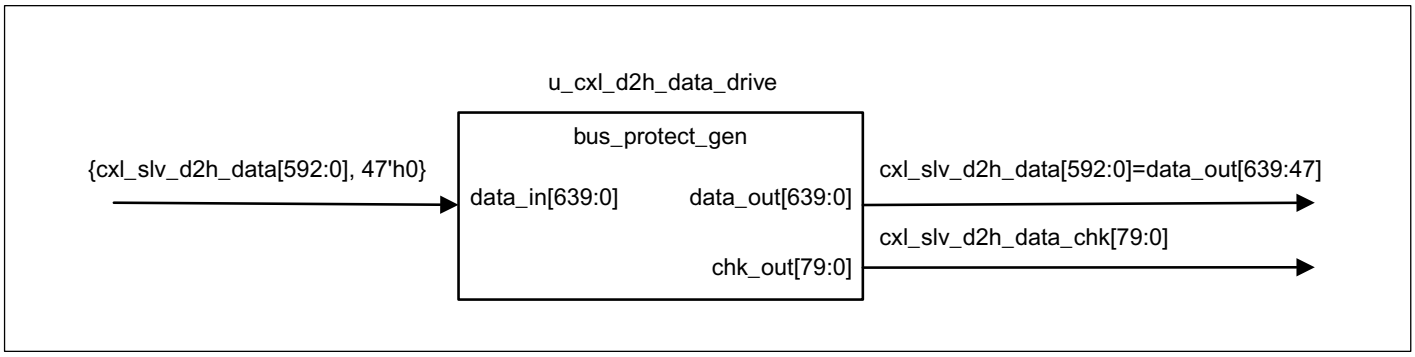
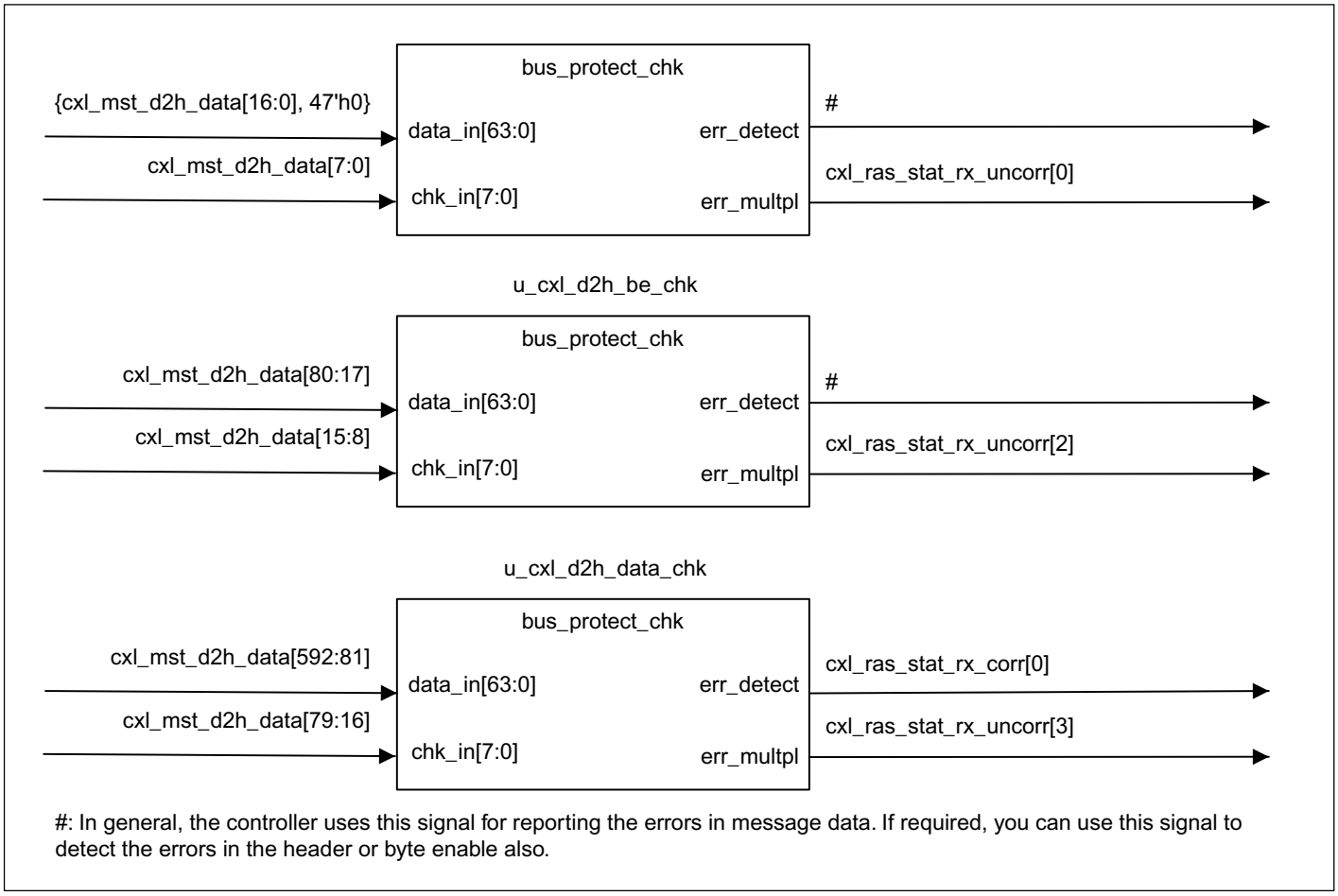


Figure 3-34 D2H Data Message Protection Code Checkers for Master Channel



You can use the input signal `cxl_ras_stat_rx_hdr_log` to log the Header information in the Header Log Register of CXL RAS Capability Structure when you assert the CXL RAS Statistics Uncorrectable input signal `cxl_ras_stat_rx_uncorr`. The `cxl_ras_stat_rx_hdr_log` signal is an N-bit input signal where N is the maximum supported Header width for a given CXL.cache/CXL.mem Message, as described in Table 3-30.

Table 3-30 Maximum Data Header Widths for CXL.cache/CXL.mem Message Headers

Data Message	Maximum Data Header Width (bits)
H2D Data Header	24
D2H Data Header	17
M2S Data Header	87
S2M Data Header	40

When CXL Channel interface data protection is enabled, PCIe RAS Debug, Error Injection, and Statistics gathering (RAS DES) feature is enabled for testing and debug of the CXL controller. For more information, see the *PCIe Controller Databook*.

Table 3-31 Mapping Between cxl_ras_stat_rx_* and MEMBAR0_RAS_*_ERROR_STATUS_REG_OFF

Signal Bit	Used for Reporting	*Register* Field Set
		MEMBAR0_RAS_CORR_ERROR_STATUS_REG_OFF
cxl_ras_stat_rx_corr[0]	Cache Data ECC Error Status	CXL_RCRB_MEMBAR0_CORR_ CACHE_DATA_ECC
cxl_ras_stat_rx_corr[1]	Mem Data ECC Error Status	CXL_RCRB_MEMBAR0_CORR_ MEM_DATA_ECC
		MEMBAR0_RAS_UNCOR_ERROR_STATUS_REG_OFF
cxl_ras_stat_rx_uncorr[0]	Cache Data Parity Error Severity	CXL_RCRB_MEMBAR0_CACHE_DATA_PAR
cxl_ras_stat_rx_uncorr[1]	Cache Address Parity Error Severity	CXL_RCRB_MEMBAR0_CACHE _ADD_PAR
cxl_ras_stat_rx_uncorr[2]	Cache Byte Enabled Parity Error Severity	CXL_RCRB_MEMBAR0_CACHE _BE_PAR
cxl_ras_stat_rx_uncorr[3]	Cache Data ECC Error Severity	CXL_RCRB_MEMBAR0_CACHE _DATA_ECC
cxl_ras_stat_rx_uncorr[4]	Mem Data Parity Error Severity	CXL_RCRB_MEMBAR0_MEM_D ATA_PAR
cxl_ras_stat_rx_uncorr[5]	Mem Address Parity Error Severity	CXL_RCRB_MEMBAR0_MEM_A DD_PAR
cxl_ras_stat_rx_uncorr[6]	Mem Byte Enabled Parity Error Severity	CXL_RCRB_MEMBAR0_MEM_BE_PAR
cxl_ras_stat_rx_uncorr[7]	Mem Data ECC Error Severity	CXL_RCRB_MEMBAR0_MEM_D ATA_ECC

3.7.2 CXL Registers

The CXL controller implements the listed registers as defined in *Compute Express Link Specification, Revision 3.0, Version 0.7*.

- `MEMBAR0_RAS_CORR_ERROR_STATUS_REG_OFF`
- `MEMBAR0_RAS_UNCOR_ERROR_STATUS_REG_OFF`
- `CXL_FATAL_ERR_CR_REG_OFF`

For more information, see the “Register Descriptions” chapter of the *Reference Manual*.

3.7.3 CXL Controller Viral Handling

The CXL controller has the following sources of Viral information

- Controller generated Uncorrected Internal Error (UIE)
- Viral notification in Retry.ack from the link partner
- Viral request from application through `cxl_viral_request` input signal

CXL Device

The CXL Device controller implements CXL Viral handling as defined in *Compute Express Link Specification, Revision 3.0, Version 0.7*. To enable the Viral feature in the CXL Device controller, set the VIRAL_EN of the CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF register to '1'. The controller logs the viral status in the VIRAL_STATUS field of the CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF register. The controller also maintains a port logic register CXL_PORT_LOGIC_VIRAL_STS_OFF which captures all the viral conditions.

Table 3-32 provides information about the events that trigger viral condition, the relevant status register fields that the controller updates to reflect viral status, and the action controller takes to notify the link partner.

Table 3-32 CXL Device Controller Viral Events

DVSEC Viral Enable[a]	Viral Source			DVSEC Viral Status[b]	Port Logic Viral Status[c]	Controller Viral Handling
	Controller UIE	cxl_viral_request[d]	Viral Notification in Retry.ack			
1	1	0	0	1	1	On UIE generation ■ The link layer forces a CRC error on the next outgoing flit. ■ If there is no traffic to send, the transmitter inserts an LLCRD flit with a CRC error.When the CXL controller receives Retry.Req as a part of the defined CRC error recovery flow, the CXL controller generates Retry.Ack control flit with embedded Viral Status information. The controller reports the corresponding error which triggered the Viral condition as Uncorrectable Internal Error (UIE) using AER.
1	0	1[e]	0	1	1	
X	0	0	1	0	1	When the CXL controller receives Retry.Ack control flit with Viral Status information, irrespective of the CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF.VIRAL_EN value, the controller sets the CXL_FATAL_ERR_RETRY_ACK_VIRAL field in the CXL_FATAL_ERR_CR_REG_OFF register and reports it as an Uncorrectable Fatal error using AER.
0	0	1	0	0	0	When CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF.VIRAL_EN is '0', the controller does not take any action.
0	1	0	0	0	0	When CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF.VIRAL_EN is '0', the controller does not take any action.

a. The value in the VIRAL_EN of the CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF register.
b. The value in the VIRAL_STATUS field of the CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF register.
c. The value in the CXL_PL_VIRAL_STATUS field of the CXL_PORT_LOGIC_VIRAL_STS_OFF register. The controller drives this value to the output signal cxl_viral_status.
d. The controller sets the CXL_AP_VIRAL_REQ field of the CXL_FATAL_ERR_CR_REG_OFF register when the application asserts cxl_viral_request.

e. The controller considers cxl_viral_request as UIE.

CXL Host

The Viral feature is always enabled for the CXL Host controller. The controller captures the viral status in the CXL_PL_VIRAL_STATUS field of the port logic register CXL_PORT_LOGIC_VIRAL_STS_OFF.

Table 3-33 provides information about the events that trigger viral condition, the relevant status register fields that the controller updates to reflect viral status, and the action the controller takes to notify the link partner.

Table 3-33 CXL Host Controller Viral Events

Viral Enable	Viral Source			Port Logic Viral Status[a]	Controller Viral Handling
	Controller UIE	cxl_viral_request[b]	Viral Notification in Retry.ack		
1	1	0	0	1	On UIE generation ■ The link layer forces a CRC error on the next outgoing flit. ■ If there is no traffic to send, the transmitter inserts an LLCRD flit with a CRC error.When the CXL controller receives Retry.Req as a part of the defined CRC error recovery flow, the CXL controller generates Retry.Ack control flit with embedded Viral Status information. The controller reports the corresponding error which triggered the Viral condition as Uncorrectable Internal Error (UIE) using AER.
1	0	1[c]	0	1	
1	0	0	1	1	When the CXL controller receives Retry.Ack control flit with Viral Status information, the controller sets the CXL_FATAL_ERR_RETRY_ACK_VIRAL field in the CXL_FATAL_ERR_CR_REG_OFF register and reports it as an Uncorrectable Fatal error using AER.

a. The controller drives the value of the CXL_PL_VIRAL_STATUS field of the CXL_PORT_LOGIC_VIRAL_STS_OFF register to the output signal cxl_viral_status.
b. The controller sets the CXL_AP_VIRAL_REQ field of the CXL_FATAL_ERR_CR_REG_OFF register when the application asserts cxl_viral_request.
c. The controller considers cxl_viral_request as UIE.

CXL Switch

The CXL Switch controller implements CXL Viral handling as defined in *Compute Express Link Specification Revision 2.0*. To enable the Viral feature in the CXL Switch controller, set the CXL_2_0_VIRAL_ENABLE of the CXL_2_0_CTRL_ALT_BUS_BASE_LIMIT_OFF register to '1'. The controller logs the viral status in the CXL_2_0_VIRAL_STATUS field of the CXL_2_0_EXT_DVSEC_HDR_2_OFF register. The controller also maintains a port logic register CXL_PORT_LOGIC_VIRAL_STS_OFF which captures all the viral conditions.

Table 3-34 provides information about the events that trigger viral condition, the relevant status register fields that the controller updates to reflect viral status, and the action controller takes to notify the link partner.

Table 3-34 CXL Switch Controller Viral Events

Extension DVSEC Viral Enable[a]	Viral Source			Extension DVSEC Viral Status[b]	Port Logic Viral Status[c]	Controller Viral Handling
	Controller UIE	cxl_viral_request[d]	Viral Notification in Retry.ack			
1	1	0	0	1	1	On controller UIE generation ■ The link layer forces a CRC error on the next outgoing flit. ■ If there is no traffic to send, the transmitter inserts an LLCRD flit with a CRC error.When the CXL controller receives Retry.Req as a part of the defined CRC error recovery flow, the CXL controller generates Retry.Ack control flit with embedded Viral Status information. The controller reports the corresponding error which triggered the Viral condition as Uncorrectable Internal Error (UIE) using AER.
X	0	0	1	0	1	When the CXL controller receives Retry.Ack control flit with Viral Status information, Irrespective of the CXL_2_0_CTRL_ALT_BUS_BASE_LIMIT_OFF.CXL_2_0_VIRAL_ENABLEvalue,the controller sets the CXL_PL_VIRAL_STATUS field of the CXL_PORT_LOGIC_VIRAL_STS_OFF register to ‘1’.
0	1	0	0	0	0	When CXL_2_0_CTRL_ALT_BUS_BASE_LIMIT_OFF.CXL_2_0_VIRAL_ENABLE is ‘0’, the controller does not take any action.
X	0	1	0	0	1	When the controller receives a UIE from the application ■ The link layer forces a CRC error on the next outgoing flit. ■ If there is no traffic to send, the transmitter inserts an LLCRD flit with a CRC error.When the CXL controller receives Retry.Req as a part of the defined CRC error recovery flow, the CXL controller generates Retry.Ack control flit with embedded Viral Status information. The controller reports the corresponding error which triggered the Viral condition as Uncorrectable Internal Error (UIE) using AER.

a. The value in the CXL_2_0_VIRAL_ENABLE field of the CXL_2_0_CTRL_ALT_BUS_BASE_LIMIT_OFF register.
b. The value in the CXL_2_0_VIRAL_STATUS field of CXL_2_0_EXT_DVSEC_HDR_2_OFF register.
c. The controller drives the value of the CXL_PL_VIRAL_STATUS field of the CXL_PORT_LOGIC_VIRAL_STS_OFF register to the output signal cxl_viral_status.
d. The controller considers cxl_viral_request as a viral notification from the application. You must connect cxl_viral_status of USP to the cxl_viral_request of DSP and vice-versa to ensure forwarding of Viral notification in both directions between a Host and a Device through the Switch.

MLD Capable CXL Controller

The output signal `cxl_viral_ldid` indicates the Viral LD-ID status. The controller logs the Viral LD-ID status in the `CXL_PL_VIRAL_LDID_STS` field of the `CXL_PORT_LOGIC_VIRAL_STS_OFF` register. The input signal `cxl_viral_ldid_request` signal is validated by `cxl_viral_request` and the output signal `cxl_viral_ldid` is validated by `cxl_viral_status`. CXS configuration does not support Viral LD-ID.

The controller sets the `CXL_PL_VIRAL_LDID_STS` field of the `CXL_PORT_LOGIC_VIRAL_STS_OFF` register with the latest Viral LD-ID vector depending on the viral source as follows:

- If the viral source is `cxl_viral_request`, the CXL controller sets it based on `cxl_viral_ldid_request`.
- If the viral source is Viral notification in Retry.ack, the controller sets it based on the Viral LD-ID field in the Retry.ack.
- If the viral source is controller generated UIE, the controller sets it to 16'FFFFh.

3.7.4 CXL Controller Data Poisoning

The CXL controller supports data poisoning when you enable data protection for CXL Message Channel Interface (that is, `CX_CXL_CHI_DATACHK >0`). Any error on Data or Byte Enable for data messages sets the Poison bit in the corresponding Data Message Header indicating poisoning of the message data.

On the Transmit path, any error detected in the Message Header field is reported and escalated through PCIe RAS DP Capability Register. Uncorrectable/Correctable error handling follows PCIe controller procedure by either causing the link down or entering RASDP Error Mode, depending on value of `AUTO_LINK_DOWN_EN` field of the `RASDP_ERROR_MODE_EN_OFF` register. For more information, see the "Reliability, Availability, and Serviceability (RAS)" section in the *PCIe Controller Databook*.

On the Receive path, CXL controller detects and reports poison detection on data messages using either CXL Correctable or Uncorrectable Error Status Register.

3.7.5 Event Counter Data Details

The following table provides the Group/Event number that is required when using the `EVENT_COUNTER_CONTROL_REG` port control register to read each of the event counters for CXL. The Group# is set by `EVENT_COUNTER_EVENT_SELECTION` and the Event# is set by `EVENT_COUNTER_EVENT_SELECTION.`

Table 3-35 Event Counter Group #8 (8-Bit CXL Error Counter Common-Lane)

Event#	Description	Note
0x00	ALMP Replication Error	L0 to Recovery transition trigger
0x01	Unexpected ALMP error	L0 to Recovery transition trigger
0x02	NUM_RETRY reached MAX_NUM_RETRY	L0 to Recovery transition trigger

3.7.6 CXL Controller Error Injection

RAS Testing is dependent on the ability to inject and correctly detect the injected errors. CXL controller implements the error injection capabilities using CXL.cache/CXL.mem Link Layer Error Injection registers to support RAS testing. For details, see the `CXL_CACHE_MEM_LL_INJ_OFF` register in the *Reference Manual*.

PCIe RAS Debug, Error Injection, and Statistics gathering (RAS DES) feature supports error injection on CXL controller internal datapath and RAMs. Some error injection features, like address parity error injection, might not be supported for RAMs associated with Channel Interfaces running at double speed of `core_clk`. For more information, see the *PCIe Controller Databook*.

3.8 Single Root I/O Virtualization (SR-IOV)

The CXL Device controller supports SR-IOV features when the parameter `CX_SRIOV_ENABLE` is enabled. The SR-IOV capability is supported only by the CXL Device RCiEP and not by the RCRB. Once the link is up in CXL mode, the physical function of an SR-IOV enabled CXL Device controller supports all the virtual functions, as in an SR-IOV enabled PCIe controller Upstream Port. The virtual function access mechanism is the same as that in a PCIe controller Upstream Port.

For more information on accessing CXL RCRB and RCRB MEMBAR0 regions using the DBI when SR-IOV is enabled, see "DBI Access Mechanism for the RCRB, RCRB MEMBAR0, and ELBI2 Regions" on page 79.

3.9 CXL Reset

3.9.1 Features and Limitations

- CXL Reset resets the non-sticky fields of all the CXL DVSECs and CXL MEMBAR0 as per CXL 2.0 specification.
- It does not reset the lock bits and fields locked using the lock bits.
- All the sticky registers retain their value upon CXL Reset.

The CXL DVSECs does not reset in case of an FLR.

3.9.2 CXL Reset Mechanism

CXL Reset mechanism initiates a Function Level Reset of all the CXL enabled Functions based on Non-CXL Function Map DVSEC. To enable CXL Reset mechanism, you must set the parameters `CX_FLR_ENABLE` and `CX_CXL_RESET_EN` to '1'. The steps to perform CXL Reset are as follows:

1. To initiate CXL Reset, set `INIT_CXL_RST` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register to '1'.
2. The controller asserts the output signal `cfg_cxl_dev_initiate_cxl_rst` to indicate that it has initiated CXL Reset.

   You can monitor the output signal `cfg_cxl_dev_cxl_rst_mem_clr_enable` to clear or randomize volatile HDM ranges during CXL Reset.
3. Your application asserts the input signal `app_cxl_rst_active` (each bit of the app_cxl_rst_active signal corresponds to one Function) to indicate the CXL Functions to be reset based on the non-CXL Function Map DVSEC. This signal must be held until the controller asserts `cfg_flr_pf_active`.
4. The controller asserts the `cfg_flr_pf_active` (each bit of the `cfg_flr_pf_active` signal corresponds to one Function) to indicate that CXL Reset is in progress.
5. Your application terminates TLP transmission from the application logic of the Functions that are reset (if the driver has not already terminated all the traffic) and asserts `app_flr_pf_done.`

The receive application logic of the Functions that are reset must not halt the receive interface for TLPs.

6. The controller initiates FLR of the Functions based on the input signal `app_flr_pf_done` (each bit of the `app_flr_pf_done` signal corresponds to one Function).
7. The controller deasserts `cfg_flr_pf_active` when the CXL Reset is complete.
8. You must monitor the output signal `cfg_flr_pf_active (each bit of the cfg_flr_pf_active` signal corresponds to one Function) to ensure that FLR of all the CXL Functions is complete. After the FLR of all the CXL Functions is complete, assert app_cxl_rst_done.
9. The controller samples the input signal `app_cxl_rst_done` and sets the `CXL_RST_CMPL` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register to '1'.

10. If CXL Reset completes with errors, you must assert the input signal `app_cxl_rst_error`.

    The controller sets the `CXL_RST_ERROR` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register based on the signal `app_cxl_rst_error`.

3.10 Cache Management

To enable Disable Caching, Cache Invalid, and Initiate Cache WriteBack and Invalidation functionality in a CXL Device, you must set the parameter `CX_CXL_CACHE_WR_INVLD_EN` to '1'.

3.10.1 Device Disable Caching

The Device Disable Caching process is as follows (for more information, see Figure 3-36):

1. To initiate Device Disable Caching, the CXL Host must write '1' to the `DISABLE_CACHE` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register in the CXL Device.
2. The CXL Device controller asserts the output signal `cfg_cxl_dev_disable_caching`.
3. The CXL Device application acknowledges cache invalid state by asserting the input signal `cxl_dev_cache_invalid`.
4. The CXL Device controller sets `CACHE_INVALID` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register to '1' when both `cxl_dev_cache_invalid` and `cfg_cxl_dev_disable_caching` are asserted.
5. The CXL Host clears the `DISABLE_CACHE` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register in the CXL Device.
6. The CXL Device controller deasserts the output signal `cfg_cxl_dev_disable_caching` to indicate that Device Disable Caching is complete.
7. The CXL Device controller clears the `CACHE_INVALID` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register when either `cxl_dev_cache_invalid` or `cfg_cxl_dev_disable_caching` is de-asserted.

Figure 3-35 CXL Device Controller and CXL Device Application Interaction

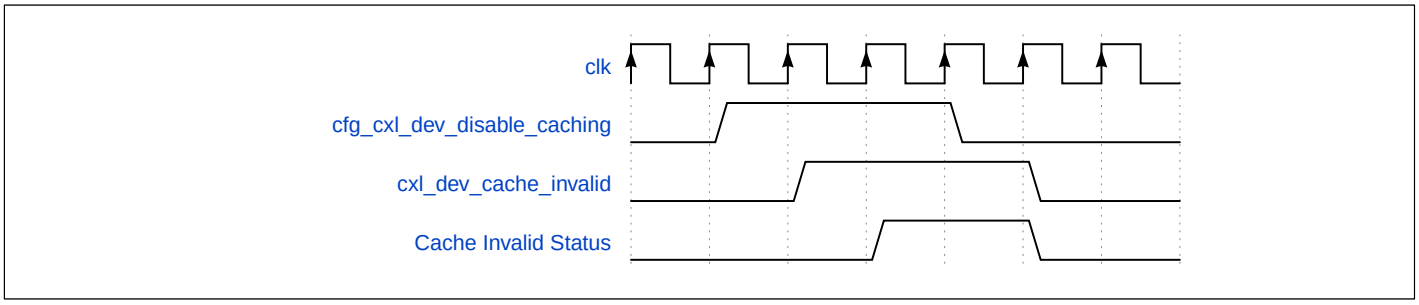
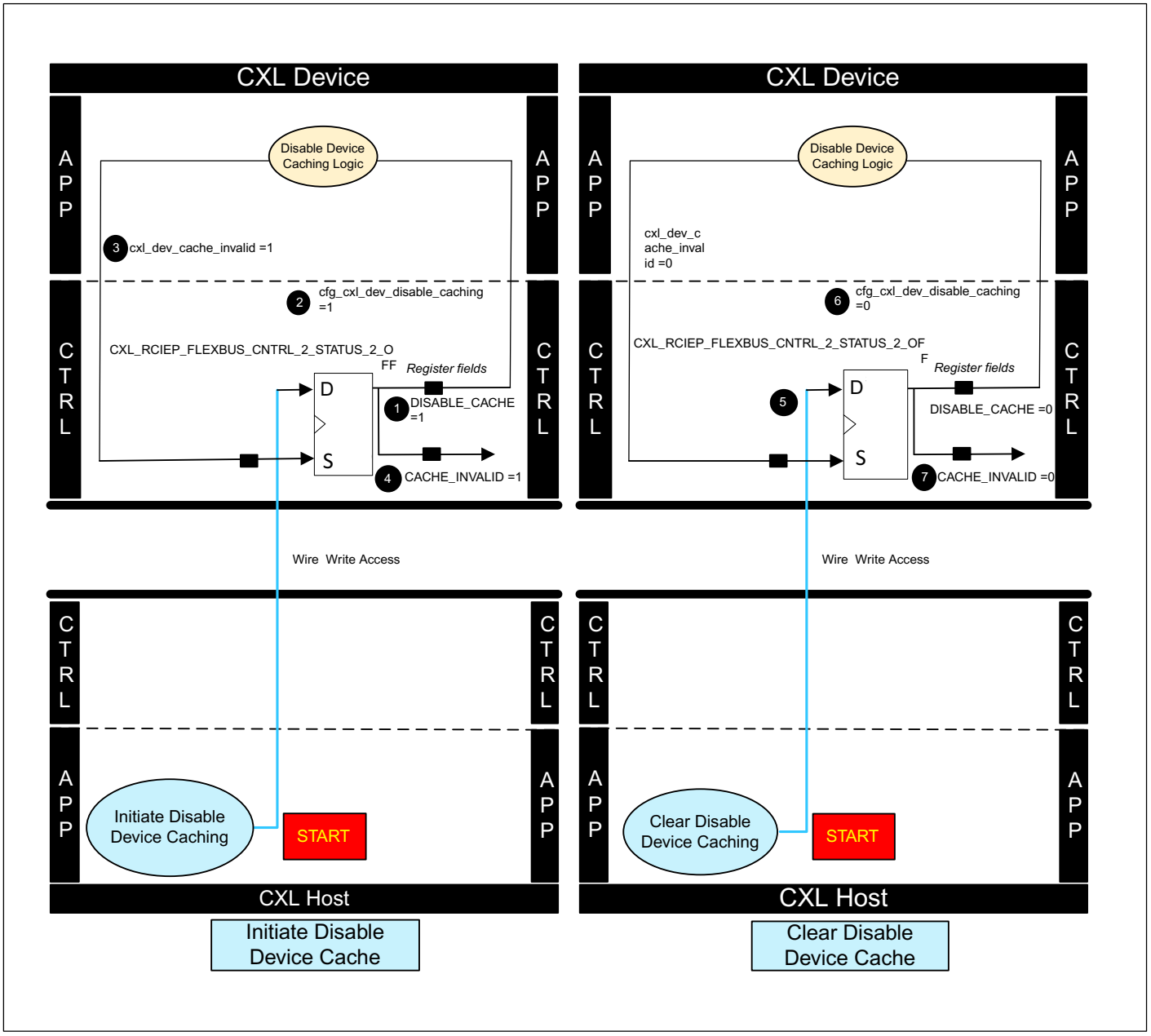


Figure 3-36 Device Disable Caching Process



3.10.2 Device Cache WriteBack and Invalidate

The Device Cache WriteBack and Invalidate process is as follows (for more information, see Figure 3-37):

1. To initiate Device Cache WriteBack and Invalidate, the CXL Host must write '1' to the `INIT_CACHE_WR_INVALID` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register in the CXL Device.
2. The CXL Device controller asserts the output signal `cfg_cxl_dev_initiate_cache_wr_invld`.
3. The CXL Device application acknowledges by asserting `cxl_dev_cache_wr_invld_cmpl` input of the CXL Device controller.

4. The CXL Device controller deasserts `cfg_cxl_dev_initiate_cache_wr_invld.`
5. The CXL Host waits for "CacheFlushed" message from the CXL Device to conclude that "Device Write-Back and Invalidate" is complete.

Figure 3-37 Device Cache WriteBack and Invalidate Process

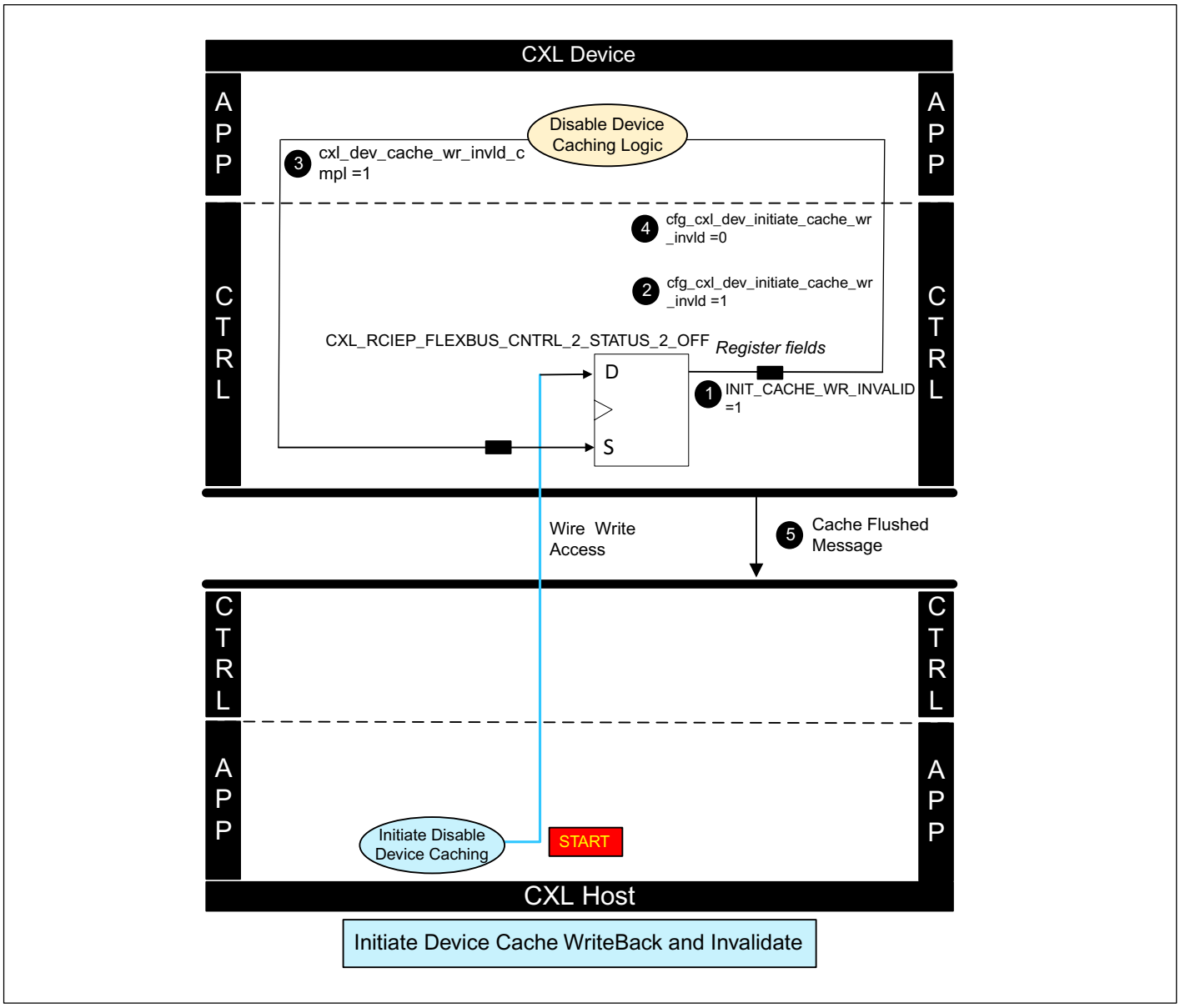
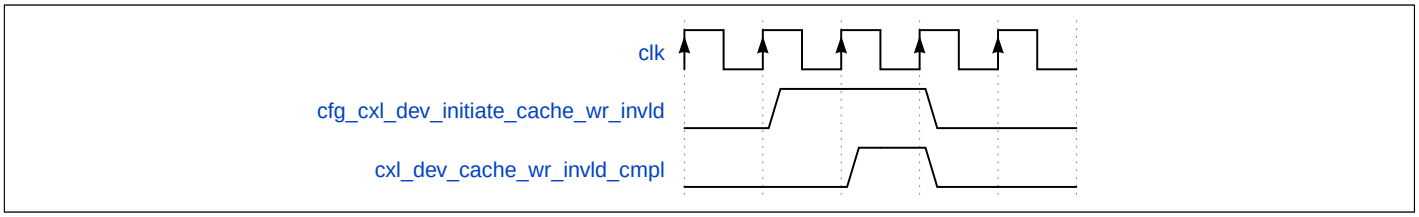


Figure 3-38 CXL Device Controller and CXL Device Application Interaction



3.11 Power Management Initialization Completion Reporting

The section describes the power management initialization completion status in the following cases:

- "Device Power Management Initialization Completion Reporting"
- "Root Port Power Management Initialization Completion Reporting"

3.11.1 Device Power Management Initialization Completion Reporting

By default, the CXL Device controller power management reporting is enabled for a CXL 2.0 Device controller.

CXL Device application can use the input signal `cxl_pm_init_cmpl` to indicate that the CXL Device has successfully completed Power Management Initialization Flow.The CXL Device controller registers the value of this signal in the `PWR_INIT_CMPL` field of the `CXL_RCIEP_FLEXBUS_CNTRL_2_STATUS_2_OFF` register.

3.11.2 Root Port Power Management Initialization Completion Reporting

The Host application can use the input signal `cxl_pm_init_cmpl` to indicate that the CXL Host has successfully completed Power Management Initialization Flow. The CXL Host controller registers the value of this signal in the `CXL_2_0_PM_INIT_CMPL` field of the `CXL_2_0_EXT_DVSEC_HDR_2_OFF` register.

4

Power Management

This chapter describes the Power Management feature implemented in the DesignWare Cores Compute Express Link (CXL) controller. The following topics are discussed:

- "Overview"
- "CXL PM Architecture"
- "Controller Operations"

4.1 Overview

The CXL controller supports the Physical Layer Power management as defined in the CXL Specification.

4.1.1 Features

- Compliant with *Compute Express Link Specification Revision 2.0*
- ASPM L1 support for CXL 1.1 and CXL 2.0 devices
- PCI-PM support for CXL 2.0 device
- Dedicated PM control over CXL.cache/CXL.mem through Native channel interface.

CXL mode does not support L1 substate and P1.CPM. It is disabled when the controller enters CXL mode.

4.2 CXL PM Architecture

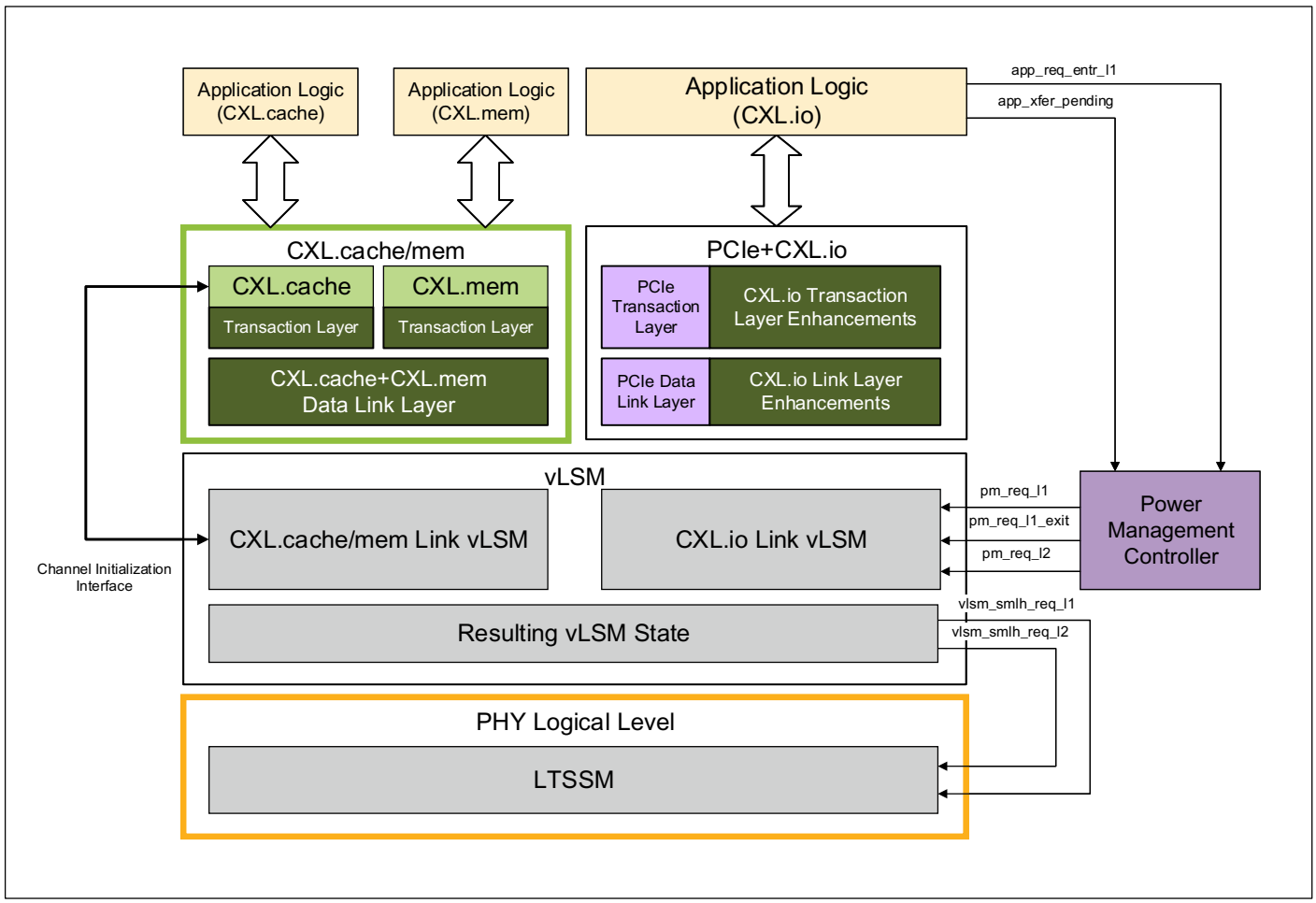
PM controller is responsible for CXL.io protocol specific mechanism to negotiate entry into PM state such as ASPM, as discussed in "CXL.io Link Power Management" on page 131.

The application implements CXL.cache/CXL.mem controller protocol specific mechanism through CXL controller transaction layer interface, discussed in "CXL.cache/CXL.mem Link Power Management" on page 110. The application must de-initialize CXL controller channel interfaces to request/acknowledge an entry into PM state.

Link vLSMs coordinate an entry and exit from Physical Layer PM states on both sides of the link using ALMPs. When both CXL.cache/CXL.mem and CXL.io Link vLSMs negotiate an entry to PM state, vLSM directs LTSSM to enter corresponding PM state.

Figure 4-1 shows the CXL PM data flow diagram.

Figure 4-1 CXL PM Data Flow Diagram



4.3 Controller Operations

This section describes the Link Power Management feature for the CXL controller. The following topics are discussed:

- "CXL.cache/CXL.mem Link Power Management"
- "CXL.io Link Power Management"

4.3.1 CXL.cache/CXL.mem Link Power Management

CXL.cache/CXL.mem support Active Link State Power Management only. PM Entry handshake is not defined between the Link Layers. ARB/MUX initiates the PM Entry on the Downstream Component. The transaction layer directs each side of the link to request ARB/MUX to enter L1 state independently. The ARB/MUX layers on both sides of the Link coordinate the entry into PM state using ALMPs. The succeeding sections define the process CXL.cache/CXL.mem Link Power Management follow for PM entry and exit.

4.3.1.1 CXL.cache/CXL.mem PM Entry and Exit Conditions

CXL.cache/CXL.mem PM Entry is requested if one of the following events is triggered:

- Application requests L1 Entry by de-asserting both `cxl_cache_slv_req` and `cxl_mem_slv_req` in ACTIVE state after TX channel interface initialization was acknowledge by CXL controller (L1 Entry).
- Application is not asserting `cxl_cache_slv_req` or `cxl_mem_slv_req` on transition to ACTIVE state while acknowledging RX channel interface initialization request from CXL controller (L1 Auto-Entry).
- PM controller requests L2 Entry.

CXL.cache/CXL.mem PM Exit is requested if one of the following events is triggered:

- Application requests L1 Exit by asserting either `cxl_cache_slv_req` or `cxl_mem_slv_req`.
- Retry buffer is not empty after L1 negotiation is complete.

4.3.1.2 CXL.cache/CXL.mem L1 Entry

CXL controller is required to de-initialize the CXL RX/TX channel interfaces on the transaction layer before entry into L1 state. The application must de-initialize the CXL transaction layer RX/TX channel interfaces before L1 Entry.

The Downstream component application deinitializes the Tx channel interface by de-asserting `cxl_[mem|cache]_slv_ack`[1] to initiate CXL.cache/CXL.mem L1 Entry.

On L1 Entry, the channel interface deinitialization occurs in TX-RX pairs.The Downstream component application deinitializes TX datapath before transmitting Request ALMP and RX datapath after receiving Status ALMP. Similarly, the Upstream component application deinitializes RX datapath when it receives Request ALMP and TX datapath before transmitting Status ALMP.

When CXL.cache and CXL.mem protocols are enabled, L1 Entry must be triggered and completed on TX and RX interfaces before the transition to the L1 state.

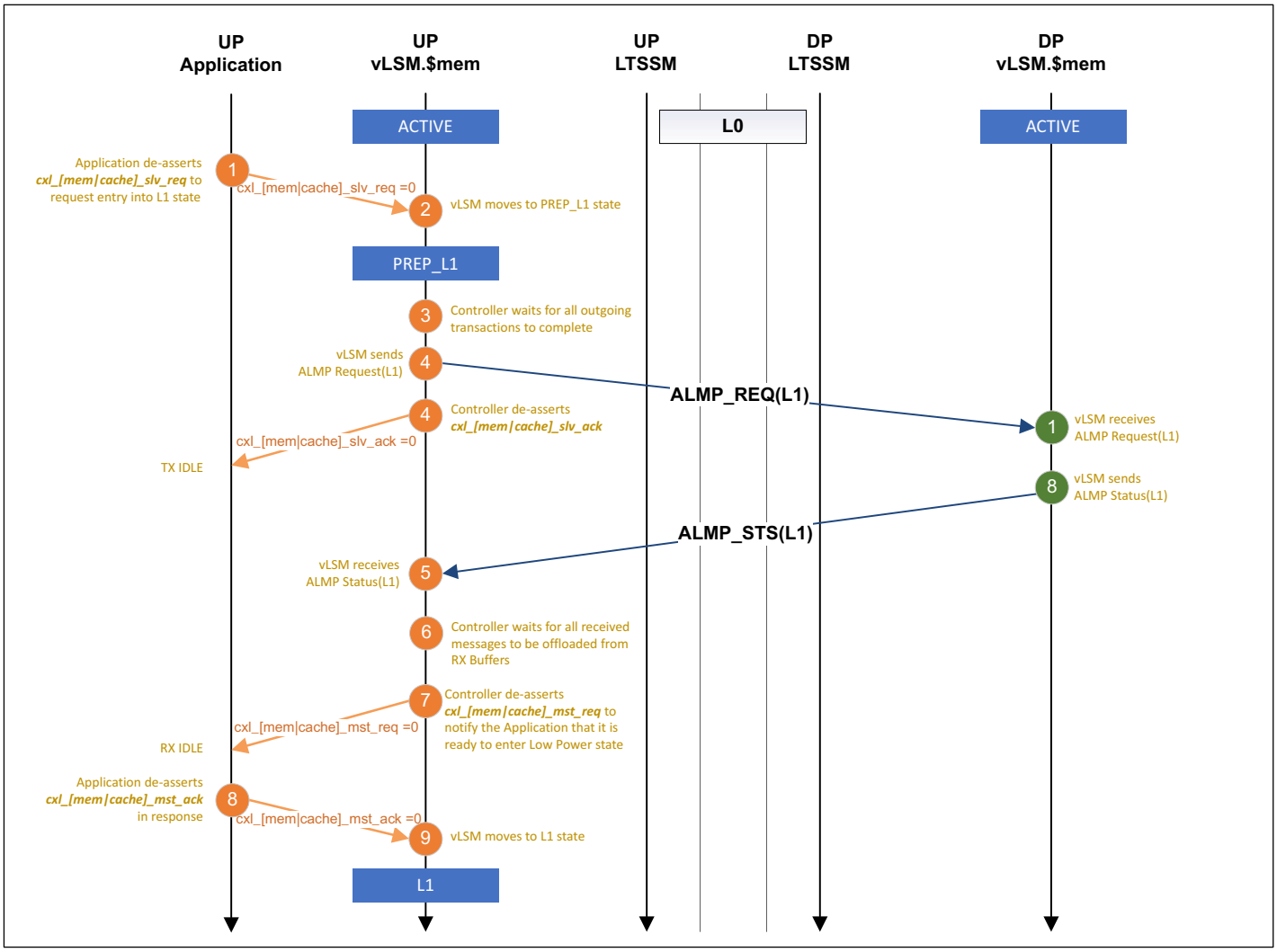
1. The signal name written as cxl_[mem|cache]_slv req in this chapter, represents that it can be either cxl_mem_slv_req or cxl_cache_slv_req.

The process description for L1 Entry for Downstream and Upstream components:

Downstream Component L1 Entry

Figure 4-2 illustrates L1 Entry data flow diagram for Downstream Component.

Figure 4-2 Downstream Component L1 Entry Data Flow Diagram



Downstream Component performs the following steps on entry to L1 state:

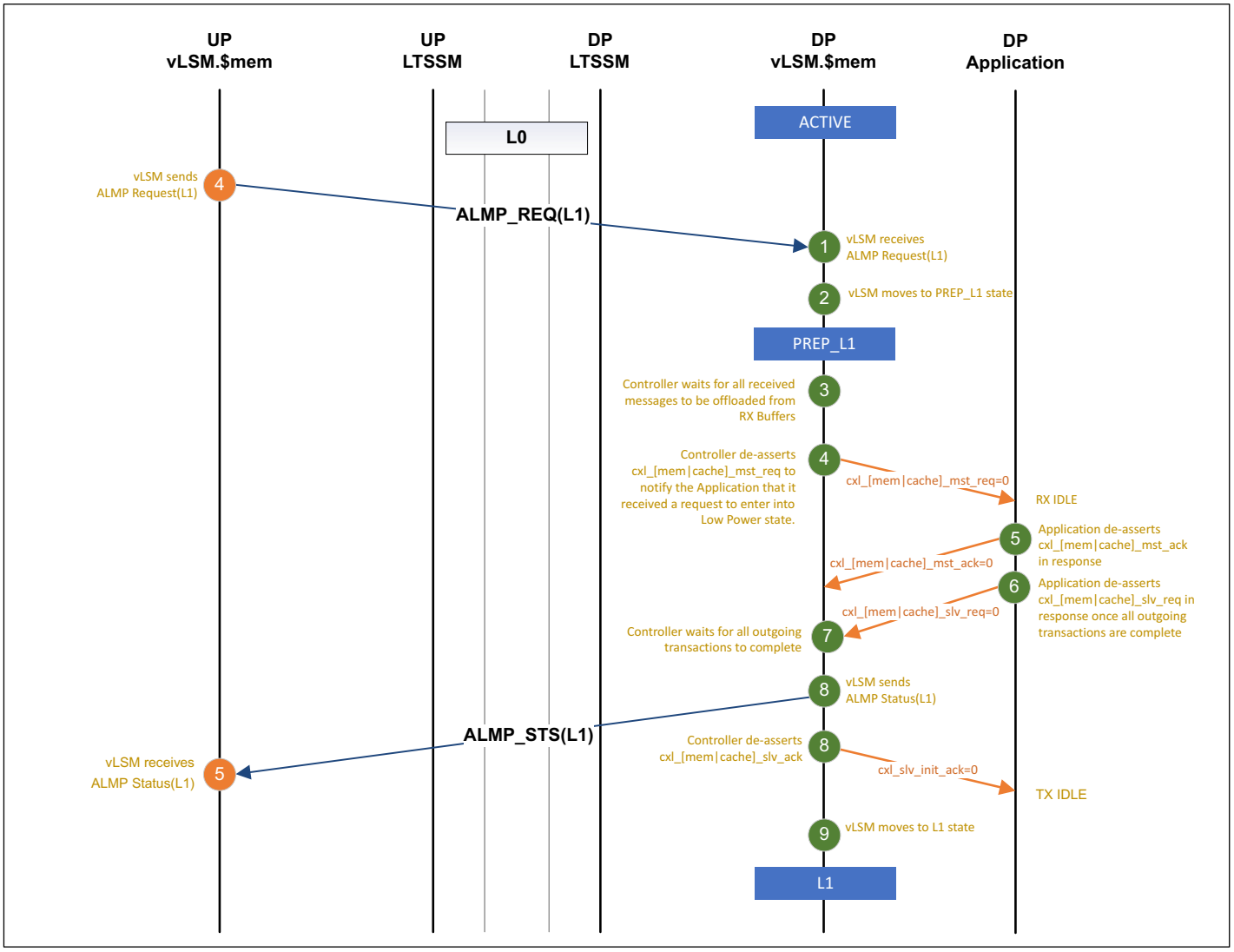
1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry.
2. In response, Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM waits for CXL controller to complete all outgoing transactions.
4. After all transactions are complete, CXL controller deasserts `cxl_[mem|cache]_slv_ack` initialization acknowledge on the TX channel interfaces, indicating Idle state of the CXL transmit datapath. Link vLSM sends Request L1 ALMP to communicate, the state of the link and waits to receive Status L1 ALMP in response.

5. When the application receives Status L1 ALMP, the controller waits for all received messages to be offloaded from RX buffers and CXL receive datapath to indicate Idle state.
6. The controller deasserts `cxl_[mem|cache]_mst_req` to request deinitialization of the RX channel interfaces and notify the application that it is ready for L1 Entry.
7. Application must deassert `cxl_[mem|cache]_mst_ack` on the RX channel interfaces in response.
8. Link vLSM transits to L1 state.

Upstream Component L1 Entry

Figure 4-3 illustrates the L1 Entry data flow diagram for Upstream Component.

Figure 4-3 Upstream Component L1 Entry Data-Flow Diagram



Upstream Component performs the following steps on entry to L1 state:

1. L1 Entry starts with Link vLSM observing Request L1 ALMP.
2. In response Link vLSM moves to the transitional state on Entry to L1.

3. After Request L1 ALMP is received, the controller waits for all received messages to be offloaded from RX buffers and CXL receive datapath to indicate Idle state.
4. The controller deasserts `cxl_[mem|cache]_mst_req` to request de-initialization of the RX channel interfaces and notify the Application that L1 Entry is requested.
5. Application must deassert `cxl_[mem|cache]_mst_ack` on the RX channel interfaces in response.
6. Application deasserts `cxl_[mem|cache]_slv_req` on the TX channel interfaces once all outgoing transactions are complete to acknowledge L1 Entry request[1].
7. CXL controller waits for all outgoing to complete to indicate CXL TX datapath is in Idle state.
8. Link vLSM sends Status L1 ALMP.

   CXL controller deasserts `cxl_[mem|cache]_slv_ack` to notify the completion of L1 entry to the Application.
9. Link vLSM transits L1 state.

4.3.1.3 CXL.cache/CXL.mem L1 Exit

The Application triggers the CXL.cache/CXL.mem L1 Exit requesting TX channel interfaces initialization on the either side of the link. In this case, the Application asserts `cxl_[mem|cache]_slv_req` on either CXL.cache/CXL.mem TX channel interfaces to wake up CXL controller. There are two possible scenarios when the application requests L1 Exit, depending on the state of the link:

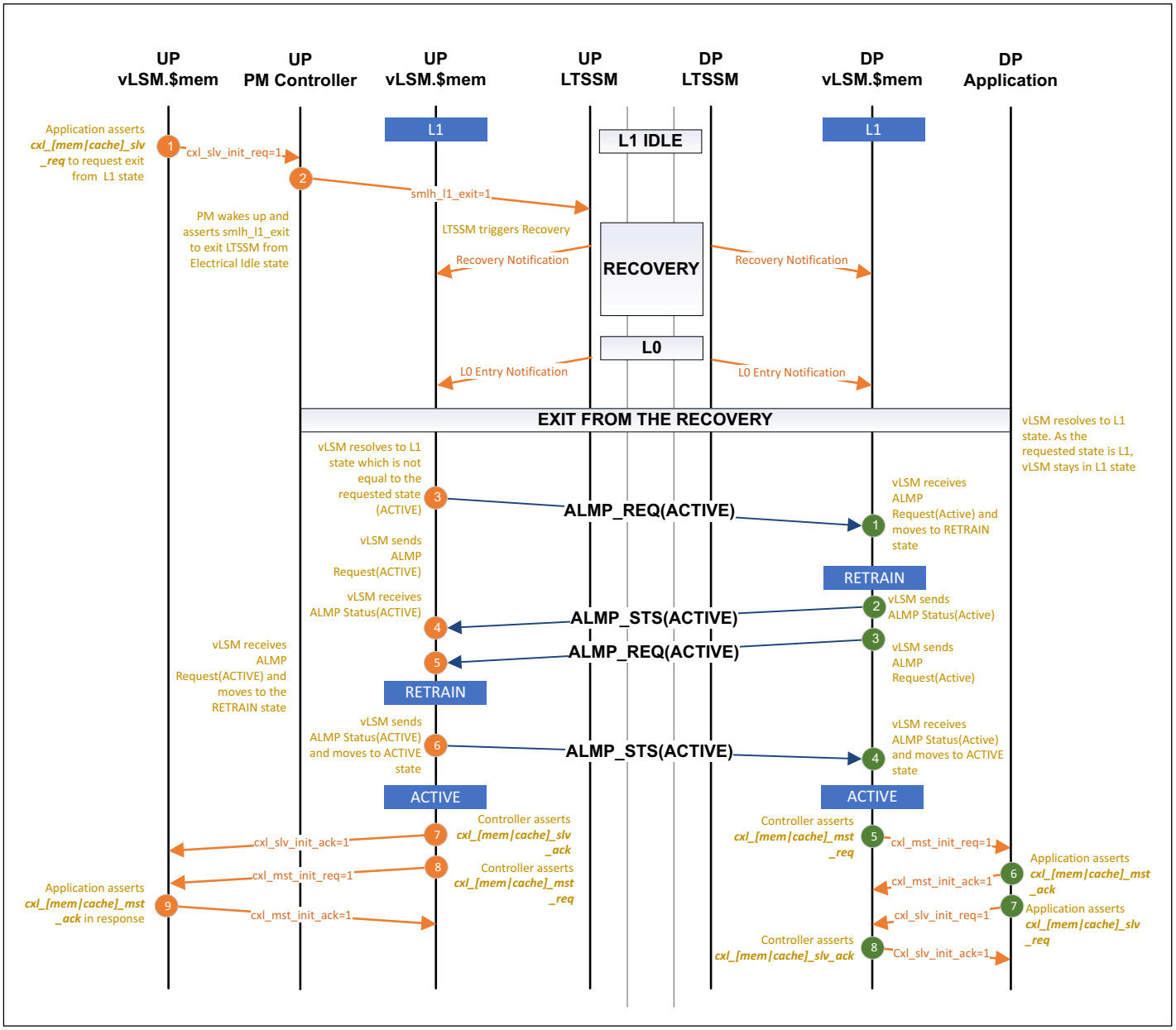
- Link is in L0 state
- Link is in L1 IDLE state

L1 Exit When Link is in L1 IDLE State

Figure 4-4 shows an example of L1 Exit scenario when the link is in L1 IDLE state. vLSM receives L1 Exit (Active Request) notification from the PM controller when the application asserts cxl_[mem|cache]_slv_req in L1 state, and the link enters to the Recovery. After an exit from the Recovery, each vLSM sends Status L1 ALMP to synchronize the state of the link. The state synchronization handshake resolves in L1 state, which does not match the Link Layer requested Active state. VLSM triggers a New Active Request and Status ALMP exchange handshake to transit Link vLSM to the Active state.

1. Upstream Component rejects L1 Entry. In this case, the Application does not deassert cxl_[mem|cache]_slv_req. The Downstream Component deassert either retries L1 Entry request or aborts L1 Entry

Figure 4-4 CXL.cache/CXL.mem L1 Exit Data Flow Diagram When Link is in L1 State



L1 Exit requester component performs the following steps to exit from L1:

1. The Application asserts `cxl_[mem|cache]_slv_req` on the TX channel interfaces to request L1 Exit.
2. After `cxl_[mem|cache]_slv_req` is observed in the L1 state, PM controller instructs LTSSM to exit from L1 IDLE state, which triggers an entry to the Recovery.
3. After State Synchronization on exit from the Recovery, vLSM resolves to L1 state after sending Status L1 ALMP and receiving Status L1 ALMP. As the resolved state (L1) does not match the requested state (ACTIVE), Link vLSM triggers Active State Request/Status handshake protocol and sends Request Active ALMP.
4. Link vLSM receives Status Active ALMP.

5. Link vLSM receives Request Active ALMP and transitions to RETRAIN state to exit from L1.
6. Link vLSM transmits Status Active ALMP and transitions to ACTIVE state.
7. CXL controller asserts `cxl_[mem|cache]_slv_ack` to acknowledge TX channel interfaces initialization request and activate TX datapath.
8. CXL controller asserts `cxl_[mem|cache]_mst_reqcxl_[mem|cache]_mst_req` to request RX channel interfaces initialization.
9. Application must assert `cxl_[mem|cache]_mst_ack` to acknowledge RX channel interfaces initialization request and complete RX datapath activation.

L1 Exit sub-ordinate component performs the following steps on exit from L1:

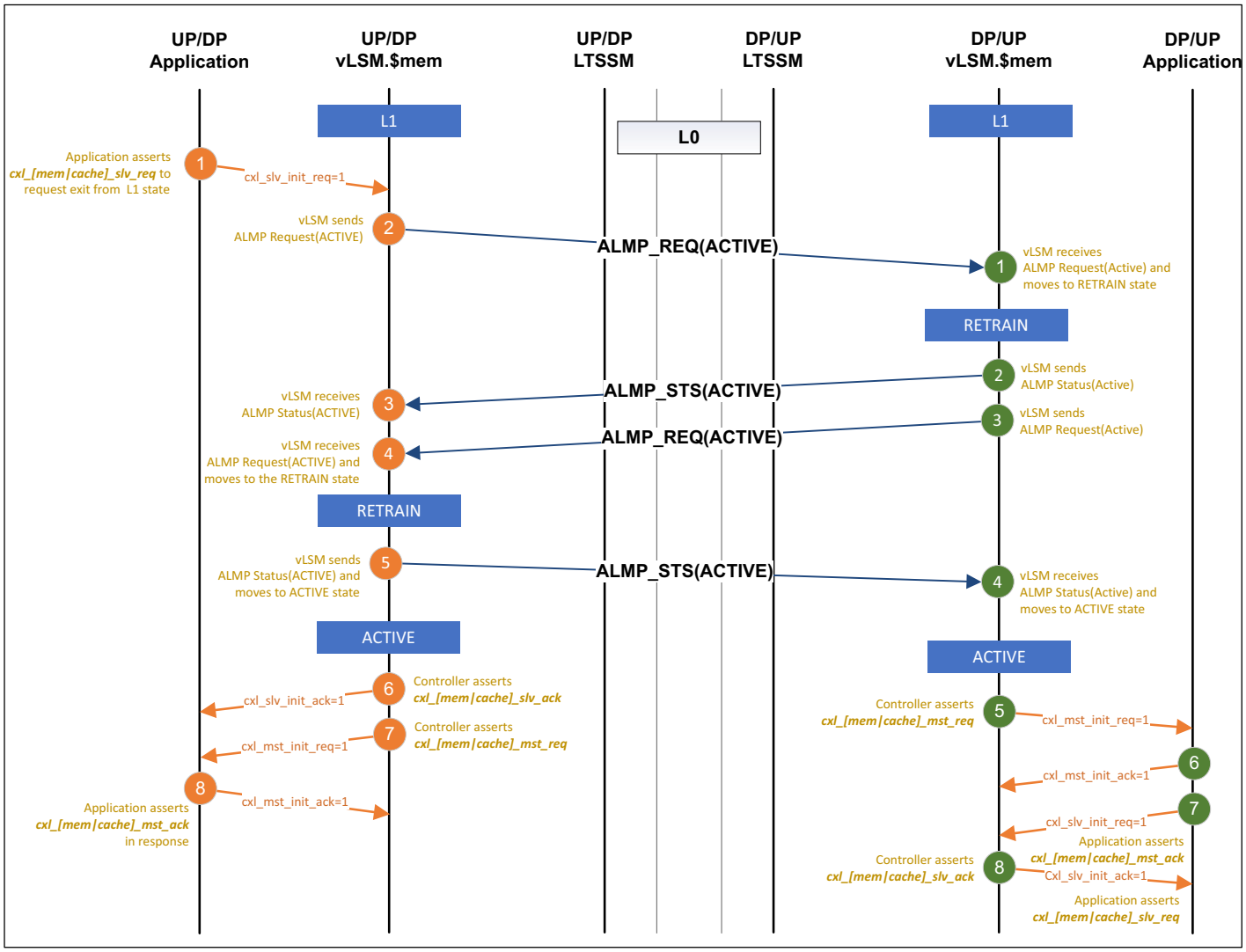
1. During State Synchronization on exit from the Recovery, vLSM resolves to L1 state after sending Status L1 ALMP and receiving Status L1 ALMP. After the Recovery, when vLSM receives Request L1 ALMP it transits to the RETRAIN state.
2. Link vLSM responds with Status Active ALMP.
3. Link vLSM sends Request Active ALMP.
4. Link vLSM receives Status Active ALMP in response and transitions to ACTIVE state.
5. CXL controller asserts `cxl_[mem|cache]_mst_req` to request RX channel interfaces initialization.
6. Application must assert `cxl_[mem|cache]_mst_ack` to acknowledge RX channel interfaces initialization request and complete RX datapath activation.
7. Application asserts `cxl_[mem|cache]_slv_req` to request TX channel interfaces initialization.
8. CXL controller asserts `cxl_[mem|cache]_slv_ack` to acknowledge TX channel interfaces initialization request and activate TX datapath.

L1 Exit When Link is in L0 State

Figure 4-5 shows an example of L1 Exit scenario when the link is in L0 state while CXL.cache/CXL.mem is in L1 state. The main difference with a previous example is that the link is not in the electrical idle state hence the physical link recovery is not required on exit from L1 and only vLSMs must go through the RETRAIN to exit from L1 to ACTIVE state.

In this scenario, the requester vLSM transmits Request Active ALMP when the application asserts `cxl_[mem|cache]_slv_req` in the L1 state. In response the sub-ordinate vLSM transits CXL.cache/CXL.mem Link vLSM to RETRAIN state to exit from L1 state and responds with Status Active ALMP followed by Request Active ALMP. The requester receives Request Active ALMP and exits to the RETRAIN state and, after transmitting Status Active ALMP in response, it completes an entry to the ACTIVE state. In its turn, the sub-ordinate vLSM completes an entry to the ACTIVE state, when it receives Status Active ALMP in response to the Request Active ALMP.

Figure 4-5 CXL.cache/CXL.mem L1 Exit Data-Flow Diagram When Link is in L0 State



L1 Exit requester component performs the following steps to exit from L1:

1. The Application asserts `cxl_[mem|cache]_slv_req` on the TX channel interfaces to request L1 Exit.
2. After `cxl_[mem|cache]_slv_req` is observed when link is in L0 state, Link vLSM sends Request Active ALMP.
3. Link vLSM receives Status Active ALMP.
4. Link vLSM receives Request Active ALMP and transitions to RETRAIN state to exit from L1.
5. Link vLSM transmits Status Active ALMP and transitions to ACTIVE state.
6. CXL controller asserts `cxl_[mem|cache]_slv_ack` to acknowledge TX channel interfaces initialization request and activate TX datapath.
7. CXL controller asserts `cxl_[mem|cache]_mst_req` to request RX channel interfaces initialization.
8. Application must assert `cxl_[mem|cache]_mst_ack` to acknowledge RX channel interfaces initialization request and complete RX datapath activation.

L1 Exit sub-ordinate component performs the following steps on exit from L1:

1. Link vLSM receives Request Active ALMP when Link vLSM is in L1 state and transitions to RETRAIN state.
2. If link is in L0 state, Link vLSM responds with Status Active ALMP.
3. Link vLSM sends Request Active ALMP.
4. Link vLSM receives Status Active ALMP in response and transitions to ACTIVE state.
5. CXL controller asserts `cxl_[mem|cache]_mst_req` to request RX channel interfaces initialization.
6. Application must assert `cxl_[mem|cache]_mst_ack` to acknowledge RX channel interfaces initialization request and complete RX datapath activation.
7. Application asserts `cxl_[mem|cache]_slv_req` to request TX channel interfaces initialization.
8. CXL controller asserts `cxl_[mem|cache]_slv_ack` to acknowledge TX channel interfaces initialization request and activate TX datapath.

4.3.1.4 CXL.cache/CXL.mem L1 Abort

After requesting L1 Entry, Downstream component might abort L1 Entry at any stage before Link vLSM transitions to L1 state. This can be broken down into following scenarios:

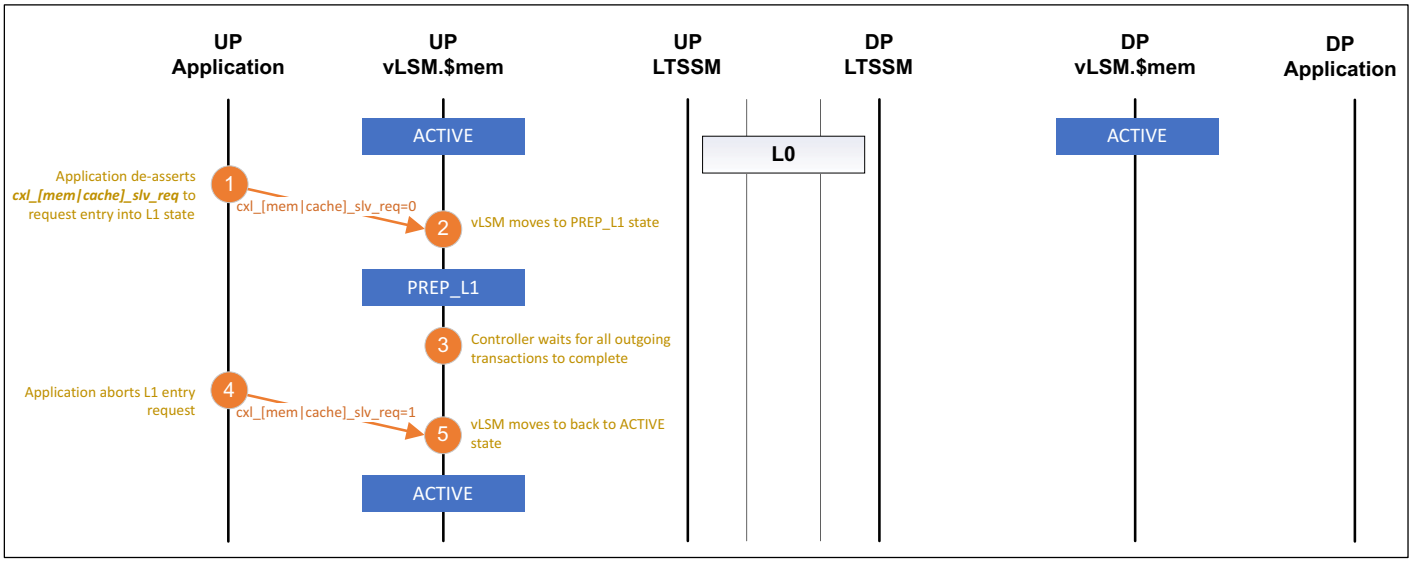
- Application aborts L1 Entry before Link vLSM sends Request L1 ALMP
- Application aborts L1 Entry after Link vLSM sends Request L1 ALMP before DP L1 acceptance
- Application aborts L1 Entry after Link vLSM sends Request L1 ALMP after DP L1 acceptance

The sub-sections below discuss these two scenarios in details.

L1 Abort Before Sending Request L1 ALMP

Figure 4-6 shows an example of L1 Abort scenario before CXL.cache/CXL.mem Link vLSM sends Request L1 ALMP. Application aborts L1 Entry by asserting cxl_[mem|cache]_slv_req in attempt to re-initialize TX channel interfaces. In this case, Link vLSM returns to the Active state from the transitional state on entry to L1 hence re-activating TX datapath. No further actions are required.

Figure 4-6 Downstream Component L1 Abort Before Request L1 ALMP



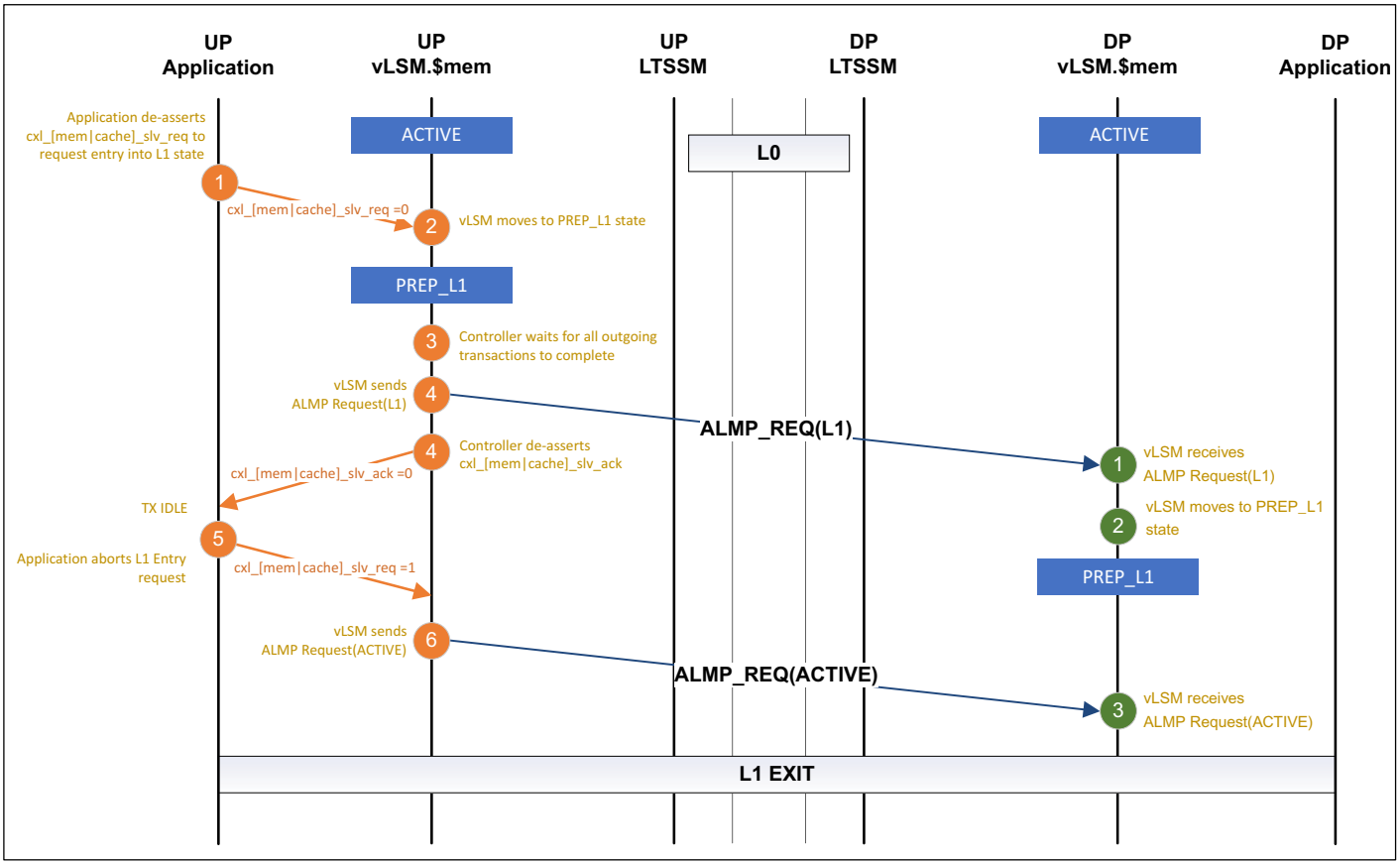
Downstream Component performs the following steps on L1 Abort request:

1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM waits for CXL controller to complete all outgoing transactions.
4. Application asserts `cxl_[mem|cache]_slv_req` to abort L1 Entry request.
5. Link vLSM transits back to the Active state if Request L1 ALMP is not sent.

L1 Abort After Sending Request L1 ALMP

Figure 4-7 shows an example of L1 Abort scenario after CXL.cache/CXL.mem Link vLSM sends Request L1 ALMP. This case is similar to L1 Exit scenario. Application aborts L1 Entry by asserting cxl_[mem|cache]_slv_req in attempt to re-initialize TX channel interfaces. Link vLSM might not return to the Active state if the Upstream Component Link vLSM is already in L1 state. Therefore, Downstream Component Link vLSM initiates the same sequence as for L1 Abort which starts with Link vLSM sending Request Active ALMP.

Figure 4-7 Downstream Component L1 Abort After Request L1 ALMP



Downstream Component performs the following steps on L1 Abort request:

1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM waits for CXL controller to complete all outgoing transactions.
4. After all transactions are complete, CXL controller deasserts `cxl_[mem|cache]_slv_ack` initialization acknowledge on the TX channel interfaces, indicating Idle state of the CXL transmit datapath. Link vLSM sends Request L1 ALMP to communicate the state of the link and waits to receive Status L1 ALMP in response.
5. The Application asserts `cxl_[mem|cache]_slv_req` on the TX channel interfaces to abort L1 Entry request.
6. Link vLSM sends Request Active ALMP and continues with L1 Exit scenario as described in "CXL.cache/CXL.mem L1 Exit" on page 113 if DP Physical Layer receives EIOS.

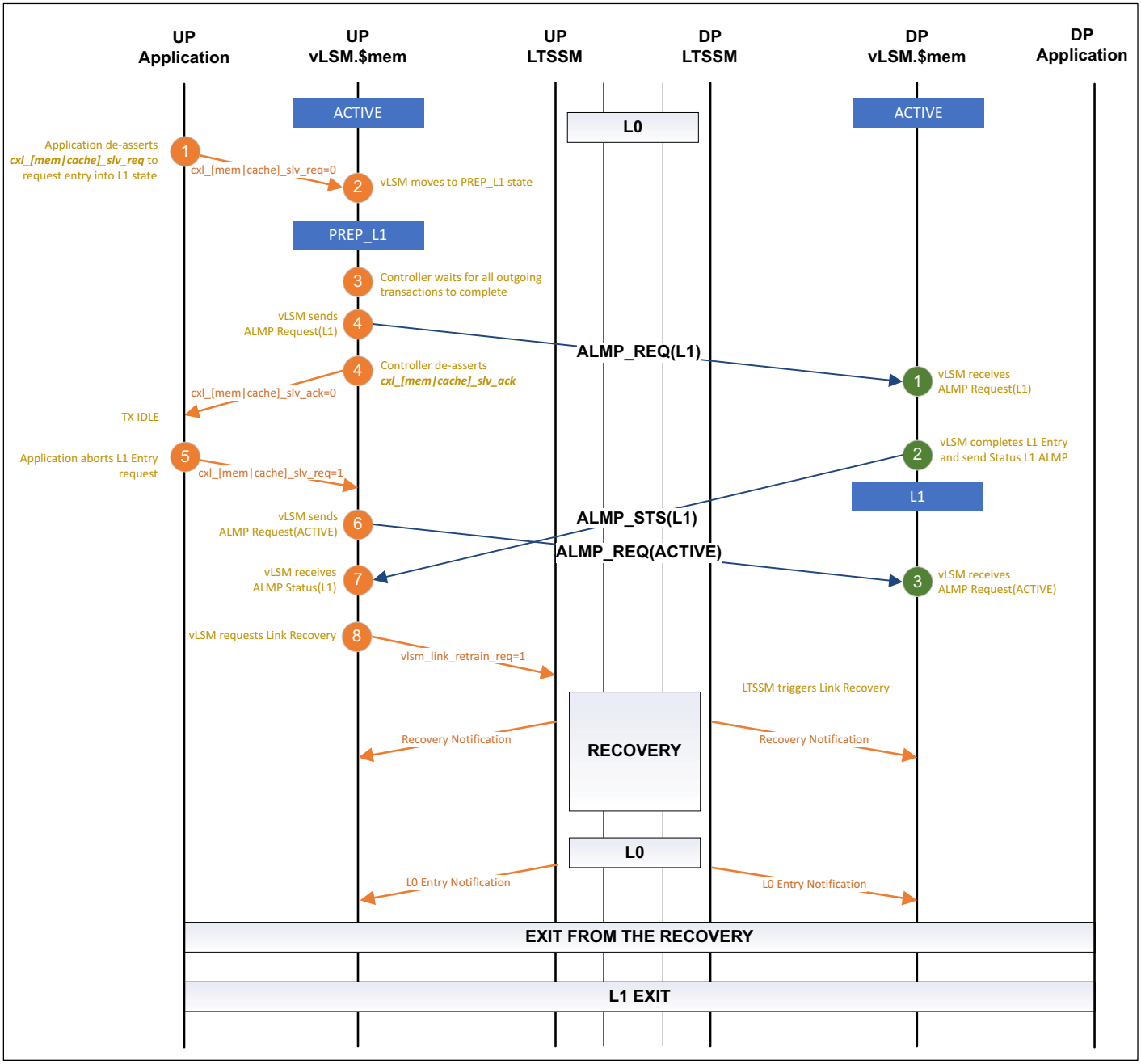
Upstream Component performs the following steps on L1 Abort request:

1. L1 Entry starts with Link vLSM observing Request L1 ALMP.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM receives Request Active ALMP at any stage before or after Link vLSM transition to L1, it continues with L1 Exit scenario as described in "CXL.cache/CXL.mem L1 Exit" on page 113 if DP Physical Layer receives EIOS.

L1 Abort After Downstream Port PM Acceptance

Figure 4-8 shows an example of L1 Abort scenario after CXL.cache/CXL.mem Link vLSM sends Request L1 ALMP but DP has already accepted L1 Entry by sending Status L1 ALMP. In this case, Application asserts cxl_[mem|cache]_slv_req and aborts L1 Entry in attempt to re-initialize TX channel interfaces. Downstream Component sends Request Active ALMP to abort requested L1 Entry while Upstream Component has already accepted L1 Entry by sending Status L1 ALMP. As a result, in response to the Request Active ALMP, Downstream Component receives Status L1 ALMP which is an unexpected ALMP and will trigger recovery. L1 Abort then continues as described in"CXL.cache/CXL.mem L1 Exit" on page 113.

Figure 4-8 Downstream Component L1 Abort After L1 Acceptance



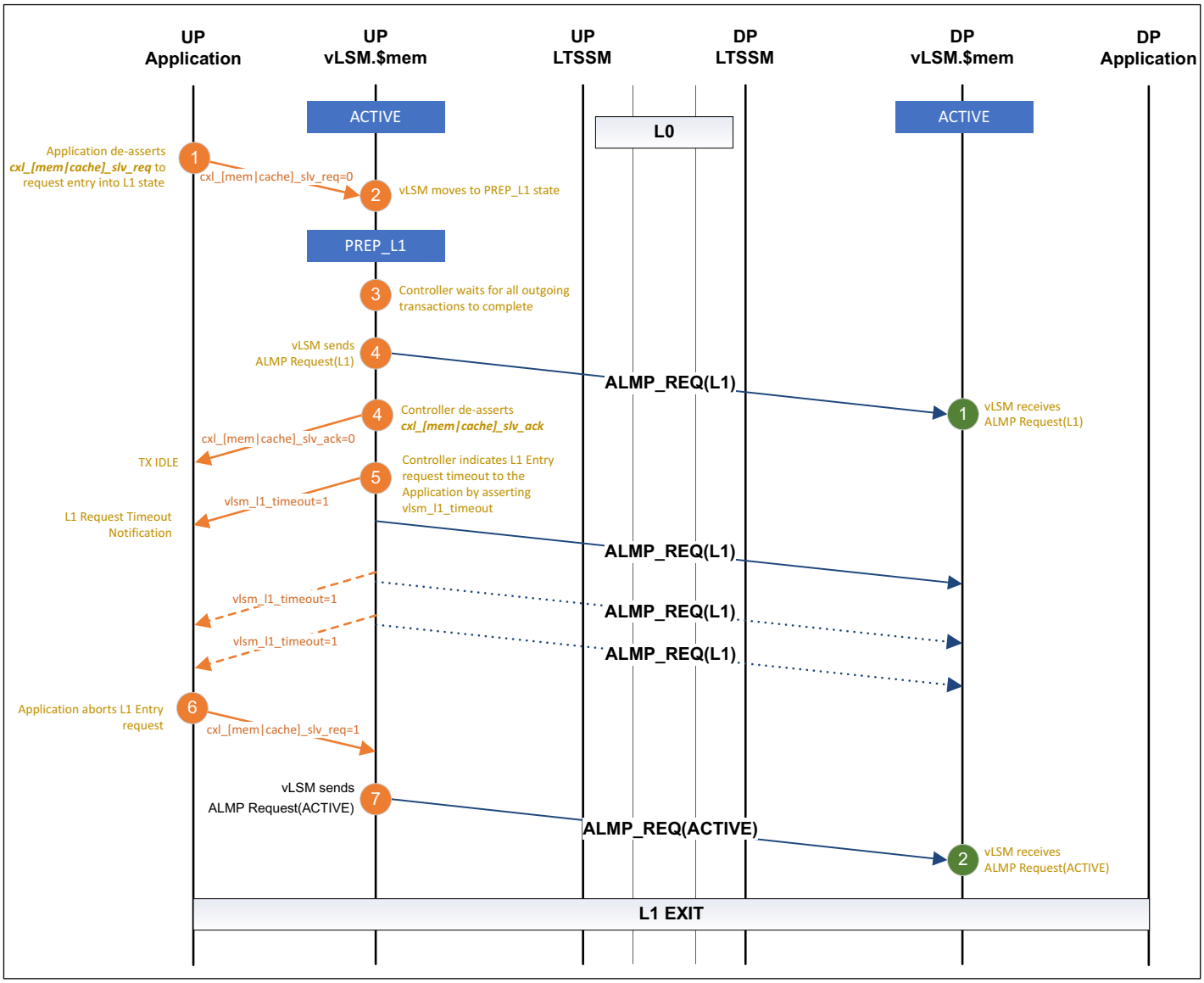
Downstream Component performs the following steps on L1 Abort request:

1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM waits for CXL controller to complete all outgoing transactions.
4. After all transactions are complete, CXL controller deasserts `cxl_[mem|cache]_slv_ack` initialization acknowledge on the TX channel interfaces, indicating Idle state of the CXL transmit datapath. Link vLSM sends Request L1 ALMP to communicate the state of the link and waits to receive Status L1 ALMP in response.
5. The Application asserts `cxl_[mem|cache]_slv_req` on the TX channel interfaces to abort L1 Entry request.
6. Link vLSM sends Request Active ALMP
7. Link vLSM receives Status L1 ALMP which triggers an unexpected ALMP error as any ALMP other than Active Status ALMP in response to Active Request ALMP is considered an unexpected ALMP by CXL Specification and must trigger recovery.
8. vLSM requests LTSSM to trigger the recovery.
9. The flow follows by Status Synchronization protocol and Exit from L1 as described in "CXL.cache/CXL.mem L1 Exit" on page 113.

4.3.1.5 CXL.cache/CXL.mem L1 Timeout

Figure 4-9 shows an example of L1 Entry timeout. Downstream Component requests L1 Entry followed by sending Request L1 ALMP to the Upstream Component. CXL Specification requires the Downstream Component ARB/MUX port to wait for at least 1 ms (not including time spent in recovery states) for a response from the Upstream Component. If no response is received from the Upstream Component, then timeout event occurs on the Downstream Component. CXL controller notifies the application about the timeout event and retries L1 Entry by sending Request L1 ALMP. After observing L1 Timeout event, application takes a call and aborts L1 Entry request as shown in Figure 4-7, otherwise ARB/MUX continues L1 Entry attempts every time timeout event is triggered.

Figure 4-9 Downstream Component Request L1 ALMP Timeout Followed by L1 Abort



Downstream Component performs the following steps on L1 Abort request:

1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM waits for CXL controller to complete all outgoing transactions.
4. After all transactions are complete, CXL controller deasserts `cxl_[mem|cache]_slv_ack` initialization acknowledge on the TX channel interfaces, indicating Idle state of the CXL transmit datapath. Link vLSM sends Request L1 ALMP to communicate the state of the link and waits to receive Status L1 ALMP in response.
5. Status L1 ALMP is not received from the Upstream Component for at least 1 ms and timeout event is triggered. vLSM asserts `vlsm_l1_timeout` to notify the Application about the timeout event.

6. The Application observes timeout event, decides to abort L1 Entry request, and asserts `cxl_[mem|cache]_slv_req` on the TX channel interfaces.
7. Link vLSM sends Request Active ALMP and continues with L1 Exit scenario as described in "CXL.cache/CXL.mem L1 Exit" on page 113.

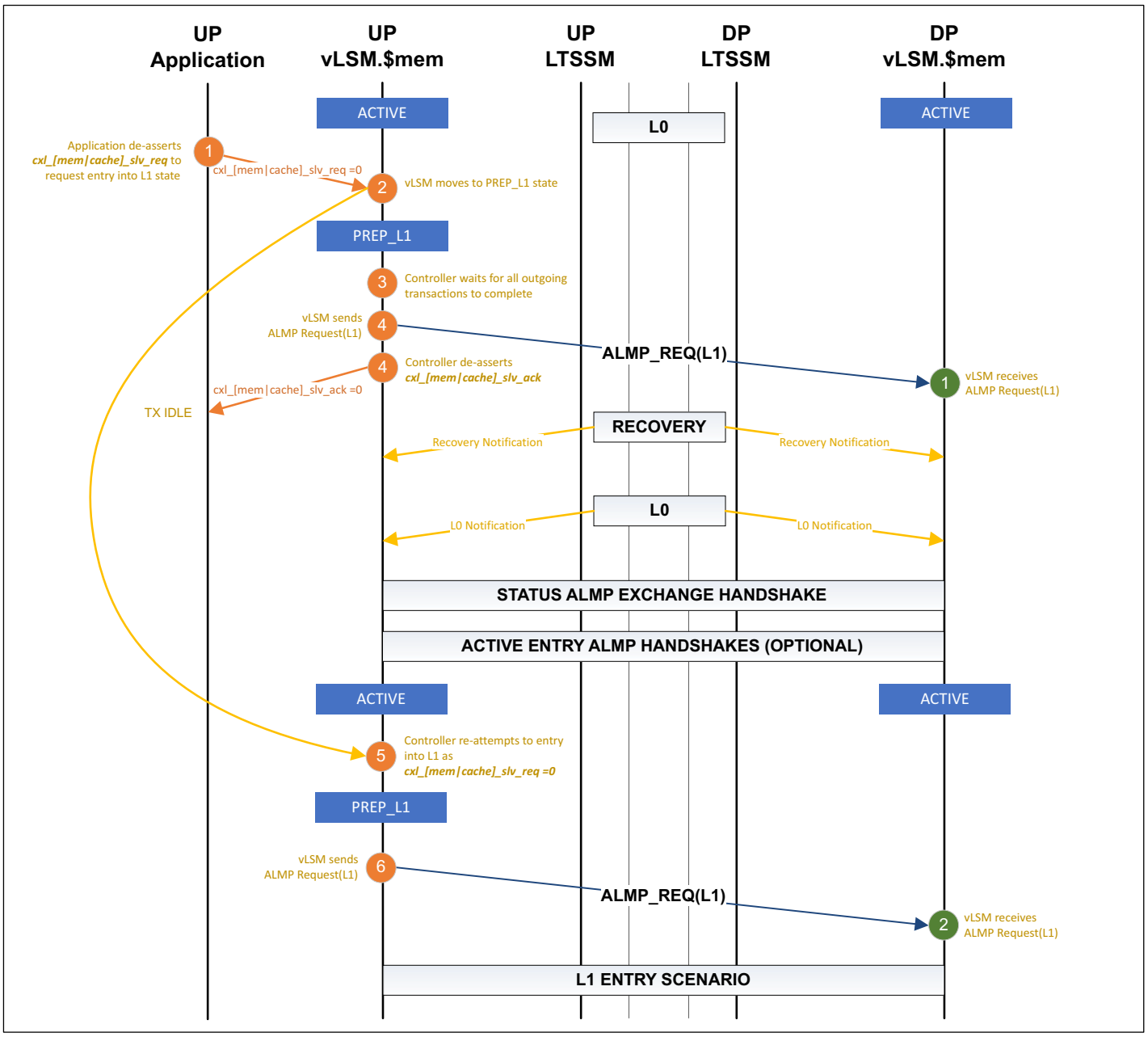
Upstream Component performs the following steps on L1 Abort request:

1. Upstream Component receives Request L1 ALMP and it does not respond with Status L1 ALMP for more than 1ms declining L1 Entry request.
2. If Link vLSM receives Request Active ALMP at any stage before or after Link vLSM transition to L1, it continues with L1 Exit scenario as described in "CXL.cache/CXL.mem L1 Exit" on page 113.

4.3.1.6 CXL.cache/CXL.mem L1 Entry and Link Recovery

Figure 4-10 below illustrates an example of Recovery triggered at any stage during L1 Entry. On exit from the Recovery, if Downstream Component observes that the application still deasserts `cxl_[mem|cache]_slv_req` requesting L1 Entry, it will re-attempt L1 Entry.

Figure 4-10 Recovery Triggered on L1 Entry



Downstream Component performs the following steps on L1 Abort request:

1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. Link vLSM waits for CXL controller to complete all outgoing transactions.
4. After all transactions are complete, CXL controller deasserts `cxl_[mem|cache]_slv_ack` initialization acknowledge on the TX channel interfaces, indicating Idle state of the CXL transmit datapath. Link vLSM sends Request L1 ALMP to communicate the state of the link and waits to receive Status L1 ALMP in response. At this stage link Recovery is triggered.

5. On exit from the Recovery, vLSM observes that the application deasserts `cxl_[mem|cache]_slv_req` requesting L1 Entry.
6. In response Link vLSM moves to the transitional state on Entry to L1 and re-attempts L1 Entry following the steps described in the section “Downstream Component L1 Entry”.

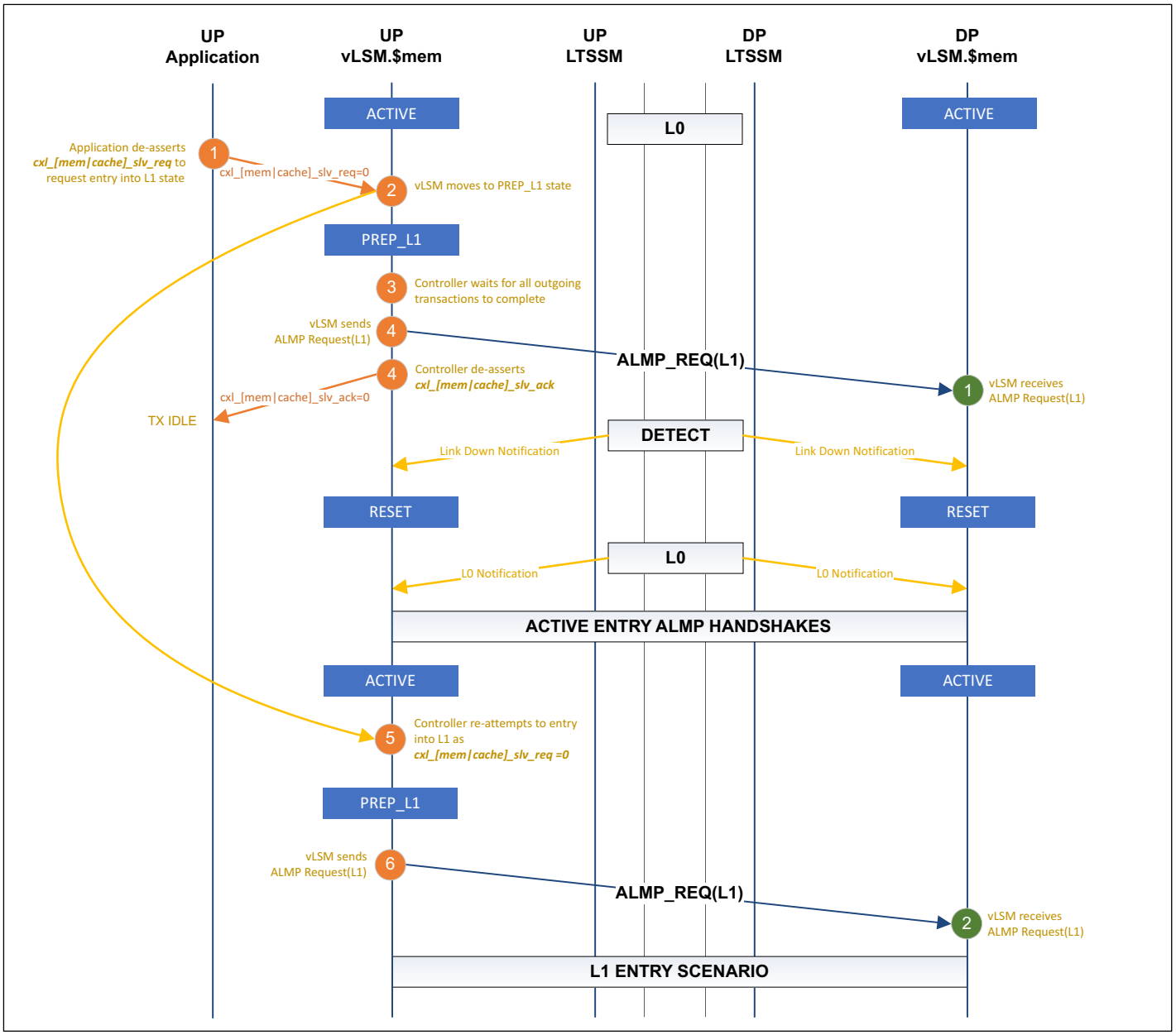
Upstream Component performs the following steps on L1 Abort request:

1. Upstream Component receives Request L1 ALMP when link recovery is triggered.
2. On exit from the Recovery, Upstream Component receives Request L1 ALMP and re-attempts L1 Entry following the steps described in the section “Upstream Component L1 Entry”.

4.3.1.7 CXL.cache/CXL.mem L1 Entry and Link Down/Link Reset

Figure 4-11 below illustrates an example of Link Down/Link Reset triggered at any stage during L1 Entry. Link Down/Link Reset results in the CXL controller reset and CXL link re-initialization. As a result, vLSM transitions to the RESET and both RX/TX channel interfaces are reset. On exit from the RESET, when link trains to the highest supported speed and eventually transits to the Active state, if vLSM observes that the application does not `deassert cxl_[mem|cache]_slv_req`, it attempts L1 Entry. This is a regular behavior of CXL.cache/CXL.mem vLSM on exit from Reset.

Figure 4-11 Link Down/Reset Triggered on L1 Entry



Downstream Component performs the following steps on L1 Abort request:

1. Application deasserts `cxl_[mem|cache]_slv_req` to request de-initialization of the TX channel interfaces and request L1 Entry
2. In response Link vLSM moves to the transitional state on Entry to L1
3. Link vLSM waits for CXL controller to complete all outgoing transactions
4. After all transactions are complete, CXL controller deasserts `cxl_[mem|cache]_slv_ack` initialization acknowledge on the TX channel interfaces, indicating Idle state of the CXL transmit datapath. Link

vLSM sends Request L1 ALMP to communicate the state of the link and waits to receive Status L1 ALMP in response. At this stage Link down/Link reset is requested.

5. On exit from the Reset, vLSM observes that the application deasserts `cxl_[mem|cache]_slv_req` requesting L1 Entry
6. In response Link vLSM moves to the transitional state on Entry to L1 and re-attempts L1 Entry following the steps described in.

Upstream Component performs the following steps on L1 Abort request:

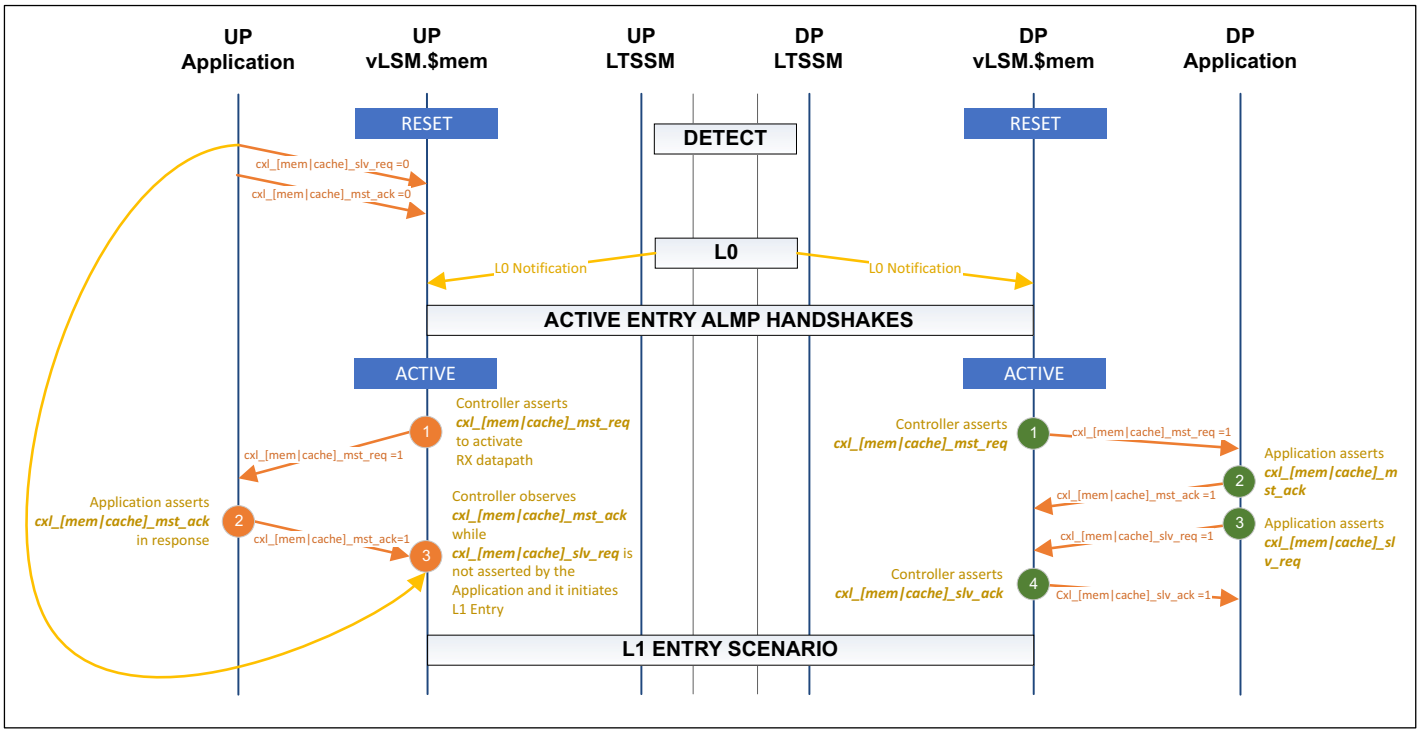
1. Upstream Component receives Request L1 ALMP when link recovery is triggered.
2. On exit from the Reset, Upstream Component receives Request L1 ALMP and re-attempts L1 Entry following the steps described in"CXL.cache/CXL.mem L1 Entry" on page 110.

4.3.1.8 CXL.cache/CXL.mem L1 Auto-Entry

On exit to ACTIVE state from RESET or RETRAIN, for example, after initial link training, CXL controller attempts to initialize channel interfaces to start transmitting and receiving CXL.cache/CXL.mem messages on the interface. After CXL.cache/CXL.mem vLSM transits to the ACTIVE state on completion of Request-Status Active ALMP handshake, CXL controller asserts `cxl_[mem|cache]_mst_req` on the RX channel interfaces to activate RX datapath, at the same time it is expected the Application to assert `cxl_[mem|cache]_slv_req` on the TX channel interfaces to activate TX datapath. When Application on Downstream Component acknowledges RX channel interfaces initialization request by asserting `cxl_[mem|cache]_mst_ack` without first observing TX channel interfaces initialization request from the Application on `cxl_[mem|cache]_slv_req`, CXL controller triggers L1 Entry. This case is similar to the regular L1 Entry scenario with only one notable difference that is TX channel interfaces are already de-initialized.

Figure 4-12 illustrates an example of L1 Entry when Application on Downstream Component does not request TX channel interfaces initialization on transition to ACTIVE state.

Figure 4-12 Downstream Component L1 Auto-Entry on Exit From Reset/Recovery



Downstream Component performs the following steps on entry to L1 state:

1. On transition to Active state from Reset, the controller asserts `cxl_[mem|cache]_mst_req` to request initialization of the RX channel interfaces and activating RX datapath
2. Application asserts `cxl_[mem|cache]_mst_ack` on the RX channel interfaces in response to acknowledge RX datapath activation
3. CXL controller observes as the application asserts `cxl_[mem|cache]_mst_ack` without first observing TX Channel Initialization request from Application on `cxl_[mem|cache]_slv_req`

   CXL controller triggers L1 Entry flow.

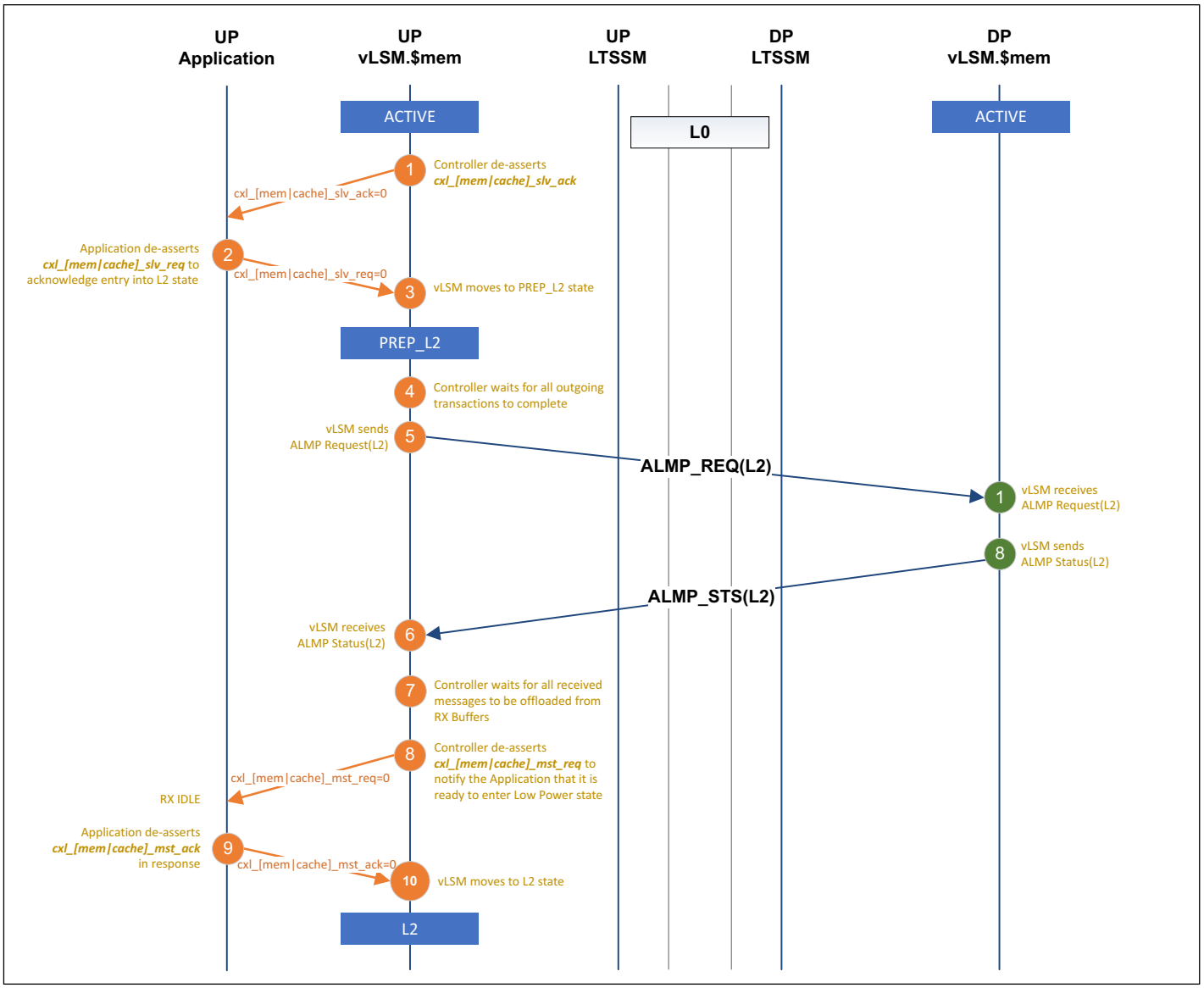
4.3.1.9 CXL.cache/CXL.mem L2 Entry

According to the CXL Specification, the link cannot operate for any other protocols if CXL.io protocol is down. As CXL.io operation is a minimum requirement, CXL controller must ensure a transition of CXL.cache/CXL.mem protocol stack into the corresponding PM state when L2 Entry is requested for CXL.io protocol. In this case, downstream component deasserts `cxl_[mem|cache]_slv_ack` on all TX channel interfaces to notify the application about the intent to go to the low power state and therefore de-initialize the interface. The application must complete all ongoing transactions and deassert `cxl_[mem|cache]_slv_req` in response. CXL controller does not define a maximum period of time for the Application to de-initialize the channel interface, but it is expected that for a given implementation it is deterministic. If the application aborts L2 Entry at some stage before de-assertion of `cxl_[mem|cache]_slv_req`, it still must follow the requirement on CXL.cache/CXL.mem protocol stack and complete TX channel interface de-initialization before CXL link re-activation.

CXL Specification does not define L2 Abort or L2 Retry conditions. Hence, it requires protocol to ensure that Upstream component is directed to enter L2 before setting up the conditions for the Downstream Component to request entry into L2 state.

Figure 4-13 illustrates CXL.cache/CXL.mem L2 Entry data-flow diagram for Downstream Component when PM Entry is requested on CXL.io protocol stack.

Figure 4-13 Downstream Component L2 Entry Data-Flow Diagram



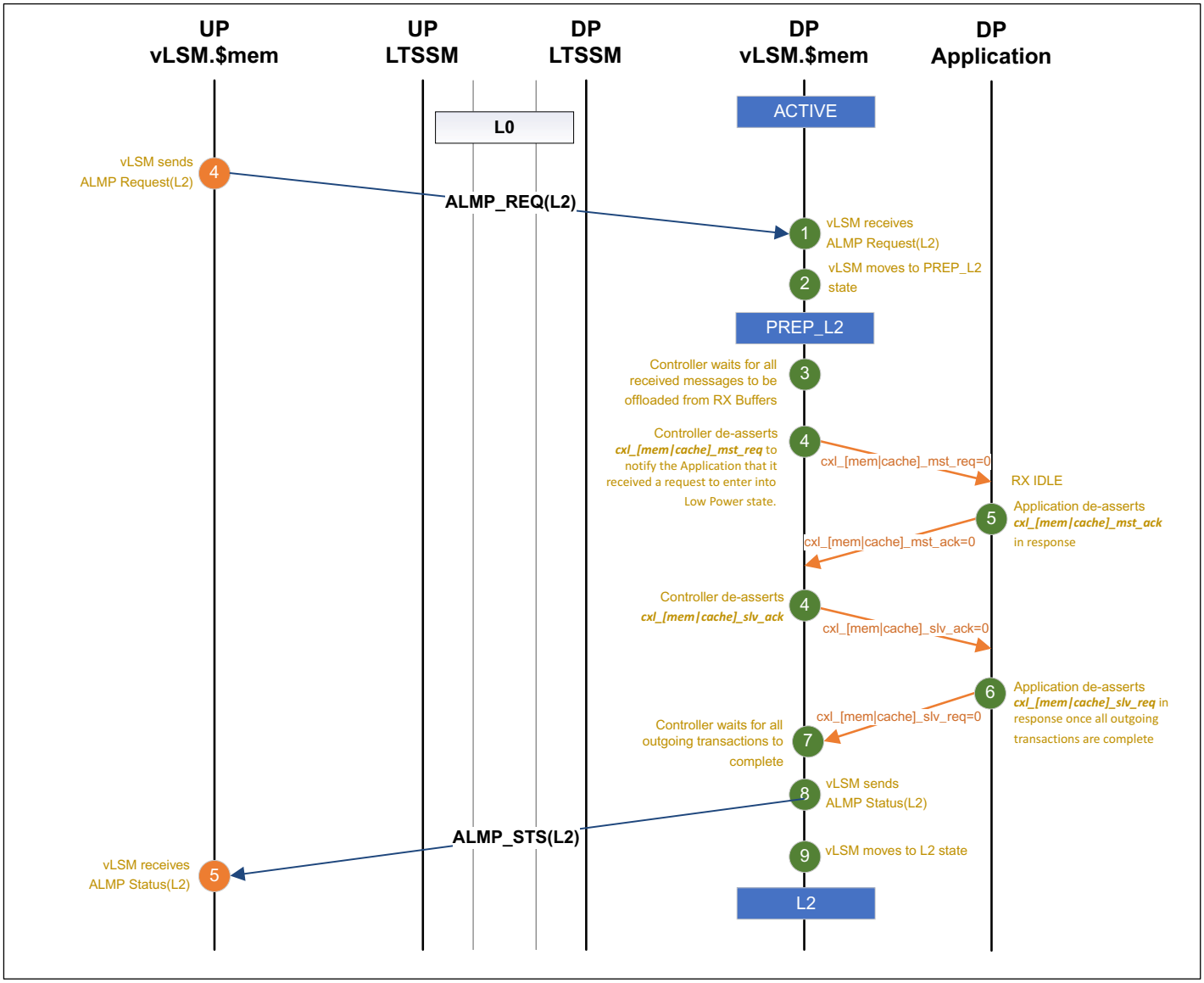
Downstream Component performs the following steps on entry to L2 state:

1. CXL controller deasserts `cxl_[mem|cache]_slv_ack` to initiate L2 Entry
2. Application completes all ongoing transaction and deasserts `cxl_[mem|cache]_slv_req` to acknowledge de-initialization of the TX channel interfaces and L2 Entry request.
3. Link vLSM moves to the transitional state on Entry to L2
4. Link vLSM waits for CXL controller to complete all outgoing transactions

5. After all transactions are complete, Link vLSM sends Request L2 ALMP to communicate the state of the link and waits to receive Status L2 ALMP in response
6. Link vLSM receives Status L2 ALMP
7. CXL controller waits for all received messages to be offloaded from RX buffers and CXL receive data-path to indicate Idle state
8. The controller deasserts `cxl_[mem|cache]_mst_req` to request de-initialization of the RX channel interfaces and notify the application that it is ready for L2 Entry
9. Application must deassert `cxl_[mem|cache]_mst_ack` on the RX channel interfaces in response
10. Link vLSM transits to L2 state

Figure 4-14 illustrates L2 Entry data-flow diagram for Upstream Component

Figure 4-14 Upstream Component L2 Entry Data-Flow Diagram



Upstream Component performs the following steps on entry to L2 state:

1. L2 Entry starts with Link vLSM observing Request L2 ALMP
2. In response Link vLSM moves to the transitional state on Entry to L2
3. After Request L2 ALMP is received, the controller waits for all received messages to be offloaded from RX buffers and CXL receive datapath to indicate Idle state
4. The controller deasserts `cxl_[mem|cache]_mst_req` to request de-initialization of the RX channel interfaces and `cxl_[mem|cache]_slv_ack` to request de-initialization of the TX channel interfaces, and notify the Application that L2 Entry is requested
5. Application must deassert `cxl_[mem|cache]_mst_ack` on the RX channel interfaces in response
6. Application must deassert `cxl_[mem|cache]_slv_req` on the TX channel interfaces once all outgoing transactions are complete to acknowledge L2 Entry request[1]
7. CXL controller waits for all outgoing to complete to indicate CXL TX datapath is in Idle state
8. Link vLSM sends Status L2 ALMP
9. Link vLSM transits to L2 state

4.3.2 CXL.io Link Power Management

CXL.io Link Power Management follows PCI Express Base Specification with the following notable differences:

- CXL 1.1 device supports ASPM L1 only
- PCI-PM is not supported when connected to CXL 1.1 device
- CXL 2.0 device supports ASPM L1 as well as PCI-PM
- L0s state is not supported

All CXL functions implement PCI Power Management Capability Structure as defined in PCI Express Base Specification and support D0 and D3 device states.

4.3.2.1 CXL.io PM Entry and Exit Conditions

CXL.io PM Entry is requested if one of the following events is triggered:

- Application requests L1 Entry by asserting `app_entr_l1`
- Application requests L1 Entry by de-asserting `app_xfer_pending` when ASPM L1 Timer is enabled
- Software programs D-state to request PM Entry

All the illustrations in the document are provided on the example of `app_entr_l1` triggering CXL.io L1 Entry.

CXL.io PM Exit is requested if one of the following events is triggered:

- Application requests L1 Exit by asserting `app_req_exit_l1`
- Application request L1 Exit by asserting `app_xfer_pending`
- There are pending transactions on any client interface

1. Unlike L1 Entry request, Upstream Component might not reject L2 Entry request once it observes both cxl_[mem|cache]_mst_req and cxl_[mem|cache]_slv_ack de-asserted on the channel interface.

- Wakeup request to exit L1 when Root-complex device application wants to send PME message and asserts `apps_pm_xmt_pme`
- Request from Endpoint device application to generate a PM Turn Off message, `apps_pm_xmt_turnoff` is asserted
- L2 Entry is requested when in L1 state, `app_ready_entr_l23` is asserted
- When the Retry buffer is not empty after L1 negotiation is complete

All the illustrations in the document are provided on the example of `app_xfer_pending` triggering CXL.io L1 Exit.

CXL.io PM Exit conditions are extended with the following internal message requests:

- Application requests LTR message by asserting `app_ltr_msg_req`
- Application requests Unlock message asserting `app_unlock_msg`
- Vendor Message or Vendor MSI request
- Message Gen or LBC Completion
- MSIX table generates interrupt

Note All the illustrations are example of app_xfer_pending triggering CXL.io L1 Exit.

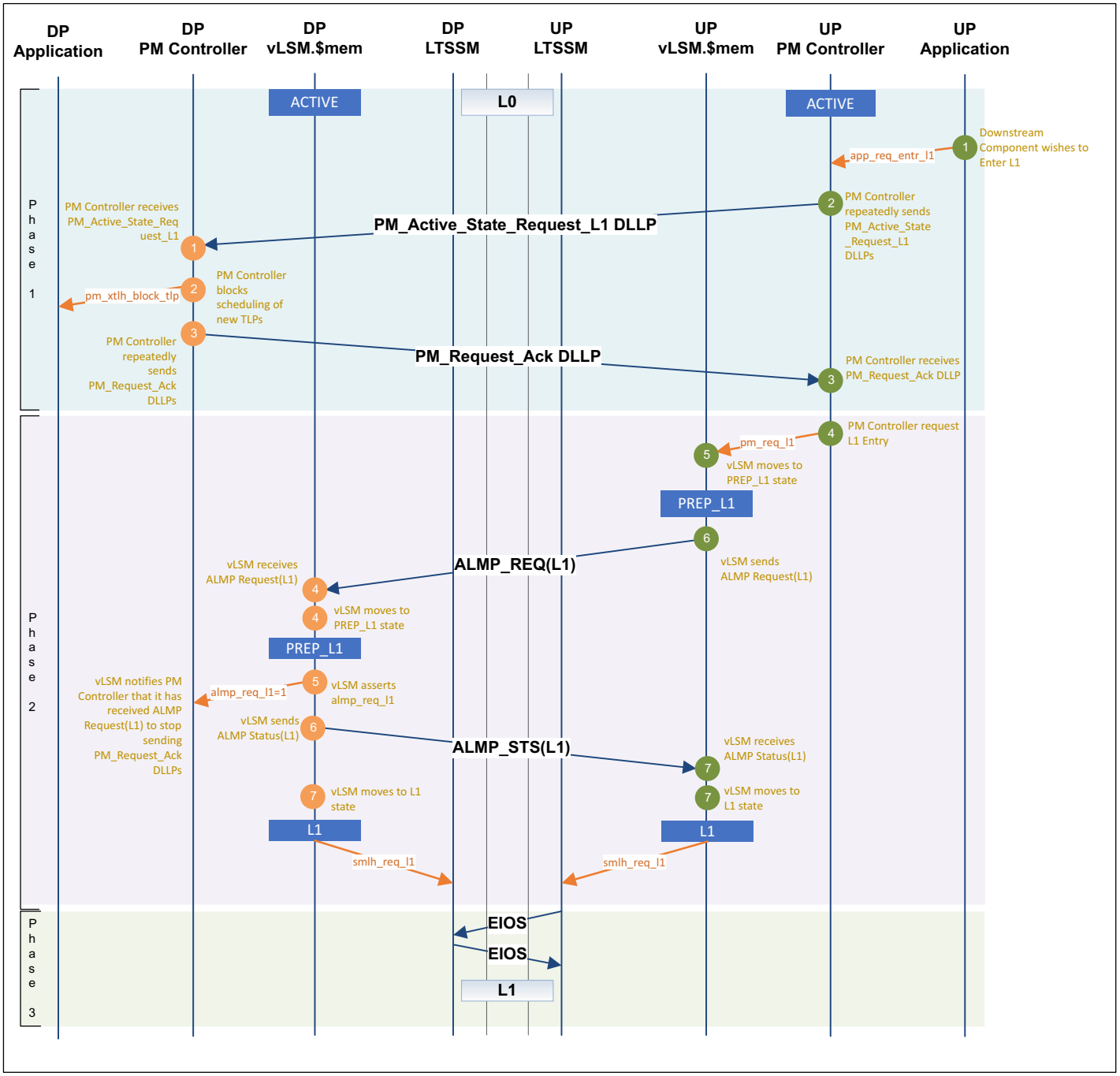
4.3.2.2 CXL.io ASPM L1 Entry

CXL.io PM Entry/Exit process is further divided into the following 3 phases as defined in CXL Specification:

- The first phase consists of completing the ASPM L1 negotiation rules as defined in the PCI Express Base Specification. All rules up to the completion of the ASPM L1 handshake are maintained, however the process of bringing the Transmit Lanes into Electrical Idle state is divided into the following two additional phases.
- The Phase 2 of L1 entry consists of bringing the CXL.io ARB/MUX interface of both sides of the Link into L1 state. Request/Status L1 ALMP handshake coordinates this entry into L1 state.
- ARB/MUX manages the Phase 3 of L1 entry that depends on the CXL.io and CXL.cache/CXL.mem Link vLSM states. If all protocol vLSMs are in L1 state, the link is directed to electrical idle to complete Phase 3.

Figure 4-15 shows an example of ASPM L1 Entry flow which implements 3-phase PM Entry/Exit process defined in the CXL Specification:

Figure 4-15 CXL.io L1 Entry Data-Flow Diagram



Downstream Component performs the following steps on entry to L1 state:

1. Downstream Component wishes to enter L1, it accumulates minimum credits, blocks all new TLPs, and requests L1 Entry by asserting `app_req_entr_l1`
2. PM controller starts repeatedly sending PM_Active_State_Request_L1 DLLPs and waits for the response from the Upstream Component
3. PM controller receives PM_Request_Ack DLLP, it stops sending PM_Active_State_Request_L1 DLLPs and completes Phase 1 Entry process

4. PM controller requests vLSM to start Phase 2 of L1 Entry by asserting `pm_req_l1`.
5. In response Link vLSM moves to the transitional state on Entry to L1
6. Link vLSM sends Request L1 ALMP and waits for acknowledgment
7. Link vLSM receives Status L1 ALMP, moves to L1 state and completes Phase 2
8. Downstream Component transitions upstream direction to electrical idle

Upstream Component performs the following steps on entry to L1 state:

1. Upstream Component receives PM_Active_State_Request_L1 DLLP
2. PM controller blocks scheduling of new TLPs and receives acknowledgment for the last TLP
3. PM controller starts repeatedly sending PM_Request_Ack DLLPs
4. Link vLSM receives Request L1 ALMP, it stops sending PM_Request_Ack DLLPs and completes Phase 1 Entry
5. Link vLSM moves to the transitional state on Entry to L1 and starts Phase 2 of L1 Entry.
6. Link vLSM sends Status L1 ALMP
7. Link VLSM moves to L1 state and completes Phase 2
8. Upstream Component transitions downstream direction to electrical idle

4.3.2.3 CXL.io ASPM L1 Exit

Components on either end of the Link may initiate exit from the L1 Link State. Like L1 Entry, the exit is hierarchical, and Phase 3 must exit before Phase 2. PM controller directs the initiation of phase 3 exit from either end of the link. The PM controller initiates exit from Phase 3 when there is an exit requested on CXL.io interface by asserting `app_xfer_pending`. The Phase 3 ASPM L1 exit is the same as exit from L1 state as defined in PCI Express Base Specification. The steps are followed until the LTSSM reaches L0 state. When Link is already in L0 state when CXL.io L1 Exit is requested, Phase 3 exit is skipped.

Phase 2 exit is initiated on L0 notification from LTSSM and involves bringing the protocol interface at the ARB/MUX out of L1 state independently, following Status Exchange and Active Entry ALMP handshake.

On completion of Phase 2, once vLSM is in Active state, Phase 1 exit completes by activating corresponding transaction layer interface.

As mentioned above, there are two possible scenarios when the application requests L1 Exit, depending on the state of the link:

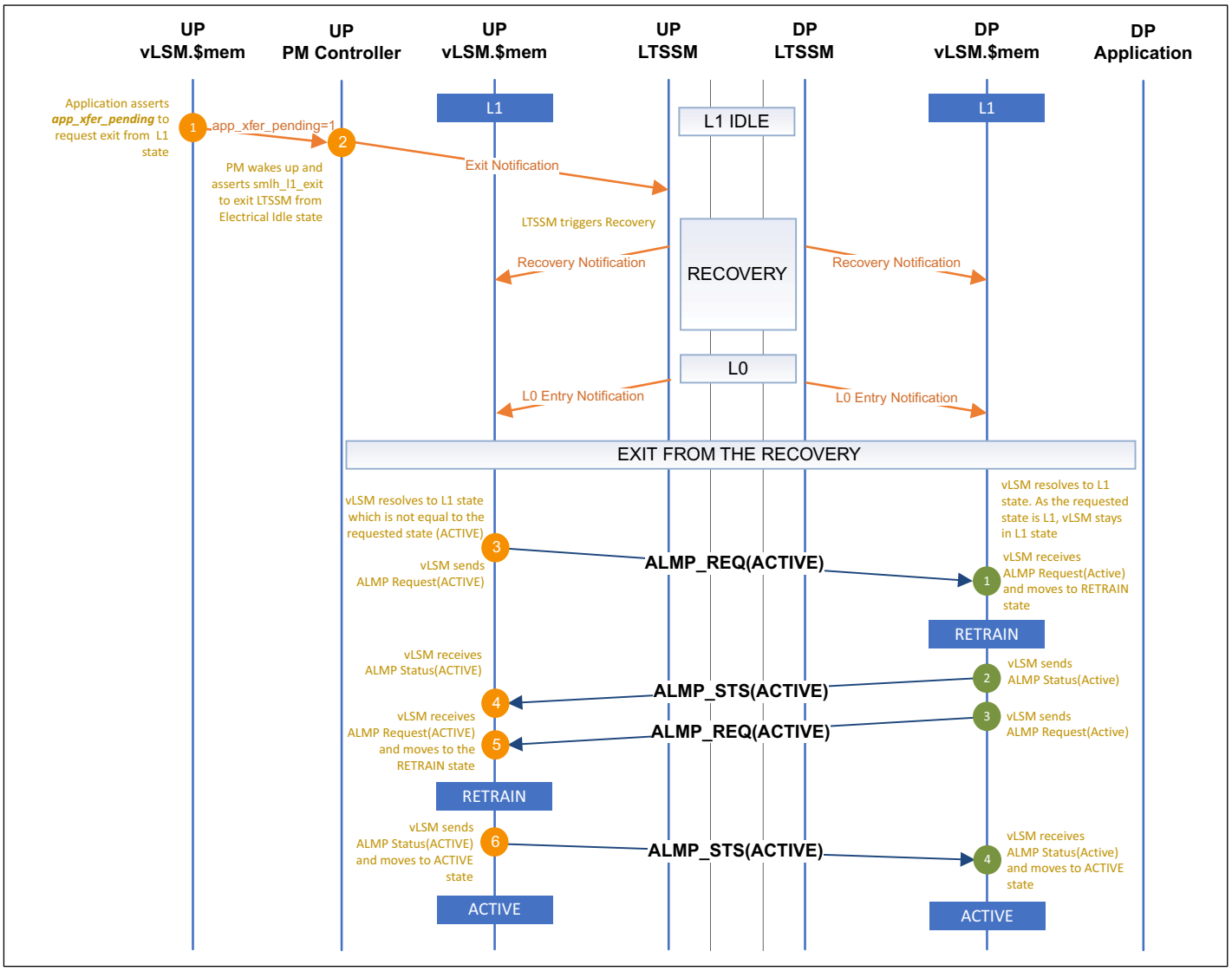
- Link is in L0 state
- Link is in L1 IDLE state

The sub-sections below discuss these two L1 Exit scenarios in detail.

CXL.io L1 Exit When Link is in L1 IDLE state

Figure 4-16 shows an example of L1 Exit scenario when the link is in L1 IDLE state. vLSM receives L1 Exit (Active Request) notification from the PM controller when the application asserts app_xfer_pending in L1 state, and the link enters to the Recovery. After an exit from the Recovery, each vLSM sends Status L1 ALMP to synchronize the state of the link. The state synchronization handshake resolves in L1 state which does not match the Link Layer requested state – Active. New Active Request and Status ALMP exchange handshake is triggered to transit Link vLSM to the Active state.

Figure 4-16 CXL.io L1 Exit Data-Flow Diagram When Link is in L1 State



L1 Exit requester performs the following steps on exit from L1 state:

1. Application wishes to exit L1 and asserts `app_xfer_pending`
2. After `app_xfer_pending` is observed, PM controller instructs LTSSM to exit from L1 IDLE state, which triggers an entry to the Recovery
3. After State Synchronization on exit from the Recovery, vLSM resolves to L1 state after sending Status L1 ALMP and receiving Status L1 ALMP. As the resolved state (L1) does not match the requested state (ACTIVE), Link vLSM triggers Active State Request/Status handshake protocol and sends Request Active ALMP.
4. Link vLSM receives Status Active ALMP
5. Link vLSM receives Request Active ALMP and transitions to RETRAIN state to exit from L1
6. Link vLSM transmits Status Active ALMP and transitions to ACTIVE state

L1 Exit sub-ordinate component performs the following steps on exit from L1:

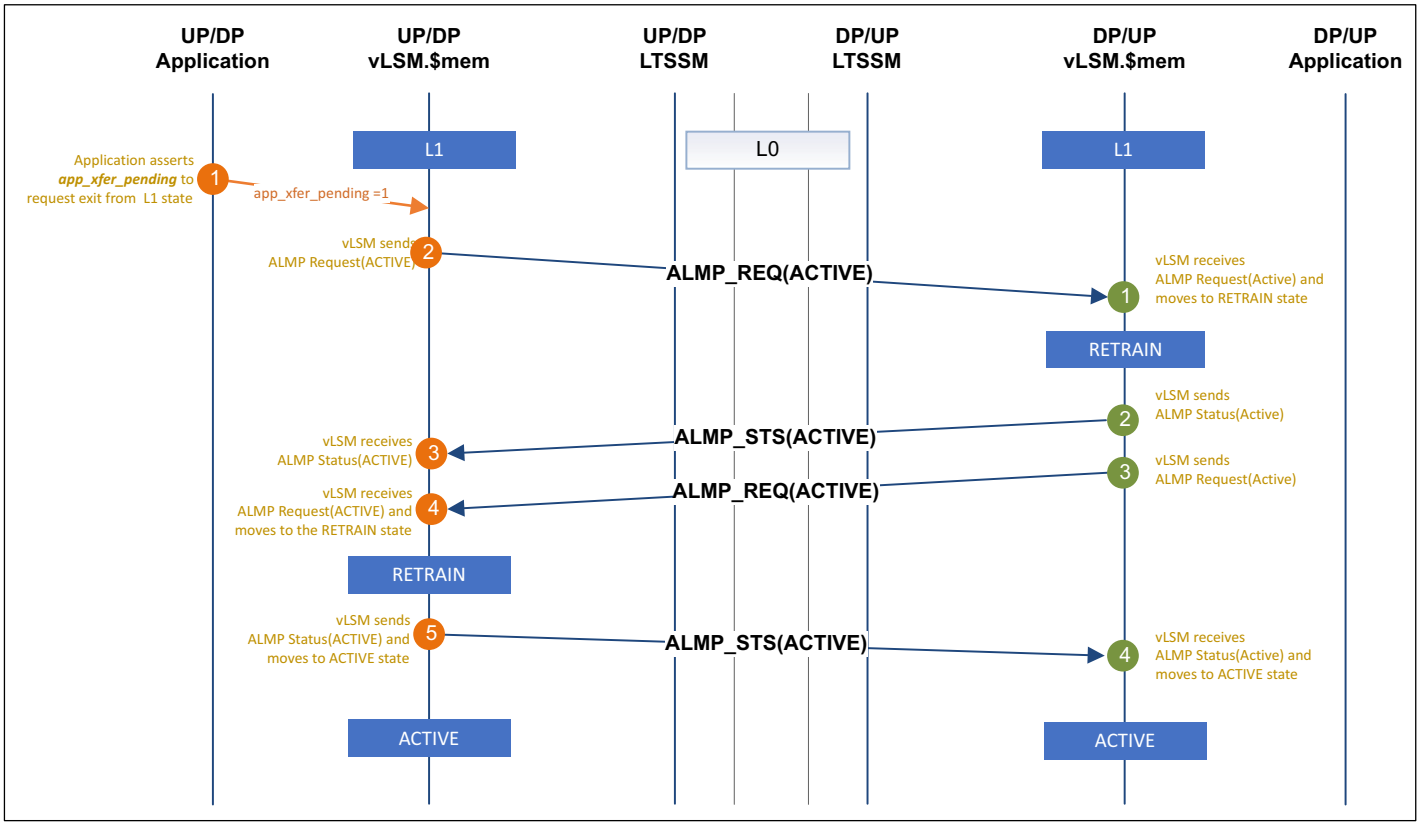
1. During State Synchronization on exit from the Recovery, vLSM resolves to L1 state after sending Status L1 ALMP and receiving Status L1 ALMP. After vLSM receives Request L1 ALMP after the Recovery, it transits to the RETRAIN state
2. Link vLSM responds with Status Active ALMP
3. Link vLSM sends Request Active ALMP
4. Link vLSM receives Status Active ALMP in response and transitions to ACTIVE state

CXL.io ASPM L1 Exit When Link is in L0 State

Figure 4-17 shows an example of L1 Exit scenario when the link is in L0 state while CXL.io is in L1 state. The main difference with a previous example is that the link is not in the electrical idle state hence the physical link recovery is not required on exit from L1 and only vLSMs must go through the RETRAIN to exit from L1 to ACTIVE state.

In this scenario, the requester vLSM transmits Request Active ALMP when the application asserts `app_xfer_pending` in L1 state. In response, the sub-ordinate vLSM transits CXL.io Link vLSM to RETRAIN state to exit from L1 state and responds with Status Active ALMP followed by Request Active ALMP. The requester receives Request Active ALMP and exits to the RETRAIN state and, after transmitting Status Active ALMP in response, it completes an entry to the ACTIVE state. In its turn, the sub-ordinate vLSM completes an entry to the ACTIVE state once it receives Status Active ALMP in response to the Request Active ALMP.

Figure 4-17 CXL.cache/CXL.mem L1 Exit Data-Flow Diagram When Link is in L0 State



L1 Exit requester component performs the following steps to exit from L1:

1. The Application asserts `app_xfer_pending` to request L1 Exit
2. After `app_xfer_pending` is observed when link is in L0 state, Link vLSM sends Request Active ALMP
3. Link vLSM receives Status Active ALMP
4. Link vLSM receives Request Active ALMP and transitions to RETRAIN state to exit from L1
5. Link vLSM transmits Status Active ALMP and transitions to ACTIVE state

L1 Exit sub-ordinate component performs the following steps on exit from L1:

1. Link vLSM receives Request Active ALMP when Link vLSM is in L1 state and transitions to RETRAIN state
2. If link is in L0 state, Link vLSM responds with Status Active ALMP
3. Link vLSM sends Request Active ALMP
4. Link vLSM receives Status Active ALMP in response and transitions to ACTIVE state

4.3.3 PM Entry Phase 3

Phase 3 is a conditional phase of PM Entry and is executed only when both CXL.cache/CXL.mem and CXL.io protocol interfaces of ARB/MUX have entered the same virtual PM state. Downstream Component initiates this phase which consists of bringing the Tx lanes to electrical Idle.

ARB/MUX directs the initiation of Phase 3 exit from either end of the link. The ARB/MUX layer initiates exit from Phase 3 when there is an exit requested on either CXL.cache/CXL.mem or CXL.io protocol interfaces.

Since the CXL ARB/MUX virtualizes the link state, there may be a race condition that results in an L1 abort scenario. In this case, the physical layer may receive an EIOS or detect Electrical Idle when the ARB/MUX is no longer requesting entry to L1. In this scenario, LTSSM initiates recovery on the link to bring it back to L0.

When the link transitions to recovery during or after entry into electrical idle, the Downstream Component triggers 1us time after reaching L0 before re-initiating entry into electrical idle as it is required by Errata F22. This is to allow enough time for an Active State Request ALMP transfer to occur in case either side wants to initiate a PM exit (and to give time for the remote ARB/MUX to stop requesting PM entry to physical layer).

Implementation

CXL controller clock must stop in Phase 3 due to clock gating. `cxl_mem_slv_req` and/or `cxl_cache_slv_req` assertion triggers a wake up in the PM controller and restarts the clock. It is possible to implement the interface pipeline registers the CXL clock domain and clock gate them during L1 state. As a result, `cxl_mem_slv_req` and `cxl_cache_slv_req` require a complete combinational path between the Application and CXL controller in L1 state. The PM controller considers `cxl_mem_slv_req` and `cxl_cache_slv_req` as asynchronous signals. They run through appropriate synchronization logic to avoid metastability before use. On the other hand, CXL controller considers `cxl_mem_slv_req` and `cxl_cache_slv_req` as synchronous signals in the presence of controller clock.

4.3.4 CXS.B CXL Controller Link Power Management

CXS.B CXL controller PM follows the same rules and handshake as it is defined for the CXL.cache/CXL.mem protocol stack with a native Channel Message Interface. The only notable difference is that CXS.B CXL controller uses CXS Activation Interface instead of CXL Channel Initialization Interface to

control the state of the interface. Use the mapping shown in Table 4-1 to refer the PM flow between CXS Activation Interface and CXL Channel Initialization Interface in the databook

Table 4-1 CXS CXL Interface Mapping

Native Interface	CXS Interface	Direction	Description
cxl_[mem|cache]_slv_req	cxsrxactivereq	Input	CXS RX Activation Request. Associated with an assertion of cxl_cache_slv_req or cxl_mem_slv_req by the Application requesting the interface initialization.
cxl_[mem|cache]_slv_ack	cxsrxactiveack	Output	CXS RX Activation Acknowledge. Associated with an assertion of cxl_cache_slv_ack or cxl_mem_slv_ack by the controller in response to cxl_cache_slv_req or cxl_mem_slv_req.
NA	cxsrxdeacthint	Output	CXS RX Deactivation Hint. Associated with de-assertion of cxl_cache_slv_ack or cxl_mem_slv_ack by the controller requesting the interface de-initialization.
cxl_[mem|cache]_mst_req	cxstxactivereq	Output	CXS TX Activation Request. Associated with an assertion of cxl_cache_mst_req or cxl_mem_mst_req by the controller requesting the interface initialization.
cxl_[mem|cache]_mst_ack	cxstxactiveack	Input	CXS TX Activation Acknowledge. Associated with an assertion of cxl_cache_mst_ack or cxl_mem_mst_ack by the Application in response to cxl_cache_mst_req or cxl_mem_mst_req.
NA	cxstxdeacthint	Input	CXS TX Deactivation Hint. Associated with de-assertion of cxl_cache_mst_ack or cxl_mem_mst_ack by the Application requesting the interface de-initialization.

4.3.4.1 CXS.B CXL Controller PM Entry and Exit Conditions

CXS.B CXL controller PM Entry is requested if one of the following events is triggered:

- Application requests L1 Entry by de-asserting `cxsrxactivereq` in ACTIVE state after CXS TX Activation Request was acknowledge by CXL controller (L1 Entry).
- Application is not asserting `cxsractivereq` on transition to ACTIVE state while acknowledging CXS TX Activation Request from CXL controller (L1 Auto-Entry).
- PM controller requests L2 Entry.

4.3.4.2 CXS.B CXL Controller L1 Entry

CXL controller requires CXS.B RX/TX interfaces to be de-activated before entry into L1 state. The Application always initiates CXS.B CXL controller L1 Entry on the Downstream Component de-asserting `cxsrxactivereq` CXS.B RX Activation Request.

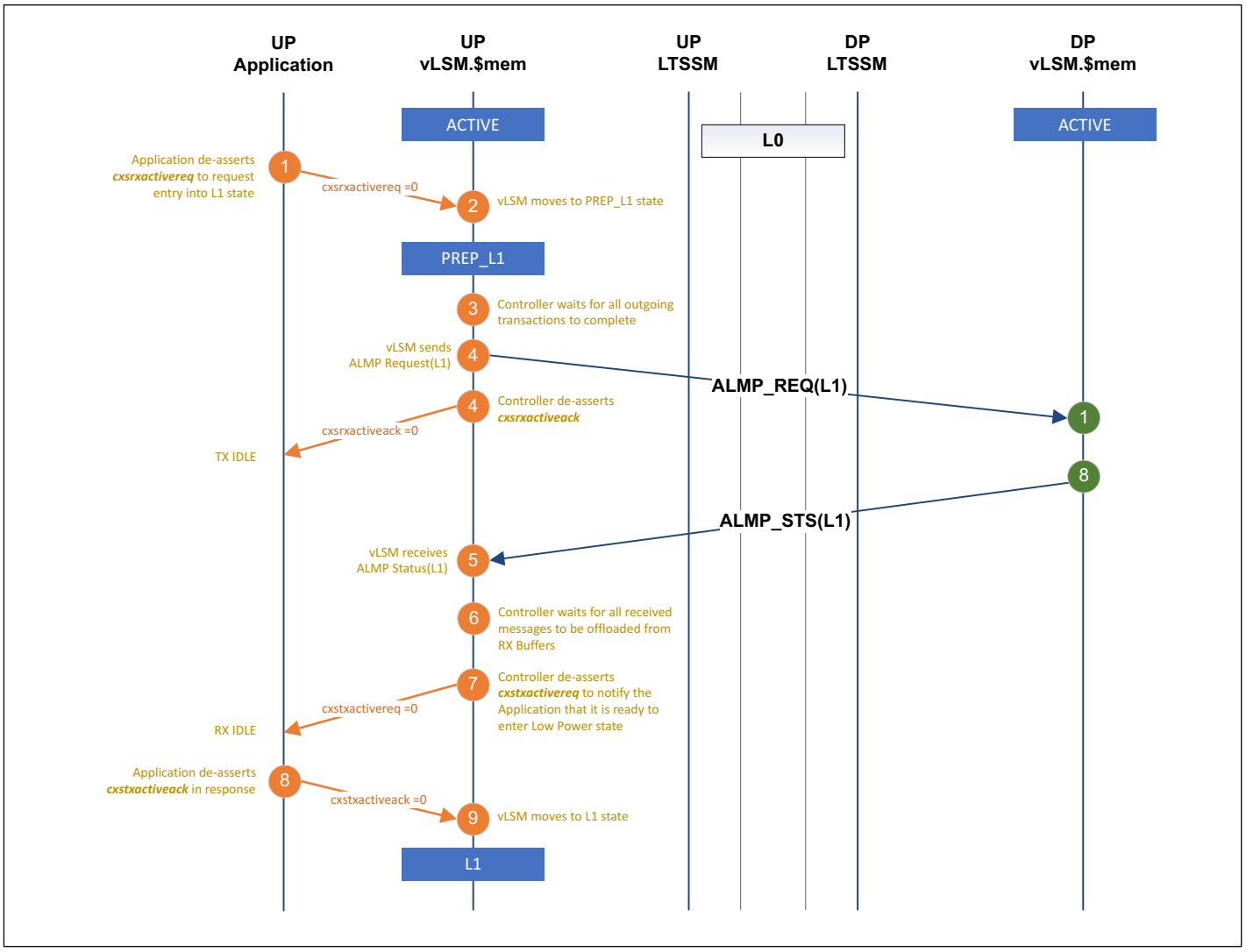
On L1 Entry, Upstream and Downstream Components deactivate the CXS.B Interface in TX-RX pairs, i.e. Downstream component de-activates TX datapath (CXS.B RX Interface) before transmitting Request L1 ALMP and RX datapath (CXS.B TX Interface) after receiving Status L1 ALMP. In its turn, Upstream

component de-activates RX datapath (CXS.B TX Interface) when Request L1 ALMP is received and TX datapath (CXS.B RX Interface) before transmitting Status L1 ALMP.

When CXL.cache/CXL.mem protocols are enabled, L1 Entry must be triggered and completed on both interfaces before transition to L1 state.

Downstream Component L1 Entry

Figure 4-18 Downstream Component L1 Entry Data-Flow Diagram



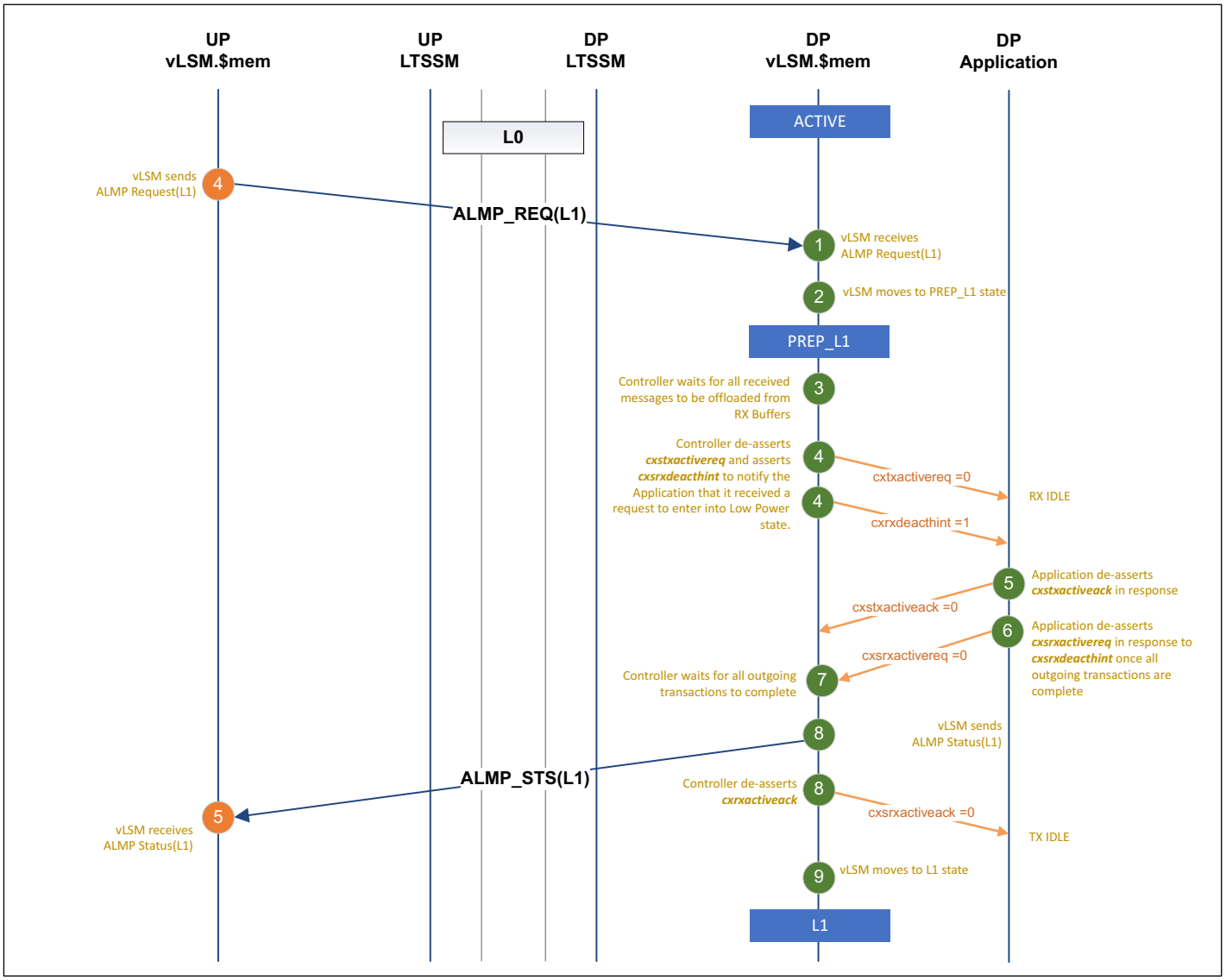
Downstream Component performs the following steps on entry to L1 state:

1. Application deasserts `cxsrxactivereq` to de-activate CXS.B RX Interface and request L1 Entry.
2. In response Link vLSM moves to the transitional state on Entry to L1.
3. In response Link vLSM moves to the transitional state on Entry to L1.
4. After all transactions are complete, CXL controller deasserts `cxsrxactiveack` activation acknowledge on the CXS.B RX Interface, indicating Idle state of the CXL transmit datapath. Link vLSM sends Request L1 ALMP to communicate the state of the link and waits to receive Status L1 ALMP in response.

5. After Status L1 ALMP is received, the controller waits for all received messages to be offloaded from RX buffers and CXL receive datapath to indicate Idle state.
6. The controller deasserts `cxstxactivereq` to request de-initialization of the CXS.B TX Interface and notify the application that it is ready for L1 Entry.
7. Application must deassert `cxstxactiveack` on the CXS.B TX Interface in response
8. Link vLSM transits to L1 state

Upstream Component L1 Entry

Figure 4-19 Upstream Component L1 Entry Data-Flow Diagram



Upstream Component performs the following steps on entry to L1 state:

1. L1 Entry starts with Link vLSM observing Request L1 ALMP
2. In response Link vLSM moves to the transitional state on Entry to L1

3. After Request L1 ALMP is received, the controller waits for all received messages to be offloaded from RX buffers and CXL receive datapath to indicate Idle state
4. The controller deasserts `cxstxactivereq` to request de-activation of the CXS.B TX Interface as well as asserts `cxsrxdeacthint` on CXS.B RX Interface to notify the Application that L1 Entry is requested
5. Application must deassert `cxstxactiveack` on the CXS.B TX Interface in response to de-assertion of `cxstxactivereq`
6. Application deasserts `cxsrxactivereq` on the CXS.B RX Interface once all outgoing transactions are complete to acknowledge L1 Entry request[1]
7. CXL controller waits for all outgoing to complete to indicate CXL TX datapath is in Idle state
8. Link vLSM sends Status L1 ALMP. CXL controller deasserts `cxsrxactiveack` to notify the Application that L1 Entry completed
9. Link vLSM transits to L1 state

1. Upstream Component may reject L1 Entry. In this case, the Application does not deassert cxsrxactivereq in response to the CXS RX Deactivation Hint. The Downstream Component may either retry L1 Entry request or abort L1 Entry.

5

CXL Switch

The CXL controller supports CXL 2.0 Switch. The following topics are discussed:

Note An MLD Downstream Switch Port supports Fabric Manager (FM) owned PPB only. It does not support a vPPB.

5.1 Alternate Memory and Bus Ranges

The CXL Upstream Switch Port supports an Alternate Memory Range and an Alternate Prefetchable Memory Range. When the Alt Memory and ID Space Enable bit in CXL 2.0 Extensions DVSEC is set, that is `ALT_MEM_ID_SPACE_EN` field in the `CXL_2_0_CTRL_ALT_BUS_BASE_LIMIT_OFF` register is '1' (irrespective of the value of the MSE bit in the PCI Command Register, that is MSE field in the `TYPE1_STATUS_COMMAND_REG` register), an Upstream Switch Port accepts memory accesses (traveling Downstream) to these ranges as a valid request and forwards them downstream. If Alt Memory and ID Space Enable bit is 0, the Upstream Switch Port treats such accesses as Unsupported Requests.

The CXL Downstream Switch Port and CXL Downstream Root Port support an Alternate Bus Range. If the Alt Bus Master Enable bit in the CXL 2.0 Extensions DVSEC is set, that is `ALT_BME` field in the `CXL_2_0_CTRL_ALT_BUS_BASE_LIMIT_OFF` register is '1'(irrespective of the value of the Bus Master Enable bit in the PCI Command Register, that is the BME field of the `TYPE1_STATUS_COMMAND_REG` register), the Downstream Switch Port accepts the Memory and I/O accesses (traveling Upstream) from a requester with bus number lying in this Alternate Bus Range and forwards them upstream. If the Alternate Bus Master Enable bit is 0, the Downstream Switch Port treats such accesses as Unsupported Requests.

Note

- The ALT_BME field only controls the forwarding of Upstream flowing memory accesses. This include transactions targeted at addresses beyond the memory, prefetchable memory, alternate memory, and alternate prefetchable memory ranges of the CXL Downstream Switch Port or CXL Root Port.
- The ALT_BME field has no bearing on the memory accesses targeted at the BARs of the CXL Downstream Switch Port or CXL Root Port. Instead, the corresponding port consume such accesses as they are not meant to be flowing Upstream.
- When ACS Upstream forwarding is enabled, the Downstream Switch Port forwards all requests Upstream irrespective of the value of the ALT_BME field of the ALT_BME register.

5.2 DPC and Virtual DPC for Downstream MLD Switch Port

For an MLD negotiated Switch Downstream Port controller, your CXL Switch application must decode the received Message TLP. If your Switch application detects any ERR_FATAL or ERR_NONFATAL message in the FMLD owned message TLP, it must assert the DPC input signals `cxl_dpc_fmld_triggered`, `cxl_dpc_fmld_trig_reason`, and `cxl_dpc_fmld_trig_err_src_id`. The controller registers the value of these signals in the `DPC_DPC_TRIG_STATUS`, `DPC_DPC_TRIG_REASON`, and `DPC_DPC_ERROR_SOURCE_ID` fields of the `DPC_STATUS_ERR_SOURCE_ID_OFF` register, respectively.

ERR_FATAL and ERR_NONFATAL messages from a Logical Device can trigger a virtual DPC. Your CXL Switch application must implement the logic to handle virtual DPC. Your CXL Switch application must drop the transmit and receive TLPs of the Logical Device that triggered virtual DPC.

Note There can be a latency of few cycles between FMLD virtual DPC or DPC trigger detection by the Switch application and trigger notification to the CXL controller. The CXL controller cannot drop the TLPs scheduled during this period.

5.3 MLD Switch Application Considerations

Your MLD Switch Application must perform the following tasks:

- Bind the Host to LD-ID.
- Maintain a mapping of LD-ID and physical function. This mapping must be same across the Switch application and Device application. Your Switch application must assign Function 0 to FMLD (LD ID 0xFFFF).
- Assign exclusive physical functions (except Function 0 because it is FM owned) to each Host.
- CXL.IO TLP Vendor local prefixing of LD-ID.
- The Switch Application must assign the same number and Device number to MLD Devices in each virtual hierarchy in the Switch.

Note

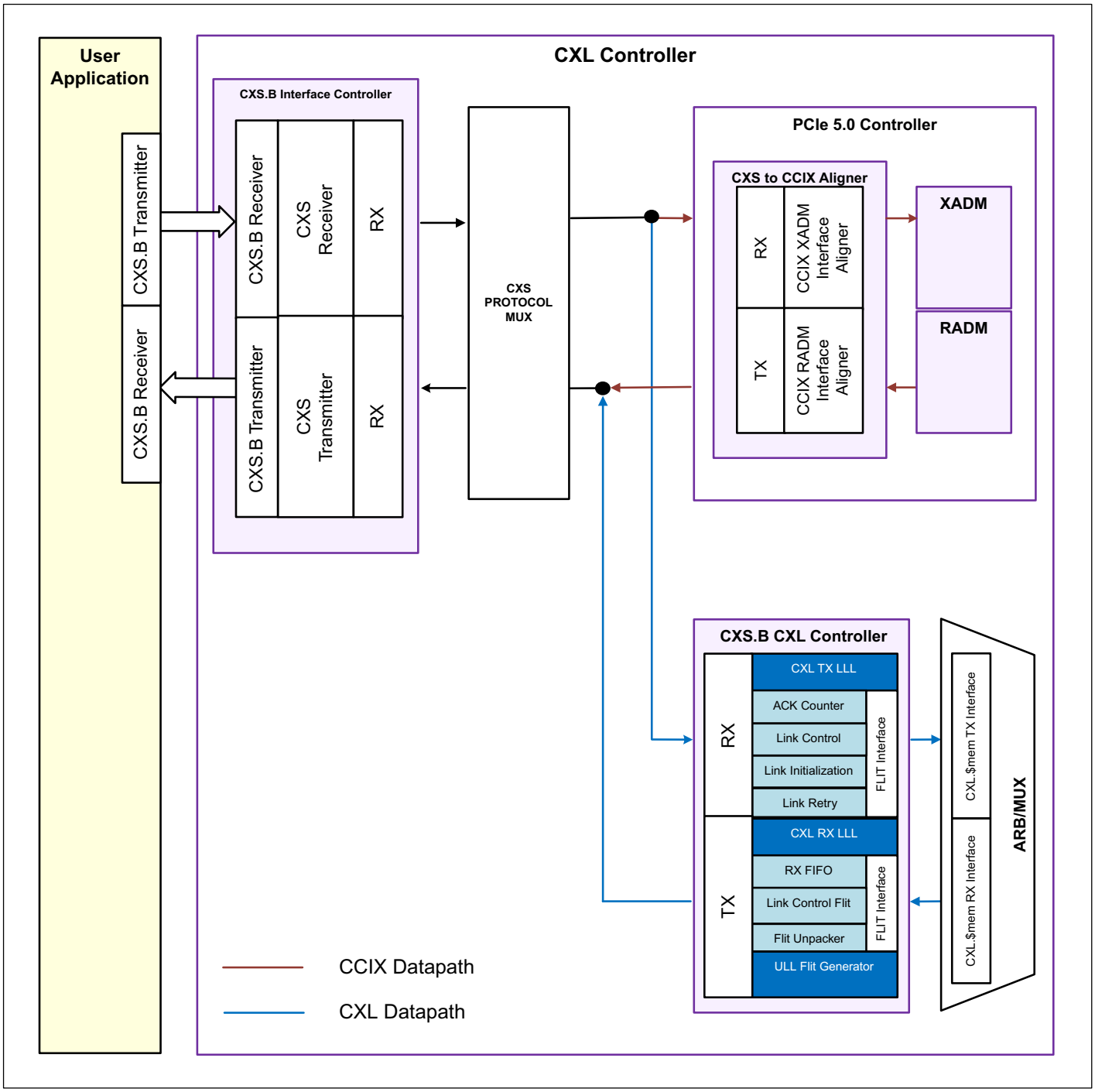
- The Host is not aware of Device LDs.
- The number of configured Physical Functions must be greater than total number of LDs supported + 1.
- If the Host initiates a Configuration write/read to its Device, it must provide a valid Bus, Device, and Function number.
- With CXL LD-ID hot reset, Device application with knowledge of LD-ID to function mapping must trigger CXL Reset for the corresponding functions. For more information on CXL Reset, see “CXL Reset” on page 100.

6

CXS.B CXL Controller

The CXS.B interface is an extended version of CXS interface which supports multiple protocols, for example CCIX 2.0 or CXL. The CXL controller supports CXS.B interface when you set the `CX_CXL_CXS_ENABLE` parameter to '1'. It is an optional interface that you can use to transmit/receive CXL outbound/inbound requests and LLCRD flits to the CXL controller. The following topics are discussed:

Figure 6-1 CXS.B CXL Link Layer Architecture



The CXS.B Interface controller controls the operation of the CXS.B interface. When link is negotiated in alternate protocol mode which enables CXL datapath, CXS.B Interface controller is attached to CXS.B CXL controller to handle CXL/CCIX2.0 protocols. When link is negotiated in PCIe mode, the CXS.B Interface controller is attached to PCIe datapath to handle CCIX protocol. The CXS protocol MUX arbitrates between the CCIX and CXL datapath and directs it to the respective destination based on the link mode.

6.1 Features and Limitations

The CXS.B CXL controller supports the following features:

- Compliant with the *Compute Express Link Specification, Revision 3.0, Version 0.7*
- Compliant with ARM CXS Interface Specification
- Compliant with ARM CXS Issue B updates defined to support multiple protocol streams
- Compliant with ARM CML Data Link Layer (DLL) to support CXL protocol over 512b CXS flits
- Compliant with ARM CCIX protocol implemented over 64B CXL flits.
- Supports all relevant features of the CXL controller: Virtual Link State Control, Link Initialization, Link Retry, Viral, RAS, and so on
- Supports single-protocol CXL-only configurations
- Supports multi-protocol CXL and CCIX over PCIe configurations

The CXS.B CXL controller has the following limitation:

- Concurrent operation of CXS Interface controller over both CXL and PCIe datapath is not supported

6.2 CXS.B CXL Link Layer

When the CXS interface is used for outbound/inbound CXL traffic, CXL Link Layer functionality defined in CXL Specification is split between CXL Tx/Rx Lower Link Layer (LLL) and Upper Link Layer (ULL).

The Lower Link Layer (LLL) is responsible for,

- Link Layer Initialization
- Link Layer Retry Handling
- CRC Handling
- Communication of acknowledge information using Ack bit insertion and Ack forcing mechanism
- Detection of Link Layer credit flits (LLCRD) and CXL protocol flits and forwarding them to ULL on the inbound side through the CXS.B interface.
- Filtering of malformed or errored flits (for example, flits with CRC error)
- Detection and triggering of Viral mode

Upper Link Layer is responsible for,

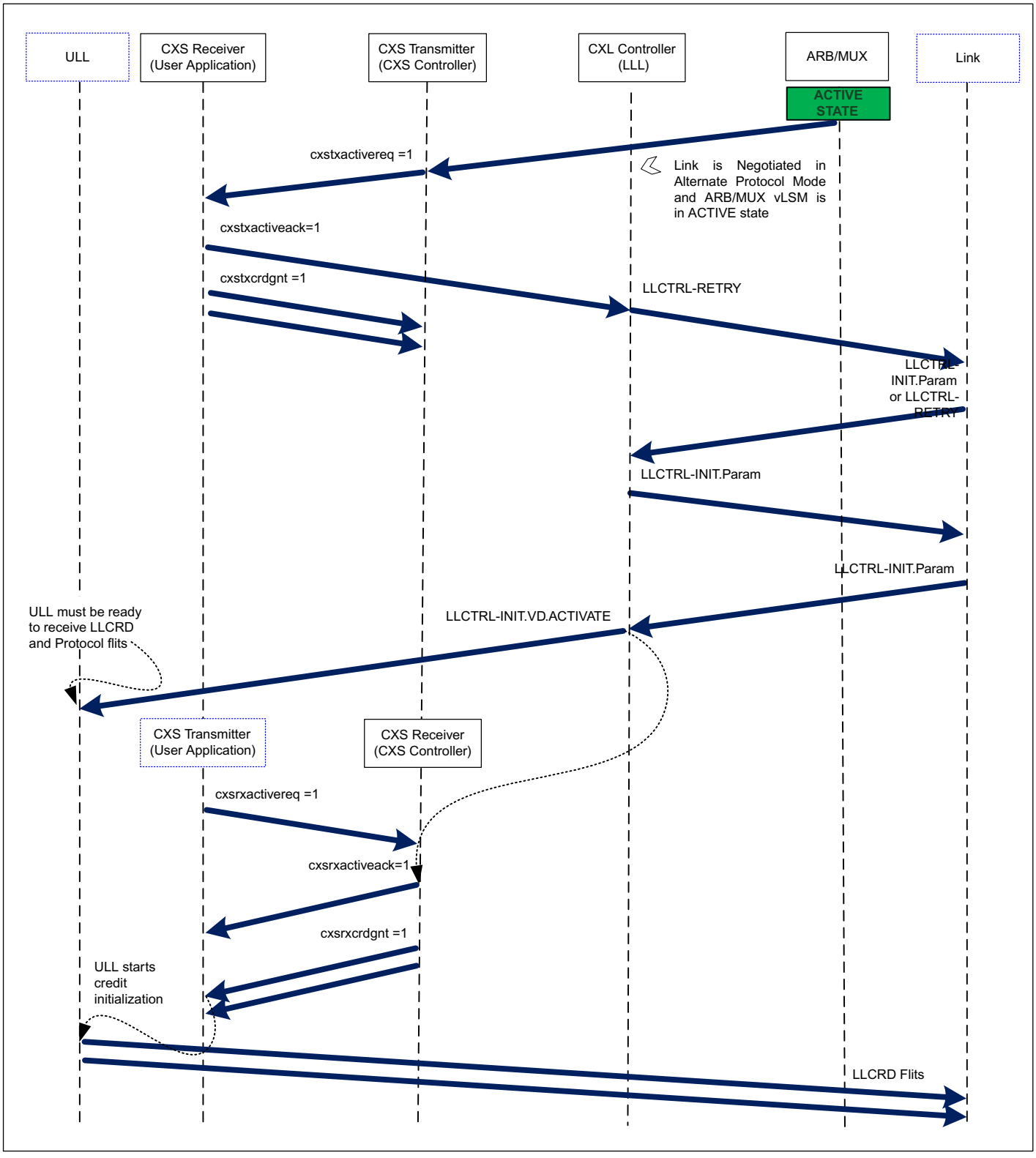
- Handling of CXL protocol flits which includes packing and unpacking of CXL messages
- Link layer credit management and receive buffers implementation
- Viral handling

6.2.1 Initialization

The following steps are performed as a part CXL Link Layer initialization sequence when the CXS interface is enabled:

- If link is negotiated in alternate protocol mode and CXL link is enabled, CXS controller asserts `cxstx-activereq` to activate CXS Transmit path after ARB/MUX vLSM moves to the ACTIVE state.
- When ULL acknowledges CXS TX activation request by asserting `cxstxactiveack`, the CXL controller starts CXL Link Layer initialization.
- The CXL controller sends LLCTRL-RETRY flit and waits for either LLCTRL-INIT.Param or LLCTRL-RETRY flit to be received.
- When the CXL controller receives LLCTRL-INIT.Param or LLCTRL-RETRY flit, it sends LLCTRL-INIT.Param flit.
- After successful exchange of LLCTRL-INIT.Param flits, the CXL controller allows CXS controller to assert `cxsrxactiveack` in response to `cxsrxactivereq`.
- The CXL controller synthesizes and sends LLCTRL-INIT.VD.ACTIVATE flit to ULL.
- On receiving LLCTRL-INIT.VD.ACTIVATE flit, ULL must start granting RX link credits through LLCRD flits.

Figure 6-2 CXS.B CXL Link Layer Initialization



The CXL controller reports any uncorrectable error condition detected by the CXL Tx/Rx LLL during link layer initialization using the CXL_FATAL_ERR_CR_REG register. For more information, see the *Reference Manual*.

6.2.2 Normal Operation

During normal operation, the following flits are exchanged between ULL and CXL Tx/Rx LLL.

- Protocol flits
- LLCRD flits

The following flow control rules are set between ULL and CXL Tx/Rx LLL.

Inbound Path

- ULL sinks any protocol or link flit once they are put on the CXS interface and immediately return CXS credits to LLL.
- ULL provides enough CXS credits and does not put any back pressure on the CXS link.

  Any violation of this rule may result in the bandwidth loss because the CXL Rx LLL drops the flits and triggers Retry mechanism if there is a back pressure from ULL. To minimize the impact of potential bubbles in the credit grants from the ULL, the CXL controller implements a buffer on CXS TX Interface that handles temporary back pressure from ULL.

Outbound Path

- ULL expects CXL controller to back pressure CXS interface at any time.

6.2.3 LLL Ack Insertion/Ack Forcing

During normal operation, the CXL Tx LLL is responsible for tracking and communicating Acknowledge information through protocol and LLCRD flits. The CXL Tx LLL inserts ACK bit on an outbound protocol or LLCRD flit as specified in the CXL Specification. If there are no protocol or LLCRD flits from ULL, the CXL Tx LLL synthesizes dummy LLCRD flit with credit return fields set to 0, to communicate Acknowledge information.

The CXL Rx LLL extracts Acknowledge information from a protocol or LLCRD flit to keep track of flits in the retry buffer. The CXL Rx LLL forwards all protocol and LLCRD flits which carry credit return information to ULL. Any LLCRD which does not contain credit return information is filtered out and is not propagate to ULL.

6.2.4 ULL Flit Generator

The CXL Tx/Rx LLL synthesizes control flits on CXS interface to communicate Link Initialization, Link Deactivation, and Viral mode information to ULL. The ULL Flit Generator is implemented as a simple link state control Finite State Machine (FSM) which inserts link control flits on transitions between states.

Figure 6-3 ULL Flit Generator

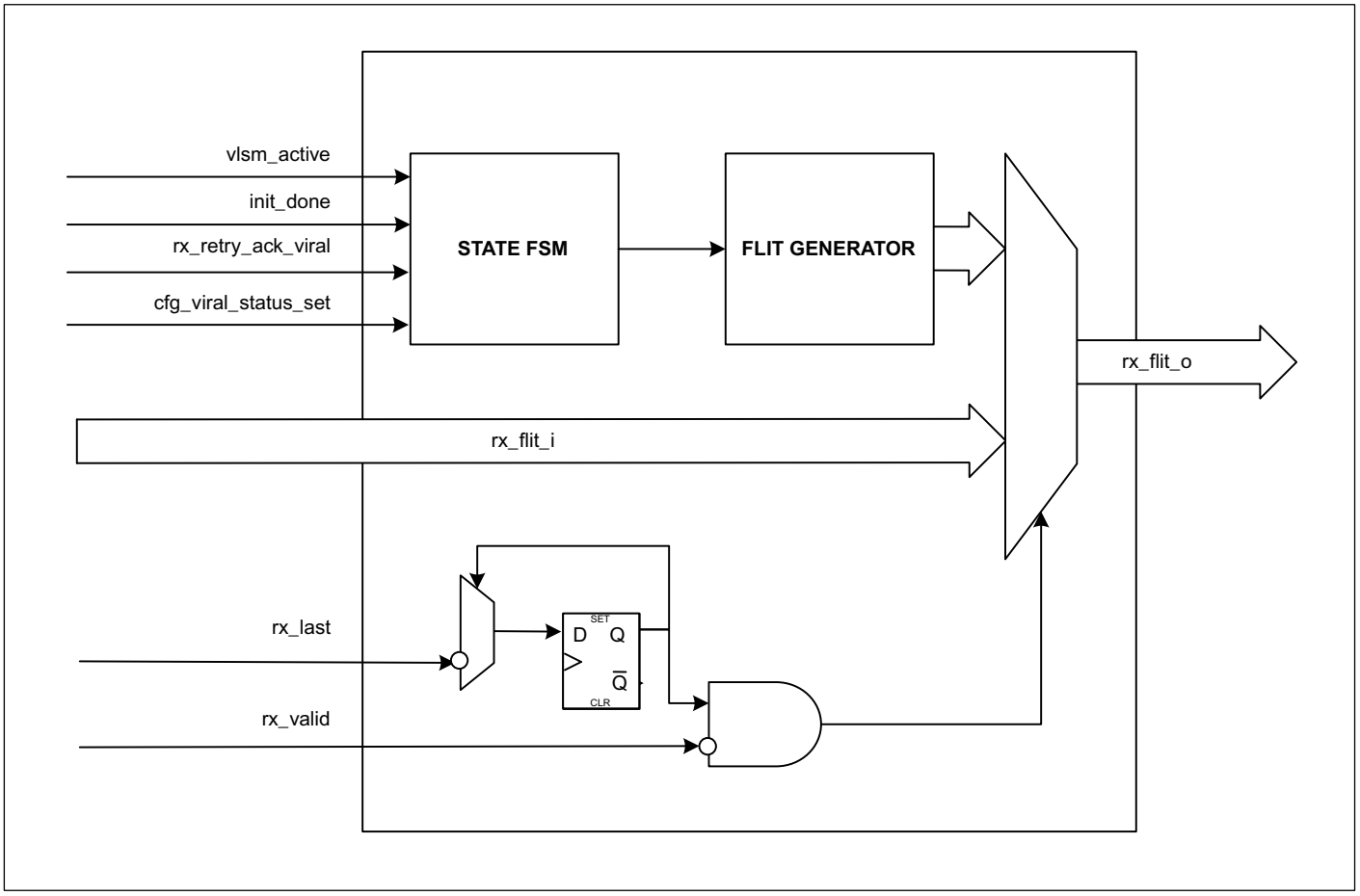


Table 6-1 ULL Flit Generator FSM State Transitions

Current State	Next State	Action
RESET	INIT	If VLSM is in ACTIVE state and the Link Initialization is complete, (LLCTRL-INIT.PARAM flit is received) transit to INIT state
INIT	ACTIVE	Synthesize INIT.VD.ACTIVATE flit on CXS.B Interface and transit to ACTIVE state
ACTIVE	DEINIT	If VLSM is no longer in ACTIVE state transit to DEINIT state
DEINIT	RESET	Synthesize INIT.VD.DEACTIVATE flit on CXS.B Interface and transit to ACTIVE state
ACTIVE	VIRAL	If either Retry.Ack flit is received with Viral Status or Viral Status bit is set by the CXL controller upon detection of Uncorrectable Error, transit to VIRAL state to communicate Viral Mode information to ULL
VIRAL	ACTIVE	Synthesize empty LLCRD flit with ENDERROR[0] bit set on CXS.B Interface

LLCTRL-INIT.VD is vendor defined INIT flit with “reserved” subtype encoding (for example, 1110) which is used to communicate CXL link status information from lower link layer (LLL) to ULL. LLCTRL-INIT.VD flit format is shown in Table 6-2.

Table 6-2 ULL. LLCTRL-INIT.VD Flit Format

Type	LLCTRL	Sub Type	Sub Type Description	Payload	Payload Description
INIT	1100	1110	INIT.VD	[3:0]	RSVD
				[5:4]	0'b01: Link ACTIVATE
					0'b10: Link DEACTIVATE
				[63:6]	RSVD

When the ULL Flit Generator FSM is not in the RESET state it propagates the received flits to CXS interface. RX Link Layer flits have a priority over generated flits within ULL Flit Generator, hence synthesized flits are inserted only when there are no pending flits on the RX Link Layer outputs.

6.2.5 ULL Viral Handling

Viral mode is an error containment mechanism. Viral information is communicated using “End Error” on the CXS.B interface control bus in both direction (that is, LLL to ULL and ULL to LLL).

Outbound Path

In the outbound path, ULL uses CXS EndError to communicate “viral” information to CXL Tx LLL. This EndError can be set on any flit sent by ULL to CXL Tx LLL. During normal operations these flits can be either protocols or LLCRD flits.

On receiving the “viral” indication from either ULL or when CXL Tx LLL itself runs into a Viral condition, CXL controller sets the `Viral_Status` bit in `CXL_RCIEP_FLEXBUS_CNTRL_STATUS_OFF` register and forces CRC error on the next outgoing flit. If there is no traffic to send, the transmitter inserts a LLCRD flit with a CRC error.

When the link partner initiates CRC error recovery flow by sending Retry.Req, the CXL Tx LLL generates Retry.Ack control flit with embedded Viral Status information. If Viral is triggered due to the internal CXL Tx LLL error and not by ULL, the corresponding error which triggered Viral condition is reported as UIE through AER.

Inbound Path

In inbound path, when CXL Rx LLL receives Retry.Ack flit with viral indication, it synthesizes LLCRD flit with CXS EndError bit set on the CXS control bus and sends it to ULL. The CXL controller drops all the errored and malformed flits on the inbound side and hence CXS EndError on LLCRD flit is detected as viral.

ULL reports this “viral” through CMN error reporting mechanisms.

6.3 CXS.B CXL Link Layer Registers

Table 6-3 describes Link Layer Capability registers defined by the CXL specification. For more information, see the CXL specification and DWC_PCIE_DSP/PF0_MEMBAR0_LINK_CAP registers in PCIe Register Descriptions PDF.

Table 6-3 CXL Link Capability Structure

CXL Register Name	Offset	PCIe Register
CXL Link Layer Capability Register	0x0	MEMBAR0_LINK_LAYER_CAP_REG_OFF MEMBAR0_LINK_LAYER_CAP_2_REG
CXL Link Control and Status Register	0x8	MEMBAR0_LINK_CNTRL_STATUS_REG_OFF
CXL Link Rx Credit Control Register	0x10	MEMBAR0_LINK_RX_CR_CNTRL_REG_OFF MEMBAR0_LINK_RX_CR_CNTRL_2_REG_OFF
CXL Link Rx Credit Return Status Register	0x18	MEMBAR0_LINK_RX_CR_RTN_STATUS_REG_OFF MEMBAR0_LINK_RX_CR_RTN_STATUS_2_REG_OFF
CXL Link Tx Credit Status Register	0x20	MEMBAR0_LINK_TX_CR_STATUS_REG_OFF MEMBAR0_LINK_RX_CR_STATUS_2_REG_OFF
CXL Link Ack Timer Control Register	0x28	MEMBAR0_LINK_ACK_TMR_CNTRL_REG_OFF
CXL Link Defeature	0x30	MEMBAR0_LINK_DEFEATURE_REG_OFF

The following registers are out of the CXL controller scope as they are fully implemented in ULL.

The application logic must forward access requests to these registers directly to the ULL instead of DBI.

- CXL Link Rx Credit Control Register
- CXL Link Rx Credit Return Status Register
- CXL Link Tx Credit Status Register
- CXL Link Defeature

Remaining all registers are implemented in PCIe controller register space.

6.4 CCIX Over 64B CXL Flits

The CXL controller optionally supports alternate protocols over CXL Link, particularly CCIX protocol implemented over 64B CXL Flits. To enable transport of CCIX messages over CXL Link, you must set the parameter `CX_CXL_CCIX_ENABLE` to '1'.

When CCIX protocol is enabled (`CXL_AP_PROT_EN =1` and `CXL_AP_PROT_TYPE =000(CCIX)`in the CXL Link Layer Control register `CXL_LL_CTRL_CR`), the CXL controller Link Layer supports the following:

- Transport of CCIX protocol messages sent on 512-bit (64B) wide Flit (protocol flit)
- Two types of CCIX protocol Flits
  - Header Flits (Flit with Header)
  - Data Chunk (Flit without Header): Information from the Header is used to identify Flits without header. Maximum of four Header less Flits can follow a Flit with Header.
- Transport of CCIX LLCRD messages as 512-bit Flits (Link Control Flit)
- CXL Link control Flits which are used for link layer initialization, retry handling, and Flow control.

When the protocol switches from CXL to CCIX, the CXL controller uses the message format shown in Figure 6-4 for protocol flit and Figure 6-5 for LLCRD flit to implement flow control.

Figure 6-4 ULL CCIX Protocol Flit Header Format

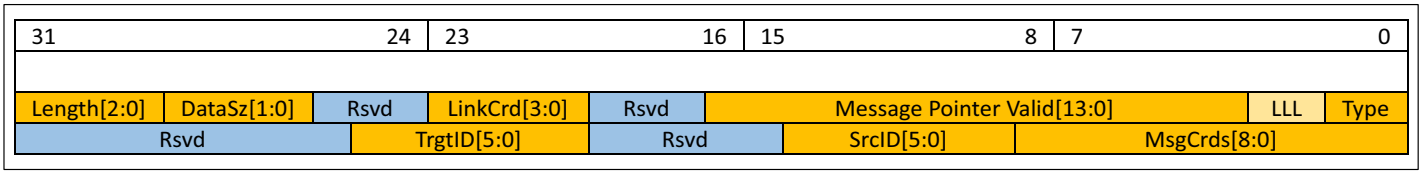
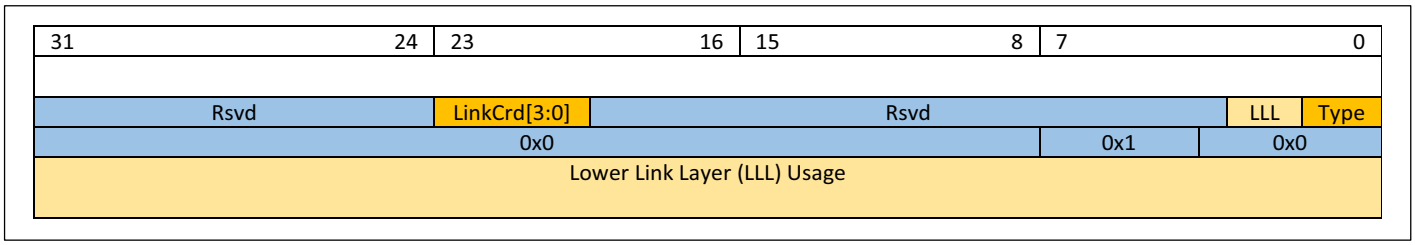


Figure 6-5 ULL CCIX LLCRD Flit Header Format



The Flit Header for CCIX Protocol and LLCRD Flits carry the following information:

- **Type Field:** It differentiates between a Protocol Flit and Link Flit.
- **Length Field:** It conveys the number of 64B "without Header" Flits which follows the current Flit. In other words, the number of header less Flits which are expected before the next Flit with Header.

> **Note** Current specification allows up to four Header-less Flits. So, the range allowed in the "Length Field" is 0 to 4.

- **LLL labeled Fields:** These are available for Lower Link Layer usage. Particularly, CXL controller uses these fields to communicate Acknowledge information with protocol and LLCRD Flits following the CXL specification.

6.5 CXL Flit Transfers Over CXS Interface

Each CXL flit is transmitted on the CXS interface as a single CXS packet incorporated in 512-bit CXS flit. Table 6-4 describes the CXS control bus layout when the CXS flit contains only one CXS packet (CXL flit).

Table 6-4 CXS Control Bus

CXCSNTL	CXSCNTL Bits	Value
START[1:0]	CXSCNTL[1:0]	01b
START0PTR[1:0]	CXSCNTL[3:2]	00b
START1PTR[1:0]	CXSCNTL[5:4]	00b
END[1:0]	CXSCNTL[7:6]	01b
ENDERROR[1:0]	CXSCNTL[9:8]	00b or 01b (Viral)
END0PTR[1:0]	CXSCNTL[13:10]	1111b
END1PTR[1:0]	CXSCNTL[17:14]	0000b

CXS protocol type (`CXSPRCLTYPE [2:0]`) must be set to 001b, when CXS interface is enabled for the CXL controller (`CX_CXL_ENABLE =1` and `CX_CXL_CXS_ENABLE =1`). CXS protocol type (`CXSPRCLTYPE [2:0]`) must be set to 000b when CXS interface is enabled for the CCIX controller (`CX_CCIX_ENABLE =1` and `CX_CXS_ENABLE =1`).

CXL protocol flits with header and protocol flits without header (that is, All Data Flit (ADF)) are sent as a stream and CXSLAST is be used to indicate end of a stream. CXL controller inserts link layer control flits at a stream boundary indicated by CXSLAST.

Any valid CXS flit after CXSLAST must be a flit with header so that the CXL controller can use this information to insert Ack indication in the flit header.

Table 6-5 CXL LLCRD and Protocol Flit Transmission Over CXS Interface

	Cycle												
Signal	0	1	2	3	4	5	6	7	8	9	10	11	12
CXSVALID	0	1	1	1	1	0	1	0	1	0	1	1	1
CXSDATA[511:0]	N/A	LLCRD	HDR	ADF	HDR	N/A	HDR	N/A	ADF	N/A	ADF	ADF	LLCRD
CXSLAST	N/A	1	0	1	1	N/A	0	N/A	0	N/A	0	1	1
CXSPRCLTYPE[1:0]	N/A	1	1	1	1	N/A	1	N/A	1	N/A	1	1	1

6.6 CXS Interface Configuration

Table 6-6 summarizes CXS interface configuration parameters for CXS.B interface.

Table 6-6 CXS Interface Configuration Parameters for CXS.B Interface

Parameter	Value	Description
CX_CXS_DATAFLITWIDTH	512	CXL controller may be enabled only over 512b CXS Interface.
CX_CXS_MAXPKTPERFLIT	Any Value	**Note:** Only one CXL flit must start and end in CXS flit. The default value is set to 2 for compliance with CXS Issue B, still any value between 2-4 is supported for CXS.B CXL controller.
CX_CXS_RX_CONTINUOUSDATA	Any Value	Does not have any impact when CXS interface is attached to CXL Data Link.
CX_CXS_TX_CONTINUOUSDATA	Any Value	Does not have any impact when CXS interface is attached to CXL Data Link.
CX_CXS_PRCL_ENABLE	FALSE or TRUE	Enables protocol type selection - CXSPRCLTYPE on CXS interface. Enabled only when CXS interface is configured to support more than one protocol.
CX_CXS_LAST_ENABLE	TRUE	Enables last of sequence indication - CXSLAST on CXS interface. Always enabled for CXS.B interface.

The Table 6-7 summarizes the configuration parameters that may help to reduce round-trip latency on CXL controller. There might be a trade-off between reduced round-trip latency and timing closure.

Table 6-7 Configuration Parameters for Reducing CXL Controller Round-Trip Latency

Parameter	Value	Description
CX_CXS_TX_OUTPUT_REG	FALSE	Disables output register on CXS TX Interface
CX_CXS_RX_INPUT_REG	FALSE	Disables input register on CXS RX Interface
CX_CXL_CXS_FIFO_BYPASS_ENABLE	TRUE	Enables CXS RX/TX FIFO Bypass Feature for CXL Data Link
CX_CXS_RAM_PIPELINE_ENABLE	FALSE	Disables pipeline register on CXS TX/RX RAM output

7

CXL 2.0 Integrity and Data Encryption (IDE)

The CXL 2.0 controller supports CXL IDE as defined in CXL 2.0 specification. To enable CXL IDE support in the CXL controller, set the internal parameter `CX_CXL_IDE_ENABLE` to '1'. The CXL controller IDE along with DesignWare® CXL 2.0 Integrity and Data Encryption (IDE) Security IP Module provides Confidentiality, Integrity, and Replay protection for data transiting the CXL link. The following topics are discussed:

7.1 Features and Limitations

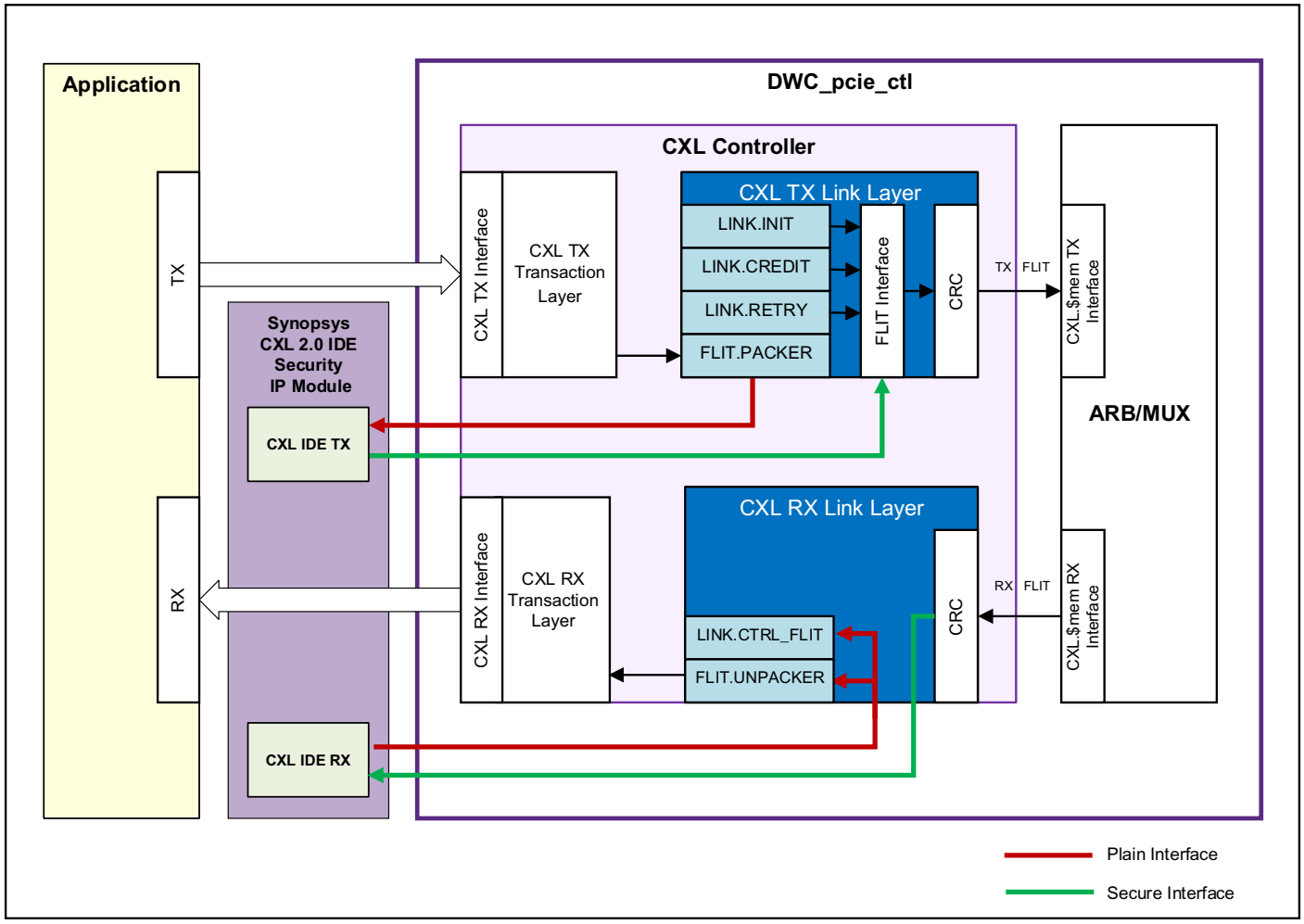
The CXL controller,

- Only supports IDE for CXL controllers with Native interface
- Supports Skid and Containment mode
- Seamlessly integrates with DesignWare CXL 2.0 Integrity and Data Encryption (IDE) Security IP Module

7.2 Architecture

The CXL controller interacts with CXL 2.0 IDE Security Module through TX/RX Plain and Secure interfaces. Figure 7-1 illustrates the high-level data flow between CXL controller and CXL 2.0 IDE Security IP Module.

Figure 7-1 CXL 2.0 IDE Security IP Module and CXL Controller Top-Level Data-Flow Diagram



CXL controller implements a transmit datapath to the CXL 2.0 IDE Security IP Module between the Flit Packer and Flit Interface. It consists of a pair of control and data signals, to and from CXL 2.0 IDE Security IP Module, referred as TX Plain interface (plain data) and TX Secure interface (encrypted data).

CXL controller implements a receive datapath to CXL 2.0 IDE Security IP Module between the CRC Checker and Flit Unpacker. It consists of a pair of control and data signals, referred to as RX Secure interface (encrypted data) and RX Plain interface (plain data), to and from CXL 2.0 IDE Security IP Module respectively. For Native CXL interface configuration, only the protocol flits must propagate through CXL 2.0 IDE Security IP Module to the RX Plain interface. For CXS+CXL[1] controller configuration, the LLCRD link control flits must also propagate through CXL 2.0 IDE Security IP Module to the RX Plain interface.

1. The 5.97a release does not support CXS IDE feature.

7.3 CXL Controller IDE Modes

The CXL controller supports the Containment and Skid mode defined in the CXL 2.0 specification. The CXL controller also supports the Bypass mode that the CXL 2.0 IDE Security IP Module supports. In addition to that, the CXL controller supports a Disable mode.

Table 7-1 IDE Mode Configuration and Operation

IDE Mode	Parameter Configuration	Input Signal Value[a]	Controller Operation
Containment	CX_CXL_IDE_CONT_ENABLE =1	cxl_ide_mode =0	Controller releases the data for further processing only after the integrity check passes.Controller propagates all the extracted messages from the incoming flit to the Application after performing the integrity check for the given MAC_Epoch.
Skid	CX_CXL_IDE_SKID_ENABLE =1	cxl_ide_mode =1	Controller releases the data for further processing without waiting to receive and check the integrity value. Controller propagates all the extracted messages from the incoming flit to the Application without waiting for the integrity check to complete.
Bypass	NA	cxl_ide_bypass =1	The IDE logic in the CXL controller is disabled. All TX and RX flit pass through the CXL 2.0 IDE Security Module, so the CXL 2.0 IDE Security Module latency impacts the TX and RX datapath latency.
Disable	NA	cxl_ide_disable =1	The IDE logic in the CXL controller is disabled.TheCXL controller multiplexes the datapath, so the CXL 2.0 IDE Security Module latency does not impact the TX and RX datapath latency.

a. The CXL 2.0 IDE Security Module drives all the input signals except cxl_ide_disable. Your application must drive the input signal cxl_ide_disable.

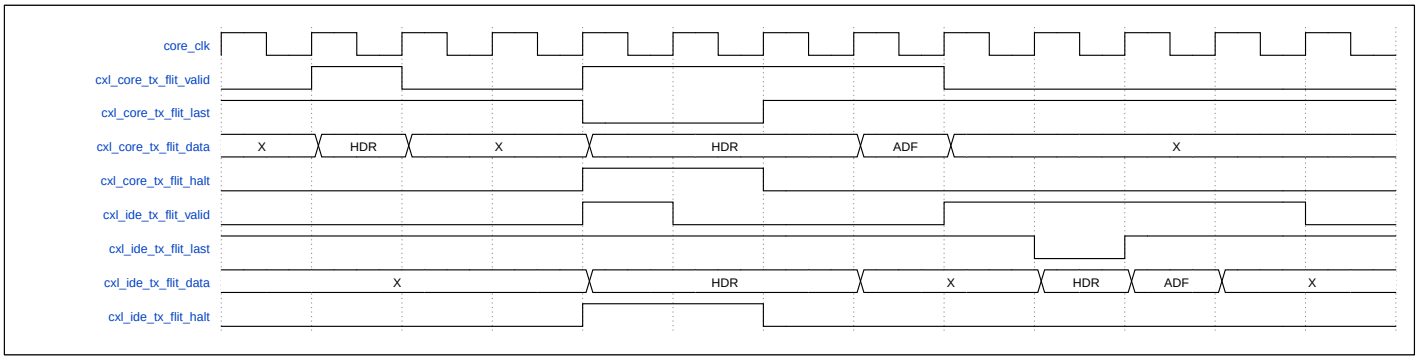
7.4 CXL Controller Transmit Datapath Operation

7.4.1 CXL 2.0 IDE Security IP Module TX Plain Interface Back Pressure

The CXL 2.0 IDE Security IP Module must assert `cxl_ide_tx_halt` to back pressure TX Plain interface. The CXL controller asserts `cxl_core_tx_halt` to back pressure TX Secure interface. The CXL 2.0 IDE Security IP Module must halt the TX Plain interface when TX Secure interface is under back pressure.

The Figure 7-2 illustrates an example of data transmission when the CXL controller halts the CXL 2.0 IDE Security IP Module.

Figure 7-2 CXL 2.0 IDE Security IP Module TX Plain Interface Back Pressure Example



7.4.2 MAC Placeholder Insertion

CXL controller with Native CXL interface implements the MAC placeholder insertion functionality. The CXL 2.0 IDE Security IP Module must replace the MAC placeholder with the pre-calculated integrity check value, for a previous MAC_Epoch.

The CXL transmitter transmits the MAC placeholder in the Slot0 header of Type H6. It sends the MAC placeholder on the first available Slot0 header immediately after the current MAC_Epoch is complete. The CXL transmitter transmits a maximum of five protocol flits belonging to the current MAC_Epoch before transmitting the MAC placeholder for the previous MAC_Epoch. The MAC placeholder contains 0's in the MAC field of H6 format.

The flit packer keeps track of the protocol flits within the MAC_Epoch. Every time the flit packer generates protocol flits equal to the Aggregation Flit Count (MAC_Epoch size), it inserts a MAC placeholder as described in the previous paragraph. The flit packer uses an Aggregation Flit Count of 5 in the Containment Mode. In Skid mode, it uses an Aggregation Flit Count of 128.

Figure 7-3 and Figure 7-4 illustrate the MAC placeholder insertion by CXL controller.

Figure 7-3 MAC Placeholder Insertion for Two Consecutive Epochs

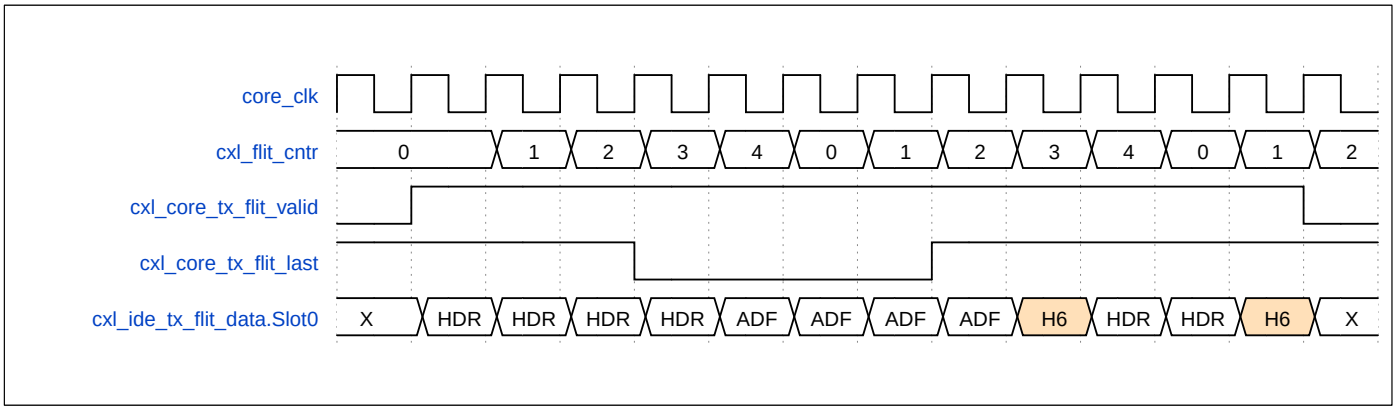
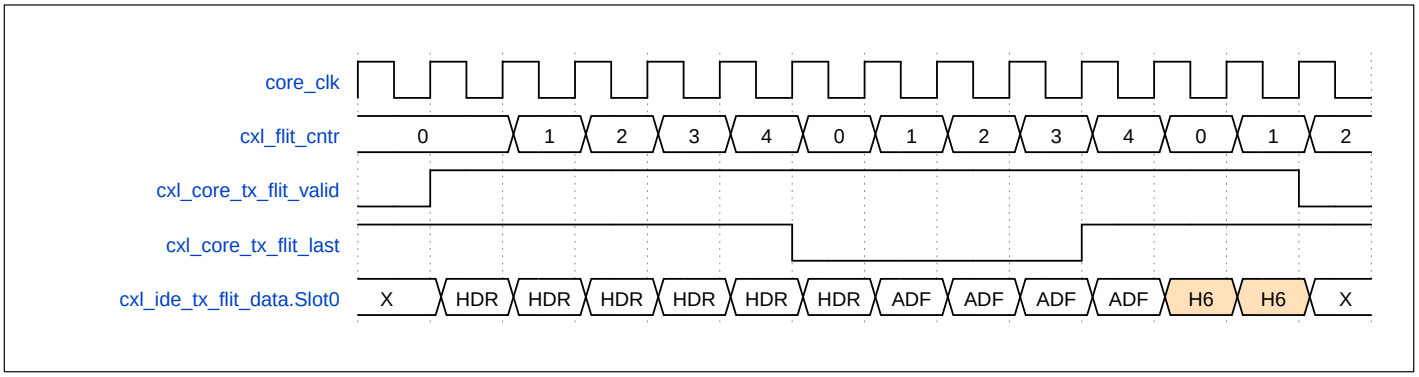


Figure 7-4 Back-To-Back MAC Placeholder Insertion for Two Consecutive Epochs



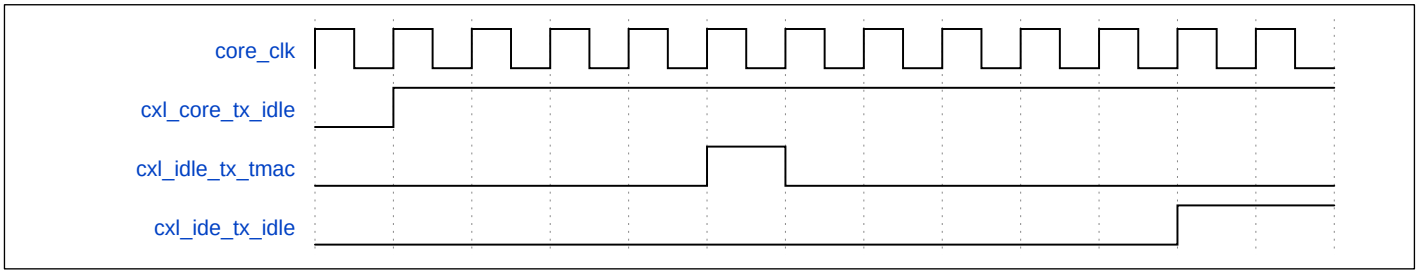
7.4.3 Early MAC Termination

The CXL controller can terminate the MAC_Epoch and transmit the MAC for the flits in a truncated MAC_Epoch when the transmitted flits are less than the Aggregation Flit Count of the flits in the current MAC_Epoch. That can happen, for example, when the link goes idle. The CXL controller asserts the `cxl_core_tx_idle` output signal indicating early MAC_Epoch termination to CXL 2.0 IDE Security IP Module.

When the controller asserts `cxl_core_tx_idle`, CXL 2.0 IDE Security IP Module can start transmitting IDE.Idle flits, as well as truncated MAC link control flit (IDE.TMAC). The CXL 2.0 IDE Security IP Module must assert the `cxl_ide_tx_tmac` input signal when it transmits IDE.TMAC. After the truncated MAC sequence is complete, the CXL 2.0 IDE Security IP Module must assert `cxl_ide_tx_idle` to confirm the CXL link idle condition.

Figure 7-5 illustrates the CXL controller and CXL 2.0 IDE Security IP Module handshake on early MAC-Epoch termination.

Figure 7-5 Early MAC_Epoch Termination



7.5 CXL Controller Receive Datapath Operation

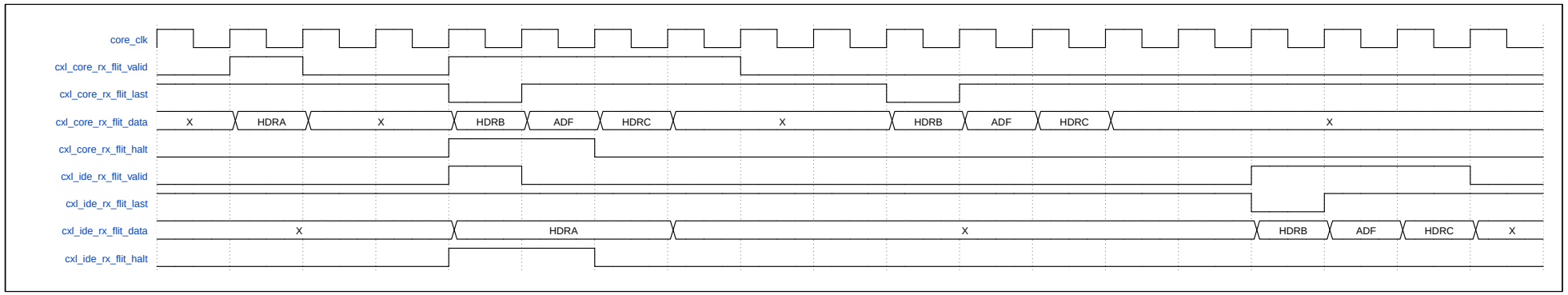
7.5.1 CXL 2.0 IDE Security IP Module RX Secure Interface Back Pressure

The CXL 2.0 IDE Security IP Module must assert `cxl_ide_rx_halt` to back pressure RX Secure interface and RX LLCTRL interface. The CXL controller drops any incoming retriable flits when the CXL 2.0 IDE Security IP Module asserts `cxl_ide_rx_halt`. CXL controller replays these dropped retriable flits using the CXL retry mechanism. So, the CXL controller triggers Retry.Req sequence when it drops retriable flit due to back pressure from CXL 2.0 IDE Security IP Module.

The CXL controller asserts `cxl_core_rx_halt` to back pressure the RX Plain interface. If CXL 2.0 IDE Security IP Module cannot consume flits on the RX Secure interface when the RX Plain interface is under back pressure, it must halt the RX Secure interface.

Figure 7-6 illustrates an example of data transmission when CXL controller halts CXL 2.0 IDE Security IP Module.

Figure 7-6 CXL 2.0 IDE Security IP Module RX Secure Interface Back Pressure Example



7.6 CXL 2.0 IDE Security IP Module Datapath Protection

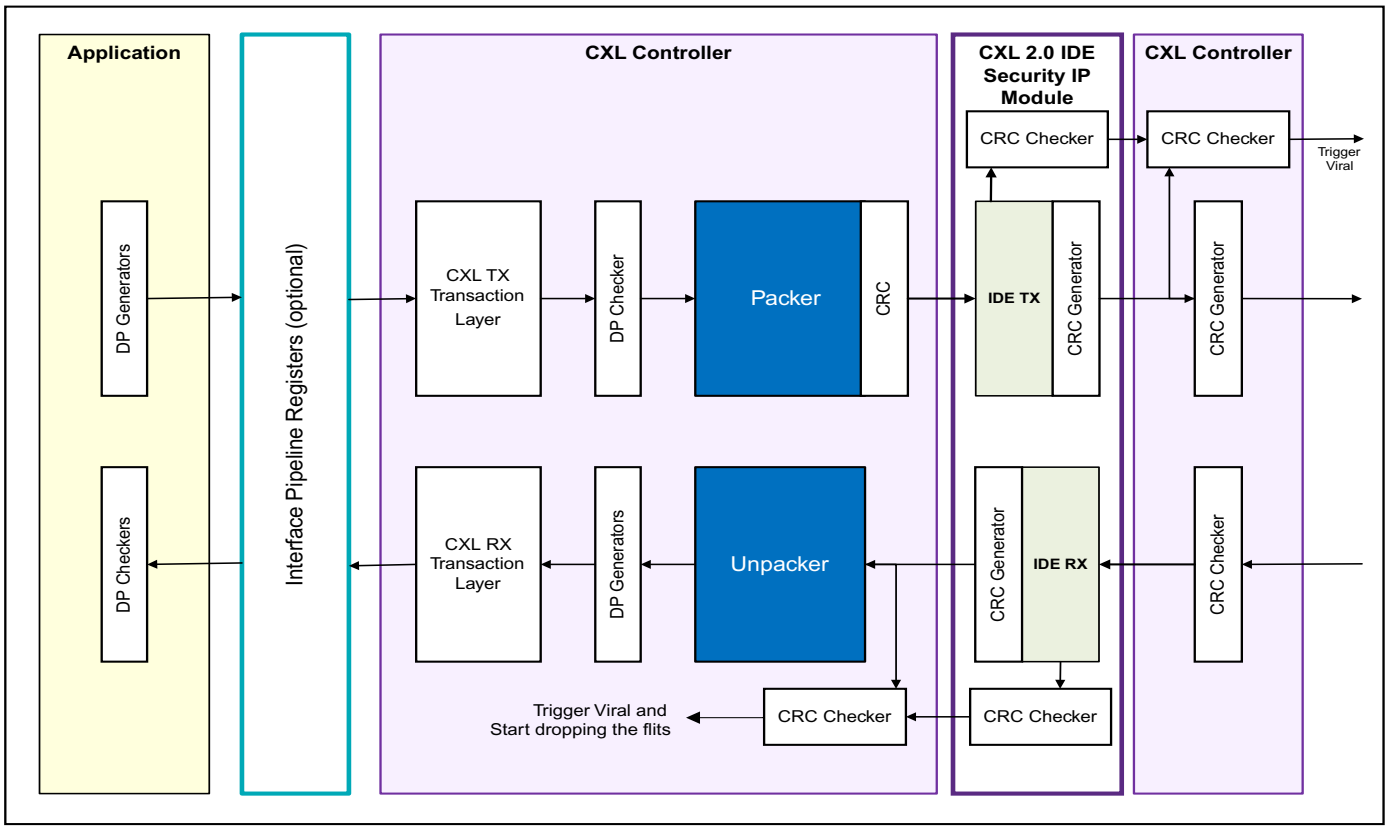
To enable CXL 2.0 IDE Security IP Module datapath protection set the internal parameter `CX_CXL_IDE_DP_ENABLE` to '1'. When you enable CXL 2.0 IDE Security IP Module datapath protection the CXL controller adds 16 bit CRC to the IDE Receiver Plain interface (`cxl_ide_rx_flit_data`) and the IDE Transmitter Plain Interface (`cxl_core_tx_flit_data`) to protect the IDE datapath and maintain the integrity of the CXL controller and CXL 2.0 IDE Security IP Module.

CXL 2.0 IDE Security IP Module must check CRC on the TX Plain interface (`CXL_IDE_TX_PLAIN_IF_ERR`) and RX Secure interface(`CXL_IDE_RX_SECURE_IF_ERR`) and report CRC error to the CXL controller on `cxl_ide_tx_crc_error` and `cxl_ide_rx_crc_error` input pins. CXL controller registers the error in the `CXL_FATAL_ERR_CR_REG` register and triggers a Viral condition. CXL 2.0 IDE Security IP Module must ensure that bad flits do not propagate to the CXL controller on RX Plain interface (`CXL_IDE_RX_PLAIN_IF_ERR`) when it detects CRC Error on (`CXL_IDE_RX_SECURE_IF_ERR`).

CXL 2.0 IDE Security IP Module must re-generate 16-bit CRC on TX Secure (`CXL_IDE_TX_SECURE_IF_ERR`) and RX Plain Interface (`CXL_IDE_RX_PLAIN_IF_ERR`) to the CXL controller. CXL controller performs CRC check and, if it detects an error, the CXL controller registers the error in the `CXL_FATAL_ERR_CR_REG` register, triggers a Viral condition, and starts dropping flits followed by the one with CRC error.

Figure 7-7 illustrates CXL 2.0 IDE Security IP Module datapath protection architecture.

Figure 7-7 CXL 2.0 IDE Security IP Module Datapath Protection Architecture



CXL controller datapath protection is discussed in "CXL Controller RAS Data Protection" on page 87.

7.7 CXL 2.0 IDE Security IP Module Error Injection

The CXL 2.0 IDE Security IP Module implements the IDE Error Injection mechanisms. The controller uses `CXL_IDE_EINJ_OFF` port logic register to control the CXL 2.0 IDE Security IP Module Error Injection. For more information, see the `CXL_IDE_EINJ_OFF` section in the "Register Descriptions" chapter of the *Reference Manual*.

The controller supports the following error injection mechanisms:

- "CXL 2.0 IDE Security IP Module MAC Error Injection" on page 168
- "CXL 2.0 IDE Security IP Module Datapath Protection Error Injection" on page 168

7.7.1 CXL 2.0 IDE Security IP Module MAC Error Injection

CXL controller implements MAC placeholder insertion into the protocol flit. This MAC placeholder is replaced with a calculated MAC integrity check by the CXL 2.0 IDE Security IP Module. CXL controller implements the following MAC Error Injections:

- **Insert MAC** - Inserts an additional instance of MAC placeholder into the Epoch
- **Delete MAC** - Deletes MAC placeholder instance in the Epoch
- **Delay MAC** - Delay MAC placeholder insertion by N positions in the Epoch

MAC error injection results in a corresponding error to be reported in CXL Error Status register implemented in the CXL 2.0 IDE Security IP Module. The error is reflected back to the CXL controller which reports IDE RX Error in the `CXL_FATAL_ERR_CR_REG_OFF` register, triggering the Viral mode in the CXL controller.

7.7.2 CXL 2.0 IDE Security IP Module Datapath Protection Error Injection

CXL controller sets `CX_CXL_IDE_DP_ENABLE =1` to implement datapath protection on TX/RX Secure and Plane Interfaces between CXL controller and CXL 2.0 IDE Security IP Module.

CXL controller implements following error injection capabilities on CXL 2.0 IDE Security IP Module TX Plain and IDE RX Secure Interfaces to trigger corresponding datapath protection error in CXL 2.0 IDE Security IP Module:

- **TX CRC Error Injection** - Error injected on the TX Plain IF to the CXL 2.0 IDE Security IP Module
- **RX CRC Error Injection** - Error injected on the RX Secure IF to the CXL 2.0 IDE Security IP Module

CXL controller allows up to three errors (`IDE_EINJ_CNT`) to be injected in the flit with one, two, or three bits flipped in the CRC field (`IDE_EINJ_CRC_BITS`).

7.7.2.1 IDE TX CRC Error Injection

1. Write `IDE_EINJ_CMD =0x4` (CRC TX Error Injection) into CXL 2.0 IDE Security IP Module Error Injection Port Logic register (`CXL_IDE_ERR_INJ`).
2. CXL transmitter corrupts flit on the CXL 2.0 IDE Security IP Module TX Plain interface.
3. CXL 2.0 IDE Security IP Module Core detects Datapath error and reports it to the CXL controller by asserting `cxl_ide_tx_crc_error`.
4. CXL controller registers the error in the CXL Fatal Error register (`CXL_FATAL_ERR`) and triggers a Viral mode in the controller.

7.7.2.2 IDE RX CRC Error Injection

1. Write `IDE_EINJ_CMD =0x5` (CRC RX Error Injection) into CXL 2.0 IDE Security IP Module Error Injection Port Logic register (`CXL_IDE_ERR_INJ`).
2. CXL receiver corrupts flit on the CXL 2.0 IDE Security IP Module RX Secure interface.
3. CXL 2.0 IDE Security IP Module Core detects datapath error and reports it to the CXL controller by asserting `cxl_ide_tx_crc_error =1`.
4. CXL controller registers the error in the CXL Fatal Error register (`CXL_FATAL_ERR`) and triggers a Viral mode in the controller.

8

Signal/Parameter/Register Descriptions

For CXL Device Port Signal/Parameter/Register descriptions, see the PCIe Controller EP Reference Manual (DWC_pcie_ctl_ep_reference.pdf).

For CXL Host Port Signal/Parameter/Register descriptions, see the PCIe Controller RP Reference Manual (DWC_pcie_ctl_rp_reference.pdf).

For CXL Dual Mode port Signal/Parameter/Register descriptions, see the PCIe Controller DM Reference Manual (DWC_pcie_ctl_dm_reference.pdf).

For CXL Switch Port Signal/Parameter/Register descriptions, see the PCIe Controller SW Reference Manual (DWC_pcie_ctl_sw_reference.pdf).