SYNOPSYS®

January 24, 2023

Design & Implementation of CXL 2.0 (DLL & TL of CXL.Mem & CXL.Cache) Controller Supporting ARM AMBA CXS Interface & CXL Native Interface

Abdelrahman Mohammed Mahdy, Mahmoud Saeed Ismaeel, Mostafa Hassanien Ahmed, Omar Tarek Amer, Shadi Gamal El-Sayed

Communication and Computer Engineering Program (Electronics Track)

Cairo University

Sponsored by: **Si-Vision**

Supervised by: **Prof. Dr. Hassan Mostafa**

You always need to refer to the first four chapters from the official CXL 2.0 standard, you'll find its link in the references section to download it, for more information and elaboration since we're only sticking to this documentation from day one in our design and implementation for this project.

This project is considered as a Digital IC Design Project and it's under the sponsorship of Si-Vision and the supervision of Prof. Dr. Hassan Mostafa & Opto-Nano Electronics Laboratory in Cairo University.

Thank You!

Table of Contents

1. About The Sponsors Si-Vision & Synopsys 7

2. Terminology / Acronyms 8

3. Abstract 10

4. Motivation and Overview 11

5. What is Our GP Revolving Around? 12

5.1 Main Features 13

5.2 Limitations 13

5.3 Clock Requirements 13

6. CXL System Architecture 13

6.1 CXL Device Types 14

6.3.1 Type 1 CXL Device 14

6.3.2 Type 2 CXL Device 16

6.3.3 Type 3 CXL Device 18

7. Transaction Layer 19

7.1 CXL.Cache 19

7.2.1 Channel Ordering 21

7.2.2 Channel Crediting 21

7.2.3 CXL.Cache Transaction Description 22

7.2 CXL.Mem 24

7.3 Transaction Ordering Summary 25

7.4 Transaction Flows to Device-Attached Memory 26

8. Data Link Layer 31

9. Market Analysis 59

10. GP Estimated Cost, Tools, and Materials. 67

10.1 Tools 67

10.2 Materials 68

10.3 Cost 68

11. GP Timeline 69

12. GP Final Deliverables 72

13. References 73

List of Figures

Figure 1: Conceptual Diagram of Accelerator Attached to Processor via CXL ______ *11*

Figure 2: High-Level Block Diagrams for the CXL Controller ______ *12*

Figure 3: CXL Device Types ______ *14*

Figure 4: Type 1 CXL Device ______ *15*

Figure 5: Type 2 CXL Device ______ *16*

Figure 6: Type 2 Host Bias ______ *18*

Figure 7: Type 2 Device Bias ______ *18*

Figure 8: Type 3 CXL Device ______ *19*

Figure 9: CXL.Cache Channels ______ *20*

Figure 10: CXL.Cache Read Behavior ______ *23*

Figure 11: CXL.Cache Device to Host Write Behavior ______ *24*

Figure 12: Upstream Ordering Summary ______ *26*

Figure 13: Cachable Read from Host ______ *27*

Figure 14: Read for Ownership from Host ______ *27*

Figure 15: Non-Cachable Read from Host ______ *28*

Figure 16: Ownership Request from Host (NDR) ______ *28*

Figure 17: Flush from Host ______ *29*

Figure 18: Weakly Ordered Write from Host ______ *29*

Figure 19: Write from Host with Invalid Host Caches ______ *30*

Figure 20: Write from Host with Valid Host Caches ______ *30*

Figure 21: CPU Flex Bus Port ______ *32*

Figure 22: Conceptual Diagram of Flex Bus Layering ______ *34*

Figure 23: Flex Bus Layers - CXL.Cache + CXL.Mem Link Layer Highlighted ______ *35*

Figure 24:CXL.cache/. mem Protocol Flit Overview ______ *36*

Figure 25: CXL.cache/.mem All Data Flit Overview ______ *36*

Figure 26: Example of a protocol flit from device to host ____ *37*

Figure 27: CXL.cache/CXL.mem Flit Header Definition ____ *37*

Figure 28: Flit Header Type Encoding ____ *38*

Figure 29: Legal values of Sz and BE Fields ____ *39*

Figure 30: ReqCrd/DataCrd/RspCrd Channel Mapping ____ *40*

Figure 31: CXL.cache/CXL.mem Credit Return Encodings ____ *40*

Figure 32: Slot Format Field Encoding ____ *41*

Figure 33: H2D/M2S Slot Formats ____ *42*

Figure 34: D2H/S2M Slot Formats ____ *42*

Figure 35: H0 - H2D Req + H2D Resp	*Figure 36: H1 - H2D Data Header + H2D Resp + H2D Resp* *43*

Figure 37: H3 - 4 H2D Data Header ____ *44*

Figure 38: H2 - H2D Req + H2D Data Header ____ *44*

Figure 39: CXL.cache/CXL.mem Link Layer Control Types ____ *48*

Figure 40: Retry Buffer and Related Pointers ____ *55*

Figure 41: Average total cost of a data breach in USD millions (Source: IBM Security, 2022) _ *61*

Figure 42: Board of Directors ____ *62*

Figure 43: Contributors ____ *64*

Figure 44: ZYNQ -7000 Arm FPGA ____ *69*

1. About The Sponsors Si-Vision & Synopsys

Si-Vision is a leading IP provider of high-performance, low-power radio frequency intellectual property (RF IP). Their solutions include Bluetooth low energy radio, low power Zigbee radio and other low power wireless radios. Their products deliver what they believe is an industry-leading combination of performance and monolithic integration, and target a broad range of applications serving consumers electronics, HIDs, smart meters, industrial mobile wireless devices, and generally wireless markets. Si-Vision has been in business since 2007, with more than 350 years of collective experience and serving Tier-1 customers all over the globe. Si-Vision team has broad experience in RF Circuits and Systems, Digital and Back-end Designs.

Synopsys, Inc. (Nasdaq: SNPS) on July 16, 2015 announced that it has acquired the Bluetooth Smart IP from Silicon Vision, a leading provider of high-performance, ultra-low-power wireless IP solutions. The acquisition expands Synopsys' extensive portfolio of DesignWare® IP for the Internet of Things (IoT), which includes security IP recently obtained through the acquisition of Elliptic Technologies, as well as logic libraries, memory compilers, non-volatile memory, data converters, interface IP, power-efficient ARC® processors, a sensor and control IP subsystem, and an embedded vision processor. The addition of Bluetooth Smart IP enables Synopsys to address the growing requirements for wireless connectivity in low-power system-on-chips (SoCs) for a range of IoT applications including wearables, beacons, portable health, smart home, industrial and wireless sensor networks.

2. Terminology / Acronyms

PCIe	PCI Express
CXL	Compute Express Link, a low-latency, high-bandwidth link that supports dynamic protocol muxing of coherency, memory access, and IO protocols, thus enabling attachment of coherent accelerators or memory devices.
Accelerator	Devices that may be used by software running on Host processors to offload or perform any type of compute or I/O task. Examples of accelerators include programmable agents (such as GPU/GPGPU), fixed-function agents, or reconfigurable agents such as FPGAs.
Reserved	The contents, states, or information are not defined at this time. Reserved register fields must be read only and must return 0 (all 0's for multi-bit fields) when read. Reserved encodings for register and packet fields must not be used. Any implementation dependent on a Reserved field value or encoding will result in an implementation that is not CXL-spec compliant. The functionality of such an implementation cannot be guaranteed in this or any future revision of this specification. Flit, Slot, and message reserved bits should be set to 0 by the sender and the receiver should ignore them.
SF	Snoop Filter
CXL.cache	Agent coherency protocol that supports device caching of Host memory.
CXL.io	PCIe-based non coherent I/O protocol with enhancements for accelerator support.

CXL.mem	Memory access protocol that supports device-attached memory.
DCOH	This refers to the Device Coherency agent on the device that is responsible for resolving coherency with respect to device caches and managing Bias states.
Flex Bus	A flexible high-speed port that is configured to support either PCI Express or Compute Express Link.
Flex Bus.CXL	CXL protocol over a Flex Bus interconnect.
Home Agent	This is the agent on the Host that is responsible for resolving system wide coherency for a given address.

3. Abstract

The massive growth in the production and consumption of data, particularly unstructured data, like images, digitized speech, and video, is resulting in a huge increase in the use of accelerators. According to the Bank of America Merrill Lynch Global Semiconductors Report from October 2, 2016, "an estimated accelerator TAM of $1.64B in 2017 is expected to grow beyond $10B in 2021." This trend towards heterogeneous computing in the data center means that, increasingly, different types of processors and co-processors must work together efficiently, while sharing memory. This disaggregation can cause systems to experience significant bottlenecks due to the use of large amounts of memory on accelerators, and the need to share this memory coherently with the Hosts to avoid unnecessary and excessive data copying. Compute Express Link (CXL), a low-latency, high-bandwidth link that supports dynamic protocol muxing of coherency, memory access, and IO protocols, thus enabling attachment of coherent accelerators or memory devices. Compute Express Link (CXL), a new open interconnect standard, targets intensive workloads for CPUs and purpose-built accelerators where efficient, coherent memory access between a Host and Device is required. PCI Express (PCIe) has been around for many years, and the recently completed version of the PCIe base specification 5.0 now enables interconnection of CPUs and peripherals at speeds up to 32GT/s. However, in an environment with large shared memory pools and many devices requiring high bandwidth, PCIe has some limitations. PCIe doesn't specify mechanisms to support coherency and can't efficiently manage isolated pools of memory as each PCIe hierarchy shares a single 64-bit address space. In addition, the latency for PCIe links can be too high to efficiently manage shared memory across multiple devices in a system. The CXL standard addresses some of these limitations by providing an interface that leverages the PCIe 5.0 physical layer and electricals, while providing extremely low latency paths for memory access and coherent caching between host processors and devices that need to share memory resources, like accelerators and memory expanders.

4. Motivation and Overview

CXL is a dynamic multi-protocol technology designed to support accelerators and memory devices. CXL provides a rich set of protocols that include I/O semantics similar to PCIe (i.e., CXL.io), caching protocol semantics (i.e., CXL.cache), and memory access semantics (i.e., CXL.mem) over a discrete or on-package link. CXL.io is required for discovery and enumeration, error report, and host physical address (HPA) lookup. CXL.mem and CXL.cache protocols may be optionally implemented by the particular accelerator or memory device usage model. A key benefit of CXL is that it provides a low-latency, high-bandwidth path for an accelerator to access the system and for the system to access the memory attached to the CXL device. Figure 1 below is a conceptual diagram showing a device attached to a Host processor via CXL.

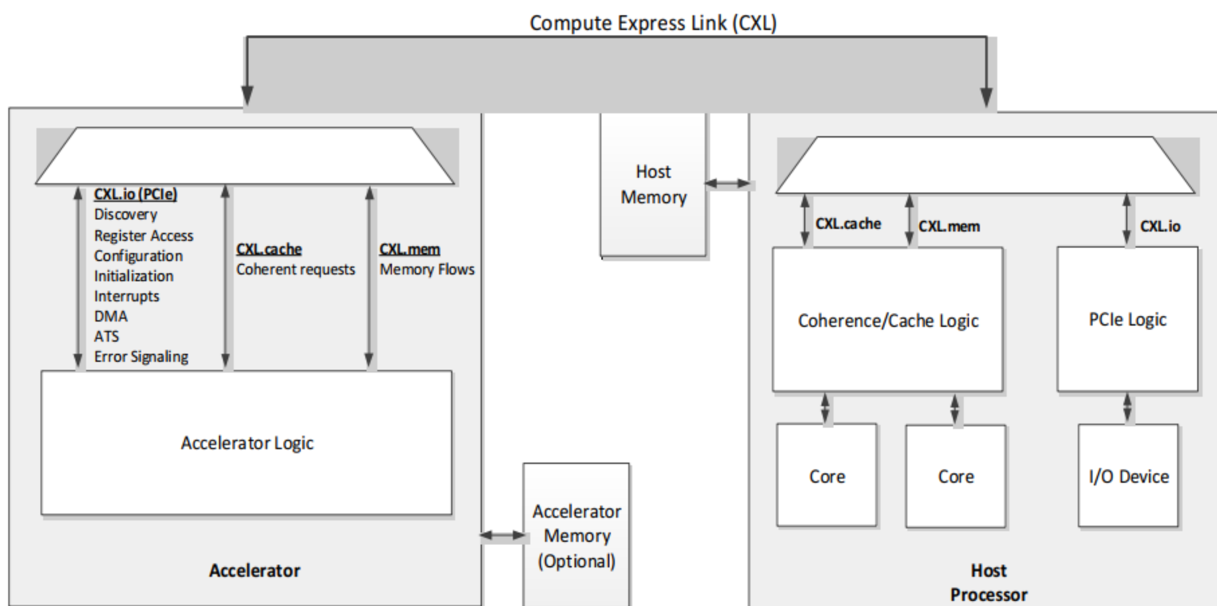


Figure 1: Conceptual Diagram of Accelerator Attached to Processor via CXL

The CXL 2.0 specification enables additional usage models beyond CXL 1.1, while being fully backwards compatible with CXL 1.1 (and CXL 1.0). It enables managed hot-plug, security enhancements, persistent memory support, memory error reporting, and

telemetry. CXL 2.0 also enables single-level switching support for fan-out as well as the ability to pool devices across multiple virtual hierarchies, including multi-domain support of memory devices.

5. What is Our GP Revolving Around?

The intended outcome of our GP is to design & implement the CXL 2.0 controller that supports CXL.Cache & CXL.Mem protocols (will be intensively discussed later on in this paper). The Controller can be configured interface with the Application Layer through one of the following interface options: 1. ARM AMBA CXS Interface. 2. CXL Native Interface.
The high-level block diagrams for the CXL Controller in the figure below show the main components of the controller for both CXS and Native Interface options.

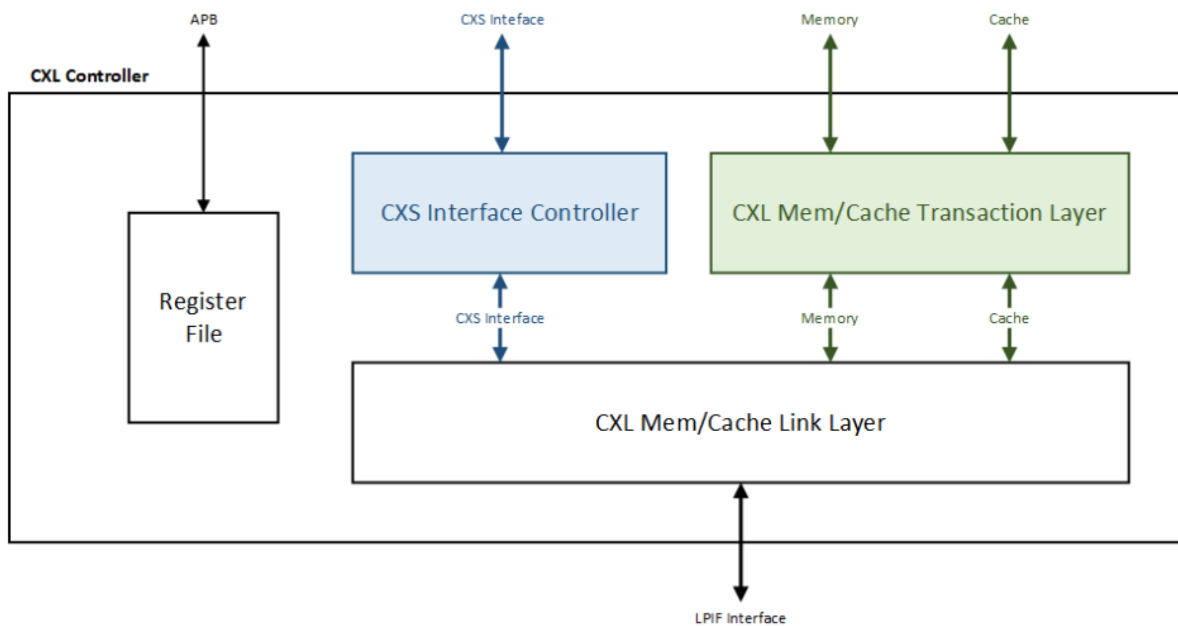


Figure 2: High-Level Block Diagrams for the CXL Controller

5.1 Main Features

1. Compliant with CXL 2.0 specifications.
2. Backward compatibility with CXL 1.1.
3. Configurable operation modes: Device, Host and Dual Mode.
4. Supporting internal data path width: 512-bit.
5. Configurable Transaction/Application Layer clock frequency.
6. APB Interface to access the controller configuration registers.
7. Data path integrity.

5.2 Limitations

1. Integrity and Data Encryption (IDE) is not supported.
2. Switch configuration is not supported.
3. Not supporting full PCIe configuration space.
4. We are not implementing the CXL.io protocol (Just CXL.Mem & CXL.Cache).
5. We are not implementing the physical layer of the CXL (Just data link layer & transaction layer).

5.3 Clock Requirements

The controller should support the following clock domains:

1. Controller primary clock (62.5 MHz).
2. Application clock (supported values as ratio to primary clock: 1:1, 2:1).
3. APB clock (32 MHz).

6. CXL System Architecture

This section describes the performance advantages and key features of CXL. CXL is a high performance I/O bus architecture used to interconnect peripheral devices that can be

either traditional non-coherent IO devices, memory devices, or accelerators with additional capabilities. The types of devices that can attach via CXL and the overall system architecture is described in the below figure:

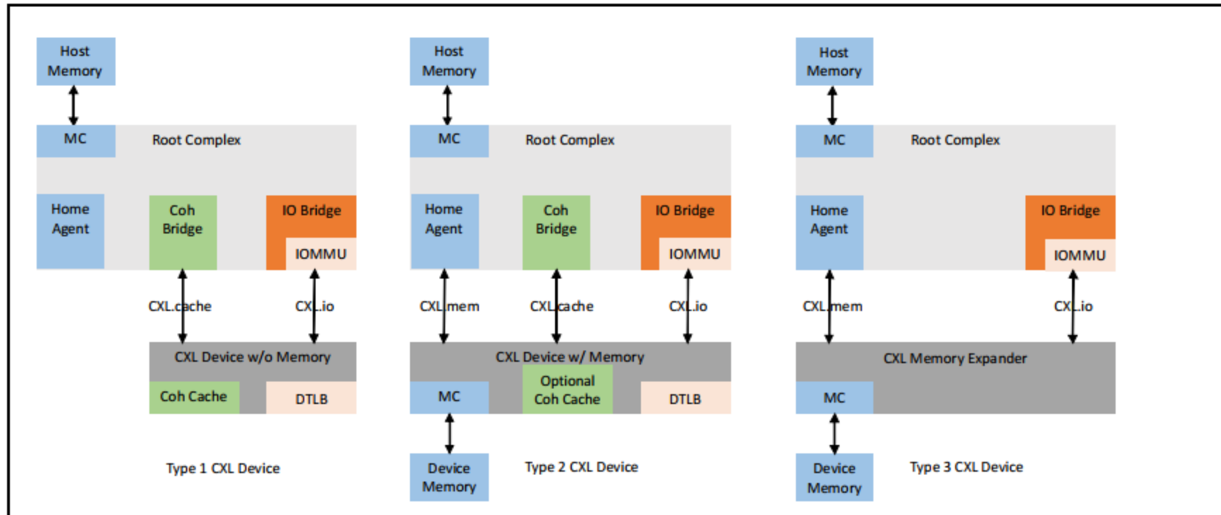


Figure 3: CXL Device Types

6.1 CXL Device Types

6.3.1 Type 1 CXL Device

Type 1 CXL devices have special needs for which having a fully coherent cache in the device becomes valuable. For such devices, standard Producer-Consumer ordering models do not work very well. One example of a device with special requirements is to perform complex atomics that are not part of the standard suite of atomic operations present on PCIe. Basic cache coherency allows an accelerator to implement any ordering model it chooses and allows it to implement an unlimited number of atomic operations. These tend to require only small amounts of cache which can easily be tracked by standard processor snoop filter mechanisms. The size of cache that can be supported for such devices depends on the host's snoop filtering

capacity. CXL supports such devices using its optional CXL.cache link over which an accelerator can use CXL.cache protocol for cache coherency transactions.

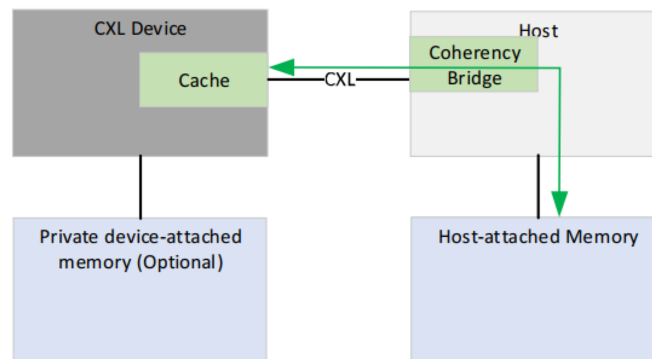


Figure 4: Type 1 CXL Device

6.3.2 Type 2 CXL Device

Type 2 devices, in addition to fully coherent cache, also have memory, for example DDR, High Bandwidth Memory (HBM) etc., attached to the device. These devices execute against memory, but their performance comes from having massive bandwidth between the accelerator and device-attached memory. The key goal for CXL is to provide a means for the Host to push operands into device-attached memory and for the Host to pull results out of device-attached memory such that it doesn't add software and hardware cost that offsets the benefit of the accelerator. This spec refers to coherent system address mapped device-attached memory as Host-managed Device Memory (HDM).

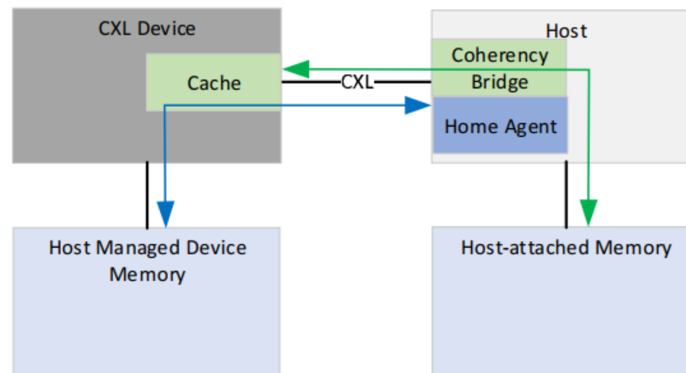


Figure 5: Type 2 CXL Device

6.3.2.1 Bias Based Coherency Model

The Host-managed Device Memory (HDM) attached to a given device is referred to as device-attached memory to denote that it is local to only that device. The Bias Based coherency model defines two states of bias for device-attached memory: Host Bias and Device Bias. When the device-attached memory is in Host Bias state, it appears to the device just as regular Host-attached memory does. That is, if the device needs to access it, it needs to send a request to the Host which will resolve coherency for the

requested line. On the other hand, when the device-attached memory is in Device Bias state, the device is guaranteed that the Host does not have the line in any cache. As such, the device can access it without sending any transaction (request, snoops, etc.) to the Host whatsoever. It is important to note that the Host itself sees a uniform view of device-attached memory regardless of the bias state. In both modes, coherency is preserved for device-attached memory.

The key benefits of Bias Based coherency model are:

1. Helps maintain coherency for device-attached memory which is mapped to system coherent address space.
2. Helps the device access its local attached memory at high bandwidth without incurring significant coherency overheads (e.g., snoops to the Host).
3. Helps the Host access device-attached memory in a coherent, uniform manner, just as it would for Host-attached memory.

6.3.2.1.1 Host Bias

The Host Bias mode typically refers to the part of the cycle when the operands are being written to memory by the Host during work submission or when results are being read out from the memory after work completion. During Host Bias mode, coherency flows allow for high throughput access from the Host to device-attached memory (as shown by the blue arrows in Figure 6) whereas device access to device-attached memory is not optimal since they need to go through the host (as shown in green arrows in Figure 6).

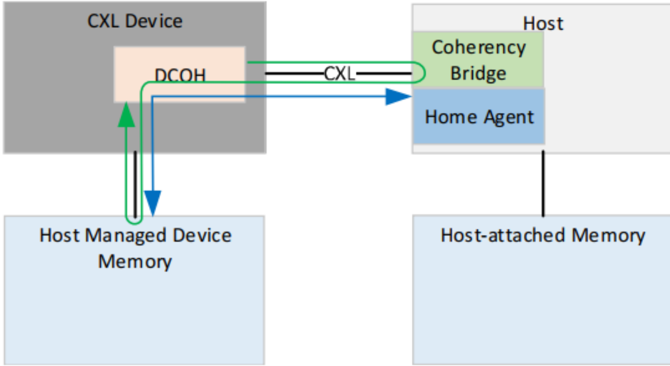


Figure 6: Type 2 Host Bias

6.3.2.1.2 Device Bias

The Device Bias mode is used when the device is executing the work, between work submission and completion, and in this mode, the device needs high bandwidth and low latency access to device-attached memory. In this mode, device can access device-attached memory without consulting the Host's coherency engines (as shown in red arrows in Figure 7). The Host can still access device-attached memory but may be forced to give up ownership by the accelerator (as shown in green arrows in Figure 7). This results in the device seeing ideal latency & bandwidth from device-attached memory, whereas the Host sees compromised performance.

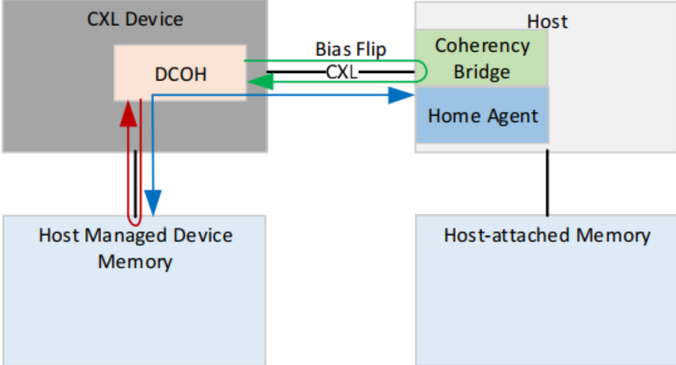


Figure 7: Type 2 Device Bias

6.3.3 Type 3 CXL Device

A Type 3 CXL Device supports CXL.io and CXL.mem protocols. An example of a Type 3 CXL device is a memory expander for the Host as shown in the figure below.

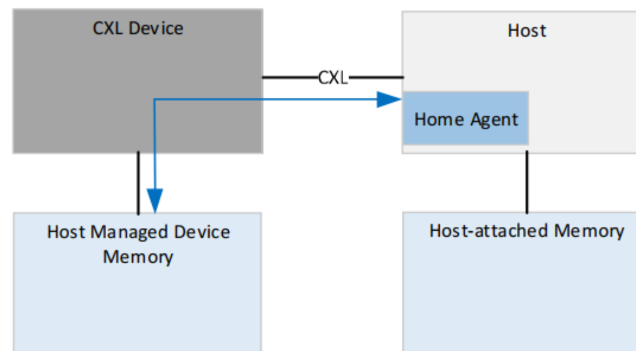


Figure 8: Type 3 CXL Device

Since this is not an accelerator, the device does not make any requests over CXL.cache. The device operates primarily over CXL.mem to service requests sent from the Host.

7. Transaction Layer

7.1 CXL.Cache

The CXL.cache protocol defines the interactions between the Device and Host as a number of requests that each have at least one associated response message and sometimes a data transfer. The interface consists of three channels in each direction: Request, Response, and Data. The channels are named for their direction, D2H for Device to Host and H2D for Host to Device, and the transactions they carry, Request, Response, and Data as shown in Figure 9. The independent channels allow different kinds of messages to use dedicated wires and achieve both decoupling and a higher effective throughput per wire.

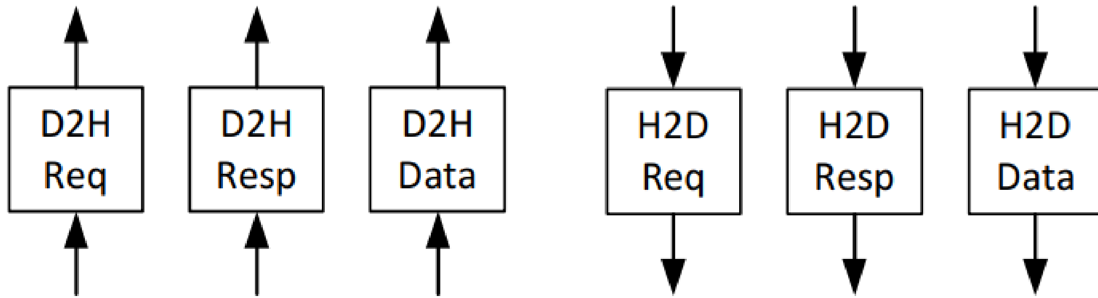


Figure 9: CXL.Cache Channels

D2H Request (Device to Host Request) carries new requests from the Device to the Host. The requests typically target memory. Each request will receive zero, one or two responses and at most one 64-byte cache line of data. The channel may be back pressured without issue. D2H Response carries all responses from the Device to the Host. Device responses to snoops indicate the state the line was left in the device caches, and may indicate that data is being returned to the Host to the provided data buffer. They may still be blocked temporarily for link layer credits, but should not require any other transaction to complete to free the credits. D2H Data carries all data and byte-enables from the Device to the Host. The data transfers can result either from implicit (as a result of snoop) or explicit write-backs (as a result of cache capacity eviction). In all cases a full 64-byte cache line of data will be transferred. D2H Data transfers must make progress or deadlocks may occur. They may be blocked temporarily for link layer credits, but must not require any other transaction to complete to free the credits.

H2D Request (Host to Device Request) carries requests from the Host to the Device. These are snoops to maintain coherency. Data may be returned for snoops. The request carries the location of the data buffer to which any return data should be written. H2D Requests may be back pressured for lack of device resources; however, the resources must free up without needing D2H Requests to make progress. H2D Response carries ordering messages and pulls for write data. Each response carries the request identifier from the original device request to

indicate where the response should be routed. For write data pull responses, the message carries the location where the data should be written. H2D Responses can only be blocked temporarily for link layer credits. H2D Data delivers the data for device read requests. In all cases a full 64-byte cache line of data will be transferred. H2D Data transfers can only be blocked temporarily for link layer credits.

7.2.1 Channel Ordering

In general, all of the CXL.cache channels must work independently of each other to ensure that forward progress is maintained. For example, since requests from the device to the Host to a given address X will be blocked by the Host until it collects all snoop responses for this address X, linking the channels would lead to deadlock. However, there is a specific instance where ordering between channels must be maintained for the sake of correctness. The Host needs to wait until Global Ordering (GO) messages, sent on H2D Response, are observed by the device before sending subsequent snoops for the same address. To limit the amount of buffering needed to track GO messages, the Host assumes that GO messages that have been sent over CXL.cache in a given cycle cannot be passed by snoops sent in a later cycle.

7.2.2 Channel Crediting

To maintain the modularity of the interface no assumptions can be made on the ability to send a message on a channel since link layer credits may not be available at all times. Therefore, each channel must use a credit for sending any message and collect credit returns from the receiver. During operation, the receiver returns a credit whenever it has processed the message (i.e., freed up a buffer). It is not required that all credits are accounted for on either side, it is sufficient that credit counter saturates when full. If no credits are available, the sender must wait for the receiver to return one. The table below describes which channels must drain to maintain forward progress and which can be blocked indefinitely.

7.2.3 CXL.Cache Transaction Description

7.2.3.1 Device to Host Requests

For device to Host requests there are four different semantics: CXL.cache Read, CXL.cache Read0, CXL.cache Read0/Write, and CXL.cache Write. All device to Host CXL.cache transactions fall into the one of these four semantics, though the allowable responses and restrictions for each request type within a given semantic are different.

CXL.Cache Read

CXL.cache Reads must have a D2H request credit and send a request message on the D2H CXL.cache request channel. CXL.cache Read requests require zero or one response (GO) message and data messages totaling a single 64-byte cache line of data. Both the response, if present, and data messages are directed at the device tracker entry provided in the initial D2H request packet's CQID field. The device entry must remain active until all the messages from the Host have been received. To ensure forward progress the device must have a reserved data buffer to be able to accept all 64 bytes of data immediately after the request is sent. However, the device may temporarily be unable to accept data from the Host due to prior data returns not draining. Once both the response message and the data messages have been received from the Host, the transaction can be considered complete and the entry deallocated from the device.

The figure below shows the elements required to complete a CXL.cache Read. Note that the response (GO) message can be received before, after, or between the data messages.

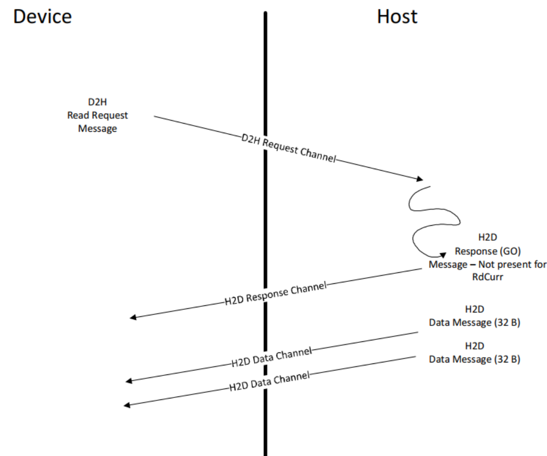


Figure 10: CXL.Cache Read Behavior

CXL.Cache Write

CXL.cache Write must have a D2H request credit before sending a request message on the D2H CXL.cache request channel. Once the Host has received the request message, it is required to send either two separate or one merged GO-I and WritePull message. The GO message must never arrive at the device before the WritePull but it can arrive at the same time in the combined message. If the transaction requires posted semantics, then a combined GO-I/WritePull message can be used. If the transaction requires non-posted semantics, then WritePull will be issued first followed by the GO-I when the non-posted write is globally observed. Upon receiving the GO-I message, the device will consider the store done from a memory ordering and cache coherency perspective, giving up snoop ownership of the cache line (if the CXL.cache message is an Evict). The WritePull message triggers the device to send data messages to the Host totaling exactly 64 bytes of data, though any number of bytes enables can be set. A CXL.cache write transaction is considered complete by the device once the device has received the GO-I message, and has sent the required data messages. At this point the entry can be deallocated from the device. The Host considers a

write to be done once it has received all 64 bytes of data, and has sent the GO-I response message. All device writes and Evicts fall into the CXL.cache Write semantic.

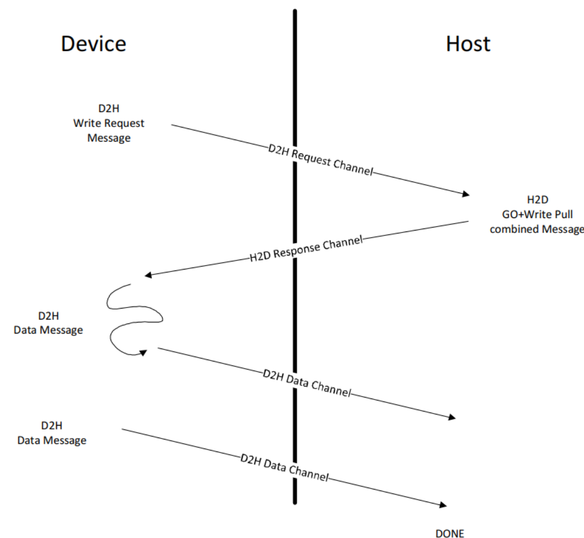


Figure 11: CXL.Cache Device to Host Write Behavior

7.2 CXL.Mem

The CXL Memory Protocol is called CXL.mem, and it is a transactional interface between the CPU and Memory. It uses the physical and link layer of Compute Express Link (CXL) when communicating across dies. The protocol can be used for multiple different Memory attach options including when the Memory Controller is located in the Host CPU, when the Memory Controller is within an Accelerator device, or when the Memory Controller is moved to a memory buffer chip. It applies to different Memory types (volatile, persistent, etc.) and configurations (flat, hierarchical, etc.) as well.

The coherency engine in the CPU interfaces with the Memory (Mem) using CXL.mem requests and responses. In this configuration, the CPU coherency engine is regarded as the CXL.mem Master and the Mem device is regarded as the CXL.mem Subordinate. The CXL.mem Master is the agent which is responsible for sourcing CXL.mem requests (reads, writes, etc.) and a CXL.mem Subordinate is the agent which is responsible for responding to CXL.mem requests (data, completions, etc.).

When the Subordinate is an Accelerator, CXL.mem protocol assumes the presence of a device coherency engine (DCOH). This agent is assumed to be responsible for implementing coherency related functions such as snooping of device caches based on CXL.mem commands and update of Meta Data fields. Support for memory with Meta Data is optional but this needs to be negotiated with the Host in advance. The negotiation mechanisms are outside the scope of this specification. If Meta Data is not supported by device-attached memory, the DCOH will still need to use the Host supplied Meta Data updates to interpret the commands. If Meta Data is supported by device-attached memory, it can be used by Host to implement a coarse snoop filter for CPU sockets.

CXL.mem transactions from Master to Subordinate are called “M2S” and transactions from Subordinate to Master are called “S2M”.

Within M2S transactions, there are two message classes:

I. Request without data - generically called Requests (Req)
II. Request with Data - (RwD)

Similarly, within S2M transactions, there are two message classes:

I. Response without data - generically called No Data Response (NDR)
II. Response with data - generically called Data Response (DRS)

7.3 Transaction Ordering Summary

Table in Figure 12 captures the upstream ordering cases and Table in Figure 13 captures the downstream ordering cases. The columns represent a first issued message and the rows represent a subsequently issued message. The table entry indicates the ordering relationship between the two messages. The table entries are defined as follows:

➔ Yes–the second message (row) must be allowed to pass the first (column) to avoid Dead lock. (When blocking occurs, the second message is required to pass the first message. Fairness must be comprehended to prevent starvation.)

➔ Y/N–there are no ordering requirements. The second message may optionally pass

the first message or be blocked by it.

→ No–the second message must not be allowed to pass the first message. This is required to support the protocol ordering model.

Row Pass Column?	.io TLPs (Col 2-5)	S2M NDR/DRS D2H Rsp/Data (Col 6)	D2H Req (Col 7)
.io TLPs (Row A-D)	PCIe Base	Yes(1)	Yes(1)
S2M NDR/DRS D2H Rsp/Data (Row E)	Yes(1)	Y/N	Yes(2)
D2H Req (Row F)	Yes(1)	Y/N	Y/N

Row Pass Column?	.io TLPs (Col 2-5)	M2S Req (Col 8)	M2S RwD (Col 9)	H2D Req (Col 10)	H2D Resp (Col 11)	H2D Data (Col 12)
.io TLPs (Row A-D)	PCIe Base	Yes(1)	Yes(1)	Yes(1)	Yes(1)	Yes(1)
M2S Req (Row G)	Yes(1)	a. No(5) b. Y/N	Y/N	Yes(2)	Y/N	Y/N
M2S RwD (Row H)	Yes(1)	Y/N	Y/N	Yes(2)	Y/N	Y/N
H2D Req (Row I)	Yes(1)	Yes(3)	Yes(3)	Y/N	a. No(4) b. Y/N	Yes(3)
H2D Resp (Row J)	Yes(1)	Yes(2)	Yes(2)	Yes(2)	Y/N	Y/N
H2D Data (Row K)	Yes(1)	Yes(2)	Yes(2)	Yes(2)	Y/N	Y/N

Figure 12: Upstream Ordering Summary

7.4 Transaction Flows to Device-Attached Memory

The transaction flow diagrams below are intended to be illustrative of the flows between the Host and device for access to device-attached Memory using the Bias Based Coherency mechanism described earlier. However, these flows are not comprehensive of every Host and device interaction. The diagrams below make the following assumptions:

1. The device contains a coherency engine which is called DCOH in the diagrams below.
2. The DCOH contains a Snoop Filter which tracks any caches (called Dev cache) implemented on the device. This is not strictly required, and the device is free to choose an implementation specific mechanism as long as the coherency rules are obeyed.
3. The DCOH contains a Bias Table lookup mechanism. The implementation of this is device specific.
4. The device specific aspects of the flow, illustrated using red flow arrows, need not conform exactly to the pictures below. These can be implemented in a device specific manner.

7.4.1 Cachable Read from Host

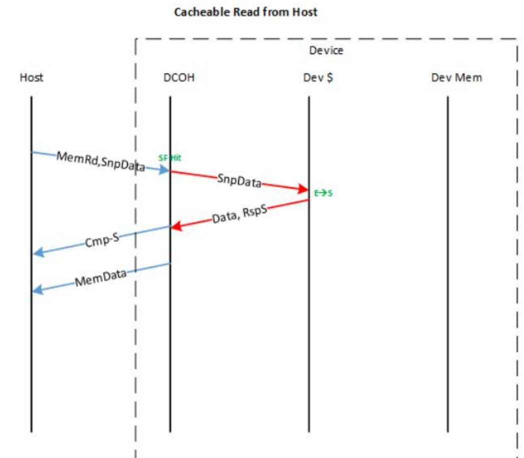


Figure 13: Cachable Read from Host

In this example, the Host requested a cacheable non-exclusive copy of the line. The non-exclusive aspect of the request is communicated using the "SnpData" semantic. In this example, the request got a snoop filter hit in the DCOH, which caused the device cache to be snooped. The device cache downgraded the state from Exclusive to Shared and returned the Shared data copy to the Host. The Host is told of the state of the line using the Cmp-S semantic.

7.4.2 Read for Ownership from Host

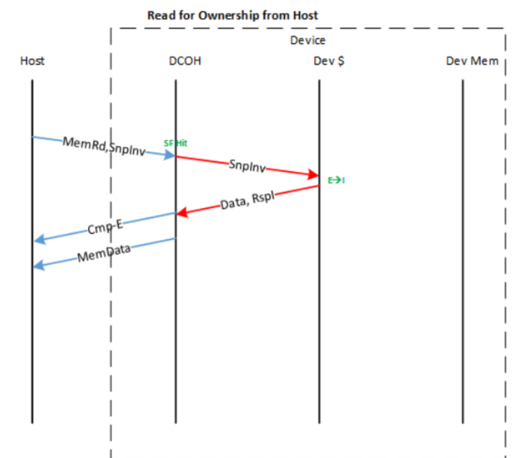


Figure 14: Read for Ownership from Host

In the above example, the Host requested a cacheable exclusive copy of the line. The exclusive aspect of the request is communicated using the "SnpInv" semantic, which asks the device to invalidate its caches. In this example, the request got a snoop filter hit in the DCOH, which caused the device cache to be snooped. The device cache downgraded the state from Exclusive to Invalid and returned the Exclusive data copy to the Host. The Host is told of the state of the line using the Cmp-E semantic.

7.4.3 Non-Cachable Read from Host

In the above example, the Host requested a non-cacheable copy of the line. The noncacheable aspect of the request is communicated using the “SnpCurr” semantic.

In this example, the request got a snoop filter hit in the DCOH, which caused the device cache to be snooped. **The device cache did not need to change its caching state**; however, it gave the current snapshot of the data. The Host is told that it is not allowed to cache the line using the Cmp semantic.

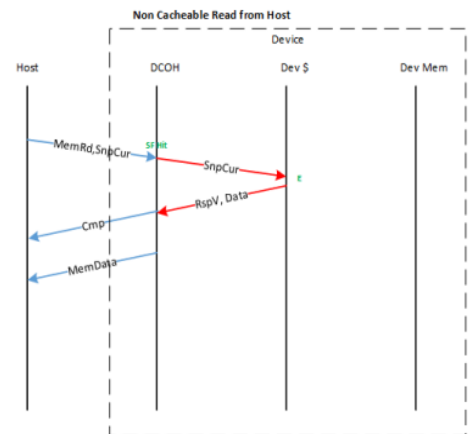


Figure 15: Non-Cachable Read from Host

7.4.4 Ownership Request from Host - No Data Required

In this example, the Host requested exclusive access to a line without requiring the device to send data. It communicates that to the device using an opcode of MemInv with a MetaValue of ‘10 (Any), which is significant in this case. It also asks the device to invalidate its caches with the SnpInv command. The device invalidates its caches and gives exclusive ownership to the Host as communicated using the Cmp-E semantic.

Figure 16: Ownership Request from Host (NDR)

7.4.5 Flush from Host

In the above example, the Host wants to flush a line from all caches, including the device's caches, to device memory. To do so, it uses an opcode of MemInv with a MetaValue of '00 (Invalid) and a SnpInv.

The device flushes its caches and returns a Cmp indication to the Host.

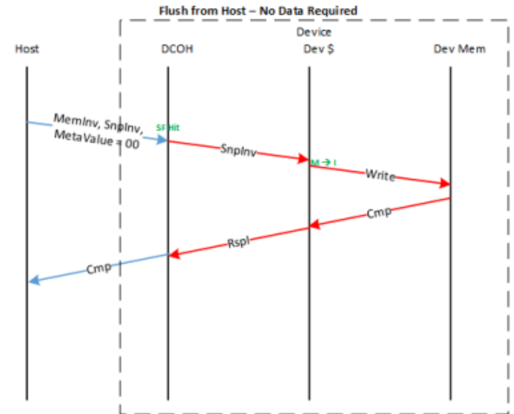


Figure 17: Flush from Host

7.4.6 Weakly Ordered Write from Host

In this example, the Host issues a weakly ordered write (partial or full line).

The weakly ordered semantic is communicated by the embedded SnpInv. In this example, the device had a copy of the line cached. This resulted in a merge within the device before writing it back to memory and sending a Cmp indication to the Host.

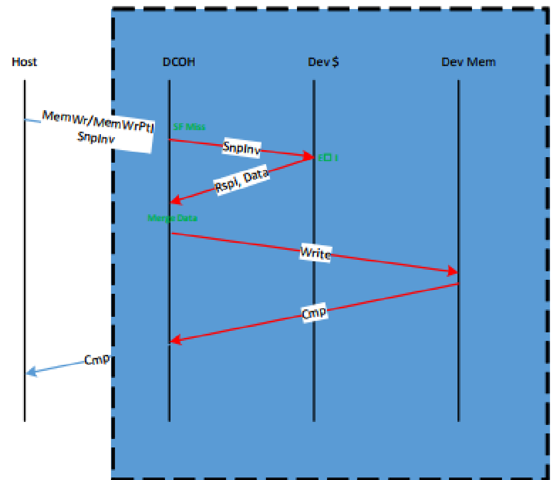


Figure 18: Weakly Ordered Write from Host

7.4.7 Write from Host with Invalid Host Caches

In the above example, the Host performed a write while guaranteeing to the device that it no longer has a valid cached copy of the line. The fact that the Host didn't need to snoop the device's caches means it previously acquired an exclusive copy of the line. The guarantee on no valid cached copy is indicated by a MetaValue of '00 (Invalid).

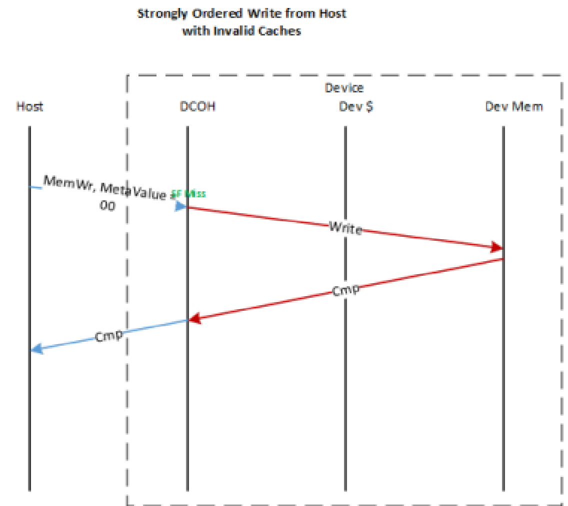


Figure 19: Write from Host with Invalid Host Caches

7.4.7 Write from Host with Valid Host Caches

This example is the same as the previous one except that the Host chose to retain a valid cacheable copy of the line after the write.

This is communicated to the device using a MetaValue of not '00 (Invalid).

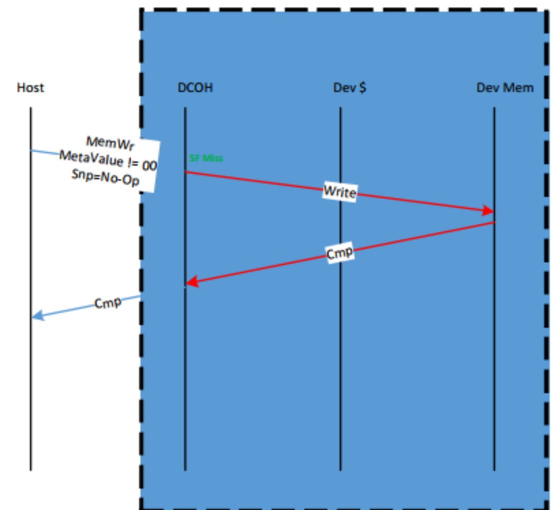


Figure 20: Write from Host with Valid Host Caches

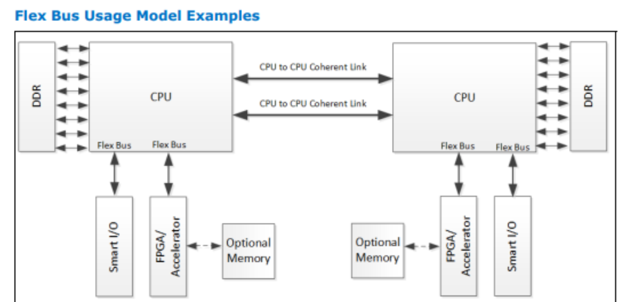
Note: You need to go back to the CXL 2.0 official standard for more examples and their illustration.

8. Data Link Layer

Before we go into details in the data link layer, we'll describe the flex bus.

A **Flex Bus** port allows designs to choose between providing native PCIe protocol or CXL over a high-bandwidth, off-package link; the selection happens during link training via alternate protocol negotiation and depends on the device that is plugged into the slot. Flex Bus uses PCIe electricals, making it compatible with PCIe retimers, and form factors that support PCIe. Figure 21 provides a high-level diagram of a Flex Bus port implementation, illustrating both a slot implementation and a custom implementation where the device is soldered down on the motherboard. The slot implementation can accommodate either a Flex Bus. CXL card or a PCIe card. One or two optional retimers can be inserted between the CPU and the device to extend the channel length. As illustrated in Figure 22, this flexible port can be used to attach coherent accelerators or smart I/O to a Host processor.

Flex Bus Usage Model Examples



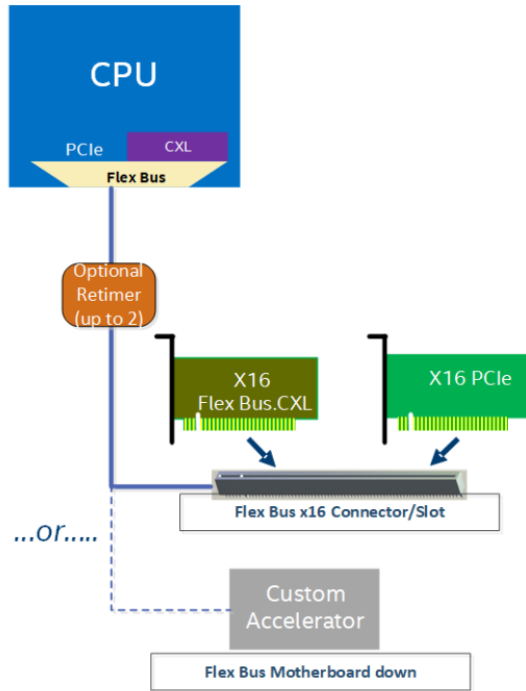
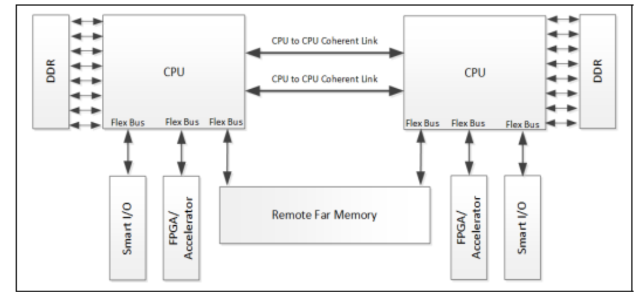
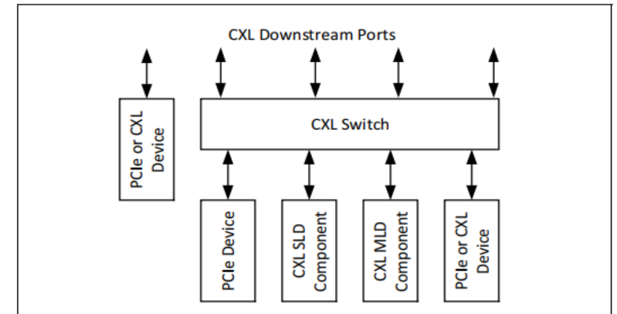


Figure 21: CPU Flex Bus Port





Flex Bus Features

Flex Bus provides a point-to-point interconnect that can transmit native PCIe protocol or dynamic multi-protocol CXL to provide I/O, caching, and memory protocols over PCIe electricals. The primary link attributes include support of the following features:

I. Native PCIe mode, full feature support as defined in the PCIe specification

II. CXL mode, as defined in this specification

III. Configuration of PCIe vs CXL protocol mode

IV. Signaling rate of 32 GT/s, degraded rate of 16GT/s or 8 GT/s in CXL mode

V. Link width support for x16, x8, x4, x2 (degraded mode), and x1 (degraded mode) in CXL mode

VI. Bifurcation (aka Link Subdivision) support to x4 in CXL mode

Flex Bus Layering Overview

Flex Bus architecture is organized as multiple layers, as illustrated in Figure 22. The CXL transaction (protocol) layer is subdivided into logic that handles CXL.io and logic that handles CXL.mem and CXL.cache; the CXL link layer is subdivided in the same manner. Note that the CXL.mem and CXL.cache logic are combined within the transaction layer and within the link layer. The CXL link layer interfaces with the CXL ARB/MUX, which interleaves the traffic from the two logic streams. Additionally, the PCIe transaction and data link layers are optionally implemented and, if implemented, are permitted to be converged with the CXL.io transaction and link layers, respectively. As a result of the link training process, the transaction and link layers are configured to operate in either PCIe mode or CXL mode. While a host CPU would most likely implement both modes, an accelerator AIC is permitted to implement only the CXL mode. The logical sub-block of the Flex Bus physical layer is a converged logical physical layer that can operate in either PCIe mode or CXL mode, depending on the results of alternate mode negotiation during the link training process.

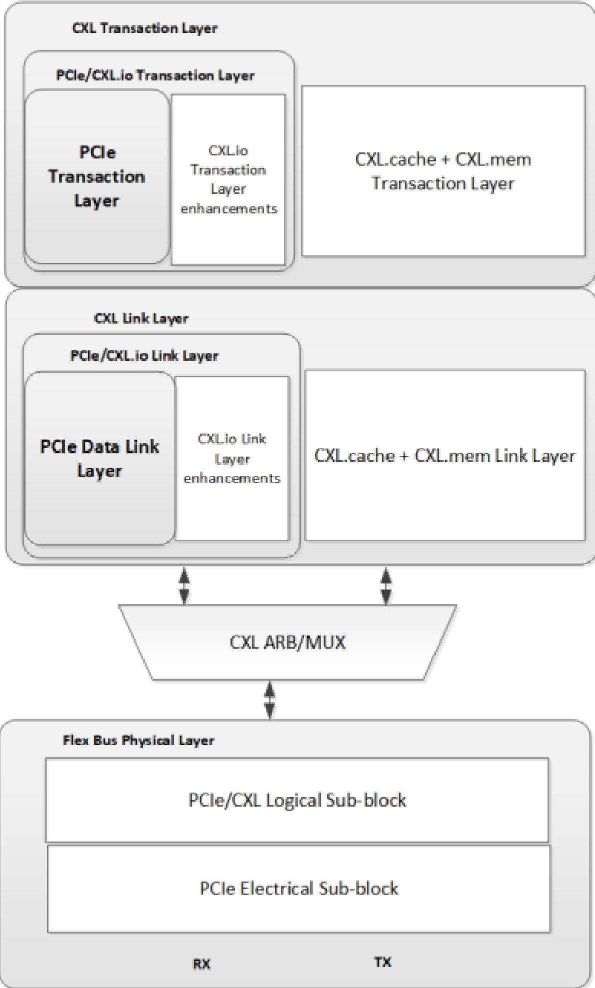


Figure 22: Conceptual Diagram of Flex Bus Layering

The figure below shows where the CXL.cache and CXL.mem link layer exists in the Flex Bus layered hierarchy.

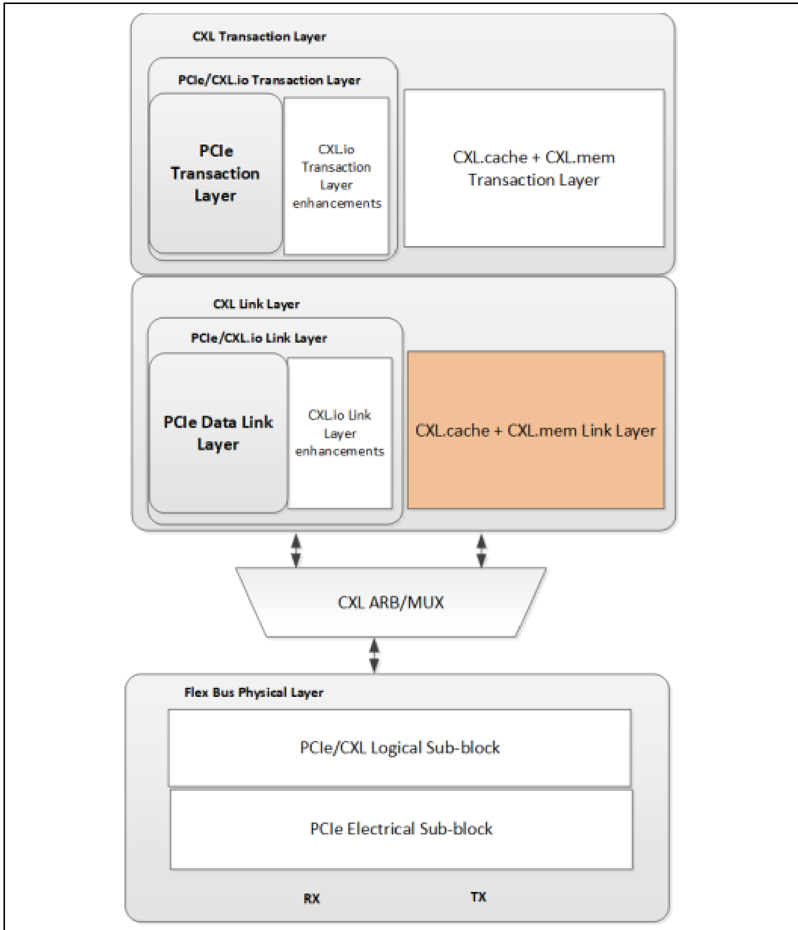


Figure 23: Flex Bus Layers - CXL.Cache + CXL.Mem Link Layer Highlighted

CXL.cache and CXL.mem protocols use a common Link Layer. This section defines the properties of this common Link Layer.

The following Figures illustrate the high-level CXL.Cache/ CXL.Mem Flit Overview.

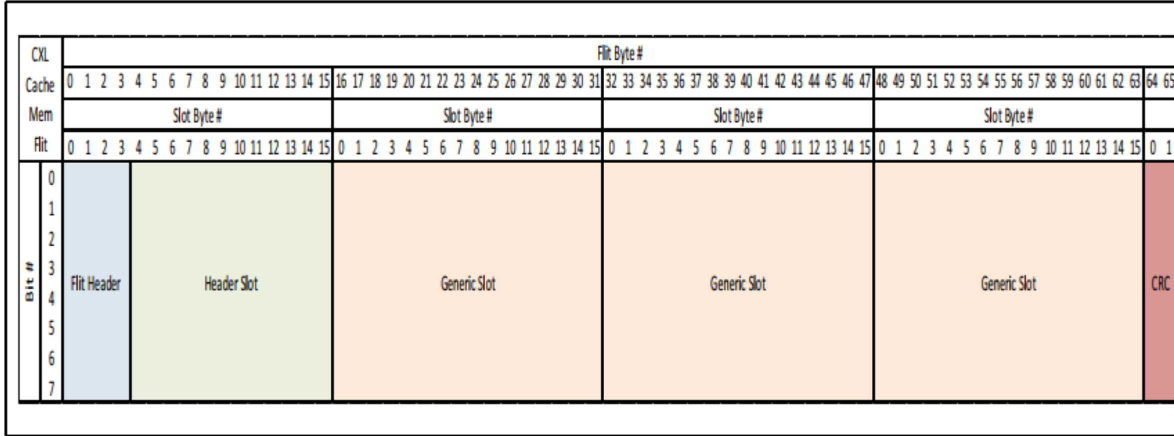


Figure 24:CXL.cache/. mem Protocol Flit Overview

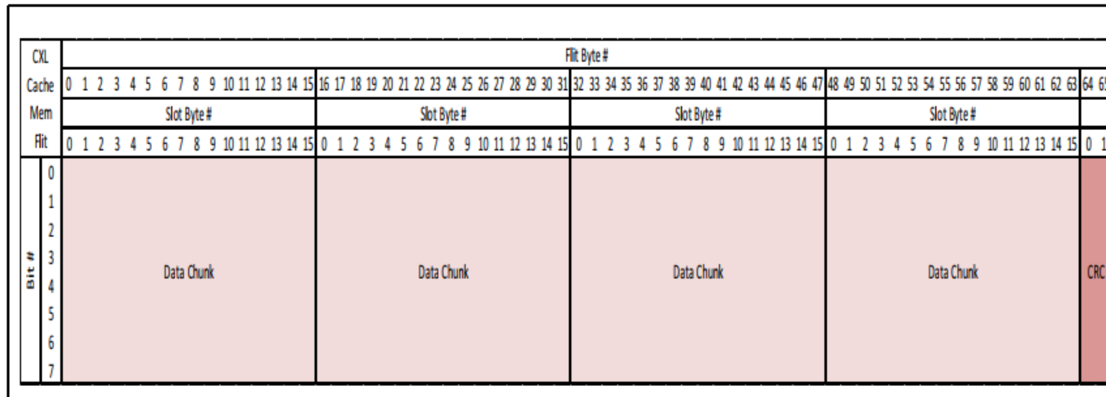


Figure 25: CXL.cache/.mem All Data Flit Overview

Example of a Protocol Flit from Device to Host

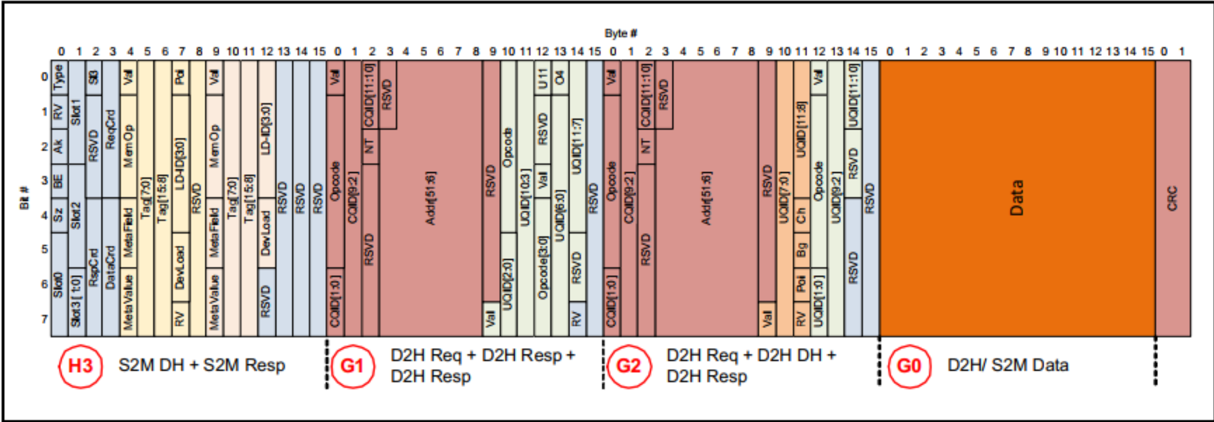


Figure 26: Example of a protocol flit from device to host

A "Header" Slot is defined as one that carries a "Header" of link-layer specific information, including the definition of the protocol-level messages contained in the rest of the header as well as in the other slots in the flit.

A "Generic" Slot can carry one or more request/response messages or a single 16B data chunk. The flit can be composed of a Header Slot and 3 Generic Slots or four 16B Data Chunks. The flit header utilizes the same definition for both the Upstream as well as the Downstream ports summarized in the table below.

CXL.cache/CXL.mem Flit Header Definition

Field Name	Brief Description	Size
Type	This field distinguishes between a Protocol or a Control Flit	1
Ak	This is an acknowledgment of 8 successful flit transfers Reserved for Retry, and Init control flits	1
BE	Byte Enable (Reserved for control flits)	1
Sz	Size (Reserved for control flits)	1
ReqCrd	Request Credit Return Reserved for Retry, and Init control flits	4
DataCrd	Data Credit Return Reserved for Retry, and Init control flits	4
RspCrd	Response Credit Return Reserved for Retry, and Init control flits	4
Slot 0	Slot 0 Format Type (Reserved for control flits)	3
Slot 1	Slot 1 Format Type (Reserved for control flits)	3
Slot 2	Slot 2 Format Type (Reserved for control flits)	3
Slot 3	Slot 3 Format Type (Reserved for control flits)	3
RSVD	Reserved	4
Total		32

Figure 27: CXL.cache/CXL.mem Flit Header Definition

In general, bits or encodings that are not defined will be marked “Reserved” or “RSVD” in this specification. These bits should be set to 0 by the sender of the packet and the receiver should ignore them. Please also note that certain fields with static 0/1 values will be checked by the receiving Link Layer when decoding a packet. For example, Control flits have several static bits defined. A Control flit that passes the CRC check but fails the static bit check should be treated as a standard CRC error or as a fatal error when in “retry_local_normal” state of the LRSM. Logging and reporting of such errors is device specific. Checking of these bits reduces the probability of silent error under conditions where the CRC check fails to detect a long burst error. However, link layer must not cause fatal error whenever it is under shadow of CRC errors, i.e., its LRSM is not in “retry_local_normal” state. This is prescribed because all-data-flit can alias to control messages after a CRC error and those alias cases may result in static bit check failure.

The following figure describes how the flit header information is encoded:

Type Encoding

	Flit Type	Description
0	Protocol	This is a flit that carries CXL.cache or CXL.mem protocol related information
1	Control	This is a flit inserted by the link layer purely for link layer specific functionality. These flits are not exposed to the upper layers.

Figure 28: Flit Header Type Encoding

The Ak field is used as part of the link layer retry protocol to signal CRC-passing receipt of flits from the remote transmitter. The transmitter sets the Ak bit to acknowledge successful receipt of 8 flits; a clear Ak bit is ignored by the receiver.

The BE (Byte Enable) and Sz (Size) fields have to do with the variable size of data messages. To reach its efficiency targets, the CXL.cache/mem link layer assumes that generally all bytes are enabled for most data, and that data is transmitted at the full cache line granularity. When all bytes are enabled, the link layer does not transmit the byte enable bits, but instead

clears the Byte Enable field of the corresponding flit header. When the receiver decodes that the Byte Enable field is clear, it must regenerate the byte enable bits as all ones before passing the data message on to the transaction layer. If the Byte Enable bit is set, the link layer Rx expects an additional data chunk slot containing byte enable information. Note that this will always be the last slot of data for the associated request.

Similarly, the **Sz** field reflects the fact that the CXL.cache/mem protocol allows transmission of data at the half cache line granularity. When the Size bit is set, the link layer Rx expects four slots of data chunks, corresponding to a full cache line. When the Size bit is clear, it expects only two slots of data chunks. In the latter case, each half cache line transmission will be accompanied by its own data header. A critical assumption of packing the Size and Byte Enable information in the flit header is that the Tx flit packer may begin at most one data message per flit.

The following table describes legal values of Sz and BE for various data transfers. For cases where a 32B split transfer is sent that includes Byte Enables, the trailing Byte Enables apply only to the 32B sent. The Byte Enable bits that are applicable to that transfer are aligned based on which half of the cacheline is applicable to the transfer (BE[63:32] for Upper half or BE [31:0] for the lower half of the cacheline). This means that each of the split 32B transfers to form a cacheline of data will include Byte Enables if Byte Enables are needed. Illegal use will cause an uncorrectable error.

Legal values of Sz and BE Fields

Type of Data Transfer	32B Transfer Possible?	BE Possible?
CXL.cache H2D Data	Yes	No
CXL.mem M2S Data	No	Yes
CXL.cache D2H Data	Yes	Yes
CXL.mem S2M Data	Yes	No

Figure 29: Legal values of Sz and BE Fields

The transmitter sets the Credit Return fields to indicate resources available in the collocated receiver for use by the remote transmitter. Credits are given for transmission per message

class, which is why the flit header contains independent Request, Response, and Data Credit Return fields. Note that there are no Requests sourced in S2M direction, and there are no Responses sourced in M2S direction. The details of the channel mapping are captured in Figure 30. Credits returned for channels not supported by the device or host should be silently discarded. The granularity of credits is per message. These fields are encoded exponentially, as delineated in Figure 31 below.

Note: Messages sent on Data channels require a single data credit for the entire messages. This means 1 credit allows for one data transfer, including the header of the message, regardless of whether the transfer is 64B, 32B or contains Byte Enables.

ReqCrd/DataCrd/RspCrd Channel Mapping

Credit Field	Credit Bit 3 Encoding	Link Direction	Channel
ReqCrd	0 - CXL.Cache	Upstream	D2H Request
		Downstream	H2D Request
	1 - CXL.Mem	Upstream	Reserved
		Downstream	M2S Request
DataCrd	0 - CXL.Cache	Upstream	D2H Data
		Downstream	H2D Data
	1 - CXL.Mem	Upstream	S2M Response with Data (DRS)
		Downstream	M2S Request with Data (RwD)
RspCrd	0 - CXL.Cache	Upstream	D2H Response
		Downstream	H2D Response
	1 - CXL.Mem	Upstream	S2M No Data Response (NDR)
		Downstream	Reserved

Figure 30: ReqCrd/DataCrd/RspCrd Channel Mapping

CXL.cache/CXL.mem Credit Return Encodings

Credit Return Encoding[3]	Protocol
0	CXL.cache
1	CXL.mem
Credit Return Encoding[2:0]	**Number of Credits**
000	0
001	1
010	2
011	4
100	8
101	16
110	32
111	64

Figure 31: CXL.cache/CXL.mem Credit Return Encodings

Finally, the Slot Format Type fields encode the Slot Format of both the header slot and of the other generic slots in the flit (if the Flit Type bit specifies that the flit is a Protocol Flit). The subsequent sections detail the protocol message contents of each slot format, but the table below provides a quick reference for the Slot Format field encoding.

Slot Format Field Encoding

Slot Format Encoding	H2D/M2S		D2H/S2M	
	Slot 0	Slots 1,2 and 3	Slot 0	Slots 1, 2 and 3
000	H0	G0	H0	G0
001	H1	G1	H1	G1
010	H2	G2	H2	G2
011	H3	G3	H3	G3
100	H4	G4	H4	G4
101	H5	G5	H5	G5
110	H6	RSVD	H6	G6
111	RSVD	RSVD	RSVD	RSVD

Figure 32: Slot Format Field Encoding

The following tables describe the slot format and the type of message contained by each format for both directions.

H2D/M2S Slot Formats

Format to Req Type Mapping	H2D/M2S	
	Type	Size
H0	CXL.cache Req + CXL.cache Resp	96
H1	CXL.cache Data Header + 2 CXL.cache Resp	88
H2	CXL.cache Req + CXL.cache Data Header	88
H3	4 CXL.cache Data Header	96
H4	CXL.mem RwD Header	87
H5	CXL.mem Req Only	87
H6	MAC slot used for link integrity.	96
G0	CXL.cache/ CXL.mem Data Chunk	128
G1	4 CXL.cache Resp	128
G2	CXL.cache Req + CXL.cache Data Header + CXL.cache Resp	120
G3	4 CXL.cache Data Header + CXL.cache Resp	128
G4	CXL.mem Req + CXL.cache Data Header	111
G5	CXL.mem RwD Header + CXL.cache Resp	119

Figure 33: H2D/M2S Slot Formats

D2H/S2M Slot Formats

Format to Req Type Mapping	D2H/S2M	
	Type	Size
H0	CXL.cache Data Header + 2 CXL.cache Resp + CXL.mem NDR	87
H1	CXL.cache Req + CXL.cache Data Header	96
H2	4 CXL.cache Data Header + CXL.cache Resp	88
H3	CXL.mem DRS Header + CXL.mem NDR	70
H4	2 CXL.mem NDR	60
H5	2 CXL.mem DRS Header	80
H6	MAC slot used for link integrity.	96
G0	CXL.cache/ CXL.mem Data Chunk	128
G1	CXL.cache Req + 2 CXL.cache Resp	119
G2	CXL.cache Req + CXL.cache Data Header + CXL.cache Resp	116
G3	4 CXL.cache Data Header	68
G4	CXL.mem DRS Header + 2 CXL.mem NDR	100
G5	2 CXL.mem NDR	60
G6	3 CXL.mem DRS Header	120

Figure 34: D2H/S2M Slot Formats

Slot Format Definition

Slot diagrams in the section include abbreviations for bit field names to allow them to fit into the diagram. In the context of diagram most abbreviations are obvious, but the abbreviation list below ensures clarity.

- SL3 = Slot3[2]
- LI3 = LD-ID[3]
- U11 = UQID[11]
- O4 = Opcode[4]

- Val = Valid
- RV = Reserved
- RSVD = Reserved
- Poi = Poison
- Tag15 =Tag[15]
- MV0 = MetaValue[0]
- MV1 = MetaValue[1]
- R11 = RspData[11]

H2D and M2S Formats

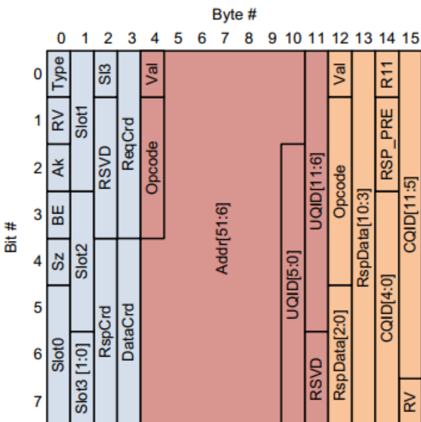


Figure 35: H0 - H2D Req + H2D Resp H2D Resp

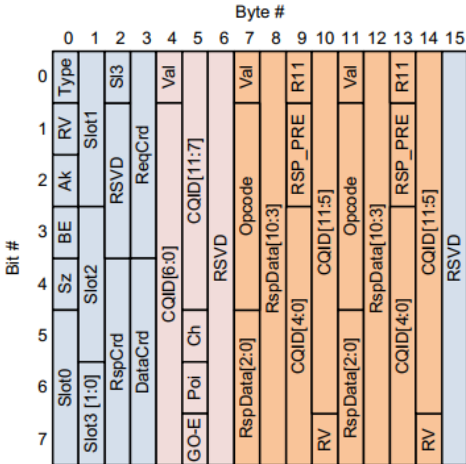


Figure 36: H1 - H2D Data Header + H2D Resp +

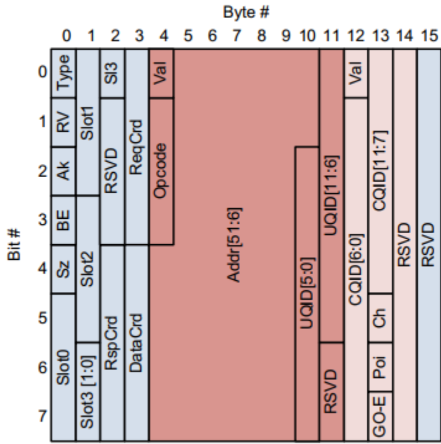


Figure 37: H3 - 4 H2D Data Header

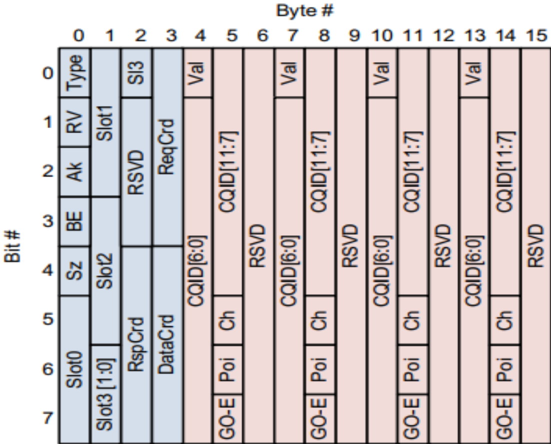


Figure 38: H2 - H2D Req + H2D Data Header

Note: Please, refer to section 2 from chapter 4 in the CXL 2.0 for more H2D and M2S formats.

Flit Packing Rules

The packing rules are defined below. It is assumed that a given queue has credits towards the RX and any protocol dependencies (SNP-GO ordering, for example) have already been considered:

• Rollover is defined as any time a data transfer needs more than one flit. Note that a data chunk which contains 128b (format G0), can only be scheduled in Slots 1, 2, and 3 of a protocol flit since Slot 0 has only 96b available, as 32b are taken up by the flit header. The following rules apply to Rollover data chunks.

— If there's a rollover of more than 3 16B data chunks, the next flit must necessarily be an all data flit.

— If there's a rollover of 3 16B data chunks, Slots 1, Slots 2 and Slots 3 must necessarily contain the 3 rollover data chunks. Slot 0 will be packed independently (it is allowed for Slot 0 to have the Data Header for the next data transfer).

— If there's a rollover of 2 16B data chunks, Slots 1 and Slots 2 must necessarily contain the 2 rollover data chunks. Slot 0 and Slot 3 will be packed independently.

— If there's a rollover of 1 16B data chunk, Slot 1 must necessarily contain the rollover data chunk. Slot 0, Slot 2 and Slot 3 will be packed independently.

— If there's no rollover, each of the 4 slots will be packed independently.

• Care must be taken to ensure fairness between packing of CXL.mem & CXL.cache transactions. Similarly, care must be taken to ensure fairness between channels within a given protocol. The exact mechanism to ensure fairness is implementation specific.

Valid messages within a given slot need to be tightly packed. Which means, if a slot contains multiple possible locations for a given message, the Tx must pack the message in the first available location before advancing to the next available location.

• Valid messages within a given flit need to be tightly packed. Which means, if a flit contains multiple possible slots for a given message, the Tx must pack the message in the first available slot before advancing to the next available slot.

• Empty slots are defined as slots without any valid bits set and they may be mixed with other slots in any order as long as other packing rules are followed. For an example refer to Figure 26 where slot H3 could have no valid bits set indicating an empty slot, but the 1st and 2nd generic slots, G1 and G2 in the example, may have mixed valid bits set.

• If a valid Data Header is packed in a given slot, the next available slot for data transfer (Slot 1, Slot 2, Slot 3 or an all-data flit) will be guaranteed to have data associated with the header. The Rx will use this property to maintain a shadow copy

of the Tx Rollover counts. This enables the Rx to expect all-data flits where a flit header is not present.

• For data transfers, the Tx must send 16B data chunks in cacheline order. That is, chunk order 01 for 32B transfers and chunk order 0123 for 64B transfers.

A MDH slot format must be chosen by the Tx only if there is more than 1 valid Data Header to pack in that slot.

• Control flits cannot be interleaved with all-data flits. This also implies that when an all-data flit is expected following a protocol flit (due to Rollover), the Tx cannot send a Control flit before the all-data flit.

• For non-MDH containing flits, there can be at most 1 valid Data Header in that flit. Also, a MDH containing flit cannot be packed with another valid Data Header in the same flit.

• The maximum number of messages that can be sent in a given flit is restricted to reduce complexity in the receiver which writes these messages into credited queues. By restricting the number of messages across the entire flit, the number of write ports into the receiver's queues are constrained. The maximum messages in a flit (sum, across all slots) is:

D2H Request --> 4

D2H Response --> 2

D2H Data Header --> 4

D2H Data --> 4*16B

S2M NDR --> 2

S2M DRS Header --> 3

S2M DRS Data --> 4*16B

H2D Request --> 2

H2D Response --> 4

H2D Data Header --> 4

H2D Data --> 4*16B

M2S Req --> 2

M2S RwD Header --> 1

M2S RwD Data --> 4*16B

For a given slot, lower bit positions are defined as bit positions that appear starting from lower order Byte #. That is, bits are ordered starting from (Byte 0, Bit 0) through (Byte 15, Bit 7).

• For multi-bit message fields like Address [MSB: LSB], less significant bits will appear in lower order bit positions.

• Message ordering within a flit is based on flit bit numbering, i.e., the earliest messages are placed at the lowest flit bit positions and progressively later messages are placed at progressively higher bit positions. Examples: An M2S Req 0 packed in Slot 0 precedes an M2S Req 1 packed in Slot 1. Similarly, a Snoop packed in Slot 1 follows a GO packed in Slot 0, and this ordering must be maintained. Finally, for Header Slot Format H1, an H2D Response packed starting from Byte 7 precedes an H2D Response packed starting from Byte 11.

Link Layer Control Flit

Link Layer Control flits do not follow flow control rules applicable to protocol flits. That is, they can be sent from an entity without any credits. These flits must be processed and consumed by the receiver within the period to transmit a flit on the channel since there are no storage or flow control mechanisms for these flits. The following table lists all the Controls Flits supported by the CXL.cache/CXL.mem link layer.

In CXL 2.0 a 3-bit CTL_FMT field is added to control messages and uses bits that were reserved in CXL1.1 control messages. All control messages used in CXL1.1 have this field encoded as 'b000 to maintain backward compatibility. This field is used to distinguish formats added in CXL 2.0 control messages that require a larger payload field. The new

format increases the payload field from 64-bits to 96-bits and uses CTL_FMT encoding of ‘b001.

Table 53. CXL.cache/CXL.mem Link Layer Control Types

LLCTRL Encoding	LLCTRL Type Name	Description	Retryable? (Enters the LLRB)
'b0001	RETRY	Link layer retry flit	No
'b0000	LLCRD	Flit containing only link layer QoS Telemetry, credit return and/or Ack information, but no protocol information.	Yes
'b0010	IDE	Integrity and Data Encryption control messages. Use in flows described in Section 11.1 which are introduced in CXL 2.0	Yes
'b1100	INIT	Link layer initialization flit	Yes
Others	Reserved	n/a	n/a

Figure 39: CXL.cache/CXL.mem Link Layer Control Types

Note: For link layer control details, refer to section 2 in chapter 4 in the CXL 2.0 standard.

Link Layer Initialization

Link Layer Initialization must be started after a physical layer link down to link up transition and the link has trained successfully to L0. During Initialization and after the Init Flit has been sent the Cache/Mem Link Layer can only send Control-Retry flits until Link Initialization is complete. The following describes how the link layer is initialized and credits are exchanged. The Tx portion of the Link Layer must wait until the Rx portion of the Link Layer has received at least one valid flit that is CRC clean before sending the ControlINIT.Param flit. Before this condition is met, the Link Layer must transmit only

Control-Retry flits, i.e., Retry. Frame/Req/Ack/Idle flits.

— If for any reason the Rx portion of the Link Layer is not ready to begin processing flits beyond Control-INIT and Control-Retry, the Tx will stall transmission of LLCTR-INIT.Param flit

— Retry. Frame/Req/Ack are sent during this time as part of the regular Retry flow.

— Retry.Idle flits are sent prior to sending a Init.Param flit even without a retry condition to ensure the remote agent can observe a valid flit.

• The Control-INIT.Param flit must be the first non-Control-Retry flit transmitted by

the Link Layer

• The Rx portion of the Link Layer must be able to receive an Control-INIT.Param flit immediately upon completion of Physical Layer initialization because the very first valid flit may be a Control-INIT.Param

• Received Control-INIT.Param values (i.e., LLR Wrap Value) must be made “active”, that is, applied to their respective hardware states within 8 flit clocks of error-free reception of Control-INIT.Param flit.

— Until an error-free INIT.Param flit is received and these values are applied, LLR Wrap Value shall assume a default value of 9 for the purposes of ESEQ tracking.

• Any non-Retry flits received before Control-INIT.Param flit will trigger an Uncorrectable Error.

• Only a single Control-INIT.Param flit is sent. Any CRC error conditions with an Control-INIT.Param flit will be dealt with by the Retry state machine and replayed from the Link Layer Retry Buffer.

• Receipt of an Control-INIT.Param flit after an Control-INIT.Param flit has already been received should be considered an Uncorrectable Error.

• It is the responsibility of the Rx to transmit credits to the sender using standard credit return mechanisms after link initialization. Each entity should know how many buffers it has and set its credit return counters to these values. Then, during normal operation, the standard credit return logic will return these credits to the sender.

• Immediately after link initialization, the credit exchange mechanism will use the LLCRD flit format.

• It is possible that the receiver will make available more credits than the sender can track for a given message class. For correct operation, it is therefore required that the credit counters at the sender be saturating. Receiver will drop all credits in receives for unsupported channels (example: Type 3 device receiving any

CXL.Cache credits).

• Credits should be sized to achieve desired levels of bandwidth considering roundtrip time of credit return latency. This is implementation and usage dependent.

Link Layer Retry

The link layer provides recovery from transmission errors using retransmission, or Link Layer Retry (LLR). The sender buffers every retryable flit sent in a local link layer retry buffer (LLRB). To uniquely identify flits in this buffer, the retry scheme relies on sequence numbers which are maintained within each device. Unlike in PCIe, CXL.cache/ .mem sequence numbers are not communicated between devices with each flit to optimize link efficiency. The exchange of sequence numbers occurs only through link layer control flits during a LLR sequence. The sequence numbers are set to a predetermined value (zero) during Link Layer Initialization and they are implemented using a wrap-around counter. The counter wraps back to zero after reaching the depth of the retry buffer. This scheme makes the following assumptions:

• The round-trip delay between devices is more than the maximum of the link layer
clock or flit period.

• All protocol flits are stored in the retry buffer.

Note that for efficient operation, the size of the retry buffer must be more than the round-trip delay. This includes:

• Time to send a flit from the sender

• Flight time of the flit from sender to receiver

• Processing time at the receiver to detect an error in the flit

• Time to accumulate and, if needed, force Ack return and send embedded Ack
return back to the sender

• Flight time of the Ack return from the receiver to the sender

• Processing time of Ack return at the original sender

Otherwise, the LLR scheme will introduce latency, as the transmitter will have to wait for the receiver to confirm correct receipt of a previous flit before the transmitter can free space in its LLRB and send a new flit. Note that the error case is not significant because transmission of new flits is effectively stalled until successful retransmission of the erroneous flit anyway.

LLR Variables

The retry scheme maintains two state machines and several state variables. Although the following text describes them in terms of one transmitter and one receiver, both the transmitter and receiver side of the retry state machines and the corresponding state variables are present at each device because of the bidirectional nature of the link. Since both sides of the link implement both transmitter and receiver state machines, for clarity this discussion will use the term “local” to refer to the entity that detects a CRC error, and “remote” to refer to the entity that sent the flit that was received erroneously.

The receiving device uses the following state variables to keep track of the sequence number of the next flit to arrive.

· **ESeq**: This indicates the expected sequence number of the next valid flit at the receiving link layer entity. ESeq is incremented by one (modulo the size of the LLRB) on error-free reception of a retryable flit. ESeq stops incrementing after an error is detected on a received flit until retransmission begins (RETRY.Ack message is received). Link Layer Initialization sets ESeq to 0. Note that there is no way for the receiver to know that an error was for a non-retryable vs retryable flit. For any CRC error it will initiate the link layer retry flow as usual, and effectively the transmitter will resend from the first retryable flit sent. The sending entity maintains two indices into its LLRB, as indicated below.

· **WrPtr**: This indexes the entry of the LLRB that will record the next new flit. When an entity sends a flit, it copies that flit into the LLRB entry indicated by the WrPtr

and then increments the WrPtr by one (modulo the size of the LLRB). This is implemented using a wrap-around counter that wraps around to 0 after reaching the depth of the LLRB. Non-Retryable Control flits do not affect the WrPtr. WrPtr stops incrementing after receiving an error indication at the remote entity (RETRY.Req message) except as described in the implementation note below, until normal operation resumes again (all flits from the LLRB have been retransmitted). WrPtr is initialized to 0 and is incremented only when a flit is put into the LLRB.

RdPtr: This is used to read the contents out of the LLRB during a retry scenario.

The value of this pointer is set by the sequence number sent with the retransmission request (RETRY.Req message). The RdPtr is incremented by one (modulo the size of the LLRB) whenever a flit is sent, either from the LLRB in response to a retry request or when a new flit arrives from the transaction layer and irrespective of the states of the local or remote retry state machines. If a flit is being sent when the RdPtr and WrPtr are the same, then it indicates that a new flit is being sent, otherwise it must be a flit from the retry buffer.

The LLR scheme uses an explicit acknowledgment that is sent from the receiver to the sender to remove flits from the LLRB at the sender. The acknowledgment is indicated via an ACK bit in the headers of flits flowing in the reverse direction. In CXL.cache, a single ACK bit represents 8 acknowledgments. Each entity keeps track of the number of available LLRB entries and the number of received flits pending acknowledgment through the following variables.

• NumFreeBuf: This indicates the number of free LLRB entries at the entity.

NumFreeBuf is decremented by 1 whenever an LLRB entry is used to store a transmitted flit. NumFreeBuf is incremented by the value encoded in the Ack/ Full_Ack (Ack is the protocol flit bit AK, Full_Ack defined as part of LLCRD message) field of a received flit. NumFreeBuf is initialized at reset time to the size of the LLRB. The maximum number of retry queues at any entity is limited to 255 (8 bit counter). Also, note that the retry buffer at any entity is never filled to its capacity,

therefore NumFreeBuf is never ‘0. If there is only 1 retry buffer entry available, then the sender cannot send a Retryable flit. This restriction is required to avoid ambiguity between a full or an empty retry buffer during a retry sequence that may result into incorrect operation. This implies if there are only 2 retry buffer entries left (NumFreeBuf = 2), then the sender can send an Ack bearing flit only if the outgoing flit encodes a value of at least 1 (which may be a Protocol flit with Ak bit set), else a LLCRD control flit is sent with Full_Ack value of at least 1. This is required to avoid deadlock at the link layer due to retry buffer becoming full at both entities on a link and their inability to send ACK through header flits. This rule also creates an implicit expectation that you cannot start a sequence of “All Data Flits” that cannot be completed before NumFreeBuf=2 because you must be able to inject the Ack bearing flit when NumFreeBuf=2 is reached.

• NumAck: This indicates the number of acknowledgments accumulated at the receiver. NumAck increments by 1 when a retryable flit is received. NumAck is decremented by 8 when the ACK bit is set in the header of an outgoing flit. If the outgoing flit is coming from the LLRB and its ACK bit is set, NumAck does not decrement. At initialization, NumAck is set to 0. The minimum size of the NumAck field is the size of the LLRB. NumAck at each entity must be able to keep track of at least 255 acknowledgments. The LLR protocol requires that the number of retry queue entries at each entity must be at least 22 entries (Size of Forced Ack (16) + Max All-Data-Flit (4) + 2) to prevent deadlock.

LLCRD Forcing

Recall that the LLR protocol requires space available in the LLRB to transmit a new flit, and that the sender must receive explicit acknowledgment from the receiver before freeing space in the LLRB. In scenarios where the traffic flow is very asymmetric, this requirement

could result in traffic throttling and possibly even starvation. Suppose that the A→B direction has very heavy traffic, but there is no traffic at all in the B→A direction. In this case A could exhaust its LLRB size, while B never has any return traffic in which to embed Acks. In CXL we want to minimize injected traffic to reserve bandwidth for the other traffic stream(s) sharing the link. To avoid starvation, CXL must permit LLCRD Control message forcing (injection of a non-traffic flit to carry an Acknowledge and Credit return), but this function must be constrained to avoid wasting bandwidth. In CXL, when B has accumulated a programmable minimum number of Acks to return, B's CXL.cache/mem link layer will inject a LLCRD flit to return an Acknowledge. The threshold of pending Acknowledges before forcing the LLCRD can be adjusted using the "Ack Force Threshold" field in the "CXL Link Layer Ack Timer Control Register". There is also a timer-controlled mechanism to force LLCRD when the timer reaches a threshold. The timer will clear whenever an ACK/CRD carrying message is sent. It will increment every link layer clock an ACK/CRD carrying message is not sent and any Credit value to return is greater than 0 or acknowledge to return is greater than 1. The reason the Acknowledge threshold value is specified as "greater than 1", as opposed to "greater than 0", is to avoid repeated forcing of LLCRD when no other retryable flits are being sent. If the timer incremented when the pending Acknowledge count is "greater than 0", there would be a continuous exchange of LLCRD messages carrying Acknowledges on an otherwise idle link; this is because the LLCRD is itself retryable and results in a returning Acknowledge in the other direction. The result is that the link layer would never be truly idle when the transaction layer traffic is idle. The timer threshold to force LLCRD is configurable using the "Ack or CRD flush retimer" field in the "CXL Link Layer Ack Timer Control Register". It should also be noted that the CXL.cache link layer must accumulate a minimum of 8 Acks to set the ACK bit in a CXL.cache and CXL.mem flit header. If LLCRD forcing occurred after the accumulation of 8 Acks, it could result in a negative beat pattern where real traffic always arrives soon after a forced Ack, but not long enough after for enough Acks to re-accumulate to set the ACK bit. In the worst case this could double the bandwidth consumption of the CXL.cache side. By

waiting for at least 16 Acks to accumulate, the CXL.cache/mem link layer ensures that it can still opportunistically return Acks in a protocol flit avoiding the need to force an LLCRD for Ack return. It is recommended that the Ack Force Threshold value be set to 16 or greater in the “CXL Link Layer Ack Timer Control Register” to reduce overhead of LLCRD injection.

It is recommended that link layer prioritize other link layer flits before LLCRD forcing.

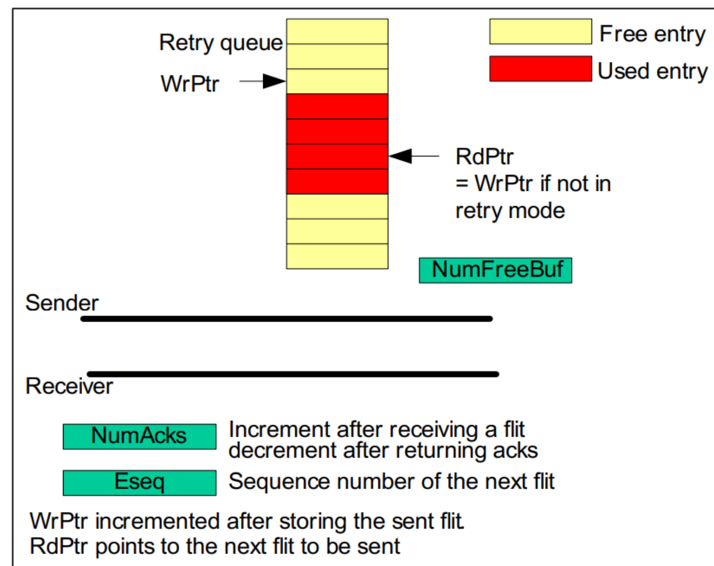


Figure 40: Retry Buffer and Related Pointers

LLR State Machines

The LLR scheme is implemented with two state machines: Remote Retry State Machine (RRSM) and Local Retry State Machine (LRSM). These state machines are implemented by each entity and together determine the overall state of the transmitter and receiver at the entity. The states of the retry state machines are used by the send and receive controllers to determine what flit to send and the actions needed to process a received flit.

Local Retry State Machines

This state machine is activated at the entity that detects an error on a received flit. The possible states for this state machine are:

• RETRY_LOCAL_NORMAL: This is the initial or default state indicating normal operation (no CRC error has been detected).

• RETRY_LLRREQ: This state indicates that the receiver has detected an error on a received flit and a RETRY.Req sequence must be sent to the remote entity.

• RETRY_LOCAL_IDLE: This state indicates that the receiver is waiting for a RETRY.Ack sequence from the remote entity in response to its RETRY.Req sequence. The implementation may require sub-states of RETRY_LOCAL_IDLE to capture, for example, the case where the last flit received is a Frame flit and the next flit expected is a RETRY.Ack.

• RETRY_PHY_REINIT: The state machine remains in this state for the duration of a physical layer retrain.

• RETRY_ABORT: This state indicates that the retry attempt has failed and the link cannot recover. Error logging and reporting in this case is device specific. This is a terminal state. The local retry state machine also has the three counters described below. The counters and thresholds described below are implementation specific.

• TIMEOUT: This counter is enabled whenever a RETRY.Req request is sent from an entity and the LRSM state become RETRY_LOCAL_IDLE. The TIMEOUT counter is disabled and the counting stops when the LRSM state changes to some state other than RETRY_LOCAL_IDLE. The TIMEOUT counter is reset to 0 at link layer initialization and whenever the LRSM state changes from RETRY_LOCAL_IDLE to RETRY_LOCAL_NORMAL or RETRY_LLRREQ. The TIMEOUT counter is also reset when the Physical layer returns from re-initialization (the LRSM transition through RETRY_PHY_REINIT to RETRY_LLRREQ). If the counter has reached its threshold without receiving a Retry.Ack sequence, then the RETRY.Req request is sent again to retry the same flit.

NUM_RETRY: This counter is used to count the number of RETRY.Req requests sent to retry the same flit. The counter remains enabled during the whole retry sequence (state is not RETRY_LOCAL_NORMAL). It is reset to 0 at initialization. It is also reset to 0 when a RETRY.Ack

sequence is received with the Empty bit set or whenever the LRSM state is RETRY_LOCAL_NORMAL and an error-free retryable flit is received. The counter is incremented whenever the LRSM state changes from RETRY_LOCAL_LLRREQ to RETRY_LOCAL_IDLE. If the counter reaches a threshold (Called MAX_NUM_RETRY), then the local retry state machine transitions to the RETRY_PHY_REINIT. The NUM_RETRY counter is also reset when the Physical layer exits from LTSSM recovery state (the LRSM transition through RETRY_PHY_REINIT to RETRY_LLRREQ). Note: It is suggested that the value of MAX_NUM_RETRY should be no less than 0xA.

· **NUM_PHY_REINIT**: This counter is used to count the number of physical layer's reinitializations generated during a LLR sequence. The counter remains enabled during the whole retry sequence (state is not RETRY_LOCAL_NORMAL). It is reset to 0 at initialization and after successful completion of the retry sequence. The counter is incremented whenever the LRSM changes from RETRY_LLRREQ to RETRY_PHY_REINIT. If the counter reaches a threshold (called MAX_NUM_PHY_REINIT) instead of transitioning from RETRY_LLRREQ to RETRY_PHY_REINIT, the LRSM will transition to RETRY_ABORT. The NUM_PHY_REINIT counter is also reset whenever a Retry.Ack sequence is received with the Empty bit set. Note: It is suggested that the value of MAX_NUM_PHY_REINIT should be no less than 0xA.

Note: For local retry state transitions, refer to section 2 from chapter 4 in the CXL 2.0 standard.

Timeout definition

After the local receiver has detected a CRC error, triggering the LRSM, the local Tx sends a RETRY.Req sequence to initiate LLR. At this time, the local Tx also starts its TIMEOUT counter. The purpose of this counter is to decide that either the Retry.Req sequence or corresponding Retry.Ack sequence has been lost, and that another RETRY.Req attempt should be made. Recall that it is a fatal error to receive multiple Retry.Ack sequences (i.e., a subsequent Ack without a corresponding Req is unexpected). To reduce the risk of this fatal error condition

we check NUM_RETRY value returned to filter out Retry.Ack messages from the prior retry sequence. This is done to remove fatal condition where a single retry sequence incurs a timeout while the Ack message is in flight. The TIMEOUT counter should be capable of handling worst-case latency for a Retry.Req sequence to reach the remote side and for the corresponding Retry.Ack sequence to return. Certain unpredictable events (such as low power transitions, etc.) that interrupt link availability could add a very large amount of latency to the RETRY round-trip. To make the TIMEOUT robust to such events, instead of incrementing per link layer clock, TIMEOUT increments whenever the local Tx transmits a flit, protocol or control. Due to the TIMEOUT protocol, it must force injection of RETRY.Idle flits if it has no real traffic to send, so that the TIMEOUT counter continues to increment.

Interaction with Physical Layer Reinitalization

On detection of a physical layer LTSSM Recovery, the receiver side of the link layer must force a link layer retry on the next flit. Forcing an error will either initiate LLR or cause a current LLR to follow the correct error path. The LLR will ensure that no retryable flits are dropped during the physical layer reinit. Without initiating a LLR it is possible that packets/flits in flight on the physical wires could be lost or the sequence numbers could get mismatched.
Upon detection of a physical layer LTSSM Recovery, the LLR RRSM needs to be reset to its initial state and any instance of Retry.Ack sequence needs to be cleared in the link layer and physical layer. The device needs to make sure it receives a Retry.Req sequence before it ever transmits a RETRY.Ack sequence.

CXL.Cache/ CXL.Mem Flit CRC

The CXL.cache Link Layer uses a 16b CRC for transmission error detection. The 16b CRC is over the 528-bit flit. The assumptions about the type errors are as follows:

- Bit ordering runs down each lane

• Bit Errors occur randomly or in bursts down a lane, with majority of errors single bit random errors.

• Random errors can statistically cause multiple bit errors in a single flit, so it is more likely to get 2 errors in a flit then 3 errors, and more likely to get 3 errors in a flit then 4 errors, and so on...

• There is no requirement for primitive polynomial (a polynomial that generates all elements of an extension field from a base field) since we do have a fixed payload. Primitive may be the result, but it's not required.

CRC-16 Polynomial and Detection Properties

The CRC polynomial to be used is 0x1f053.

• The 16b CRC Polynomial has the following properties:

• All Single, double, and triple bit errors detected

• Polynomial selection based on best 4-bit error detection characteristics and perfect 1, 2, 3-bit error detection.

9. Market Analysis

Microchip Technology

The explosion of modern applications such as Artificial Intelligence, Machine Learning and deep learning is changing the very nature of computing and transforming businesses. These applications have opened myriad ways for companies to improve their business development processes, operations, security and provide better customer experiences. To support these applications, platforms are being designed to utilize SoCs that can process large data sets in cloud data centers, have specialized processing power to service the use cases, create customized solutions, and scale this market. The market size of AI was valued at $65.48 billion in 2020, and is projected to reach $1,581.70 billion by 2030, growing at a CAGR of 38.0% from 2021 to 2030, according to a recent report by Allied Market Research.

With this exponential growth comes rising concerns about the security on these platforms running mission-critical applications in emerging markets such as healthcare, automotive, and data analytics. Security is one of the major factors that is contributing towards the complexity as well as cost of development and maintenance of these systems. In fact, per the 2022 report published by IBM Security, the global average total cost of a data breach

increased to USD 4.35 million in 2022 and has been the highest in history. What's even more concerning is that it took an average of 207 days to identify the breach and 70 days to contain the breach.

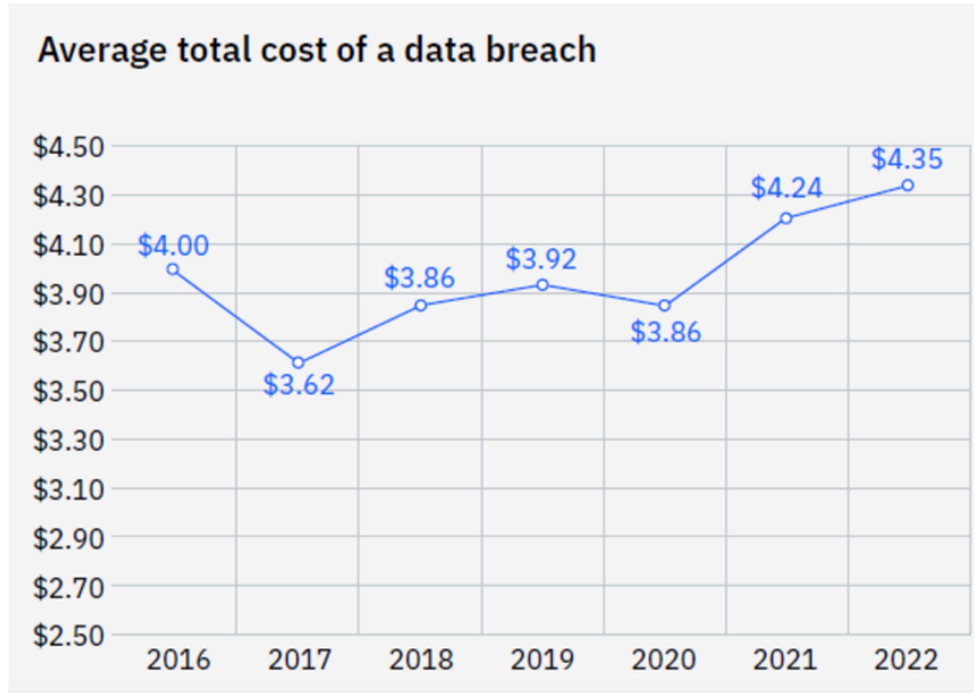
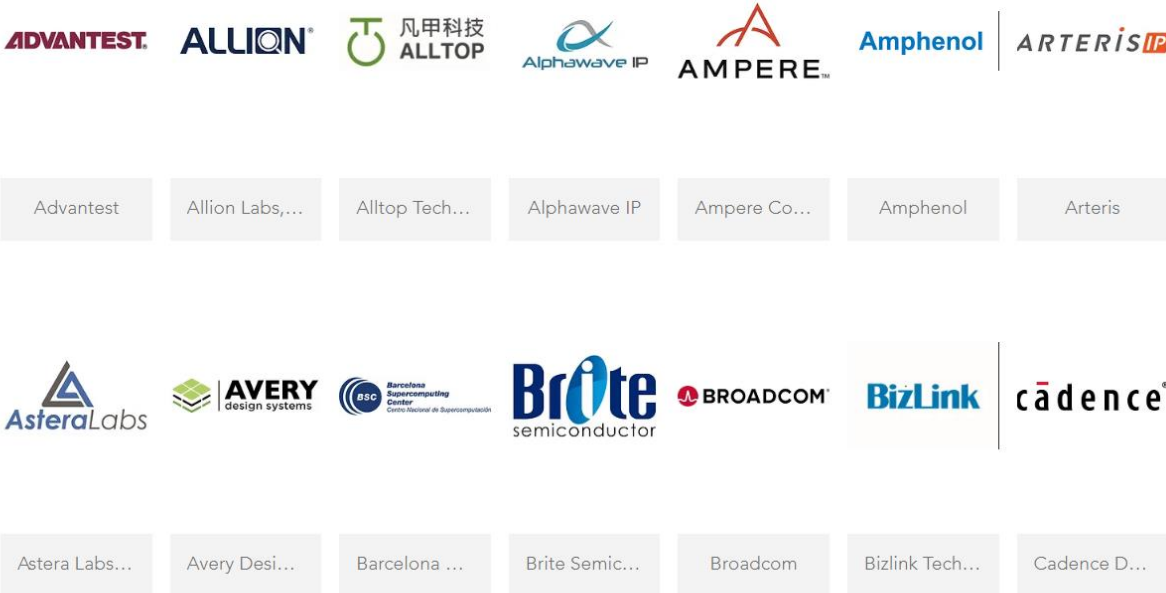


Figure 41: Average total cost of a data breach in USD millions (Source: IBM Security, 2022)

In addition to the cost, the complexity of threats that can breach these platforms by malicious actors has been increasing significantly over the past decade and it needs to be addressed with concrete security measures at the hardware, software, and protocol level on platform SoCs.

The **CXL Consortium** is an open industry standard group formed to develop technical specifications that facilitate breakthrough performance for emerging usage models while supporting an open ecosystem for data center accelerators and other high-speed enhancements.

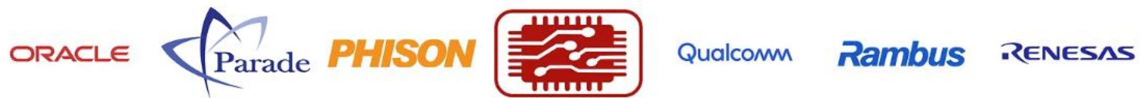
Figure 42: Board of Directors



Credo Sem… | Elastics.cloud | Enfabrica | Ericsson | Foxconn | Fujitsu Limi… | GigaIO

New H3C T… | Hitachi, Ltd. | Inspur | IntelliProp, … | JumpTrading | Keysight_Si… | KIOXIA Cor…

Microchip T… | Mobiveil, Inc. | Montage T… | Nanning te… | Netlist Inc | Numascale | Nyriad



Oracle Cor… | Parade Tec… | Phison Elec… | Primesoc T… | Qualcomm | Rambus_Lo… | Renesas

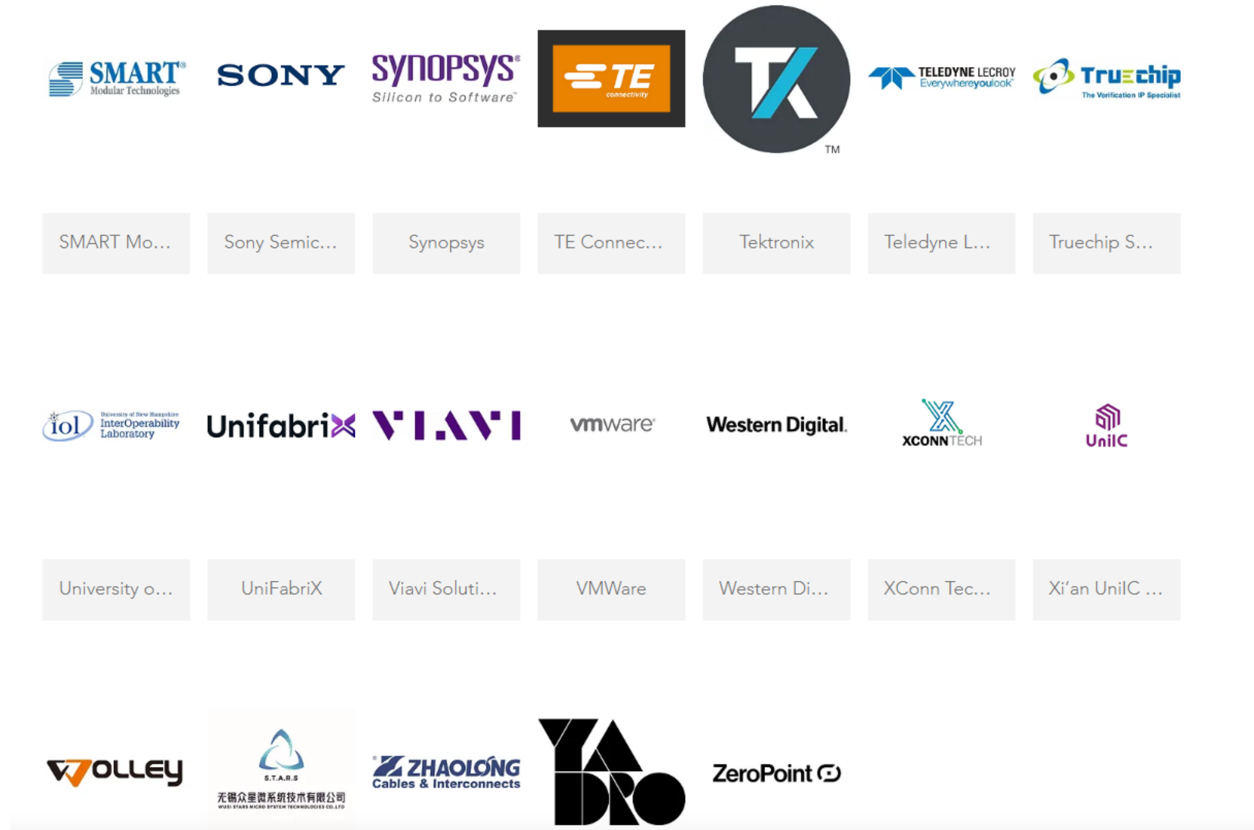


Figure 43: Contributors

Quotes from the CXL Consortium:

"Today's data center requires continuous innovation across the entire compute ecosystem including memory interface technologies in order to meet the performance and scalability demands of our customers," said Raghu Nambiar, corporate vice president, Data Center Ecosystems and Solutions at AMD. "Microchip's new SMC 2000 utilizes CXL interfaces for memory expansion and can greatly improve system performance. We are excited to work with Microchip to deliver a cohesive memory solution for our mutual customers and propel the computer industry forward to meet these next-generation data center needs."

"Cadence collaborated closely with Microchip on CXL verification and compliance testing, leveraging multiple Cadence Verification IP offerings to fine-tune the interconnect technology needed to advance performance for AI and HPC applications," said Paul

Cunningham, senior vice president and general manager of the System & Verification Group at Cadence. “Microchip’s release of the SMC 2000 CXL controller provides the memory bandwidth and capacity expansion required for the next generation of CPUs and GPUs to accelerate high-performance compute.”

“Dell is a strong promoter of CXL and is actively participating in the CXL Consortium and standards development. CXL provides the flexible infrastructure needed to optimize the TCO of current and emerging workloads on our future systems,” said Stuart Berke, fellow and VP at Dell. “We are excited to see Microchip’s SMC 2000 CXL-based Smart Memory Controllers enter the CXL memory ecosystem.”

“The momentum behind CXL is currently being fueled by the need for low-latency and high-bandwidth I/O solutions,” said Jim Pappas, Director of Technology Initiatives at Intel. “Microchip, with their SMC 2000 Smart Memory Controller, is a key contributor to the developing ecosystem, and we are pleased to see their investment to drive broader deployment of CXL devices and to enable rapid industry adoption.”

“As an active member of the CXL Consortium, Lenovo is committed to developing this important standard and helping build the ecosystem around the new CXL interconnect,” said Greg Huff, Chief Technology Officer, Lenovo Infrastructure Solutions Group. “We are excited to be part of developing solutions that enable a new era of data center performance and efficiency, working with Microchip to foster the growth and adoption of innovative CXL products in future Lenovo systems.”

“We strongly support the development of a rich ecosystem of innovative memory technologies that enhance system-scale capacity and performance,” said Raj Hazra, senior vice president and general manager of Micron’s Compute and Networking business unit. “CXL is a groundbreaking innovation that will open the door to composable system

architecture, facilitating new ways to connect Micron's industry-leading memory and storage."

"Having introduced the world's first ASIC-based CXL DRAM module, along with an open-source software toolkit, Samsung will continue to drive the commercialization of CXL products in collaboration with our customers and partners to meet the growing demand for data-heavy applications," said Cheolmin Park, vice president of Memory Global Sales & Marketing at Samsung Electronics, and director of the CXL Consortium. "We're delighted that Microchip's SMC 2000 Smart Memory Controllers will be able to deliver the memory performance and capacity scaling that the data center industry needs to manage increasingly memory-intensive workloads more cost efficiently."

"CXL memory solutions are expected to create many new opportunities in the future for the industry, with continuous emergence of more complex memory-bound future applications. It will allow customers to manage memories more efficiently through additional scaling in memory bandwidth and capacity at lower TCO. SK hynix expects that Microchip's SMC 2000 memory controller will provide a desirable solution to satisfy such needs and accelerate expanding the overall CXL ecosystem," said Uksong Kang, vice president of DRAM Product and Planning at SK hynix.

"SMART has designed Microchip's SMC 2000 into our CXL E3.S Memory Module (XMM) which is being adopted in new CXL-enabled platforms," states Satya Iyer, SMART Modular's vice president of Specialty Memory. Iyer continues, "SMART has extensive experience launching new products based on emerging industry interconnect standards, such as OpenCAPI DDIMMs, and is now working closely with Microchip to enable XMMs as one of the CXL products in our portfolio."

10. GP Estimated Cost, Tools, and Materials.

In fact, we are going through the digital IC design flow for this project to deliver an intellectual property (IP) for the CXL.Mem & CXL.Cache for both transaction layer and data link layer for the CXL 2.0 standard.

The cost is actually inherent in the tools that we use for simulation and verification for our design. These tools are provided to us by Si-Vision's exclusive contractor in the middle east, Synopsys.

10.1 Tools

1. **VCS**: The Synopsys VCS® functional verification solution is the primary verification solution used by a majority of the world's top semiconductor companies. VCS provides the industry's highest performance simulation and constraint solver engines. VCS' simulation engine natively takes full advantage of multicore processors with state-of-the-art Fine-Grained Parallelism (FGP) technology, enabling users to easily speed up high-activity, long-cycle tests by allocating more cores at runtime.
2. **SpyGlass Linting Tool:** Inefficiencies during RTL design usually surface as critical design bugs during the late stages of design implementation. If detected, these bugs will often lead to iterations, and if left undetected, they will lead to silicon re-spins. The SpyGlass® product family is the industry standard for early design analysis with the most in-depth analysis at the RTL design phase. SpyGlass provides an integrated solution for analysis, debug and fixing with a comprehensive set of capabilities for structural and electrical issues all tied to the RTL description of design.

3. **TestMax DFT:** Synopsys TestMAX DFT is a comprehensive, advanced design-for-test (DFT) tool that addresses the cost challenges of testing designs across a range of complexities. TestMAX DFT supports all essential DFT, including boundary scan, scan chains, core wrapping, test points, and compression. These DFT structures are implemented through TestMAX Manager for early validation of the corresponding register transfer level (RTL), or with Synopsys synthesis tools to generate netlists.

4. **Verdi Tool:** The Verdi® Automated Debug System is the centerpiece of the Verdi SoC Debug Platform and enables comprehensive debug for all design and verification flows. It includes powerful technology that helps you comprehend complex and unfamiliar design behavior, automate difficult and tedious debug processes and unify diverse and complicated design environments.

10.2 Materials

As mentioned in many occasions in this paper, the only document that we're only sticking to is the CXL's 2.0 first four chapters from the official standard (You will find its link in the references section).

10.3 Cost

The only Hardware kit that we will use is the Zynq FPGA in the figure below. It may be changed later according to the needs. But for sure we are going to test on an FPGA kit from ONE-Lab.

Note that this FPGA kit will be rented from ONE-Lab in Cairo University and not fully purchased as it's only used for the testing and prototyping purposes.

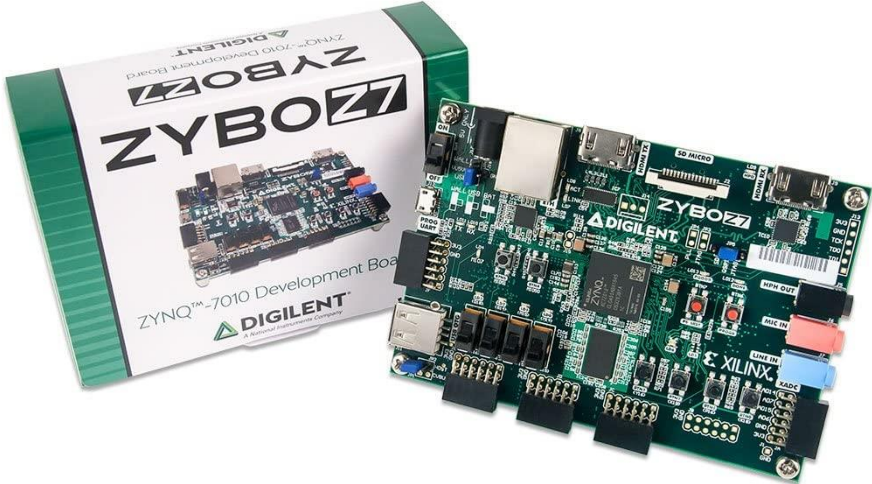


Figure 44: ZYNQ -7000 Arm FPGA

11. GP Timeline

Tasks	Description	Deliverables	Time (Weeks)
Study System Verilog	1. SV for RTL Design 2. SV always block types 3. Using interfaces to simplify module connectivity 4. Using struct to create encapsulated data structure		2

Synopsys Verdi Tool	1. Getting familiar with the tool2. Knowing how to run a simulation, open a wave, add signals, and debug.		1
CXL protocol	1. CXL 2.0 Specs Chapters 1 to 42. ARM CXS Specs		3
System architecture phase	1. Putting architecture to the whole system, describing main modules, high level connectivity, different clock domains.2. Preparing different (basic) scenarios.	1. Presentation describes the main features for the CXL protocol	3
System implementation phase	1. Describing every module on its own (Ports, main functionality, block diagram, timing diagrams)	1. Module level design documents	3
Module level RTL Implementation and testing	1. Implementing all modules as described in the previous step2. Unit testing of each module		4

System Integration & testing	1. Integrating all modules together and test the system 2. Hitting the testing scenarios prepared in architecture, and add any corner cases needed		5
ASIC flow	1. Lint/CDC checks (Spyglass)2. Synthesis (DC)3. DFT-TMAX (DC-TestMAX)4. Formality Checking (FV)	1. Clean lint and CDC2. Power, area and timingreports3. DFT ready design andStuck-at-coverage4. Clean FV	5
Thesis preparation	All the documentation made in previous stages will be added to thesis		3
Total			**29**

12. GP Final Deliverables

I. Micro-Architecture documentation for the controller and the main blocks.

II. Synthesizable System Verilog RTL

III. Simulation environment and tests

IV. Move through ASIC flow up to frontend Synthesis

V. (Synthesis + DFT + Formality checking)

13. References

https://www.computeexpresslink.org/download-the-specification (DOWLOAD THE STANDARD FROM HERE).

https://www.computeexpresslink.org/post/insight-into-cxl-2-0-security-features-and-benefits

https://www.businesswire.com/news/home/20190917005948/en/Compute-Express-Link-Consortium-CXL-Officially-Incorporates

https://www.synopsys.com/implementation-and-signoff/test-automation/testmax-dft.html

https://www.synopsys.com/implementation-and-signoff/test-automation/testmax-dft.html

https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html

https://www.synopsys.com/verification/simulation/vcs.html