ZeBu CXL Transactor User Manual

Version V-2024.03-SP1, February 2025

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Feedback

Contents

About This Manual

This manual describes how to use the ZeBu CXL transactor with your design being emulated in ZeBu.

Related Documentation

For more information about the ZeBu-supported features and limitations, see the ZeBu Release Notes in the ZeBu documentation package corresponding to your software version.

For more relevant information about the usage of the present transactor, see Using Transactors in the training material.

Synopsys Statement on Inclusivity and Diversity

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

1

Introduction

The ZeBu CXL transactor implements a Compute Express Link interface that can function as Host or Device and carry bit rates of 2.5, 5.0, 8.0, 16.0, 32.0, and 64.0 giga-transfers per second (GT/s).

This section explains the following topics:

Supported Transactor Configurations

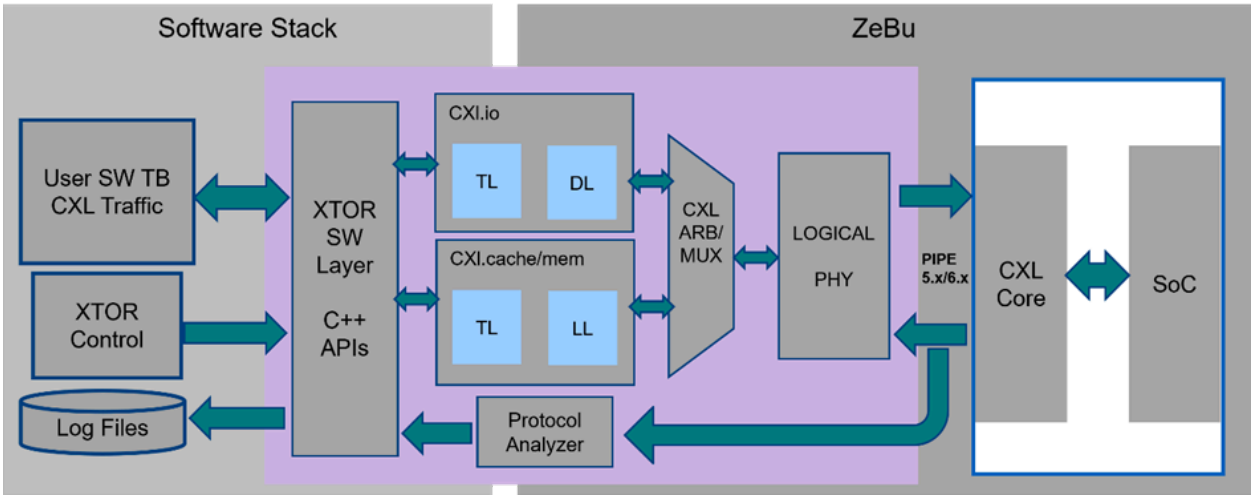
The ZeBu CXL transactor supports the following configurations:

- Link width -x16
- PIPE width
  - Original: 32 bits
  - SerDes: 40 bits(CXL 1.1, CXL 2.0), 80 bits(CXL 3.0)

Feedback

The following figure illustrates an example connection for the CXL Transactor:

Figure 1 CXL Transactor



To connect the PIPE interfaces of the transactor and the DUT, use a PCIe lane model (included with the PCIe Transactor package) or a custom lane model. The PCIe lane model is designed to model the behavior of PIPE sideband of the PHY transceivers.

The lane model included in the package provides support for the following CXL Link widths and Original PIPE/ SerDes PIPE widths:

Table 1 Supported PCIe Lane Model Configurations

Link Width	x1, x2, x4, x8, x16
Original PIPE Width (bits)	8, 16, 32, 64
SerDes PIPE Width (bits)	10, 20, 40, 80

Supported Protocol Specification

The following table lists supported protocol specifications

Table 2 Supported Protocol Specification

Protocol Specification	**Revision**
Compute Express Link (CXL) Specification	3.0
PCI Express® Base Specification	6.0

Feedback

Table 2 Supported Protocol Specification (Continued)

Protocol Specification	Revision
Intel PHY Interface for the PCI Express*, SATA, USB 3.2, DisplayPort*, and USB4* Architectures (PIPE) Specification	4.4.1, 5.1, 6.0

Supported Protocol Features

The following table lists supported protocol PCIe and CXL features

Table 3 Supported PCIe and CXL Features

Category	Features
Device Configuration	Host, Device, Device Mode
Link Rate (GT/s)	2.5, 5.0, 8.0, 16.0, 32.0, 64.0
Device Type	Type1, Type2 and Type3
Protocol	CXL.io, CXL.cache, CXL.mem
PIPE Optional Features	RxStandby/RxStandbyStatus Handshake
	RxValid synchronous to RxCLK in SerDes PIPE mode
Sideband Signals	CLKREQ#, WAKE#, PERST#
Fault Isolation	Advanced error reporting (AER)
Transaction Layer	All Cache requests and responses
	All Mem requests and responses
	Multiple CXL.cache D2H request and response interfaces to maximize throughput
	Multiple CXL.mem S2M Response/No response interfaces to maximize throughput
	Buried Cache state Support
	Back Invalidate Snoop
	BiConflict Request and Acknowledgement

Feedback

Table 3 Supported PCIe and CXL Features (Continued)

Category	Features
Transaction Layer	Interrupts (MSI, MSI-X, INTx)
	Process Address Space ID (PASID) & PASID Translation
	Atomic Operations
	ID-Based Ordering
	Latency Tolerance Reporting (LTR)
	Extended Tag
	TLP Processing Hints
	TLP Prefix
	Deferred Memory Write (DMWr)
	Designated Vendor-Specific Extended Capability (DVSEC)
	Optimized Buffer Flush/Fill(OBFF) till Gen5
	Downstream port containment(DPC)
Link Layer	68B, Standard 256B Flit
	68B CRC detection
	68B Flit Retry Mechanism
	Link Layer Initialization
	Flow Control Credit (FCC)
	Ack/Nak
ARB/MUX	CXL.io and CXL.Cachemem vLSM
	All vLSM States
	Status Syncronization Protocol
	Unexpected ALMP support

Feedback

Table 3 Supported PCIe and CXL Features (Continued)

Category	Features
FlexBus Physical Layer	256B mode retry mechanism
	256B CRC detection
	Equalization procedure
	Electrical Idle
	LTSSM
	Support for down configuration
	Precoding
	Gray Coding
	Hot-plug
	SRIS & SRNS Clocking Architecture
	Hot Reset, Cold Reset and Link Disable
CXL.io specific features	
Power Management	ASPM L1
	PCI-PM L1
	L1 Sub-states
	L2 till CXL 2.0
Security	IDE till CXL 2.0
SR-IOV	Support for 1 VF
Alternative Routing-ID Interpretation (ARI)	Support for 1 PF till Gen6
Address Translation Services (ATS)	ATS Invalidation Page Request Services

Supported Transactor Features

The following table lists supported transactor features

Feedback

Table 4 Supported CXL Transactor Features

Feature	Description
Testbench user interface provided at transaction-level protocol.	Provides a higher level of abstraction with packet-based design at TL and LL levels.
Easy integration with all ZeBu transactors	Facilitates quick building of a complete system-level test environment for complex SoC designs using various protocol transactors.
Protocol Analyzer controllable using APIs.	Facilitates efficient protocol-specific debug in complex SoC designs.

Limitations

The CXL transactor when configured as Device has the following limitations:

- Only single HW function, Virtual Function

The CXL transactor when configured as either Host or Device has the following limitations:

- Latency Optimized Flit in CXL3.0
- Virtual Channel is supported
- Port Based Routing (PBR) is not supported
- CXL 3.0 IDE
- L0p is not supported
- L2 state for CXL3.0
- Quality of Service (QoS)
- Global Persistent Flush
- HotPlug
- Memory Pooling W/ MLDs
- CXL.io traffic limited to 16 DWORD payload
- For CXL.io PPM testing is not supported as Elastic Buffers are not implemented.
- CXL3.0 driver Maxpayload size is limited to 1024Bytes only
- Data Object Exchange

Feedback

- Support for Loopback and Compliance is not present
- Does not support the following signals, with PIPE 4.4.1 connection:
  - mac_phy_asyncpowerchangeack
  - mac_phy_txcommonmode_disable
  - mac_phy_rxelecidle_disable
  - mac_phy_encodedecodebypass
  - mac_phy_blockaligncontrol
  - mac_phy_rxpolarity
  - mac_phy_deepmph
  - mac_phy_txmargin
  - mac_phy_txswing
  - mac_phy_txcompliance

2

Installing and Validating CXL Package

This section explains the following topics:

- Licensing
- Installation
- Package Directory Structure
- Validating the Installed Package

Licensing

To use the Zebu CXL transactor, install the following FLEXlm license feature:

Table 5 Installation protocol and corresponding license string

Protocol	License String
CXL 3.0	hw_xtormm_cxl3
CXL 2.0, CXL 1.1	hw_xtormm_cxl

Note:

Each license token is applicable for one Transactor instance.

If the hw_xtormm_cxl3 license is not installed, the restrictCxl3() should be called in TB to prevent CXL3 license being checked out and resulting in error.

Refer Configuration section for restrictCxl3() details.

Installation

To install the ZeBu CXL transactor, see the ZeBu Transactor Installation Guide.

Package Directory Structure

After the ZeBu CXL transactor is successfully installed, the transactor package consists of the following directories and files:

Table 6 ZeBu CXL Directory Structure

Directory	Sub directory/Files	Description
doc	ZeBu_XTOR_CXL_svs_UM.pdf	ZeBu CXL transactor User Manual
	ZeBu_VS_UM.pdf	ZeBu Vertical Solutions User Manual
	html/index.html	HTML documentation
example/src	bench	Testbench related files
	dut	Wrapper files
	env	Environment files
example/zebu	Makefile, zrci.tcl, Readme	• Makefile: Makefile contains the compilation flow for UC using the UTF wrapper file • Readme • tcl files
include	xtor_cxl_svs.hh	APIs for configuring transactor
	xtor_cxl_svs_defines.hh	Enums/defines/struct definition
	xtor_pcie_svs_tlp.hh	APIs related to TLPs
	xtor_pcie_svs_tlp_defines.hh	Enums/defines related to TLPs
	wrapper_cxl_svs.hh	APIs for configuring wrapper/transactor
	wrapper_cxl_svs_defines.hh	Enums/defines/struct definition
vlog/vcs	xtor_cxl_svs.sv	Driver
	xtor_pcie_svs_lanes_model.sv	PCIe Lane model
	Xtor_cxl_lpif_svs.sv	LPIF Driver
	xtor_cxl_lpif_lanes_model_svs.sv	CXL LPIF lane model
libABI1	libxtor_cxl_svs.so	.so file of CXL transactor

Feedback

Table 6 ZeBu CXL Directory Structure (Continued)

Directory	Sub directory/Files	Description
misc	cxl_wrapper_svs.sv	Wrapper module
	cxl_wrapper_lpif_svs.sv	LPIF wrapper module

The CXL Monitor is installed using a separate package ($ZEBU_IP_ROOT/monitor_cxl_svs). See CXL Monitor for details.

Validating the Installed Package

After installing the CXL Transactor, validate your installation for licensing. Also, run the example provided with the package.

This section consists of the following sub-sections:

- Setting Environment
- Running the Example
- Validating the Run

Setting Environment

Ensure the following environment variables are set correctly before running the examples:

Table 7 Environment Variables

Variable	Remark
ZEBU_IP_ROOT	Set to the package installation directory.
ZEBU_ROOT	Set to a valid ZeBu installation for compilation and run.
FILE_CONF	Set to your system architecture file (for example, on ZeBu Server. system: .../config/zse_configuration).
REMOTECMD	Set to perform remote ZeBu jobs.

Running the Example

To compile and run the example, perform the following steps:

1. Go to the <install_path>/xtor_cxl_svs/example/zebu directory.
2. Use the following Makefile target for compilation:

   ```
   $ make compile
   ```

   TG_COMP specifies the compile target. If not specified, by default, the transactor compiles for CXL3.0 and sets TG_COMP to 16x80b.
3. For compiling the CXL2.0 test, specify the following:

   ```
   $make compile TG_COMP=16x80b or TG_COMP=16x40b
   ```
4. When the example project is compiled successfully, you can run it using one of the following modes and its respective method:
   - **C++ testbench**

     ```
     $make run TG_RUN=cxl3p0_testbench
     ```
   - **zRci Testbench**

     ```
     $ make run TG_RUN=cxl3p0_testbench_zrci
     ```

Note:

To compile and run the example in pure simulation mode, specify the following:

- `$ make compile SIMULATOR=1`
- `$ make run SIMULATOR=1`

Validating the Run

The following message is displayed after a successful run:

```
-- ZeBu : zServer : End of run :
-- ZeBu : zServer : Unit 0 masterClk cycle counter : 201,120,815
 (0x0bfcdc2f)
-- ZeBu : zServer : Unit 0 driverClk cycle counter : 8,360,804
 (0x007f9364)
-- ZeBu : zServer : Unit 0 driverClkDut cycle counter : 8,360,804
 (0x007f9364)
-- ZeBu : zServer : Unit 0 tickClk edge counter : 188,849 (0x0002e1b1)
-- ZeBu : zServer : Unit 0 globalTime counter : 188,849 (0x0002e1b1)
```

Run Logs generated during ZeBu run can be found under rundir_*.zebu/log.

See the Troubleshooting section for more information.

3

Introducing CXL Wrapper Module

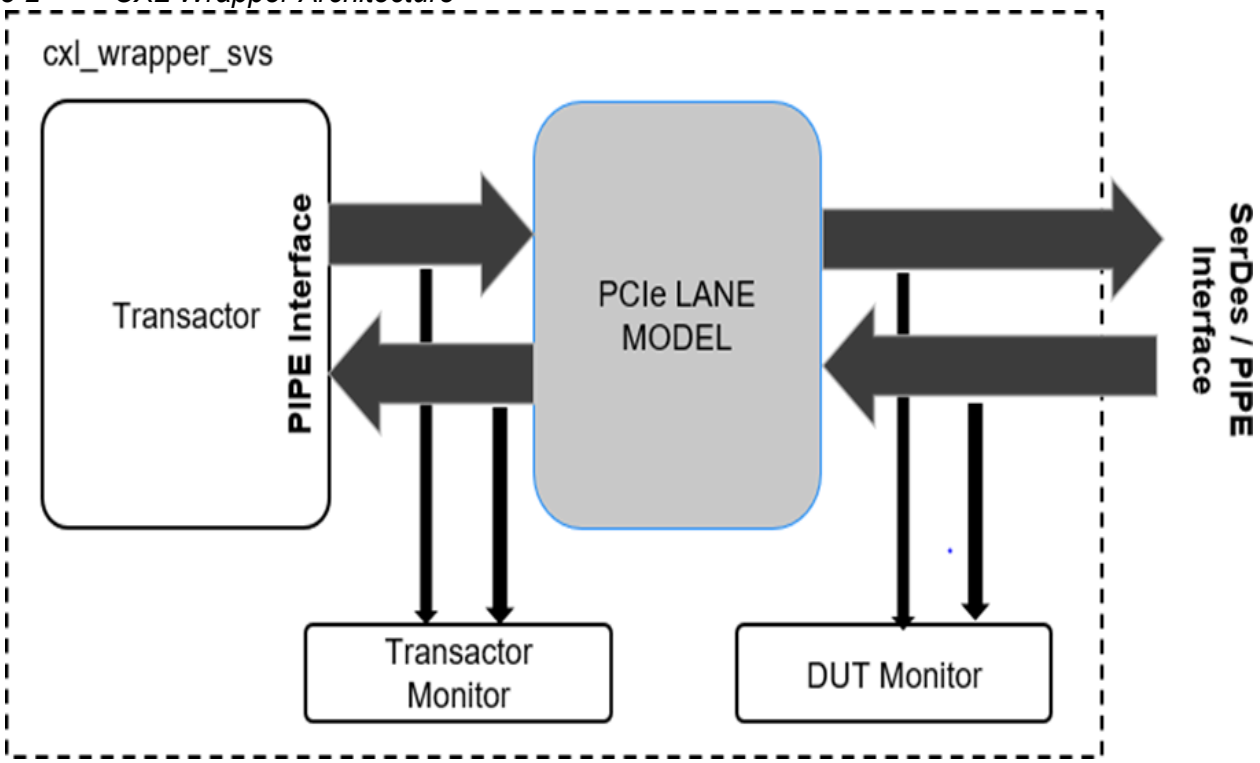
This section provides the following information on the CXL Wrapper module:

Components of the CXL Wrapper

The cxl _wrapper_svs is a wrapper module that enables ease of usage of the CXL Transactor by encapsulating CXL transactor’s connection with the PCIe lane model and monitor.

Feedback

Figure 2 CXL Wrapper Architecture



The wrapper consists of the following components, which are instantiated and connected in the same module:

Table 8 CXL Wrapper Components

Wrapper Component	Module Name
CXL transactor	xtor_cxl_svs
PCIe lane model	xtor_pcie_svs_lanes_model
Transactor monitor	monitor_top_inst_xtor
DUT monitor	monitor_top_inst_dut

Note:

See $ZEBU_IP_ROOT/xtor_cxl_svs/misc/cxl_wrapper_svs.sv for more details of the module.

You can enable/disable the monitors using the parameters IMPLEMENT_MONITOR_XTOR and IMPLEMENT_MONITOR_DUT. Only One Parameter can be set at a time.

CXL Wrapper Parameters

The following table lists the cxl_wrapper_svs parameters:

Table 9 The cxl_wrapper_svs Parameters

Parameter Name	Default Value	Possible Value(s)	Description
DUT_LANES	16	1, 2, 4, 8, 16	Number of DUT lanes.
PIPE_G1	Original PIPE: 32 SerDes PIPE: 4 (upto CXL2), 8 (CXL3)	Original PIPE (in bits): 8, 16, 32 SerDes PIPE (in symbols): 1, 2, 4	PIPE width at Gen1
PIPE_G2		Original PIPE (in bits): 8, 16, 32 SerDes PIPE (in symbols): 1, 2, 4	PIPE width at Gen2
PIPE_G3		Original PIPE (in bits): 16, 32, 64 SerDes PIPE (in symbols): 2, 4, 8	PIPE width at Gen3
PIPE_G4		Original PIPE (in bits): 8, 16, 32, 64 SerDes PIPE (in symbols): 1, 2, 4, 8	PIPE width at Gen4
PIPE_G5		Original PIPE (in bits): 32, 64 SerDes PIPE (in symbols): 4, 8	PIPE width at Gen5
PIPE_G6		SerDes PIPE (in symbols): 8	PIPE width at Gen6
FREQ_G1	625	(in 100 kHz unit): 625, 1250, 2500, 10000	PHY frequency at Gen1
FREQ_G2	1250	(in 100 kHz unit): 1250, 2500, 5000, 10000	PHY frequency at Gen2
FREQ_G3	2500	(in 100 kHz unit): 1250, 2500, 5000, 10000	PHY frequency at Gen3
FREQ_G4	5000	(in 100 kHz unit): 2500, 5000, 10000, 20000	PHY frequency at Gen4

Feedback

Table 9 The cxl_wrapper_svs Parameters (Continued)

Parameter Name	Default Value	Possible Value(s)	Description
FREQ_G5	10000	(in 100 kHz unit): 5000, 10000	PHY frequency at Gen5
FREQ_G6	10000	(in 100 kHz unit): 10000	PHY frequency at Gen6
IS_ROOTPORT	1	0: Endpoint 1: Root Complex	Transactor configuration
MAX_PHY_RATE	6	1, 2, 3, 4, 5, 6	Max Speed of transactor
CXL_MAXPAYLOAD	1024	(in DW): 32, 64, 128, 256, 512, 1024	CXL Max Payload Size of the transactor
IMPLEMENT_MONITOR_XTOR	0	0: Not Enabled 1: Enabled	Enables transactor side monitor
IMPLEMENT_MONITOR_DUT	1	0: Not Enabled 1: Enabled	Enables DUT side monitor
DUT_PIPE_VERSION	6	4, 5, 6	PIPE version supported by DUT
DUT_RXSTANDBY_ACTIVE_VALUE	1	0: Not Supported 1: Supported	Controls RxStandby/RxStandbyStatus handshake support of DUT

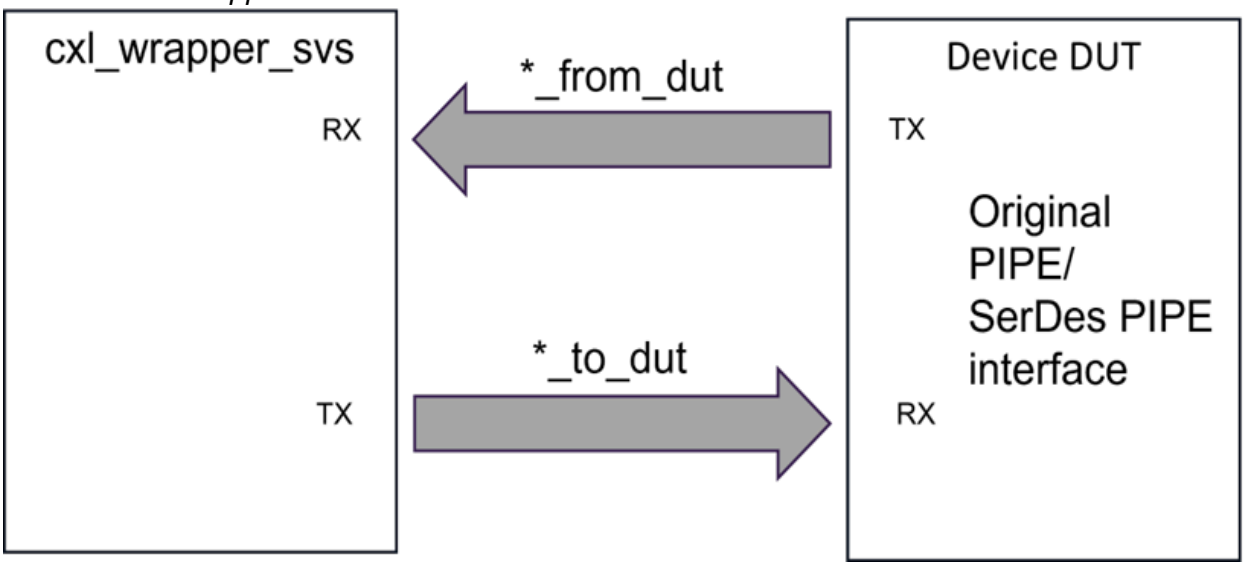
CXL Wrapper Signal List

When using the cxl_wrapper_svs module, the DUT interface signaling *_from_dut/ *_to_dut connection is established, irrespective of whether the DUT is Host or Device.

Feedback

The following figure illustrates an example connection between the CXL wrapper and DUT:

Figure 3 CXL Wrapper and DUT connection



This section describes the signals list for the following:

- Common Signals for Original and SerDes PIPE
- Original PIPE Only Signal List
- SerDes PIPE Only Signal List

Common Signals for Original and SerDes PIPE

The following table lists the common signals for original and SerDes PIPE:

Table 10 Common Signal for Original and SerDes PIPE

Name	Size(bits)	Direction	Description
Clock and Reset Shared Signals			
perst_n	1	Input	Active Low Fundamental Reset signal.
fastest_pcie_clk	1	Input	Global clock which indicates the fastest clock at Transactor side. Its frequency should be set up according to MAX_PHY_RATE parameter and by default, it is Gen5 speed.
npor_out	1	Output	Reflects the status of the perst_n.

Feedback

Table 10 Common Signal for Original and SerDes PIPE (Continued)

Name	Size(bits)	Direction	Description
Clock and Reset Shared Signals			
pl_rstn_from_dut	1	Input	Active Low signal. Provides Reset signaling from DUT.
phy_clk_from_dut	1	Input	PHY clock from DUT. Its frequency should be aligned with the DUT PIPE width.
Shared Signals			
clkreq_oen_from_dut	1	Input	Active low signal. Used for Low Power signaling.
clkreq_in_n_to_dut	1	Output	Active low signal. Provides Low Power clock signaling status.
ltssm_state_from_dut	6	Input	DUT state machine encoding.
rate_info_from_dut	4	Input	PHY Rate signaling from DUT.
powerdown_from_dut	4	Input	Powers up or down the transceiver.
Per Lane Signals			
pclkchangeok_to_dut	1	Output	Optional. Clock change acknowledgment grant.
pclkchangeack_from_dut	1	Input	Optional. Clock change acknowledgment request.
rxstandby_from_dut	1	Input	Determine if the PHY Rx is active when the PHY is in P0 or P0s. 0: Active 1: Standby Used when the parameter DUT_RXSTANDBY_ACTIVE_VALUE is set to 1.
rxstandbystatus_to_dut	1	Output	Status to indicate RxStandby state: 0: Active 1: Standby Used when the parameter DUT_RXSTANDBY_ACTIVE_VALUE is set to 1.
txdetectrx_from_dut	1	Input	Used to tell the PHY to begin a receiver detection operation or to begin loopback.

Feedback

Table 10 Common Signal for Original and SerDes PIPE (Continued)

Name	Size(bits)	Direction	Description
Clock and Reset Shared Signals			
txdatavalid_from_dut	1	Input	This signal allows the MAC to instruct the PHY to ignore the data interface for one clock cycle.
txelecidle_from_dut	PIPE 4.4.1: 1 PIPE 5.1 and above: 4	Input	Forces Tx output to electrical idle when asserted.
phystatus_to_dut	1	Output	Used to communicate the completion of several PHY functions including stable PCLK after Reset# de-assertion, power management state transitions, rate change, and receiver detection.
rxstatus_to_dut	3	Output	Encodes receiver status and error codes for the received data stream when receiving data.
rxelecidle_to_dut	1	Output	Indicate receiver detection of an Electrical Idle for each lane.
rxvalid_to_dut	1	Output	Indicate Symbol lock and valid data for each lane.
txdata_from_dut	Original PIPE: 8, 16, 32, or 64 SerDes PIPE: 10, 20, 40, or 80	Input	PHY Tx data bus.
rxdata_to_dut		Output	PHY Rx data bus.
m2p_messagebus_from_dut	8	Input	Low Pin Count message bus.
p2m_messagebus_to_dut	8	Output	Low Pin Count message bus.
PIPE 4.4.1			
rxeqeval_from_dut	1	Input	The PHY starts evaluation of the far end transmitter TX EQ settings when the MAC asserts this signal.
getlocalpresetcoefficients_from_dut	1	Input	A MAC holds this signal high for one PCLK cycle requesting a preset to co-efficient mapping for the preset on LocalPresetIndex[3:0] to coefficients on LocalTxPresetCoefficient[17:0].

Feedback

Table 10 Common Signal for Original and SerDes PIPE (Continued)

Name	Size(bits)	Direction	Description
Clock and Reset Shared Signals			
invalidrequest_from_dut	1	Input	Indicates that the Link Evaluation feedback requested a link partner TX EQ setting that was out of range.
txdeemph_from_dut	18	Input	Selects transmitter de-emphasis.
fs_from_dut	6	Input	Provides the FS value advertised by the link partner.
lf_from_dut	6	Input	Provides the LF value advertised by the link partner.
rxpresethint_from_dut	3	Input	Provides the RX preset hint for the receiver.
localpresetindex_from_dut	5	Input	Index for local PHY preset coefficients requested by the MAC.
localtxcoefficientsvalid_to_dut	1	Output	A PHY holds this signal high for one PCLK cycle to indicate that the LocalTxPresetCoefficients[17:0] bus correctly represents the coefficients values for the preset on the LocalPresetIndex bus.
localtxpresetcoefficients_to_dut	18	Output	These are the coefficients for the preset on the LocalPresetIndex[3:0] after a GetLocalPresetCoeffcients request: [5:0] C-1 [11:6] C0 [17:12] C+1
localfs_to_dut	6	Output	Provides the FS value for the PHY.
localIf_to_dut	6	Output	Provides the LF value for the PHY.
linkevaluationfeedbackfiguremerit_to_dut	8	Output	Provides the PHY link equalization evaluation Figure of Merit value.
linkevaluationfeedbackdirectionchange_to_dut	6	Output	Provides the link equalization evaluation feedback in the direction change format.

Note:

If PclkChangeAck is not supported by DUT, tie this signal value to 1 for each lane supported by DUT.

Original PIPE Only Signal List

The following table lists the original PIPE only signals:

Table 11 Original PIPE Only Signal List

Name	Size(in bits)	Direction	Description
Per Lane Signals			
txstartblock_from_dut	1	Input	Only used at 8.0 GT/s, 16 GT/s, and 32 GT/s. Enables MAC to communicate with PHY about starting byte for a 128b block.
txsyncheader_from_dut	2	Input	Only used at 8.0 GT/s, 16 GT/s, and 32 GT/s. Provides the sync header for the PHY to use in the next 130b block.
txdatak_from_dut	PIPE_G* /8	Input	Data/Control for the symbols of Tx data.
rxdatavalid_to_dut	1	Output	Allows the PHY to instruct the MAC to ignore the data interface for one clock cycle.
rxstartblock_to_dut	1	Output	Only used at the 8.0 GT/s, 16 GT/s, and 32 GT/s. Allows the PHY to tell the MAC that the starting byte for a 128b block.
rxsyncheader_to_dut	2	Output	Only used at 8.0 GT/s, 16 GT/s, and 32 GT/s. Provides the sync header for the MAC to use in the next 130b block.
rxdatak_to_dut	PIPE_G* /8	Output	Data/Control for the symbols of Rx data.

SerDes PIPE Only Signal List

The following table lists the signals for the SerDes PIPE:

Table 12 SerDes PIPE Only Signal List

Name	Size(in bits)	Direction	Description
Per Lane Signals			
rxclk_to_dut	1	Output	Recovered clock used for RxData in the SerDes architecture.

Feedback

Supported CXL Wrapper Configurations

The CXL transactor is a fixed 32-bit PIPE width and x16 lanes implementation. The PCIe lane model available in the wrapper module is used to adapt the transactor PIPE interface to the DUT PIPE interface.

Table 13 Supported CXL Configurations

Generation/Rate	XTOR		DUT		TxDataValid/RxDataValid Strobe Rate
	PCLK	Width (SerDes) (bits)	PCLK (100 kHz)	Width (SerDes) (bits)	
Gen1 (2.5 GT/s)	625	32 (40)	2500	8 (10)	N/A
			1250	16 (20)	N/A
			625	32 (40)	N/A
			312.5	N/A	N/A
			10000	8 (10)	Yes (1 in 4 PCLKs)
Gen2 (5.0 GT/s)	125	32 (40)	5000	8 (10)	N/A
			2500	16 (20)	N/A
			1250	32 (40)	N/A
			625	N/A	N/A
			10000	8 (10)	Yes (1 in 2 PCLKs)
Gen3 (8.0 GT/s)	250	32 (40)	10000	Not Supported	N/A
			10000	32 (40)	Yes (1 in 4 PCLKs)
			10000	16 (20)	Yes (1 in 2 PCLKs)
			5000	16 (20)	N/A
			2500	32 (40)	N/A
			1250	64 (80)	N/A

Feedback

Table 13 Supported CXL Configurations (Continued)

Generation/Rate	XTOR		DUT		TxDataValid/RxDataValid Strobe Rate
	PCLK	Width (SerDes) (bits)	PCLK (100 kHz)	Width (SerDes) (bits)	
Gen4 (16.0 GT/s)	5000	32 (40)	20000	8 (10)	N/A
			10000	16 (20)	N/A
			5000	32 (40)	N/A
			2500	64 (80)	N/A
			10000	32 (40)	Yes (1 in 2 PCLKs)
Gen5 (32.0 GT/s)	10000	32 (40)	40000	Not Supported	N/A
			20000	Not Supported	N/A
			10000	32 (40)	N/A
			5000	64 (80)	N/A
Gen6 (64.0 GT/s)	10000	N/A (80)	10000	N/A (80)	N/A
			20000	N/A (40)	N/A
			40000	N/A (20)	N/A

4

Integrating CXL Transactor and DUT

This section explains the following topics:

- Topologies and Connections
- Initialization
- Configuration
- Running the Transactor Example

Topologies and Connections

This section describes the following CXL Transactor testbench topologies and their connections:

Table 14 CXL Transactor Topologies

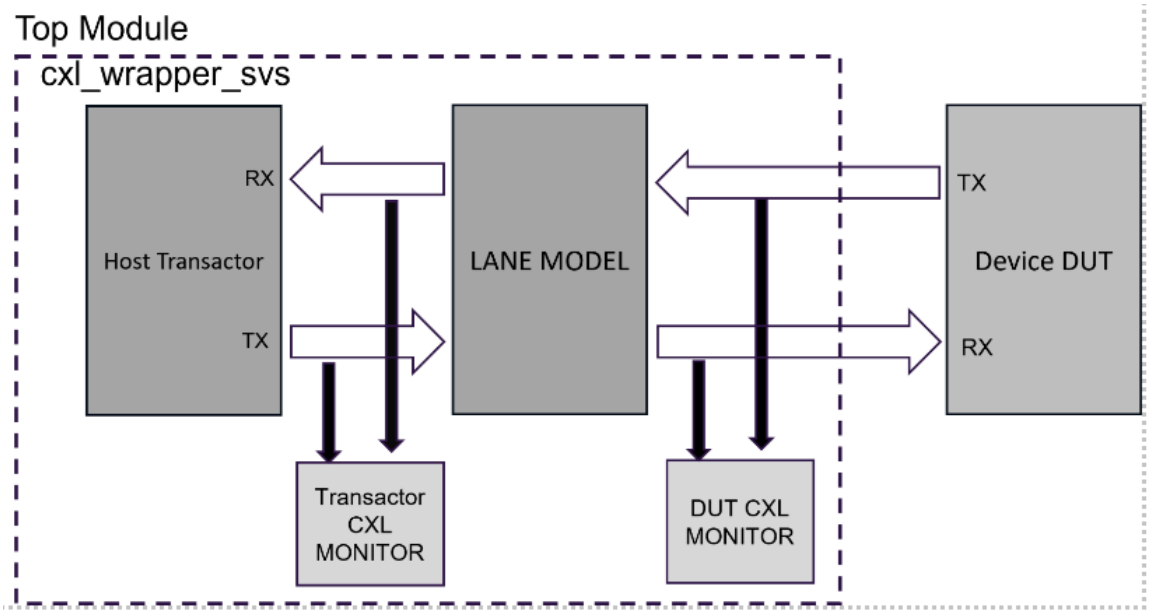
Topology	Description	Reference Figure
Host Transactor and Device DUT	Host instance of cxl_wrapper_svs connected with Device DUT	See Figure 4
Device Transactor and Host DUT	Device instance of cxl_wrapper_svs connected with Host DUT	See Figure 9
Host Transactor and PHY DUT	Host instance of cxl transactor connected with PHY DUT	Figure 11
Device Transactor and PHY DUT	Device instance of cxl transactor connected with PHY DUT	Figure 15

Host Transactor and Device DUT

The cxl_wrapper_svs parameter “IS_ROOTPORT” for this topology is set to 1.

Feedback

Figure 4 Host Transactor with Device DUT



To create a connection between the Device DUT and the Host transactor, perform the following steps:

1. Instantiate the following mandatory components in the top-level wrapper module :
   - CXL Wrapper as Host: cxl_wrapper_svs
   - Device DUT
2. Connect the cxl_wrapper_svs and DUT. see the reference figures in Table 15:

Table 15 CXL Wrapper and DUT Connection

DUT PIPE mode	DUT_PIPE_VERSION	Reference Figures
Original PIPE	5	See Figure 5
Original PIPE	4	See Figure 6
SerDes PIPE	6	See Figure 7
SerDes PIPE	5	See Figure 7
SerDes PIPE	4	See Figure 8

Feedback

The DUT can use either PIPE 4 or PIPE 5 interfaces. The unused inputs dedicated to PIPE 4 should be tied to 0, and the unused output should not be connected to any interface.

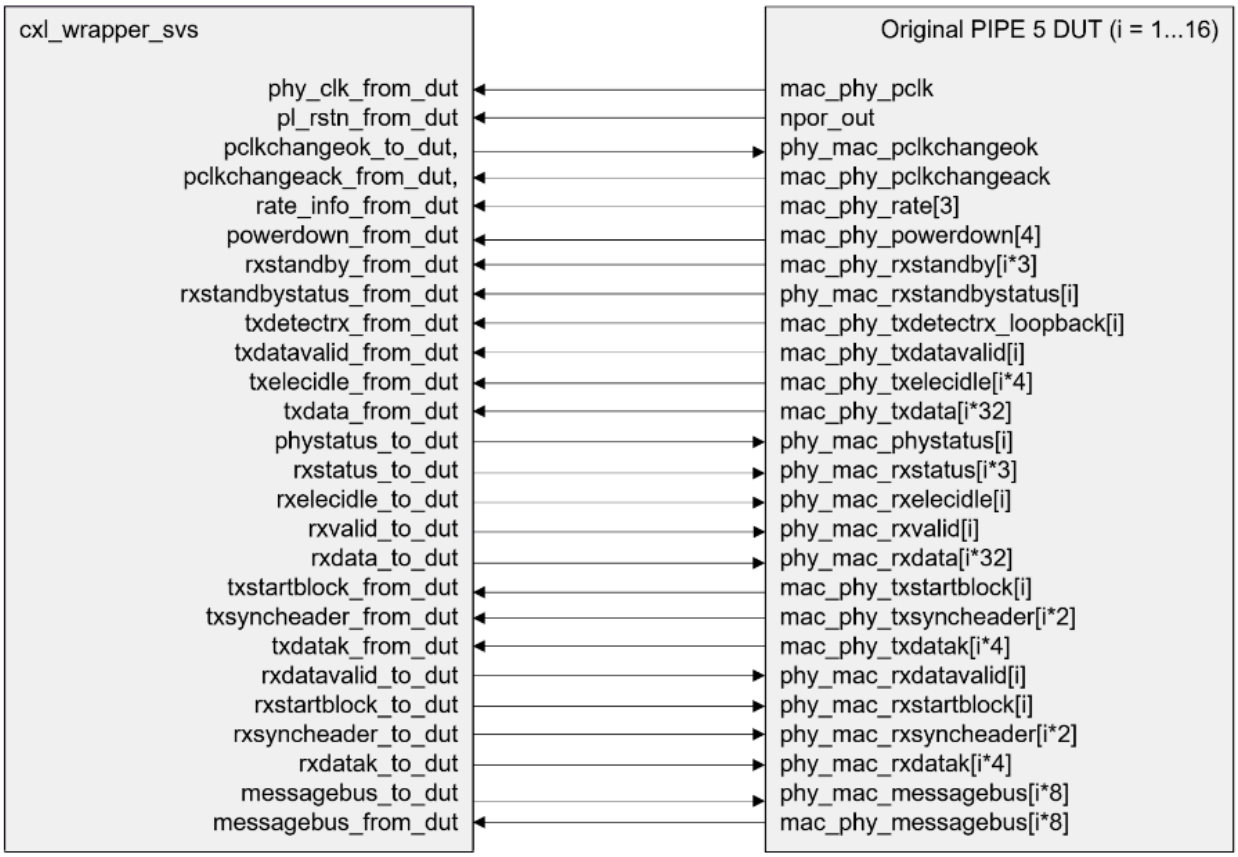
3. Connect *zceiClockPort* or *RTL clock* for clock as shown in Connecting Clocks section.

Note:
The following package contains a reference example for the hardware pin connection: *$ZEBU_IP_ROOT/xtor_cxl_svs/example/src/dut/wrapper.v.*

Original PIPE Interface with DUT

The following figure illustrates original PIPE 5 DUT to wrapper connection:

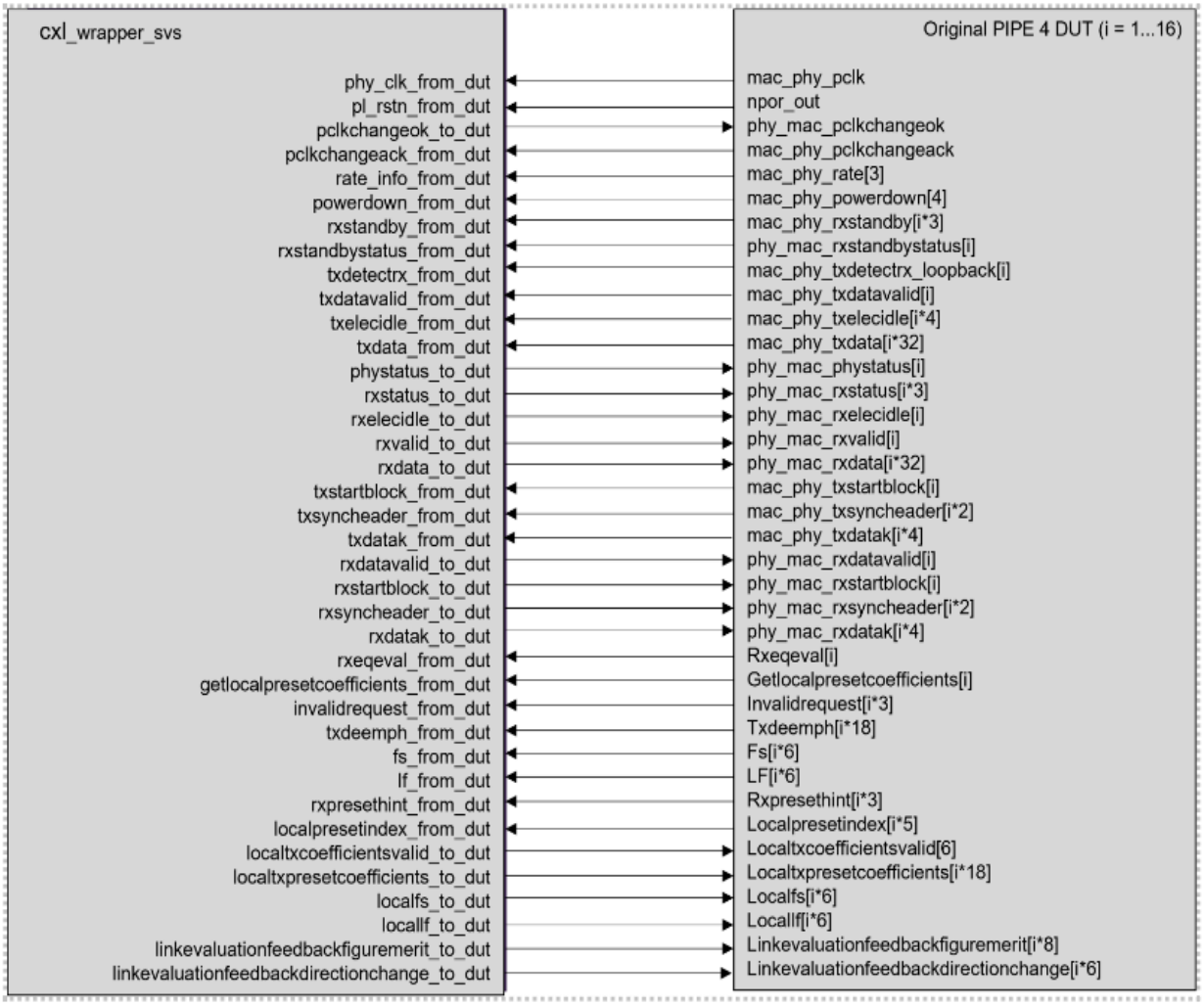
Figure 5 OriginalPIPE 5 DUT to Wrapper Connection



Feedback

The following figure illustrates the Original PIPE 4 DUT to Wrapper Connection:

Figure 6 Original PIPE 4 DUT to Wrapper Connection



Feedback

SerDes PIPE Interface with DUT

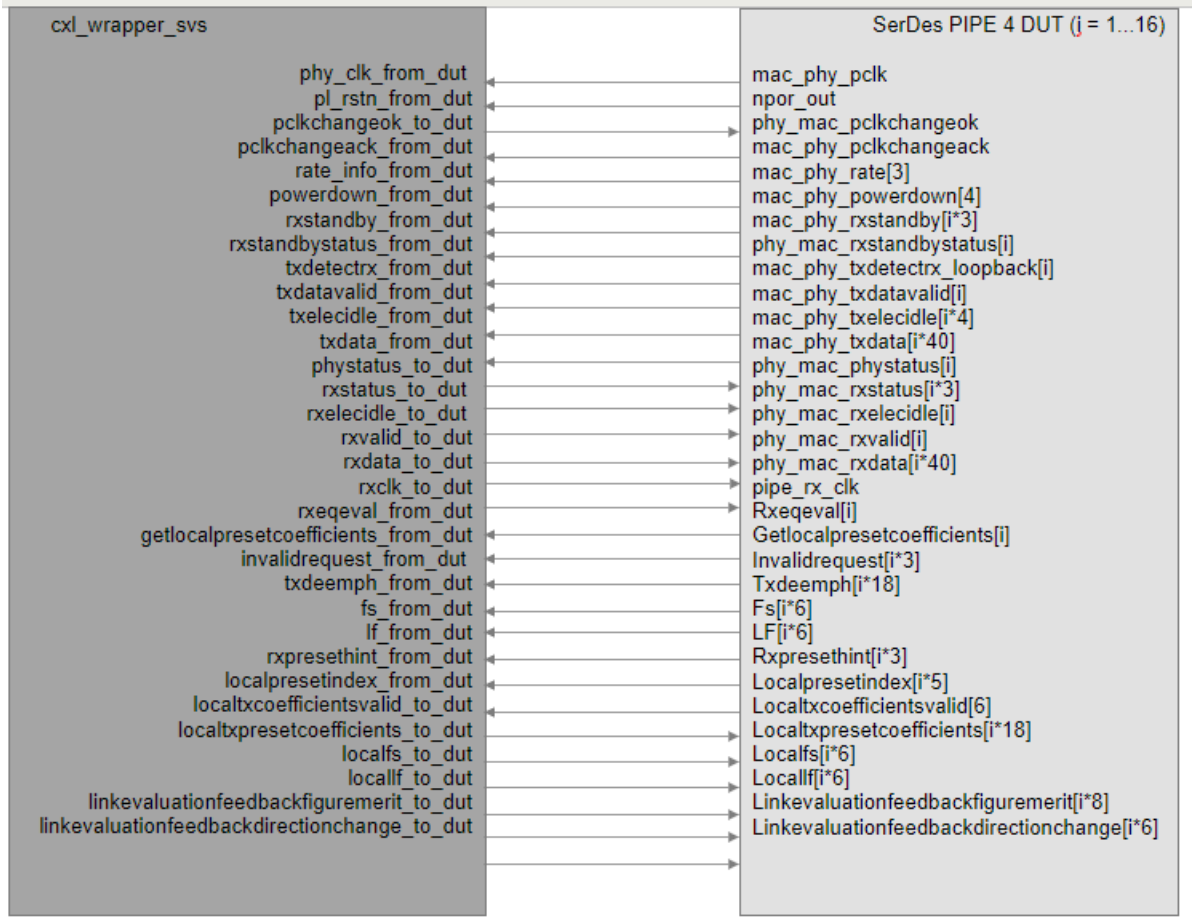
The following figure illustrates the SerDes PIPE 5/PIPE 6 DUT to wrapper connection:

Figure 7 SerDes PIPE 5/6 DUT to Wrapper Connection

Feedback

The following figure illustrates the SerDes PIPE4 DUT to wrapper connection:

Figure 8 SerDes PIPE 4 DUT to Wrapper Connection

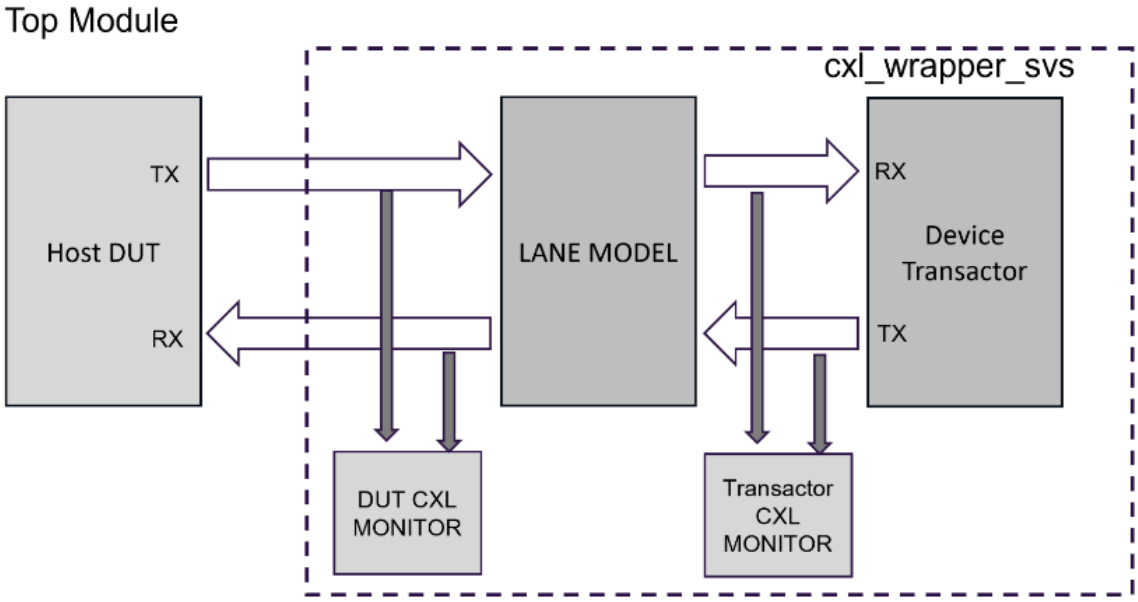


Device Transactor and Host DUT

The cxl_wrapper_svs parameter “IS_ROOTPORT” in this topology is set to 0.

Feedback

Figure 9 Device Transactor with Host DUT



To create a connection between the Host DUT and the Device transactor, perform the following steps:

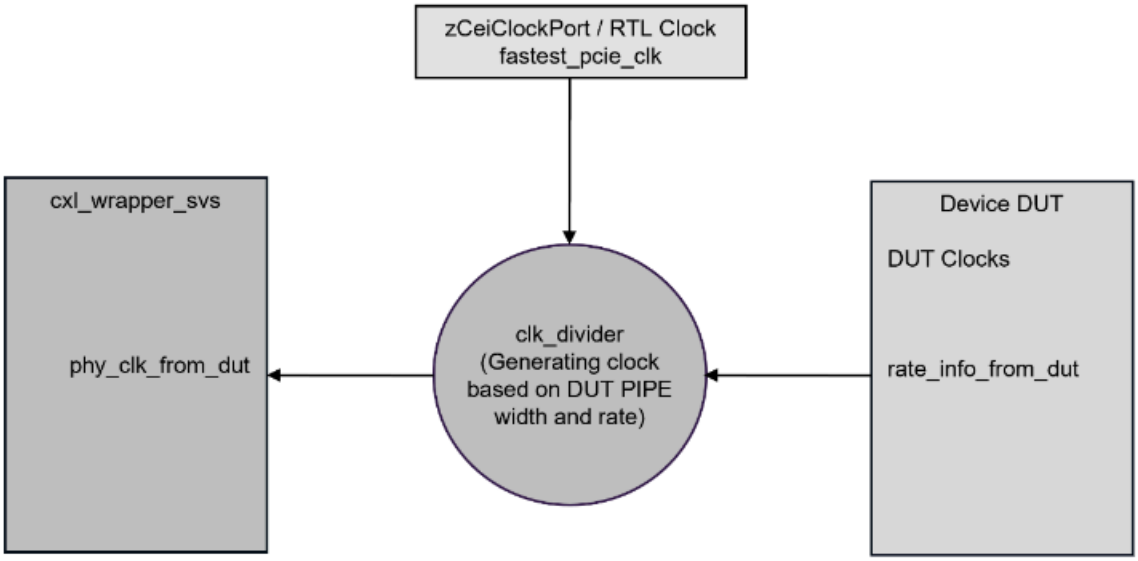
1. Instantiate the following mandatory components in the top-level wrapper module:
   - CXL Wrapper as Device: cxl_wrapper_svs
   - Host DUT
2. Connect the cxl_wrapper_svs and DUT according to the reference figures in the Table 14
3. Connect zceiClockPort or "RTL clock" for clock as shown in Connecting Clocks section.

Connecting Clocks

The following figure illustrates the connection between the primary clock and the DUT clock.

Feedback

Figure 10 Connecting the phy_clk_from_dut clock to a primary clock



The cxl_wrapper_svs is synchronous to the phy_clk_from_dut clock. Ensure to connect it to the appropriate source according to the operating rate (Gen1, Gen2, Gen3, Gen4, Gen5, Gen6) and width of the PIPE DUT interface. Also, perform the clock source selection in the top-level wrapper.

For example, the requirements for phy_clk_from_dut input of the cxl_wrapper_svs module when connected to 8bits (Gen1), 8 bits (Gen2), 16 bits (Gen3), 32 bits (Gen4), and 32 bits (Gen5), the Original PIPE DUTs are as follows:

- phy_clk_from_dut: 250 MHz at Gen1 rate
- phy_clk_from_dut: 500 MHz at Gen2 rate
- phy_clk_from_dut: 500 MHz at Gen3 rate
- phy_clk_from_dut: 500 MHz at Gen4 rate
- phy_clk_from_dut: 500 MHz at Gen5 rate

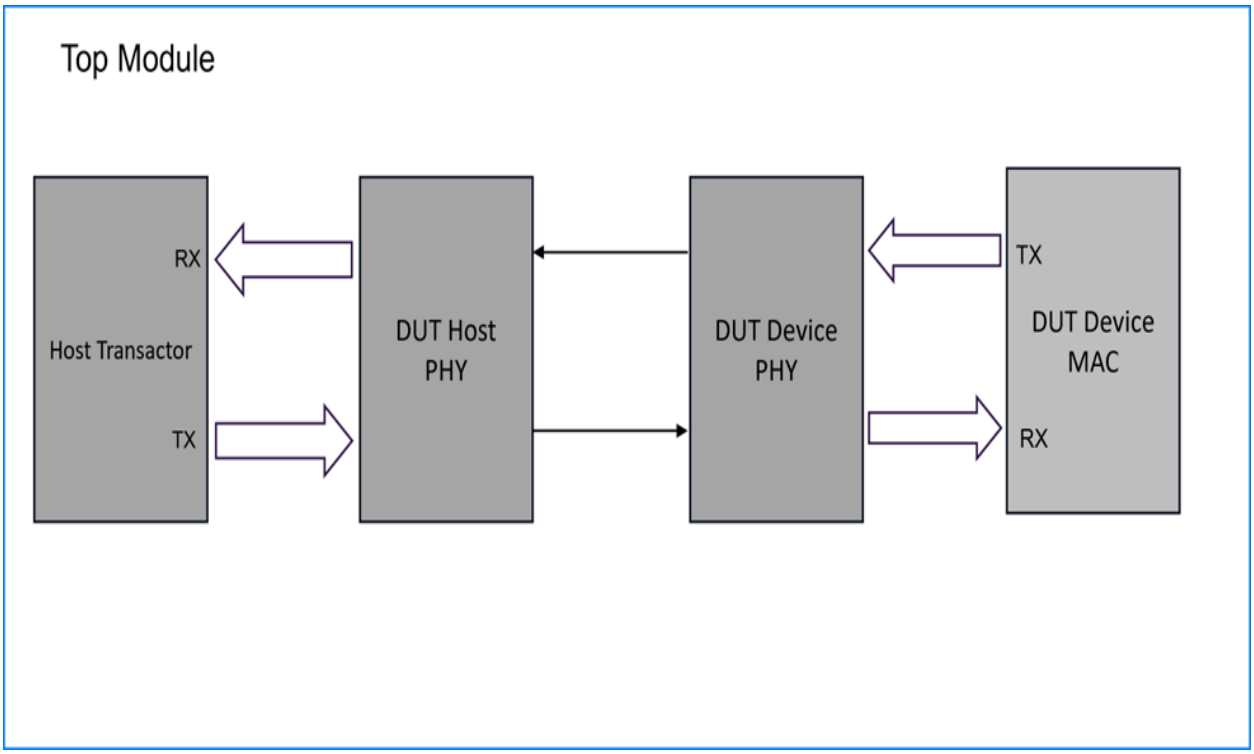
Feedback

Host Transactor and PHY DUT

For this topology, the xtor_svs_svs module, device_type signal, is set to 4. Host transactor is connected to the PHY DUT with the following configuration:

- PIPE width of 32-bit (Original PIPE) / 40-bit (SerDes PIPE)
- x16 link width

Figure 11 Host Transactor and PHY DUT



To create a connection between the PHY DUT and Host transactor, perform the following steps:

1. In the top-level wrapper module, instantiate the following mandatory components:
   - CXL Transactor as Host: xtor_cxl_svs
   - PHY DUT

Feedback

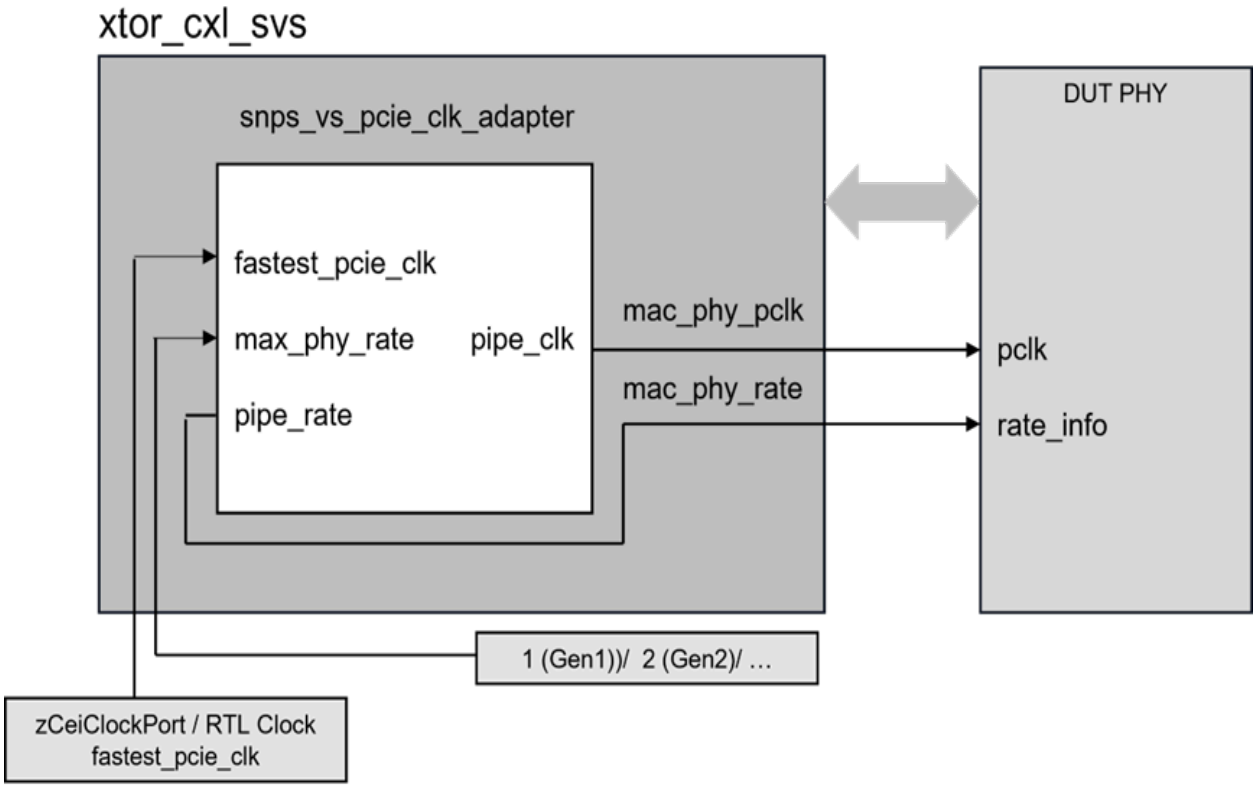
2. Connect the xtor_cxl_svs interface parameters and signals (described in Transactor Interface Parameter and Transactor Interface Signal List) with PHY DUT according to the reference figures in the following table:

Table 16 Transactor and PHY DUT connection

DUT PIPE Mode	Reference Figure
Original PIPE	
SerDes PIPE	

3. Connect zceiClockPort or RTL clock for transactor as shown in the following figure:

Figure 12 Clock Connection



- The transactor physical clock generator consists of the following blocks:
  - fastest_pcie_clk/2
  - fastest_pcie_clk/4

- fastest_pcie_clk/8
- fastest_pcie_clk/16

The programmable mux block selects from the above 4 sub-clocks according to the current data rate and connects the selected sub clock to the output pclk port.

Transactor Interface Parameter

The following table lists the xtor_cxl_svs parameter:

Table 17 Parameter List

Parameter Name	Default Value	Possible Values	Description
PCIE_MAXPAYLOAD	1024	(in DW): 32, 64, 128, 256, 512, 1024	PCIe Max Payload Size of the Transactor.

Transactor Interface Signal List

This section describes the Transactor signals list for the following:

- Common Signals for Original and SerDes PIPE
- Original PIPE Only Signal List
- SERDES PIPE Only Signal List

Common Signals for Original and SerDes PIPE

The following table lists the common signals for the original and SerDes PIPE in Transactor:

Name	Size (in bits)	Direction	Description
Clock and Reset Shared Signals			
perst_n	1	Input	Active Low Fundamental Reset signal.

Name	Size (in bits)	Direction	Description
fastest_pcie_clk	1	Input	Global clock which indicates the fastest clock at Transactor side. This clock is internally divided to generate lower speed (Gen1, Gen2, and so on.) clocks.
npor_out	1	Ouput	Reflects the status of the perst_n
mac_phy_pclk	1	Output	PIPE clock output from the transactor. Its frequency should be aligned with the PIPE width
Shared Signals			
device_type	4	Input	Allowed values for this signal are as follows: • 4 = ROOT PORT • 0 = ENDPOINT. If the DUT is dual function (can act as RC and EP), this pin can be forced, before any clock start, to the required mode of operation.
max_phy_rate	3	Input	Maximum rate at which the link can go up to. Allowed values for this signal are as follows: • 1 = Gen1 • 2 = Gen2 • 3 = Gen3 • 4 = Gen4 • 5 = Gen5 • 6 = Gen6
wake	1	Output	Used to wake-up from L2 Low Power state. This signal is applicable for EP DUT only.

Name	Size (in bits)	Direction	Description
clkreq_in_n	1	Input	Used by the controller to determine when to enter and exit L1 Sub-states when using the CLKREQ#-based mechanism. Refer Optional CLKREQ# Sideband Signals for more information.
ltssm_state	6	Output	LTSSM state machine encoding. Refer ltssm_state Encoding for the encoding information.
l1sub_state	4	Output	L1 substate encoding. Refer l1sub_state Encoding for the encoding information.
cfg_l1sub_en	1	Output	Indicates whether L1 substate is enabled or not. Based on this, CLKREQ# is driven.
clkcycle	64	Output	Provides fastest_pcie_clk counter information of Transactor
local_ref_clk_req_n	1	Output	Used to request entry to L1 sub-state. For EP transactor, it is also used to request reference clock removal.
mac_phy_powerdown	4	Output	Powers up or down the transceiver.
mac_phy_rxelecidle_disable	1	Output	Used for to control L1.2 state transition. As of now, its functionality is not present.
mac_phy_txcommonmode_disable	1	Output	Used for to control L1.2 state transition. As of now, its functionality is not present.

Feedback

Name	Size (in bits)	Direction	Description
mac_phy_asyncpowerchangeack	1	Output	Provides response to PhyStatus when power state changes and PCLK is removed. As of now, its functionality is not present.
mac_phy_width	2	Output	Controls the PIPE data path width.
mac_phy_pclk_rate	4	Output	Control the PIPE PCLK rate.
mac_phy_rate	3	Output	Control the link signaling rate.
Per Lane Signals			
lane_model_info	1	Input	Used to check the compatibility of Lane Model version with Transactor.
lane_model_ctrl	1	Output	Provides rate specific information used by Lane Model.
mac_phy_dirchange	1	Output	Indicates the PHY to perform Figure of Merit or Direction Change evaluation during equalization. • 0 = PHY performs Figure of Merit • 1 = PHY performs Direction Change This signal is left unconnected if PHY does not support it.
phy_mac_rxelecidle	1	Input	Indicates receiver detection of an Electrical Idle for each lane.

Feedback

Name	Size (in bits)	Direction	Description
phy_mac_phystatus	1	Input	Used to communicate the completion of several PHY functions including stable PCLK and Max PCLK after Reset# deassertion, PM state transitions, rate change, and receiver detection.
phy_mac_rxvalid	1	Input	Indicates symbol lock and valid data on RxData and RxDataK and further qualifies RxDataValid when used.
phy_mac_rxstatus	3	Input	Encodes receiver status and error codes for the received data stream when receiving data.
phy_mac_rxstandbystatus	1	Input	The PHY uses this signal to indicate its RxStandby state. • 0: Active • 1: Standby
mac_phy_txdatavalid	1	Output	This signal allows the MAC to instruct the PHY to ignore the data interface for one clock cycle.
mac_phy_txdetectrx_loopback	1	Output	Used to tell the PHY to begin a receiver detection operation.
mac_phy_txelecidle	4	Output	Forces Tx output to electrical idle when asserted except in loopback.
mac_phy_rxstandby	1	Output	Determine if the PHY Rx is active when the PHY is in P0 or P0s. - 0: Active, 1: Standby

Name	Size (in bits)	Direction	Description
phy_mac_messagebus	8	Input	The PHY multiplexes command, any required address, and any required data for sending read and write requests to access MAC PIPE registers and for sending read completion responses and write ack responses to MAC initiated requests.
mac_phy_messagebus	8	Output	The MAC multiplexes command, any required address, and any required data for sending read and write requests to access the PHY PIPE registers and for sending read completion responses and write ack responses to PHY initiated requests.

SERDES PIPE Only Signal List

The following table lists the signals for the SerDes PIPE Interface in transactor:

Name	Size (in bits)	Direction	Description
Per Lane Signals			
mac_phy_txdata_serdes	Gen5: 40 Gen6: 80	Output	Data/Control for the symbols of Tx data.
phy_mac_rxdata_serdes		Input	Parallel data input bus for Rx differential pair.

Connections

This section explains the following transactor connection with PHY DUT:

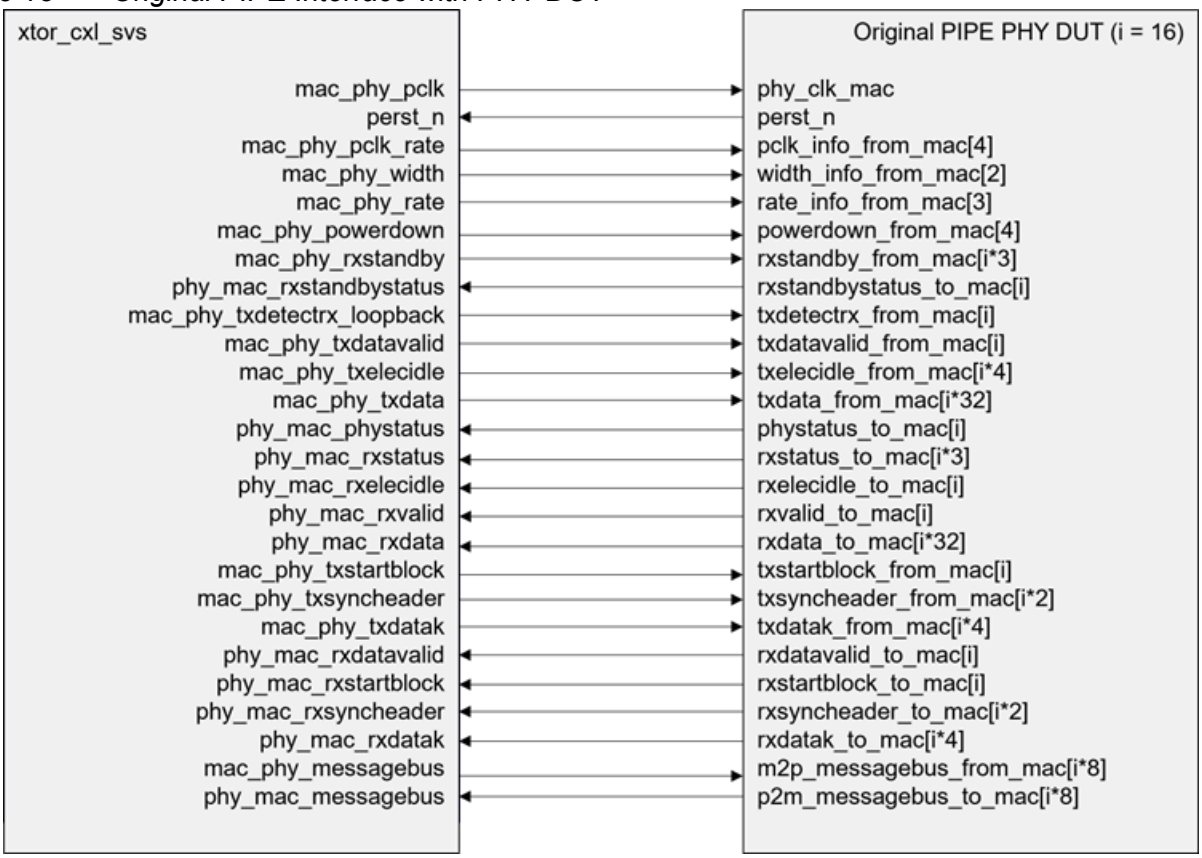
- **Original PIPE Interface with PHY DUT**
- **SerDes PIPE Interface with PHY DUT**

Feedback

Original PIPE Interface with PHY DUT

The following figure illustrates original PIPE PHY DUT to transactor connection:

Figure 13 Original PIPE Interface with PHY DUT

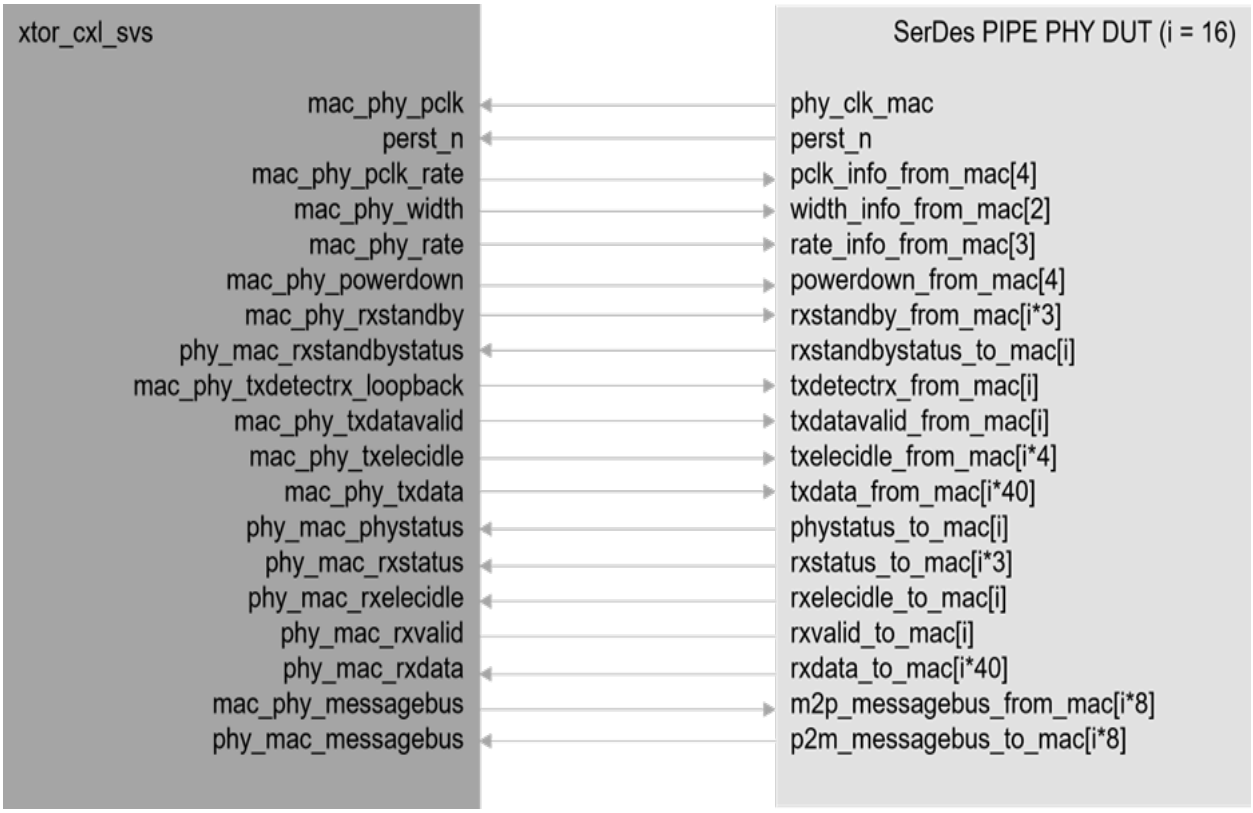


Feedback

SerDes PIPE Interface with PHY DUT

The following figure illustrates SerDes PIPE PHY DUT to transactor connection:

Figure 14 SerDes PIPE Interface with PHY DUT



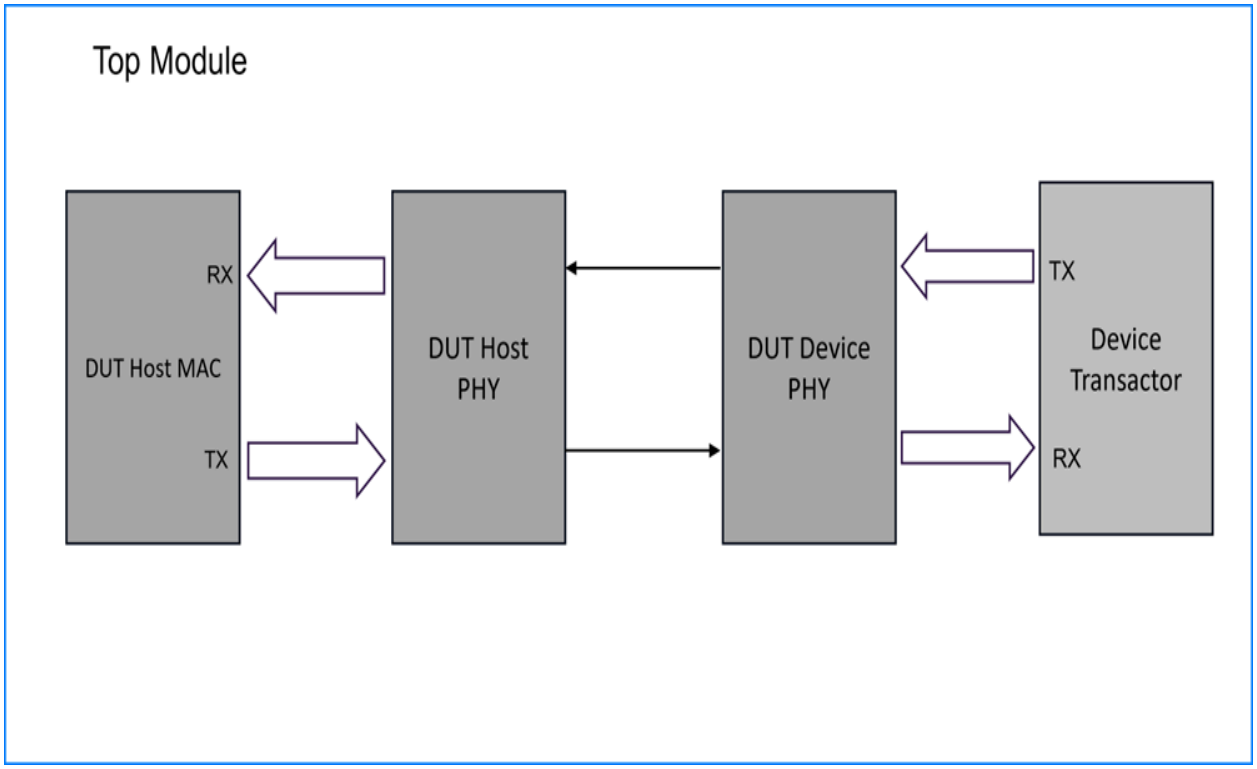
Feedback

Device Transactor and PHY DUT

For this topology, the xtor_cxl_svs module,device_type signal, is set to 0. Device transactor is connected to the PHY DUT with the following configuration:

- PIPE width of 32-bit (Original PIPE) / 40-bit (SerDes PIPE)
- x16 link width

Figure 15 Device Transactor and PHY DUT



To create a connection between the PHY DUT and the EP transactor, perform the following steps:

1. In the top-level wrapper module, instantiate the following mandatory components:
   - CXL Transactor as Device: xtor_cxl_svs
   - PHY DUT

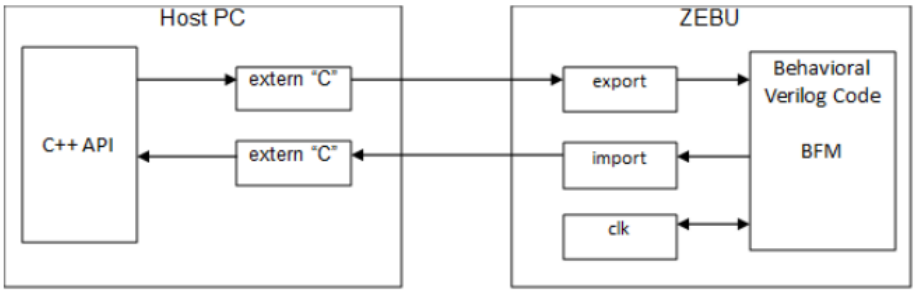
Feedback

2. Connect the xtor_cxl_svs interface parameters and signals (described in Transactor Interface Parameter and Transactor Interface Signal List) with PHY DUT according to the reference figures in Table 16 .
3. Connect zceiClockPort or RTL clock for Transactor as explained in Host Transactor and PHY DUT.

Initialization

The following figure illustrates the process of initializing a transactor:

Figure 16 Initializing a Transactor



Perform the following steps to initialize the transactor:

1. Include necessary CXL header files along with other ZeBu/zemi3 header files.

```
#include "xtor_pcie_svs_tlp.hh"

#include "xtor_cxl_svs.hh"

using namespace XTOR_PCIE SVS;

using namespace XTOR_CXL_SVS;
```

2. Declare and initialize CXL driver handles (xtor_cxl _svs) in the main method.

```
int main(int argc, char *argv[]){

xtor_cxl_svs* rx;

xtor_cxl_svs: Register();

ZEMI3Manager zemi3 = ZEMI3Manager::open(database.c_str());

zemi3->buildXtorList((database+"/xtor_dpi.Ist").c_str());

zemi3->init();
```

Feedback

```
rc= static_cast<xtor_cxl_svs*>
 (Xtor::getNewXtor(xtor_cxl_svs::getXtorTypeName(),[Mode], [Path]));
assert (rc!= NULL);

[Mode]: In emulation mode, it should be set as
 XtorRuntimeMode_t::XtorRuntimeZebuMode.

In simulation mode, it should be set as
 XtorRuntimeMode_t::XtorRuntimeSimulationMode.

[Path]: Verilog hierarchical path of the Transactor instance. It is a
 mandatory field in Simulation mode, optional in emulation and SEM.
```

3. Set the parameters to configure the transactor:

```
if (strncmp(xtor->getDriverInstanceName(),"wrapper.cxl_driver",21)==0)
 {
(xtor->setConfigParam("PCIE_TIMING", "1");
(xtor->setConfigParam("PCIE_TARGETSPEED", "5"));
(xtor->setConfigParam("PCIE_MAXPAYLOAD", "128"));
(xtor->setConfigParam("PCIE_NO_EQ_NEEDED_DISABLE", "true"));
(xtor->setConfigParam("PCIE_BAR0SIZE", (uint32_t)MASKBAR0));
(xtor->setConfigParam("PCIE_BAR0TYPE", (uint32_t)MEM32_BAR0));
(xtor->setConfigParam("PCIE_BAR1SIZE", (uint32_t)MASKBAR1));
(xtor->setConfigParam("PCIE_BAR1TYPE", (uint32_t)MEM32_BAR1));
(xtor->setConfigParam("CXL_FB_DSP_CACHE_ENABLE",       "true"));
(xtor->setConfigParam("CXL_FB_DSP_MEM_ENABLE",       "true"));
(xtor->setConfigParam("CXL_FB_DSP_CXL2P0_ENABLE",       "true"));
xtor->setLog ((char*) (xtor->getDriverInstanceName ()), true);
rc = xtor;
xtor_tb_rc = new cxl_xtor_tb(rc, tlp,0);
rc->setDebugLevel(DEBUG_LOW);
}
```

4. Register callback.

```
rc->register_callback(rc_receive_callback);
```

5. Start zemi3.

```
zemi3->start();
```

6. Wait for all the transactors to be initialized.

```
unit32_t exit_code=1
while (exit_code!=0){
if (!Xtor::AllXtorInitDone()){
*exit_code=1;
}
else
{
*exit_code=0
```

Feedback

```
}
}
```

Configuration

This section explains how to configure the transactor:

- Configuration Parameters
- APIs

Configuration Parameters

CXL transactor provides configuration parameters that are static and must be provided before starting the transactor initialization. These parameters are accessed using the setConfigParam() and getConfigParam() methods.

The following table lists the key configuration parameters for the CXL transactor:

Table 18 Configuration Parameters

Parameters	Default Value	Details
PCIE_TIMING	1	Specify one of the following values: • 0: Indicates real mode. In this mode, 1024 OS are exchanged during link training (as per spec) and LTSSM timeout values are actual values (2ms/ 12ms/ 24ms/ 48ms). • 1: Indicates fast link mode. In this mode, link up can be achieved through exchange of less number of OS and timers are also scaled down to 1024 factor (1ms downscale to 1024ns).
PCIE_TARGETSPEED	1	Controls the target speed at which you want to achieve the link up. It manipulates the Target Link Speed field of Link Control 2 register. This parameter is only applicable for RC Transactor. Valid values: 1, 2, 3, 4, 5, 6
PCIE_MAXPAYLOAD	4096	Configures the max payload capability for the device, which means, it configures the Max_Payload_Size Supported field of the Device Capabilities Register. Valid values: 128, 256, 512, 1024, 2048, 4096
PCIE_NO_EQ_NEEDED_DISABLE	true	Controls if the specified port is permitted to indicate that it requires equalization. Valid values: true/false

Feedback

Table 18 Configuration Parameters (Continued)

Parameters	Default Value	Details
PCIE_EQ_BYPASS_HIGH_RATE_DISABLE	false	Controls if the specified port is permitted to indicate support for equalization bypass to highest common link data. Valid values: true/false.
PCIE_EQ_PHASE23_ENABLE	false	Enables Recovery.Equalization Phase 2 and 3. Valid values: true/false.
PCIE_BAR<n>SIZE	0	Configures the size of Bar. For RC Transactor: n = 0, 1. For EP Transactor: n= 0, 1, 2, 3, 4, 5. Valid values: 0xff-0xffffffff. The value 0 is used for disabling the BAR.
PCIE_BAR<n>TYPE	0	Configures the type of BAR. For RC Transactor: n = 0, 1. For EP Transactor: n= 0, 1, 2, 3, 4, 5. Valid values: 0x0 - mem32 access, 0x1- IO access, 0x4 - mem64 access
PCIE_DEVICE_ID	0xABCD	Configures the Device ID register in the configuration space.
PCIE_VENDOR_ID	0x16C3	Configures the Vendor ID register in the configuration space.
PCIE_MAXLINKSPEED	6/5/4	Max link speed for the transactor. Valid values: 1, 2, 3, 4, 5, 6 for 2.5 GT/s, 5.0 GT/s, 8.0 GT/s, 16.0 GT/s, 32.0 GT/s, or 64.0 GT/s correspondingly. The default value is: • 6, if Gen6 license check is ok • 5, if Gen5 license check is ok • Otherwise, 4
PCIE_FLIT_MODE_ENA	false	Enables the FLIT Mode in transactor. Set the value of the parameter to true if PCIE_TARGETSPEED is 6. Valid values: true/false.
PCIE_TLP_BYPASS	false	When true: configuration requests from Host are routed to application interface rather than core. So send the completions for configuration requests from Testbench. Enables Testbench to override BAR settings. Valid for Device XTOR only. Valid values: true/false

Feedback

Table 18 Configuration Parameters (Continued)

Parameters	Default Value	Details
PCIE_SUBID	0	PCIe Subsystem ID[31:16] and Sbsystem Vendor ID[15:0] of the transactor. Only applicable for Type0 Config space i.e. EP
XTOR_NB_RESET_DE_ASSERTED	0	Number of reset event deasserted before starting the global scheduler.
PCIE_XTOR_CLOCK_CHECKER_DISABLE	false	Disables the clock checker logic which checks the phy_clk_dut_ratio w.r.t fastest_pcie_clk settings.
CXL_CLK_TICK_PERIOD	8	Valid values: true/false Configures the number of clk before callingCB registered with registerProgressCB.
CXL_FB_DSP_CACHE_ENABLE	false	Enable CXL Cache support.
CXL_FB_DSP_MEM_ENABLE	false	Enable CXL Mem support.
CXL_FB_DSP_CXL2P0_ENABLE	false	Enable CXL2.0 feature support.
CXL_FB_DSP_MLD_ENABLE	false	Enable MLD feature.
CXL_FB_RANGE1_MEMVALID	0	Configuring DVSEC Range1 Register fields.
CXL_MOD_TS_ALT_PROTOCOL_SUPPORT	true	Controls the APN capability. When Disabled on Device end, CXL host will link up in PCIe mode. Valid Values: true/false
CXL_FB_RANGE1_MEMACTIVE	0	Enables the CXL Range 1 memory to be fully initialized and available for use.
CXL_FB_RANGE1_MEDIATYPE	0	Enables Memory Media Characteristics Valid Values : 000 - Volatile memory (Reserved for CXL 2.0 and above) 001 - Non-volatile memory (Reserved for CXL 2.0 and above) 010 - The memory characteristics are communicated via CDAT all other settings are reserved.

Feedback

Table 18 Configuration Parameters (Continued)

Parameters	Default Value	Details
CXL_FB_RANGE 1_MEMCLASS	0	Enables the class of memory Valid Values : 000 – Memory Class (Reserved for CXL 2.0 and above) 001 – Storage Class (Reserved for CXL 2.0 and above) 010 - The memory characteristics are communicated via CDAT all other settings are reserved.
CXL_FB_RANGE 1_DESINTL	0	Enables the desired memory interleave. Valid Values : 00000 - No Interleave 00001 - 256 Byte Granularity 00010 - 4K Interleave 00011– 512 Bytes 00100 – 1024 Bytes 00101 – 2048 Bytes 00110 – 8192 Bytes 00111 – 16384 Bytes all other settings are reserved.
CXL_FB_RANGE 1_SIZELOW	0	Corresponds to bits 31:28 of CXL Range 1 memory size.
CXL_FB_RANGE 1_SIZEHIGH	0	Corresponds to bits 63:32 of CXL Range 1 memory size.
CXL_FB_RANGE 1_BASELOW	0	Corresponds to bits 31:28 of CXL Range 1 base in the host address space.
CXL_FB_RANGE 1_BASEHIGH	0	Corresponds to bits 63:32 of CXL Range 1 base in the host address space.
CXL_FB_RANGE 2_MEMVALID	0	Configuring DVSEC Range2 Register fields.
CXL_FB_RANGE 2_MEMACTIVE	0	Enables the CXL Range 2 memory to be fully initialized and available for software use.

Feedback

Table 18 Configuration Parameters (Continued)

Parameters	Default Value	Details
CXL_FB_RANGE2_MEDIATYPE	0	Enables Memory Media Characteristics. Valid Values : 000 - Volatile memory (Reserved for CXL 2.0 and above) 001 - Non-volatile memory (Reserved for CXL 2.0 and above) 010 - The memory characteristics are communicated via CDAT all other settings are reserved.
CXL_FB_RANGE2_MEMCLASS	0	Enables the class of memory Valid Values: 000 – Memory Class (Reserved for CXL 2.0 and above) 001 – Storage Class (Reserved for CXL 2.0 and above) 010 - The memory characteristics are communicated via CDAT all other settings are reserved
CXL_FB_RANGE2_DESINTL	0	Enables the desired memory interleave. Valid Values: 00000 - No Interleave 00001 - 256 Byte Granularity 00010 - 4K Interleave 00011– 512 Bytes 00100 – 1024 Bytes 00101 – 2048 Bytes 00110 – 8192 Bytes 00111 – 16384 Bytes all other settings are reserved.
CXL_FB_RANGE2_SIZELOW	0	Corresponds to bits 31:28 of CXL Range 1 memory size.
CXL_FB_RANGE2_SIZEHIGH	0	Corresponds to bits 63:32 of CXL Range 1 memory size.
CXL_FB_RANGE2_BASELOW	0	Corresponds to bits 31:28 of CXL Range 1 base in the host address space.
CXL_FB_RANGE2_BASEHIGH	0	Corresponds to bits 63:32 of CXL Range 1 base in the host address space.

Feedback

Note:
Other configuration parameters are available in the file $ZEBU_IP_ROOT/xtor_cxl_svs/doc/html/index.html.

The following example describes how to configure different equalization modes for transactor using the following configuration parameters:

- PCIE_EQ_PHASE23_ENABLE
- PCIE_NO_EQ_NEEDED_DISABLE
- PCIE_EQ_BYPASS_HIGH_RATE_DISABLE

Set the parameters using the following commands:

```
xtor->setConfigParam("PCIE_NO_EQ_NEEDED_DISABLE", "false");
xtor->setConfigParam("PCIE_EQ_BYPASS_HIGH_RATE_DISABLE", "true");
xtor->setConfigParam("PCIE_EQ_BYPASS_HIGH_RATE_DISABLE", "true")
```

The following table lists the various possible values of the PCIE_NO_EQ_NEEDED_DISABLE, PCIE_EQ_BYPASS_HIGH_RATE_DISABLE, and PCIE_EQ_PHASE23_ENABLE parameters and the final transactor result:

Table 19 Transactor Equalization Settings

PCIE_NO_EQ_NEEDED_DISABLE	PCIE_EQ_BYPASS_HIGH_RATE_DISABLE	PCIE_EQ_PHASE23_ENABLE	Final Equalization Mode	
			Gen 5	Gen 6
false	false	false	G1 -> G5 (NO EQ)	G1 -> G6 (NO EQ)
false	false	true	G1 -> G5 (NO EQ)	G1 -> G6 (NO EQ)
false	true	false	G1 -> G5 (NO EQ)	G1 -> G6 (NO EQ)
false	true	true	G1 -> G5 (NO EQ)	G1 -> G6 (NO EQ)
true	false	false	G1 -> G5 (EQ01)	G1 -> G6 (EQ01)
true	false	true	G1 -> G5 (EQ0123)	G1 -> G6 (EQ0123)
true	true	false	G1 -> G3 (EQ01) -> G4 (EQ01) -> G5 (EQ01)	G1 -> G3 (EQ01) -> G4 (EQ01) -> G5 ->G6(EQ01)

Feedback

Table 19 Transactor Equalization Settings (Continued)

PCIE_NO_EQ_NEEDED_DISABLE	PCIE_EQ_BYPASS_HIGH_RATE_DISABLE	PCIE_EQ_PHASE23_ENABLE	Final Equalization Mode	
			Gen 5	Gen 6
true	true	true	G1 -> G3 (EQ0123) -> G4 (EQ0123) -> G5 (EQ0123)	G1 -> G3 (EQ0123) -> G4 (EQ0123) -> G5(EQ0123) ->G6 (EQ0123)

APIs

The following table lists the key APIs for the CXL transactor:

Table 20 Transactor APIs

Name	Description
Configuration APIs	
restrictCxl3	API to restrict checking out CXL3 license in case CXL3 is not supported by device. It must be called before new constructor of the transactor .
wait_for	Blocking method to wait for a particular event to happen .
analyzerStart	Start CXL Protocol Analyzer.
analyzerStop	Stop CXL Protocol Analyzer.
reqFor	Perform operation based on driving req along with provided value.
register_callback	Registers user's testbench callback function for receiving incoming TLPs and notifications from transactor.
runClk	Waits for CXL fastest clock.
initBFM	Manually initialize CXL transactor when globalScheduler is not used.
getDeviceType	Retrieves Whether transactor is host or device.
speedChange	Initiate Link speed change with input as target speed.
cxlChannelInitTx	Request the initialization for the specified transmitting (mem / cache) channel.

Feedback

Table 20 Transactor APIs (Continued)

Name	Description
Configuration APIs	
cxlChannelDeinitTx	Request the de-initialization for the specified transmitting (mem / cache) channel.
cxlChannelInitRx	Request the initialization for the specified receiving (mem / cache) channel .
cxlChannelDeinitRx	Request the de-initialization for the specified receiving (mem / cache) channel.
getDriverInstanceName	Retrieves the XTOR instance name found in environment.It can be used in comparing the xtor instance name and provide the configurations according to instance mode.
Traffic APIs	
cxlCfgWrite	Backdoor write of transactor CXL Configuration Space. Refer Usage example below.
cxlCfgRead	Backdoor read of transactor CXL Configuration Space. Refer Usage example below.
send	Sends a TLP on the Link.
readCfg0	Frontdoor Configuration Read 0.
writeCfg0	Frontdoor Configuration Write 0 .
pushTxTlp	Sends a TLP on the link
get_D2H0_Req_credit	Return the number of available credits for D2H0 req channel.
get_D2H1_Req_credit	Return the number of available credits for D2H1 req channel .
get_D2H_Resp_credit	Return the number of available credits for D2H0 resp channel .
get_D2H_Data_credit	Return the number of available credits for D2H0 data channel .
get_H2D_Req_credit	Return the number of available credits for H2D req channel .
get_H2D0_Resp_credit	Return the number of available credits for H2D0 resp channel.
get_H2D1_Resp_credit	Return the number of available credits for H2D1 resp channel.
get_H2D_Data_credit	Return the number of available credits for H2D data channel.
get_M2S_Rwd_credit	Return the number of available credits for M2S rwd channel.

Feedback

Table 20 Transactor APIs (Continued)

Name	Description
Configuration APIs	
get_M2S_Req_credit	Return the number of available credits for M2S req channel.
get_S2M_Drs_credit	Return the number of available credits for S2M drs channel.
get_S2M_Ndr_credit	Return the number of available credits for S2M ndr channel.
grant_D2H0_Req_credit	Grant the number of credits for the D2H0 req channel.
grant_D2H1_Req_credit	Grant the number of credits for the D2H0 req channel.
grant_D2H_Resp_credit	Grant the number of credits for the D2H0 req channel.
grant_D2H_Data_credit	Grant the number of credits for the D2H0 data channel.
grant_H2D_Req_credit	Grant the number of credits for the H2D req channel.
grant_H2D0_Resp_credit	Grant the number of credits for the H2D0 resp channel.
grant_H2D1_Resp_credit	Grant the number of credits for the H2D1 resp channel.
grant_H2D_Data_credit	Grant the number of credits for the H2D data channel.
grant_M2S_Rwd_credit	Grant the number of credits for the M2S rwd channel.
grant_M2S_Req_credit	Grant the number of credits for the M2S req channel.
grant_S2M_Drs_credit	Grant the number of credits for the S2M drs channel.
grant_S2M_Ndr_credit	Grant the number of credits for the S2M ndr channel.
D2H0_Req	Transmit D2H req on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
D2H_Resp	Transmit D2H resp on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
D2H_Data	Transmit D2H data on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
H2D_Req	Transmit H2D req on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
H2D0_Resp	Transmit H2D rsp on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.

Feedback

Table 20 Transactor APIs (Continued)

Name	Description
Configuration APIs	
H2D_Data	Transmit H2D data on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
S2M_Ndr	Transmit S2M ndr on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
S2M_Drs	Transmit S2M drs on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
M2S_Req	Transmit M2S req on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
M2S_Rwd	Transmit M2S rwd on the link. Access is specified using packet objects or passed as a pointer to a 32 bits dword matching the packet structure.
setTxDly	Configures the insertion of delay in clock cycles between any TLP transmit packets (requests or response).

Note:

Other configuration APIs are available in the file*$ZEBU_IP_ROOT/xtor_cxl_svs/include/xtor_cxl_svs.hh.*

APIs like get_<request>_credits() returns CXL channel credit(channel between HW and SW) and not CXL Link credits(defined by Link Layer in Specification). The CXL channel interface credit is limited up to 8.

Enabling SRIS in the Transactor

By default, the Separate Reference Clocks with Independent Spread Spectrum Clocking (SRIS) feature is disabled in CXL transactor.

Enabling the SRIS feature affects the SKP insertion interval of the port to compensate for differences in frequencies between bit rates at two ends of a Link.

To enable the SRIS feature at run time, specify following just after all the transactors are initialized:

```
rc->reqFor(sris_en, true);
```

Table 21 SRIS Supported Values

SRIS feature (Enable / Disable)	SKP Ordered Set Intervals (8b/10b Encoding)	SKP Ordered Set Intervals (128b/130b Encoding)
false	Normal (between 1180 and 1538 Symbol times)	Normal (between 370 to 375 Blocks)
true	Short (less than 154 Symbol times)	Short (less than 38 Blocks)

Enabling Backdoor Access

Enable the backdoor access that is write or read to any config space register as shown in the following example:

```
ep->cxlCfgWrite(TYPEO_STATUSCOMMAND, 0x00000007);
cfg_reg_read_value ep->cxlCfgRead(TYPEO_BAR1);
cfg_reg_write_value = 0x40000000;
ep->cxlCfgWrite (TYPEO_BAR1, cfg_reg_write_value);
```

Sending CXL.mem Request

Initiate a CXL.mem request from Host transactor as shown in the following example:

```
xtor_cxl_m2s_req_t m2s_req;
m2s_req.op = m2sReq_Rd;
m2s_req.mf = m2sMf_NoOp;
m2s_req.mv=m2sMv_Invalid;
m2s_req.typ = m2sSnp_NoOp;
m2s_req.tag = 0xA;
m2s_req.ad = 0x1234;
xtor->M2S_Req(m2s_req);
```

Sending CXL.cache Request

```
xtor_cxl_h2d_req_ t req1;
req1.op = h2dReq_SnpData;
req1.ad = 0x40;
req1.uqid = 0x5;
xtor->H2D_Req(req1);
```

Sending H2D Response and H2D Data

```
void rc_receive_callback (xtor_cxl_svs_event_t evt_type, void* pkt, void*
 xtor)
{
if(evt_type == XTOR_CXL_SVS::cacheD2HReq_cb}{\ // Retrieve the D2H
 Request
```

Feedback

```
req = &((xtor_cxl_svs_D2HReq_cb_param*)(pkt))->req:) // user can access
 req->op, req->ad, req->cqid etc
xtor_cxl_h2d_resp_t resp;
bool send_h2d_data = false;
if(d2h_req->op == d2hReq_RdOwn){ resp.op = h2dResp_Go;
resp.rsp_data = h2dCache_Exclusive; send_h2d_data = true;
}
if(d2h_req->op == d2hReq_CleanEvict){ resp.op = h2dResp_GO_WritePull;
 resp.rsp_data = req->cqid;
send_h2d_data = false;
}
resp.rsp_pre h2dResp_HCH;
resp.cqid = req->cqid;
xtor->H2D0_Resp(resp);
// Send the H2D data based on D2H Opcode received if(send_h2d_data =
 true){
xtor_cxl_h2d_data_t h2d_data;
h2d_data.cqid = req->cqid;
h2d_data.chunk_valid = 1;
h2d_data.poison = 0;
h2d_data.GoErr = 0; for(int i=0; i<16; i++)
h2d_data.data[i]= <data value>; xtor->H2D_Data(h2d_data);
```

Running the Transactor Example

To run the transactor perform the following steps:

1. Use the following command to compile the RTL:

   ```
   zCui -u <UTF_project>.utf -w<work_directory> -n -c
   ```

   where, <UTF_project>.utf file provides all inputs needed by the ZeBu compiler with UC.

   The default value for <work_directory> is zCui.work.

2. Use the following command on the emulator machine to run a test scenario:

   ```
   zRci[<UCLI script> --zebu-work <zebu.work>]
   ```

   where, <UCLI script> includes the zRci UCLI commands to control transactors and debug them. It also includes zemi3 commands.

3. Use the below set of commands to run any topology example:

   - To compile the example, the <compile_target> can be used as follows:

     ```
     make compile TG_COMP=<compile_target>
     ```

   - Use the run target to run the example testbench, as shown below:

     ```
     make run TG_RUN=<run_testbench>
     ```

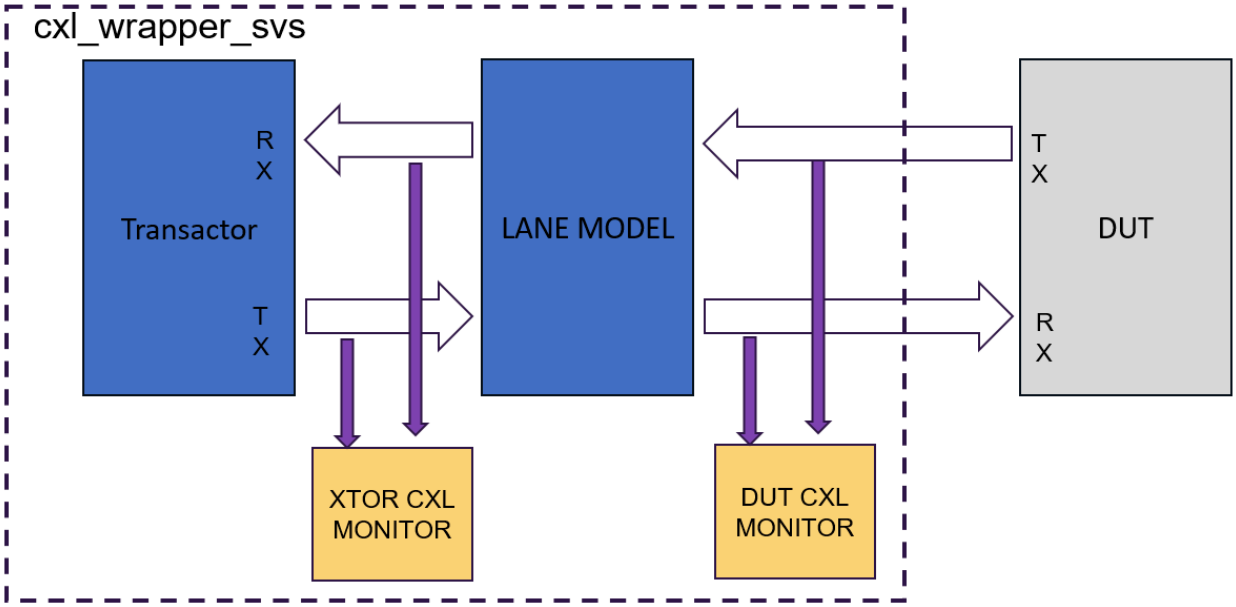
5

CXL Monitor

The CXL Monitor analyzes the physical channel traffic and reports it in the form of logs and PA FSDB (graphical packet format) to understand the PIPE traffic. The CXL Monitor is available in the monitor_cxl_svs package. It is a passive monitor instantiated (default) inside a cxl_wrapper_svs wrapper module. The monitor picks the data from bus and processes it based on the data and display information of OS,LTSSM states, CXL CACHE , CXL MEM , CXL IO TLP, CXL IO DLLP, Credits, FLIT INFO.

CXL Monitor uses the Verdi Protocol Analyzer, which helps in the generation of PA FSDB.

Figure 17 CXL Monitor



This section explains the following topics:

- Features
- Limitations
- FlexLM License Requirement

Feedback

- CXL Monitor Integration
- Using the CXL monitor

Features

- Protocol Analyzer taps the PIPE interface and reports packet based or symbol based information
- Identifies the CXL IO flits (TLPs, DLLPs with CXL IO flit)
- Identifies CXL CACHE MEM flits
- Identifies the LTSSM state and rate of link operation
- Displays packets in both Tx and Rx directions with packet start and end times
- Works for x16 and lower possible lane configuration(x8, x4..)
- Supports Gen1, Gen2, Gen3, Gen4, Gen5, and Gen6
- Supports multiple start and stop during the emulation run
- Supports logging of CXL.io/CacheMem traffic for CXL3.0 in textfile and Protocol Analyzer
- Supports logging of CXL.io/CacheMem traffic for CXL2.0 in textfile
- Supports CXL.io traffic for CXL2.0 in the Protocol Analyzer

Limitations

Currently does not support CXL.Cache and CXL.Mem traffic for CXL2.0 in the Protocol Analyzer.

Multiple analyzer_start/stop is not supported for CXL.io protocol for CXL3.0

ZDPI Monitor does not support online mode.

FlexLM License Requirement

You need the hw_xtormm_monitors FLEXnet license to use the CXL monitor.

CXL Monitor Integration

The CXL Monitor is available in the monitor_cxl_svs<num>.sh package.

Feedback

Perform the following steps to enable the CXL monitor in environment:

1. Install the monitor_cxl_svs<num>.sh package.
2. For connecting CXL monitor, refer to the $ZEBU_IP_ROOT/xtor_cxl_svs/misc/cxl_wrapper_svs.sv file.

   In CXL Wrapper, the following two instances of monitor module, monitor_cxl_svs, are created:

   - monitor_top_inst_dut: Monitor instance on DUT-Lane model PIPE interface.
   - monitor_top_inst_xtor: Monitor instance on XTOR-Lane model PIPE interface.

   The cxl_wrapper_svs module contains the following parameters:

Table 22 CXL Wrapper Parameters

Parameter	Default Value	Remarks
IMPLEMENT_MONITOR_DUT	1	Enables DUT CXL monitor on DUT-Lane model interface.
IMPLEMENT_MONITOR_XTOR	0	Enables XTOR CXL monitor on XTOR-Lane model interface.

While instantiating the cxl_wrapper_svs wrapper module in the environment, set the above parameters as per the requirement.

Note:
Only one parameter can be set at a time.

Using the CXL monitor

The CXL Monitor operates in two modes, ZEMI3 and ZDPI. To use the CXL monitor in ZEMI3/ ZDPI mode, the following is required:

- Configuring the Hardware
- Configuring the Software (Testbench)
- Starting/Stopping the CXL Monitor
- Generating the Log File and the FSDB File
- Customizing Data Log

Feedback

Configuring the Hardware

Perform the following updates to the hardware:

1. Instantiate the monitor parallel to the device to which you want to attach the monitor.
2. Select one of the following modes of operation:
   a. **ZEMI3 Mode**
      i. Specify the following on the VCS command line:

```
+define+ZEMI3_CXL_ANALYZER
```

      ii. Specify the following in the UTF command list:

```
zemi3 -timestamp true
```

   b. **ZDPI Mode**
      i. Specify the following on the VCS command line:

```
+define +ZDPI_CXL_ANALYZER
```

      ii. Specify the following in the UTF command list:

```
dpi_synthesis -enable ALL
zforce -rtlname
<monitor_intsance_path>.xtor_cxl_svs_monitor.en analyzer
```

      The following is the example of <monitor_instance_path>:

```
wrapper.cxl_wrapper_svs.genblk1.monitor_top_inst_xtor.xtor_cxl_sv
s_monitor.en_analyzer
```

Configuring the Software (Testbench)

Perform the following updates to the software to use CXL Monitor:

1. Include necessary CXL monitor header files in the testbench:

```
#include "monitor_cxl_svs.hh"
using namespace MONITOR_CXL_SVS;
```

2. Declare and initialize CXL monitor handles (monitor_cxl_svs) in the main method as shown below:

```
int main(int argc, char * argv[]) {
string mondriverList[1];
monitor_cxl_svs* cxl_monitor = NULL;
std::string monitor_type(MONITOR_TYPE);
```

Feedback

```
std::string monitor_side(MONITOR_SIDE);
monitor_cxl_svs::set_mon_mode(monitor_type);
monitor_cxl_svs::Register();
if (monitor_side == "DUT")
mondriverList[0] =
{"<monitor_instance_path>.xtor_cxl_svs_monitor"};
//E.g., <monitor_instance_path>=
 wrapper.cxl_wrapper_svs.genblk0.monitor_top_inst_dut
else
mondriverList[0]=
{"<monitor_instance_path>.xtor_cxl_svs_monitor"};
//E.g.,
 <monitor_instance_path>=wrapper.cxl_wrapper_svs.genblk1.monitor_top_i
nst_xtor
//In case of emulation
if (monitor_type == "ZEMI3")
cxl_monitor =
 static_cast<monitor_cxl_svs*>(Xtor::getNewXtor(monitor_cxl_svs::getXt
orTypeName(),XtorRuntimeZebuMode));
else
cxl_monitor =
 static_cast<monitor_cxl_svs*>(Xtor::getNewXtor(monitor_cxl_svs::getXt
orTypeName(),XtorRuntimeZebuMode,(mondriverList[0]).c_str()));
//In case of simulation for ZEMI3 monitor
cxl_monitor =
 static_cast<monitor_cxl_svs*>(Xtor::getNewXtor(monitor_cxl_svs::getXt
orTypeName(),XtorRuntimeSimulationMode,(mondriverList[0]).c_str()));
```

The above code snippet shows the example to create and initialize one instance of CXL Monitor. In the similar manner, declare another instance of CXL Monitor if required and add it to the array "`mondriverList`" for initialization.

3. Invoke the monitor APIs *analyzerStart()* and *analyzerStop()* to start and stop the analyzer, respectively. The usage of these APIs is described in Starting/Stopping the CXL Monitor.

Use the Linker flag -lmonitor_cxl_svs while running the testbench.

In case of ZEMI3 mode, binary report file (postproc_dumpfile0) is generated and in case of ZDPI mode, ztdb file (zdpi.ztdb) is generated. Also, the following is the zRci equivalent command to CCall:

```
ccall -dump_offline zdpi.ztdb ccall -enable_offline
force en_analyzer monitor internal signal - deposit
..
..
ccall - disable
```

Feedback

Refer to the files below, which include the software modifications needed to use CXL monitor and are located in the $ZEBU_IP_ROOT/monitor_cxl_svs/example/src/bench/ directory:

- cxl3p0_testbench.cc
- cxl2p0_testbench.cc

Starting/Stopping the CXL Monitor

The start/stop feature of CXL monitor allows you to start or stop the monitor multiple times for capturing the traffic. This improves debugging capability. To enable this feature, perform the following steps:

1. Invoke start/stop call of APIs from the testcase after CXL monitor handle is initialized, as shown below:

```
<monitor_handle>.analyzerStart();
.
<Traffic exchange>
.
<monitor_handle>.analyzerStop();
```

For an example of how to use the analyzerStart() and analyzerStop() APIs, refer to the file cxl3p0_testbench.cc, located in the $ZEBU_IP_ROOT/xtor_cxl_svs/example/src/bench/ directory.

Multi-start/stopcall of APIs can be invoked, as shown below:

```
<monitor_handle>.analyzerStart();
.
<Traffic exchange>
.
<monitor_handle>.analyzerStop();
.
.
<monitor_handle>.analyzerStart();
.
<Traffic Exchange>
.
<monitor_handle>.analyzerStop();
```

In ZEMI3 mode, each analyzerStart() -analyzerStop() pair generates separate binary dump file, which is identifiable by the number at last of file name, as shown in the following example:

```
*. postproc_dumpfile0, *. postproc_dumpfile1
```

Here, 0 or 1 corresponds to order of invoking analyzerStart -analyzerStop pair.

In ZDPI mode, the ztdb file zdpi.ztdb is appended with the traffic that is captured from each pair of analyzerStart - analyzerStop.

2. Generate the corresponding symbol log and PA FSDB for corresponding dump files.

   Post-process these Binary dump files individually to get post-processed log and FSDB dump for Post analyzer. (See Generating the Log File and the FSDB File)

Generating the Log File and the FSDB File

Perform the following steps to generate the log file and FSDB file for ZEMI3 or ZDPI Monitor:

- ZDPI Monitor
- ZEMI3 Monitor
- Convert Ztdb to Binary Dump
- Convert Binary File to Log and PA FSDB (Post-processing)
- Log File

ZDPI Monitor

ZDPI monitor only works in emulation mode and offline mode. Perform the following steps to generate the log and FSDB files for the CXL ZDPI Monitor:

1. Compile the testcase with makefile switches IMPLEMENT_MONITOR=1, MONITOR_TYPE=ZDPI and MONITOR_SIDE=XTOR/DUT (either XTOR or DUT)

   For example:

   ```
   make compile TG_COMP=16x80b TG_RUN=cxl3p0_testbenchIMPLEMENT_MONITOR=1
    MONITOR_TYPE=ZDPI MONITOR_SIDE=DUT
   ```

2. Runthe testcase with makefile switches IMPLEMENT_MONITOR=1, MONITOR_TYPE=ZDPI and MONITOR_SIDE=XTOR/DUT (same as used in “make compile” command)

   For example:

   ```
   make run TG_COMP=16x80b TG_RUN=cxl3p0_testbench IMPLEMENT_MONITOR=1
    MONITOR_TYPE=ZDPI MONITOR_SIDE=DUT
   ```

   Note:

   All parameters will be picked by default from makefile if not provided in command.

3. Convert Ztdb to Binary Dump
4. Convert Binary File to Log and PA FSDB (Post-processing)

ZEMI3 Monitor

Perform the following steps to generate the log and FSDB files for the CXL ZEMI3 monitor:

1. Compile the testcase with makefile switches IMPLEMENT_MONITOR=1, MONITOR_TYPE=ZDPI and MONITOR_SIDE=XTOR/DUT (either XTOR or DUT)

   For Example:

   ```
   make compile TG_COMP=16x80b TG_RUN=cxl3p0_testbench
    IMPLEMENT_MONITOR=1 MONITOR_TYPE=ZEMI3 MONITOR_SIDE=DUT
   ```

2. Runthe testcase with makefile switches IMPLEMENT_MONITOR=1, MONITOR_TYPE=ZEMI3 ZDPI and MONITOR_SIDE=XTOR/DUT (same as used in “make compile” command)
3. For Example:

   ```
   make run TG_COMP=16x80b TG_RUN=cxl3p0_testbench IMPLEMENT_MONITOR=1
    MONITOR_TYPE=ZEMI3 MONITOR_SIDE=DUT
   ```

   Note:

   All parameters will be picked by default from makefile if not provided in command.

4. Generating the Log File and the FSDB File

Convert Ztdb to Binary Dump

To generate the binary dump file in case of ZDPI monitor, specify the following command:

```
zdpiReport -z <zcui dir> -f <import_file> -| <path to generated
 post_procdumpfile_0 ztdb> -synchronize -|<zebu xtor shared object path>
-| <monitor shared object path>
```

For example:

```
zdpiReport -z 16x32b.zebu.work/zebu.work/ -f import_list
 -i rundir_cxl3p0_testbench.zebu/post_procdumpfile_0.ztdb
  -synchronize  -l $ZEBU_IP_ROOT/lib/libZebuXtor.so -l
 $ZEBU_IP_ROOT/lib/libmonitor_pcie_svs.so
```

In the above example:

- Use the synchronize switch for performing the process Import as per RTL invocation. Without this switch, the process import is random, which may cause loss of traffic.
- Create a file, import_list and enter " I_SendPIPEData_Cxl " and pass it with -f switch in the preceding command.
- In the ZDPI mode for multi-instance, pass scope in import list as :

```
scope <mon_inst_path>
```

Convert Binary File to Log and PA FSDB (Post-processing)

Binary file will be created after running the testcase with IMPLEMENT_MONITOR=1 switch in <rundir>

For Example:

```
rundir_cxl3p0_testbench.simu/wrapper.cxl_wrapper_svs.genblk0.monitor_top_
inst_dut.xtor_cxl_svs_monitor.postproc_dumpfile()
```

Once we have generated the binary dump file, to convert binary file to log and PA FSDB, specify the following command:

Use same parameters as used in “make compile” and “make run” commands

```
make generate_monitor_PA TG_RUN=<TG_RUN value> MONITOR_TYPE=<ZEMI3/ZDPI>
 MONITOR_SIDE=<XTOR/DUT> IMPLEMENT_MONITOR=1
```

To run post processing with custom dumpfile path

```
make generate_monitor_PA IMPLEMENT_MONITOR=1 USR_BINDMP=1
 PA_DUMPFILE_PATH=<absolute path to binary dump>
```

It will create default rundir provided in makefile and store results in it.

Switch	Description	Default value	Possible values
MONITOR_TYPE	Monitor type can be ZEMI3 or ZDPI	ZEMI3	ZEMI3,ZDPI
MONITOR_SIDE	Monitor side can be: DUT - if monitor is connected at DUT-Lane model interface. XTOR – if monitor is connected at XTOR-Lane model interface	XTOR	XTOR,DUT

Feedback

To run the deliverable example in simulation mode (only valid for ZEMI3 monitor), add SIMULATOR=1 in the compile, run and post processing commands.

Note:

All the switches must be same as provided in the "make run" command otherwise, "make generate_monitor_PA" will result in error.

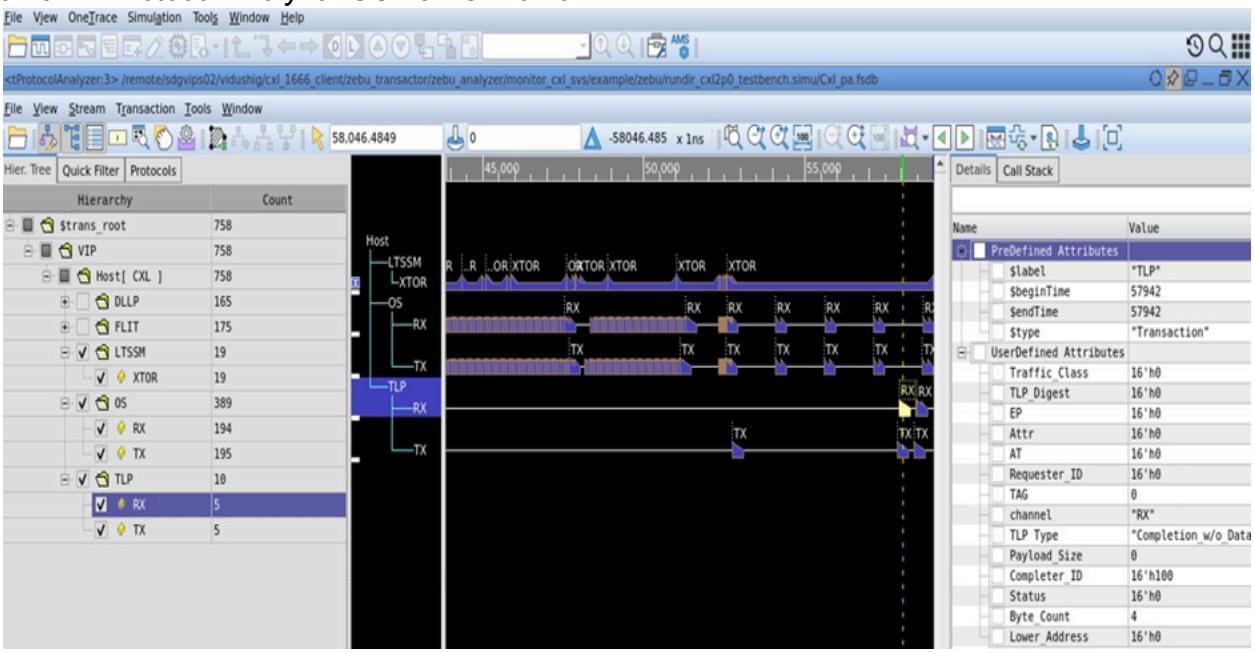
Open the FSDB file created by post-processing in Verdi, use the following command:

```
verdi -ssf <rundir>/Cxl_pa.fsdb -workMode protocolDebug
```

where, Cxl_pa.fsdb is the FSDB file created.

Select the packet from hierarchy table and click on any purple packet to display information.

Figure 18 Protocol Analyzer GUI for IO Traffic



Feedback

Figure 19 Protocol Analyzer GUI for CXL 3.0 CacheMem Traffic

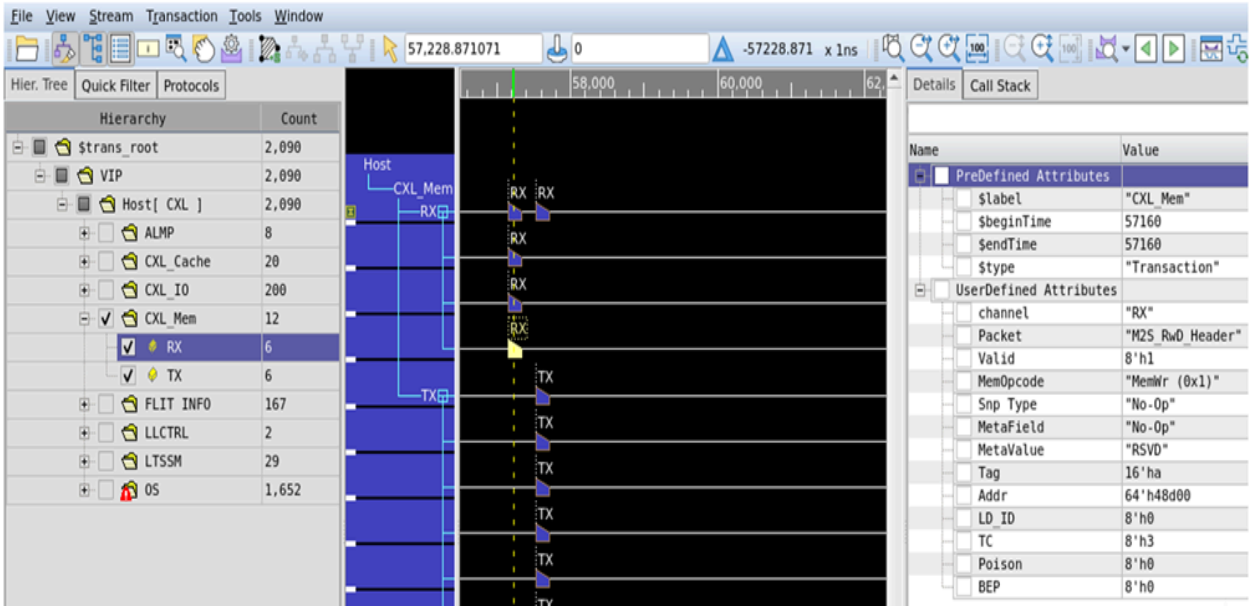
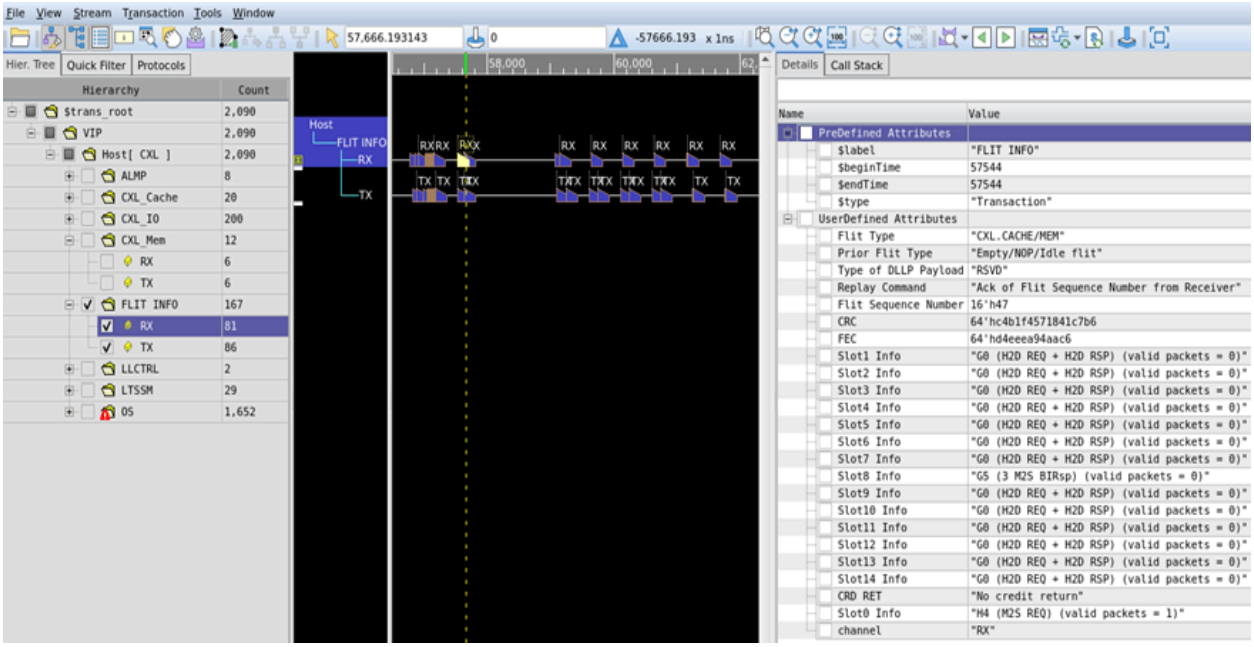


Figure 20 Protocol Analyzer GUI for CXL 3.0 Flit information



Log File

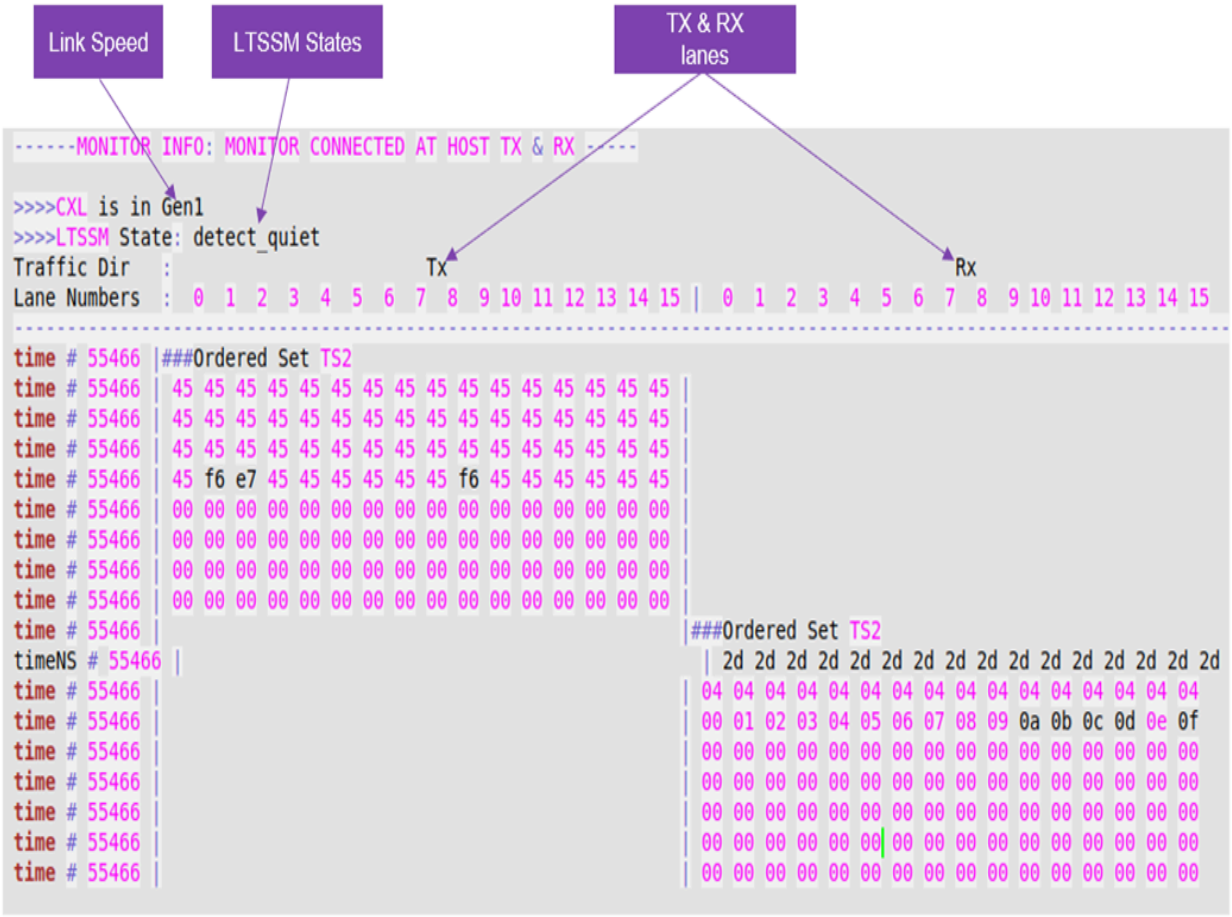
The text file with the unscrambled PIPE data is generated in the run directory/monitor. The log file can be generated based on traffic or based on lane type.

Feedback

Figure 21 Traffic Based Log File

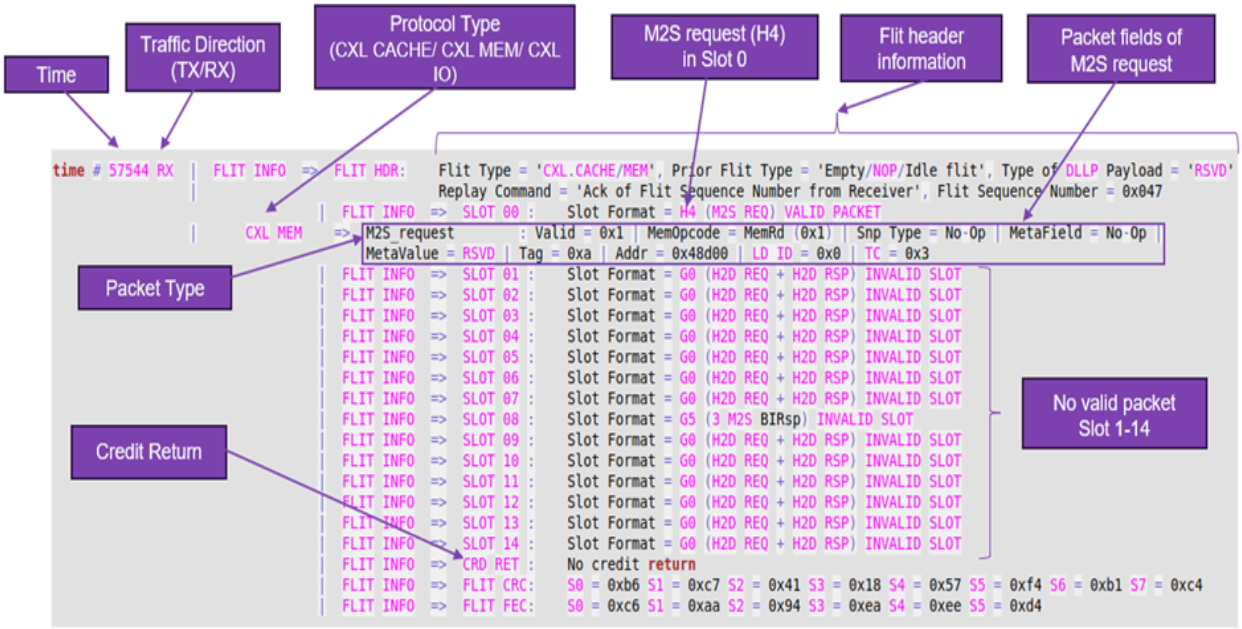
Feedback

Figure 22 Lane Based Log File



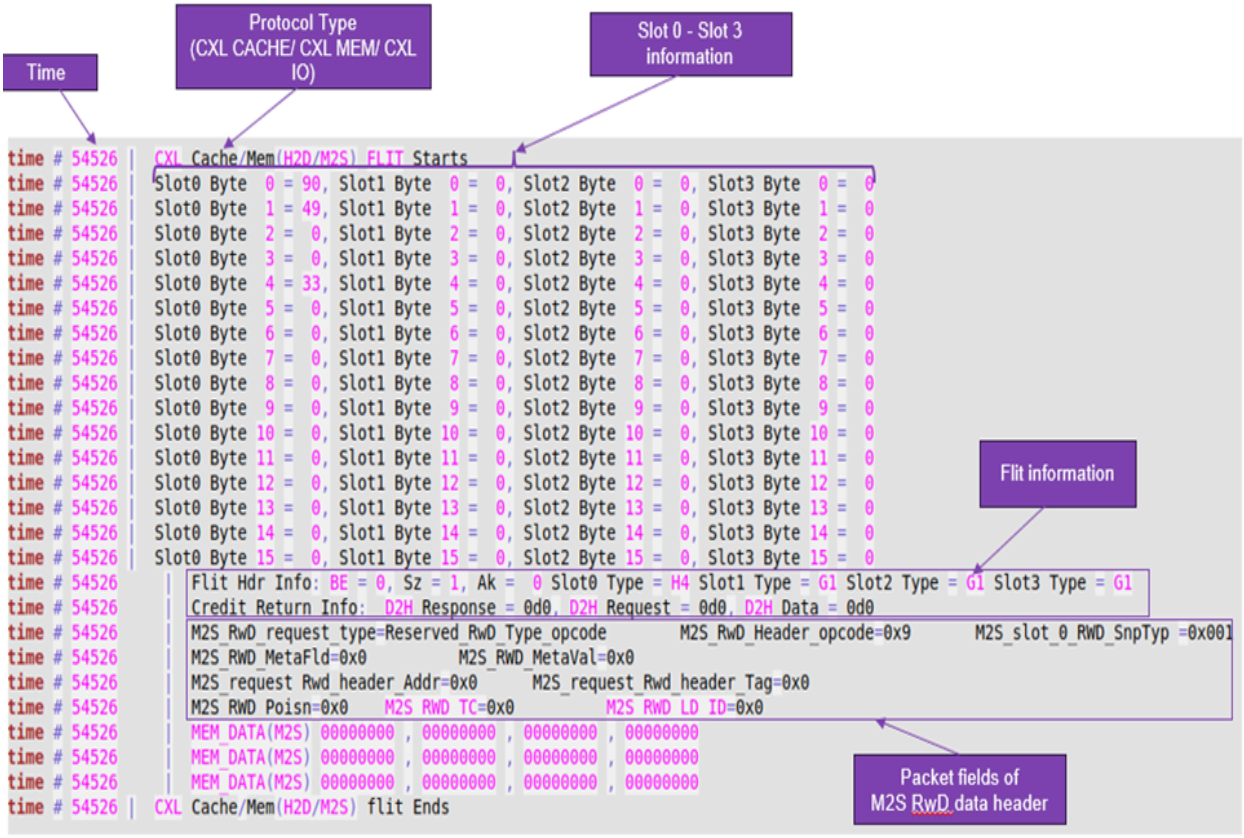
Feedback

Figure 23 Flit Information for CXL.Cache/Mem 256B Flit



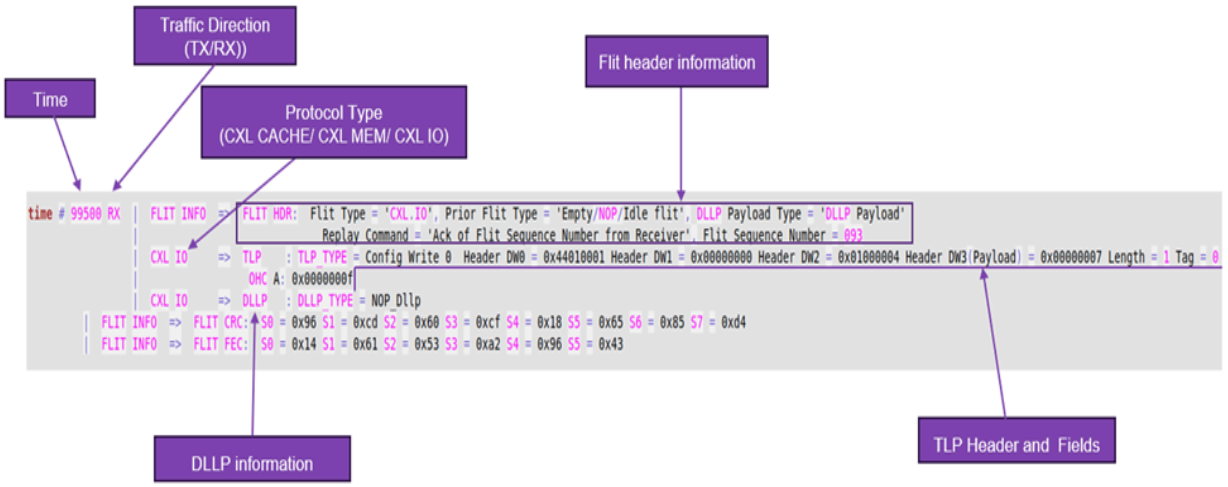
Feedback

Figure 24 Flit information for CXL.Cache/Mem 68B Flit



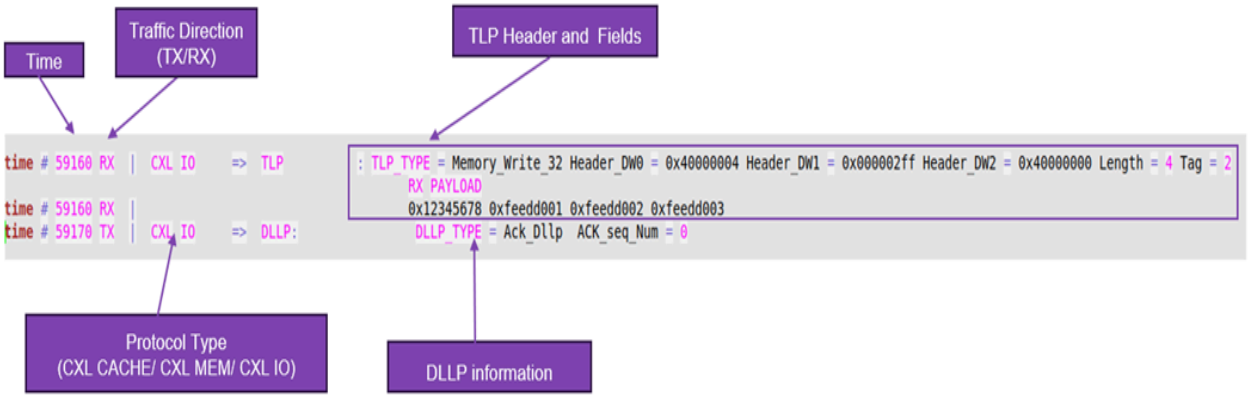
Traffic in the above figure is transmitted from Host(TX) and received at Device(RX)

Figure 25 Flit Information for CXL.IO 256B Flit



Feedback

Figure 26 Flit Information for CXL.IO 68B Flit

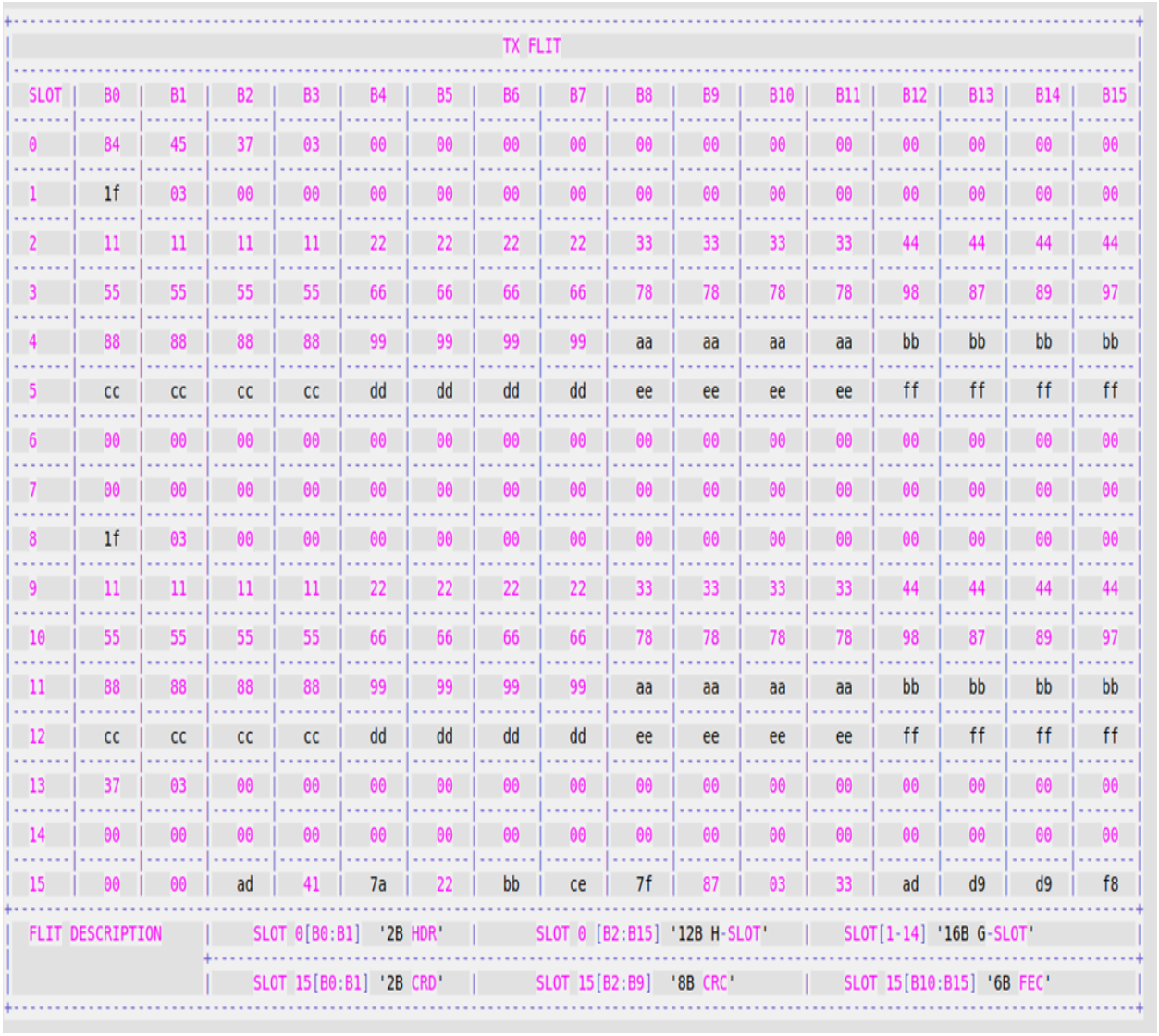


Flit info provides full flit info such as flit header info, every slot type, if data in slot valid or not, along with packet information.

Traffic in the above figure is transmitted from Host(TX) and received at Device(RX)

Feedback

Figure 27 CXL 3.0 256B flit display



TX FLIT

SLOT	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
0	84	45	37	03	00	00	00	00	00	00	00	00	00	00	00	00
1	1f	03	00	00	00	00	00	00	00	00	00	00	00	00	00	00
2	11	11	11	11	22	22	22	22	33	33	33	33	44	44	44	44
3	55	55	55	55	66	66	66	66	78	78	78	78	98	87	89	97
4	88	88	88	88	99	99	99	99	aa	aa	aa	aa	bb	bb	bb	bb
5	cc	cc	cc	cc	dd	dd	dd	dd	ee	ee	ee	ee	ff	ff	ff	ff
6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
8	1f	03	00	00	00	00	00	00	00	00	00	00	00	00	00	00
9	11	11	11	11	22	22	22	22	33	33	33	33	44	44	44	44
10	55	55	55	55	66	66	66	66	78	78	78	78	98	87	89	97
11	88	88	88	88	99	99	99	99	aa	aa	aa	aa	bb	bb	bb	bb
12	cc	cc	cc	cc	dd	dd	dd	dd	ee	ee	ee	ee	ff	ff	ff	ff
13	37	03	00	00	00	00	00	00	00	00	00	00	00	00	00	00
14	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
15	00	00	ad	41	7a	22	bb	ce	7f	87	03	33	ad	d9	d9	f8

FLIT DESCRIPTION | SLOT 0[B0:B1] '2B HDR' | SLOT 0 [B2:B15] '12B H-SLOT' | SLOT[1-14] '16B G-SLOT'
| SLOT 15[B0:B1] '2B CRD' | SLOT 15[B2:B9] '8B CRC' | SLOT 15[B10:B15] '6B FEC'

The above figure depicts the byte level information of 256B flit for CXL 3.0

Customizing Data Log

Customize the data log for the following:

- Mask printing of Order sets, DLLP, TLP, Cache/Mem packets,Flit information.
- Choose between lanes-based (Raw data on lanes) or traffic-based printing (Traffic is logged with more verbosity).

Feedback

You can customize the data logs using the following parameters:

Table 23 Data Log Parameters

Parameter	Default Value	Description	Applicable CXL Version
CXL_VS_MONITOR_PA_OS_FSDB_EN	true	Prints ordered sets if true	CXL 2.0,CXL 3.0
CXL_VS_MONITOR_PA_IO_EN	true	Prints all CXL IO traffic if true	CXL 3.0
CXL_VS_MONITOR_PA_TLP_FSDB_EN	true	Prints CXL IO TLPs if true	CXL 2.0,CXL 3.0
CXL_VS_MONITOR_PA_DLLP_FSDB_EN	true	Prints CXL IO DLLPs if true	CXL 2.0,CXL 3.0
CXL_VS_MONITOR_PA_CACHE_MEM_EN	true	Prints all CXL Cache/Mem traffic if true	CXL 2.0, CXL 3.0
CXL_VS_MONITOR_PA_CREDIT_INFO_EN	false	Prints all Credit info if true	CXL 2.0, CXL 3.0
CXL_VS_MONITOR_PA_FLIT_INFO_EN	false	Prints Full FLIT Info if true	CXL 2.0, CXL 3.0
CXL_VS_MONITOR_PA_FLIT_DISPLAY_ON	false	Prints FLIT in Tabular format if true	CXL 3.0
CXL_VS_MONITOR_PA_IDLE_NOP_FLIT_FILTER	false	Filters IDLE flits printing when Flit display is on.	CXL 3.0
CXL_VS_MONITOR_PA_PKT_LOG_EN	true	Prints High Abstraction Data if true else Raw Lane wise data	CXL 2.0, CXL 3.0
CXL_VS_MONITOR_PA_CACHE_FSDB_EN	true	Logs CXL Cache traffic if true	CXL 3.0
CXL_VS_MONITOR_PA_MEM_FSDB_EN	true	Logs all CXL Mem traffic if true	CXL 3.0

Feedback

Table 23 Data Log Parameters (Continued)

Parameter	Default Value	Description	Applicable CXL Version
CXL_VS_MONITOR_PA_FSDB_DUMP_EN	true	Generates .fsdb file to view in Protocol Analyzer if true	CXL 2.0,CXL 3.0
CXL_VS_MONITOR_PA_VERBOSE_EN	true	Prints high verbose data if true	CXL 3.0

Specify these parameters in the testbench_post_PA.cc test case, that is used for postprocessing as shown below:

```
xtor_cxl_mon_pktprntCfg_flag_t set_flg;
set_flg.CXL_VS_MONITOR_PA_VERBOSE_EN =true;


monitor_cxl_svs::doCxlPostProcessing(PA_DUMPFILE.c_str(),mon_file,set_flg
,connected_to_dev_sub);
```

By default, traffic direction in log and protocol analyzer is in accordance with Host, if you want it to be aligned with the Device, set the flag "connected_to_dev_sub" in testbench_post_PA.cc as shown below:

```
connected_to_dev_sub = true;
```

6

Troubleshooting

This section highlights common errors and frequently asked questions, along with the list of signals, that can be used to debug issues related to linking, enumeration and IO/Cache/Mem transactions as described in following sections:

Integration or Transactor Bring-Up

The following are the common checks during the transactor integration or bring-up:

Scope Error Due to Incorrect ZEBU_IP_ROOT

The following error is reported due to incorrect usage of the ZEBU_IP_ROOT during the compile and runtime.

To fix the issue, set the value of the ZEBU_IP_ROOT correctly.

```
svGetScopeFromName :
 Error:scope.xtor_cxl_svs.SNPS_PCIE_TX.SNPS_PCIE_TX_MEM' is unknown.
```

Missing Define

The following error message is displayed in case of a missing define:

```
Error-[SM_EALT] Level and edge triggered signals in same sensitivity list
<$ZEBU_IP_ROOT>/vlog/vcs/monitor_cxl_svs.sv, 3142
<Message details suppressed due to encrypted design unit>
```

To fix the error, check if the+define+XTOR_CXL_SERDES_SVS_SYNTHESIS is specified during HW compilation.

Incorrect Initialization

The following errors are reported when a CXL object is not created but is initialized at time 0. The hardware part becomes active as soon as clk starts.

```
### fatal error in ZEMI3 RUN [ZXTOR0300F] : Imported function
"ZEBU_VS_SNPS_UDPI__rst_assert" not found
terminate called after throwing an instance of 'zer_Exception'
what(): Imported function "ZEBU_VS_SNPS_UDPI__rst_assert" not found

Internal Error: Unknown scope 0x555958fb62b0 for zebu_vs_udpi_rst_and_clk
import ZEBU_VS_SNPS_UDPI__BuildSerialNumber.
```

To fix these errors, ensure that the SW part is also active by that time. That is, create and initialize CXL transactor object by time 0 only.

Setting PIPE Width and Frequency

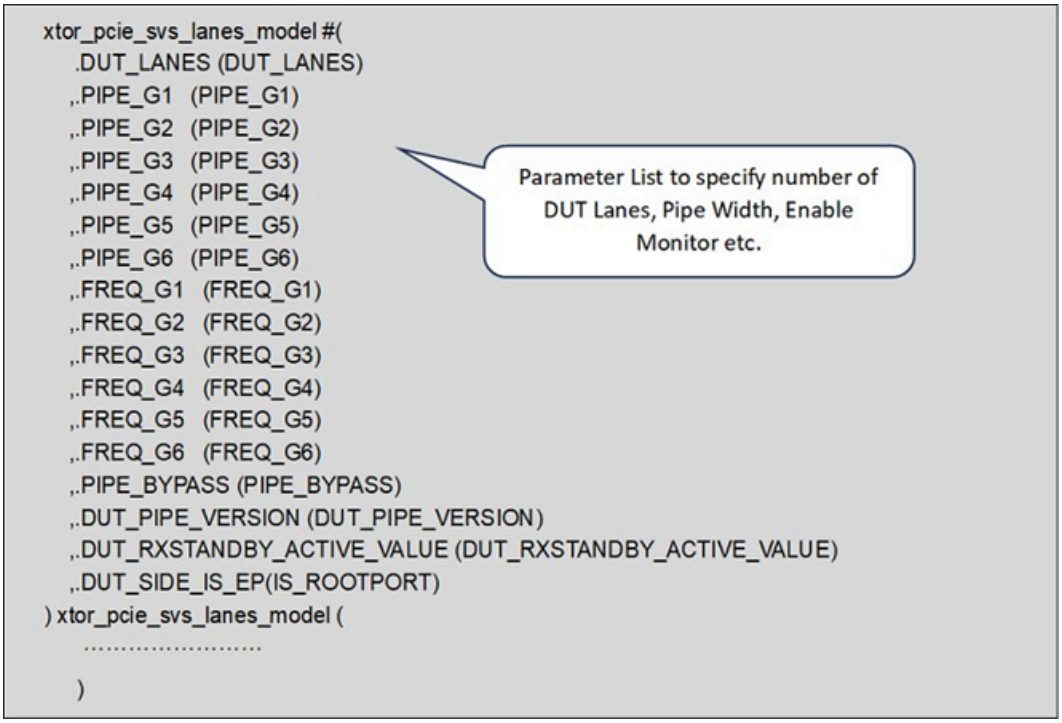
Set the PIPE WIDTH and FREQUENCY settings with respect to the DUT during the bring-up of CXL transactor. While integrating the CXL transactor, along with the lane model, set the following parameters:

- DUT_LANES: Set the number of lanes for the DUT.
- PIPE_G1to PIPE_G5: Set the DUT PIPE width for G1 to G5.
- FREQ_G1to FREQ_G5: Set the DUT frequency for G1 to G5

To set the PIPE width and frequency with respect to the DUT, see Introducing CXL Wrapper Module .

Feedback

The following figure illustrates setting the lane model parameters:



Variable Link Width

If the DUT supports a link width less than the maximum, that is, x16 and the pipe width is original PIPE and 40-bit SerDes PIPE, the link width could be x1,x2,x4,x8, or x16. In this case, perform the following checks for a quick bring-up:

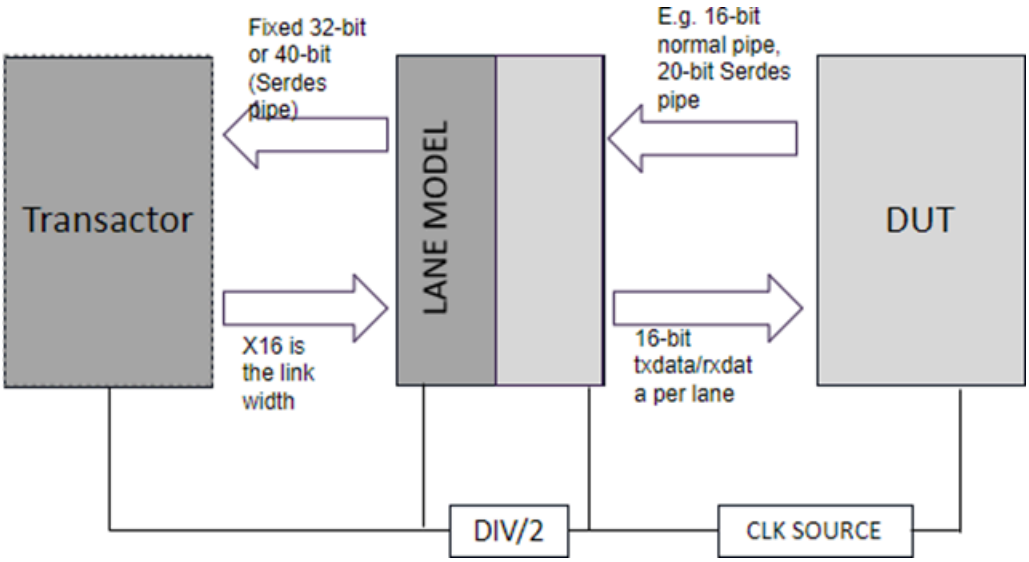
- The connections from the DUT to Lane model should be connected as per the maximum link width supported.
- In the C / zRCI function, specify the PCIE_MAXLINKWIDTH as per the required link width.

Link Bifurcation

If DUT supports the PIPE width less than the maximum value of original PIPE and 40-bitSerDes PIPE, the possible values are 8/16/32 bit for original PIPE and 10/20/40 bit for SerDes PIPE.

Feedback

The following figure shows an example of 16 lanes,16-bits original PIPE/20-bits SerDes PIPE and target speed of GEN5:



In this case, perform the following checks for a quick bring-up:

- The connection from the lane model to the transactor is fixed at 32-bit original PIPE or 40-bit SerDes PIPE with number of lanes connected 16.
- The fastest_pcie_clk is the maximum frequency clock at the transactor side, which is 32-bits PIPE width. It can be lower than the maximum frequency clock at the DUT side when:
  - the DUT PIPE width is lower than 32-bits.
  - the DUT side is in throttling mode.
- max_phy_rate and fastest_pcie_clk is consistent and set to same max target speed at the transactor side.
- Maintain clock ratios as per the PIPE width as explained in Maintaining Clock Ratios.

Note:

Use the pcie_wrapper_svs module in the example/src/dut area to avoid duplication of effort and error prone connection between the lane model and the transactor driver.

PIPE Signal Width Mismatch

Check for any mismatch in the width of PIPE signal of DUT and lanes model. For example, since the txdatavalid_from_dut is a per lane signal, ensure that it is connected for each

lane properly. Not fixing the mismatch in the width of PIPE signal may lead to link-up issues.

Maintaining Clock Ratios

The fastest_pcie_clk and the phy_clk_dut has to maintain the ratio as per the PIPE width. For example, if the DUT pipe width is 16-bit then the ratio between fastest_pcie_clk and the phy_clk_dut(maximum speed here GEN5) should be 1:2.

Consider an example where max_phy_rate is 4, pipe width of DUT = 16, Gen4 frequency (fastest_pcie_clk) of DUT is 1000MHz, then the Gen4 frequency (fastest_pcie_clk_xtor) of the transactor must be 500MHz due to 32-bits.

Assuming fastest_pcie_clk is used to generate/clock divide DUT Clocks Gen1/Gen2/Gen3/Gen4, connect fastest_pcie_clk_xtor to transactor and Lanes Model, which is half of fastest_pcie_clk. This is due to the PIPE width difference between DUT and XTOR.

The lane model clocks (phy_clk_dut and phy_clk_xactor) are generated as per the operating frequency.

The following table lists the clock frequencies when max_phy_rate is Gen4.

PCIe Generation	Clock
Gen1	fastest_pcie_clk/8
Gen2	fastest_pcie_clk/4
Gen3	fastest_pcie_clk/2
Gen4	fastest_pcie_clk

Linkup

The following are the common checks during the transactor linkup:

- Link Up Failing at Speed Change
- DUT LTSSM State Stuck at Polling Compliance
- Link is Stuck in Recovery.RcvrLock and Recovery.Speed
- CXL Transactor Link Not Up When DUT Max Rate Is Gen4
- Unable to Retrain PCIe Link
- Fixing the Halt Observed in the Equalization State

Feedback

- LTSSM in Detect Substates
- LTSSM in Polling Substates
- LTSSM in Configuration Substates
- Using the alias Functionality

Link Up Failing at Speed Change

The following INFO message is reported when the ltssm state is changing to Gen1 and then failing during speed change (especially in B2B setup), as shown below:

```
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: L0
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: recovery_receiverlock
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: recovery_receiverconfig
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: recovery_speed
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: recovery_receiverlock
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: recovery_equalization1
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: recovery_speed

INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: pre_detect_Quiet
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: detect_quiet
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: detect_active
INFO : wrapper.cxl_wrapper_svs.cxl_driver  [xtor_cxl_svs] Current ltssm
state: polling_active
```

To fix this issue, check the value driven on the pclkchangeack_from_dut port. Ensure that it is tied to 1 instead of 0 as shown below:

```
assign ep_pclkchangeack_from_dut={DUT_LANES{1'b1}};
```

Feedback

DUT LTSSM State Stuck at Polling Compliance

To fix the issue of DUT LTSSM state stuck at polling compliance, check for the following:

1. Check if DUT_LANES, PIPE_G* and FREQ_G* parameters to lanes models are set according to DUT configuration as explained in Setting PIPE Width and Frequency.
2. Check if the reset signals of following modules are toggled and properly connected.
   - The pl_rstn_dut, pl_rstn_xactor and npor_out sgnals of the xtor_pcie_svs_lanes_model module.
   - The npor_out and perst_n signals of the xtor_cxl_svs module.
3. Check if lane model clocks (phy_clk_dut and phy_clk_xactor) are generated as per the operating frequency.

The following table lists the clock frequencies when max_phy_rate is Gen5.

Table 24 Clock Frequencies when max_phy_rate is Gen5

PCIe Generation	Clock
Gen1	fastest_pcie_clk/16
Gen2	fastest_pcie_clk/8
Gen3	fastest_pcie_clk/4
Gen4	fastest_pcie_clk/2
Gen5	fastest_pcie_clk

By default, transactor is configured to the Fast Link Mode (scaledown). To disable this mode, if the DUT is operating in real mode (that is, LTSSM timeout values are actual values defined in specification), specify the following configuration parameter:

```
xtor->setConfigParam("PCIE_TIMING",(uint32_t)0);
```

Note:

You might see link up issue during Gen1 and LTSSM stuck in Polling state if DUT is in the real mode and transactor is in the fast link mode.

Link is Stuck in Recovery.RcvrLock and Recovery.Speed

Check if the clocks are generated properly, as described in Maintaining Clock Ratios.

Feedback

When a link reaches Gen1 but does not proceed to higher rates, that is, Gen2/3/4/5, check for the following:

- If the pclkchangeack_from_dut is connected to DUT: If the DUT does not support this signal, tie it to 1'b1 for each lane in the lane model, as shown below:

  ```
  pclkchangeack_from_dut( {DUT_LANES{1'b1}}
  ```

- The txelecidle_from_dut connection: If the DUT supports PIPE4.x, the width is 1 bit per lane and DUT_PIPE_VERSION should be set to 4 in the lane model parameters.
- If the signal rxstandby_from_dut is driven during speed change: Set the DUT_RXSTANDBY_ACTIVE_VALUEparameter to 1'b0 if DUT does not support RxStandBy value as 1.

CXL Transactor Link Not Up When DUT Max Rate Is Gen4

When the DUT max rate is Gen4, the following changes are needed while connecting the transactor.

1. Make the following changes in the hardware connection file:

   ```
   Xtor_cxl_svs cxl_driver_ep (.device_type (ENDPOINT),.max_phy_rate
    (3'h4),.clkcycle(ep_clkcycle).....
   ```

2. Make the following changes in the software testbench:

   ```
   xtor->setConfigParam("PCIE_TARGETSPEED", (unit32_t)4);
   ```

3. Also,ensure that the testbench is NOT waiting for Gen5, as shown in the following example:

   ```
   xtor->wait_for(RateGen5);//should not be called
   ```

If the problem persists, check if the clocks are generated properly, as explained in Maintaining Clock Ratios

Unable to Retrain PCIe Link

LTSSM state changes may not appear in the generated log file for link retrain phase. To fix this, check the waveform for LTSSM state changes as it might already be happening while the link is going through retrain phase. The ltssm_info signal is used to check the current ltssm state and its encoding is provided in ltssm_state Encoding.

Feedback

Fixing the Halt Observed in the Equalization State

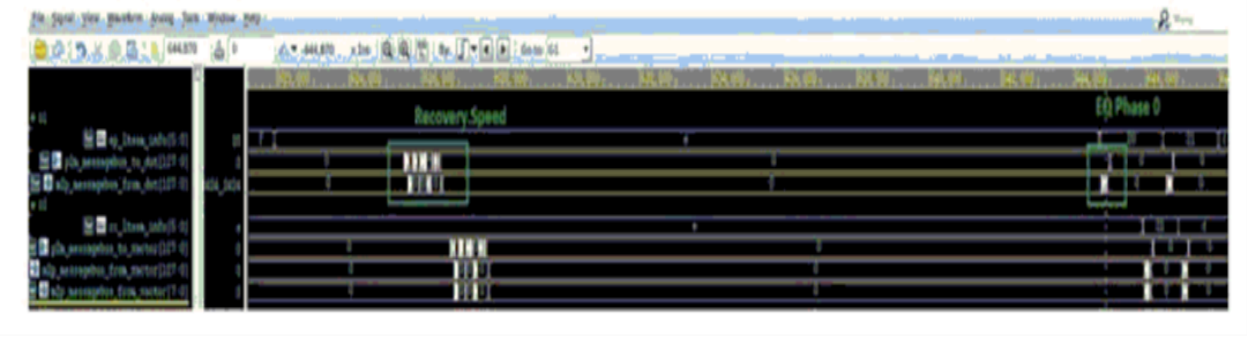
If LTSSM state is at halt in the EQ.0 state, there could be an issue between transactor and lane model. To fix this, check whether DUT supports PIPE 4.4 version or PIPE 5.1 version. If it supports PIPE 4.x version, do not connect P2M and M2P signals.

However, if DUT only supports PIPE 5.1, check that the m2p and p2m message bus are not driven by 0 and connected properly as shown in the following example:

```
==============wrapper.v of example ======================
`ifdef PIPE_511
assign ep_rxeqeval ={DUT_LANES{1'b0}};
assign ep_getlocalpresetcoefficients ={DUT_LANES{1'b0}};
assign ep_invalidrequest ={DUT_LANES{1'b0}};
assign ep_txdeemph ={DUT_LANES{18'b0}};
assign ep_fs ={DUT_LANES{6'b0}};
assign ep_lf ={DUT_LANES{6'b0}};
assign ep_rxpresethint ={DUT_LANES{3'b0}};
assign ep_localpresetindex ={DUT_LANES{5'b0}};
assign ep_rxpolarity ={DUT_LANES{1'b0}};
assign ep_p2m_messagebus[DUT_LANES*8-1:0] = ep_p2m_messagebus_from_dut;
`else // PIPE 4.4.1
wire [DUT_LANES-1:0] ep_rxeqeval_from_dut;
wire [DUT_LANES-1:0] ep_getlocalpresetcoefficients_from_dut;
wire [DUT_LANES-1:0] ep_invalidrequest_from_dut;
wire [DUT_LANES*18-1:0] ep_txdeemph_from_dut;
wire [DUT_LANES*6-1:0] ep_fs_from_dut;
wire [DUT_LANES*6-1:0] ep_lf_from_dut;
wire [DUT_LANES*3-1:0] ep_rxpresethint_from_dut;
wire [DUT_LANES*5-1:0] ep_localpresetindex_from_dut;
wire [DUT_LANES-1:0] ep_rxpolarity_from_dut;
assign ep_rxeqeval =ep_rxeqeval_from_dut;
assign ep_getlocalpresetcoefficients
 =ep_getlocalpresetcoefficients_from_dut;
assign ep_invalidrequest =ep_invalidrequest_from_dut;
assign ep_txdeemph =ep_txdeemph_from_dut;
assign ep_fs =ep_fs_from_dut;
assign ep_lf =ep_lf_from_dut;
assign ep_rxpresethint =ep_rxpresethint_from_dut;
assign ep_localpresetindex =ep_localpresetindex_from_dut;
assign ep_rxpolarity =ep_rxpolarity_from_dut;
assign ep_p2m_messagebus = {DUT_LANES{8'b0}};
`endif
```

Feedback

For PIPE 5.1, ensure that the P2M and M2P signals are driven, as shown in the following figure:



In the above figure:

- During Recovery.Speed PHY (Lane Model), transactor initiates LocalFS/LocalLF exchange using the p2m_messagebus_to_dut signal and then DUT provides the acknowledgment using the m2p_messagebus_from_dut signal.
- During the equalization Phase0, DUT has to initiates local coefficient exchange first using the m2p_messagebus_from_dut signal and PHY has to provide acknowledgment using the p2m_messagebus_to_dut signal.

Note:

These handshakes are also needed when you want to bypass EQ Phase2 and EQ Phase3 state.

LTSSM in Detect Substates

The following explains various substates for the Detect substate:

- Detect.Quiet State
- Detect. Active State

Detect.Quiet State

During LTSSM Detect.Quiet state:

- The controller is in the Electrical Idle state.
- The LTSSM state moves to Detect.Active state after a timeout of 12ms or when any Lane exits Electrical Idle.

Feedback

Detect. Active State

During the LTSSM Detect.Active state:

- DUT MAC asserts txdetectrx_loopback_from_dut to request wrapper to perform receiver detection.
- Wrapper performs receiver detection and asserts phystatus_to_dut for 1 PCLK cycle for acknowledgement.
- Wrapper provides result on rxstatus_to_dut (b'000: Receiver NOT present, b'011 Receiver Present)
- If Receiver is NOT detected, then LTSSM state goes back to Detect.Quiet and it moves to Detect.Active after 12ms timeout. This repeats until Receiver is detected.

The following figure illustrates the LTSSM Detect.Active state:

LTSSM in Polling Substates

The polling sub-state is used for establishing bit-lock, symbol lock, and configuring lane polarity by the transmission and reception of Training Ordered Sets.

The following explains various substates for the Polling substate:

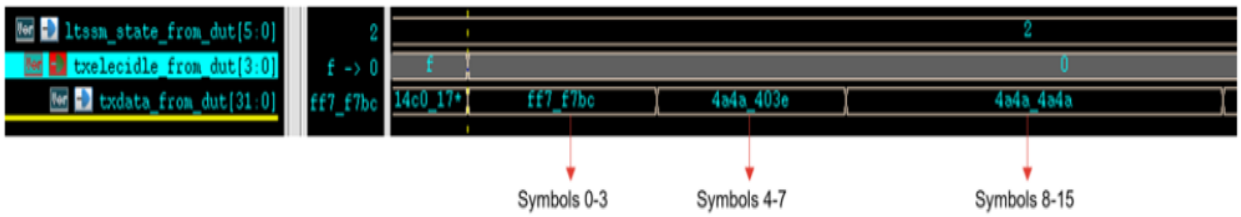
- Polling.Active
- Polling.Active and Polling.Configuration

Feedback

Polling.Active

During the LTSSM Polling.Active state, the DUT MAC:

- Drives txelecidle_from_dut to low
- Sends 1024** TS1 ordered sets on all lanes txdata_from_dut where the receiver is detected during the Detect state, as shown in the following figure:



- Receives 8 consecutive TS1 ordered sets on all detected lanes from its link partner rxdata_to_dut [Wrapper drives rxelecidle_to_dut to low], as shown in the following figure:

Polling.Active and Polling.Configuration

After sending atleast 1024** TS1s and receiving 8 consecutive training sequences, the next state is Polling.Configuration, if any one of the following conditions are satisfied:

- TS1s with Link and Lane set to PAD were received with the compliance Receive bit cleared to 0b (bit 4 of Symbol 5)
- TS1s with Link and Lane set to PAD were received with the Loopback bit of Symbol 5 set to 1b.
- TS2s were received with Link and Lane set to PAD.
- After 24ms timeout, a predetermined subset of the lanes exits electrical idle and any lane receives 9 consecutive TS1/TS2 Ordered sets.

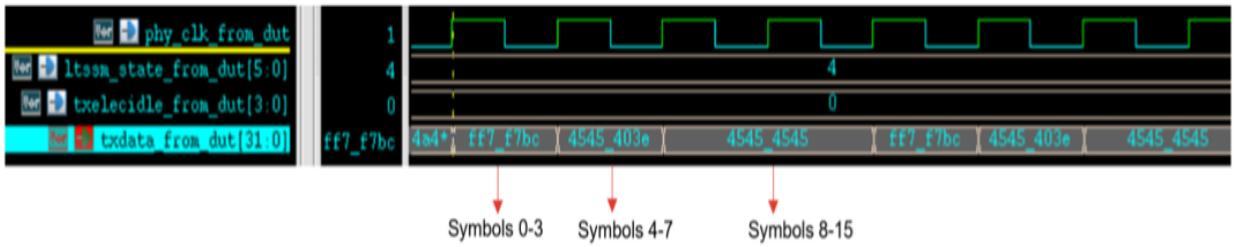
Note:

When Fast Link Mode is enabled using the following configuration parameter during Polling.Active, the controller sends only 16 TS1s as opposed to 1024 TS1s required by the PCIE spec.

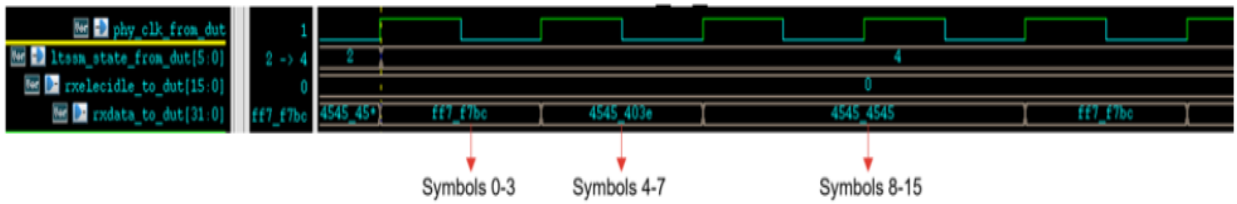
Feedback

```
xtor()->setConfigParam("PCIE_TIMING",(uint32_t)
 XTOR_CXL_SVS_SHORT); where XTOR_CXL_SVS_SHORT=0
```

- The LTSSM state moves to Polling.Compliance, if all the lanes from the predetermined set of lanes do not detect an exit from electrical idle. This indicates there is a problem with some lanes of the link.
- The LTSSM state moves to Detect, if all the predetermined set of lanes exited electrical idle but did not receive the required TS Ordered Sets. This requires the link to be reset and re-trained.
- LTSSM loops between Polling.Active <-> Polling.Compliance or Polling.Active <-> Detect until the conditions to move to Polling.Configuration are met.
- During Polling.Configuration (LTSSM State = 0x4), the DUT MAC sends TS2s with Link and Lane numbers (Symbols 1 & 2) set to PAD (0xF7) on all detected lanes txdata_from_dut, as shown in the following figure:



- The LTSSM state moves to Config after receiving eight consecutive TS2s with Link and Lane set to PAD on all detected lanes rxdata_to_dut.



LTSSM in Configuration Substates

In configuration substates, both link partners exchange TS OS to determine the link number and lane numbers.

Feedback

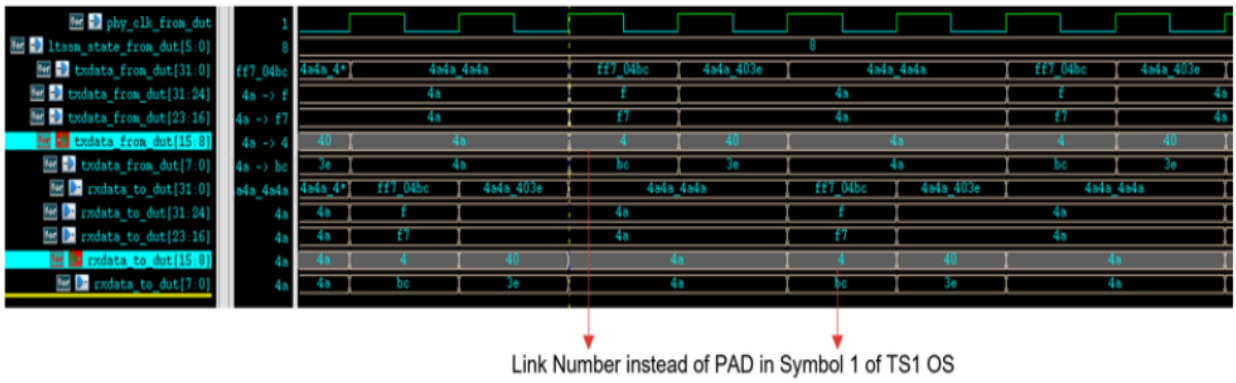
The following are the Configuration substates:

- Configuration.LinkWidth.Start (LTSSM State = 0x7)

  In this state, DUT MAC starts sending Link Number instead of PAD for Symbol 1. After the DUT MAC receives TS1 ordered sets with Link number as Symbol1, the LTSSM proceeds to next state Configuration.LinkWidth.Accept

- Configuration.LinkWidth.Accept (LTSSM State = 0x8)

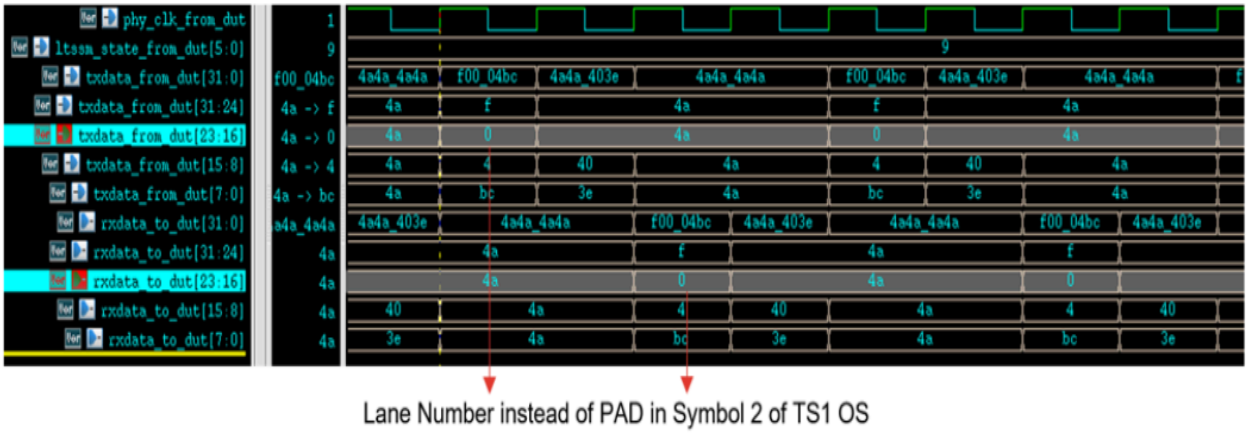
  Both link partners accept the Link Width sent during TS1 ordered sets and move to next state Configuration.LaneNum.wait.



- Configuration.LaneNum.wait (LTSSM State = 0x9)

  In this state, DUT MAC sends the lane number assignments on symbol 2 TS1 ordered sets for each lane txdata_from_data.

  DUT MAC waits until it receives the lane number assignments from its link partner on Symbol2 in TS1 ordered sets for each lane rxdata_to_dut.



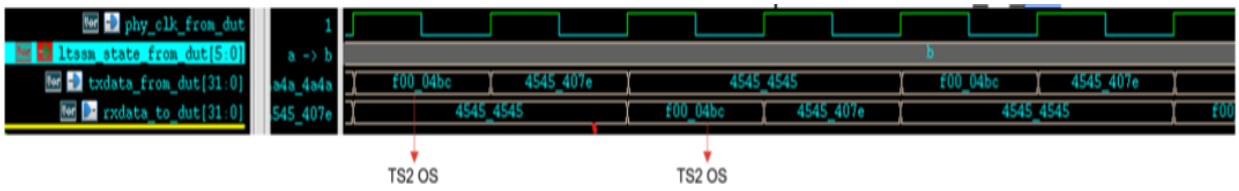
- Configuration.LaneNum.Accept (LTSSM State = 0xA)

Feedback

- Configuration.Complete (LTSSM State = 0xB)

  In this state, DUT MAC sends TS2 ordered sets with link and lane number assignments for all lanes txdata_from_dut.

  DUT MAC waits until it receives the TS2 ordered sets from its link partner on rxdata_to_dut.



  At the end of Configuration.Idle, both sides should have link number and lane numbers assigned.

  The next state is L0 which mean Link is up in Gen1 successfully. LTSSM state = 0x11.

Feedback

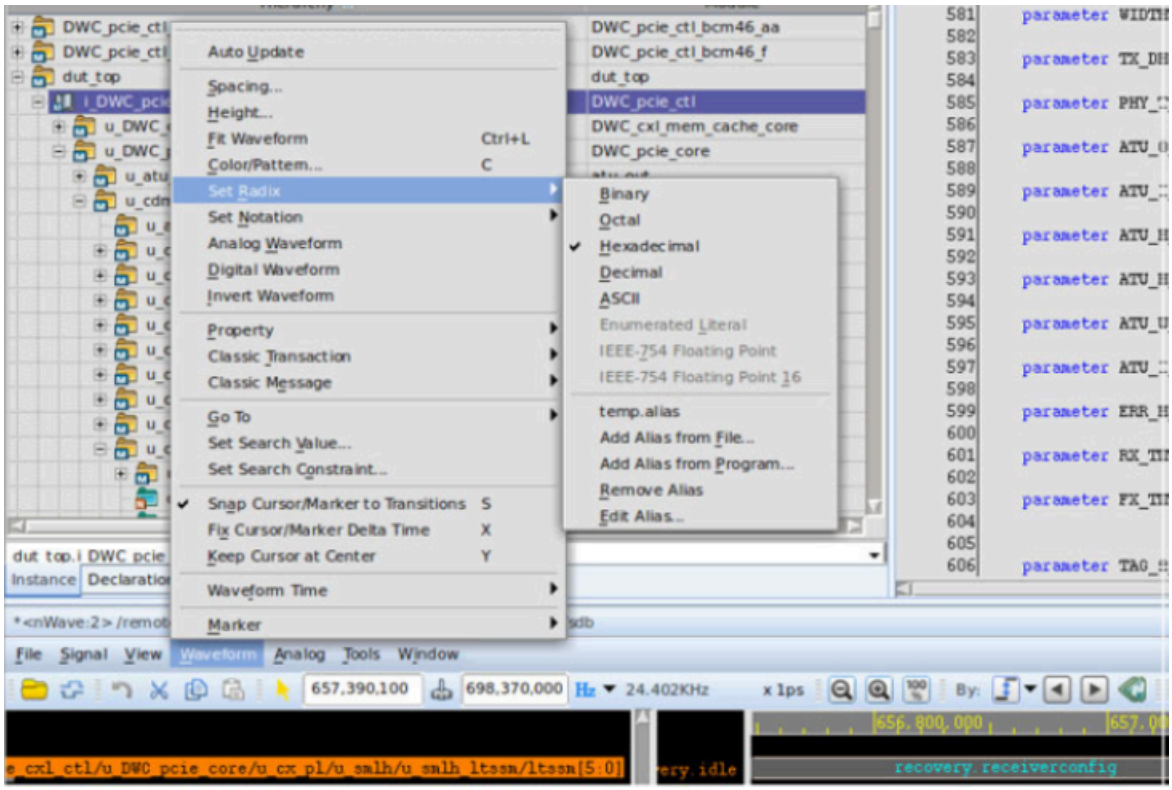
Using the alias Functionality

For ease of display, use the alias functionality in the Verdi waveform. Perform the following steps to enable this functionality:

1. Create the ltssm.alias file as show in the following figure:

Feedback

2. From the Waveform window in Verdi, select the ltssm signal on which you want to apply the alias on, as shown in the following figure:



3. Click on the Add Alias from file option and load the respective ltssm.alias file.

   After the reset is done, the phy_mac_phystatus must go low on all active lanes, LTSSM begins with Detect.Quiet state, as shown in the following figure:

Runtime

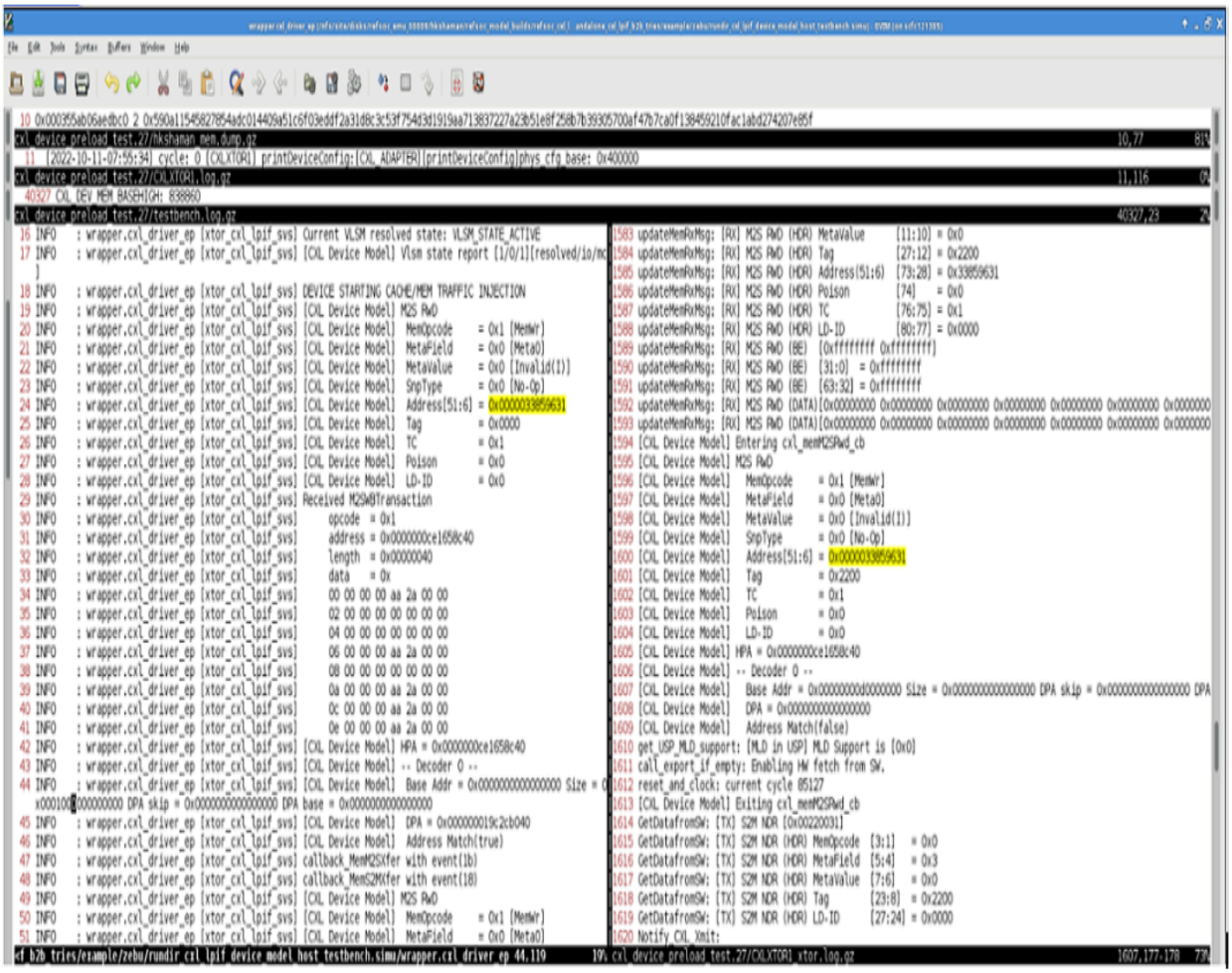
The section explains the following topics:

- The Dump Mem API not Reporting
- Warm Reset Functionality not Working

Feedback

The Dump Mem API not Reporting

When comparing the passing logs (LHS) with failing ones (RHS), the Address Match (true) text is displayed, as shown in the following figure, Whereas in the user logs, Address Match (false) is observed.



The above figure shows that the address does not match if one of the HDM decoder range is dropped.

The dump mem API does not generate report, if:

- The decoder is disabled.
- If the decoder is enabled, ensure addressMatch is true.

To fix this, disable the decoder and re-run the test.

Warm Reset Functionality not Working

In emulation, the RC performs warm reset and switches to the DETECT QUIET mode. However, EP does not move to the DETECT QUIET mode. This happens when the cxl_training() API is used and the also the following value is xtor->setConfigParam("PCIE_AUTOTRAINING_ENA","false"); used for setConfigParam:

```
xtor->setConfigParam("PCIE_AUTOTRAINING_ENA","false");
```

To fix the issue, use the reqFor API instead of the cxl_training API as shown below:

```
device->cxl_warm_reset();
device->cxl_xtor()->reqFor(CXL_SVS::linktraining_ena, true);
```

EP Transactor not Sending the Completions Correctly

For memory read requests initiated by the host, the testbench of EP transactor is responsible for sending completions.

So, the first thing is to check is if completions are sent by the testbench correctly.

- If Completion TLP is sent directly in callbacks (that is, ep_receive_callback), use the non-blocking method pushtlp(tlp) instead of send(tlp) which is blocking.

  As the clocks are stopped whenever callbacks are triggered, blocking call may result in a halt.
- Alternatively, you can push completion TLP in a queue and send it in another thread using send(tlp); The IP ROOT example the same.

Run Time Halt Observed After Reset

The following run time halt is observed after reset.

```
INFO : wrapper.cxl_driver_rc [xtor_cxl_svs] -wrapper.cxl_driver_rc- Reset
 de-asserted ...

DEBUG1: wrapper.cxl_driver_rc [xtor_cxl_svs] isResetActive() status for
 [wrapper.cxl_driver_rc]
```

```
INFO : wrapper.cxl_driver_rc [xtor_cxl_svs] -wrapper.cxl_driver_rc- Reset
 asserted ...

INFO : wrapper.cxl_driver_rc [xtor_cxl_svs] -wrapper.cxl_driver_rc- Reset
 de-asserted ...
```

To resolve this issue, ensure that the XtorScheduler loop is called properly after transactor handle is constructed in the testbench, as shown in the following example:

```
Zemi3->init();
……
//Xtor construction
Xtor=static_cast<xtor_cxl_svs*>(Xtor::getNewXtor(xtor_cxl_svs::getXtorTyp
eName(), XtorRuntimeZebuMode))
Zemi3->start();
……
// XtorScheduler loop
uint32_t initializing = 1;
while (initializing != 0) {
if (!Xtor::AllXtorInitDone()) {
initializing = 1;
} else {
initializing = 0;
}
}
```

Error During Stress Testing of MSI

The following error message is reported by the transactor during the stress testing of MSI:

```
ERROR: sendMSI: Grant has not been received for previous MSI request
```

To fix this, ensure at least 4-5 cycles delay between two consecutive sendMSI() API. The delay is required because, internally the IP responds to MSI requests in 3 cycles with grant signal and adding delay would be good to send MSI after receiving the grant for previous MSI. The delay can be added using the cxl_idle () API.

Transactor Not Working at Default Frequency

It is recommended to check on ZeBu runtime/ ZTime. If using older ZeBu versions, add the following advance UTF commands, as shown in the following example:

```
timing_analysis -post_fpga BACK_ANNOTATED
ztopbuild -advanced_command {enable zmem_clock_domain_instrument}
zpar -advanced_command {System improvedMemoryTiming true}
ztopbuild -advanced_command {zcorebuild_command *
Unknown macro: {timing -analyze
 ZFILTER_ASYNC_SET_RESET_PATH,ZFILTER_ENABLE_ASYNCSR_DATAPATH}
}
```

Feedback

```
ztopbuild -advanced_command {clock_localization -stop_at_async_set_reset
 no}
ztopbuild -advanced_command {clock_handling filter_glitches_synchronous
 -improve_skewtime yes}
zpar -advanced_command {System enableSDFChecker true}
```

Note that the above UTF commands are given just for reference. However, to understand the exact UTF commands needed, contact the Support team.

Infrastructure Errors

This section explains the following infrastructure-related errors and their resolution:

- Internal Error: no platform defined for import DPI call ZEBU_VS_SNPS_UDPI_BuildSerialNumber
- Fatal error: Scope not set prior to calling export function
- Internal Error: unknown scope -x8627a580 for zebu_vs_udpi_rst_and_clk import DPI ZEBU_VS_SNPS_UDPI_rst_assert call
- Fatal error: svt_dpi module either not loaded or not in $root
- Fatal error: driver is already associated to a Transactor SW object
- Fatal error: DPI scope not found for path wrapper.cxl_driver_ep.SNPS_PCIE_RST_CLK
- pushTlp: ERROR : Tx tlp Fifo is Full (128/128)
- No <svt_hw_platform> defined for import DPI call

Internal Error: no platform defined for import DPI call ZEBU_VS_SNPS_UDPI_BuildSerialNumber

The following are the primary reasons for this error:

- Transactor’s software part is not constructed/initialized before any HW signal activity occurs.
- The init sequence is not in the correct order. Ensure the init sequence order is as shown below:

```
zemi3 = ZEMI3Manager::open(zebuWork,designFeatures);
board = zemi3->getBoard();
zemi3->buildXtorList(); // manually add the xtor or use buildXtorList
zemi3->init();
// Register the transactor
```

Feedback

```
xtor_cxl_svs::Register("xtor_cxl_svs");
// Construct the transactor
test =
 static_cast<xtor_cxl_svs*>(Xtor::getNewXtor(xtor_cxl_svs::getXtorType
Name(), XtorRuntimeZebuMode)) ;
// Call XTOR init method if exists for the transactor
test->init(board,"cxl_device_wrapper.uart_driver_0");
zemi3->start();
```

Note:

Call the Zemi3->start() method only after the transactor is constructed.

Fatal error: Scope not set prior to calling export function

The following error is reported if the scheduler is called before the transactor handle is constructed in the testbench.

```
fatal error in ZEMI3 RUN [ZXTOR0030F] : Scope not set prior to calling
 exported function "ZEBU_VS_SNPS_UDPI__configreadwrite"
terminate called without an active exception
```

Internal Error: unknown scope -x8627a580 for zebu_vs_udpi_rst_and_clk import DPI ZEBU_VS_SNPS_UDPI_rst_assert call

To fix this error, check if the transactor software is constructed for each hardware instance. If yes, whether it is constructed at appropriate place.

Fatal error: svt_dpi module either not loaded or not in $root

This error occurs in VCS simulation setup if you are using the two-step VCS compile flow, that is, VLOGAN followed by VCS.

To resolve this issue, perform the following steps:

1. Specify svt_dpi module as one of the top module in your VCS command.
2. Ensure that the file svt_dpi.sv is compiled in the vlogan command.

A similar error can occur for the svt_systemverilog_threading and svt_dpi_globals modules. Follow the above steps for these modules as well.

```
vcs -fulll64 ... -top hwtop dumpvars svt_dpi svt_systemverilog_threading
 svt_dpi_globals
```

Feedback

Fatal error: driver is already associated to a Transactor SW object

This error occurs when transactor software is not constructed properly in simulation.

To fix this issue, check the usage of getNewXtor(..) for each hierarchical instance mapped to different transactor software instance, as shown below:

```
xtor=static_cast<xtor_cxl_svs*>(Xtor::getNewXtor(xtor_cxl_svs::getXtorTyp
eName(), XtorRuntimeZebuMode, <inst_name>));
```

Fatal error: DPI scope not found for path wrapper.cxl_driver_ep.SNPS_PCIE_RST_CLK

The following errors may be reported with zRCi testbench when zemi3 configuration is not set in the Tcl file:

```
INFO : wrapper.cxl_driver_ep [SNPS/XTOR/HDLDRVR/FOUND ] RTL driver
 instance DpiOnly "xtor_cxl_svs" detected.
DEBUG : wrapper.cxl_driver_ep [xtor_cxl_svs] value of path is
 wrapper.pcie_driver_ep.SNPS_PCIE_RST_CLK
svGetScopeFromName : Error: scope
 'wrapper.cxl_driver_ep.SNPS_PCIE_RST_CLK' is unknown.
ERROR : wrapper.cxl_driver_ep [xtor_cxl_svs] DPI scope not found for path
 wrapper.pcie_driver_ep.SNPS_PCIE_RST_CLK
```

To resolve the above errors, check for the following settings:

```
#zemi3 -enable
#zemi3 -config ../16x32b.zebu.work/zebu.work/xtor_dpi.lst
#zemi3 -lib ../16x32b.zebu.work/zebu.work/xtor_cxl_svs.so
```

If you are using VHS, and the above configuration is set, move all the pci0 VM related code to post_board_init instead of post_board_open or move the transactor initialized after zemi3->init() is called.

pushTlp: ERROR : Tx tlp Fifo is Full (128/128)

The following error is reported when the TLP FIFO buffer is in the PCIe software:

```
INFO : [xtor_cxl_svs] pushTlp: ERROR : Tx tlp Fifo is Full (128/128).
INFO : [xtor_cxl_svs] pushTlp: ERROR : Increase Fifo size with
 PCIE_TXTLPFIFO_SIZE ConfigParam (current is 128).
```

This implies TLP FIFO buffer is full in the CXL software. This happens when the testbench is doing a stress test and continuously feeding thousands of TLP transactions with payload bytes > 128.

Feedback

By default, PCIE_MAXPAYLOAD is set to 128 and PCIE_TXTLPFIFO_SIZE is set to 128. If TLP transaction sent with payload bytes >128, for example, 512, then the transactor will sends 4 or 5 split TLP transactions and buffer can fill up fast.

To resolve this error, perform the following steps:

1. Set the correct PCIE_MAXPAYLOAD configuration parameter using the setConfigParam function.
2. Ensure Host DUT sends Configuration TLP to set max.payload size in Device Control Register or configure it through backdoor, if EP is the transactor.

For more information on setting the max payload size, see How to set the max TLP payload size?

1. Increase the PCIE_TXTLPFIFO_SIZE configuration parameter size, which is less than or equal to 4096.
2. Set the PCIE_TXTLPFIFO_SIZE configuration parameter before initBFM/scheduler loop is called.
3. If FIFO is 80% full, wait for the pending TLPs to finish.

```
if (FIFO is 80% full == 0)
{
xtor->wait_pending_txtlp();
}
```

No <svt_hw_platform> defined for import DPI call

The following error is reported when no hardware platform is defined for the import DPI call:

```
Error : [cxl_svs_imp] - No <svt_hw_platform> defined for import DPI call.
Error : Check a valid <svt_hw_platform> is provided to transactor
 constructor through <svt_c_runtime_cfg>.
Error : Check software transactor object is already created before reset
 is released.
Error : Check software transactor object is already created before
 toggling the clock.
```

To resolve the error, ensure that the transactor object is created before calling zemi3->start() and after zemi3->init()

With transactor wrapper, the transactor object is created during xtor_wrapper_svs::cxl_init() call with a runtime object pointer, which is used to initialize the hw_platform.

Monitor Related Errors

This section describes the errors reported for the PCIe monitor:

Undefined symbol: _ZN7ZEBU_IP15MONITOR_CXL_SVS15monitor_cxl_svs8RegisterEPKc

To resolve this error, check if -lmonitor_cxl_svs option is passed in your scripts.

The DPI export function/task 'monitor_en_CXL_DPI' not found

The following error is reported when the DPI export function/task is not found.

```
Error-[DPI-DXFNF] DPI export function not found
```

The DPI export function/task 'monitor_en_CXL_DPI' called from a user/external C/C++/DPI-C code originated from import DPI function 'run_init' at file '../../src/dut/wrapper.v'(line 20) is not defined or visible.

Check the called DPI export function/task is defined in the mentioned module, or check if the DPI declaration of the DPI import function/task which invokes that DPI export function/task is made with 'context'.

Another work-around is using svGetScopeFromName/svSetScope to explicitly set the scope to the module which contains the definition of the DPI export function/task.

To resolve this error, check if the right path is set for the hierarchical instance set (highlighted), while constructing the monitor:

```
cxl_monitor = new monitor_cxl_svs("monitor_cxl_core",
 "wrapper.monitor_top_inst_rc.xtor_cxl_svs_monitor", xsched, runtime);
```

Cannot read Mudb version

The following FATAL error is reported when the monitor cannot read the Mudb version:

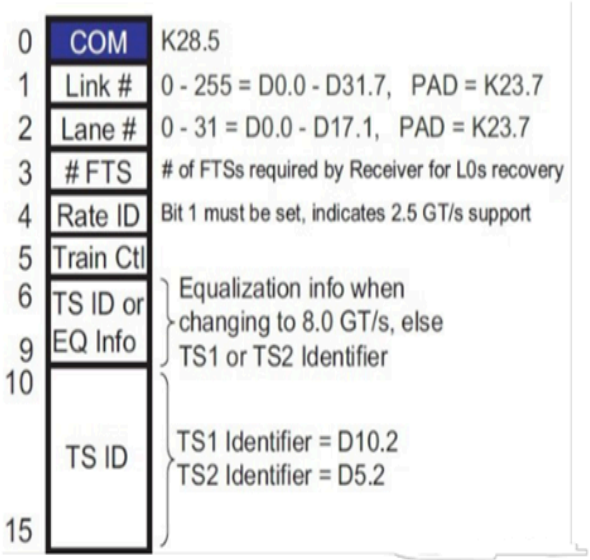
```
zdpiReport: FATAL : CORE0001E : Could not load database: Cannot read MuDb
 version from
```

Feedback

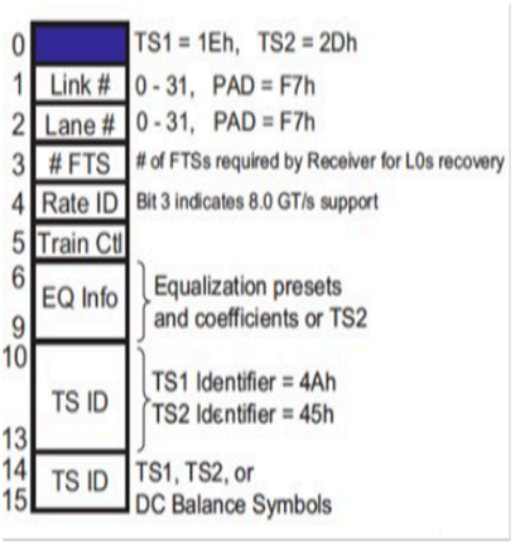
To resolve this error, specify a valid path for zCui.work.

Understanding TS1 and TS2 Ordered Sets

The following figure illustrates the TS1 and TS2 ordered sets in Gen1, Gen2 and Gen3 mode.



TS1 and TS2 Ordered Sets in Gen1 or Gen2 Mode



TS1 and TS2 Ordered Sets in Gen3 Mode

The control symbols are COM (K28.5) = 0xBC , PAD (K23.7) = 0xF7

In Gen1/Gen2

The 16 symbols of TS1 Ordered Set with Link and Lane numbers set to PAD appear like (from Symbol 15 - Symbol 0)

```
4a4a_4a4a 4a4a_4a4a 4a4a_403e 0ff7_f7bc
```

The 16 symbols of TS2 Ordered set with Link and Lane numbers set to PAD appear like (from Symbol 15 - Symbol 0)

```
4545_4545 4545_4545 4545_403e 0ff7_f7bc
```

Frequently Asked Questions

This section explains the following frequently asked questions related to the CXL Transactor:

How to dynamically change the speed and width?

You can dynamically change the speed of the CXL transactor using the PCIE_TARGETSPEED param.

Note that the dynamic speed change upto Gen6 is supported.

You can dynamically change the width of the CXL transactor using the PCIE_MAXLINKWIDTH parameter.

How to set the max TLP payload size?

To set the max payload size to 4096, perform the following steps:

1. Set the Max_Payload_Size Supported bits in Device Capabilities Register as shown below:

   ```
   xtor->setConfigParam("PCIE_MAXPAYLOAD", (uint32_t)4096);
   ```

2. If Max_Payload_Size is required to be set to value less than Max_Payload_Size supported, configure Max_Payload_Size of function in Device Control Register either through frontdoor CFG TLP from host or through backdoor from the transactor.

   If you are unable to configure the Max_Payload_Size of Device using Host in the Device control Register through CFG TLP, use the backdoor access by applying the following code in the testbench.

   ```
   // Setting Max Payload via Backdoor to configure
    Device_Control_Device_Status Reg bits [7:5], Addr: pcie_cap_id + 0x8
   ```

Feedback

```
uint16_t cfg_reg;
device->cxl_BfmCfgRd(ep->cxl_xtor()->getCapPtr(pcie_cap_id)+0x8,&cfg
_re g);
device->cxl_xtor()->print("Value of Device Control Status Reg : %x",
 cfg_reg);
cfg_reg = cfg_reg & 0xFFFFFFBF; cfg_reg = cfg_reg | 0x000000A0;
device->pcie_BfmCfgWr(ep->pcie_xtor()->getCapPtr(pcie_cap_id)+0x8,
 4,cfg_reg);
device->cxl_xtor()->print("Value of Device Control Status Reg :
%x",cfg_reg);
```

By performing the above steps, FIFO will be able to accommodate more TLPs since there are no split TLPs. That is, you can load 10000 MemWr transactions at a time.

How to verify HOT reset?

Hot reset can only be initiated from Host side.

Transactor is Host

For achieving hot reset scenario if transactor is Host, then run the following code:

Step 1: Entering the Hot Reset state

```
//Set the Unmask SBR bit in Port Control Extensions register
uint32_t rdata;
//For CXL 3.0
rdata = rc->cxlXtorCfgRead(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL3+0xC);
rdata |= 0x1;
rc->cxlXtorCfgWrite(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL3+0xC,rdata);
//For CXL 2.0
rdata = rc->cxlXtorCfgRead(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL2+0xC);
rdata |= 0x1;
rc->cxlXtorCfgWrite(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL2+0xC,rdata);
```

Step 2: Exiting from the Hot Reset state

```
//Set the Secondary bus reset field of bridge control register
rdata = 0;
rdata = rc->cxlXtorCfgRead(XTOR_CXL_SVS_TYPE1_BRIDGECTRLINTPININTLINE);
rdata |= 0x00400000;
rc->cxlXtorCfgWrite(XTOR_CXL_SVS_TYPE1_BRIDGECTRLINTPININTLINE,rdata);
//Wait for XTOR to reach HOT Reset LTSSM state
rc->wait_for(XTOR_CXL_SVS::STATE_HOT_RESET_ENTRY);
// Disabling the secondry bus reset bit in bridge control register.
rdata = rc->cxlXtorCfgRead(XTOR_CXL_SVS_TYPE1_BRIDGECTRLINTPININTLINE);
rdata &= 0xffbfffff;
rc->cxlXtorCfgWrite(XTOR_CXL_SVS_TYPE1_BRIDGECTRLINTPININTLINE,rdata);
//Wait for XTOR to reach Detect Quiet LTSSM state
rc->wait_for(XTOR_CXL_SVS::STATE_DETECT_QUIET);
```

Feedback

Step 3: Apply Reset

```
//Assert HW Reset
rc->reqFor(XTOR_CXL_SVS::hw_reset_n, true);
//Wait for few clock cycles
src->wait_for_cycle(100);
//De-assert HW reset
rc->reqFor(XTOR_CXL_SVS::hw_reset_n, false);
```

Step 4: Perform initial configuration

```
//Initialize XTOR again
rc->initBFM();
//Wait for linkup(L0)
rc->wait_for(linkup_done);
```

Transactor is Device

For achieving hot reset scenario if transactor is Device, then run the following code:

Step 1: Entering the Hot Reset state

```
//Wait for XTOR to reach HOT Reset LTSSM state
ep->wait_for(XTOR_CXL_SVS::STATE_HOT_RESET_ENTRY);
```

Step 2: Exiting the Hot Reset state

```
//Wait for XTOR to reach Detect Quiet LTSSM state
ep->wait_for(XTOR_CXL_SVS::STATE_DETECT_QUIET);
```

Step 3: Apply Reset

```
//Assert HW Reset
ep->reqFor(XTOR_CXL_SVS::hw_reset_n, true);
//Wait for few clock cycles in sync with Host's clock cycle wait
ep->wait_for_cycle(100);
//De-assert HW reset
ep->reqFor(XTOR_CXL_SVS::hw_reset_n, false);
```

Step 4: Initial configuration

```
//Initialize XTOR again
ep->initBFM();
////Wait for linkup(L0)
// ep->wait_for(linkup_done);
```

Feedback

How to verify Link Disable?

Link Disable can only be initiated from Host side.

For achieving Link Disable scenario if transactor is Host, then run the following code:

Step 1: Entering the Link Disable state

```
//Set Unmask Link Disable bit in Port Control Extensions register
uint32_t rdata;
//For CXL 3.0
rdata = rc->cxlXtorCfgRead(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL3+0xC);
rdata |= 0x2;
rc->cxlXtorCfgWrite(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL3+0xC,rdata);
//For CXL 2.0
rdata = rc->cxlXtorCfgRead(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL2+0xC);
rdata |= 0x2;
rc->cxlXtorCfgWrite(XTOR_CXL_2_0_EXT_DVSEC_PORT_CXL2+0xC,rdata);
```

Step 2: Exiting the Link Disable state

```
//Set Link Disable bit in Link Control register
rc->cxlXtorCfgWrite (rc->getCapPtr(cxl_cap_id)+0x10, 0x10);
//Wait for XTOR to reach Disabled LTSSM state
rc->wait_for(STATE_DISABLED);
//Resetting Link Disable bit in Link Control register
rc->cxlXtorCfgWrite (rc->getCapPtr(cxl_cap_id)+0x10, 0x00);
//Wait for XTOR to reach Detect Quiet LTSSM state
rc->wait_for(XTOR_CXL_SVS::STATE_DETECT_QUIET);
```

Step 3: Applying Reset

```
//Assert HW Reset
rc->reqFor(XTOR_CXL_SVS::hw_reset_n, true);
//Wait for few clock cyclesrc->wait_for_cycle(100);
//De-assert HW reset
rc->reqFor(XTOR_CXL_SVS::hw_reset_n, false);
```

Step 4: Initial Configuration

```
//Initialize XTOR again
rc->initBFM();
//Wait for linkup(L0)
rc->wait_for(linkup_done);
```

For achieving hot reset scenario if transactor is EP, then run the following code:

Step 1: Entering the Link Disable state

```
//Wait for XTOR to reach DISABLED LTSSM state
ep->wait_for(STATE_DISABLED);
```

Step 2: Exiting the Link Disable state

```
//Wait for XTOR to reach Detect Quiet LTSSM state
ep->wait_for(XTOR_CXL_SVS::STATE_DETECT_QUIET);
```

Step 3: Apply Reset

```
//Assert HW Reset
ep->reqFor(XTOR_CXL_SVS::hw_reset_n, true);
//Wait for few clock cycles in sync with Host's clock cycle wait
ep->wait_for_cycle(100);
//De-assert HW reset
ep->reqFor(XTOR_CXL_SVS::hw_reset_n, false);
```

Step 4: Initial Configuration

```
//Initialize XTOR again
ep->initBFM();
//Wait for highest speed reached (Eg: Gen6/Gen5)
 ep->wait_for(RateGen6);
//Wait for linkup(L0)
 ep->wait_for(linkup_done);
```

How to verify Link Retrain?

Link Retrain can only be initiated from Host side.

Transactor is Host

For achieving link retrain scenario if transactor is Host, then run the following code:

Step 1: Setting of Retrain Link bit in Link Control Register

```
// Read the current Link Control Register value
rddata = rc->cxlXtorCfgRead((0x70 + 0x10));
rc->print("Back Door Link Status Link Control register value READ: %x\n",
 rddata);
// Set the "Retrain Link" bit 5 to 1
rddata = rddata & 0xFFFFFFDF;
rddata = rddata | 0x20;
// Write the modified content back to Link Control Register
rc->cxlXtorCfgWrite((0x70 + 0x10), rddata);
rddata = rc->cxlXtorCfgRead((0x70 + 0x10));
rc->print("Back Door Link Status Link Control register value READ: %x\n",
 rddata);
```

Step 2: Waiting for LTSSM to reach Recovery State

```
//Wait for XTOR to reach Recovery State
```

Step 3: Waiting for linkup (LTSSM to L0)

```
//Wait for linkup(L0)rc->wait_for(linkup_done);
```

Transactor is Device

For achieving link retrain scenario if transactor is Device, then run the following code:

Step 1: Waiting for LTSSM to reach Recovery State

```
//Wait for XTOR to reach Recovery State ep
 ->wait_for(XTOR_CXL_SVS::STATE_RCVRY_LOCK);
```

Step 2: Waiting for linkup (LTSSM to L0)

```
//Wait for linkup(L0)ep->wait_for(linkup_done);
```

How to verify MSIx Capability?

MSIx capability can only be enabled from Host side.

Transactor is Host

For achieving MSIx scenario if transactor is Host, then run the following code in run thread:

Step 1: Enable MSIx Capability and Discover MSI-X Table offset

```
     msix_cap_ptr = (rc->getCapPtr(msix_cap_id));

     // READING THE MSIX_TABLE OFFSET value
     rc->print("MSIX_DEBUG: HOST XTOR: Reading Table Offset/Table BIR
Register for MSI-X.\n");
     msix_table_offset = rc->cxlXtorCfgRead((msix_cap_ptr + 0x4)>>2);

     // Table BIR is configured for BAR 5 and it is located at
xtor_tb_rc->bar_base[2]
     bar5_base_address   = 0xA0000000;
     msix_table_bir      = msix_table_offset & 0x7;
     msix_table_offset   = msix_table_offset >> 3;
     msix_vector_address = msix_table_offset + bar5_base_address; // for
programing the the msix vector int he EP BAR space
     rc->print("MSIX_DEBUG: HOST XTOR: Read Table Offset/Table BIR
Register for MSI-X. Offset= 0x%0x, BIR = 0x%0x MSIx Vector address =
0x%0x.\n",msix_table_offset,msix_table_bir,msix_vector_address);

     //Initiate Memory Write for updating MSIx Vector table
```

Feedback

```
      MSIX_address_data[0] = 0x000ACAFC;
      MSIX_address_data[1] = 0x00000001;
      MSIX_address_data[2] = 0x00000ABF;
      MSIX_address_data[3] = 0x00000000;

      len=4 ;
      tlp->setTag(xtor_tb_rc->rc_tag);
      tlp->writeMem32(msix_vector_address,MSIX_address_data,len);
      rc->pushTxTlp(tlp);
      xtor_tb_rc->rc_tag++;
      rc->print("MSIX_DEBUG: HOST XTOR: Initiated Memory Write at address
 = %0x for updating MSIx Vector table.\n",msix_vector_address);
    //Reading Message Control Register for MSIx.
      rc->print("MSIX_DEBUG: HOST XTOR: Reading Message Control Register
 for MSIx.\n");
      msix_cap_reg_read_value = rc->cxlXtorCfgRead((msix_cap_ptr +
 0x4)>>2);
      rc->print("MSIX_DEBUG: HOST XTOR: Read Message Control Register for
 MSIx %0x.\n",msix_cap_reg_read_value);
      // Initiate Configuration Write for Enabling MSIx in Message
 Control Register
      msix_cap_reg_read_value |= (0x1<<31);
      tlp->setTag(xtor_tb_rc->rc_tag);
      tlp->writeCfg0(msix_cap_reg_read_value,(msix_cap_ptr>>2),0x01,0,0);
      ++xtor_tb_rc->rc_tag;
      rc->pushTxTlp(tlp);
      rc->print("MSIX_DEBUG: HOST XTOR: Initiated Configuration
 Write for Enabling MSIx in Message Control Register
 %0x.\n",msix_cap_reg_read_value);
```

Step 2: Waiting for MSI-X Interrupt from Device

```
// Waiting for MSIx Interrupt from Device
   rc->print("MSIX_DEBUG: HOST XTOR: Waiting for MSIX Interrupt from
 EP.\n");
   while(MSIX_INT_received==false){
      rc->wait_for(Anything);
  }
// Received MSIX Interrupt from Device
  rc->print("MSIX_DEBUG: HOST XTOR: Received MSIX Interrupt from EP.\n");
```

Step 3: Modify Callback thread (rc_receive_callback) to have below code

```
if(tlp->isMemWrite()) {
     XTOR_TB_RC->print("++++ RC Callabck:  Rcvd memory Write tlp \n");
     if((*((uint32_t*)(tlp->getPayload())) == 0x00000ABF)){
  // MSIX_INTERRUPT RECEIVED. Received Memory write with data 0x00000ABF.
                XTOR_TB_RC->print("MSI_DEBUG: Host XTOR: MSIX_INTERRUPT
 RECEIVED. Received Memory write with data 0x00000ABF.\n");
                MSIX_INT_received_mutex.lock();
                MSIX_INT_received = true;
                MSIX_INT_received_mutex.unlock();
```

Transactor is Device

For achieving MSI scenario if transactor is Device, do the following configuration

```
(ep->setConfigParam("PCIE_BAR5SIZE",        "0x00FFFFFF"));// for MSI-X
      (ep->setConfigParam("PCIE_BAR5TYPE",       "0x00"));
      (ep->setConfigParam("PCIE_MSIX_TABLE_SIZE",      "2"));
      (ep->setConfigParam("PCIE_MSIX_TABLE_OFFSET",    "0x00000045"));
```

then run the following code in run thread:

Step 1: Send MSIx Interrupt

```
// Send MSIX interrupt
  ep->print("MSIX_DEBUG: DEVICE XTOR: Initiating MSIX Vector 0.\n");
  ep->sendMSIX(0x00000ABF);
  ep->print("MSIX_DEBUG: DEVICE XTOR: Initiated MSIX Vector 0.\n");
```

How to access Device's configuration registers by frontdoor mechanism?

Configuration Write/Read can only be initiated from Host side.

Transactor is Host

1: Frontdoor Configuration write e.g if user wants to write BAR0 with base address 0x80000000

```
xtor_tb_rc->write_pcie_configuration(0,0x80000000,
 XTOR_CXL_SVS_TYPE0_BAR0>>2, BUSNO, DEVICENO,FUNCTIONNO,0xf);
```

2: Frontdoor Configuration read e.g if user wants to read BAR0

```
xtor_tb_rc->read_pcie_configuration(0,XTOR_CXL_SVS_TYPE0_BAR0>>2, BUSNO,
 DEVICENO,FUNCTIONNO);
```

7

Appendix

This section explains the following topics:

Transactor Interface Signal Encoding

This section explains transactor interface encoding for the following:

ltssm_state Encoding

The following table reports the encoding of the transactor LTSSM state as provided on the ltssm_state[5:0] output sideband signal:

Table 25 ltssm_state Encoding

LTSSM State	LTSSM value
Detect.Quiet	0x0
Detect.Active	0x1
Polling.Active	0x2
Polling.Compliance	0x3
Polling.Configuration	0x4
Pre Detect Quiet	0x5
Detect_wait	0x6
Config.LinkWidthStart	0x7

Table 25 ltssm_state Encoding (Continued)

LTSSM State	LTSSM value
Config.LinkWidthAccept	0x8
Config.LaneNumWait	0x9
Config.LaneNumAccept	0xA
Config.Complete	0xB
Config.Idle	0xC
Recovery.ReceiverLock	0xD
Recovery.Speed	0xE
Recovery.ReceiverConfig	0xF
Recovery.Idle	0x10
Recovery.Eq0	0x20
Recovery.Eq1	0x21
Recovery.Eq2	0x22
Recovery.Eq3	0x23
L0	0x11
L0s	0x12
L123_send_idle	0x13
L1 idle	0x14
L2 idle	0x15
L2 wake	0x16
Disabled.Entry	0x17
Disabled.Idle	0x18
Disabled	0x19
Loopback.Entry	0x1A
Loopback.Active	0x1B

Feedback

Table 25 ltssm_state Encoding (Continued)

LTSSM State	LTSSM value
Loopback.Exit	0x1C
Loopback.Exit_timeout	0x1D
Hotreset.Entry	0x1E
Hotreset.Idle	0x1F

l1sub_state Encoding

The following table reports encoding of the transactor L1 sub-states as provided on the l1sub_state[3:0] output sideband signal:

L1 Substate	Value
L1_U	0x0
L1_0	0x1
L1_0_WAIT4_ACK	0x2
L1_0_WAIT4_CLKREQ	0x3
L1_N_ENTRY	0x4
L1_1	0x5
L1_N_EXIT	0x6
L1_N_ABORT	0x7
L1_2_ENTRY	0xC
L1_2	0xD
L1_2_EXIT	0xE

Optional CLKREQ# Sideband Signals

CLKREQ# is an open-drain, active low signal that is driven low by a PCIe function. It is recommended to be used for any add-in card or system board that supports power management and L1 power sub-states features.

Feedback

As it is not possible to have a direct model for open-drain signals inside ZeBu, the transactor interface implements the following signals:

- cfg_l1sub_en
- clkreq_in_n

The following figure illustrates the sideband signal when the transactor acts as a RootPort:

Figure 28 Transactor as a Rootport