ZeBu PCIe Transactor User Guide

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 PCI Express (PCIe) 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 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 PCIe transactor implements a PCI Express interface that can function as Root Complex (RC) or Endpoint (EP) and carry bit rates of 2.5, 5.0, 8.0, 16.0, 32.0, and 64.0 Gigatransfer per second (GT/s).

This section explains the following topics:

- Supported Transactor Configurations
- Supported PCIe Specifications
- Supported PCIe Protocol Features
- Supported Transactor Features
- Limitations

Supported Transactor Configurations

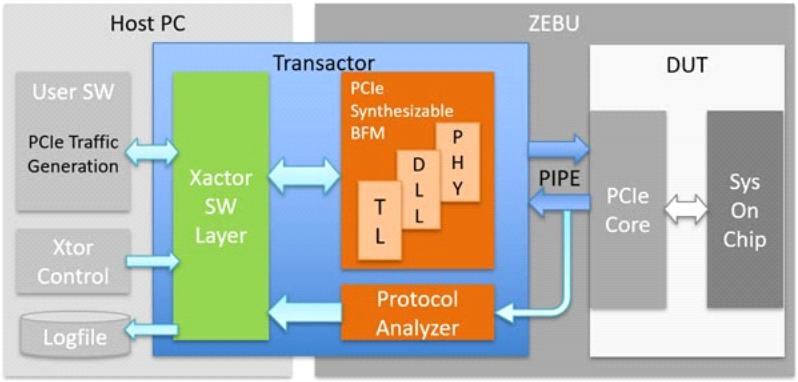
The ZeBu PCIe transactor supports the following configurations:

- Link width - x16
- PIPE width
  - Original: 32 bits
  - SerDes: 40 bits (upto Gen5), 80 bits (Gen6)

Feedback

The following figure illustrates a sample connection for the PCIe Transactor:

Figure 1 PCIe 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 PCIe 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 PCIe Specifications

The following table lists the supported protocol specifications for the ZeBu PCIe Transactor:

Table 2 Supported PCIe Protocol Specifications

Protocol Specification	**Revision**
PCI Express® Base Specification	6.0

Feedback

Table 2 Supported PCIe Protocol Specifications (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 PCIe Protocol Features

The following table lists the supported PCIe protocol features:

Table 3 Supported PCIe Protocol Features

Category	Features
Device Configuration	Root Complex, Endpoint
Link Rate (GT/s)	2.5, 5.0, 8.0, 16.0, 32.0, 64.0
PIPE Optional Features	RxStandby/RxStandbyStatus Handshake
	RxValid synchronous to RxCLK in SerDes PIPE mode
Sideband Signals	CLKREQ#, WAKE#, PERST#
Flit Features	FLIT support for 8b/10b encoding and 128b/130b encoding
	CRC detection
	Flit Retry
	FEC
	FLIT mode TLP header, DLLP, NOP, NOP2
Fault Isolation	Advanced error reporting (AER)
Transaction Layer	Flit Mode and Non-Flit mode Transaction Layer Packets
	Interrupts (MSI, MSI-X, INTx)
	Process Address Space ID (PASID) & PASID Translation
	Atomic Operations
	ID-Based Ordering

Feedback

Table 3 Supported PCIe Protocol Features (Continued)

Category	Features
	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)
Data Link Layer	Flow Control Credit (FCC)
	Ack/Nak
Physical Layer	Equalization procedure
	Electrical Idle
	LTSSM
	L0p
	Lane Reversal
	Support for down configuration
	Precoding
	Gray Coding
	Hot-plug
	Hot Reset, Cold Reset, Link Disable
	SRIS and SRNS Clocking Architecture
Power Management	ASPM L1 and L0s
	PCI-PM L1
	L1 Sub-states

Table 3 Supported PCIe Protocol Features (Continued)

Category	Features
	L2
Security	IDE
SR-IOV	Support for 1 VF
Alternative Routing-ID Interpretation (ARI)	Support for 1 PF
Address Translation Service (ATS)	ATS Invalidation
	Page Request Message

Supported Transactor Features

The following table lists the supported ZeBu PCIe transactor features:

Table 4 Supported PCIe Transactor Features

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

Limitations

This version of the ZeBu PCIe transactor, when configured as either RC or EP, has the following limitations:

- Polarity Inversion is not supported.
- Lane Margins at Receiver is not supported.
- Only one hardware function is supported (applicable for Transactor configured as EP).
- Only one Virtual Function is supported (applicable for Transactor configured as EP).

Feedback

- Only one Virtual Channel is supported.
- Loopback and Compliance features are not supported.
- For all speeds in Flit Mode (Gen6 driver), Max_Payload_Size (MPS) is limited to 1024 bytes (256 DW) only.
- Supported combinations of PIPE width and PCLK_Rate at each speed are available in Table 14. Remaining configurations, not listed in Table 14, are currently not supported.
- PPM Testing is not supported as Elastic Buffers are not implemented.
- Since PCIe Lane Model provides limited PHY functionality, it has the following limitations in supporting Equalization feature:
  - LF, FS, and Preset Coefficients of PHY cannot be changed dynamically.
  - Custom LF, FS, and Preset Coefficients values can be provided through module parameters, and they remain same for speeds from Gen3 to Gen6.
- With PIPE 4.x connection, the functionality to connect the following signals is not available:

Table 5 Signal List

Transactor	PCIe Wrapper
mac_phy_asyncpowerchangeack	asyncpowerchangeack_from_dut
mac_phy_txcommonmode_disable	txcommonmode_disable_from_dut
mac_phy_rxelecidle_disable	rxelecidle_disable_from_dut
mac_phy_txcompliance	txcompliance_from_dut
-	rxpolarity_from_dut

The functionality of below PIPE signals (with Legacy Pin interface) / Message Bus register fields (with Low Pin Count interface) is not supported:

- TxSwing
- EncodeDecodeBypass
- BlockAlignControl

2

Installing and Validating the Transactor Package

This section explains the following topics:

Licensing

You need the following FLEXlm licenses, based on the PCIe speed, to use the ZeBu PCIe transactor.

Table 6 PCIe Transactor Licensing

License Feature	Description
hw_xtormm_pcie6	Allows to use PCIe speeds up to Gen6
hw_xtormm_pcie5	Allows to use PCIe speeds up to Gen5
hw_xtormm_pcie	Allows to use PCIe speeds up to Gen4

Note:

- Each license token is applicable for one transactor instance.
- If the hw_xtormm_pcie6 license is not available, call the the restrictGen6() API in the testbench to prevent the checkout of Gen6 license and the resulting error. Refer to Table 24 for restrictGen6() and other license checkout API details.

Installation

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

Feedback

Package Directory Structure

After the ZeBu PCIe transactor is successfully installed, it consists of the following files:

Table 7 ZeBu PCIe Directory Structure

Directory	Sub directory/Files	Description
doc	ZeBu_XTOR_PCIe_svs_UM.pdf	PCIe Transactor User Manual
	ZeBu_VS_UM.pdf	Zebu Vertical Solutions User Manual
	html/index.html	HTML documentation
example/src	bench	testbench related files
	dut	wrappers files
	env	environment files
example/zebu	Makefile, Readme	Makefile: Makefile contains the compilation flow for UC using the UTF wrapper file. Readme files: Contains more information of the example.
include	xtor_pcie_svs.hh	APIs for configuring transactor
	xtor_pcie_svs_defines.hh	Enums/defines/struct definition
	xtor_wrapper_pcie_svs.hh	APIs for configuring wrapper/transactor
	xtor_wrapper_pcie_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
vlog/vcs	xtor_pcie_svs.sv	Driver
	xtor_pcie_svs_lanes_model.sv	PCIe Lane model
libABI1	libxtor_pcie_svs.so	.so file of PCIe transactor
misc	pcie_wrapper_svs.sv	Wrapper module

The PCIe Monitor is installed using the separate package (*$ZEBU_IP_ROOT/monitor_pcie_svs*). For more information, see PCIe Monitor.

Validating the Installed Package

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

This section consists of the following subsections:

- Setting the Environment
- Running the Example
- Validating the Run

Setting the Environment

Ensure that the following environment variables are set correctly before running the example:

Table 8 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 Serversystem: *../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_pcie_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 at Gen5 and sets TG_COMP to 16x32b_b2b.

   For compiling Gen6 test, specify the following:

   ```
   $ make compile TG_COMP=16x80b_b2b
   ```

Set the Makefile target XTOR_PCIE_SVS_ARI_ENA to 1 during compilation to enable ARI capability, which allows the support of up to 256 Physical Functions in the EP transactor.

3. 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=xtor_pcie_svs_test
   ```

   zRci Testbench:

   ```
   $ make run ZRCI=1 TG_RUN=xtor_pcie_svs_test
   ```

Note:

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

$ make compile SIMULATOR=1

$ make run SIMULATOR=1 TG_RUN = xtor_pcie_svs_test

The deliverable example uses VCS slave mode in simulation. Therefore, interactive mode of Verdi is not supported.

Validating the Run

The following message is displayed after a successful run:

```
-- ZeBu : zServer : End of run :
-- ZeBu : zServer : Unit 0 masterClk cycle counter : 19,444,300
 (0x0128b24c)
-- ZeBu : zServer : Unit 0 driverClk cycle counter : 1,346,871
 (0x00148d37)
-- ZeBu : zServer : Unit 0 driverClkDut cycle counter : 1,346,871
 (0x00148d37)
-- ZeBu : zServer : Unit 0 tickClk edge counter : 327,014 (0x0004fd66)
-- ZeBu : zServer : Unit 0 globalTime counter : 163,507,000 (0x09beeb38)
```

Run logs generated during a ZeBu run can be found under rundir_*.zebu/ xtor_pcie_svs_test.log. See Troubleshooting section for more information.

3

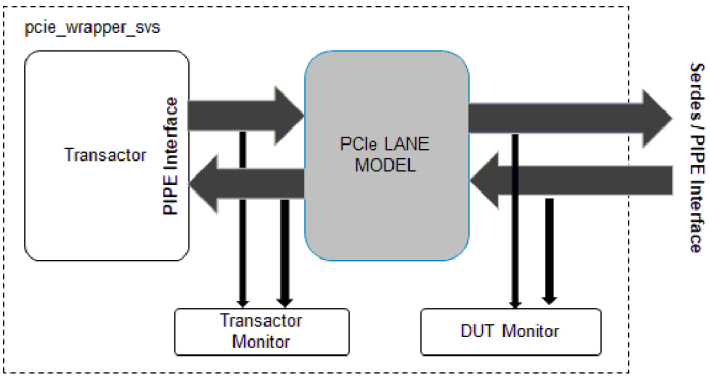
Introducing PCIe Wrapper

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

Components of PCIe Wrapper

The *pcie_wrapper_svs* is a wrapper module that enables ease of usage of the PCIe Transactor by encapsulating PCIe Transactor's connection with the PCIe Lane Model and Monitor.

Figure 2 PCIe Wrapper Architecture



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

Table 9 PCIe Wrapper Components

Wrapper Component	Module Name
PCIe Transactor	xtor_pcie_svs

Feedback

Table 9 PCIe Wrapper Components (Continued)

Wrapper Component	Module Name
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_pcie_svs/misc/pcie_wrapper_svs.sv for more details of the module.

You can enable/disable the monitors using the IMPLEMENT_MONITOR_XTOR and IMPLEMENT_MONITOR_DUT parameters. You can set only one of these parameters at a time.

PCIe Wrapper Parameters

The following table lists the pcie_wrapper_svs parameters.

Table 10 The pcie_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 Gen5), 8 (Gen6)	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

Feedback

Table 10 The pcie_wrapper_svs Parameters (Continued)

Parameter Name	Default Value	Possible Value(s)	Description
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
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
PCIE_MAXPAYLOAD	1024	(in DW): 32, 64, 128, 256, 512, 1024	PCIe 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	5	4, 5, 6	PIPE version supported by DUT
DUT_RXSTANDBY_ACTIVE _VALUE	1	0: Not Supported 1: SupportedSet the value of the parameter based on the value supplied by DUT.	Controls RxStandby/RxStandbyStatus handshake support of DUT.

Feedback

Table 10 The pcie_wrapper_svs Parameters (Continued)

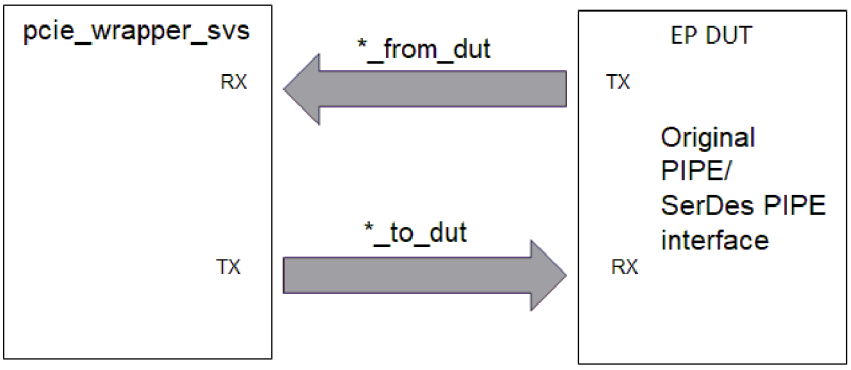
Parameter Name	Default Value	Possible Value(s)	Description
LOCAL_FS	30h	Refer PCIe Base Specification Revision 6.0 Section- 4.2.4.1 (Rules for Transmitter Coefficients)	Local FS value for the PHY. This is applicable for speeds from Gen3 to Gen6.
LOCAL_LF	18h		Local LF value for the PHY. This is applicable for speeds from Gen3 to Gen6.
LOCAL_TXPRESETCOEFFICIENTS	0C900h		LocalTxPresetCoefficients[17:0] value defined as follows: [5:0] C-1 [11:6] C0 [17:12] C+1. This is applicable for speeds from Gen3 to Gen6.

PCIe Wrapper Signal List

When using the pcie_wrapper_svs module, the DUT interface signaling *_from_dut/ *_to_dut connection is established, irrespective of whether the DUT is EP or RC.

The following figure illustrates a sample connection between the PCIe wrapper and DUT:

Figure 3 PCIe 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

Feedback

Common Signals for Original and SerDes PIPE

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

Table 11 Common Signals for Original and SerDes PIPE

Name	Size (in bits)	Direction	Description
Clock and Reset Shared Signals			
perst_n	1	Input	Active Low Fundamental Reset signal.
fastest_pcie_clk	1	Input	Global clock that 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 perst_n.
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	43	Input	PHY Rate signaling from DUT.
powerdown_from_dut	4	Input	Powers up or down the transceiver.
wake	1	Output	Sideband signal to implement exit mechanism from L2 power management state.
Per Lane Signals			
pclkchangeok_to_dut	1	Output	Optional. Clock change acknowledgment grant.

Feedback

Table 11 Common Signals for Original and SerDes PIPE (Continued)

Name	Size (in bits)	Direction	Description
pclkchangeack_from_dut	1	Input	Optional. Clock change acknowledgment request. See Note.
rxstandby_from_dut	1	Input	Determines if the PHY Rx is active when the PHY is in P0 or P0s.0: Active1: StandbyUsed when the parameter DUT_RXSTANDBY_ACTIVE_VALUE is set to 1.
rxstandbystatus_to_dut	1	Output	Status to indicate RxStandby state: • 0: Active • 1: StandbyUsed when the parameter DUT_RXSTANDBY_ACTIVE_VALUE is set to 1.
txdetectrx_from_dut	1	Input	Indicates PHY to begin a receiver detection operation.
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: 1PIPE 5.1 and above: 4	Input	Forces Tx output to electrical idle when asserted.
phystatus_to_dut	1	Output	Indicates 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 anElectrical 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.

Feedback

Table 11 Common Signals for Original and SerDes PIPE (Continued)

Name	Size (in bits)	Direction	Description
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].
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	Indicates FS value provided by the link partner.
lf_from_dut	6	Input	Indicates the LF value provided 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.

Feedback

Table 11 Common Signals for Original and SerDes PIPE (Continued)

Name	Size (in bits)	Direction	Description
locallf_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.
linkevaluationfeedbackdirectionchan ge_ 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 12 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, 16.0 GT/s, and 32.0 GT/s. Allows the MAC to tell the PHY that the starting byte for a 128b block.
txsyncheader_from_dut	2	Input	Only used at 8.0 GT/s, 16.0 GT/s, and 32.0 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, 16.0 GT/s, and 32.0 GT/s. Allows the PHY to tell the MAC that the starting byte for a 128b block.

Feedback

Table 12 Original PIPE Only Signal List (Continued)

Name	Size(in bits)	Direction	Description
rxsyncheader_to_dut	2	Output	Only used at 8.0 GT/s, 16.0 GT/s, and 32.0 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 13 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.

Supported PCIe Wrapper Configurations

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

The following table lists the supported PCIe configurations:

Table 14 Supported PCIe Configurations

Generation / Rate	XTOR		DUT		TxDataValid/RxDataValid Strobe Rate
	PCLK (100 kHz)	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

Feedback

Table 14 Supported PCIe Configurations (Continued)

Generation / Rate	XTOR		DUT		TxDataValid/RxDataValid Strobe Rate
	PCLK (100 kHz)	Width (SerDes) (bits)	PCLK (100 kHz)	Width (SerDes) (bits)	
			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
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

Feedback

Table 14 Supported PCIe Configurations (Continued)

Generation / Rate	XTOR		DUT		TxDataValid/RxDataValid Strobe Rate
	PCLK (100 kHz)	**Width (SerDes) (bits)**	**PCLK (100 kHz)**	**Width (SerDes) (bits)**	
Gen6 (64.0 GT/s)	10000	N/A (80)	40000	N/A (20)	N/A
			20000	N/A (40)	N/A
			10000	N/A (80)	N/A

4

Integrating PCIe Transactor and DUT

This section explains the following topics:

Topologies and Connections

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

Table 15 PCIe Transactor Topologies

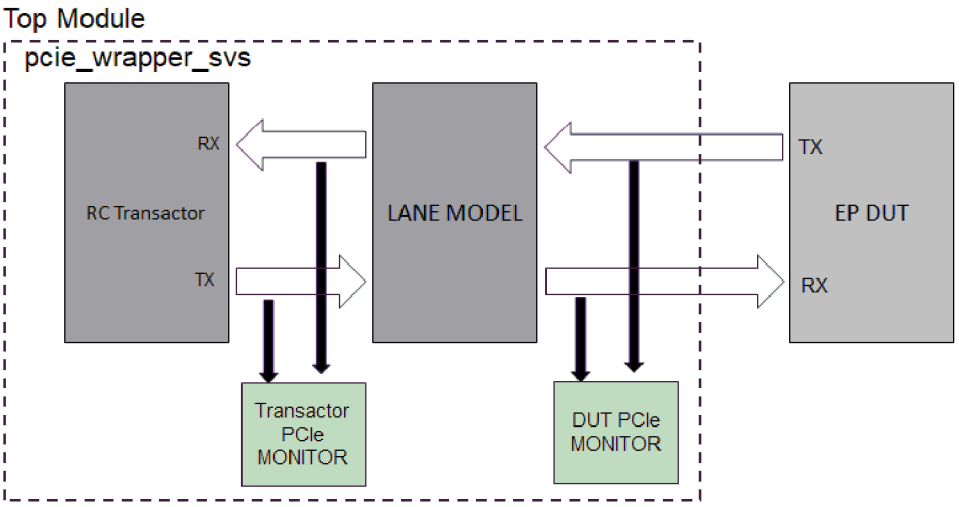
Topology	*Description*	*Reference Figure*
RC Transactor and EP DUT	RC instance of pcie_wrapper_svs connected with EP DUT	See Figure 4
EP Transactor and RC DUT	EP instance of pcie_wrapper_svs connected with RC DUT	See Figure 10

RC Transactor and EP DUT

For this topology, the pcie_wrapper_svs parameter *IS_ROOTPORT* is set to 1.

Feedback

Figure 4 RC Transactor with EP DUT



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

1. In the top-level wrapper module, instantiate the following mandatory components:
   - PCIe Wrapper as RC: pcie_wrapper_svs
   - EP DUT
2. Connect the pcie_wrapper_svs and DUT according to the reference figures in the below table:

Table 16 PCIe 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

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.

Feedback

3. Connect zceiClockPort or RTL clock for Clock as described in the Connecting Clocks section.

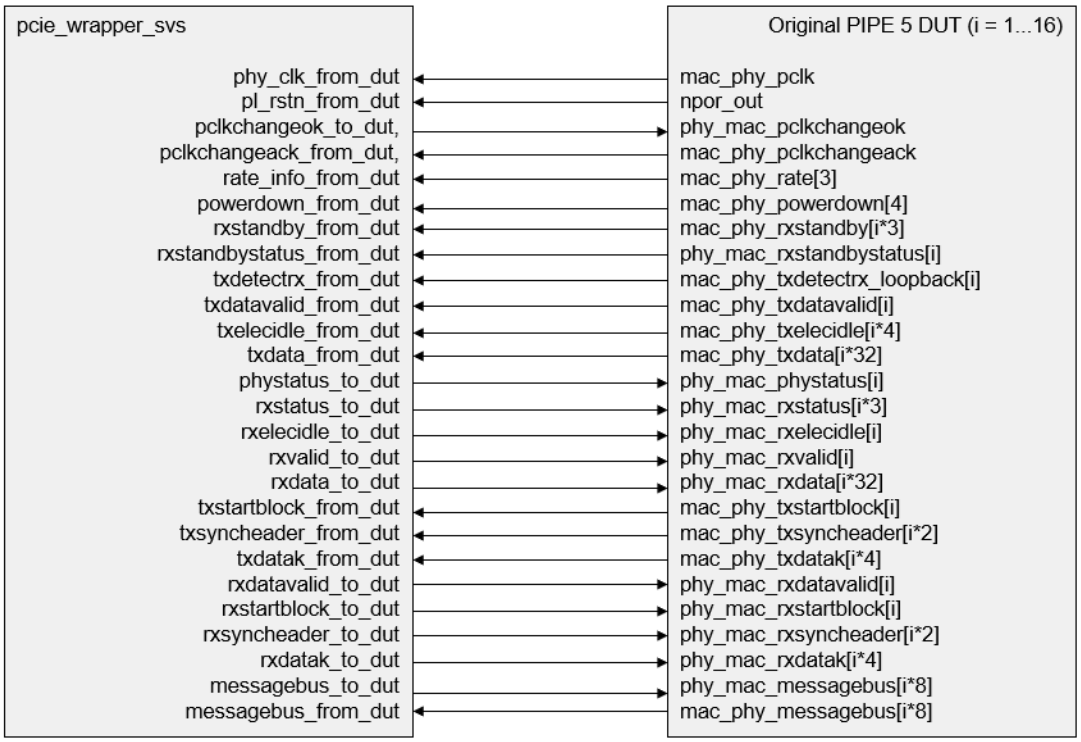
   Note:

   For the hardware pin connection, there is a reference example provided within this package: $ZEBU_IP_ROOT/xtor_pcie_svs/example/src/dut/wrapper.v.

Original PIPE Interface with DUT

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

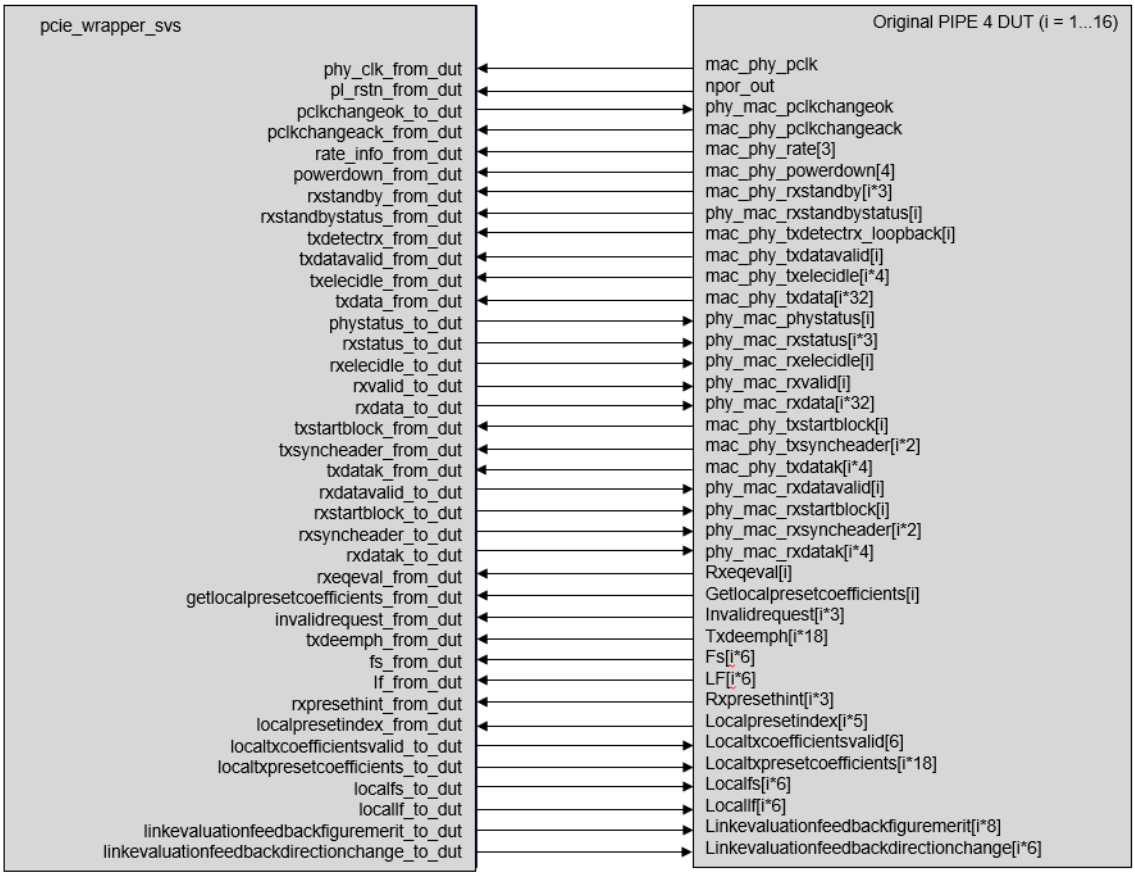
Figure 5 Original PIPE 5 DUT to Wrapper Connection



Feedback

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

Figure 6 Original PIPE 4 DUT to Wrapper Connection

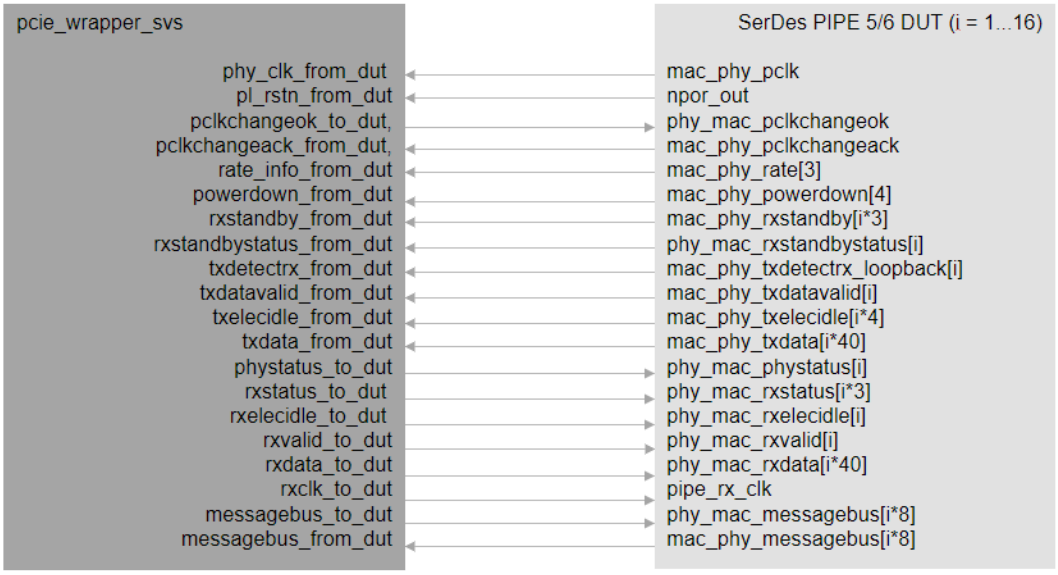


Feedback

SerDes PIPE Interface with DUT

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

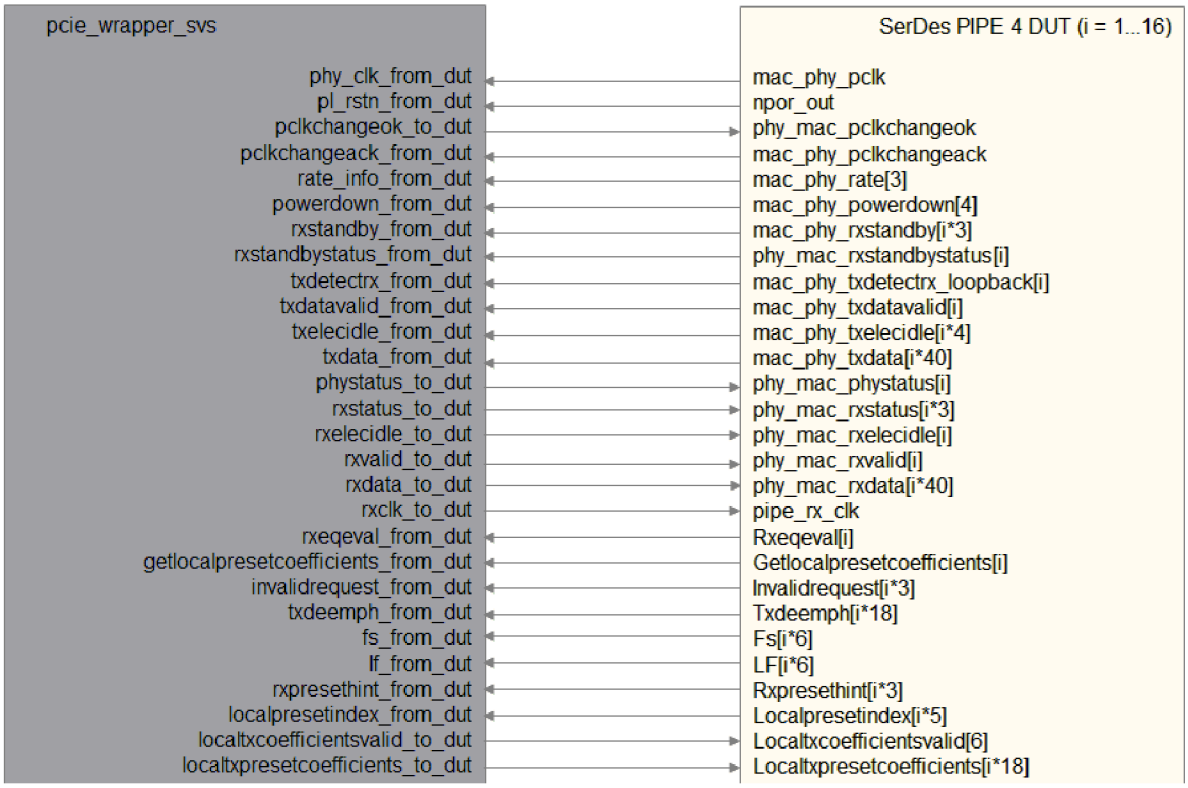
Figure 7 SerDes PIPE 5/PIPE 6 DUT to Wrapper Connection



Feedback

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

Figure 8 SerDes PIPE 4 DUT to Wrapper Connection

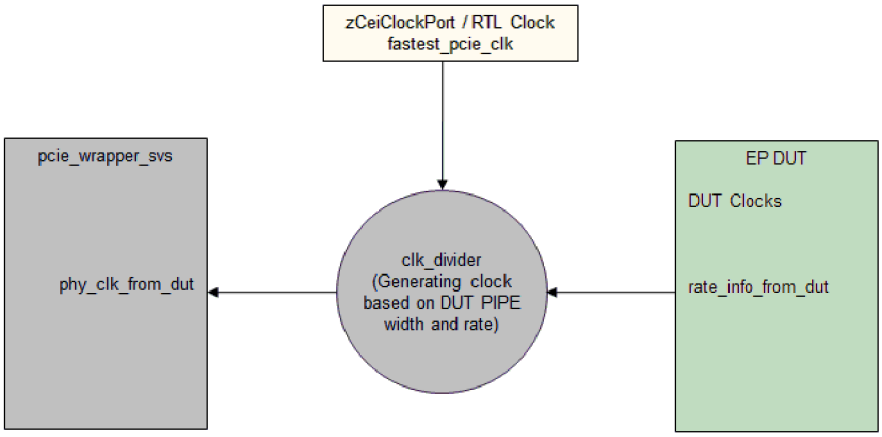


Connecting Clocks

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

Feedback

Figure 9 Connecting the phy_clk_from_dut Clock to a Primary Clock



The pcie_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 pcie_wrapper_svs module connected to 8bits (Gen1), 8 bits (Gen2), 16 bits (Gen3), 32 bits (Gen4), and 32 bits (Gen5). Original PIPE DUT are as follows:

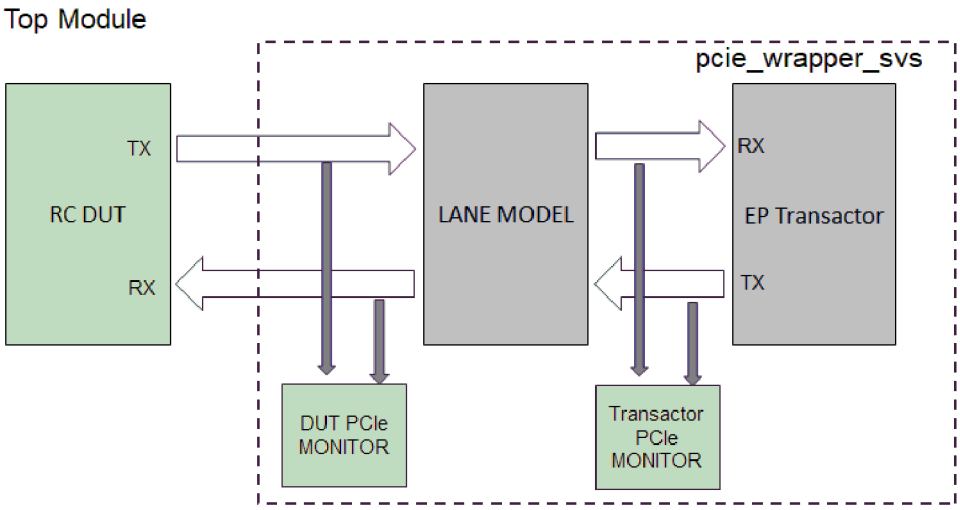
- phy_clk_from_dut: 250 MHz at Gen1 rate.
- phy_clk_from_dut: 500 MHz at Gen2 rate.
- phy_clk_from_dut: 500 MHz at Gen3 rate.
- phy_clk_from_dut: 500 MHz at Gen4 rate.
- phy_clk_from_dut: 500 MHz at Gen5 rate.

EP Transactor and RC DUT

For this topology, the pcie_wrapper_svs parameter *IS_ROOTPORT* is set to 0.

Feedback

Figure 10 EP Transactor with RC DUT



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

1. In the top-level wrapper module, instantiate the following mandatory components:
   - PCIe Wrapper as EP: pcie_wrapper_svs
   - RC DUT
2. Connect the pcie_wrapper_svs and DUT according to the reference figures in Table 16.
3. Connect the zceiClockPort or RTL clock for Clock as shown in Connecting Clocks section.

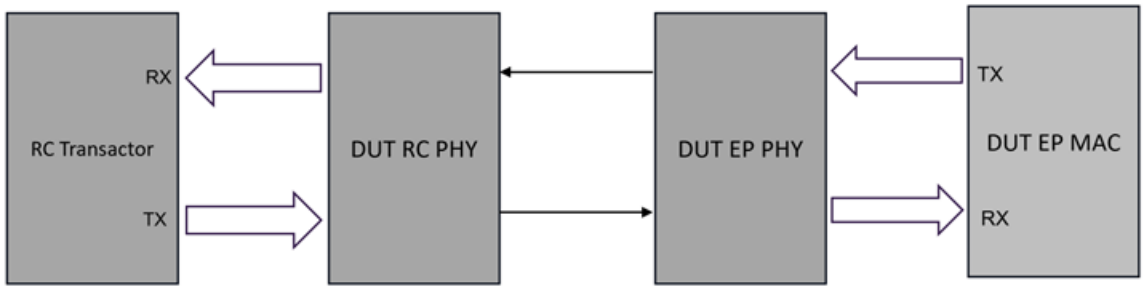
RC Transactor and PHY DUT

For this topology, the `xtor_pcie_svs` module, `device_type` signal, is set to 4. RC 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

Feedback

Figure 11 RC Transactor and PHY DUT



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

1. In the top-level wrapper module, instantiate the following mandatory components:
   - PCIe Transactor as RC: xtor_pcie_svs
   - PHY DUT
2. Connect the xtor_pcie_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:

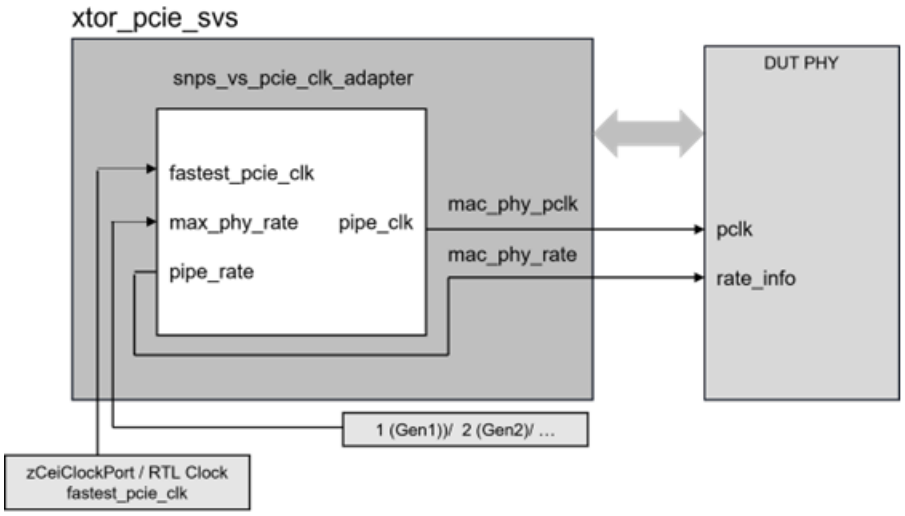
Table 17 PCIe Transactor and PHY DUT connection

DUT PIPE Mode	Reference Figure
Original PIPE	Figure 13
SerDes PIPE	Figure 14

Feedback

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_pcie_svs parameter:

Table 18 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.

Feedback

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:

Table 19 Original and SerDes PIPE Signals

Name	Size (in 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. 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.

Feedback

Table 19 Original and SerDes PIPE Signals (Continued)

Name	Size (in bits)	Direction	Description
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.
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.

Feedback

Table 19 Original and SerDes PIPE Signals (Continued)

Name	Size (in bits)	Direction	Description
mac_phy_txcommonmode_disable	1	Output	Used for to control L1.2 state transition. As of now, its functionality is not present.
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.
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.

Feedback

Table 19 Original and SerDes PIPE Signals (Continued)

Name	Size (in bits)	Direction	Description
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
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.

Original PIPE Only Signal List

The following table lists the original PIPE only signals in the transactor:

Table 20 Original PIPE Only Signal List

Name	Size (in bits)	Direction	Description
Per Lane Signals			

Feedback

Table 20 Original PIPE Only Signal List (Continued)

Name	Size (in bits)	Direction	Description
mac_phy_txstartblock	1	Output	Only used at 8.0 GT/s, 16 GT/s, and 32 GT/s. Allows the MAC to tell the PHY that the starting byte for a 128b block.
mac_phy_txsyncheader	2	Output	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.
mac_phy_txcompliance	1	Output	Used when transmitting the PCIe* compliance pattern. As of now, its functionality is not present.
mac_phy_txdatak	4	Output	Data/Control for the symbols of Tx data.
mac_phy_txdata	32	Output	Parallel data input bus for Tx differential pair.
phy_mac_rxdatavalid	1	Input	Allows the PHY to instruct the MAC to ignore the data interface for one clock cycle.
phy_mac_rxstartblock	1	Input	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.
phy_mac_rxsyncheader	2	Input	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.
phy_mac_rxdatak	4	Input	Data/Control for the symbols of Rx data.

Feedback

Table 20 Original PIPE Only Signal List (Continued)

Name	Size (in bits)	Direction	Description
phy_mac_rxdata	32	Input	Parallel data input bus for Rx differential pair.

SERDES PIPE Only Signal List

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

Table 21 SerDes PIPE Only Signal List

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:

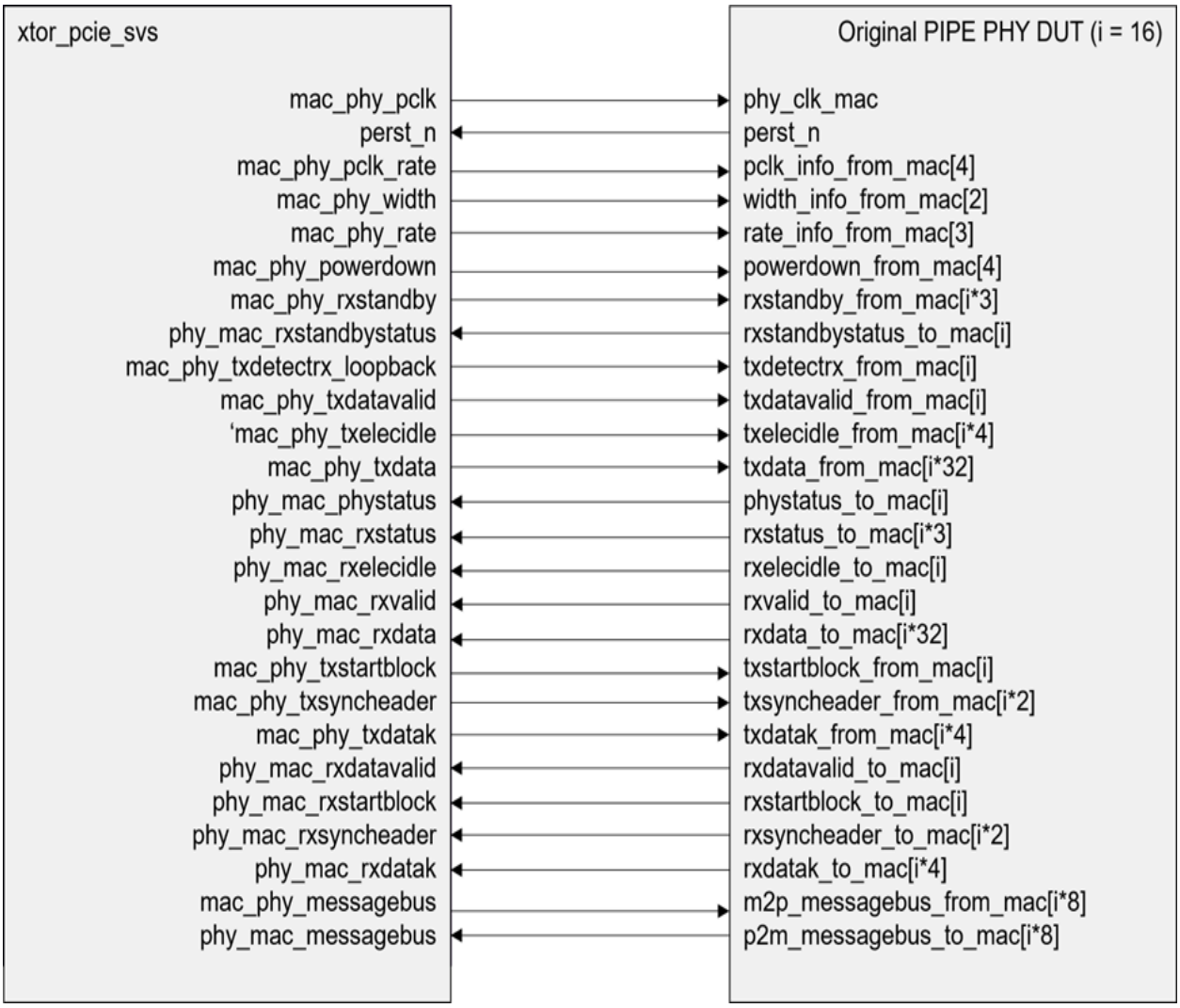
- Original PIPE Interface with PHY DUT
- SerDes PIPE Interface with PHY DUT

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

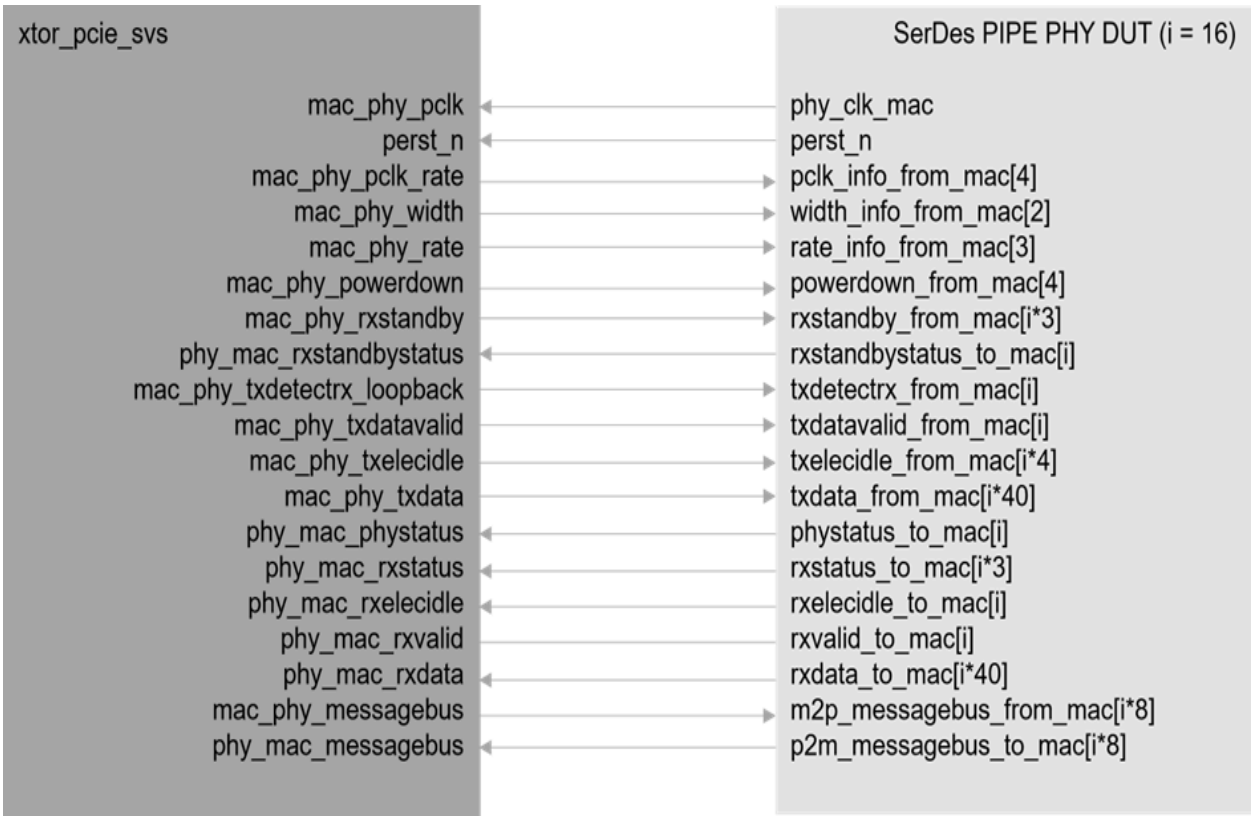


Feedback

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



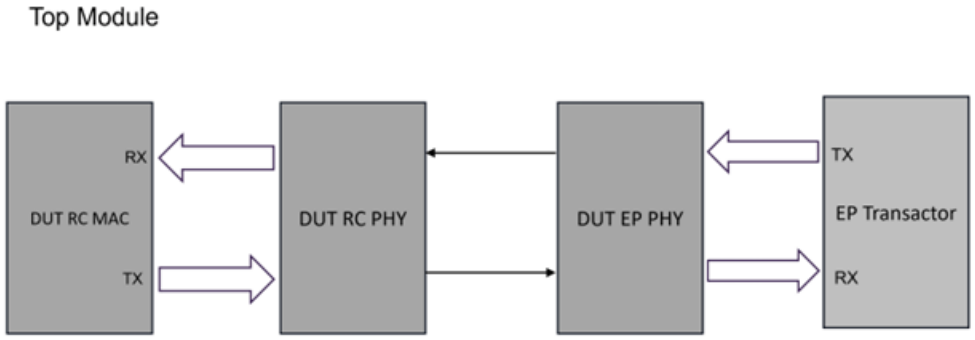
EP Transactor and PHY DUT

For this topology, the *xtor_pcie_svs*module,*device_type* signal, is set to 0. EP 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

Feedback

Figure 15 EP Transactor and PHY DUT



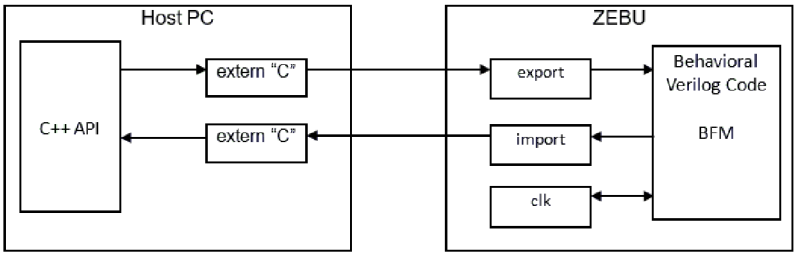
To create a connection between the PHY DUT and the EP transactor, perform the following steps:

1. In the top-level wrapper module, instantiate the following mandatory components:
   - PCIe Transactor as EP: xtor_pcie_svs
   - PHY DUT
2. Connect the xtor_pcie_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 17 .
3. Connect zceiClockPort or RTL clock for Transactor as explained in RC Transactor and PHY DUT.

Initialization

The following figure illustrates the process of initializing a transactor:

Figure 16 Transactor Initialization



The following are the steps to initialize the transactor:

1. Include necessary PCIe header files along with other ZeBu/Zemi3 header files.

```
#include "xtor_pcie_svs.hh"
…..
using namespace XTOR_PCIE_SVS;
```

2. Declare and initialize PCIe driver handles (xtor_pcie_svs) in the main method.

```
int main(int argc, char * argv[]) {
xtor_pcie_svs* rc;
xtor_pcie_svs::Register();
ZEMI3Manager *zemi3 = ZEMI3Manager::open(database.c_str());
….
zemi3->buildXtorList((database+"/xtor_dpi.lst").c_str());
zemi3->init();
rc=static_cast<xtor_pcie_svs*>(Xtor::getNewXtor(xtor_pcie_svs::getXtor
TypeName(), XtorRuntimeZebuMode))
```

Note:

Note: Second argument of getNewXtor() API is set as XtorRuntimeMode_t::XtorRuntimeZebuMode in the Emulation mode, and as XtorRuntimeMode_t::XtorRuntimeSimulationMode in the Simulation mode.

3. Set the transactor parameters to configure the transactor:

```
if
 (strncmp(xtor->getDriverInstanceName(),"wrapper.pcie_driver_rc",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->setLog ((char*) (xtor->getDriverInstanceName ()), true);
rc = xtor;
xtor_tb_rc = new pcie_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.

```
uint32_t exit_code= 1;
while (exit_code != 0) {
if (!Xtor::AllXtorInitDone()) {
*exit_code=1;
} else {
*exit_code=0;
}
}
```

Configuration

This section explains how to configure the transactor in the following sections:

- Configuration Parameters
- Configuration APIs

Configuration Parameters

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

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

Table 22 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 (that is, 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 (that is, 1ms become 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.

Table 22 Configuration Parameters (Continued)

Parameters	Default Value	Details
PCIE_MAXPAYLOAD	4096	Configures the max payload capability for the device, that is, 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
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_SUBID	0	PCIE Subsystem ID [31:16] and Subsystem Vendor ID [15:0] of the transactor. This parameter is only applicable for Type 0 Config Space, that is, EP.
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. Default value is:6, if Gen6 license check is ok.5, if Gen5 license check is ok.Otherwise, 4.

Table 22 Configuration Parameters (Continued)

Parameters	Default Value	Details
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.
XTOR_NB_RESET_ DE_ASSERTED	0	Indicates the number of reset de-asserted events before starting the Global Scheduler.
PCIE_TLP_BYPASS	false	Enables the Testbench to override BAR settings. Valid for EP transactor only. Valid values: true/false When true, configuration requests from RC are routed to application interface rather than core. So, the completions for configuration requests from the testbench are sent.
PCIE_DROP_ABORT_ TLP	true	Controls if erroneous TLPs are dropped or transferred to the application interface. Valid values: true, false When true, received TLPs with error or exit status are dropped. When false, erroneous TLP are transferred to the application interface.
PCIE_XTOR_CLOCK_ CHECKER_DISABLE	false	Disables the clock checker logic that checks the phy_clk_dut w.r.t fastest_pcie_clk ratio at Gen1.
PCIE_ASPML1	false	Controls the ASPM L1 support. Valid values: true, false Set the value to true to enable ASPM L1 support.
PCIE_ASPML1_1	false	Controls the ASPM L1.1 support. Valid values: true, false Set the value to true to enable ASPM L1.1 support.
PCIE_ASPML1_2	false	Controls the ASPM L1.2 support. Valid values: true, false Set the value to true to enable ASPM L1.2 support.
PCIE_MAXLINKWIDTH	16	Controls the maximum link width for the transactor. Valid values: 1, 2, 4, 8, 16 for x1, x2, x4, x8, x16 link correspondingly. This parameter is changed when there are unused lanes in the system.

Table 22 Configuration Parameters (Continued)

Parameters	Default Value	Details
PCIE_PART_LANES_RXEI_EXIT_ENA	false	Controls the link up with partially active lane number. Valid values: true, false When true, it allows LTSSM transition from `Polling.Active` to `Polling.Configuration` state based on Rx 8 TSs on any lanes which are Rx EI exit too after 24ms timeout.

Note:

Other configuration parameters are available in the file $ZEBU_IP_ROOT/xtor_pcie_svs/doc/html/index.html

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

- PCIE_EQ_PHASE23_ENABLE
- PCIE_NO_EQ_NEEDED_DISABLE
- PCIE_EQ_BYPASS_HIGH_RATE_DISABLE

Set the above 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_PHASE23_ENABLE ", "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 23 Transactor Equalization Settings

PCIE_NO_EQ_NEEDED_DISABLE	PCIE_EQ_BYPASS_HIGH_RATE_DISABLE	PCIE_EQ_PHASE23_ENABLE	Final Equalization Mode at Target Link Speed	
			Gen5	Gen6
false	false	false	G1 -> G5 (NO EQ)	G1 -> G6 (NO EQ)
false	false	true	G1 -> G5 (NO EQ)	G1 -> G6 (NO EQ)

Table 23 Transactor Equalization Settings (Continued)

PCIE_NO_EQ_NEEDED_DISABLE	PCIE_EQ_BYPASS_HIGH_RATE_DISABLE	PCIE_EQ_PHASE23_ENABLE	Final Equalization Mode at Target Link Speed	
			Gen5	Gen6
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 -> G5 -> G6 (EQ01)
true	false	true	G1 -> G5 (EQ0123)	G1 -> G5 -> G6 (EQ0123)
true	true	false	G1 -> G3 (EQ01) -> G4 (EQ01) -> G5 (EQ01)	G1 -> G3 (EQ01) -> G4 (EQ01) -> G5 (EQ01)-> G6 (EQ01)
true	true	true	G1 -> G3(EQ0123) -> G4 (EQ0123) -> G5 (EQ0123)	G1 -> G3(EQ0123) -> G4 (EQ0123) -> G5 (EQ0123) -> G6 (EQ123)

Configuration APIs

The following table lists the key configuration APIs for the PCIe transactor:

Table 24 Transactor Configuration APIs

Name	Description
PCIeXtorCfgWrite	Backdoor write of transactor PCIe Configuration Space
PCIeXtorCfgRead	Backdoor read of transactor PCIe Configuration Space
wait_for	Blocking method to wait for a particular event to happen
analyzerStart	Start PCIe Protocol Analyzer
analyzerStop	Stop PCIe Protocol Analyzer

Table 24 Transactor Configuration APIs (Continued)

Name	Description
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.
pushTxTlp	Sends a TLP on the Link
wait_for_cycle	Waits for design clock cycles.
readCfg0	Frontdoor Configuration Read 0
getDriverInstanceName()	Retrieves the transactor instance name present in environment. It can be used in comparing the Xtor instance name and provide the configurations according to instance mode.
getCapPtr()	Returns the address pointer to the PCIe capability structure.
setTag()	Set the Tag value for the TLP.
readCfg0	Creates Configuration Read Type 0 request TLP
writeCfg0	Creates Configuration Write Type 0 request TLP
linkWidthChange()	Used to change the link width through the state transition L0 → Recovery → Configuration → L0 in Non-Flit mode.
setTxDly()	Configures the delay in fastest_pcie_clock cycles for the TLP when it is applied to TL layer.
initiate_L0p()	Used to change the link width through L0p in Flit mode.
License Checkout APIs	
restrictGen6()	API to restrict checking out Gen6 license in case Gen6 is not supported by device. It must be called before new constructor of the transactor.
restrictGen5()	API to restrict checking out Gen5 license in case Gen5 is not supported by device. It must be called before new constructor of the transactor.
restrictSubSys()	API to restrict checking out PCIe Sub-System license (hw_xtormm_pcie_subsys). It must be called before new constructor of the transactor.

Table 24 Transactor Configuration APIs (Continued)

Name	Description
allowSubSys()	API to allow checking out PCIe Sub-System license (hw_xtormm_pcie_subsys). It must be called before new constructor of the transactor.

Note:

For more information on ther configuration APIs, see $ZEBU_IP_ROOT/xtor_pcie_svs/doc/html/index.html.

The following are some of the examples of using the configuration APIs:

- Enabling SRIS in the Transactor
- Enabling Backdoor Access
- Sending TLPs
- Hot Plug Feature

Enabling SRIS in the Transactor

By default, the Separate Reference Clocks with Independent Spread Spectrum Clocking (SRIS) feature is disabled in PCIe 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 the following, after all the transactors are initialized.

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

Table 25 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 and 375 Blocks)
true	Short (less than 154 Symbol times)	Short (less than 38 Blocks)

Feedback

Enabling Backdoor Access

Enable the backdoor access as shown below:

```
ep->PCIeXtorCfgWrite(TYPE0_STATUSCOMMAND, 0x00000007);
cfg_reg_read_value = ep->PCIeXtorCfgRead(TYPE0_BAR1);
cfg_reg_write_value |= 0x40000000;
ep->PCIeXtorCfgWrite (TYPE0_BAR1, cfg_reg_write_value);
```

Sending TLPs

Send the TLPs for read and write transactions as shown below:

```
pcie_svs_tlp* tlp;
tlp = new pcie_svs_tlp;
tlp->readCfg0(TYPE0_BAR1>>2,0x1,0,0);
//cfg_reg_write_value get value from Completion and modify
tlp->writeCfg0(cfg_reg_write_value, TYPE0_BAR1>>2,0x1,0,0);
Memory Read /Memory Write
// initiating memory write transaction.
len=5 ;
uint32_t addr = xtor_tb_rc->mem32_addr(len);
tlp->setTag(xtor_tb_rc->rc_tag);
tlp->writeMem32(addr,xtor_tb_rc->_data,len) ;
rc->pushTxTlp(tlp);
tlp->printTLP(Full);
xtor_tb_rc->rc_tag++;
// Initiating memory read transaction.
len=5 ;
tlp->setTag(TAG_KILL);
tlp->readMem32(addr,len) ;
tx_tag = tlp->getTag();
rc->pushTxTlp(tlp);
```

Note:

To initiate TLPs simultaneously from multiple threads, each pushTxTlp()API call needs to be protected using mutex lock()and unlock()functions.

Hot Plug Feature

RC Transactor and EP DUT

In case of an RC transactor and EP DUT, move the EP to Hot-Reset and then enable the Hot-Plug feature. To do so, perform the following steps:

1. Move the transactor to the Hot Reset State as shown below:

```
uint32_t rdata;
rc->print("---- RC XTOR: initializing HOT reset by Setting Secondary
 bus reset field of bridge control register \n");
rdata =
 rc->PCIeXtorCfgRead(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE);
rdata |= 0x00400000;
```

Feedback

```
rc->PCIeXtorCfgWrite(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE,rdat
a);
rc->print("----RC XTOR: wait for LTSSM State =STATE_HOT_RESET");
rc->wait_for((Ltssm_state_t)STATE_HOT_RESET);
// disabling secondry bus reset bit in bridge control register.
rdata =
 rc->PCIeXtorCfgRead(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE);
rc->print("---- RC XTOR: default value of Bridge Control register=%0x
 \n ",rdata);
rdata &= 0xffbfffff;
rc->PCIeXtorCfgWrite(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE,rdat
a);
```

2. Once device reaches the Hot-Reset state, wait for Detect and apply the Hardware reset to the transactor, as show below:

```
rc->print("----RC XTOR: wait for LTSSM State =STATE_DETECT_QUIET");
rc->wait_for(STATE_DETECT_QUIET);
rc->print("----RC XTOR Applying Hw reset\n");
rc->reqFor(hw_reset_n, true);
rc->wait_for_cycle(10);
rc->reqFor(hw_reset_n, false);
```

3. Apply the initial configuration to the transactor and wait for the link-up after the hardware reset, as shown below:

```
rc->print("----RC XTOR cfg xtor after detect \n");
rc->while (!Xtor::AllXtorInitDone());
rc->print("----RC XTOR Waiting for L0\n");
rc->wait_for(linkup_done);
```

EP Transactor and RC DUT

In case of RC DUT and EP Transactor, EP Transactor cannot initiate the Hot Reset transition to enable the Hot-Plug feature. Applying Dynamic reset from the EP Transactor side provides behavior comparable to Hot-Plug as described in the steps below:

1. Apply reset as shown below:

```
rc->reqFor(hw_reset_n, true); //----------- Applying Reset---------
rc->wait_for_cycle(100);
rc->reqFor(hw_reset_n, false); //----------Removing Reset--------
```

2. Apply the initial configuration as shown below:

```
rc->wait_for((Ltssm_state_t)STATE_DETECT_QUIET);
rc->while(!Xtor::AllXtorInitDone());//--- Applying Initial
 Configuration--
rc->wait_for(linkup_done);
```

Feedback

Callbacks

PCIe transactor notifies the testbench about all events and received traffic using the event-based callback mechanism.

The various callback events provided by PCIe transactor are broadly classified as follows:

Table 26 Callbacks

Category	Callback Event	Description
Transaction Layer	tlp_received_cb	Notifies reception of TLP by the transactor.
	error_trig_cb	Notifies detection of error/ unexpected operation.
	tlp_xmit_cb	Notifies successful transmission of a TLP to PCIe core.
	INTx_cb	Notifies reception of INTx event.
Data Link Layer	DLup_change_cb	Notifies DL Up transition.
	Credits_info_cb	Notifies update of Flow Control Credit for the link.
Physical Layer	ltssm_change_cb	Notifies LTSSM state change.
	rate_change_cb	Notifies PCIe operating speed change.
	l1sub_state_change_cb	Notifies L1 sub-state change.
	core_req_rst_cb	Notifies reset assertion/ de-assertion.
	l0p_state_change_cb	Notifies link width change via L0p.
	flit_mode_change_cb	Notifies Flit mode negotiation.
Configuration Space	BAR_change_cb	Notifies BAR update.
	INT_change_cb	Notifies update to MSI Enable and MSI-X Enable bits.
	ReqSize_change_cb	Notifies change of Max_Payload_Size and Max_Read_Request_Size bits.

Feedback

Table 26 Callbacks (Continued)

Category	Callback Event	Description
	rcb_change_cb	Notifies change of Read Completion Boundary (RCB) bits.
	RegExt_cb	Notifies access to external ELBI configuration space register.
	BusDev_change_cb	Notifies Bus Number/ Device Number update.
Transactor-Specific	clock_tick_cb	Notifies configured period of fastest PCIe cycle tick ended.
	devType_change_cb	Notifies change of Device Type, that is, RC or EP.
	TxQSize_wm_cb	Notifies detection of configured limit for the TLP transmit queue.

For more information on the struct associated with each of the above events, see $ZEBU_IP_ROOT/xtor_pcie_svs/include/xtor_pcie_svs_defines.hh file.

Event-based callback mechanism can be used in the testbench as follows:

1. Implement callback function in the testbench as below:

```
void receive_callback(xtor_pcie_svs_event_t evt_type, void* pkt, void*
 ctxt) {

    xtor_pcie_svs_testbench * tb   =
 ((xtor_pcie_svs_testbench_ctxt*)(ctxt))->tb;
    xtor_pcie_svs           * xtor =
 ((xtor_pcie_svs_testbench_ctxt*)(ctxt))->xtor;

    // This event gets triggered when a TLP is received by Xtor
    if(evt_type == tlp_received_cb) {
        xtor_pcie_svs_tlp * tlp =
 ((xtor_pcie_svs_tlp_received_cb_param *)(pkt))->tlp;
       if((tlp->isMesg()) && (tlp->getMessageCode() == 0x50)){
         xtor->print("Received Set Slot Power Limit message\n"); }
}
// This event gets triggered when PCIe operating rate gets changed
else if(evt_type == rate_change_cb)
 { xtor_pcie_svs_rate_change_cb_param*evt_pkt=((xtor_pcie_svs_rate_cha
nge_cb_param*)(pkt));
 xtor->print("Speed is Gen %d\n", evt_pkt->pipe_rate+1); }
 return;
```

```
}
```

2. Register the callback using the register_callback() method before transactor initialization as below:

```
ctxt= new xtor_pcie_svs_testbench_ctxt; ctxt->tb=this; ctxt->xtor=x;
 x->register_callback(receive_callback,ctxt);
```

Running the Transactor Example

To compile the RTL, use the following command:

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

Default value for <work_directory> is zCui.work.

To run a test scenario, use the following command on the emulator machine:

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

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

where, <UCLI script> includes the zRci UCLI commands to control Transactors and debug them. It also includes ZEMI3 commands.

An example of Topology 1: RC Transactor - EP DUT, which is covered in RC Transactor and EP DUT, is available in the xtor_pcie_svs package. In this example, an x4 Lanes EP DUT imitating a device executing DMA upstream transfers is connected to pcie_wrapper_svs, which is composed of a transactor functioning as a Root Complex.

To compile the example, the *compile target 4x32b_endpoint* can be used as follows:

```
make TG_COMP=4x32b_endpoint compile
```

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

```
make TG_RUN=xtor_pcie_svs_test run
make TG_RUN=xtor_wrapper_pcie_svs_test run
```

where,

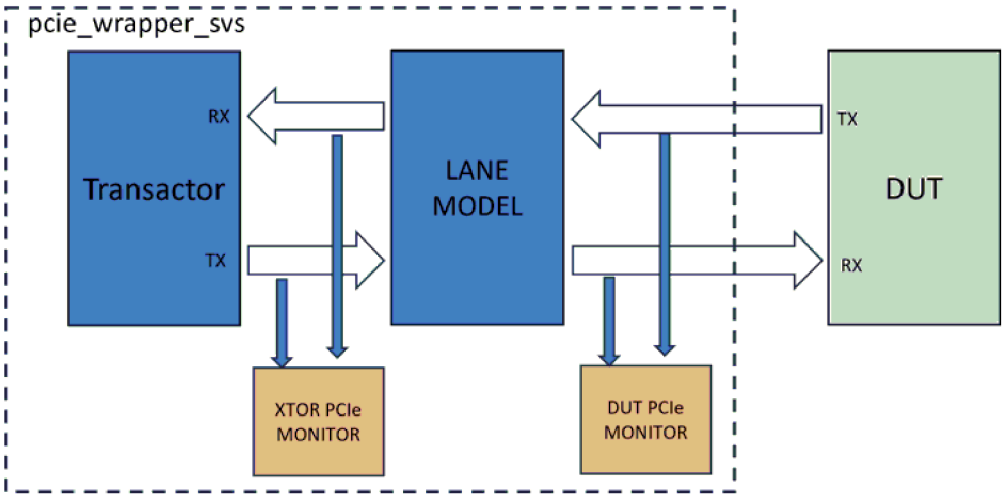
- *xtor_pcie_svs_test*: Performs link bring up and basic transactions between RC and EP.
- *xtor_wrapper_pcie_svs_test*: RC Transactor performs link bring up and DUT initialization. It also responds to the DMA accesses received from the EP.

5

PCIe Monitor

PCIe Monitor analyzes the physical channel traffic and reports it in the form of log and PA FSDB (graphical packet format) to understand the PIPE traffic. The PCIe Monitor is available in the *monitor_pcie_svs* package. It is a passive monitor instantiated (default) inside a *pcie_wrapper_svs* wrapper module. It picks the data from bus and processes it based on TLP, DLP, OS, and ltssm. The software uses the integrated Protocol Analyzer, which helps in the generation of PA FSDB.

Figure 17 PCIe Monitor



This section explains the following topics:

- Features
- Limitations
- FlexLM License Requirement
- PCIe Monitor Integration
- Using the PCIe Monitor
- Customizing Data Log

Features

The PCIe Monitor provides the following features:

- Identifies the LTSSM state and rate of link operation.
- Displays TLPs, DLLPs, and packets of ordered sets in both Tx and Rx directions with packet start and end times.
- Optionally, Analyzer also reports a data file with unscrambled PIPE data.
- Works for link width x1, x2, x4, x8 and x16 with a pipe width of 16 and 32 bits.
- Displays total count of different types of packets exchanged over the link.
- Supports Gen1, Gen2, Gen3, Gen4, Gen5, Gen6.
- Supports processing of incoming data when it is pre-coded.
- Supports multiple start stop during the emulation run.
- Supports Serdes Architecture.
- Supports multi instance monitor logging in ZDPI monitor mode.

Limitations

The following are the current limitations of the ZeBu PCIe Monitor:

- Multi-start/stop of monitor using analyzerStart() and analyzerStop() APIs is not supported for all speeds in the Flit Mode.
- ZDPI Monitor does not support online mode.

FlexLM License Requirement

You need the *hw_xtormm_monitors* FLEXnet license to use the PCIe monitor.

PCIe Monitor Integration

The PCIe Monitor is available in the *monitor_pcie_svs<num>.sh* package.

Feedback

Perform the following steps to enable the PCIe monitor:

1. Install the monitor_pcie_svs.sh package.
2. For connecting PCIe monitor, refer to the *$ZEBU_IP_ROOT/xtor_pcie_svs/misc/pcie_wrapper_svs.sv* file. In PCIe wrapper, the below two instances of monitor module, `monitor_pcie_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 pcie_wrapper_svs contains the following parameters:

Table 27 PCIe Wrapper Parameters

Parameter	Default Value	Description
IMPLEMENT_MONITOR_DUT	1	Enables DUT PCIe monitor on DUT-Lane model PIPE interface.
IMPLEMENT_MONITOR_XTOR	0	Enables XTOR PCIe monitor on XTOR-Lane model PIPE interface.

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

Note:

You can enable either the IMPLEMENT_MONITOR_XTOR parameter or the IMPLEMENT_MONITOR_DUT parameter at a time.

Using the PCIe Monitor

The PCIe Monitor operates in two modes, ZEMI3 and ZDPI.

To use the PCIe monitor in ZEMI3/ZDPI mode, the following is required:

- Configuring the Hardware
- Configuring the Software (Testbench)
- Starting and Stopping the PCIe Monitor
- Generating the Log File and the FSDB File

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:

ZEMI3 MODE

Specify the following on the VCS command line:

```
+define+ZEMI3_PCIE_ANALYZER
```

Specify the following in the UTF command list:

```
xtors -add monitor_pcie_svs_core -type ZEMI3
zemi3 -timestamp true
```

ZDPI Mode

Specify the following on the VCS command line:

```
+define +ZDPI_PCIE_ANALYZER
```

Specify the following in the UTF command list:

```
dpi_synthesis -enable ALL
zforce -rtlname
<monitor_instance_path>.xtor_pcie_svs_monitor.en_analyzer
```

The following is the example of <monitor_instance_path>:

```
wrapper.pcie_wrapper_svs.genblk1.monitor_top_inst_xtor
```

To increase frequency limit add the following in the UTF file:

```
ztopbuild -advanced_command {set_fast_waveform_capture -
fwc_frequency 8000 -dpi_frequency 8000}
ztopbuild -advanced_command {set_fast_waveform_capture -
sampling_frequency {100MHz}}
```

Configuring the Software (Testbench)

To use the PCIe Monitor, perform the following steps:

1. Perform the following updates to the software (testbench):
   a. Include necessary PCIe monitor header files in the testbench:

```
#include "monitor_pcie_svs.hh"
using namespace MONITOR_PCIE_SVS;
```

b. Declare and initialize PCIe monitor handles (monitor_pcie_svs) in the main method as shown below:

```
int main(int argc, char * argv[]) {
monitor_pcie_svs* tmp_mon_pcie_svs
string monitor_type = "<monitor_type>" ; // (ZEMI3/ZDPI)
monitor_pcie_svs::set_mon_mode(monitor_type);
monitor_pcie_svs::Register();
string mondriverList[1] =
 {"<monitor_instance_path>.xtor_pcie_svs_monitor"};
// E.g., <monitor_instance_path>=
 wrapper.pcie_wrapper_svs.genblk1.monitor_top_inst_xtor
int itMondriver =0; …
if (monitor_type == "ZEMI3")
{ mondriverList[0]=""; } // will retreive driver from dpi list
while((itMondriver<1) &&
 ((tmp_mon_pcie_svs=static_cast<monitor_pcie_svs*>(Xtor::getNewXtor(
monitor_pcie_svs::(mondriverList[itMondriver]).c_str()))!=NULL)){ i
tMondriver ++; }
… }
```

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

c. Invoke the monitor APIs analyzerStart() and analyzerStop() to start and stop the Analyzer, respectively. The usage of these APIs is described in Starting and Stopping the PCIe Monitor.

2. Use the Linker flag -lmonitor_pcie_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
```

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

```
xtor_pcie_svs_testbench.cc
xtor_pcie_svs_testbench.hh
```

Starting and Stopping the PCIe Monitor

The start/stop feature of the PCIe 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 anywhere from the testcase after PCIe monitor handles 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 xtor_pcie_svs_testbench.cc, located in the $ZEBU_IP_ROOT/xtor_pcie_svs/example/src/bench/ directory.

   Multi-start/stop call of APIs can be invoked from the testcase, 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 report file, which is identifiable by the number listed at the end of the file name, as shown in the following example:

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

   Here, 0 or 1 corresponds to the 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 corresponding symbol log and PA FSDB for corresponding dump files.

Feedback

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

Note:
In ZEMI3 mode, the binary report file size is restricted to 5GB by default. You can change the setting by using pre_analyzerStart() API in the testbench as shown below:

```
mon_cfg_struct_t cfg_t;
cfg_t.maxbindmp_size = <value>;
<monitor_handle>->pre_analyzerStart(cfg_t);
```

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

ZDPI monitor works only in emulation mode and offline mode.

Perform the following steps to generate the log and FSDB files for the PCIe ZDPI Monitor:

Method 1

1. Run the testcase.
2. Convert Ztdb to Log and PA FSDB (Post-processing)

Method 2

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

ZEMI3 Monitor

ZEMI3 monitor works in both simulation and emulation modes.

Perform the following steps to generate the log and FSDB files for the PCIe ZEMI3 Monitor:

1. Run the test case
2. Convert Binary File to Log and PA FSDB (Post-processing).

Convert Ztdb to Log and PA FSDB (Post-processing)

This is the default method of generating Log and PA FSDB in the ZDPI monitor mode.

To generate the report, specify the following command:

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

Consider the following example:

```
zdpiReport -z 16x32b.zebu.work/zebu.work/ -f import_list -i
 rundir_pcie_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_zdpi` and specify it with the -f switch in above command.

The data log is generated with the monitor instance hierarchical path added as prefix in its name.

For example:

```
wrapper.pcie_wrapper_svs.genblk1.monitor_top_inst_xtor.xtor_pcie_svs_moni
tor.RC_analyzer_dump.dat
```

Convert Ztdb to Binary Dump

1. Set the PCIE_VS_MONITOR_ZDPI_PA_BINDMP_TX_RX_EN environment variable to 1.
2. Run the testcase.
3. To generate the report, specify the following command:

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

   See Convert Ztdb to Log and PA FSDB (Post-processing) for zdpiReport command example.

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

To convert binary file to log and PA FSDB, specify the following command:

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

Feedback

To convert custom binary file to log and PA FSDB, specify the following command:

```
make generate_monitor_PA TG_COMP=<TG_COMP value> TG_RUN=<TG_RUN value>
 USR_BINDMP=1 PA_DUMPFILE_PATH=< Absolute path to the binary dump file
 "postproc_dumpfile*"> TG_COMP=<TG_COMP value> IMPLEMENT_MONITOR=1
```

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

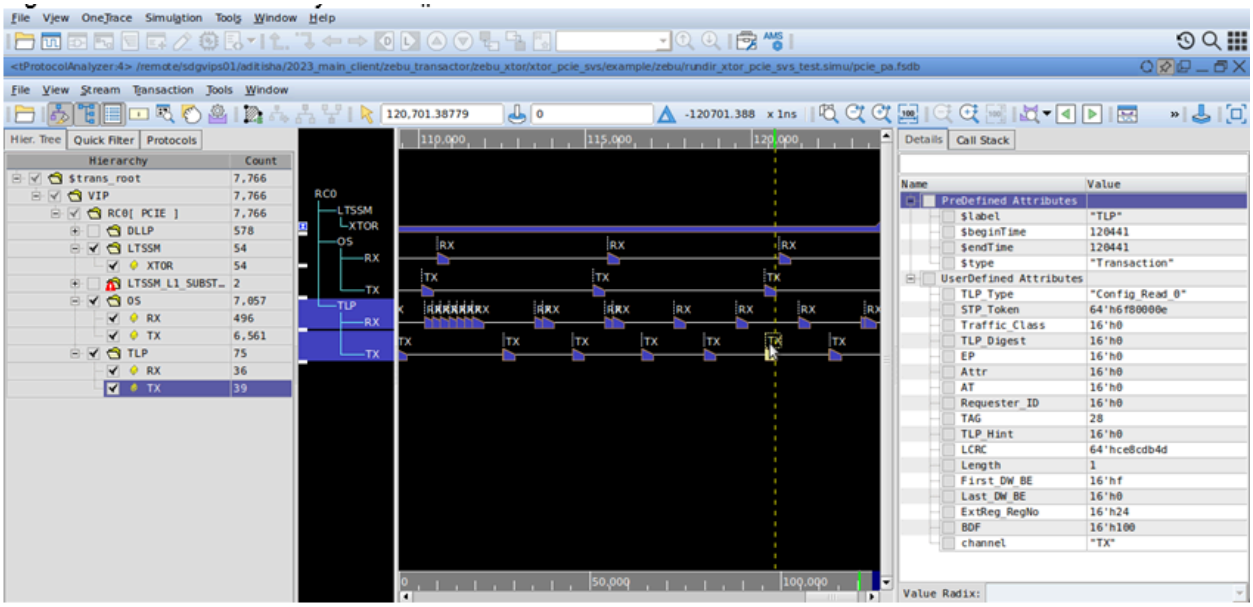
Open the FSDB file created by post-processing in Verdi, using the following command:

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

where, pcie_pa.fsdb is the FSDB file created.

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

Figure 18 Protocol Analyzer GUI



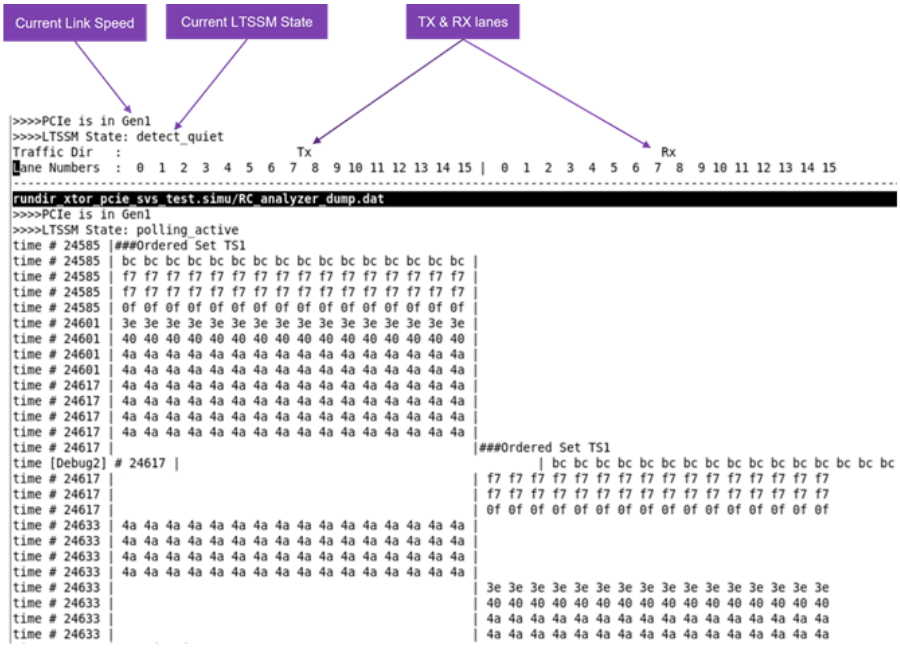
Log file

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

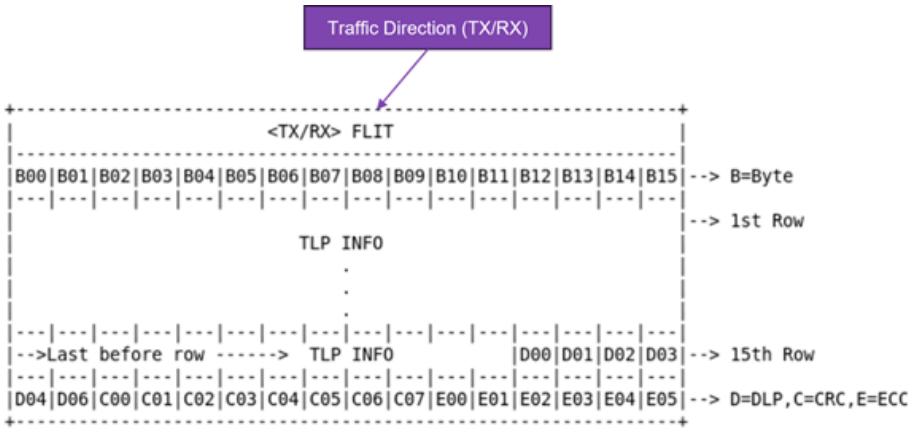
Figure 19 Traffic Based Log File

Figure 20 Lane Based Log File



Feedback

Figure 21 256B Flit Display



Customizing Data Log

Customize the data log for the following:

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

When binary output is converted to Data Log in ZEMI3 or ZDPI monitor modes, you can customize the data logs using the following parameters:

Table 28 Customizing Data Log

Parameter	Default Value	Description
PCIE_VS_MONITOR_PA_OS_FSDB_EN	true	Prints ordered sets in traffic log
PCIE_VS_MONITOR_PA_TLP_FSDB_EN	true	Prints TLP in traffic log
PCIE_VS_MONITOR_PA_DLLP_FSDB_EN	true	Prints DLLP in traffic log
PCIE_VS_MONITOR_PA_FSDB_EN	true	Enables FSDB for PA
PCIE_VS_MONITOR_PA_PKT_LOG_EN	true	Prints traffic-based log. However, set the value of the parameter to false if you want to print lane-based log.
PCIE_VS_MONITOR_PA_FLIT_DISPLAY	false	Prints flit information in tabular format.

Feedback

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

```
xtor_pcie_mon_pktprntCfg_flag_t pkt_flag;
pkt_flag.PCIE_VS_MONITOR_PA_PKT_LOG_EN =false;
monitor_pcie_svs::doPciePostProcessing
 (PA_DUMPFILE.c_str(),mon_file,LaneWidth,PipeWidth,true,NULL,pkt_flag);
```

When Ztdb is converted to Data Log in the ZDPI monitor mode, you can customize the data logs using the following environment variable before running the testcase:

Table 29 Environment Variables

Environment Variable	Default Value	Description
PCIE_VS_MONITOR_ZDPI_PA_PKT_EN	1	Prints traffic-based log. However, set the value of this environment variable to false if you want to print lane-based log.
PCIE_VS_MONITOR_ZDPI_PA_FSDB_DIS	0	Disables FSDB for PA
PCIE_VS_MONITOR_ZDPI_PA_OS_FSDB_DIS	0	Disables printing of ordered sets in traffic log
PCIE_VS_MONITOR_ZDPI_PA_TLP_FSDB_DIS	0	Disables printing of TLP in traffic log
PCIE_VS_MONITOR_ZDPI_PA_DLLP_FSDB_DIS	0	Disables printing of DLLP logging in traffic log
PCIE_VS_MONITOR_ZDPI_PA_DEBUG_PRINT_EN	0	Enables printing of high verbose data for debuggin
PCIE_VS_MONITOR_ZDPI_PA_BYPASS_IMP_EN	0	Bypasses import definition
PCIE_VS_MONITOR_ZDPI_PA_FLIT_DISPLAY	0	Prints flit information in tabular format
PCIE_VS_MONITOR_ZDPI_PA_BINDMP_TX_RX_EN	0	Enables binary dump generation

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 transactions, as described in the following sections:

- Integration or Transactor Bring-Up
- Linkup
- Runtime
- Infrastructure Errors
- Monitor Related Errors
- Understanding TS1 and TS2 Ordered Sets
- Frequently Asked Questions

Integration or Transactor Bring-Up

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

- Setting PIPE Width and Frequency
- Variable Link Width
- Link Bifurcation
- PIPE Signal Width Mismatch
- Maintaining Clock Ratios

Setting PIPE Width and Frequency

One of the common mistakes during the bring-up of PCIe transactor is incorrectly setting PIPE WIDTH and FREQUENCY with respect to the DUT.

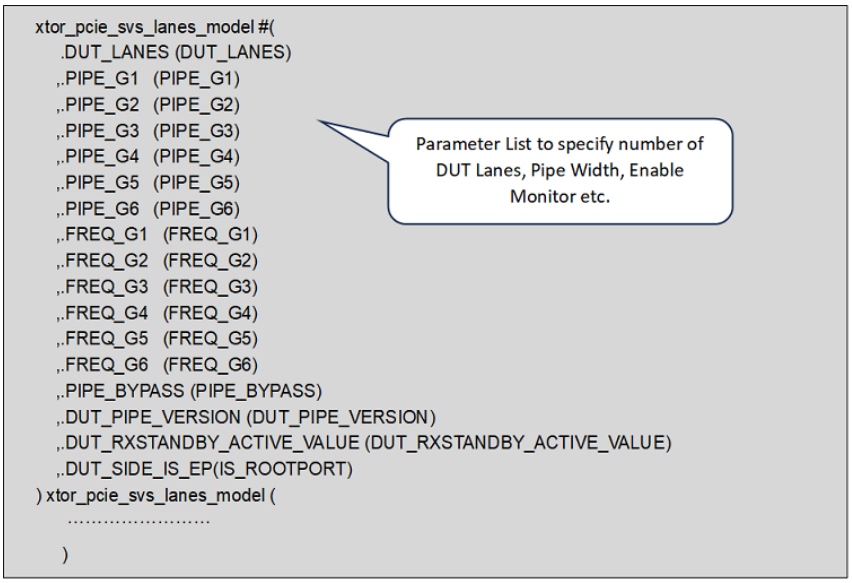
Feedback

While integrating the PCIE transactor, along with the lane model, set the following parameters correctly:

- DUT_LANES: Set the number of lanes for the DUT.
- PIPE_G1 to PIPE_G5: Set the DUT PIPE width for G1 to G5.
- FREQ_G1 to FREQ_G5: Set the DUT frequency for G1 to G5.

To set the PIPE width and frequency with respect to the DUT, see Table 14.

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:

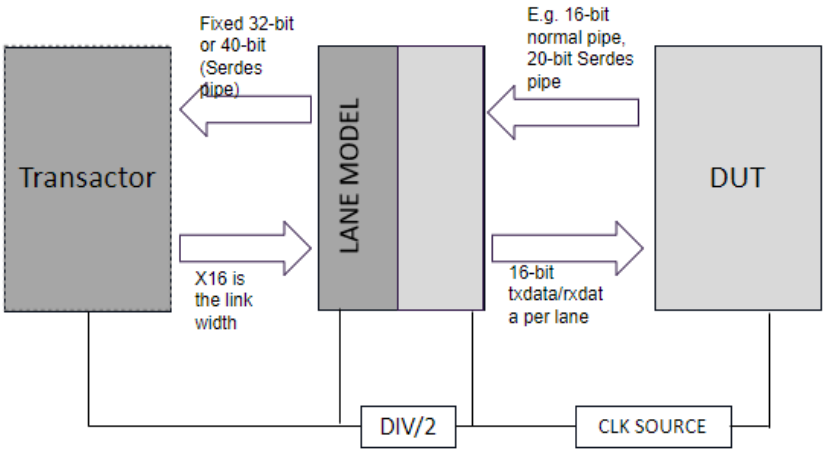
- The connections from the DUT to Lane model should be connected as per the maximum link width supported.
- In the C / zRCI function, specify the PCIE_MAXLINKWIDTH as per the required link width.

Feedback

Link Bifurcation

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

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 and 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.

Feedback

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:

- DUT LTSSM State Stuck at Polling Compliance
- Link is Stuck in Recovery.RcvrLock and Recovery.Speed
- PCIe Transactor Link Not Up When DUT Max Rate Is Gen4

Feedback

- Unable to Retrain PCIe Link
- Fixing the Halt Observed in the Equalization State
- LTSSM in Detect Substates
- LTSSM in Polling Substates
- LTSSM in Configuration Sub-states
- Using the alias Functionality

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 the DUT configuration as explained inIntegration or Transactor Bring-Up .
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_pcie_svs module.
3. Check if lane model clocks (phy_clk_dut and phy_clk_xactor) are generated as per the operating frequency.

Table 30 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

Feedback

For example, in case of max_phy_rate is Gen5, clock frequencies are expected as below:

4. 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);
   ```

   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.

5. Check for any mismatch in the width of txelecidle_from_dut signal of DUT and PCIe wrapper. For example, if the width of txelecidle signal is 1-bit per lane for DUT and 4-bit per lane for PCIe Wrapper, the remaining bits need to be concatenated, as shown below:

   ```
   .txelecidle_from_dut({3'b000,txelecidle_from_ep_dut[3],3'b000,txelecidle_from_ep_dut[2],3'b000,txelecidle_from_ep_dut[1],3'b000,txelecidle_from_ep_dut[0]}),
   ```

6. Check if the link width is configured correctly according to DUT as explained in Variable Link Width.

Link is Stuck in Recovery.RcvrLock and Recovery.Speed

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

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: If the DUT does not support RxStandBy/RxStandByStatus handshake, set the DUT_RXSTANDBY_ACTIVE_VALUE parameter to 0.

Feedback

PCIe 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_pcie_svs pcie_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, perform the following steps to 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 re-train phase. The ltssm_info signal is used to check the current ltssm state and its encoding is provided in ltssm_state Encoding.

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}};
```

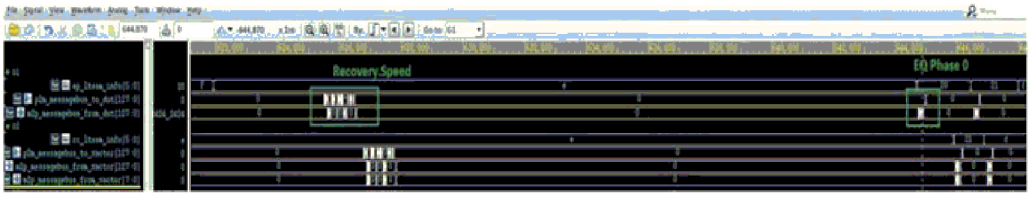
Feedback

```
     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
=================================================
```

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



Feedback

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.

The PCIe transactor is always in the active state, that is, transactor does not wait for 12ms timeout before transitioning from Detect.Quiet to Detect.Active state. After the transactor is in the Detect.Quiet state, it waits for 1500 cycles of clock in P1 powerdown state. 1500 is a comfortable time period to ensure the transactor is out of reset phase.

- If you want the DUT to exit Detect.Quiet state before 12ms timeout, it is possible as the transactor is already in the active state.
- However, if you want to wait for 12ms timeout to exit Detect.Quiet state, do not send the valid traffic on the lanes, that is, do not break the electrical idle conditions of the lane.

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.

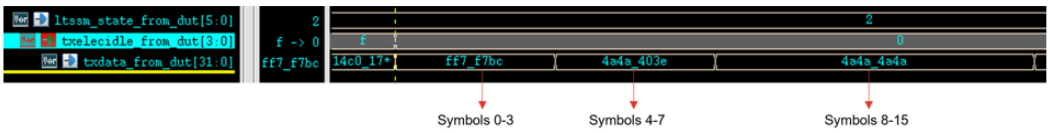
The following explains various substates for the Polling substate:

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

Polling.Active

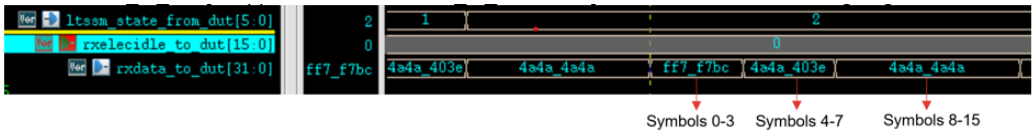
During the LTSSM Polling.Active state, the DUT MAC:

- 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:



Feedback

- 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.

  xtor()->setConfigParam("PCIE_TIMING",(uint32_t) XTOR_PCIE_SVS_SHORT);

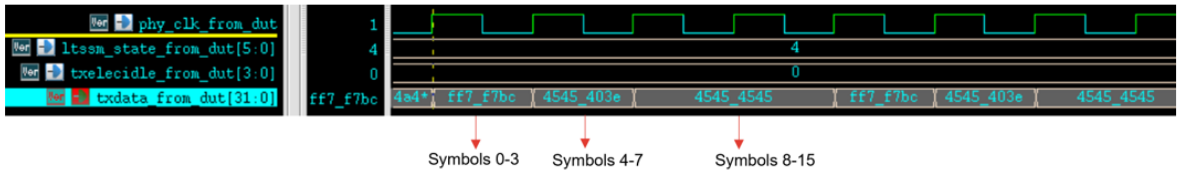
  `where XTOR_PCIE_SVS_SHORT=0.`

- 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.

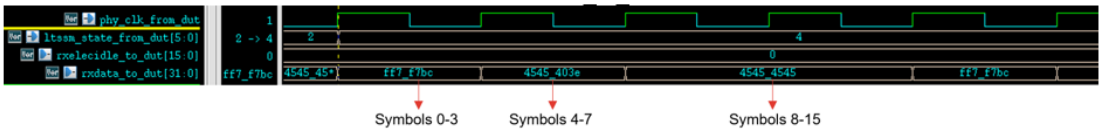
LTSSM loops between Polling.Active <-> Polling.Compliance or Polling.Active <-> Detect until the conditions to move to Polling.Configuration are met.

Feedback

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 Sub-states

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

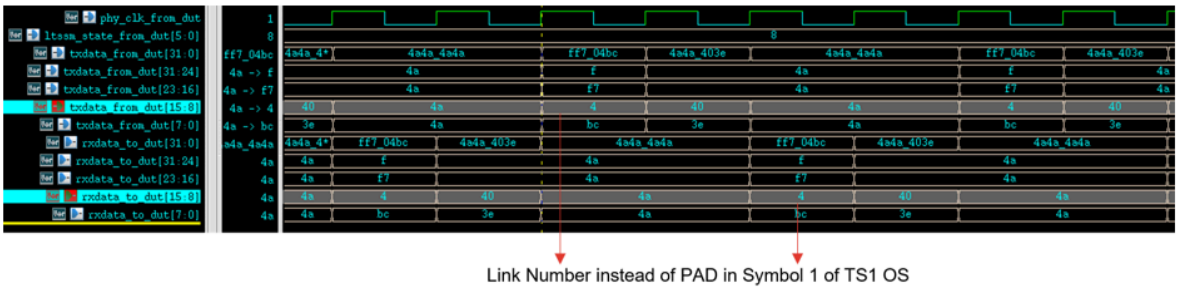
The following are the Configuration sub-states:

- 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.

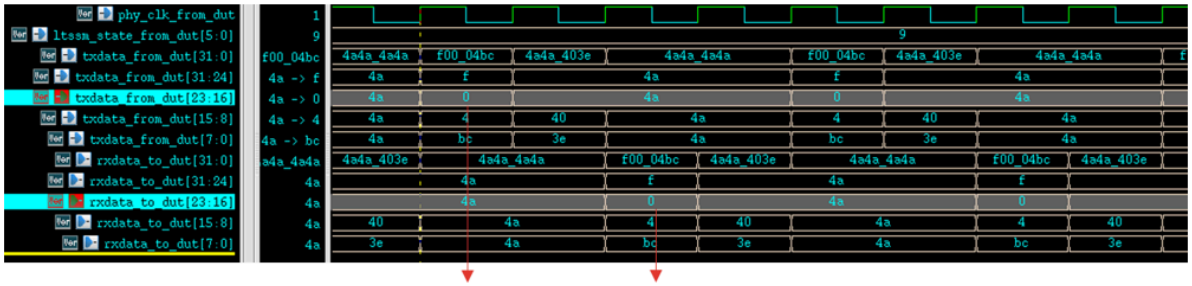


- Configuration.LaneNum.wait (LTSSM State = 0x9)

Feedback

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

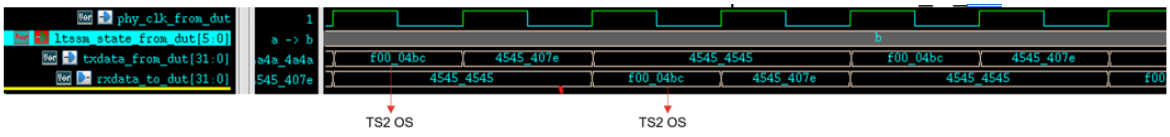
DUT MAC waits until it receives the lane number assignments from its link partner on Symbol2 in TS1 ordered sets for each lane rxdata_to_dut.



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



- Configuration.Idle (LTSSM State = 0xC)

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.

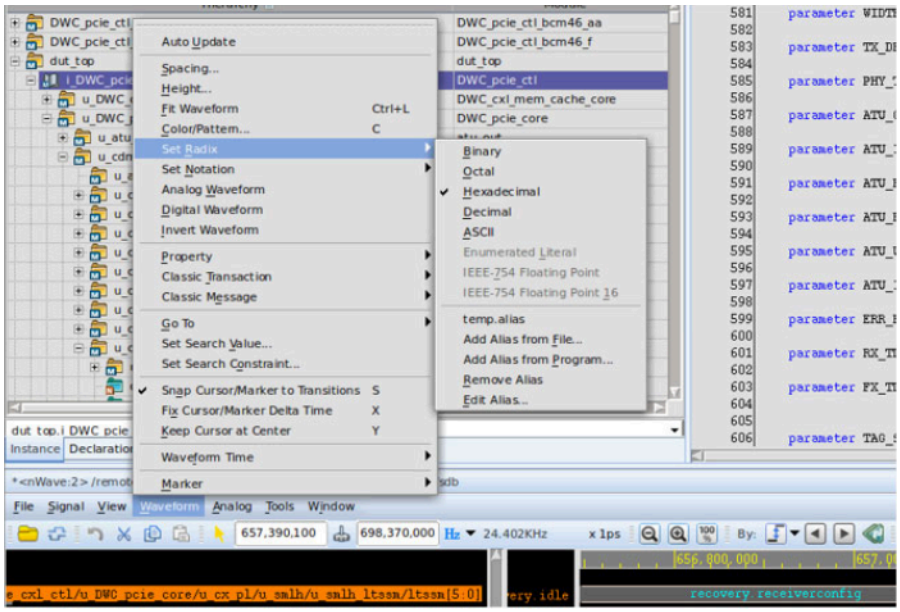
Feedback

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:

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:



Feedback

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 following are the common checks during the runtime:

- EP Transactor not Sending the Completions Correctly
- Runtime Halt Observed After Reset
- Error During Stress Testing of MSI
- Transactor Not Working at Default Frequency
- Completions from DUT getting dropped due to Completion Timeout

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. That is, ensure that the completions are sent by the testbench correctly.

So, the first thing is to

- If Completion TLP is sent directly in callbacks (that is, ep_receive_callback), use the non-blocking method pushTxtlp(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.

Runtime Halt Observed After Reset

The following run time halt is observed after reset.

```
INFO  : wrapper.pcie_driver_rc [xtor_pcie_svs] -wrapper.pcie_driver_rc-
 Reset de-asserted  ...
DEBUG1: wrapper.pcie_driver_rc [xtor_pcie_svs] isResetActive() status for
 [wrapper.pcie_driver_rc]
```

```
INFO  : wrapper.pcie_driver_rc [xtor_pcie_svs] -wrapper.pcie_driver_rc-
 Reset 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_pcie_svs*>(Xtor::getNewXtor(xtor_pcie_svs::getXtorT
ypeName(), 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. This is needed because internally IP responds to MSI requests in 3 cycles with grant signal. It would be good to send MSI after receiving the grant for previous MSI. The delay can be added using the pcie_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}
}
ztopbuild -advanced_command {clock_localization -stop_at_async_set_reset
 no}
```

Feedback

```
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.

Completions from DUT getting dropped due to Completion Timeout

For non-posted requests initiated by the transactor, it is possible that DUT sends corresponding completions after Completion Timeout due to longer TLP datapath.

In this case, check if the completions are received from DUT at the interface.

If the completions are received from DUT, configure the Completion Timeout Value field of Device Control 2 Register in PCI Express Capability Structure of transactor through backdoor as per the DUT requirement.

Below is the code snippet for changing the Completion Timeout Value field for the transactor:

```
  unsigned ep_cap_base_addr= 0x0;
    uint32_t rdata=0;
    ep_cap_base_addr = (ep->getCapPtr(pcie_cap_id));
    rdata=ep->PCIeXtorCfgRead(ep_cap_base_addr+0x28);
    rdata |= 0x00000006; // Increasing from default 50us timeout to 65ms
 to 210ms range.
    ep->PCIeXtorCfgWrite(ep_cap_base_addr+0x28,rdata);
```

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.pcie_driver_ep.SNPS_PCIE_RST_CLK

- pushTlp: ERROR : Tx tlp Fifo is Full (128/128)
- No <svt_hw_platform> defined for import DPI call
- Segmentation Fault in getDriverInstanceName() API

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
xtor_pcie_svs::Register("xtor_pcie_svs");

// Construct the transactor
test =
 static_cast<xtor_pcie_svs*>(Xtor::getNewXtor(xtor_pcie_svs::getXtorTy
peName(), XtorRuntimeZebuMode)) ;

// Call XTOR init method if exists for the transactor
test->init(board,"pcie_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.

Similarly, you might face an error with . Follow the above steps for these modules as well.

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

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_pcie_svs*>(Xtor::getNewXtor(xtor_pcie_svs::getXtorT
ypeName(), XtorRuntimeZebuMode, <inst_name>));
```

Fatal error: DPI scope not found for path wrapper.pcie_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.pcie_driver_ep [SNPS/XTOR/HDLDRVR/FOUND ] RTL driver
 instance DpiOnly "xtor_pcie_svs" detected.
DEBUG : wrapper.pcie_driver_ep [xtor_pcie_svs] value of path is
 wrapper.pcie_driver_ep.SNPS_PCIE_RST_CLK
```

Feedback

```
svGetScopeFromName : Error:  scope
 'wrapper.pcie_driver_ep.SNPS_PCIE_RST_CLK' is unknown.
ERROR : wrapper.pcie_driver_ep [xtor_pcie_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_pcie_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_pcie_svs] pushTlp: ERROR : Tx tlp Fifo is Full (128/128).
INFO : [xtor_pcie_svs] pushTlp: ERROR : Increase Fifo size with
 PCIE_TXTLPFIFO_SIZE ConfigParam (current is 128).
```

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

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.

Feedback

```
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 : [pcie_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::pcie_init() call with a runtime object pointer, which is used to initialize the hw_platform.

Segmentation Fault in getDriverInstanceName() API

The following error is reported when the getDriverInstanceName() API is compared with the wrong hierarchical path of the transactor instance in the testbench:

```
Thread 66 "zRci" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffeb4d35700 (LWP 40624)]
0x00007fffcc271690 in ZEBU_IP::Xtor::getDriverInstanceName() ()

 from /
zebu_project/share/SNPS/ZEBU_IP_ROOT/S-2023.03-SP1-T-20240229/lib/libZebu
Xtor.so
Missing separate debuginfos, use: debuginfo-install
 gmp-6.0.0-15.el7.x86_64 nss-pam-ldapd-0.8.13-22.el7.x86_64
 sssd-client-1.16.4-37.el7.x86_64 zlib-1.2.7-18.el7.x86_64
(gdb) thread apply all bt

Thread 66 (Thread 0x7ffeb4d35700 (LWP 40624)):
#0  0x00007fffcc271690 in ZEBU_IP::Xtor::getDriverInstanceName() ()

 from /
zebu_project/share/SNPS/ZEBU_IP_ROOT/S-2023.03-SP1-T-20240229/lib/libZebu
Xtor.so
```

To correct the error, ensure that the correct hierarchical path of the transactor instance is used in the testbench.

Monitor Related Errors

This section describes the errors reported for the PCIe monitor:

- Undefined symbol: _ZN7ZEBU_IP15MONITOR_PCIE_SVS15monitor_pcie_svs8RegisterEPKc
- The DPI export function/task 'monitor_en_PCIE_DPI' not found
- Cannot read Mudb version

Undefined symbol: _ZN7ZEBU_IP15MONITOR_PCIE_SVS15monitor_pcie_svs8RegisterEPKc

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

The DPI export function/task 'monitor_en_PCIE_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_PCIE_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.
  Please 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:

```
pcie_monitor = new monitor_pcie_svs("monitor_pcie_core",
 "wrapper.monitor_top_inst_rc.xtor_pcie_svs_monitor", xsched, runtime);
```

Feedback

Cannot read Mudb version

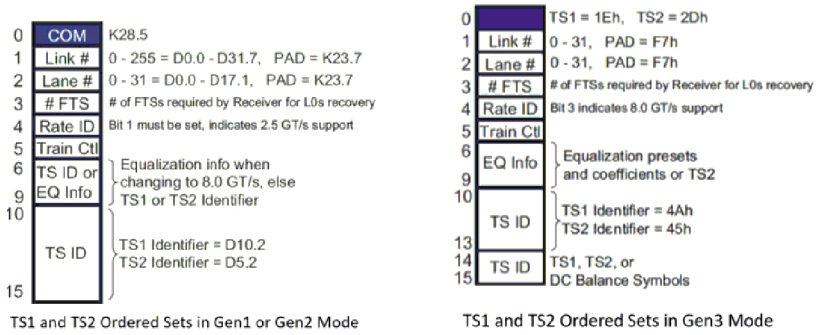
The following FATAL error is reported when the monitor cannot read the Mudb version:

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

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.



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 PCIe Transactor:

- How to dynamically change the speed and width?
- How to set the max TLP payload size?

- How to verify Hot Reset?
- How to verify Link Disable?
- How to verify Dynamic Reset?
- How to verify ASPM L1?
- How to verify ASPM L1.1?
- How to verify ASPM L1.2?
- How to verify L2?
- How to verify Dynamic Linkwidth Change in Non-Flit mode?
- How to verify Linkwidth Change via L0p in Flit mode?

How to dynamically change the speed and width?

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

Note that the dynamic speed change upto Gen6 is supported.

You can dynamically change the width of the PCIe 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 follows:

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

2. You can also set the value of Max_Payload_Size to a value lesser than Max_Payload_Size supported. To do so, configure Max_Payload_Size of function in Device Control Register, either through frontdoor CFG TLP from RC or through backdoor from the transactor.

   If you are unable to configure the Max_Payload_Size of EP using RC 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
   uint16_t cfg_reg;
   device->pcie_BfmCfgRd(ep->pcie_xtor()->getCapPtr(pcie_cap_id)+0x8,&cfg
   _re g);
   ```

Feedback

```
device->pcie_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->pcie_xtor()->print("Value of Device Control Status Reg :
%x",cfg_reg);
```

After completing these steps, FIFO accommodates 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 RC side.

Transactor is RC

For achieving hot reset scenario if transactor is RC, perform the following steps:

Step1: Entering the Hot Reset state

```
rdata = 0;
rdata = rc->PCIeXtorCfgRead(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE);
rdata |= 0x00400000;
rc->PCIeXtorCfgWrite(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE,rdata);

//Wait for XTOR to reach HOT Reset LTSSM state
rc->wait_for(STATE_HOT_RESET);
```

Step 2: Exiting from the Hot Reset state

```
// Disabling the secondary bus reset bit in bridge control register.
rdata = rc->PCIeXtorCfgRead(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE);
rdata &= 0xffbfffff;
rc-> PCIeXtorCfgWrite(XTOR_PCIE_SVS_TYPE1_BRIDGECTRLINTPININTLINE,rdata);

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

Step 3: Apply Reset

```
//Assert HW Reset
rc->reqFor(hw_reset_n, true);

//Wait for few clock cycles
rc->wait_for_cycle(100);

//De-assert HW reset
rc->reqFor(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 EP

For achieving the hot reset scenario if the transactor is EP, performing the following steps:

Step 1: Entering the Hot Reset state

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

Step 2: Exiting the Hot Reset state

```
ep->wait_for(STATE_DETECT_QUIET);
```

Step 3: Apply Reset

```
//Assert HW Reset
ep->reqFor(hw_reset_n, true);

//Wait for few clock cycles in sync with RC's wait cycles
ep->wait_for_cycle(100);

//De-assert HW reset
ep->reqFor(hw_reset_n, false);
```

Step 4: Initial configuration

```
//Initialize XTOR again
ep->initBFM();

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

How to verify Link Disable?

Link Disable can only be initiated from the RC side.

For testing the Link Disable scenario, if the transactor is RC, perform the following steps:

Step 1: Entering the Link Disable state

```
//Set Link Disable bit in Link Control register
uint32_t link_sts_ctrl;
uint32_t pcie_cap_ptr;
pcie_cap_ptr = rc->getCapPtr(pcie_cap_id);
link_sts_ctrl = rc->PCIeXtorCfgRead(pcie_cap_ptr+0x10);
```

Feedback

```
link_sts_ctrl |= 0x10;
rc->PCIeXtorCfgWrite(pcie_cap_ptr+0x10, link_sts_ctrl);

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

Step2: Exiting the Link Disable state

```
//Resetting Link Disable bit in Link Control register
link_sts_ctrl &= ~(1 << 4) ;
rc->PCIeXtorCfgWrite(pcie_cap_ptr+0x10, link_sts_ctrl);

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

Step 3: Applying Reset

```
//Assert HW Reset
rc->reqFor(hw_reset_n, true);

//Wait for few clock cycles
rc->wait_for_cycle(100);

//De-assert HW reset
rc->reqFor(hw_reset_n, false);
```

Step 4: Initial Configuration

```
//Initialize XTOR again
rc->initBFM();

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

For testing the Link Disable scenario, if the transactor is EP, perform the following steps:

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(STATE_DETECT_QUIET);
```

Step 3: Applying Reset

```
//Assert HW Reset
ep->reqFor(hw_reset_n, true);

//Wait for few clock cycles in sync with RC’s wait cycles
ep->wait_for_cycle(100);
```

Feedback

```
//De-assert HW reset
ep->reqFor(hw_reset_n, false);
```

Step 4: Initial Configuration

```
//Initialize XTOR again
ep->initBFM();

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

How to verify Dynamic Reset?

For testing the Dynamic Reset scenario, if the transactor is RC or EP, perform the following steps:

1. Applying Reset:

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

2. Initial configuration:

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

How to verify ASPM L1?

The below Configuration parameter setting is mandatory for enabling ASPM L1 feature in transactor:

```
xtor->setConfigParam("PCIE_ASPML1", "true");
```

When Transactor is RC

For testing the ASPM L1 scenario, if the transactor is RC, perform the following steps:

Step1: Checking if the ASPM L1 is supported

```
// Check ASPM Support bits in Link Capabilities Register of Xtor
uint32_t pcie_cap_base;
uint32_t rdata;
pcie_cap_base = rc->getCapPtr(pcie_cap_id);
rdata = rc->PCIeXtorCfgRead((pcie_cap_base + 0xC));
```

```
if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  rc->print("---- RC XTOR supports ASPM L1\n");
}
else
{
  rc->error("---- RC XTOR does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}

// Check ASPM Support bits in Link Capabilities Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base + 0xC)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  rc->print("---- EP DUT support ASPM L1 \n");
}
else
{
  rc->error("---- EP DUT does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}
```

Step 2: Entering the ASPM L1 state

```
// Set ASPM Control bits to 2'b10 in Link Control Register of Xtor
rdata=rc->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata |=0x02;
rc->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM Control bits to 2'b10 in Link Control Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata |=0x02;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Wait for Xtor to reach L1.Idle LTSSM state
rc->wait_for(STATE_L1_IDLE);
```

Feedback

Step 3: Exiting the ASPM L1 state

```
// Initiate TLP to exit from ASPM L1 state
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();

// Set ASPM Control bits to 2'b00 in Link Control Register of DUT
rdata &=0xFFFFFFFC;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Set ASPM Control bits to 2'b00 in Link Control Register of Xtor
rdata=rc->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata &=0xFFFFFFFC;
rc->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);
```

When Transactor is EP

For testing the ASPM L1 scenario, if the transactor is EP, perform the following steps:

Step1: Checking if ASPM L1 is supported

```
// Check ASPM Support bits in Link Capabilities Register of Xtor
uint32_t pcie_cap_base;
uint32_t rdata;
pcie_cap_base = ep->getCapPtr(pcie_cap_id);
rdata = ep->PCIeXtorCfgRead((pcie_cap_base + 0xC));

if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  ep->print("---- EP XTOR supports ASPM L1\n");
}
else
{
  ep->error("---- EP XTOR does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}
```

Step2: Entering the ASPM L1 state

```
// Set ASPM Control bits to 2'b10 in Link Control Register of Xtor
rdata=ep->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata |=0x02;
ep->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);
```

Feedback

```
//Wait for XTOR to reach L1.Idle LTSSM state
ep->wait_for(STATE_L1_IDLE);
Step 3: Exiting the ASPM L1 state
//Wait for XTOR to reach Recovery.RcvrLock LTSSM state
ep->wait_for(STATE_RCVRY_LOCK);

//Wait for XTOR to reach L0 LTSSM state
ep->wait_for(STATE_L0);

// Set ASPM Control bits to 2'b00 in Link Control Register of Xtor
rdata=ep->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata &=0xFFFFFFFC;
ep->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);
```

How to verify ASPM L1.1?

The below Configuration parameter settings are mandatory for enabling ASPM L1.1 feature in the transactor:

```
xtor->setConfigParam("PCIE_ASPML1", "true");
xtor->setConfigParam("PCIE_ASPML1_1", "true");
```

When Transactor is RC

For testing the ASPM L1.1 scenario, if the transactor is RC, perform the following steps:

Step1: Checking if ASPM L1 and ASPM L1.1 are supported

```
// Check ASPM Support bits in Link Capabilities Register of Xtor
uint32_t pcie_cap_base;
uint32_t l1_pm_sub_ext_cap_base;
uint32_t rdata;
pcie_cap_base = rc->getCapPtr(pcie_cap_id);
rdata = rc->PCIeXtorCfgRead((pcie_cap_base + 0xC));

if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  rc->print("---- RC XTOR supports ASPM L1\n");
}
else
{
  rc->error("---- RC XTOR does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}

// Check ASPM Support bits in Link Capabilities Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base + 0xC)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
```

```
xtor_tb->wait_all_cpl();
if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  rc->print("---- EP DUT support ASPM L1 \n");
}
else
{
  rc->error("---- EP DUT does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}

// Check ASPM L1.1 Supported bit in L1 PM Substates Capabilities Register
 of Xtor
l1_pm_sub_ext_cap_base   = (ep->getExtCapPtr(l1sub_extcap_id));
rdata = rc->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base + 0x4));
if((rdata & 0x18)==0x18)
rc->print("---- RC XTOR supports ASPM L1_1\n");
else
{
  rc->error("---- RC XTOR does not support ASPM L1_1. Hence, test is not
 applicable\n");
  exit(911);
}

// Check ASPM L1.1 Supported bit in L1 PM Substates Capabilities Register
 of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((l1_pm_sub_ext_cap_base + 0x4)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
if((rdata & 0x18)==0x18)
{
   rc->print("---- EP DUT support ASPM L1.1 \n");
}
else
{
  rc->error("---- EP DUT does not support ASPM L1.1. Hence, test is not
 applicable \n");
  exit(911);
}
```

Step 2: Entering the ASPM L1 state:

```
 // Set ASPM Control bits to 2'b10 in Link Control Register of Xtor
rdata=rc->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata |=0x02;
rc->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM Control bits to 2'b10 in Link Control Register of DUT
xtor_tb->record_cpl(&rdata);
```

Feedback

```
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata |=0x02;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Set ASPM L1.1 Enable to 1 in L1 PM Substates Control 1 Register of
 Xtor
rdata=rc->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata |=0x08;
rc->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);

// Set ASPM L1.1 Enable to 1 in L1 PM Substates Control 1 Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata |=0x08;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Wait for Xtor to reach L1.Idle LTSSM state
rc->wait_for(STATE_L1_IDLE);
```

Step3: Entering the ASPM L1.1 Substate

```
// De-assert CLKREQ# signal to enter L1.1 Substate
rc->reqFor(ctrl_clk_reset_n, true);

// Wait for Xtor to reach L1.1 Substate
rc->wait_for(STATE_L1_1);
```

Step4: Exiting the ASPM L1.1 Substate

```
// Wait for few clock cycles
rc->wait_for_cycle(500);

// Assert CLKREQ# signal to exit L1.1 Substate
rc->reqFor(ctrl_clk_reset_n, false);
```

Feedback

Step 5: Exiting the ASPM L1 state:

```
//Wait for XTOR to reach Recovery.RcvrLock LTSSM state
rc->wait_for(STATE_RCVRY_LOCK);

//Wait for XTOR to reach L0 LTSSM state
rc->wait_for(STATE_L0);

// Set ASPM Control bits to 2'b00 in Link Control Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata &=0xFFFFFFFC;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Set ASPM Control bits to 2'b00 in Link Control Register of Xtor
rdata=rc->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata &=0xFFFFFFFC;
rc->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM L1.1 Enable to 0 in L1 PM Substates Control 1 Register of
 Xtor
rdata=rc->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata &=0xFFFFFFF3;
rc->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);

// Set ASPM L1.1 Enable to 0 in L1 PM Substates Control 1 Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata &=0xFFFFFFF3;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();
```

When Transactor is EP

For testing the ASPM L1.1 scenario, if the transactor is EP, perform the following steps:

Step1: Checking if ASPM L1 and ASPM L1.1 are supported

Feedback

```
// Check ASPM Support bits in Link Capabilities Register of Xtor
uint32_t pcie_cap_base;
uint32_t l1_pm_sub_ext_cap_base;
uint32_t rdata;
pcie_cap_base = ep->getCapPtr(pcie_cap_id);
rdata = ep->PCIeXtorCfgRead((pcie_cap_base + 0xC));
if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  ep->print("---- EP XTOR supports ASPM L1\n");
}
else
{
  ep->error("---- EP XTOR does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}
// Check ASPM L1.1 Supported bit in L1 PM Substates Capabilities Register
 of Xtor
l1_pm_sub_ext_cap_base   = (ep->getExtCapPtr(l1sub_extcap_id));
rdata = ep->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base + 0x4));
if((rdata & 0x18)==0x18)
  ep->print("---- EP XTOR supports ASPM L1_1\n");
else
{
  ep->error("---- EP XTOR does not support ASPM L1_1. Hence, test is not
 applicable\n");
  exit(911);
}
```

Step2: Entering the ASPM L1 state

```
// Set ASPM Control bits to 2'b10 in Link Control Register of Xtor
rdata=ep->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata |=0x02;
ep->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM L1.1 Enable to 1 in L1 PM Substates Control 1 Register of
 Xtor
rdata=ep->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata |=0x08;
ep->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);

//Wait for XTOR to reach L1.Idle LTSSM state
ep->wait_for(STATE_L1_IDLE);
```

Step3: Entering the ASPM L1.1 Substate

```
// De-assert CLKREQ# signal to enter L1.1 Substate
ep->reqFor(ctrl_clk_reset_n, true);

// Wait for Xtor to reach L1.1 Substate
ep->wait_for(STATE_L1_1);
```

Feedback

Step4: Exiting the ASPM L1.1 Substate

```
// Wait for few clock cycles
ep->wait_for_cycle(500);

// Assert CLKREQ# signal to exit L1.1 Substate
ep->reqFor(ctrl_clk_reset_n, false);
```

Step5: Exiting the ASPM L1 state:

```
//Wait for XTOR to reach Recovery.RcvrLock LTSSM state
ep->wait_for(STATE_RCVRY_LOCK);

//Wait for XTOR to reach L0 LTSSM state
ep->wait_for(STATE_L0);

// Set ASPM Control bits to 2'b00 in Link Control Register of Xtor
rdata=ep->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata &=0xFFFFFFFC;
ep->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM L1.1 Enable to 0 in L1 PM Substates Control 1 Register of
 Xtor
rdata=ep->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata &=0xFFFFFFF3;
ep->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);
```

How to verify ASPM L1.2?

The below configuration parameter settings are mandatory for enabling ASPM L1.2 feature in transactor:

```
xtor->setConfigParam("PCIE_ASPML1", "true");
xtor->setConfigParam("PCIE_ASPML1_2", "true");
```

When Transactor is RC

For testing the ASPM L1.2 scenario, if the transactor is RC, perform the following steps:

Step1: Checking if ASPM L1 and ASPM L1.2 are supported

```
// Check ASPM Support bits in Link Capabilities Register of Xtor
uint32_t pcie_cap_base;
uint32_t l1_pm_sub_ext_cap_base;
uint32_t rdata;
pcie_cap_base = rc->getCapPtr(pcie_cap_id);
rdata = rc->PCIeXtorCfgRead((pcie_cap_base + 0xC));

if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  rc->print("---- RC XTOR supports ASPM L1\n");
}
```

Feedback

```
else
{
  rc->error("---- RC XTOR does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}

// Check ASPM Support bits in Link Capabilities Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base + 0xC)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  rc->print("---- EP DUT support ASPM L1 \n");
}
else
{
  rc->error("---- EP DUT does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}

// Check ASPM L1.2 Supported bit in L1 PM Substates Capabilities Register
 of Xtor
l1_pm_sub_ext_cap_base   = (ep->getExtCapPtr(l1sub_extcap_id));
rdata = rc->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base + 0x4));
if((rdata & 0x14)==0x14)
rc->print("---- RC XTOR supports ASPM L1_2\n");
else
{
  rc->error("---- RC XTOR does not support ASPM L1_2. Hence, test is not
 applicable\n");
  exit(911);
}

// Check ASPM L1.2 Supported bit in L1 PM Substates Capabilities Register
 of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((l1_pm_sub_ext_cap_base + 0x4)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
if((rdata & 0x14)==0x14)
{
   rc->print("---- EP DUT support ASPM L1.2 \n");
}
else
{
```

```
  rc->error("---- EP DUT does not support ASPM L1.2. Hence, test is not
 applicable \n");
  exit(911);
}
```

Step2: Entering the ASPM L1 state

```
 // Set ASPM Control bits to 2'b10 in Link Control Register of Xtor
rdata=rc->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata |=0x02;
rc->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM Control bits to 2'b10 in Link Control Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata |=0x02;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Set ASPM L1.2 Enable to 1 in L1 PM Substates Control 1 Register of
 Xtor
rdata=rc->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata |=0x04;
rc->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);

// Set ASPM L1.2 Enable to 1 in L1 PM Substates Control 1 Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata |=0x04;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Wait for Xtor to reach L1.Idle LTSSM state
rc->wait_for(STATE_L1_IDLE);
```

Step3: Entering the ASPM L1.2 Substate

```
// De-assert CLKREQ# signal to enter L1.2 Substate
rc->reqFor(ctrl_clk_reset_n, true);
```

Feedback

```
// Wait for Xtor to reach L1.2 Substate
rc->wait_for(STATE_L1_2);
```

Step4: Exiting the ASPM L1.2 Substate

```
// Wait for few clock cycles
rc->wait_for_cycle(500);

// Assert CLKREQ# signal to exit L1.2 Substate
rc->reqFor(ctrl_clk_reset_n, false);
```

Step5: Exiting the ASPM L1 state:

```
//Wait for XTOR to reach Recovery.RcvrLock LTSSM state
rc->wait_for(STATE_RCVRY_LOCK);

//Wait for XTOR to reach L0 LTSSM state
rc->wait_for(STATE_L0);

// Set ASPM Control bits to 2'b00 in Link Control Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata &=0xFFFFFFFC;
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((pcie_cap_base+0x10)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();

// Set ASPM Control bits to 2'b00 in Link Control Register of Xtor
rdata=rc->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata &=0xFFFFFFFC;
rc->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM L1.2 Enable to 0 in L1 PM Substates Control 1 Register of
 Xtor
rdata=rc->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata &=0xFFFFFFF3;
rc->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);

// Set ASPM L1.2 Enable to 0 in L1 PM Substates Control 1 Register of DUT
xtor_tb->record_cpl(&rdata);
tlp->setTag(xtor_tb->rc_tag);
tlp->readCfg0(((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();
rdata &=0xFFFFFFF3;
```

```
tlp->setTag(xtor_tb->rc_tag);
xtor_tb->rc_tag++;
tlp->writeCfg0(rdata,((l1_pm_sub_ext_cap_base+ 0x8)>>2),0x01,0,0);
rc->pushTxTlp(tlp);
xtor_tb->wait_all_cpl();
```

When Transactor is EP

For testing the ASPM L1.2 scenario, if the transactor is EP, perform the following steps:

Step1: Checking if ASPM L1 and ASPM L1.2 are supported

```
// Check ASPM Support bits in Link Capabilities Register of Xtor
uint32_t pcie_cap_base;
uint32_t l1_pm_sub_ext_cap_base;
uint32_t rdata;
pcie_cap_base = ep->getCapPtr(pcie_cap_id);
rdata = ep->PCIeXtorCfgRead((pcie_cap_base + 0xC));
if((rdata & 0xC00)==0xC00 ||(rdata & 0x800) == 0x800)
{
  ep->print("---- EP XTOR supports ASPM L1\n");
}
else
{
  ep->error("---- EP XTOR does not support ASPM L1. Hence, test is not
 applicable \n");
  exit(911);
}

// Check ASPM L1.2 Supported bit in L1 PM Substates Capabilities Register
 of Xtor
l1_pm_sub_ext_cap_base   = (ep->getExtCapPtr(l1sub_extcap_id));
rdata = ep->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base + 0x4));
if((rdata & 0x14)==0x14)
  ep->print("---- EP XTOR supports ASPM L1_2\n");
else
{
  ep->error("---- EP XTOR does not support ASPM L1_2. Hence, test is not
 applicable\n");
  exit(911);
}
```

Step2: Entering the ASPM L1 state

```
// Set ASPM Control bits to 2'b10 in Link Control Register of Xtor
rdata=ep->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata |=0x02;
ep->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM L1.2 Enable to 1 in L1 PM Substates Control 1 Register of
 Xtor
rdata=ep->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata |=0x04;
```

Feedback

```
ep->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);

//Wait for XTOR to reach L1.Idle LTSSM state
ep->wait_for(STATE_L1_IDLE);
```

Step3: Entering the ASPM L1.2 Substate:

```
// De-assert CLKREQ# signal to enter L1.2 Substate
ep->reqFor(ctrl_clk_reset_n, true);

// Wait for Xtor to reach L1.2 Substate
ep->wait_for(STATE_L1_2);
```

Step4: Exiting the ASPM L1.2 Substate

```
// Wait for few clock cycles
ep->wait_for_cycle(500);

// Assert CLKREQ# signal to exit L1.2 Substate
ep->reqFor(ctrl_clk_reset_n, false);
```

Step5: Exiting the ASPM L1 state

```
//Wait for XTOR to reach Recovery.RcvrLock LTSSM state
ep->wait_for(STATE_RCVRY_LOCK);

//Wait for XTOR to reach L0 LTSSM state
ep->wait_for(STATE_L0);

// Set ASPM Control bits to 2'b00 in Link Control Register of Xtor
rdata=ep->PCIeXtorCfgRead((pcie_cap_base+0x10));
rdata &=0xFFFFFFFC;
ep->PCIeXtorCfgWrite(((pcie_cap_base+0x10)),rdata);

// Set ASPM L1.2 Enable to 0 in L1 PM Substates Control 1 Register of
 Xtor
rdata=ep->PCIeXtorCfgRead((l1_pm_sub_ext_cap_base+ 0x8));
rdata &=0xFFFFFFF3;
ep->PCIeXtorCfgWrite((l1_pm_sub_ext_cap_base+ 0x8),rdata);
```

How to verify L2?

For testing the L2 scenario, if the transactor is RC, perform the following steps:

Step1: Entering the L1 state

```
// Set PowerState bits to 2'b11 in Power Management Control/Status
 Register of RC Xtor
rc->PCIeXtorCfgWrite(((rc->getCapPtr(pm_cap_id))+4),0x00000103);

// Set PowerState bits to 2'b11 in Power Management Control/Status
 Register of EP DUT
```

Feedback

```
tlp->setTag(xtor_tb->rc_tag);
tlp->writeCfg0(0x103, ((rc->getCapPtr(pm_cap_id))+4)>>2,0x01,0,0,0xf);
rc->pushTxTlp(tlp);
++(xtor_tb->rc_tag);
xtor_tb->wait_all_cpl();

// Enable entry to L2 for RC Xtor
rc->reqFor(allow_l2_entry, true);

// Wait for Xtor to reach L1.Idle LTSSM state
rc->wait_for(STATE_L1_IDLE);
```

Step2: Entering the L2 state

```
// Wait for few clock cycles
rc->wait_for_cycle(100);

// Transmit PME_Turn_OFF Msg TLP from RC Xtor
tlp->setTag(xtor_tb->rc_tag);
tlp->message(0x19,0x3,0x0,0x0);
rc->pushTxTlp(tlp);
xtor_tb->rc_tag++;

// Wait for Xtor to reach L0 State
rc->wait_for(STATE_L0);

// Wait for Xtor to reach L2 State
rc->wait_for(STATE_L2_IDLE);
```

Step3: Exiting the L2 state

```
// Wait for few clock cycles
rc->wait_for_cycle(100);

 // Wait for Xtor to reach Detect.Quiet State
rc->wait_for(STATE_DETECT_QUIET);
```

Step4: Applying reset

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

Step5: Initial Configuration

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

Feedback

Transactor is EP

For testing the L2 scenario, if the transactor is EP, perform the following steps:

Step1: Entering the L1 state

```
// Enable entry to L2 for EP Xtor
ep->reqFor(allow_l2_entry, true);

// Set PowerState bits to 2'b11 in Power Management Control/Status
 Register of EP Xtor
ep->PCIeXtorCfgWrite(((ep->getCapPtr(pm_cap_id))+4),0x00000103);

// Wait for Xtor to reach L1.Idle LTSSM state
ep->wait_for(STATE_L1_IDLE);
```

Step2: Entering the L2 state

```
// Wait for few clock cycles
ep->wait_for_cycle(100);

// Wait for Xtor to reach L2 State
ep->wait_for(STATE_L2_IDLE);
```

Step3: Exiting the L2 state

```
// Wait for few clock cycles
ep->wait_for_cycle(100);

// Initiate exit from L2 state using Wake# signal
ep->reqFor(wake_l2, true);

 // Wait for Xtor to reach L2.TransmitWake State
ep->wait_for(STATE_L2_WAKE);
```

Step4: Applying reset

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

Step5: Initial configuration:

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

Feedback

How to verify Dynamic Linkwidth Change in Non-Flit mode?

For testing the Dynamic Link width Change scenario, if the transactor is RC or EP, perform the following steps:

Step1: Checking the negotiated link width after linkup

```
// Check Negotiated Link Width bits in Link Status Register of Xtor
#define XTOR_PCIE_SVS_LINKSTATUSLINKCONTROL_OFF 0x10
uint32_t regVal;
uint8_t negotiatedLinkWidth, newWidth;
regVal =
 <rc/ep>->PCIeXtorCfgRead(<rc/ep>->getCapPtr(pcie_cap_id)+XTOR_PCIE_SVS_L
INKSTATUSLINKCONTROL_OFF);
negotiatedLinkWidth = (regVal>>20)&0x3f;
```

Step2: Initiating Link width Downsize

```
// Reducing the link width by half using linkWidthChange() API of Xtor
<rc/ep>->linkWidthChange(negotiatedLinkWidth/2);
```

Step3: Checking the new negotiated link width

```
// Check new Negotiated Link Width bits in Link Status Register of Xtor
regVal =
 <rc/ep>->PCIeXtorCfgRead(<rc/ep>->getCapPtr(pcie_cap_id)+XTOR_PCIE_SVS_L
INKSTATUSLINKCONTROL_OFF);
newWidth = (regVal>>20)&0x3f;
if(newWidth != negotiatedLinkWidth/2) {
  xtor_tb->ex_error_cnt = xtor_tb->ex_error_cnt + 1;
  <rc/ep>->print("Negotiated Link width %d does not match with expected
 Link width %d\n",newWidth,negotiatedLinkWidth/2);
}
else {
  <rc/ep>->print("Negotiated Link width is %d \n",newWidth);
}
```

Step4: Initiating link width upsize

```
// Increasing the link width to original value by using linkWidthChange()
 API of Xtor
<rc/ep>->linkWidthChange(negotiatedLinkWidth);
```

Step5: Checking the new negotiated link width

```
// Check new Negotiated Link Width bits in Link Status Register of Xtor
regVal =
 <rc/ep>->PCIeXtorCfgRead(<rc/ep>->getCapPtr(pcie_cap_id)+XTOR_PCIE_SVS_L
INKSTATUSLINKCONTROL_OFF);
newWidth = (regVal>>20)&0x3f;
if(newWidth != negotiatedLinkWidth) {
  xtor_tb->ex_error_cnt = xtor_tb->ex_error_cnt + 1;
```

Feedback

```
  <rc/ep>->print("Negotiated Link width %d does not match with expected
 Link width %d\n",newWidth,negotiatedLinkWidth);
}
else {
  <rc/ep>->print("Negotiated Link width is %d \n",newWidth);
}
```

How to verify Linkwidth Change via L0p in Flit mode?

For testing the Link width Change scenario via L0p, if the transactor is RC or EP, perform the following steps:

Step1: Checking the negotiated link width after linkup

```
// Check Negotiated Link Width bits in Link Status Register of Xtor
#define XTOR_PCIE_SVS_LINKSTATUSLINKCONTROL_OFF 0x10
uint32_t regVal;
uint8_t negotiatedLinkWidth, newWidth;
regVal =
 <rc/ep>->PCIeXtorCfgRead(<rc/ep>->getCapPtr(pcie_cap_id)+XTOR_PCIE_SVS_L
INKSTATUSLINKCONTROL_OFF);
negotiatedLinkWidth = (regVal>>20)&0x3f;
```

Step2: Initiating Link width Downsize

```
// Reducing the link width by half using initiate_L0p() API of Xtor
<rc/ep>->initiate_L0p(negotiatedLinkWidth/2);
```

Step3: Checking the new negotiated link width

```
// Check new Negotiated Link Width bits in Link Status Register of Xtor
regVal =
 <rc/ep>->PCIeXtorCfgRead(<rc/ep>->getCapPtr(pcie_cap_id)+XTOR_PCIE_SVS_L
INKSTATUSLINKCONTROL_OFF);
newWidth = (regVal>>20)&0x3f;
if(newWidth != negotiatedLinkWidth/2) {
  xtor_tb->ex_error_cnt = xtor_tb->ex_error_cnt + 1;
  <rc/ep>->print("Negotiated Link width %d does not match with expected
 Link width %d\n",newWidth,negotiatedLinkWidth/2);
}
else {
  <rc/ep>->print("Negotiated Link width is %d \n",newWidth);
}
```

Step4: Initiating Link width Upsize

```
// Increasing the link width to original value by using initiate_L0p()
 API of Xtor
<rc/ep>->initiate_L0p(negotiatedLinkWidth);
```

Step5: Checking the new negotiated link width

```
// Check new Negotiated Link Width bits in Link Status Register of Xtor
regVal =
 <rc/ep>->PCIeXtorCfgRead(<rc/ep>->getCapPtr(pcie_cap_id)+XTOR_PCIE_SVS_L
INKSTATUSLINKCONTROL_OFF);
newWidth = (regVal>>20)&0x3f;
if(newWidth != negotiatedLinkWidth) {
  xtor_tb->ex_error_cnt = xtor_tb->ex_error_cnt + 1;
  <rc/ep>->print("Negotiated Link width %d does not match with expected
 Link width %d\n",newWidth,negotiatedLinkWidth);
}
else {
  <rc/ep>->print("Negotiated Link width is %d \n",newWidth);
}
```

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 31 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

Feedback

Table 31 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_eidle	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 31 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:

Table 32 l1sub_state Encoding

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

Feedback

Optional CLKREQ# Sideband Signals

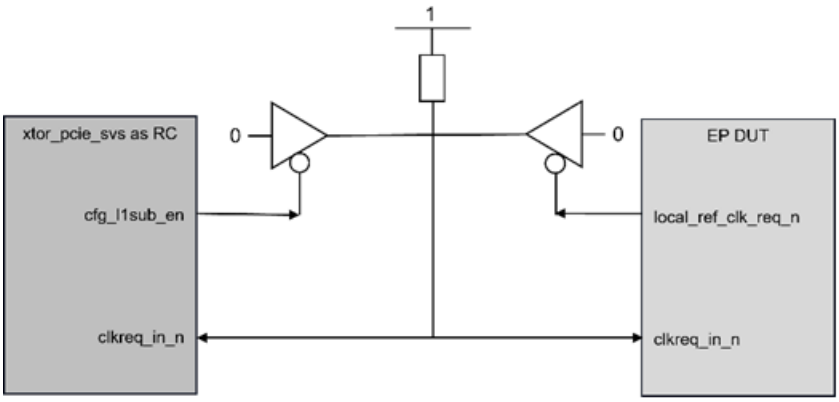
CLKREQ# is an open-drain, active low signal that is driven low by a PCIe function. It is recommended to be used for any add-in card or system board that supports power management and L1 power sub-states features.

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:

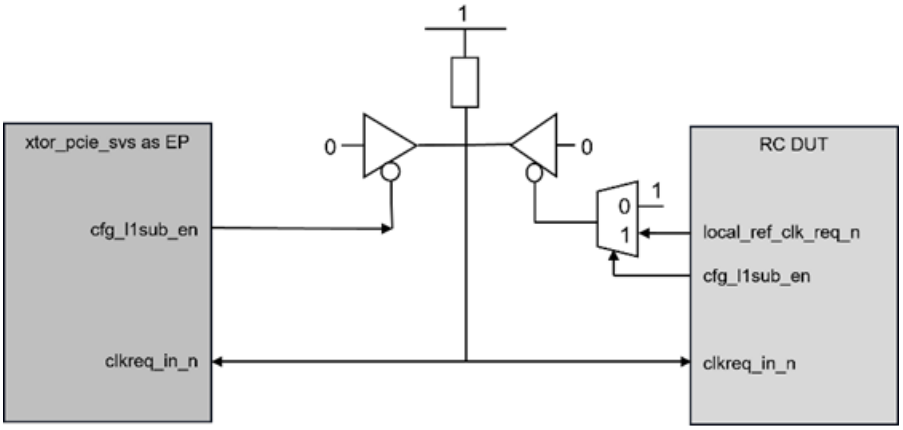
Figure 22 Transactor as Rootport



Feedback

The following figure illustrates the sideband signal when the transactor acts as an Endpoint:

Figure 23 Transactor as Endpoint



When the DUT uses the auxiliary CLKREQ# signal, the DUT wrapper is connected to both the transactor's and the DUT's clkreq interfaces and perform resolution. Construct the clkreq_in_n signal in the top-level wrapper according to the status of the cfg_l1sub_en signal:

- If one of the cfg_l1sub_en signal is active (0), set the clkreq_in_n signal to 0.
- If all cfg_l1sub_en signals are inactive (1), ensure that the clkreq_in_n signal is driven with value local_ref_clk_req_n.

```
wire rc_cfg_l1sub_en, rc_local_ref_clk_req_n;
wire ep_cfg_l1sub_en, ep_local_ref_clk_req_n;
wire rc_clkreq_oen;
wire ep_clkreq_oen;
wire clkreq_in_n;
assign rc_clkreq_oen = (rc_cfg_l1sub_en) ? rc_local_ref_clk_req_n : 1'b0;
assign ep_clkreq_oen = (ep_cfg_l1sub_en) ? ep_local_ref_clk_req_n : 1'b0;
assign clkreq_in_n = ((rc_clkreq_oen==1 ) | (ep_clkreq_oen==1 )) ? 1'b1 :
 1'b0;
```