

LogiCORE IP Aurora 64B/66B v7.3

Product Guide

PG074 December 18, 2012

Table of Contents

SECTION I: SUMMARY

IP Facts

Chapter 1: Overview

Feature Summary	9
Applications	9
Unsupported Features	10
Licensing and Ordering Information	10

Chapter 2: Product Specification

Standards	12
Performance	12
Resource Utilization	14
Port Descriptions	17

Chapter 3: Designing with the Core

General Design Guidelines	28
Using the Build Script	29
Clocking	29
Reset and Power Down	36
Top-Level Architecture	37
Flow Control	51
User K-Block Interface	57
Status, Control, and the Transceiver Interface	60
Core Features	64

SECTION II: VIVADO DESIGN SUITE

Chapter 4: Customizing and Generating the Core

GUI	67
Output Generation.....	73

Chapter 5: Constraining the Core

Device, Package, and Speed Grade Selections.....	79
Clock Frequencies	79
Clock Management	81
Clock Placement.....	81
Banking.....	81
Transceiver Placement	81
I/O Standard and Placement.....	81

Chapter 6: Detailed Example Design

Directory and File Contents.....	82
Quick Start Example Design	82
Detailed Example Design.....	83
Generating the Core.....	97
Implementing the Example Design.....	98

SECTION III: ISE DESIGN SUITE

Chapter 7: Customizing and Generating the Core

GUI	100
Output Generation.....	108

Chapter 8: Constraining the Core

Device, Package, and Speed Grade Selections.....	115
Clock Frequencies	115

Chapter 9: Detailed Example Design

Directory and File Contents.....	118
Quick Start Example Design	118
Detailed Example Design.....	119
Generating the Core.....	121
Simulating the Example Design.....	123
Implementing the Example Design.....	124

SECTION IV: APPENDICES

Appendix A: Verification, Compliance, and Interoperability

Appendix B: Migrating

Introduction	127
Overview of Major Changes	128
Block Diagrams	129
Signal Changes	130
Migration Steps	131
GUI Changes	133
Limitations	134

Appendix C: Debugging

Finding Help on Xilinx.com	135
Debug Tools	137
Simulation Debug	137
General Checks	140
Interface Debug	140

Appendix D: Generating a GT Wrapper File from the Transceiver Wizard

Case 1: Virtex-7/Kintex-7 FPGA Wrapper Compatibility	141
Case 2: Virtex-6 GTX FPGA Wrapper Compatibility	142
Case 3: Virtex-6 GTH FPGA Wrapper Compatibility	143

Appendix E: Additional Resources

Xilinx Resources	144
References	144
Technical Support	145
Revision History	145
Notice of Disclaimer	146

SECTION I: SUMMARY

IP Facts

Overview

Product Specification

Designing with the Core

Introduction

Aurora 64B/66B is a scalable, lightweight, high data rate, link-layer protocol for high-speed serial communication. The protocol is open and can be implemented using Xilinx FPGA technology.

The ISE® Design Suite and Vivado™ Design Suite produce source code for Aurora 64B/66B cores. The cores can be simplex or full-duplex, and feature one of two simple user interfaces and optional flow control.

Features

- Aurora 64B/66B cores supported on the ISE and Vivado Design Suites
- General-purpose data channels with throughput range from 600 Mb/s to over 200 Gb/s
- Supports up to 16 GTX transceivers, 12 Virtex®-6 FPGA GTH transceivers, or 16 Virtex-7 FPGA GTH transceivers
- Aurora 64B/66B protocol specification v1.2 compliant (64B/66B encoding)
- Low resource cost with very low (3%) transmission overhead
- Easy-to-use AXI4-Stream (framing) or streaming interface and optional flow control
- Automatically initializes and maintains the channel
- Full-duplex or simplex operation

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family ⁽¹⁾	Virtex-7 ⁽²⁾ , Kintex™-7 ⁽²⁾ , Virtex-6 ⁽³⁾
Supported User Interfaces	AXI4-Stream
Resources ⁽⁴⁾	See Table 2-1 through Table 2-6 .
Provided with Core	
Design Files	ISE: Verilog and VHDL Vivado: RTL
Example Design	Verilog and VHDL
Test Bench	Verilog and VHDL
Constraints File	ISE: UCF Vivado: XDC
Simulation Model	Not Provided
Supported S/W Driver	N/A
Tested Design Flows⁽⁵⁾	
Design Entry	ISE Design Suite v14.4 Vivado Design Suite ⁽⁶⁾ v2012.4
Simulation	ISE: Mentor Graphics ModelSim, Xilinx ISim, Cadence Incisive Enterprise Vivado: Mentor Graphics ModelSim, Vivado Simulator
Synthesis	ISE: XST, Synopsys Synplify Pro Vivado: Vivado Synthesis
Support	
Provided by Xilinx @ www.xilinx.com/support	

Notes:

1. For a complete listing of supported devices, see the [release notes](#) for this core.
2. For more information, see *7 Series FPGAs Overview* ([DS180](#)).
3. For more information, see *Virtex-6 Family Overview* ([DS150](#)).
4. For more complete performance data, see [Performance, page 12](#).
5. For the supported versions of the tools, see the [Xilinx Design Tools: Release Notes Guide](#).
6. Supports only 7 series devices.

Overview

Note: This core release supports only Virtex®-7 and Kintex™-7 devices. Virtex-6 families are listed for backward compatibility with previous core releases.

This product guide describes the function and operation of the LogiCORE™ IP Aurora 64B/66B v7.3 core and provides information about designing, customizing, and implementing the core.

Aurora 64B/66B is a lightweight, serial communications protocol for multi-gigabit links (Figure 1-1). It is used to transfer data between devices using one or many GTX/GTH transceivers. Connections can be *full-duplex* (data in both directions) or *simplex* (data in either one of the directions).

The LogiCORE IP Aurora 64B/66B core supports the AMBA® protocol AXI4-Stream user interface. It implements the Aurora 64B/66B protocol using the high-speed serial GTX or GTH transceivers in applicable Virtex-7, Kintex-7, and Virtex-6 LXT, SXT, and HXT, devices. The core can use up to 16 Virtex-6, Kintex-7, or Virtex-7 FPGA GTX or GTH transceivers and up to 12 GTH transceivers in a Virtex-6 HXT device running at any supported line rate to provide a low-cost, general-purpose, data channel with throughput from 600 Mb/s to over 200 Gb/s.

Aurora 64B/66B cores are verified for protocol compliance using an array of automated simulation tests.

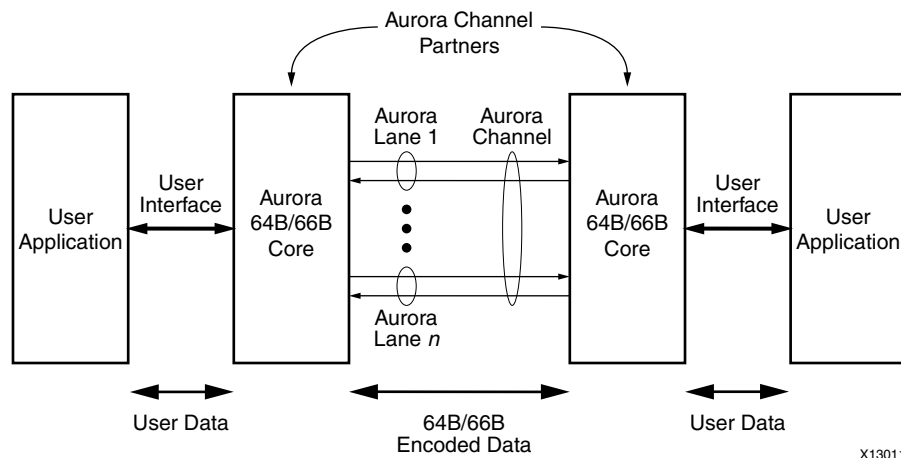


Figure 1-1: Aurora 64B/66B Channel Overview

Aurora 64B/66B cores automatically initialize a channel when they are connected to an Aurora 64B/66B channel partner. After initialization, applications can pass data across the channel as *frames* or *streams* of data. Aurora 64B/66B *frames* can be of any size, and can be interrupted any time by high priority requests. Gaps between valid data bytes are automatically filled with *idles* to maintain lock and prevent excessive electromagnetic interference. *Flow control* is optional in Aurora 64B/66B, and can be used to throttle the link partner transmit data rate, or to send brief, high-priority messages through the channel.

Streams are implemented in Aurora 64B/66B as a single, unending frame. Whenever data is not being transmitted, idles are transmitted to keep the link alive. Excessive bit errors, disconnections, or equipment failures cause the core to reset and attempt to initialize a new channel. The Aurora 64B/66B core can support a maximum of two symbols skew in the receive of a multi-lane channel. The Aurora 64B/66B protocol uses 64B/66B encoding. The 64B/66B encoding offers improved performance because of its very low (3%) transmission overhead, compared to 25% overhead for 8B/10B encoding.



RECOMMENDED: *Although the Aurora 64B/66B core is a fully-verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. For best results, previous experience building high-performance, pipelined FPGA designs using Xilinx implementation tools and user constraints files (UCF)/Xilinx Design Constraints (XDC) is recommended.*

Read [Status, Control, and the Transceiver Interface in Chapter 3](#) carefully.

Consult the PCB design requirements information in the following manuals.

- *Virtex-6 FPGA GTX Transceivers User Guide* ([UG366](#))
- *Virtex-6 FPGA GTH Transceivers User Guide* ([UG371](#))
- *7 Series FPGAs GTX/GTH Transceivers User Guide* ([UG476](#))

Contact your local Xilinx representative for a closer review and estimation for your specific requirements.

Feature Summary

The LogiCORE IP Aurora 64B/66B core implements the Aurora 64B/66B protocol using the high-speed serial transceivers on the Virtex-6 LXT, SXT, HXT, and lower-power FPGA families, and Virtex-7/Kintex-7 FPGAs. The core supports the AMBA® protocol AXI4-Stream user interface.

The Aurora 64B/66B core is based on the *Aurora 64B/66B Protocol Specification* ([SP011](#)) and uses the high-speed serial GTX or GTH transceivers in applicable Virtex-7, Kintex-7, and Virtex-6, FPGAs. The core is delivered as open-source code and supports Verilog and VHDL design environments. Each core comes with an example design and supporting modules.

Applications

Aurora 64B/66B cores can be used in a wide variety of applications because of their low resource cost, scalable throughput, and flexible data interface. Examples of Aurora 64B/66B core applications include:

- **Chip-to-chip links:** Replacing parallel connections between chips with high-speed serial connections can significantly reduce the number of traces and layers required on a PCB. The Aurora 64B/66B core provides the logic needed to use GTX/GTH transceivers, with minimal FPGA resource cost.
- **Board-to-board and backplane links:** Aurora 64B/66B uses standard 64B/66B encoding, which is the preferred encoding scheme for 10-Gigabit Ethernet making it compatible with many existing hardware standards for cables and backplanes. Aurora 64B/66B can be scaled, both in line rate and channel width, to allow inexpensive legacy hardware to be used in new, high-performance systems.
- **Simplex connections (unidirectional):** In some applications there is no need for a high-speed back channel. The Aurora 64B/66B simplex protocol provides several ways to perform unidirectional channel initialization, making it possible to use the GTX/GTH transceivers when a back channel is not available, and to reduce costs due to unused full-duplex resources.
- **ASIC applications:** Aurora 64B/66B is not limited to FPGAs, and can be used to create scalable, high-performance links between programmable logic and high-performance ASICs. The simplicity of the Aurora 64B/66B protocol leads to low resource costs in ASICs as well as in FPGAs, and design resources like the Aurora 64B/66B bus functional model (BFM) with automated compliance testing make it easy to get an Aurora 64B/66B connection up and running. Contact Xilinx Sales or auroramkt@xilinx.com for information on licensing Aurora for ASIC applications.

Unsupported Features

There are no unsupported features in Aurora 64B/66B.

Licensing and Ordering Information

This Xilinx LogiCORE IP module is provided at no additional cost with the Xilinx Vivado™ Design Suite and ISE® Design Suite tools under the terms of the [Xilinx End User License](#). Information about this and other Xilinx LogiCORE IP modules is available at the [Xilinx Intellectual Property](#) page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your [local Xilinx sales representative](#).

To use the Aurora 64B/66B core with an application specific integrated circuit (ASIC), a separate paid license agreement is required under the terms of the [Xilinx Core License Agreement](#). Contact Aurora Marketing at auroramkt@xilinx.com for more information.

For more information, visit the [Aurora 64B/66B product page](#).

Product Specification

Figure 2-1 shows a block diagram of the implementation of the Aurora 64B/66B core.

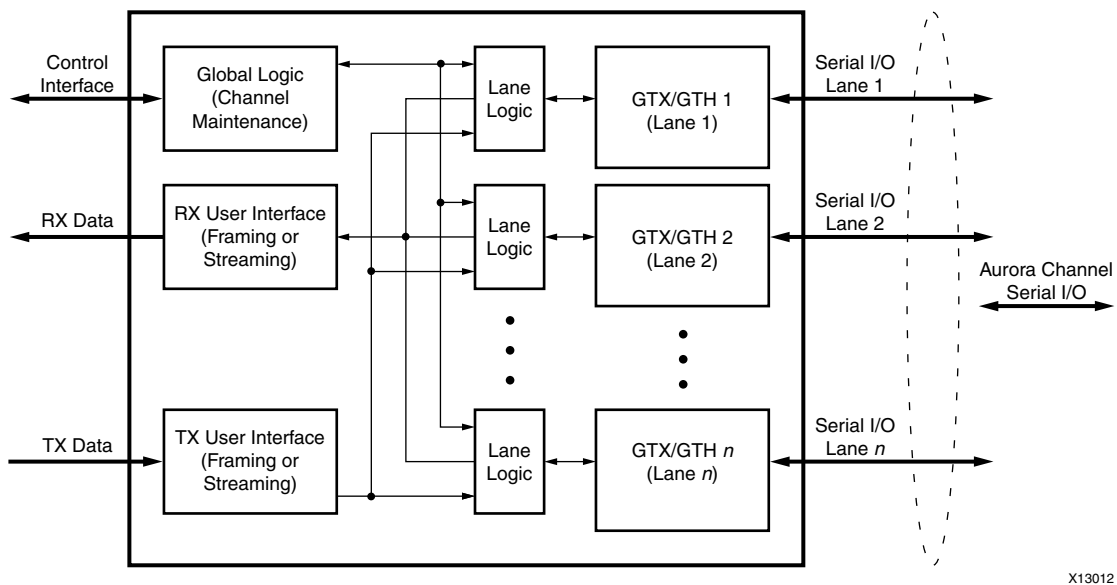


Figure 2-1: Aurora 64B/66B Core Block Diagram

The major functional modules of the Aurora 64B/66B core are:

- **Lane logic:** Each GTX/GTH transceiver is driven by an instance of the lane logic module, which initializes each individual GTX/GTH transceiver and handles the encoding and decoding of control characters and error detection.
- **Global logic:** The global logic module in the Aurora 64B/66B core performs the channel bonding for channel initialization. While the channel is operating, it keeps track of the Not Ready idle characters defined by the Aurora 64B/66B protocol and monitors all the lane logic modules for errors.
- **RX user interface:** The receive (RX) user interface moves data from the channel to the application. Streaming data is presented using a simple stream interface equipped with a data bus and *valid* and *ready* signals for flow control operation. Frames are presented using a standard AXI4-Stream interface. This module also performs flow control functions.

- **TX user interface:** The transmit (TX) user interface moves data from the application to the channel. A stream interface with valid and ready signals are used for streaming data. A standard AXI4-Stream interface is used for data frames. The module also performs flow control TX functions. The module has an interface for controlling clock compensation (the periodic transmission of special characters to prevent errors due to small clock frequency differences between connected Aurora 64B/66B cores). Normally, this interface is driven by a standard clock compensation manager module provided with the Aurora 64B/66B core, but it can be turned off, or driven by custom logic to accommodate special needs.

Standards

The Aurora 64B/66B core is compliant with the *Aurora 64B/66B Protocol Specification v1.2* ([SP011](#)).

Performance

This section details the performance information for various core configurations.

Maximum Frequencies

The Aurora 64B/66B cores listed in [Table 2-1, page 14](#) through [Table 2-4, page 15](#) were run at 156.25 MHz in devices with speed grades ranging from -1 to -3.

Latency

Latency through an Aurora 64B/66B core is caused by pipeline delays through the protocol engine (PE) and through the GTX/GTH transceivers. The PE pipeline delay increases as the AXI4-Stream interface width increases. The GTX/GTH transceivers delays are fixed per the features of the GTX/GTH transceivers.

This section outlines expected latency for the Aurora 64B/66B core AXI4-Stream user interface in terms of USER_CLK cycles for Virtex®-7/Kintex™-7 FPGA GTX/GTH transceiver based designs. For the purposes of illustrating latency, the Aurora 64B/66B modules are partitioned into GTX/GTH transceivers logic and protocol engine (PE) logic implemented in the FPGA logic.

Note: [Figure 2-2 in Latency of the Frame Path](#) does not include the latency incurred due to the length of the serial connection between each side of the Aurora 64B/66B channel.

Latency of the Frame Path

Figure 2-2 illustrates the latency of the frame path.

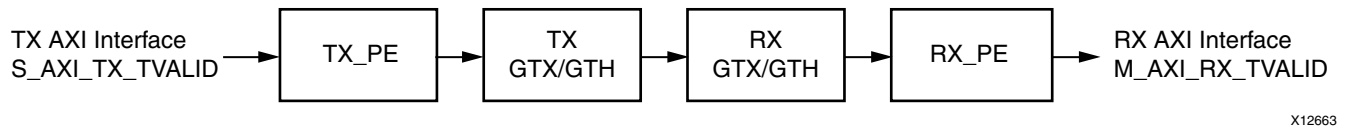


Figure 2-2: Latency of the Frame Path

Maximum latency for designs using GTX transceivers from first assertion on S_AXI_TX_TVALID to M_AXI_RX_TVALID is approximately 37 USER_CLK cycles in simulation.

Maximum latency for designs using GTH transceivers from first assertion on S_AXI_TX_TVALID to M_AXI_RX_TVALID is approximately 45 USER_CLK cycles in simulation.

The pipeline delays are designed to maintain the clock speed.

Throughput

Aurora 64B/66B core throughput depends on the number of the transceivers and the target line rate of the transceivers selected. Throughput varies from 0.58 Gb/s to 203.3 Gb/s for a single-lane design to a 16-lane design, respectively. The throughput was calculated using 3% overhead of Aurora 64B/66B protocol encoding and 0.6 Gb/s to 13.1 Gb/s line rate range.

Resource Utilization

Table 2-1 through Table 2-6 show the number of look-up tables (LUTs) and flip-flops (FFs) used in selected Aurora 64B/66B *framing* and *streaming* modules in the ISE® and Vivado™ Design Suite. The Aurora 64B/66B core is also available in configurations not shown in the tables. The tables do not include the additional resource usage for flow controls. Resource utilization in the following tables do not include the additional resource usage for the example design modules, such as FRAME_GEN and FRAME_CHECK.

Table 2-1: Virtex-7 Family GTX Transceiver Resource Usage for Streaming

Virtex-7 Family (GTX Transceiver)		Streaming		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
1	LUTs	410	190	339
	FFs	697	229	519
2	LUTs	781	195	698
	FFs	1216	229	1037
4	LUTs	1450	253	1301
	FFs	2193	231	2011
8	LUTs	2736	156	2606
	FFs	4144	235	3959
16	LUTs	5321	164	5091
	FFs	8049	244	7855

Table 2-2: Virtex-7 Family GTX Transceiver Resource Usage for Framing

Virtex-7 Family (GTX Transceiver)		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
1	LUTs	448	190	345
	FFs	697	229	519
2	LUTs	779	194	708
	FFs	1218	229	1039
4	LUTs	1448	253	1300
	FFs	2196	231	2014
8	LUTs	2750	148	2681
	FFs	4144	235	3959

Table 2-2: Virtex-7 Family GTX Transceiver Resource Usage for Framing (Cont'd)

Virtex-7 Family (GTX Transceiver)		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
16	LUTs	5416	162	5146
	FFs	8048	244	7855

Table 2-3: Virtex-6 LXT/SXT/HXT Family GTX Transceiver Resource Usage for Streaming

Virtex-6 LXT/SXT/HXT Family (GTX Transceiver)		Streaming		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
1	LUTs	474	147	395
	FFs	804	244	613
2	LUTs	921	153	810
	FFs	1411	245	1217
4	LUTs	1722	152	1558
	FFs	2785	247	2363
8	LUTs	3327	158	3123
	FFs	5386	251	4655
16	LUTs	6247	566	5773
	FFs	10536	1369	9239

Table 2-4: Virtex-6 LXT/SXT/HXT Family GTX Transceiver Resource Usage for Framing

Virtex-6 LXT/SXT/HXT Family (GTX Transceiver)		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
1	LUTs	514	153	402
	FFs	804	244	613
2	LUTs	935	144	435
	FFs	1411	247	813
4	LUTs	1705	151	1562
	FFs	2785	247	2363
8	LUTs	3510	142	3145
	FFs	5386	251	4655

Table 2-4: Virtex-6 LXT/SXT/HXT Family GTX Transceiver Resource Usage for Framing (Cont'd)

Virtex-6 LXT/SXT/HXT Family (GTX Transceiver)		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
16	LUTs	6363	575	5867
	FFs	10586	1369	9239

Table 2-5: Virtex-6 HXT Family GTH Transceiver Resource Usage for Framing

Virtex-6 HXT Family GTH Transceiver		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX Only	RX Only
1	LUTs	2698	1691	1066
	FFs	1320	571	909
2	LUTs	4946	3187	2035
	FFs	2485	967	1705
4	LUTs	9637	6051	3927
	FFs	4773	1780	3237
8	LUTs	19254	12099	7859
	FFs	9481	3550	6430

Table 2-6: Virtex-6 HXT Family GTH Transceiver Resource Usage for Streaming

Virtex-6 HXT Family GTH Transceiver		Streaming		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX Only	RX Only
1	LUTs	2703	1690	1137
	FFs	1311	585	909
2	LUTs	4993	3126	2019
	FFs	2485	996	1705
4	LUTs	9589	6009	3886
	FFs	4773	1827	3237
8	LUTs	18836	12675	7729
	FFs	9482	3500	6430

Port Descriptions

The parameters used to generate each Aurora 64B/66B core determine the interfaces available (Figure 2-3) for that specific core. The Aurora 64B/66B cores have four to seven interfaces:

- [User Interface, page 18](#)
- [User Flow Control Interface, page 20](#)
- [Native Flow Control Interface, page 21](#)
- [User K-Block Interface, page 22](#)
- [GTX/GTH Transceiver Interface, page 25](#)
- [Clock Interface, page 26](#)
- [Clock Compensation Interface, page 27](#)

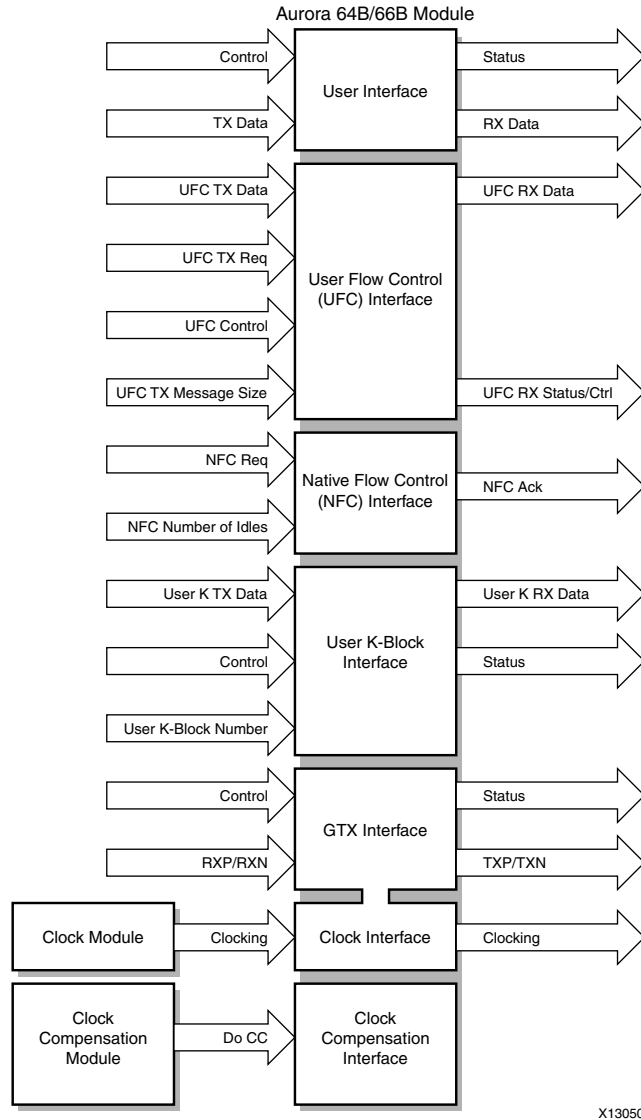


Figure 2-3: Top-Level Interface

User Interface

This interface includes all the ports needed to read and write *streaming* or *framed* data to and from the Aurora 64B/66B core. AXI4-Stream ports are used if the Aurora 64B/66B core is generated with a framing interface; for streaming modules, the interface consists of a simple set of data ports with data valid and ready ports. Full-duplex cores include ports for both transmit (TX) and receive (RX); simplex cores use only the ports they require in the direction they support. The width of the data ports in all interfaces depends on the number of GTX/GTH transceivers used by the core. CRC is computed on the data interface for every frame in the framing interface, if the CRC32 option is selected.

Framing Interface Ports (AXI4-Stream)

Table 2-7 lists the AXI4-Stream TX data ports and their descriptions.

Table 2-7: AXI4-Stream User I/O Ports (TX)

Name	Direction	Description
S_AXI_TX_TDATA[0:(64n-1)]	Input	Outgoing data (Ascending bit order).
S_AXI_TX_TREADY	Output	Asserted (active-High) during clock edges when signals from the source are accepted (if S_AXI_TX_TVALID is also asserted). Deasserted (active-Low) on clock edges when signals from the source are ignored.
S_AXI_TX_TLAST	Input	Signals the end of the frame (active-High).
S_AXI_TX_TKEEP[0:8n-1]	Input	Specifies the number of valid bytes in the last data beat; valid only while S_AXI_TX_TLAST is asserted. The Aurora core supports continuous aligned stream and continuous unaligned stream of data and expects the data to be filled continuously from LSB to MSB. There cannot be invalid bytes interleaved with the valid S_AXI_TX_TDATA bus.
S_AXI_TX_TVALID	Input	Asserted (active-High) when AXI4-Stream signals from the source are valid. Deasserted (active-Low) when AXI4-Stream control signals and/or data from the source should be ignored.

See [Framing Interface](#), page 40 for more information.

Streaming Ports

Table 2-8 lists the streaming TX data ports.

Table 2-8: Streaming User I/O Ports (TX)

Name	Direction	Description
S_AXI_TX_TDATA[0:(64n-1)]	Input	Outgoing data (Ascending bit order).
S_AXI_TX_TREADY	Output	Asserted (active-High) during clock edges when signals from the source are accepted (if S_AXI_TX_TVALID is also asserted). Deasserted (active-Low) on clock edges when signals from the source are ignored.
S_AXI_TX_TVALID	Input	Asserted (active-High) when AXI4-Stream signals from the source are valid. Deasserted (active-Low) when AXI4-Stream control signals and/or data from the source should be ignored.

Table 2-9 lists the streaming RX data ports. These ports are included on full-duplex and simplex RX framing cores.

Table 2-9: Streaming User I/O Ports (RX)

Name	Direction	Description
M_AXI_RX_TDATA[0:(64n-1)]	Output	Incoming data from channel partner (Ascending bit order).
M_AXI_RX_TVALID	Output	Asserted (active-High) when data and control signals from an Aurora 64B/66B core are valid. Deasserted (active-Low) when data and/or control signals from an Aurora 64B/66B core should be ignored.

See [Streaming Interface, page 49](#) for more information.

User Flow Control Interface

If the core is generated with User Flow Control (UFC) enabled, a UFC interface is created. The TX side of the UFC interface consists of a request, valid, and ready ports that are used to start a UFC message, and a port to specify the length of the message. You supply the message data to the UFC data port immediately after a UFC request, depending on valid and ready ports of the UFC interface; this in turn deasserts the ready port of the user data interface indicating that the core is no longer ready for normal data, thereby allowing UFC data to be written to the UFC data port.

The RX side of the UFC interface consists of a set of AXI4-Stream ports that allows the UFC message to be read as a frame. Full-duplex modules include both TX and RX UFC ports; simplex modules retain only the interface they need to send data in the direction they support. [Table 2-10](#) describes the ports for the UFC interface.

Table 2-10: UFC I/O Ports

Name	Direction	Description
UFC_TX_REQ	Input	Asserted (active-High) to request a UFC message be sent to the channel partner. Requests are processed after a single cycle, unless another UFC message is in progress and not on its last cycle. After a request, the S_AXI_UFC_TX_TDATA bus is ready to send data within two cycles unless interrupted by a higher priority event.
UFC_TX_MS[0:7]	Input	Specifies the number of bytes in the UFC message (the message size). The maximum UFC message size is 256. The value specified at UFC_TX_MS is one less than the actual amount of bytes transferred. For example, a value of 3 will transmit 4 bytes of data; and a value of 0 will transfer 1 byte.

Table 2-10: UFC I/O Ports (Cont'd)

Name	Direction	Description
S_AXI_UFC_TX_TREADY	Output	Asserted (active-High) when an Aurora 64B/66B core is ready to read data from the S_AXI_UFC_TX_TDATA interface. This signal is asserted one clock cycle after UFC_TX_REQ is asserted and no high priority requests in progress. S_AXI_UFC_TX_TREADY continues to be asserted while the core waits for data for the most recently requested UFC message. The signal is deasserted for CC, CB, and NFC requests, which are higher priority. While S_AXI_UFC_TX_TREADY is asserted, S_AXI_TX_TREADY is deasserted.
S_AXI_UFC_TX_TDATA[0:(64n-1)]	Input	Input bus for UFC message data to the Aurora channel. Data is read from the bus into the channel only when both S_AXI_UFC_TX_TVALID and S_AXI_UFC_TX_TREADY are asserted on a positive USER_CLK edge. If the number of bytes in the message is not an integer multiple of the bytes in the bus, on the last cycle, only the bytes needed to finish the message starting from the left of the bus are used.
S_AXI_UFC_TX_TVALID	Input	Asserted (active-High) when data on S_AXI_UFC_TX_TDATA is valid. If deasserted while S_AXI_UFC_TX_TREADY is asserted, Idle blocks are inserted in the UFC message.
M_AXI_UFC_RX_TDATA[0:(64n-1)]	Output	Incoming UFC message data from the channel partner.
M_AXI_UFC_RX_TVALID	Output	Asserted (active-High) when the values on the M_AXI_UFC_RX_TDATA port is valid. When this signal is not asserted, all values on the M_AXI_UFC_RX_TDATA port should be ignored.
M_AXI_UFC_RX_TLAST	Output	Signals (active-High) the end of the incoming UFC message.
M_AXI_UFC_RX_TKEEP[0:(8n-1)]	Output	Specifies the number of valid bytes of data presented on the M_AXI_UFC_RX_TDATA port on the last word of a UFC message. Valid only when M_AXI_UFC_RX_TLAST is asserted. Maximum size of UFC is 256 bytes.

See [User Flow Control](#), page 53 for more information.

Native Flow Control Interface

If the core is generated with native flow control (NFC) enabled, an NFC interface is created. This interface includes a request and an acknowledge port that are used to send NFC messages, an NFC XOFF bit that when asserted sends XOFF code to the lane partner to stop transmission, and a 16-bit port to specify the NFC PAUSE count (number of idle cycles requested) and NFC XOFF.

Note: NFC completion mode is not applicable to streaming designs.

Table 2-11 lists the ports for the NFC interface.

Table 2-11: NFC I/O Ports

Name	Direction	Description
S_AXI_NFC_TX_TVALID	Input	Asserted (active-High) to request an NFC message be sent to the channel partner. Must be held until S_AXI_NFC_TX_TREADY is asserted.
S_AXI_NFC_TX_TREADY	Output	Asserted (active-High) when an Aurora 64B/66B core accepts an NFC request.
S_AXI_NFC_TX_TDATA[0:15]	Input	S_AXI_NFC_TX_TDATA[8:15]: Indicates how many USER_CLK cycles the channel partner must wait before it can send data when it receives the NFC message. Must be held until S_AXI_NFC_TX_TREADY is asserted. The number of USER_CLK cycles without data is equal to S_AXI_NFC_TX_TDATA + 1. S_AXI_NFC_TX_TDATA[7] - Indicates NFC_XOFF. Assert to send an NFC_XOFF message, requesting that the channel partner stop sending data until it receives a non-XOFF NFC message or reset.

See [Native Flow Control, page 51](#) for more information.

User K-Block Interface

If the core is generated with the User K-block feature enabled, a User K interface is created. User K-blocks are special single block codes that include control blocks that are not decoded by the Aurora interface, but are instead passed directly to the user application. These blocks can be used to implement application specific control functions. The TX side consists of valid and ready ports that are used to start a User K transmission along with the block number port to indicate which of the nine User K-blocks needs to be transmitted. The User K data is transmitted after the core provides a ready for the User K interface. It also indicates to the user interface that it is no longer ready for normal data, thereby allowing User K data to be written to the User K data port. The User K blocks are single block codes.

The receive side of the User K interface consists of an RX valid signal to indicate the reception of User K-block. Full-duplex modules include both TX and RX User K ports; simplex modules retain only the interface they need to send data in the direction they support.

Table 2-12 lists the ports for the User K-block interface.

Table 2-12: User K-Block I/O Ports

Name	Direction	Description
S_AXI_USER_K_TX_TDATA[0:(64n-1)]	Input	User K-block data is 64-bit aligned. Signal Mapping per lane: S_AXI_USER_K_TX_TDATA={4'h0,USER K BLOCK NO[0:3],S_AXI_USER_K_TDATA[0:56n-1]}.
S_AXI_USER_K_TX_TVALID	Input	Asserted (active-High) when User K data on S_AXI_USER_K_TX_TDATA port is valid.
S_AXI_USER_K_TX_TREADY	Output	Asserted (active-High) when the Aurora 64B/66B core is ready to read data from the S_AXI_USER_K_TX_TDATA interface.
M_AXI_USER_K_RX_TVALID	Output	Asserted (active-High) when User K data on M_AXI_USER_K_RX_TDATA port is valid.
M_AXI_USER_K_RX_TDATA[0:(64n-1)]	Output	Receive User K-blocks from the Aurora lane is 64-bit aligned. Signal Mapping per lane: M_AXI_USER_K_RX_TDATA={4'h0,RX USER K BLOCK NO[0:4n-1],RX USER K DATA[0:56n-1]}

See [User K-Block Interface, page 57](#) for more information.

Status and Control Ports

Table 2-13 describes the function of the status and control ports for full-duplex cores.

Table 2-13: Status and Control Ports for Full-Duplex Cores

Name	Direction	Description
CHANNEL_UP	Output	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
LANE_UP[0:m-1] ⁽¹⁾	Output	Asserted (active-High) for each lane upon successful lane initialization, with each bit representing one lane. The Aurora 64B/66B core can only receive data after all LANE_UP signals are asserted.
HARD_ERR	Output	Hard error detected (active-High, asserted until Aurora 64B/66B core resets). See Table 2-16, page 25 for more details.
LOOPBACK[2:0]	Input	See the <i>Virtex-6 FPGA GTX Transceivers User Guide</i> , the <i>Virtex-6 FPGA GTH Transceivers User Guide</i> , and the <i>7 Series FPGAs GTX/GTH Transceivers User Guide</i> for details about loopback. See References in Appendix E .
POWER_DOWN	Input	Drives the power-down input to the GTX/GTH transceiver (active-High).
RESET	Input	Resets the Aurora 64B/66B core (active-High).
SOFT_ERR	Output	Soft error detected in the incoming serial stream. See Table 2-16, page 25 for more details (active-High, asserted for a single clock).
RXP[0:m-1]	Input	Positive differential serial data input pin.

Table 2-13: Status and Control Ports for Full-Duplex Cores (Cont'd)

Name	Direction	Description
RXN[0:m-1]	Input	Negative differential serial data input pin.
TXP[0:m-1]	Output	Positive differential serial data output pin.
TXN[0:m-1]	Output	Negative differential serial data output pin.

Notes:

1. m is the number of GTX/GTH transceivers

Table 2-14 describes the function of the status and control ports for simplex-TX cores.

Table 2-14: Status and Control Ports for Simplex-TX Cores

Name	Direction	Description
TX_CHANNEL_UP	Output	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
TX_LANE_UP[0:m-1] ⁽¹⁾	Output	Asserted (active-High) for each lane upon successful lane initialization, with each bit representing one lane. The Aurora 64B/66B core can only transmit data after all TX_LANE_UP signals are asserted.
TX_HARD_ERR	Output	Hard error detected (active-High, asserted until Aurora 64B/66B core resets). See Table 2-16, page 25 for more details.
POWER_DOWN	Input	Drives the power-down input to the GTX/GTH transceiver (active-High).
TX_SYSTEM_RESET	Input	Resets the Aurora 64B/66B core (active-High).
TX_SOFT_ERR	Output	Soft error detected in the transmit logic. See Table 2-16, page 25 for more details (active-High, asserted for a single clock).
TXP[0:m-1]	Output	Positive differential serial data output pin.
TXN[0:m-1]	Output	Negative differential serial data output pin.

Notes:

1. m is the number of GTX and GTH transceivers.

Table 2-15 describes the function of the status and control ports for simplex-RX cores.

Table 2-15: Status and Control Ports for Simplex-RX Cores

Name	Direction	Description
RX_CHANNEL_UP	Output	Asserted (active-High) when Aurora channel initialization is complete and the channel is ready to receive data.
RX_LANE_UP[0:m-1] ⁽¹⁾	Output	Asserted (active-High) for each lane upon successful lane initialization, with each bit representing one. The Aurora 64B/66B core can only receive data after all RX_LANE_UP signals are asserted.
RX_HARD_ERR	Output	Hard error detected (active-High, asserted until Aurora 64B/66B core resets). See Table 2-16, page 25 for more details.
POWER_DOWN	Input	Drives the power-down input to the GTX/GTH transceiver (active-High).
RX_SYSTEM_RESET	Input	Resets the Aurora 64B/66B core (active-High).

Table 2-15: Status and Control Ports for Simplex-RX Cores (Cont'd)

Name	Direction	Direction
RX_SOFT_ERR	Output	Soft error detected in the receive logic. See Table 2-16, page 25 for more details. (active-High, asserted for a single clock).
RXP[0:m-1]	Input	Positive differential serial data input pin.
RXN[0:m-1]	Input	Negative differential serial data input pin.

Notes:

1. *m* is the number of GTX and GTH transceivers.

See [Status and Control Ports, page 60](#) for more information.

GTX/GTH Transceiver Interface

This interface includes the serial I/O ports of the GTX/GTH transceivers and the control and status ports of the Aurora 64B/66B core. This interface is your access to control functions such as reset, loopback, and power down. The DRP interface can be used to access or update the serial transceiver parameters and settings through the AXI4-Lite or Native DRP interface.

Table 2-16 summarizes the error conditions that the Aurora 64B/66B core can detect and the error signals used to alert the user application.

Table 2-16: Error Signals in Full-Duplex Cores

Signal	Description
HARD_ERR/ TX_HARD_ERR/ RX_HARD_ERR	<p>TX Overflow/Underflow: The elastic buffer for TX data overflows or underflows. This can occur when the user clock and the reference clock sources are not running at the same frequency.</p> <p>RX Overflow/Underflow: The clock correction and channel bonding FIFO for RX data overflows or underflows. This can occur when the clock source frequencies for the two channel partners are not within ± 100 ppm.</p> <p>Soft Errors: There are too many soft errors within a short period of time. The block sync state machine used for alignment automatically attempts to realign if too many invalid sync headers are detected.</p>
SOFT_ERR/ TX_SOFT_ERR/ RX_SOFT_ERR	<p>Invalid SYNC Header: The 2-bit header on the 64-bit block was not a valid control or data header.</p> <p>Invalid BTF: A control block was received with an unrecognized value in the block type field (BTF). This is usually the result of a bit error.</p>

See [Error Signals in Aurora 64B/66B Cores, page 61](#) for more information.

Clock Interface



IMPORTANT: This interface is most critical for correct Aurora 64B/66B core operation. The clock interface has ports for the reference clocks that drive the GTX/GTH transceivers and ports for the parallel clocks that the Aurora 64B/66B core shares with application logic.

Table 2-17 describes the Virtex®-7, Kintex™-7, and Virtex-6 FPGA Aurora 64B/66B core clock ports. In GTX/GTH transceiver designs, the reference clock can be from GTXQ/GTHQ, which is a differential input clock for each GTX/GTH tile. The reference clock for a GTX/GTH tile is provided through the CLKIN port.

Table 2-17: Clock Ports for a Virtex-6, Virtex-7, and Kintex-7 FPGA Aurora 64B/66B Core

Name	Direction	Description
DCM_NOT_LOCKED/ MMCM_NOT_LOCKED	Input	If a PLL is used to generate clocks for the Aurora 64B/66B core, the DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal should be connected to the inverse of the PLL LOCKED signal. The clock modules provided with the Aurora 64B/66B core use the PLL for clock division. The DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal from the clock module should be connected to the DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal on the Aurora 64B/66B core.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application. The USER_CLK is the output of a BUFG whose input is derived from TX_OUT_CLK. The rate is one fourth the rate of TX_OUT_CLK, which is determined by the clock rate settings of the serial transceivers in the core.
TX_OUT_CLK	Output	Clock signal from Virtex-6 FPGA GTX/GTH transceivers or Virtex-7/Kintex-7 GTX transceiver. The GTX/GTH transceiver generates TX_OUT_CLK from its reference clock based on its PLL speed setting. This clock, when buffered, should be used as the user clock for logic connected to the Aurora 64B/66B core.
SHIM_CLK	Input	Parallel clock used by the internal synchronization logic of the serial transceivers in the Aurora 64B/66B core. SHIM_CLK is double the rate of USER_CLK. Note: This clock is not available for Virtex-6 FPGA GTH transceivers and Virtex-7/Kintex-7 FPGA GTX transceivers.
SYNC_CLK	Input	Parallel clock used by internal synchronization logic of the serial transceivers in the Aurora 64B/66B core. SYNC_CLK is double the rate of SHIM_CLK.
PLL_LOCK	Output	Active-High, asserted when TX_OUT_CLK is stable. When this signal is deasserted (Low), circuits using TX_OUT_CLK should be held in reset.

For more details on the clock interface, see [Clocking, page 29](#).

Clock Compensation Interface

This interface is included in modules that transmit data, and is used to manage clock compensation. Whenever the DO_CC port is driven High, the core stops the flow of data and flow control messages, then sends clock compensation sequences. Each Aurora 64B/66B core is accompanied by a clock compensation management module that is used to drive the clock compensation interface in accordance with the *Aurora 64B/66B Protocol Specification* ([SP011](#)). When the same physical clock is used on both sides of the channel, DO_CC should be tied Low.

All Aurora 64B/66B cores include a clock compensation interface for controlling the transmission of clock compensation sequences. [Table 2-18](#) describes the function of the clock compensation interface ports.

Table 2-18: Clock Compensation I/O Ports

Name	Direction	Description
DO_CC	Input	The Aurora 64B/66B core sends CC sequences on all lanes on every clock cycle when this signal is asserted. Connects to the DO_CC output on the CC module.

For more details on the clock compensation interface, see [Clock Compensation Interface, page 27](#).

Designing with the Core

This chapter includes guidelines and additional information to make designing with the core easier. It includes these sections:

- [General Design Guidelines](#)
- [Using the Build Script](#)
- [Clocking](#)
- [Reset and Power Down](#)
- [Top-Level Architecture](#)
- [Flow Control](#)
- [User K-Block Interface](#)
- [Status, Control, and the Transceiver Interface](#)

General Design Guidelines

All Aurora 64B/66B implementations require careful attention to system performance requirements. Pipelining, logic mappings, placement constraints and logic duplications are all methods that help boost system performance.

Keep It Registered

To simplify timing and increase system performance in an FPGA design, keep all inputs and outputs registered between the user application and the core. This means that all inputs and outputs from user application should come from or connect to a flip-flop. While registering signals might not be possible for all paths, it simplifies timing analysis and makes it easier for the Xilinx tools to place-and-route the design.

Recognize Timing Critical Signals

The UCF/XDC file provided with the example design for the core identifies the critical signals and the timing constraints that should be applied.

Use Supported Design Flows

The core is delivered as Verilog and VHDL source code. The example implementation scripts provided currently use XST as synthesis tool for the example design that is delivered with the core. Other synthesis tools can be used.

Make Only Allowed Modifications

The Aurora 64B/66B core is not user modifiable. Any modifications might have adverse effects on the system timings and protocol compliance. Supported user configurations of the Aurora 64B/66B core can only be made by selecting options from the CORE Generator™ or Vivado™ IP catalog tool.

Using the Build Script

A shell script called `implement.sh` and a batch script called `implement.bat` are delivered with the Aurora 64B/66B core in the `implement` subdirectory. These scripts can be used to ease implementation of the Aurora 64B/66B core. Run the script to synthesize the Aurora 64B/66B core using XST. The design runs with the `example_design` module as the top level module, which has built-in frame generator and frame checker modules to generate and verify data integrity. Ensure that the XILINX environment variable is set properly.

Clocking

Good clocking is critical for the correct operation of the Virtex®-7, Kintex™-7, and Virtex-6 FPGA Aurora 64B/66B core. The core requires a low-jitter reference clock to drive the high-speed TX clock and clock recovery circuits in the GTX/GTH transceiver. It also requires at least one frequency-locked parallel clock for synchronous operation with the user application.

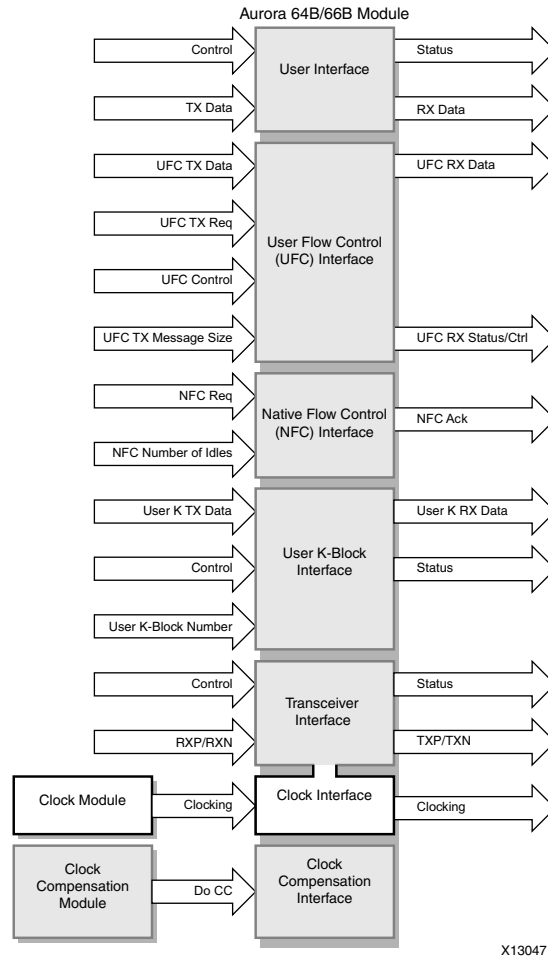


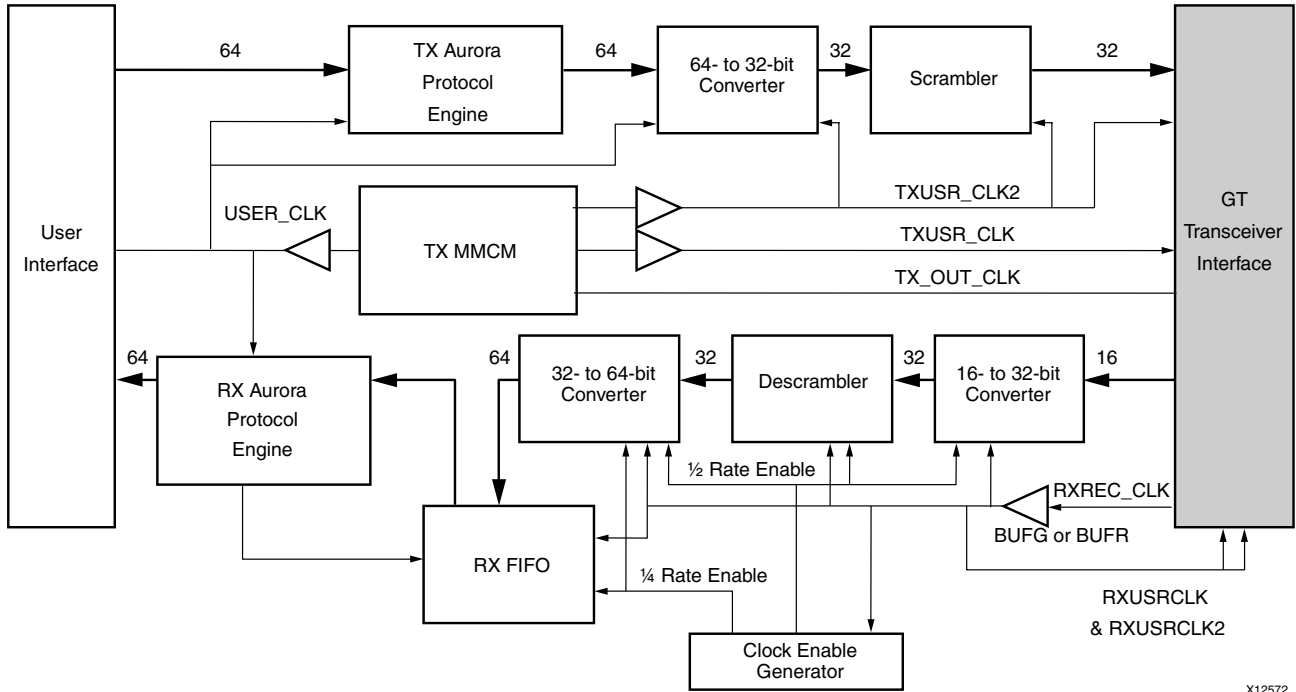
Figure 3-1: Top-Level Clocking

Each Aurora 64B/66B core is generated with an example directory that includes a design called `example_design`. This design instantiates the Aurora 64B/66B core that was generated and demonstrates a working clock configuration for the core. First-time users should examine the `aurora example design` and use it as a template when connecting the clock interface.

Clock Interface and Clocking

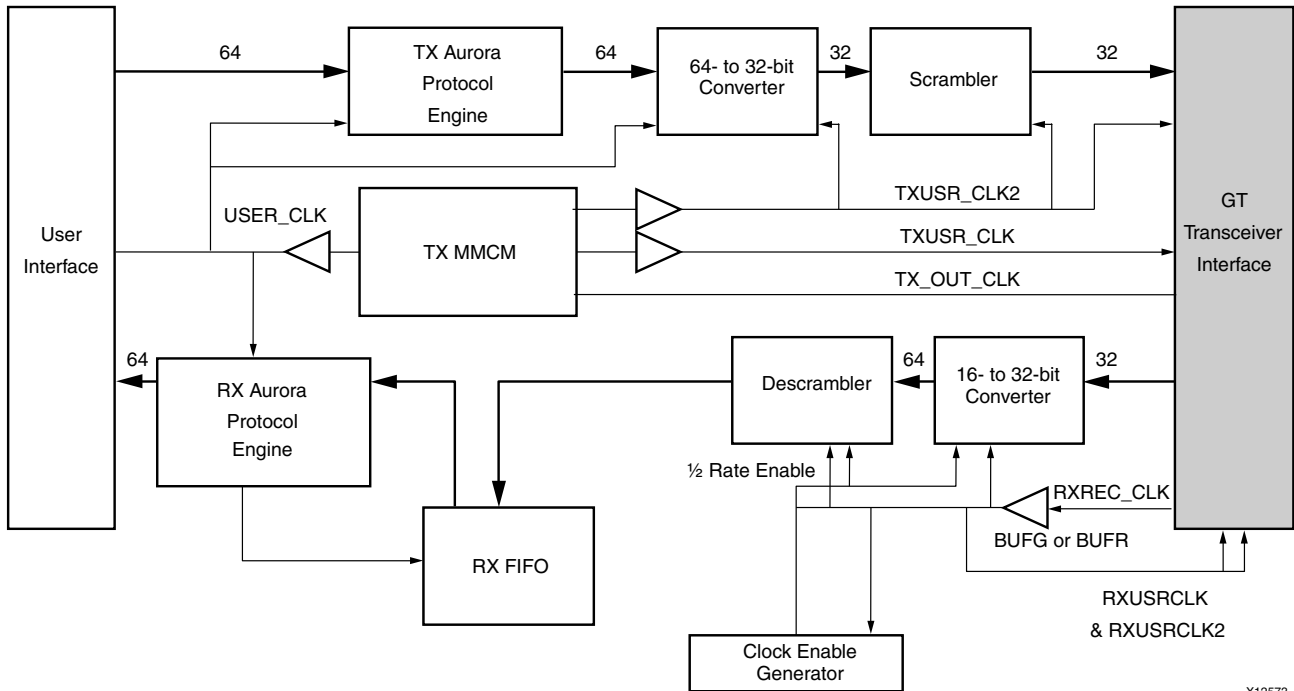
Aurora 64B/66B Clocking Architecture

Figure 3-2 shows the clocking architecture in the Aurora 64B/66B core for Virtex-6 FPGA GTX transceivers. Figure 3-3 shows the clocking architecture in the Aurora 64B/66B core for Virtex-6 FPGA GTH transceivers and Virtex-7/Kintex-7 FPGA GTX transceivers.



X12572

Figure 3-2: Aurora 64B/66B Clocking for Virtex-6 FPGA GTX Transceivers



X12573

Figure 3-3: Aurora 64B/66B Clocking for Virtex-6 FPGA GTH Transceivers and Virtex-7/Kintex-7 FPGA GTX Transceivers

Parallel Clocks

Connecting USER_CLK, SYNC_CLK, SHIM_CLK, and TX_OUT_CLK

The Virtex-7, Kintex-7, and Virtex-6 FPGA Aurora 64B/66B cores use three phase-locked parallel clocks. The first is USER_CLK, which synchronizes all signals between the core and the user application. All logic touching the core must be driven by USER_CLK, which in turn must be the output of a global clock buffer (BUFG). USER_CLK runs at one-fourth (for Virtex-6 FPGA GTX transceivers) and half (for Virtex-6 FPGA GTH transceivers and Virtex-7/Kintex-7 FPGA GTX transceivers) the rate of TX_OUT_CLK, which is determined by the Clock Settings for the design. The TX_OUT_CLK is selected so that the data rate of the parallel side of the module matches the data rate of the serial side of the module, taking into account 64B/66B encoding and decoding.

The second phase-locked parallel clock is SHIM_CLK used only for Virtex-6 FPGA GTX transceiver designs. This clock must also come from a BUFG and is half the rate of the TX_OUT_CLK. It is connected directly to the Aurora 64B/66B core to drive the internal synchronization logic of the high-speed serial GTX transceivers.

The third phase-locked parallel clock is SYNC_CLK. This clock must also come from a BUFG and is equal to the rate of TX_OUT_CLK. It is also connected to the Aurora 64B/66B core to drive the internal synchronization logic of the serial transceiver.

To make it easier to use the two parallel clocks, a clock module is provided in a subdirectory called `clock_module` under `example_design`. The ports for this module are described in [Table 2-17, page 26](#); If the clock module is used, the `DCM_NOT_LOCKED/MMCM_NOT_LOCKED` signal should be connected to the `DCM_NOT_LOCKED/MMCM_NOT_LOCKED` output of the clock module, `TX_OUT_CLK` should connect to the clock module `CLK` port, and `PLL_LOCK` should connect to the clock module `PLL_NOT_LOCKED` port. If the clock module is not used, connect the `DCM_NOT_LOCKED/MMCM_NOT_LOCKED` signal to the inverse of the `LOCKED` signal from any PLL used to generate either of the parallel clocks, and use the `PLL_LOCK` signal to hold the PLLs in reset during stabilization if `TX_OUT_CLK` is used as the PLL source clock.

Usage of BUFG in the Aurora 64B/66B Core

The Aurora 64B/66B core uses four BUFGs for a given core configuration using Virtex-6 FPGA GTX transceivers or Virtex-7/Kintex-7 FPGA GTX transceivers, and uses six BUFGs for a given configuration using Virtex-6 FPGA GTH transceivers or Virtex-7 FPGA GTH transceivers. Aurora 64B/66B is an eight-byte-aligned protocol, and the datapath from the user interface is 8-bytes aligned. For Virtex-6 FPGAs with GTX transceivers, the core configures the transmit path as four bytes and the receive path as two bytes. It generates the respective clocks and enables to conserve the clocking resource available in the device. For Virtex-6 FPGA GTH transceiver and Virtex-7/Kintex-7 FPGA GTX/GTH transceiver devices, the core configures the transmit path as eight bytes and the receive path as four bytes. It generates the clock enable on the receive side for proper sampling of data.

The CB/CC logic is internal to the core, which is primarily based on the received recovered clock from the serial transceiver. The BUFG usage is constant for any core configuration and does not increase with any core feature.

Reference Clocks for FPGA Designs

Aurora 64B/66B cores require low-jitter reference clocks for generating and recovering high-speed serial clocks in the GTX/GTH transceivers. Each reference clock can be set to Virtex-6, Virtex-7, and Kintex-7 FPGA reference clock input ports: GTXQ/GTHQ. Reference clocks should be driven with high-quality clock sources whenever possible to decrease jitter and prevent bit errors. DCMs should never be used to drive reference clocks, because they introduce too much jitter.

For multi-lane designs, the Aurora 64B/66B wizard allows selecting clocks one Quad above and one Quad below the selected Quad per north-south clocking criteria. A second reference clock source can be selected if the quad selection exceeds the 3-Quad boundary. A third clock source is included for Virtex-6 FPGA GTH transceivers based on lane selection and for line rates from 9.92 Gb/s. For details on north-south clocking, see the *Virtex-6 FPGA GTX Transceivers User Guide* ([UG366](#)), the *Virtex-6 FPGA GTH Transceivers User Guide* ([UG371](#)), and the *7 Series FPGAs GTX/GTH Transceivers User Guide* ([UG476](#)).

Clock Compensation

The clock compensation feature allows up to ± 100 ppm difference in the reference clock frequencies used on each side of an Aurora channel. This feature is used in systems where a separate reference clock source is used for each device connected by the channel, and where the same `USER_CLK` is used for transmitting and receiving data.

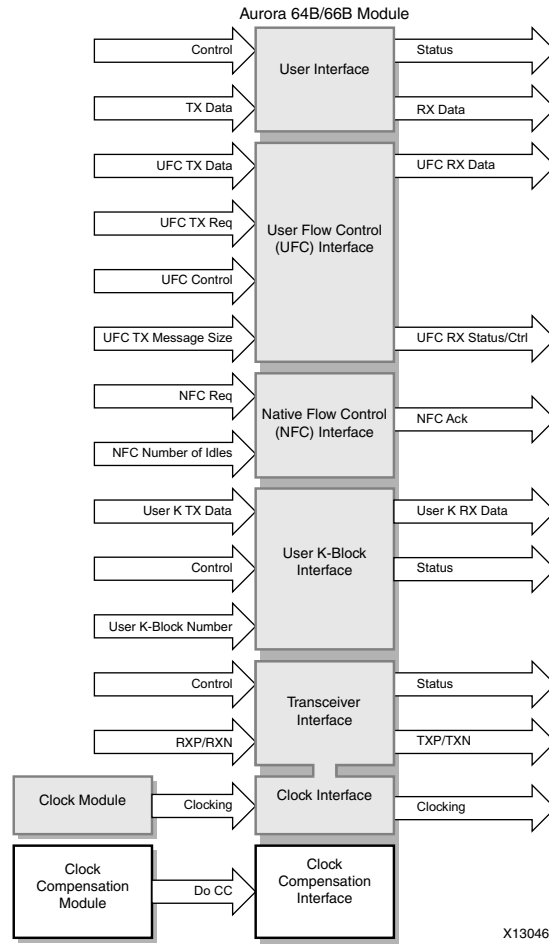


Figure 3-4: Top-Level Clock Compensation Interface

The Aurora 64B/66B core clock compensation interface enables full control over the core clock compensation features. A standard clock compensation module is generated with the Aurora 64B/66B core to provide Aurora-compliant clock compensation for systems using separate reference clock sources; users with special clock compensation requirements can drive the interface with custom logic. If the same reference clock source is used for both sides of the channel, the interface can be tied to ground to disable clock compensation.

Clock Compensation Interface

Figure 3-5 and Figure 3-6 are waveform diagrams showing how the DO_CC signal works.

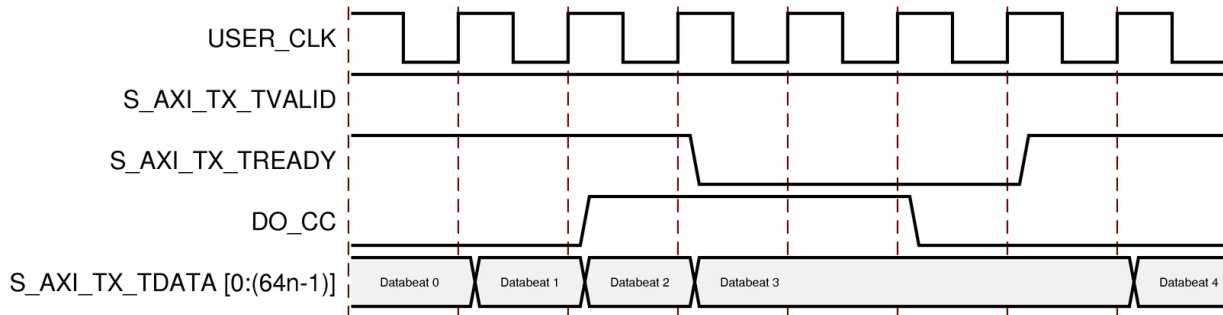


Figure 3-5: Streaming Data with Clock Compensation Inserted

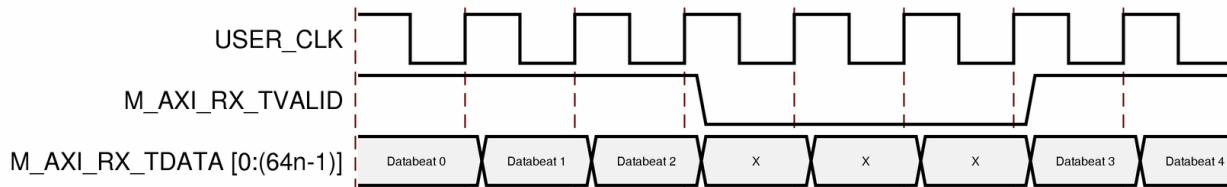


Figure 3-6: Data Reception Interrupted by Clock Compensation

The Aurora protocol specifies a clock compensation mechanism that allows up to ± 100 ppm difference between reference clocks on each side of an Aurora channel. To perform Aurora-compliant clock compensation, DO_CC must be asserted for three cycles every 10,000 cycles. While DO_CC is asserted, S_AXI_TX_TREADY is deasserted on the TX user interface while the channel is being used to transmit clock compensation sequences.

A standard clock compensation module is generated along with each Aurora 64B/66B core from the CORE Generator™ tool, in the cc_manager subdirectory under example_design. It automatically generates pulses to create Aurora compliant clock compensation sequences on the DO_CC port. This module should always be connected to the clock compensation port on the Aurora module, except in special cases. Table 3-1 shows the port description for the standard CC module.

Table 3-1: Standard CC I/O Port

Name	Direction	Description
DO_CC	Output	Connect this port to the DO_CC input of the Aurora 64B/66B core.
CHANNEL_UP	Input	Connect this port to the CHANNEL_UP output of a full-duplex core, or to the TX_CHANNEL_UP output of a TX-only simplex port.

Clock compensation is not needed when both sides of the Aurora channel are being driven by the same clock (see [Figure 3-6, page 35](#)) because the reference clock frequencies on both sides of the module are locked. In this case, DO_CC should be tied to ground.

Other special cases when the standard clock compensation module is not appropriate are possible. The DO_CC port can be used to send clock compensation sequences at any time, for any duration to meet the needs of specific channels. The most common use of this feature is scheduling clock compensation events to occur outside of frames, or at specific times during a stream to avoid interrupting data flow.



IMPORTANT: *In general, customizing the clock compensation logic is not recommended, and when it is attempted, it should be performed with careful analysis, testing, and consideration of these guidelines:*

- Clock compensation sequences should last at least three cycles to ensure they are recognized by all receivers.
- Be sure the duration and period selected are sufficient to correct for the maximum difference between the frequencies of the clocks that will be used.
- Do not perform multiple clock compensation sequences within 8 cycles of one another.

Reset and Power Down

Reset

The RESET/ TX_SYSTEM_RESET/ RX_SYSTEM_RESET signals on the control and status interface are used to set the Aurora 64B/66B core to a known starting state. Resetting the core stops any channels that are currently operating; after reset, the core attempts to initialize a new channel.

On full-duplex modules, the RESET signal resets both the TX and RX sides of the channel when asserted on the positive edge of USER_CLK.

Power Down

When POWER_DOWN is asserted, the GTX/GTH transceivers in the Aurora 64B/66B core are turned off, putting them into a non-operating low-power mode. When POWER_DOWN is deasserted, the core automatically resets. Be careful when asserting this signal on cores that use TX_OUT_CLK (see [Clock Interface and Clocking, page 30](#)). TX_OUT_CLK stops when the GTX/GTH transceivers are powered down. See the *Virtex-6 FPGA GTX Transceivers User Guide*, the *Virtex-6 FPGA GTH Transceivers User Guide*, and the *7 Series FPGAs GTX/GTH Transceivers User Guide* for details about powering down GTX/GTH transceivers.

Timing

Figure 3-7 shows the timing for the RESET signal. In a quiet environment, t_{CU} is generally less than 500 clocks; In a noisy environment, t_{CU} can be much longer.

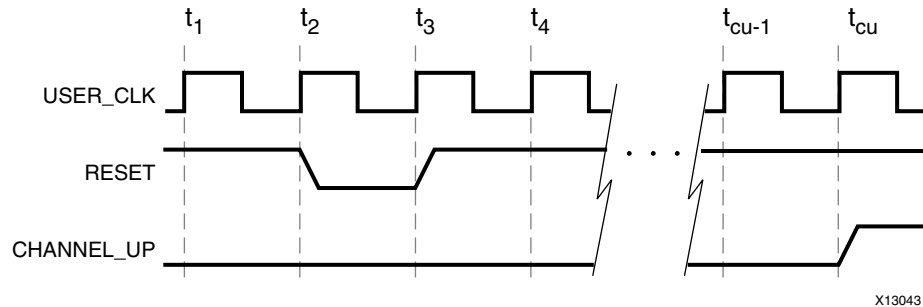


Figure 3-7: Reset and Power Down Timing

Top-Level Architecture

The Aurora 64B/66B top-level (block level) file instantiates the Aurora lane module, the TX and RX AXI4-Stream modules, the global logic module, and the wrapper for the GTX/GTH transceiver. This top-level wrapper file is instantiated in the example design file together with clock, reset circuit, and frame generator and checker modules.

Figure 3-8 shows the Aurora 64B/66B top level for a duplex configuration. The top-level file is the starting point for a user design.

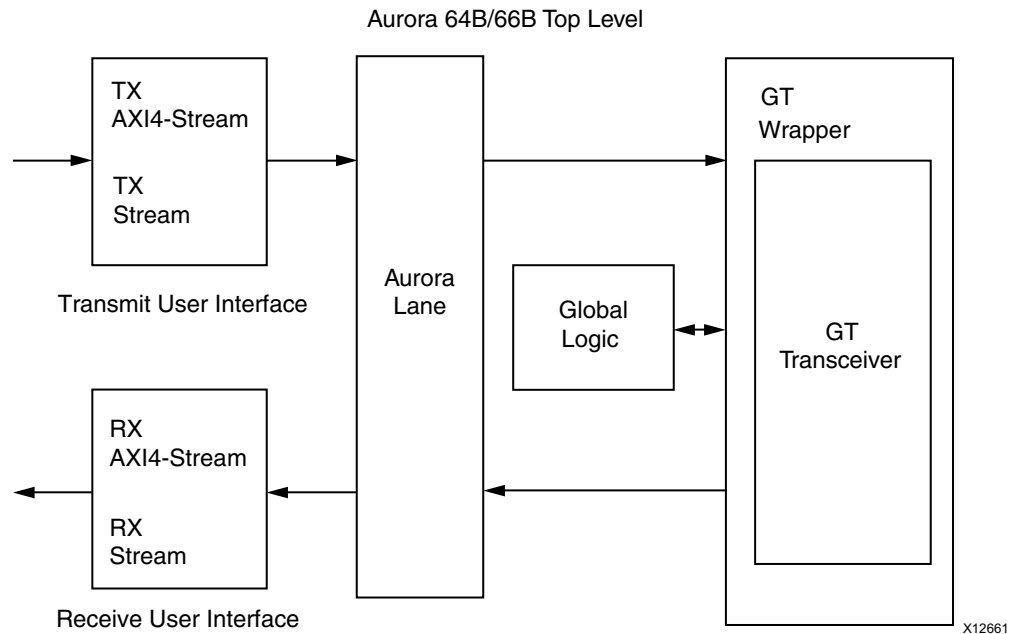


Figure 3-8: Top-Level Architecture

The following sections describe the streaming and framing interfaces in detail. User interface logic should be designed such that it complies with timing requirements of the respective interface as explained in the subsequent sections.

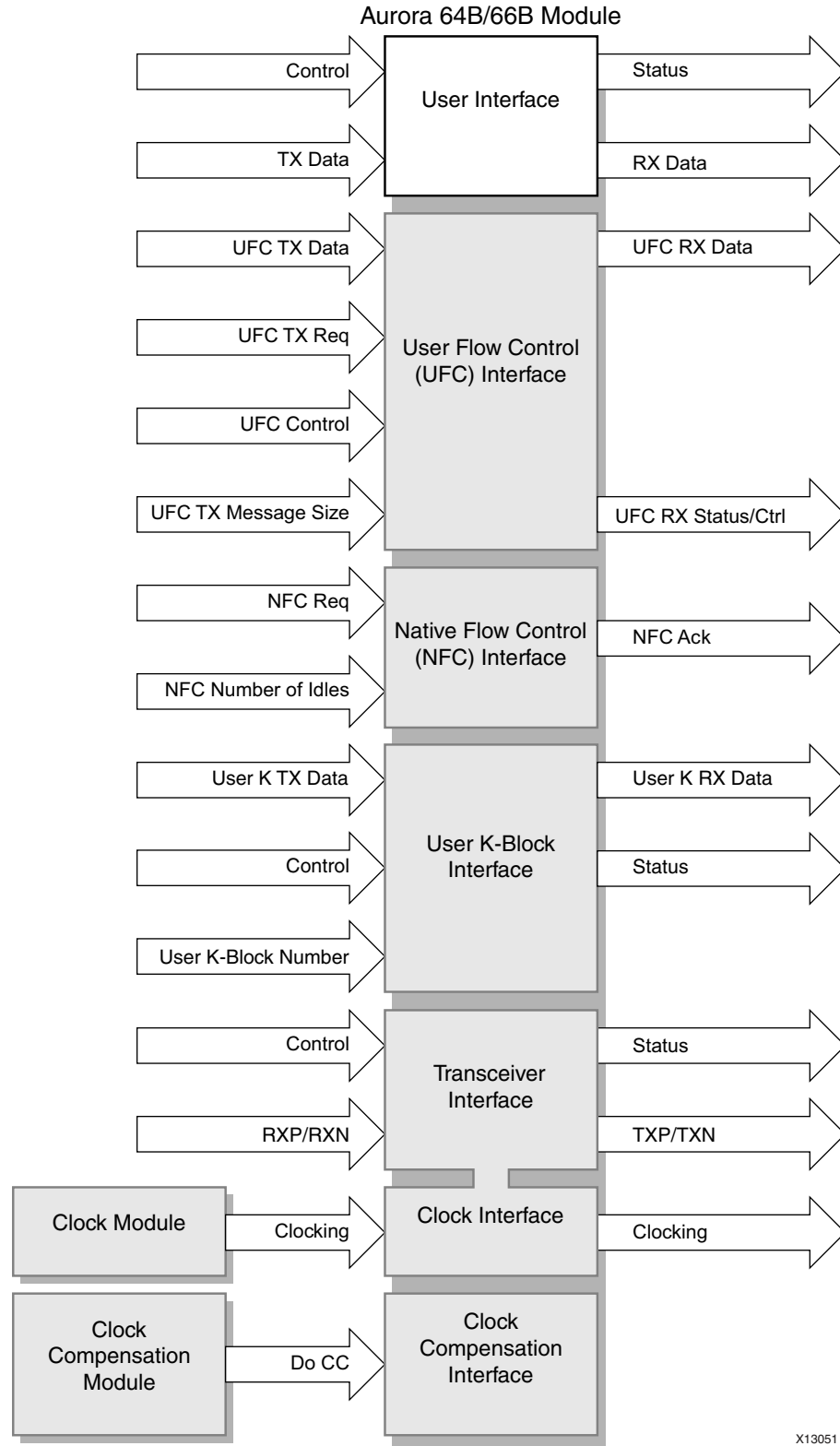


Figure 3-9: Top-Level User Interface

Note: The user interface signals vary depending upon the selections made when generating an Aurora 64B/66B core in the CORE Generator™ tool.

Framing Interface

Figure 3-10 shows the framing user interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for TX and RX data.

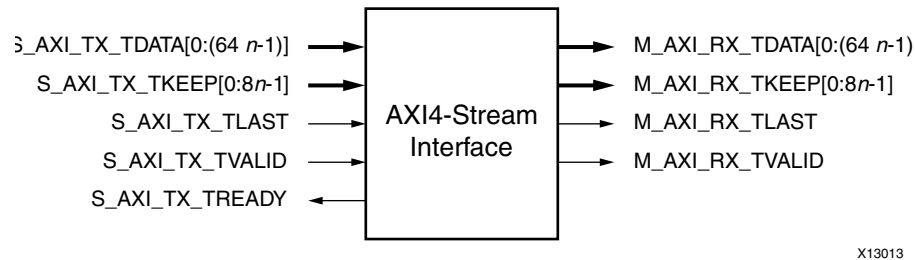


Figure 3-10: Aurora 64B/66B Core Framing Interface (AXI4-Stream)

To transmit data, the user application should manipulate the control signals to cause the core to do the following:

- Take data from the user application on the `S_AXI_TX_TDATA` bus
- Encapsulate and stripe the data across lanes in the Aurora channel (`S_AXI_TX_TLAST`)
- Pause data (that is, insert idles) (`S_AXI_TX_TVALID`)

When the core receives data, it does the following:

- Detects and discards control bytes (idles, clock compensation)
- Asserts framing signals (`M_AXI_RX_TLAST`)
- Recovers data from the lanes
- Assembles data for presentation to the user application on the `M_AXI_RX_TDATA` bus along with valid number of bytes (`M_AXI_RX_TKEEP`) during the `M_AXI_RX_TLAST` cycle

AXI4-Stream Bit Ordering

The AXI4-Stream user interface of Aurora 64B/66B cores uses ascending ordering. The cores transmit and receive the most significant bit of the least significant byte first. Figure 3-11 shows the organization of an n -byte example of the AXI4-Stream data interfaces of an Aurora 64B/66B core.

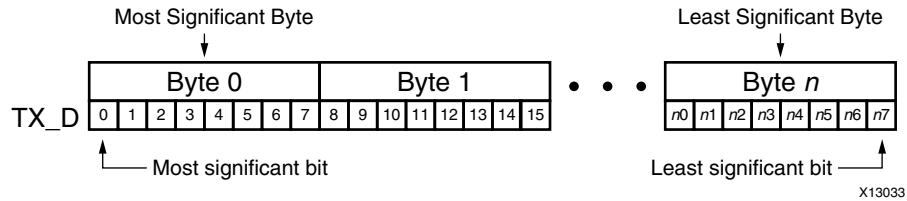


Figure 3-11: AXI4-Stream Interface Bit Ordering

Transmitting Data

AXI4-Stream is a synchronous interface. The Aurora 64B/66B core samples the data on the interface only on the positive edge of `USER_CLK`, and only on the cycles when both `S_AXI_TX_TREADY` and `S_AXI_TX_TVALID` are asserted (active-High).

When AXI4-Stream signals are sampled, they are only considered valid if `S_AXI_TX_TVALID` and `S_AXI_TX_TREADY` signals are asserted. The user application can deassert `S_AXI_TX_TVALID` on any clock cycle; this causes the Aurora core to ignore the AXI4-Stream input for that cycle. If this occurs in the middle of a frame, idle symbols are sent through the Aurora channel, which eventually result in a idle cycles during the frame when it is received at the RX user interface.

AXI4-Stream data is only valid when it is framed. Data outside of a frame is ignored. To end a frame, assert `S_AXI_TX_TLAST` while the last word (or partial word) of data is on the `S_AXI_TX_TDATA` port. If the CRC option is selected, CRC is calculated and inserted into the data stream after the last data word. This re-calculates `S_AXI_TX_TKEEP` based on the number of valid CRC bytes and asserts `S_AXI_TX_TLAST` accordingly.

Data Strobe

AXI4-Stream allows the last word of a frame to be a partial word. This lets a frame contain any number of bytes, regardless of the word size. The `S_AXI_TX_TKEEP` bus is used to indicate the number of valid bytes in the final word of the frame. The bus is only used when `S_AXI_TX_TLAST` is asserted. `TKEEP` is the number of valid bytes in the `S_AXI_TX_TDATA` bus. `TKEEP` associates validity to a particular byte in the last data beat of a frame. If `TKEEP` is "0F" in the last beat of data with `S_AXI_TX_TLAST` asserted high, then 4 (LSB bytes) out of 8 bytes are valid and byte4 to byte7 are not valid. All 1s in the `S_AXI_TX_TKEEP` value indicate all bytes in the `S_AXI_TX_TDATA` port are valid. `S_AXI_TX_TKEEP` does not specify the position of the valid bytes, but is the number of valid bytes on the last beat of data with `S_AXI_TX_TLAST` asserted. Core expects `TKEEP` to be left aligned from LSB. See [Appendix B, Migrating](#) for limitations on the types of data stream supported by the core.

Aurora 64B/66B Frames

The TX submodule translates each user frame that it receives through the TX interface to an Aurora 64B/66B frame. The core starts an Aurora 64B/66B frame by sending a data block with the first word of data, and ends the frame by sending a separator block containing the last bytes of the frame. Idle blocks are inserted whenever data is not available. Blocks are eight bytes of scrambled data or control information with a two-bit control header (a total of 66 bits). All data in Aurora 64B/66B is sent as part of a data block or a separator block (a separator block consists of a count field, indicating how many bytes are valid in that particular block).

Table 3-2 shows a typical Aurora 64B/66B frame with an even number of data bytes.

Length

The user application controls the channel frame length by manipulating the `S_AXI_TX_TVALID` and `S_AXI_TX_TLAST` signals. The Aurora 64B/66B core converts these to data blocks, idle blocks, and separator blocks, as shown in Table 3-2.

Table 3-2: Typical Channel Frame

Data Byte 0	Data Byte 1	Data Byte 2	Data Byte 3	...	Data Byte $n - 2$	Data Byte $n - 1$	Data Byte n
SEP (1E)	Count (4)	Data Byte 0	Data Byte 1	Data Byte 2	Data Byte 3	x	x

Example A: Simple Data Transfer

Figure 3-12 shows an example of a simple data transfer on a AXI4-Stream interface that is n bytes wide. In this case, the amount of data being sent is $3n$ bytes and so requires three data beats. `S_AXI_TX_TREADY` is asserted, indicating that the AXI4-Stream interface is ready to transmit data. When the Aurora 64B/66B is not sending data, it sends idle blocks.

To begin the data transfer, the user application asserts `S_AXI_TX_TVALID` and provides the first n bytes of the user frame. Because `S_AXI_TX_TREADY` is already asserted, data transfer begins on the next clock edge. The data bytes are placed in data blocks and transferred through the Aurora channel.

To end the data transfer, the user application asserts `S_AXI_TX_TLAST`, `S_AXI_TX_TVALID`, the last data bytes, and the appropriate value on the `S_AXI_TX_TKEEP` bus. In this example, `S_AXI_TX_TKEEP` is set to `F` to indicate that all bytes are valid in the last data beat. The Aurora 64B/66B core sends the final word of data in data blocks, and must send an empty separator block on the next cycle to indicate the end of the frame. `S_AXI_TX_TREADY` is reasserted on the next cycle so that more data transfers can continue. As long as there is no new data, the Aurora 64B/66B core sends idles.

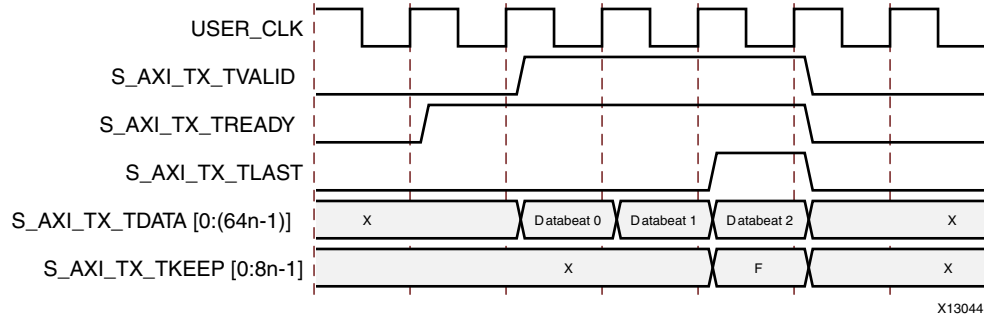


Figure 3-12: Simple Data Transfer

Example B: Data Transfer with Pause

Figure 3-13 shows how the user application can pause data transmission during a frame transfer. In this example, the user application is sending $3n$ bytes of data, and pauses the data flow after the first n bytes. After the first data word, the user application deasserts **S_AXI_TX_TVALID**, causing the TX Aurora 64B/66B core to ignore all data on the bus and transmit idle blocks instead. The pause continues until **S_AXI_TX_TVALID** is deasserted.

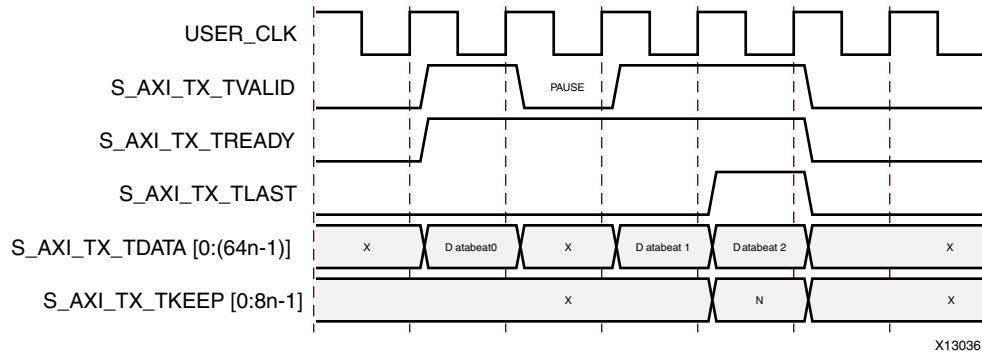
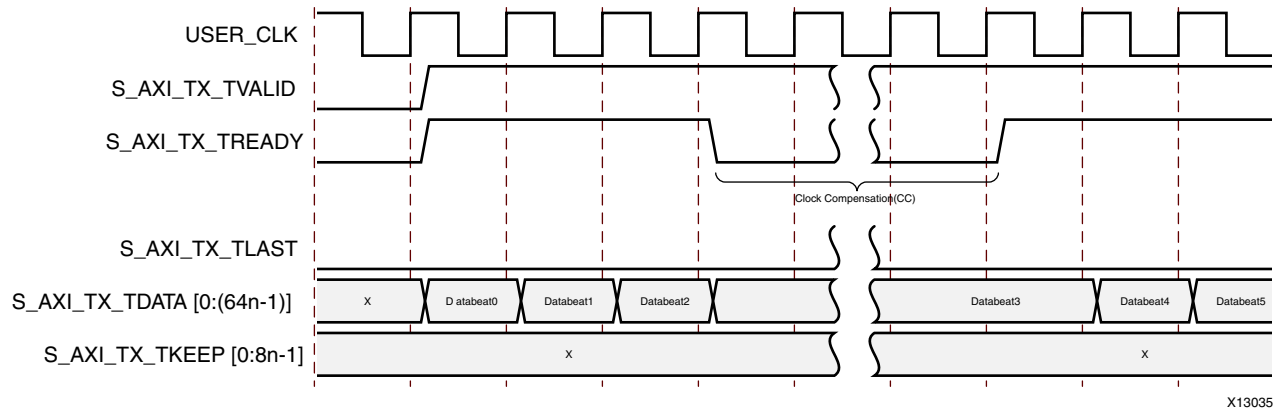


Figure 3-13: Data Transfer with Pause

Example C: Data Transfer with Clock Compensation

The Aurora 64B/66B core automatically interrupts data transmission when it sends clock compensation sequences. The clock compensation sequence imposes three cycles of PAUSE every 10,000 cycles.

Figure 3-14 shows how the Aurora 64B/66B core pauses data transmission during the clock compensation sequence.



X13035

Notes:

1. When clock compensation is used, uninterrupted data transmission is not possible. See [Clock Compensation, page 33](#) for more information about when clock compensation is required.

Figure 3-14: Data Transfer Paused by Clock Compensation

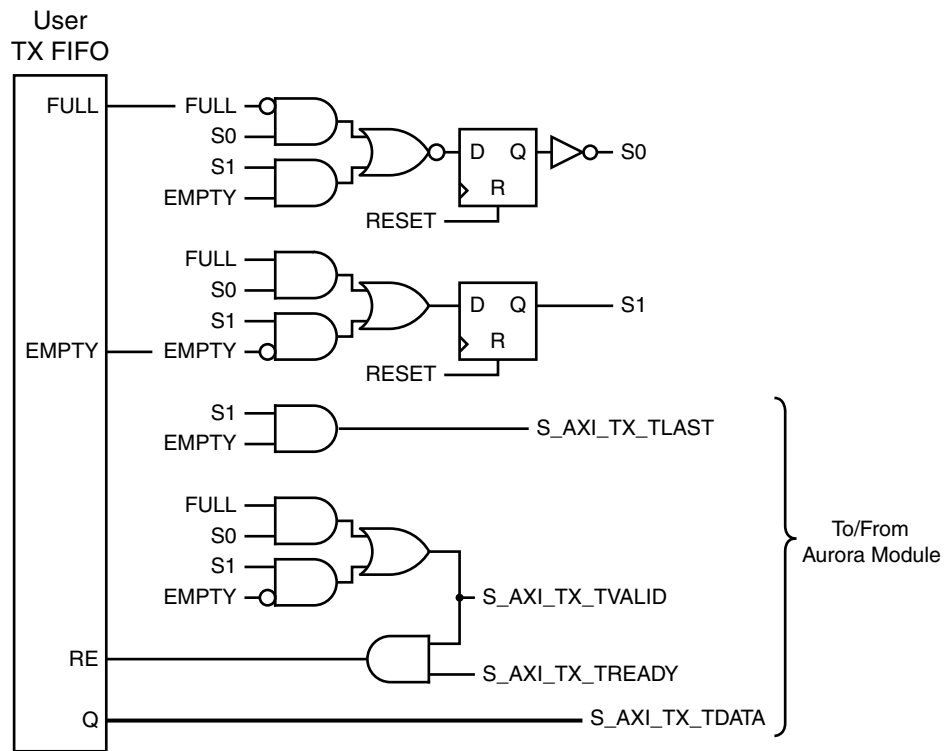
TX Interface Example

This section illustrates a simple example of an interface between a transmit FIFO and the AXI4-Stream interface of an Aurora 64B/66B core.

To review, in order to transmit data, the user application asserts `S_AXI_TX_TVALID`, `S_AXI_TX_TREADY` indicates that the data on the `S_AXI_TX_TDATA` bus is transmitted on the next rising edge of the clock, assuming `S_AXI_TX_TVALID` remains asserted.

Figure 3-15 is a diagram of a typical connection between an Aurora 64B/66B core and the data source (in this example, a FIFO), including the simple logic needed to generate, `S_AXI_TX_TVALID` and `S_AXI_TX_TLAST` from typical FIFO buffer status signals. While `RESET` is `FALSE`, the example application waits for a FIFO to fill, then generates the `S_AXI_TX_TVALID` signal. These signals cause the Aurora 64B/66B core to start reading the FIFO by asserting the `S_AXI_TX_TREADY` signal.

The Aurora 64B/66B core encapsulates the FIFO data and transmits it until the FIFO is empty. Now the example application tells the Aurora 64B/66B core to end the transmission using the `S_AXI_TX_TLAST` signal.



X13053

Figure 3-15: Transmitting Data

Receiving Data

When the Aurora 64B/66B core receives an Aurora 64B/66B frame, it presents it to the user application through the RX AXI4-Stream interface after discarding the control information, idle blocks, and clock compensation blocks.

The Aurora 64B/66B core has no built-in buffer for user data. As a result, there is no `M_AXI_RX_TREADY` signal on the RX AXI4-Stream interface. The only way for the user application to control the flow of data from an Aurora channel is to use one of the core optional flow control features. In most cases, a FIFO should be added to the RX datapath to ensure no data is lost while flow control messages are in transit.

The Aurora 64B/66B core asserts the `M_AXI_RX_TVALID` signal when the signals on its RX AXI4-Stream interface are valid. Applications should ignore any values on the RX AXI4-Stream ports sampled while `M_AXI_RX_TVALID` is deasserted (active-Low).

`M_AXI_RX_TVALID` is asserted concurrently with the first word of each frame from the Aurora 64B/66B core. `M_AXI_RX_TLAST` is asserted concurrently with the last word or partial word of each frame. The `M_AXI_RX_TKEEP` port indicates the number of valid bytes in the final word of each frame. It uses the same byte indication procedure as `S_AXI_TX_TKEEP` and is only valid when `M_AXI_RX_TLAST` is asserted.

If the CRC option is selected, the received data stream is computed for the expected CRC value. This block re-calculates the `M_AXI_RX_TKEEP` value and asserts `M_AXI_RX_TLAST` correspondingly.

The Aurora 64B/66B core can deassert `M_AXI_RX_TVALID` anytime, even during a frame.

[Example A: Data Reception with Pause](#) shows the reception of a typical Aurora 64B/66B frame.

Example A: Data Reception with Pause

[Figure 3-16](#) shows an example of $3n$ bytes of received data interrupted by a pause. Data is presented on the `M_AXI_RX_TDATA` bus. When the first n bytes are placed on the bus, the `M_AXI_RX_TVALID` output is asserted to indicate that data is ready for the user application. On the clock cycle following the first data beat, the core deasserts `M_AXI_RX_TVALID`, indicating to the user application that there is a pause in the data flow.

After the pause, the core asserts `M_AXI_RX_TVALID` and continues to assemble the remaining data on the `M_AXI_RX_TDATA` bus. At the end of the frame, the core asserts `M_AXI_RX_TLAST`. The core also computes the value of `M_AXI_RX_TKEEP` bus and presents it to the user application based on the total number of valid bytes in the final word of the frame.

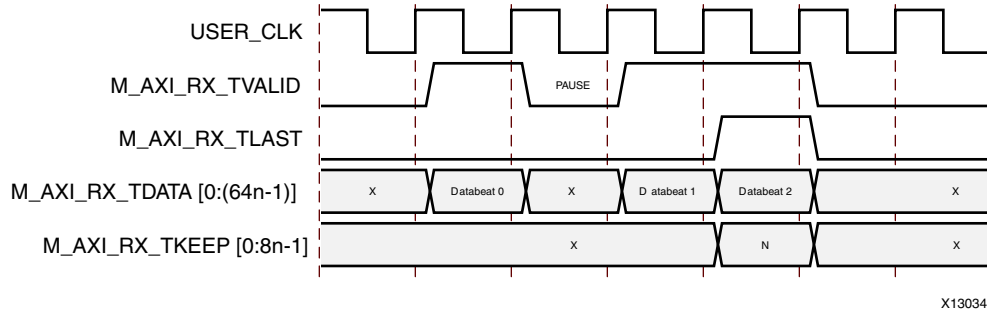


Figure 3-16: Data Reception with Pause

RX Interface Example

The RX AXI4-Stream interface of an Aurora 64B/66B core can be implemented with a simple FIFO. To receive data, the FIFO monitors the M_AXI_RX_TVALID signal. When valid data is present on the M_AXI_RX_TDATA port, M_AXI_RX_TVALID is asserted. Because M_AXI_RX_TVALID is connected to the FIFO WE port, the data and framing signals and KEEP value are written to the FIFO.

Framing Efficiency

There are two factors that affect framing efficiency in the Aurora 64B/66B core:

- Size of the frame
- Data invalid request from gear box that occurs after every 32 USER_CLK cycles

The clock compensation (CC) sequence, which uses three USER_CLK cycles on every lane every 10,000 USER_CLK cycles, consumes about 0.03% of the total channel bandwidth.

The gear box in GTX/GTH transceivers requires periodic pause to account for the clock divider ratio and 64B/66B encoding. This appears as a back pressure in the AXI4-Stream interface and user data needs to be stopped for 1 cycle after every 32 cycles (Figure 3-17).

The User Interface has the S_AXI_TX_TREADY signal from the Aurora core being deasserted (active-Low) for 1 cycle once every 32 cycles. The pause cycle is used to compensate the Gearbox for the 64B/66B encoding.

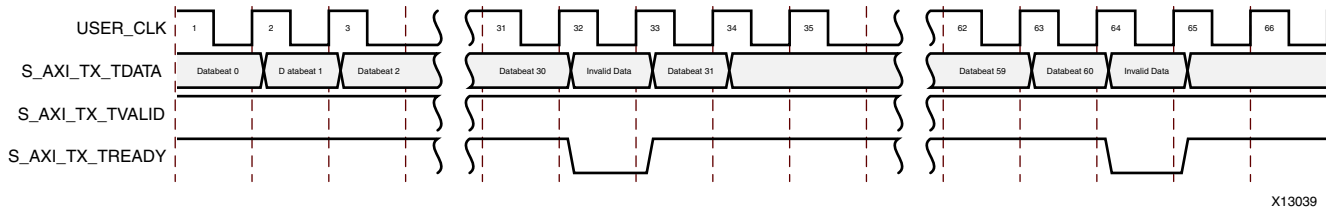


Figure 3-17: Framing Efficiency

For more information on gear box pause in GTX/GTH transceivers, see the *Virtex-6 FPGA GTX Transceivers User Guide (UG366)*, the *Virtex-6 FPGA GTH Transceivers User Guide (UG371)*, and the *7 Series FPGAs GTX/GTH Transceivers User Guide (UG476)*.

The Aurora 64B/66B core implements the Strict Aligned option of the Aurora 64B/66B protocol. No data blocks are placed after Idle blocks or SEP blocks on a given cycle. The restriction of not placing data blocks after SEP blocks reduces framing efficiency in a multilane Aurora 64B/66B core.

Example

Table 3-3 is an example calculated after including overhead for clock compensation. It shows the efficiency for a single-lane channel and illustrates that the efficiency increases as frame length increases.

Table 3-3: Efficiency Example

User Data Bytes	Framing Efficiency %
100	96.12
1,000	99.18
10,000	99.89

Table 3-4 shows the overhead in single-lane channel when transmitting 256 bytes of frame data. The resulting data unit is 264 bytes long due to the SEP block used to end the frame. This results in 3.03% overhead in the transmitter. In addition, clock compensation blocks must be transmitted for three cycles every 10,000 cycles, resulting in an additional 0.03% overhead in the transmitter.

Table 3-4: Typical Overhead for Transmitting 256 Data Bytes

Lane	Clock	Function
[D0:D7]	1	Channel frame data
[D8:D15]	2	Channel frame data
.		
.		
.		
[D248:D255]	32	Channel frame data
Control block	33	SEP0 block

Streaming Interface

Figure 3-18 shows an example of an Aurora 64B/66B core configured with a streaming user interface.

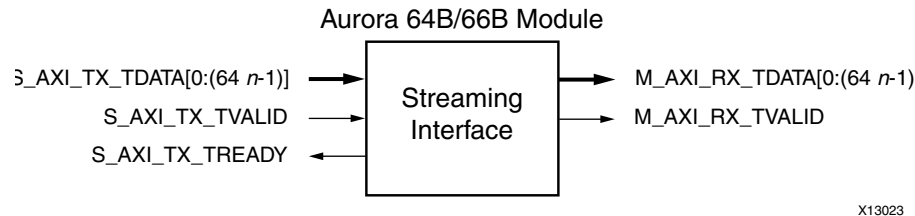


Figure 3-18: Aurora 64B/66B Core Streaming User Interface

Transmitting and Receiving Data

The streaming interface allows the Aurora channel to be used as a pipe. Words written into the TX side of the channel are delivered, in order after some latency, to the RX side. After initialization, the channel is always available for writing, except when the `DO_CC` signal is asserted to send clock compensation sequences. Applications transmit data through the `S_AXI_TX_TDATA` port, and use the `S_AXI_TX_TVALID` port to indicate when the data is valid (asserted active-High). The streaming Aurora interface expects data to be filled for the entire `S_AXI_TX_TDATA` port width (integral multiple of eight bytes). The Aurora 64B/66B core deasserts `S_AXI_TX_TREADY` (active-Low) when the channel is not ready to receive data. Otherwise, `S_AXI_TX_TREADY` remains asserted.

When `S_AXI_TX_TVALID` is deasserted, gaps are created between words. These gaps are preserved, except when clock compensation sequences are being transmitted. Clock compensation sequences are replicated or deleted by the CC logic to make up for frequency differences between the two sides of the Aurora channel. As a result, gaps created when `DO_CC` is asserted can shrink and grow. For details on the `DO_CC` signal, see [Clock Compensation, page 33](#).

When data arrives at the RX side of the Aurora channel it is presented on the `M_AXI_RX_TDATA` bus and `M_AXI_RX_TVALID` is asserted. The data must be read immediately or it will be lost. If this is unacceptable, a buffer must be connected to the RX interface to hold the data until it can be used.

Figure 3-19 shows a typical example of a streaming data transfer. The example begins with neither of the ready signals asserted, indicating that both the user logic and the Aurora 64B/66B core are not ready to transfer data. During the next clock cycle, the Aurora 64B/66B core indicates that it is ready to transfer data by asserting `S_AXI_TX_TREADY`. One cycle later, the user logic indicates that it is ready to transfer data by asserting the `S_AXI_TX_TVALID` signal and placing data on the `S_AXI_TX_TDATA` bus. Because both ready signals are now asserted, data D0 is transferred from the user logic to the Aurora 64B/66B core. Data D1 is transferred on the following clock cycle.

In this example, the Aurora 64B/66B core deasserts its ready signal, `S_AXI_TX_TREADY`, and no data is transferred until the next clock cycle when, once again, the `S_AXI_TX_TREADY` signal is asserted. Then the user application deasserts `S_AXI_TX_TVALID` on the next clock cycle, and no data is transferred until both ready signals are asserted.

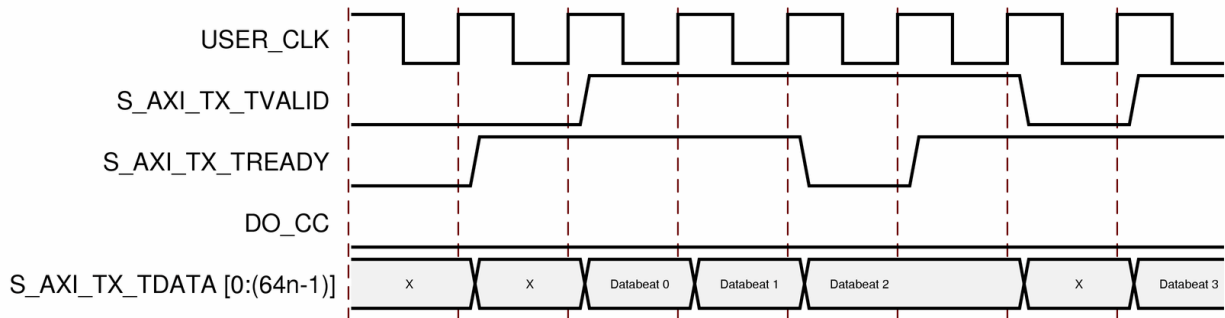


Figure 3-19: Typical Streaming Data Transfer

Figure 3-20 shows a typical example of streaming data reception.

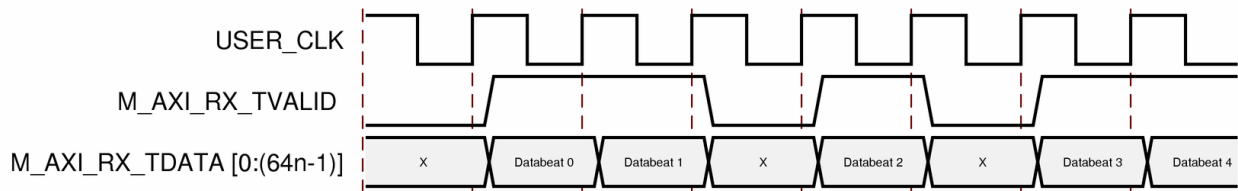


Figure 3-20: Typical Streaming Data Reception

Flow Control

This section explains how to use Aurora flow control. Two optional flow control interfaces are available. *Native flow control* (NFC) is used for regulating the data transmission rate at the receiving end of a full-duplex channel. *User flow control* (UFC) is used to accommodate high-priority messages for control operations.

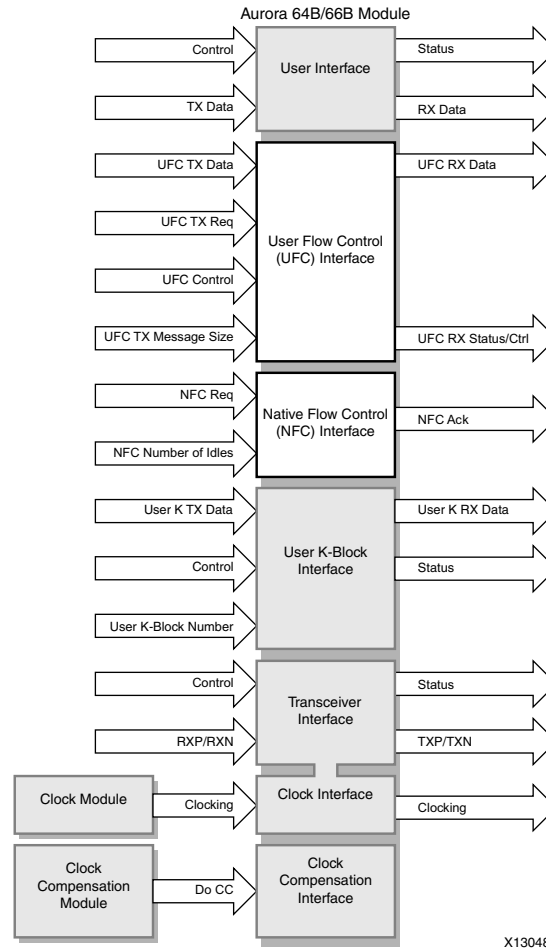


Figure 3-21: Top-Level Flow Control

Native Flow Control

The Aurora 64B/66B protocol includes native flow control (NFC) to allow receivers to control the rate at which data is sent to them by specifying a number of cycles that the channel partner cannot send data. The data flow can even be turned off completely by requesting that the transmitter temporarily send only idles (XOFF). NFC is typically used to prevent FIFO overflow conditions. For detailed explanation of NFC operation, see the *Aurora 64B/66B Protocol Specification* ([SP011](#)).

To send an NFC message to a channel partner, the user application asserts `S_AXI_NFC_TX_TVALID` and writes an 8-bit Pause count to `S_AXI_NFC_TX_TDATA[0:7]`. The Pause code indicates the minimum number of cycles the channel partner must wait after receiving an NFC message before it can resume sending data. The user application must hold `S_AXI_NFC_TX_TVALID`, `S_AXI_NFC_TX_TDATA[0:7]`, and `S_AXI_NFC_TX_TDATA[8]` (`NFC_XOFF`) (if used) until `S_AXI_NFC_TX_TREADY` is asserted on a positive `USER_CLK` edge, indicating the Aurora 64B/66B core will transmit the NFC message. Aurora 64B/66B cores cannot transmit data while sending NFC messages. `S_AXI_TX_TREADY` is always deasserted on the cycle following an `S_AXI_NFC_TX_TREADY` assertion. NFC Completion mode is available only for the framing Aurora 64B/66B interface.

Example A: Transmitting an NFC Message

Figure 3-22 shows an example of the transmit timing when the user application sends an NFC message to a channel partner using a AXI4-Stream interface.

Note: Signal `S_AXI_TX_TREADY` is deasserted for one cycle to create the gap in the data flow in which the NFC message is placed.

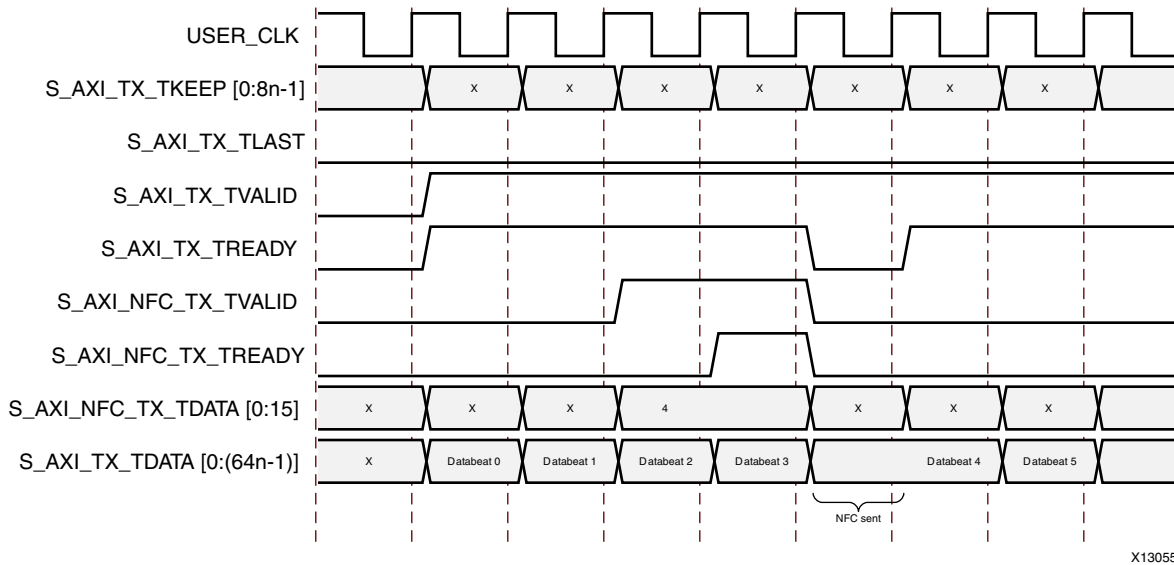


Figure 3-22: Transmitting an NFC Message

Example B: Receiving a Message with NFC Idles Inserted

Figure 3-23 shows an example of the signals on the TX user interface when an NFC message is received. In this case, the NFC message sends the number 8'b01, requesting two cycles without data transmission. The core deasserts S_AXI_TX_TREADY on the user interface to prevent data transmission for two cycles. In this example, the core is operating in Immediate NFC mode. Aurora 64B/66B cores can also operate in completion mode, where NFC Idles are only inserted before the first data bytes of a new frame. If a completion mode core receives an NFC message while it is transmitting a frame, it finishes transmitting the frame before deasserting S_AXI_TX_TREADY to insert idles.

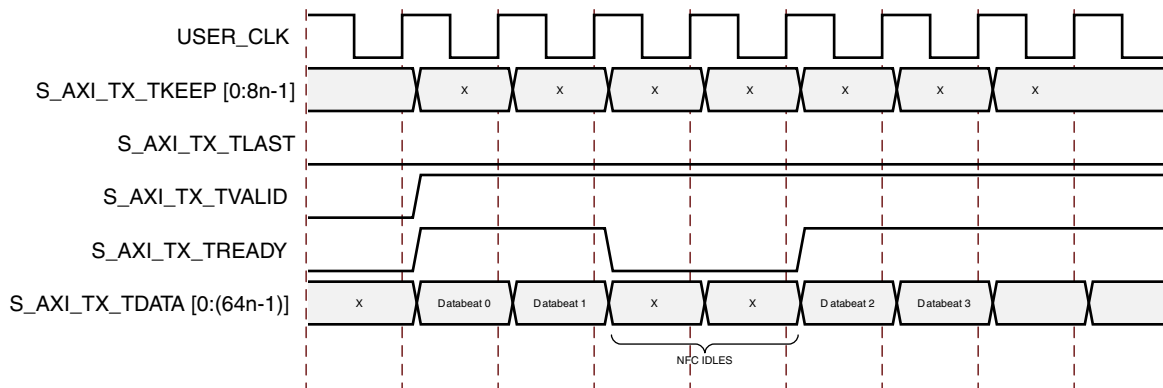


Figure 3-23: Transmitting a Message with NFC Idles Inserted

User Flow Control

The Aurora 64B/66B protocol includes user flow control (UFC) to allow channel partners to send control information using a separate in-band channel. The user application can send short UFC messages to the core's channel partner without waiting for the end of a frame in progress. The UFC message shares the channel with regular frame data, but has a higher priority than frame data. UFC messages are interruptible by high-priority control blocks such as CC/NR/CB/NFC blocks.

Transmitting UFC Messages

UFC messages can carry from 1 to 256 data bytes. The user application specifies the length of the message by driving the number of bytes required minus one on the UFC_TX_MS port. For example, a value of 3 will transmit 4 bytes of data; and a value of 0 will transfer 1 byte.

To send a UFC message, the user application asserts UFC_TX_REQ while driving the UFC_TX_MS port with the desired SIZE code for a single cycle. After a request, a new request cannot be made until S_AXI_UFC_TX_TREADY is asserted for the final cycle of the previous request. The data for the UFC message must be placed on the S_AXI_UFC_TX_TDATA port and the S_AXI_UFC_TX_TVALID signal must be asserted whenever the bus contains valid message data.

The core deasserts `S_AXI_TX_TREADY` while sending UFC data, and keeps `S_AXI_UFC_TX_TREADY` asserted until it has enough data to complete the message that was requested. If `S_AXI_UFC_TX_TVALID` is deasserted during a UFC message, Idles are sent in the channel, `S_AXI_TX_TREADY` remains deasserted, and `S_AXI_UFC_TX_TREADY` remains asserted. If a CC request, CB request, or NFC request is made to the core, `S_AXI_UFC_TX_TREADY` is deasserted while the requested operation is performed, because CC, CB, and NFC have higher priority.

Example A: Transmitting a Single-Cycle UFC Message

The procedure for transmitting a single cycle UFC message is shown in Figure 3-24. In this case a 4-byte message is being sent on an 8-byte interface.

Note: Signals `S_AXI_TX_TREADY` and `S_AXI_UFC_TX_TREADY` are deasserted for a cycle before the core accepts message data: this cycle is used to send the UFC header.

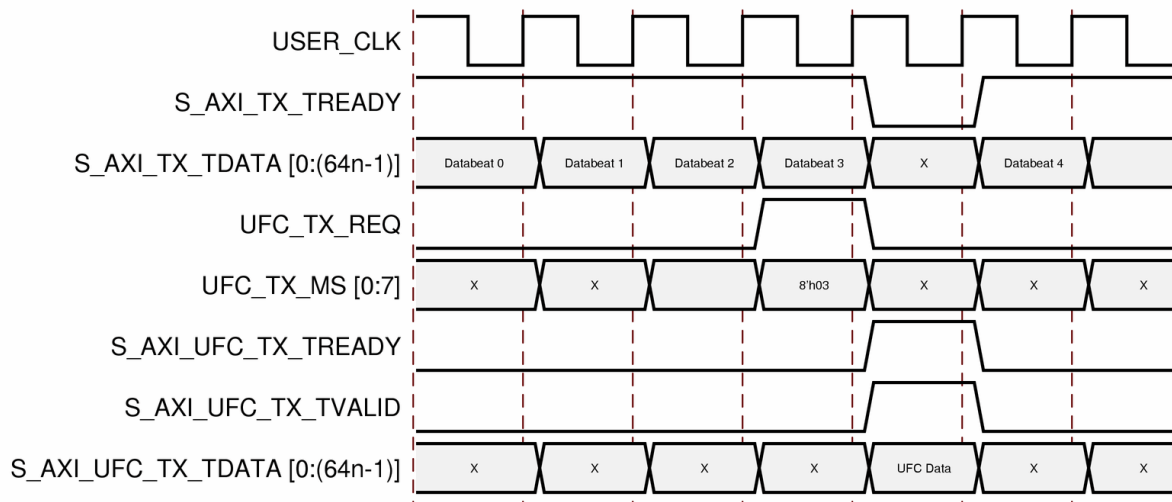


Figure 3-24: Transmitting a Single-Cycle UFC Message

Example B: Transmitting a Multicycle UFC Message

The procedure for transmitting a two-cycle UFC message is shown in Figure 3-25. In this case the user application is sending a 16-byte message using an 8-byte interface.

`S_AXI_UFC_TX_TREADY` is asserted for three cycles; one cycle for the UFC header, and two cycles for UFC data.

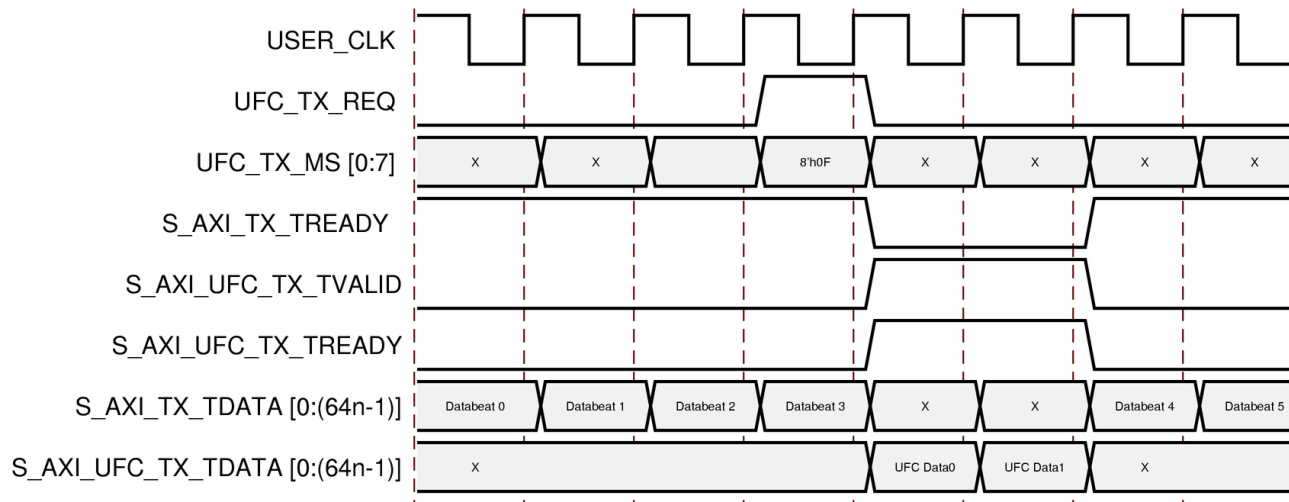


Figure 3-25: Transmitting a Multi-Cycle UFC Message

Receiving User Flow Control Messages

When the Aurora 64B/66B core receives a UFC message, it passes the data from the message to the user application through a dedicated UFC AXI4-Stream interface. The data is presented on the `M_AXI_UFC_RX_TDATA` port; assertion of `M_AXI_UFC_RX_TVALID` indicates the start of the message data and `M_AXI_UFC_RX_TLAST` indicates the end. `M_AXI_UFC_RX_TKEEP` is used to show the number of valid bytes on `M_AXI_UFC_RX_TDATA` during the last cycle of the message (for example, while `M_AXI_UFC_RX_TLAST` is asserted). Signals on the `UFC_RX` AXI4-Stream interface are only valid when `M_AXI_UFC_RX_TVALID` is asserted.

Example C: Receiving a Single-Cycle UFC Message

Figure 3-26 shows an Aurora 64B/66B core with an 8-byte data interface receiving a 4-byte UFC message. The core presents this data to the user application by asserting `M_AXI_UFC_RX_TVALID` and `M_AXI_UFC_RX_TLAST` to indicate a single cycle frame. The `M_AXI_UFC_RX_TKEEP` bus is set to 4, indicating only the four most significant bytes of the interface are valid.

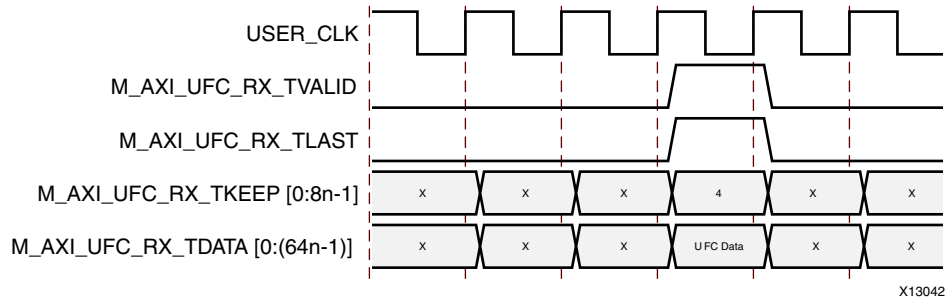


Figure 3-26: Receiving a Single-Cycle UFC Message

Example D: Receiving a Multicycle UFC Message

Figure 3-27 shows an Aurora 64B/66B core with an 8-byte interface receiving a 15-byte message.

Note: The resulting frame is two cycles long, with M_AXI_UFC_RX_TKEEP set to 7 on the second cycle indicating that all seven bytes of the data are valid.

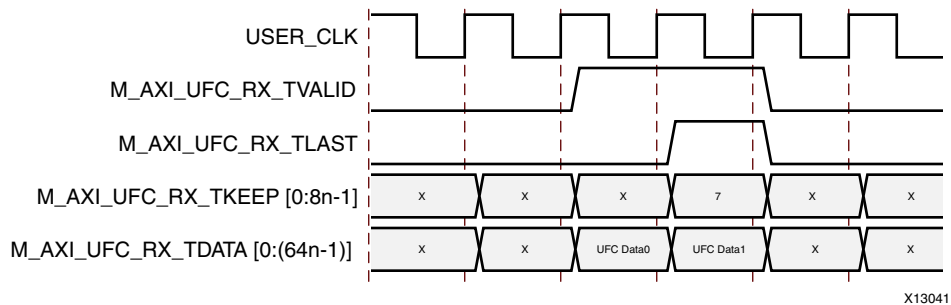
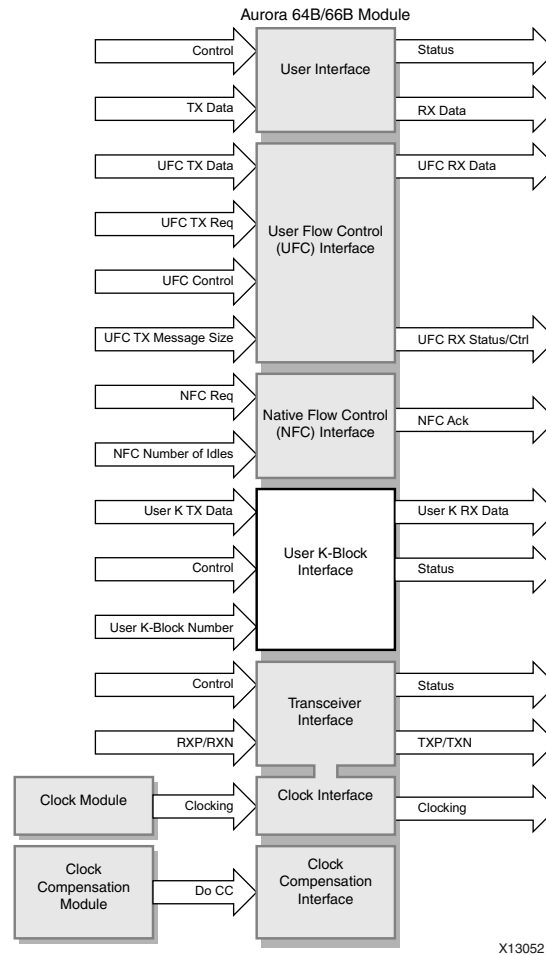


Figure 3-27: Receiving a Multi-Cycle UFC Message

User K-Block Interface

This section describes short single block data transmission and reception.



X13052

Figure 3-28: Top-Level User K-Block Interface

User K-blocks are special single block codes which include control blocks that are not decoded by the Aurora interface, but are instead passed directly to the user application. These blocks can be used to implement application-specific control functions. There are nine available User K-blocks (Table 3-5). Their priority is lower than UFC but higher than user data.

Table 3-5: Valid Block Type Field (BTF) Values for User K-Block

User K-Block Name	User K-Block BTF
User K-Block 0	0xD2
User K-Block 1	0x99
User K-Block 2	0x55
User K-Block 3	0xB4
User K-Block 4	0xCC
User K-Block 5	0x66
User K-Block 6	0x33
User K-Block 7	0x4B
User K-Block 8	0x87

The User K-block is not differentiated for streaming or framing designs. Each block code of User K is eight bytes wide and is encoded with a User K BTF, which is indicated by the user application in `S_AXI_USER_K_TX_TDATA[4 : 7]` as User K Block No. The User K-block is a single block code and is always delineated by User K Block No. User should provide the User K Block No as specified in [Table 2-12, page 23](#). It can have only seven bytes of `S_AXI_USER_K_TDATA`.

Transmitting User K-Blocks

`S_AXI_USER_K_TX_TREADY` is asserted by Aurora and is prioritized by CC, CB, NFC, and UFC. After placing `S_AXI_USER_K_TX_TDATA` and along with User K Block No and `S_AXI_USER_K_TX_TVALID` is asserted, the user application can change `S_AXI_USER_K_TX_TDATA` if required when `S_AXI_USER_K_TX_TREADY` is asserted ([Figure 3-29](#)). This enables the Aurora core to select appropriate User K BTF among the nine User K-blocks. The data available during assertion of `S_AXI_USER_K_TX_TREADY` is always serviced.

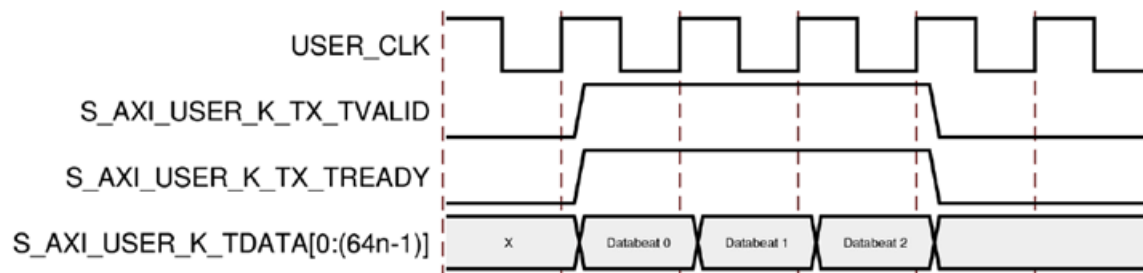


Figure 3-29: Transmitting User K Data and User K-Block Number

Receiving User K-Blocks

The receive BTF is decoded and the block number for the corresponding BTF is passed on to the user application as such (Figure 3-30). The user application can validate the M_AXI_USER_K_RX_TDATA available on the bus when M_AXI_USER_K_RX_TVALID is asserted.

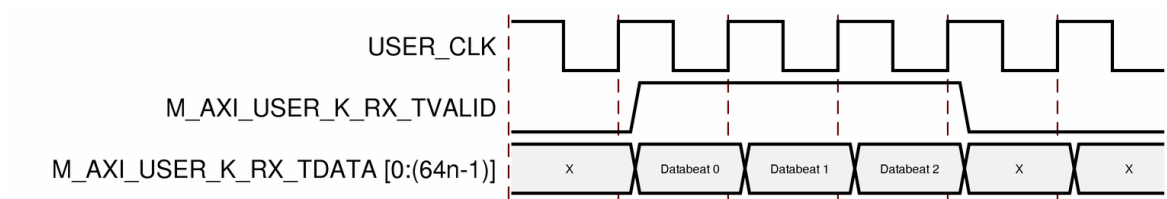


Figure 3-30: Receiving User K Data and User K-Block Number

Status, Control, and the Transceiver Interface

The status and control ports of the Aurora 64B/66B core allow user applications to monitor the Aurora channel and use built-in features of the GT transceiver interface. This section provides diagrams and port descriptions for the Aurora 64B/66B core's status and control interface, along with the GTX/GTH serial I/O interface.

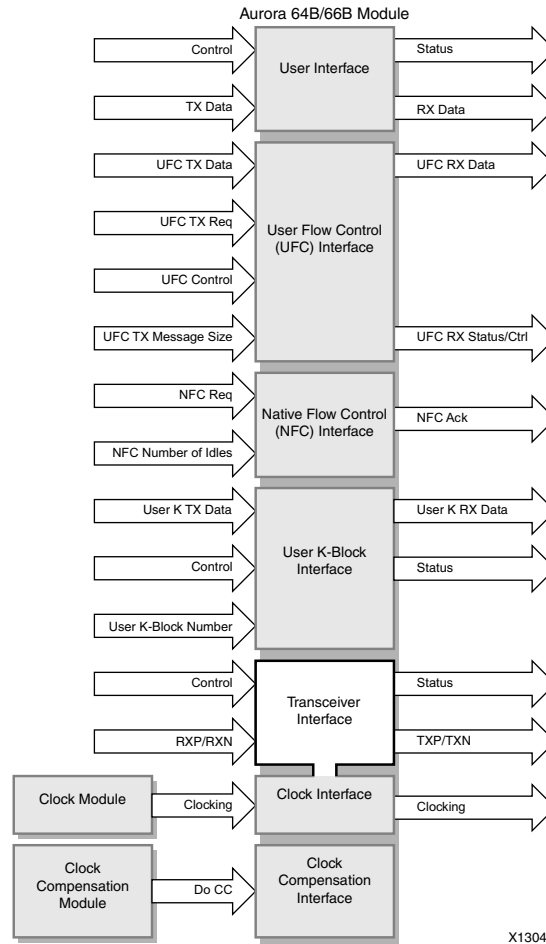


Figure 3-31: Top-Level GTX Interface

Status and Control Ports

Aurora 64B/66B cores are full-duplex/simplex, and provide a TX and an RX Aurora channel connection. The Aurora 64B/66B core does not require any sideband signals for simplex mode of operation. Figure 3-32 shows the status and control interface for an Aurora 64B/66B core.

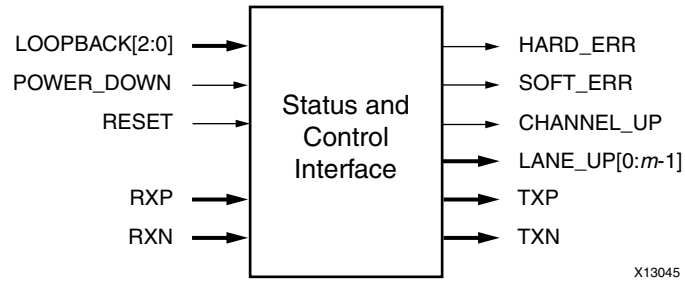


Figure 3-32: Status and Control Interface for the Aurora 64B/66B Core

Error Signals in Aurora 64B/66B Cores

Equipment problems and channel noise can cause errors during Aurora channel operation. The 64B/66B encoding allows the Aurora 64B/66B core to detect some bit errors that occur in the channel. The core reports these errors by asserting the SOFT_ERR signal on every cycle they are detected.

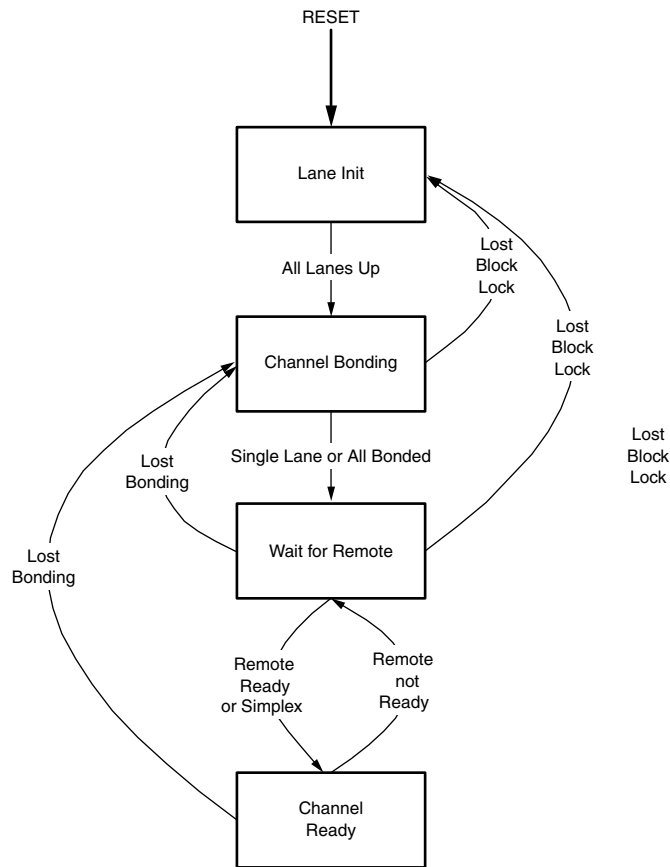
The core also monitors each high-speed serial GTX/GTH transceiver for hardware errors such as buffer overflow and loss of lock. The core reports hardware errors by asserting the HARD_ERR signal. Catastrophic hardware errors can also manifest themselves as burst of soft errors. The core uses the Block Sync algorithm described in the *Aurora 64B/66B Protocol Specification* ([SP011](#)) to determine whether to treat a burst of soft errors as a hard error.

Whenever a hard error is detected, the Aurora 64B/66B core automatically resets itself and attempts to re-initialize. In most cases, this allows the Aurora channel to be reestablished as soon as the hardware issue that caused the hard error is resolved. Soft errors do not lead to a reset unless enough of them occur in a short period of time to trigger the block sync state machine.

See [Table 2-16, page 25](#) for descriptions of the error signals.

Initialization

Aurora 64B/66B cores initialize automatically after power up, reset, or hard error. Aurora 64B/66B core modules on each side of the channel perform the Aurora initialization procedure until the channel is ready for use. The LANE_UP bus indicates which lanes in the channel have finished the lane initialization portion of the initialization procedure. This signal can be used to help debug equipment problems in a multi-lane channel. CHANNEL_UP is asserted only after the core completes the entire initialization procedure.



X13040

Figure 3-33: Initialization Overview

Aurora 64B/66B cores can receive data before CHANNEL_UP is asserted. Only the M_AXI_RX_TVALID signal on the user interface should be used to qualify incoming data. CHANNEL_UP can be inverted and used to reset modules that drive the TX side of a full-duplex channel, because no transmission can occur until after CHANNEL_UP. If user application modules need to be reset before data reception, one of the LANE_UP signals can be inverted and used. Data cannot be received until after all the LANE_UP signals are asserted.

Aurora Simplex Operation

Simplex Aurora 64B/66B cores do not have any sideband connection and use timers to declare that the partner is out of initialization and is ready for data transfer. These simplex cores transmit periodic channel bonding characters to ensure the channel partner is bonded. If at any time during the data transfer, the link is broken or re-initialized, the channel will auto recover after the periodic channel bond character is sent to the partner and does not require any reset.

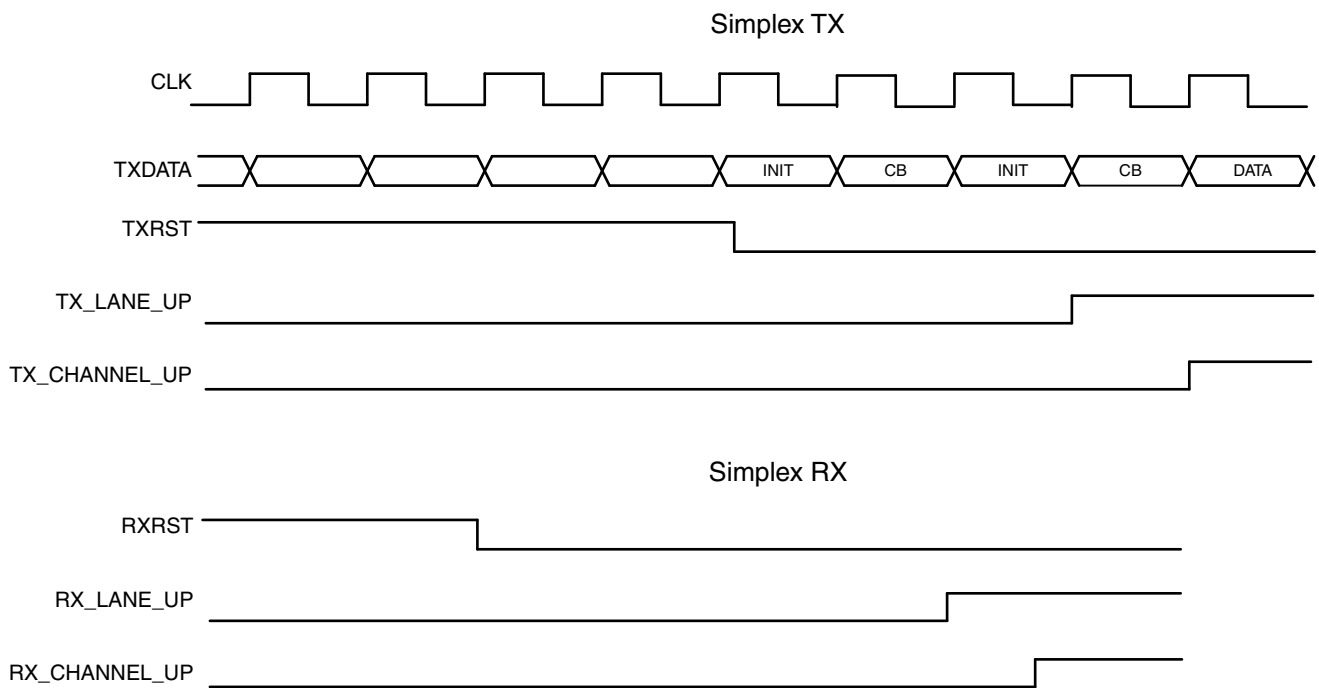
After the link is broken, the Aurora 64B/66B initialization state machine (Figure 3-33) moves from the Channel Ready state to the Wait for Remote state. It waits in this state until the periodic Channel Bond character is received by the partner and then moves to the Channel Ready state. The data transferred during the re-initialization process is lost.

The user application can modify the timer value based on their channel requirement. For simulation, the timer value (-GSIMPLEX_TIMER) can be modified on the simulate_mti.do to test for different timer values. For implementation, the SIMPLEX_TIMER_VALUE parameter must be modified on <component name>.v[hd].

Simplex Reset Sequencing

Figure 3-34 shows the reset timing sequence for Simplex TX and Simplex RX. In this sequence:

1. Simplex TX should deassert the RESET port after Simplex RX to ensure that Simplex RX is initialized and ready to receive from the transmitter.
2. Simplex RX asserts RX_LANE_UP and RX_CHANNEL_UP based on the initialization sequence received.
3. Simplex TX asserts TX_LANE_UP and TX_CHANNEL_UP based on the SIMPLEX_TIMER_VALUE set.



UG775_c6_04_042111

Figure 3-34: Simplex Reset Timing Sequence

DRP Interface

The DRP interface controls or monitors the status of the transceiver block. The user application can access or update the serial transceiver settings by writing/reading the values through the DRP ports. The Native interface provides the native transceiver DRP interface. The AXI4-Lite interface can also be selected to access the DRP ports through it.

Core Features

This section describes the following features of the Aurora 64B/66B core.

- [CRC](#)
- [Using ChipScope Pro Cores](#)
- [Hot-Plug Logic](#)

CRC

A 32-bit CRC, implemented for framing user data interface, is available in the `<component name>_crc_top.v[hd]` module. The `CRC_VALID` and `CRC_PASS_FAIL_N` signals indicate the result of a received CRC with a transmitted CRC (see [Table 3-6](#)).

Table 3-6: CRC Module Ports

Port Name	Direction	Description
CRC_VALID	Output	Active-High signal that samples the CRC_PASS_FAIL_N signal.
CRC_PASS_FAIL_N	Output	CRC_PASS_FAIL_N is asserted High when the received CRC matches the transmitted CRC. This signal is not asserted if the received CRC is not equal to the transmitted CRC. The CRC_PASS_FAIL_N signal should always be sampled with the CRC_VALID signal.

Using ChipScope Pro Cores

The ChipScope™ Pro ICON, ILA, and VIO cores aid in debugging and validating the design in board and are provided with the Aurora 64B/66B core. Select the **CHIPSCOPE** option from the core GUI (see [Figure 9-2, page 122](#)) to include it as a part of the example design.

Hot-Plug Logic

Hot-plug logic in Aurora 64B/66B designs with Virtex-7, Kintex-7, and Virtex-6 FPGAs is based on the received clock compensation characters. Reception of clock compensation characters at RX interface of Aurora infers communication channel is active and not broken. If clock compensation characters are not received in a predetermined time, the hot-plug logic resets the core and the transceiver. The clock compensation module must be used for Aurora 64B/66B designs with Virtex-7, Kintex-7, and Virtex-6 FPGAs.

SECTION II: VIVADO DESIGN SUITE

Customizing and Generating the Core

Constraining the Core

Detailed Example Design

Customizing and Generating the Core

This chapter includes information on using Vivado™ Design Suite tools to customize and generate the LogiCORE™ IP Aurora 64B/66B core.

GUI

The Aurora 64B/66B core can be customized to suit a wide variety of requirements using the Vivado IP catalog. This chapter details the available customization parameters and how these parameters are specified within the IP catalog interface.

Using the IP Catalog

The Aurora 64B/66B IP catalog is presented when you select the Aurora 64B/66B core in the Vivado IP catalog. For help starting and using the IP catalog, see the Vivado design tools documentation [Ref 3]. [Figure 4-1, page 68](#) and [Figure 4-2, page 69](#) show features that are described in corresponding sections.

IP Catalog

[Figure 4-1](#) and [Figure 4-2](#) show the catalog. The left side displays a representative block diagram of the Aurora 64B/66B core as currently configured. The right side consists of user-configurable parameters. Details on the customizing options are provided in the following subsections, starting with [1 Component Name, page 69](#).

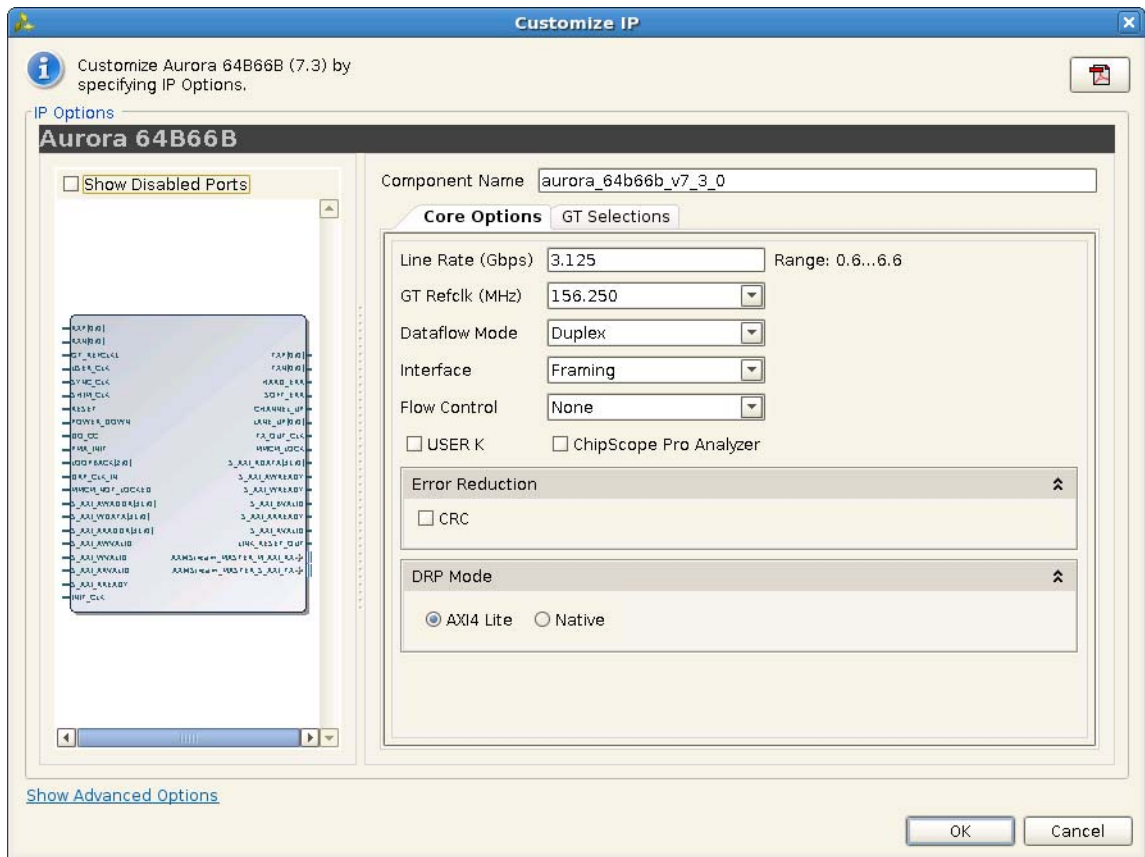


Figure 4-1: Aurora 64B/66B IP Catalog Page 1

3 GT Reference Clock Frequency

Select a reference clock frequency from the drop-down list. Reference clock frequencies are given in megahertz, and depend on the line rate selected. For best results, select the highest rate that can be practically applied to the reference clock input of the target device.

Default: 156.25 MHz

4 Data Flow Mode

Select the options for the direction of the channel that the Aurora 64B/66B core supports. Simplex Aurora 64B/66B cores have a single, unidirectional serial port that connects to a complementary simplex Aurora 64B/66B core. Two options are provided as RX-only simplex or TX-only simplex. These options select the direction of the channel that the Aurora 64B/66B core supports.

Duplex - Aurora 64B/66B cores have both TX and the corresponding RX on the other side for communication.

Default: Duplex

5 Interface

Select the type of datapath interface used for the core. Select **Framing** to use a complete AXI4-Stream interface that allows encapsulation of data frames of any length. Select **Streaming** to use a simple word-based interface with a data valid signal to stream data through the Aurora channel.

Default: Framing

6 Flow Control

Select the required option to add flow control to the core. *User* flow control (UFC) allows applications to send each other brief, high-priority messages through the Aurora channel. *Native* flow control (NFC) allows full-duplex receivers to regulate the rate of the data sent to them. Immediate mode allows idle codes to be inserted within data frames while completion mode only inserts idle codes between complete data frames.

Available options are:

- None
- UFC only
- Immediate Mode - NFC
- Completion Mode - NFC
- UFC + Immediate Mode - NFC
- UFC + Completion Mode - NFC

For the streaming interface, only immediate mode is available. For the framing interface, both immediate and completion modes are available.

Default: None

7 User K

Select to add User K interface to the core. User K-blocks are special single-block codes passed directly to the user application. These blocks are used to implement application-specific control functions.

Default: Unchecked

8 CRC

Select the option to insert CRC32 in the data stream.

Default: Unchecked

9 DRP

Select the required interface to control or monitor the Transceiver interface using the Dynamic Reconfiguration Port (DRP).

Available options are:

- Native
- AXI4_Lite

Default: Native

10 Columns

Select appropriate column from the drop-down list. This option is applicable only for Virtex-7 and Kintex™-7 devices. For other devices, the drop-down box is not visible.

Default: left

11 Lanes

Select the number of lanes (GTX/GTH transceivers) to be used in the core. The valid range depends on the target device selected.

Default: 1

12 GT_TYPE

Select the type of serial transceiver from the drop-down list. This option is applicable only for Virtex-7 XT devices. For other devices, the drop-down box is not visible.

Available options are:

- GTX
- V7GTH

Default: gtx

13 Lane Assignment

See the diagram in the information area in [Figure 4-1, page 68](#). Each numbered row represents a GT tile and each active box represents an available GTX/GTH transceiver. For each Aurora lane in the core, starting with Lane 1, select a GTX/GTH transceiver and place the lane by selecting its number in the GTX/GTH placement box.

14 GT REFCLK1 and GT REFCLK2

Select reference clock sources for the GTX/GTH tiles from the drop-down list in this section.

Default: GT REFCLK Source 1: GTXQn/ GTHQn; GT REFCLK Source 2: None;

Note: n depends on the serial transceiver (GTX/GTH) position.

ChipScope Pro Analyzer

Select to add ChipScope™ Pro cores to the Aurora 64B/66B core. (See [Using ChipScope Pro Cores, page 64](#).) This option provides a debugging interface that shows the core status signals in the ChipScope Pro analyzer tool.

Default: Unchecked














OK

Click **OK** to generate the core. (See [Generating the Core, page 97](#).) The modules for the Aurora 64B/66B core are written to the Vivado IP catalog tool project directory using the same name as the top level of the core.

Output Generation

The customized Aurora 64B/66B core is delivered as a set of HDL source modules in the language selected in the IP catalog tool project with supporting script and documentation files. These files are arranged in a predetermined directory structure under the project directory name provided to the IP catalog when the project is created as shown in this section.

Directory and File Structure

-  **<project directory>**
Top-level project directory; name is user-defined.
-  **<project directory>/<component name>**
Core readme file
 -  **<component name>/doc**
Product documentation
 -  **<component name>/example_design**
Example design files
 -  **/example_design/cc_manager**
Verilog/VHDL design files for the clock management block
 -  **/example_design/clock_module**
Verilog/VHDL design files for the clocking blocks
 -  **/example_design/gt**
Verilog/VHDL design files for the GTX/GTH transceiver
 -  **/example_design/traffic_gen_and_check**
Verilog/VHDL design files for the frame generator and checker
 -  **<component name>/implement**
Implementation scripts and support files
 -  **/implement/results**
Implement script results
 -  **<component name>/simulation**
Simulation test bench and simulation script files
 -  **/simulation/functional**
Functional simulation files
 -  **<component name>/src**
Verilog/VHDL files for the core

Directory and File Contents

The Aurora 64B/66B core directories and their associated files are defined in the following sections.

<project directory>

The `project` directory contains the Vivado design tools project files.

Table 4-1: **project Directory**

Name	Description
<project directory>	
<component name>.v[hd]	Aurora 64B/66B core top-level file
<component name>.xdc	Aurora 64B/66B core design constraints (only for 7 series devices)

[Back to Top](#)

<project directory>/<component name>

The `component name` directory contains the core file.

Table 4-2: **component name Directory**

Name	Description
<project directory>/<component name>	
aurora_64b66b_v7_3_readme.txt	Readme file

[Back to Top](#)

<component name>/doc

The `doc` directory contains a Xilinx website redirect to the product documentation.

Table 4-3: **doc Directory**

Name	Description
<component name>/doc	
pg074-aurora-64b66b.pdf	<i>LogiCORE IP Aurora 64B/66B v7.3 Product Guide</i>
aurora_64b66b_v7_3_vinfo.html	Version information file

[Back to Top](#)

<component name>/example_design

The `example_design` directory contains the example design files provided with the core.

Table 4-4: `example_design` Directory

Name	Description
<component name>/example_design	
<component name>_exdes.v[hd]	Example design source file
k7_icon.ngc k7_ila.ngc k7_vio.ngc	NGC files for the debug cores compatible with the ChipScope Pro Analyzer tool
<component name>_exdes.xdc	Aurora 64B/66B example design constraints
<component name>_reset_logic.v[hd]	Aurora 64B/66B reset logic

[Back to Top](#)

/example_design/cc_manager

The `cc_manager` directory contains the clock compensation source file.

Table 4-5: `cc_manager` Directory

Name	Description
<component name>/example_design/cc_manager	
<component name>_standard_cc_module.v[hd]	Clock compensation module source file

[Back to Top](#)

/example_design/clock_module

The `clock_module` directory contains the clock module source file.

Table 4-6: `clock_module` Directory

Name	Description
<component name>/example_design/clock_module	
<component name>_clock_module.v[hd]	Clock module source file
<component name>_enable_generator.v[hd]	clock enable generator module

[Back to Top](#)

/example_design/gt

The `gt` directory contains the Verilog/VHDL wrapper files for the GTX/GTH transceiver.

Table 4-7: `gt` Directory

Name	Description
<component name>/example_design/gt	
<component name>_gt_wrapper.v[hd] <component name>_multi_wrapper.v[hd] <component name>_gtx.v[hd] ⁽¹⁾	Verilog/VHDL wrapper files for the GTX/GTH transceiver

1. For Virtex-7/Kintex-7 FPGA GTX/GTH transceivers.

[Back to Top](#)

/example_design/traffic_gen_and_check

The `traffic_gen_and_check` directory contains frame generator and frame checker modules for Aurora 64B/66B core.

Table 4-8: `traffic_gen_and_check` Directory

Name	Description
<component name>/example_design/traffic_gen_and_check	
<component name>_frame_check.v[hd] <component name>_frame_gen.v[hd]	Example design traffic generation and checker files

[Back to Top](#)

<component name>/implement

The `implement` directory contains scripts and support files for both Linux and Windows operating systems. These scripts automate the process of synthesizing and implementing the files needed for the example design.

Table 4-9: `implement` Directory

Name	Description
<component name>/implement	
synplify.prj implement_synplify.sh implement_synplify.bat	Synplify Pro script files for Aurora 64B/66B example design

[Back to Top](#)

/implement/results

The `results` directory is created by the `implement` script, after which the `implement` script results are placed in the `results` directory.

Table 4-10: **results Directory**

Name	Description
<component name>/implement/results	
Implement script result files	

[Back to Top](#)

<component name>/simulation

The `simulation` directory contains the test bench files for the example design.

Table 4-11: **simulation Directory**

Name	Description
<component name>/simulation	
<component name>_tb.v[hd]	Test bench file for simulating the example design

[Back to Top](#)

/simulation/functional

The `functional` directory contains functional simulation scripts provided with the core.

Table 4-12: **functional Directory**

Name	Description
<component name>/simulation/functional	
<code>simulate_mti.do</code>	ModelSim macro file that compiles the example design sources, the structural simulation model, and the demonstration test bench then runs the functional simulation to completion
<code>wave_mti.do</code>	ModelSim macro file that opens a Wave window
<code>simulate_mti.sh</code>	Linux shell script to invoke ModelSim and run example design
<code>simulate_mti.bat</code>	Windows batch file to invoke ModelSim and run example design
<code>simulate_isim.sh</code>	ISim macro file that compiles the example design sources and the structural simulation model. The demonstration test bench then runs the functional simulation to completion in the Linux operating system.
<code>simulate_isim.bat</code>	ISim macro file that compiles the example design sources and the structural simulation model. The demonstration test bench then runs the functional simulation to completion in the Windows operating system.
<code>wave_isim.tcl</code>	ISim macro file that opens a Wave window with top-level signals.

[Back to Top](#)

/simulation/timing

The `timing` directory contains timing simulation scripts provided with the core.

Table 4-13: timing Directory

Name	Description
<component name>/simulation/timing	
simulate_mti.do	ModelSim macro file that compiles the SDF files of the core and the demonstration test bench then runs the timing simulation to completion
wave_mti.do	ModelSim macro file that opens a Wave window

[Back to Top](#)

<component name>/src

The `src` directory contains the source files related to the Aurora example design.

Table 4-14: src Directory

Name	Description
<component name>/src	
<component name>_64B66B.v[hd] <component name>_64B66B_descrambler.v[hd] <component name>_64B66B_scrambler.v[hd] <component name>_aurora_lane.v[hd] <component name>_aurora_pkg.vhd (VHDL Only) <component name>_aurora_to_gtx.v[hd] <component name>_block_sync_sm.v[hd] <component name>_cbcc_gtx_6466.v[hd] <component name>_ch_bond_code_gen.v[hd] <component name>_channel_err_detect.v[hd] <component name>_channel_init_sm.v[hd] <component name>_err_detect.v[hd] <component name>_global_logic.v[hd] <component name>_gtx_to_aurora.v[hd] <component name>_lane_init_sm.v[hd] <component name>_rx_ll.v[hd] <component name>_rx_ll_datapath.v[hd] <component name>_sym_dec.v[hd] <component name>_sym_gen.v[hd] <component name>_tx_ll.v[hd] <component name>_tx_ll_control_sm.v[hd] <component name>_tx_ll_datapath.v[hd] <component name>_ll_to_axi.v[hd] <component name>_axi_to_ll.v[hd]	Aurora 64B/66B source files

[Back to Top](#)

Constraining the Core

This chapter is relevant to the Vivado™ Design Suite.

Device, Package, and Speed Grade Selections

Not Applicable

Clock Frequencies

Aurora 64B/66B example design clock constraints can be grouped into the following three categories:

- GT reference clock constraint

The Aurora 64B/66B core uses one minimum reference clock and two maximum reference clocks for the design. The number of GT reference clocks is derived based on transceiver selection (that is, lane assignment in the second page GUI). The GT REFCLK value selected in the first page of the GUI is used to constrain the GT reference clock. The **create_clock** XDC command is used to constrain GT reference clocks.

- CORECLK clock constraint

CORECLKs are the clock based on which the core functions. CORECLKS such as USER_CLK and SYNC_CLK are derived out of TXOUTCLK generated by the GT transceiver based on the applied reference clock and the divider settings of the GT transceiver. The Aurora 64B/66B core calculates the USER_CLK/SYNC_CLK frequency based on the line rate and GT interface width. RXRECCLK_32 and RXRECCLK_64 are the received recovered clock constraint derived out of RXRECCLK for capturing the receive data from GT transceiver. The **create_clock** XDC command is used to constrain all CORECLKs.

- System clock constraint



RECOMMENDED: *The Aurora 64B/66B example design uses a debounce circuit to sample PMA_INIT asynchronously clocked by the system clock. It is recommended to have the system clock frequency lower than the GT reference clock frequency. The `create_clock` XDC command is used to constrain the system clock.*

False Paths

The system clock and user clock are not related to one another. No phase relationship exists between those two clocks. Those two clocks domains need to set as false paths. The `set_false_path` XDC command is used to constrain the false paths.

Example Design

The generated example design is a 10.3125 Gb/s line rate and a 156.25 MHz reference clock. The XDC file generated for the XC7K325T-FFG900-2 device follows:

```
##### CLOCK CONSTRAINTS #####
# User Clock Constraint: the value is selected based on the line rate of the module
create_clock -name TS_user_clk_i -period 6.206 [get_pins clock_module_i/
user_clk_net_i/O]
# SYNC Clock Constraint
create_clock -name TS_sync_clk_i -period 3.103 [get_pins clock_module_i/
sync_clock_net_i/O]

# Reference clock constraint for GTX
create_clock -name GTXQ0_left_i -period 6.402 [get_pins IBUFDS_GTXE2_CLK1/O]
# 50MHz board System Clock Constraint
create_clock -name TS_INIT_CLK -period 20 [get_pins IBUFDS_INIT_CLK/O]

##### No cross clock domain analysis. Domains are not related #####
set_false_path -from TS_INIT_CLK -to TS_user_clk_i
set_false_path -from TS_rxrecclk_32 -to TS_user_clk_i
set_false_path -from TS_user_clk_i -to TS_rxrecclk_32
set_false_path -through [get_pins gt_wrapper_i/cbcc_gtx0_i/fdr_fifo*/Q]

##### GT CLOCK Locations #####
# Differential SMA Clock Connection
set_property LOC R8 [get_ports GTXQ0_P]
set_property LOC R7 [get_ports GTXQ0_N]

##### GT Locations #####
set_property LOC GTXE2_CHANNEL_X0Y0 [get_cells gt_wrapper_i/gt_multi_gt_i/GTX_INST/
gtxe2_i]
```

Clock Management

Not Applicable

Clock Placement

Not Applicable

Banking

Not Applicable

Transceiver Placement

The **set_property** XDC command is used to constrain the GT transceiver location. This is provided as a tool tip on the second page of the XGUI. Sample XDC is provided for reference.

I/O Standard and Placement

The positive differential clock input pin (ends with _P) and negative differential clock input pin (ends with _N) are used as the GT reference clock. The **set_property** XDC command is used to constrain the GT reference clock pins.

Detailed Example Design

This chapter is relevant to the Vivado™ Design Suite.

Directory and File Contents

See [Output Generation, page 73](#) for the directory structure and file contents of the example design.

Quick Start Example Design

The quick start instructions provide a step-by-step procedure for generating an Aurora 64B/66B core, implementing the core in hardware using the accompanying example design, and simulating the core with the provided demonstration test bench (`demo_tb`). For detailed information about the example design provided with the Aurora 64B/66B core, see [Detailed Example Design](#).

The quick start example design consists of these components:

- An instance of the Aurora 64B/66B core generated using the default parameters
 - Full-duplex with a single GTX transceiver
 - AXI4-Stream user interface
- A top-level example design (`<component name>_exdes`) with an XDC file for the KC724 board
- A demonstration test bench to simulate two instances of the example design

Detailed Example Design

Each Aurora 64B/66B core includes an example design (`<component name>_exdes`) that uses the core in a simple data transfer system. For more details about the `example_design` directory, see [Output Generation, page 73](#).

The example design consists of two main components:

- Frame generator ([FRAME_GEN, page 85](#)) connected to the TX interface
- Frame checked ([FRAME_CHECK, page 91](#)) connected to the RX user interface

[Figure 6-1](#) shows a block diagram of the example design for a full-duplex core. [Table 6-1, page 84](#) describes the ports of the example design.

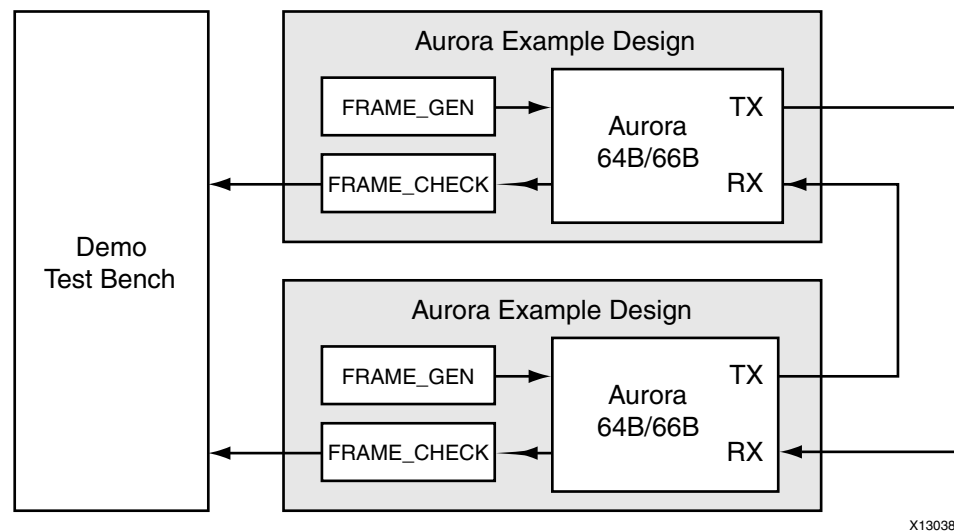


Figure 6-1: Example Design

The example design uses all the interfaces of the core. There are separate AXI4-Stream interfaces for optional flow control. Simplex cores without a TX or RX interface have no `FRAME_GEN` or `FRAME_CHECK` block, respectively. The frame generator produces a random stream of data for cores with a streaming/framing interface.

The scripts provided in the `implement` and `functional` subdirectories can be used to quickly get an Aurora 64B/66B design up and running on a board, or perform a quick simulation of the module. The design can also be used as a reference for connecting the trickier interfaces on the Aurora 64B/66B core, such as the clocking interface.

When using the example design on a board, be sure to edit the `<component name>_exdes` file in the `example_design` subdirectory to supply the correct pins and clock constraints. [Table 6-1](#) describes the ports available in the example design.

Table 6-1: Example Design I/O Ports

Port	Direction	Description
RXN[0:m-1]	Input	Negative differential serial data input pin.
RXP[0:m-1]	Input	Positive differential serial data input pin.
TXN[0:m-1]	Output	Negative differential serial data output pin.
TXP[0:m-1]	Output	Positive differential serial data output pin.
RESET	Input	Reset signal for the example design. The active-High reset is debounced using a USER_CLK signal generated from the reference clock input.
<reference clock(s)>	Input	The reference clocks for the Aurora 64B/66B core are brought to the top level of the example design. See Clock Interface and Clocking in Chapter 3 for details about the reference clocks.
<core error signals>	Output	The error signals from the Aurora 64B/66B core Status and Control interface are brought to the top level of the example design and registered. See Status, Control, and the Transceiver Interface in Chapter 3 for details.
<core channel up signals>	Output	The channel up status signals for the core are brought to the top level of the example design and registered. See Status, Control, and the Transceiver Interface in Chapter 3 for details.
<core lane up signals>	Output	The lane up status signals for the core are brought to the top level of the example design and registered. Cores have a lane up signal for each GTX/GTH transceiver they use. See Status, Control, and the Transceiver Interface in Chapter 3 for details.
PMA_INIT	Input	The reset signal for the PCS and PMA modules in the GTX/GTH transceivers is connected to the top level through a debouncer. The signal is debounced using the INIT_CLK. See the Reset section in the <i>Virtex-6 FPGA GTX Transceivers User Guide</i> , the <i>Virtex-6 FPGA GTH Transceivers User Guide</i> , or the <i>7 Series FPGAs GTX/GTH Transceivers User Guide</i> for further details on GT RESET.
INIT_CLK	Input	INIT_CLK is used to register and debounce the PMA_INIT signal. INIT_CLK must not come from a GTX/GTH transceiver, and should be set to a slow rate, preferably slower than the reference clock.
DATA_ERR_COUNT[0:7]	Output	Count of the number of frame data words received by the FRAME_CHECK that did not match the expected value.
UFC_ERR	Output	Asserted (active-High) when UFC data words received by the FRAME_CHECK that did not match the expected value.
USER_K_ERR	Output	Asserted (active-High) when User K data words received by the FRAME_CHECK that did not match the expected value.

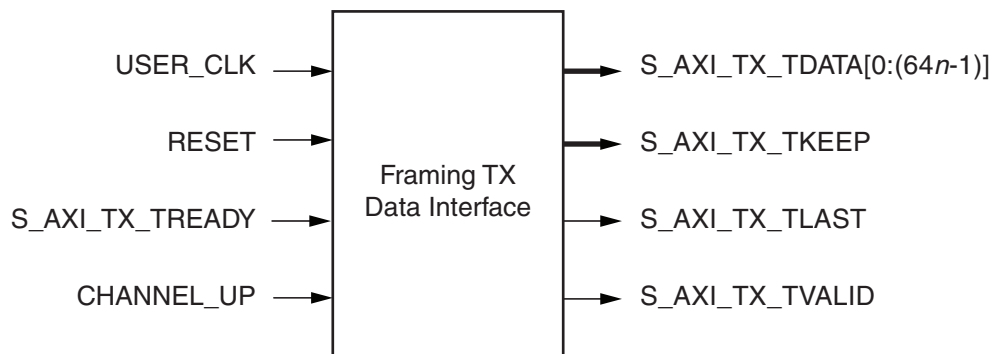
FRAME_GEN

Framing TX Data Interface

To transmit the user data, the FRAME_GEN user data state machine manipulates control signals to do the following:

- After the Aurora interface is out of RESET and reaches CHANNEL_UP state, pseudo-random data is generated using the user data linear feedback shift register (LFSR) and connected to S_AXI_TX_TDATA bus.
- Generates the S_AXI_TX_TLAST for the current frame based on two counters. An 8-bit counter is used to determine the size of the frame and another 8-bit counter to keep track of number of user data bytes sent. Frame size counter is initialized and incremented by one for every frame.
- S_AXI_TX_TKEEP bus is connected to lower bits of user data LFSR to generate SEP and SEP7 conditions.
- S_AXI_TX_TVALID is asserted according to AXI4-Stream protocol specification.
- User data state machine state transitions are controlled by S_AXI_TX_TREADY provided by Aurora's AXI4-Stream interface.
- Various kinds of frame traffic are generated including single cycle frame.

Figure 6-2 shows the FRAME_GEN framing user interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for TX data.



UG775_c10_02_050211

Figure 6-2: Aurora 64B/66B Core Framing TX Data Interface (FRAME_GEN)

Table 6-2 lists the FRAME_GEN framing TX data ports and their descriptions.

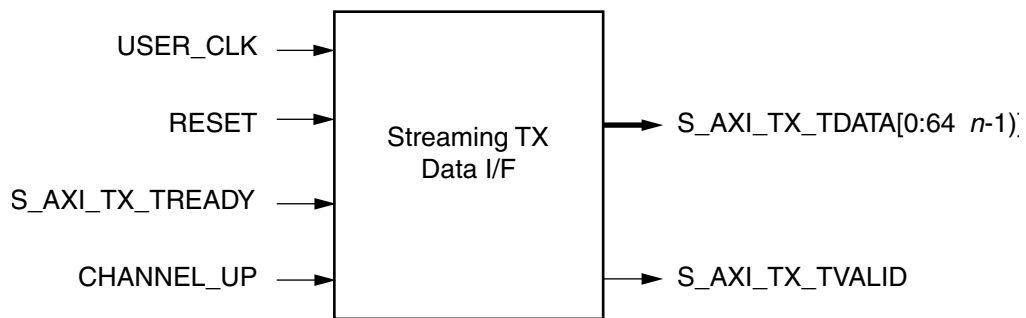
Table 6-2: FRAME_GEN Framing User I/O Ports (TX)

Name	Direction	Description
S_AXI_TX_TDATA[0:(64n-1)]	Output	User frame data. Width is 64*n where n is the number of lanes.
S_AXI_TX_TKEEP[0:n-1]	Output	Specifies the number of valid bytes in the last data beat; Valid only while S_AXI_TX_TLAST is asserted High.
S_AXI_TX_TVALID	Output	Asserted (active-High) when AXI4-Stream signals from the source are valid. Deasserted (Low) when AXI4-Stream control signals and/or data from the source should be ignored.
S_AXI_TX_TLAST	Output	Signals the end of the frame data (active-High).
S_AXI_TX_TREADY	Input	Asserted (active-High) during clock edges when signals from the source are accepted (if S_AXI_TX_TVALID is also asserted). Deasserted (Low) on clock edges when signals from the source are ignored.
CHANNEL_UP	Input	Asserted when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

Streaming TX Data Interface

Streaming TX data interface is similar to framing TX data interface without framing delimiters, S_AXI_TX_TLAST, and S_AXI_TX_TKEEP. To transmit the user data, the FRAME_GEN user data state machine manipulates control signals to do the following:

- After the Aurora interface is out of RESET and reaches CHANNEL_UP state, pseudo-random data is generated using LFSR and connected to S_AXI_TX_TDATA bus.
- LFSR generates new data for every assertion of S_AXI_TX_TREADY.
- S_AXI_TX_TVALID is always asserted.



X13022

Figure 6-3: Aurora 64B/66B Core Streaming TX Data Interface (FRAME_GEN)

Table 6-3 lists the FRAME_GEN streaming TX data ports and their descriptions.

Table 6-3: FRAME_GEN Streaming User I/O Ports (TX)

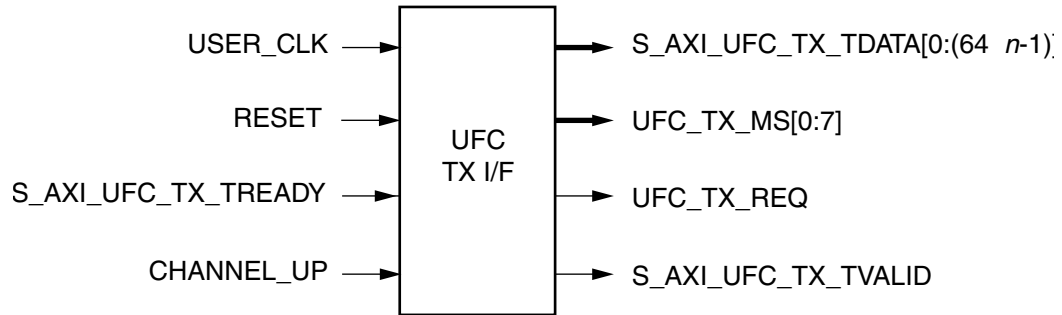
Name	Direction	Description
S_AXI_TX_TDATA[0:(64n-1)]	Output	Outgoing frame data. Width is 64*n where n is the number of lanes.
S_AXI_TX_TVALID	Output	Asserted (active-High) when AXI4-Stream signals from the source are valid. Deasserted (Low) when AXI4-Stream control signals and/or data from the source should be ignored.
S_AXI_TX_TREADY	Input	Asserted (active-High) during clock edges when signals from the source are accepted (if S_AXI_TX_TVALID is also asserted). Deasserted (Low) on clock edges when signals from the source are ignored.
CHANNEL_UP	Input	Asserted when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

UFC TX Interface

To transmit the UFC data, the FRAME_GEN UFC state machine manipulates control signals to do the following:

- Asserts UFC_TX_REQ after CHANNEL_UP indication from the Aurora TX interface.
- UFC_TX_MS[0:7] is also transmitted along with UFC_TX_REQ. UFC_TX_MS transmits zero initially for the first UFC frame and is incremented by one for the following UFC frames until it reaches 255 (maximum value).
- S_AXI_UFC_TX_TVALID is asserted after placing the UFC_TX_REQ.
- S_AXI_UFC_TX_TDATA is transmitted after receiving S_AXI_UFC_TX_TREADY from the Aurora TX interface.
- UFC frame transmission frequency is controlled by the UFC_IFG parameter

Figure 6-4 shows the FRAME_GEN UFC TX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for UFC TX data.



X13027

Figure 6-4: Aurora 64B/66B Core UFC TX Interface (FRAME_GEN)

Table 6-4 lists the FRAME_GEN UFC TX data ports and their descriptions.

Table 6-4: FRAME_GEN UFC User I/O Ports (TX)

Name	Direction	Description
UFC_TX_REQ	Output	Asserted to request a UFC message to be sent to the channel partner (active-High). Requests are processed after a single cycle, unless another UFC message is in progress and not on its last cycle. After a request, the S_AXI_UFC_TX_TDATA bus is ready to send data within two cycles unless interrupted by a higher priority event.
UFC_TX_MS[0:7]	Output	Specifies the number of bytes in the UFC message (the Message Size). The max UFC Message Size is 256. The value specified at UFC_TX_MS is one less than the actual amount of bytes transferred. For example, a value of 3 will transmit 4 bytes of data.
S_AXI_UFC_TX_TDATA [0:(64n-1)]	Output	Output bus for UFC message data to the Aurora channel. Data is read from the bus into the channel only when both S_AXI_UFC_TX_TVALID and S_AXI_UFC_TX_TREADY are asserted on a positive USER_CLK edge. If the number of bytes in the message is not an integer multiple of the bytes in the bus, on the last cycle, only the bytes needed to finish the message starting from the left of the bus are used.
S_AXI_UFC_TX_TVALID	Output	Assert (active-High) when data on S_AXI_UFC_TX_TDATA is valid. If deasserted while S_AXI_UFC_TX_TREADY is asserted, Idle blocks are inserted in the UFC message.
S_AXI_UFC_TX_TREADY	Input	Asserted (active-High) when an Aurora 64B/66B core is ready to read data from the S_AXI_UFC_TX_TDATA interface. This signal is asserted one clock cycle after UFC_TX_REQ is asserted and no high priority requests in progress. S_AXI_UFC_TX_TREADY continues to be asserted while the core waits for data for the most recently requested UFC message. The signal is deasserted for CC and NFC requests, which are higher priority. While S_AXI_UFC_TX_TREADY is asserted, S_AXI_TX_TREADY is deasserted.
CHANNEL_UP	Input	Asserted when Aurora channel initialization is complete and channel is ready to send data.

Table 6-4: FRAME_GEN UFC User I/O Ports (TX) (Cont'd)

Name	Direction	Description
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

NFC TX Interface

To transmit the NFC frame, the FRAME_GEN NFC state machine manipulates control signals to do the following:

- NFC state machine waits until TX user data transmission and enters into NFC XON mode.
- S_AXI_NFC_TX_TDATA value is transmitted along with S_AXI_NFC_TX_TVALID.
- After predefined period of time, NFC state machine enters into NFC XOFF mode.
- NFC state transitions are governed by S_AXI_NFC_TX_TREADY
- NFC frame transmission frequency is controlled by NFC_IFG parameter.

Figure 6-5 shows the FRAME_GEN NFC TX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for NFC TX data.

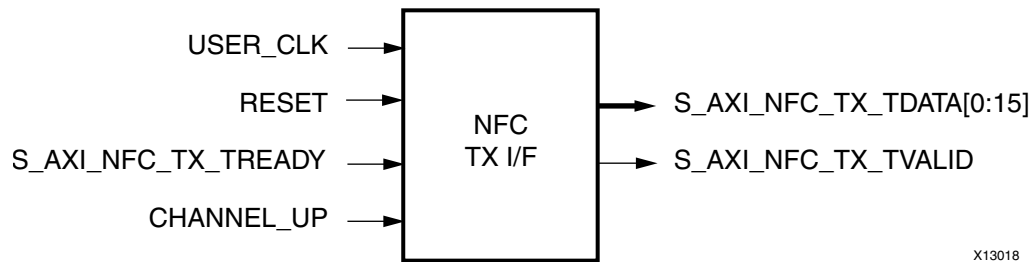


Figure 6-5: Aurora 64B/66B Core NFC TX Interface (FRAME_GEN)

Table 6-5 lists the FRAME_GEN NFC TX data ports and their descriptions.

Table 6-5: FRAME_GEN NFC User I/O Ports (TX)

Name	Direction	Description
S_AXI_NFC_TX_TVALID	Output	Asserted to request an NFC message to be sent to the channel partner (active-High). Must be held until S_AXI_NFC_TX_TREADY is asserted.
S_AXI_NFC_TX_TDATA [0:15]	Output	Indicates how many USER_CLK cycles the channel partner must wait before it can send data when it receives the NFC message. Must be held until S_AXI_NFC_TX_TREADY is asserted. The number of USER_CLK cycles without data is equal to S_AXI_NFC_TX_TDATA[8:15] + 1. S_AXI_NFC_DATA[7] (active-High) is mapped to NFC_XOFF, which requests the channel partner to stop sending data until it receives a non-XOFF NFC message or is reset. Signal Mapping: S_AXI_NFC_TX_TDATA = {6'h0, NFC XOFF bit, NFC Data}
S_AXI_NFC_TX_TREADY	Input	Asserted when an Aurora core accepts an NFC request (active-High).
CHANNEL_UP	Input	Asserted when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

User K TX Interface

To transmit the User K data, FRAME_GEN manipulates control signals to do the following:

- S_AXI_USER_K_TX_TVALID is asserted after User K inter-frame gap.
- Pre-defined User K data is transmitted along with User K Block No. User K Block No is set as zero for the first User K-block and is incremented by one for the following User K-blocks until it reaches 8.
- User K transmission frequency is controlled by USER_K_IFG parameter.

Figure 6-6 shows the FRAME_GEN User K TX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for User K TX data.

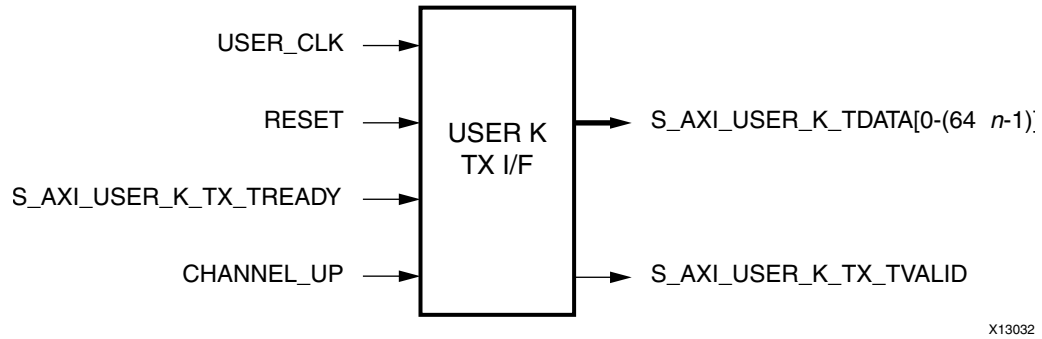


Figure 6-6: Aurora 64B/66B Core User K TX Interface (FRAME_GEN)

Table 6-6 lists the FRAME_GEN User K TX data ports and their descriptions.

Table 6-6: FRAME_GEN User K User I/O Ports (TX)

Name	Direction	Description
S_AXI_USER_K_TDATA [0: (n*64-1)]	Output	User K-block data. S_AXI_USER_K_TX_TDATA = {4'h0, USER K BLOCK NO, USER K DATA[0:56n-1]}
S_AXI_USER_K_TX_TVALID	Output	Asserted (active-High) when User K data on S_AXI_USER_K_TDATA port is valid.
S_AXI_USER_K_TX_TREADY	Input	Asserted (active-High) when the Aurora 64B/66B core is ready to read data from the S_AXI_USER_K_TX_TDATA interface.
CHANNEL_UP	Input	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

FRAME_CHECK

Framing RX Data Interface

The expected frame RX data is computed by LFSR. The received user data is validated by checking against following AXI4-Stream protocol rules:

Start the frame when M_AXI_RX_TVALID is asserted

1. M_AXI_RX_TKEEP bus is valid during M_AXI_RX_TLAST assertion.
2. M_AXI_RX_TVALID should be asserted during comparison of expected to actual data:

Incoming RX data through M_AXI_RX_TDATA port is registered and compared with calculated RX data internal to FRAME_CHECK. If the incoming RX data does not match with expected RX data, an 8-bit counter is incremented. This error counter is indicated to the user application through DATA_ERR_COUNT port. The Error counter freezes counting when it reaches 255.

Note: The counter can be cleared by applying reset.

Figure 6-7 shows the FRAME_CHECK framing user interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for RX data.

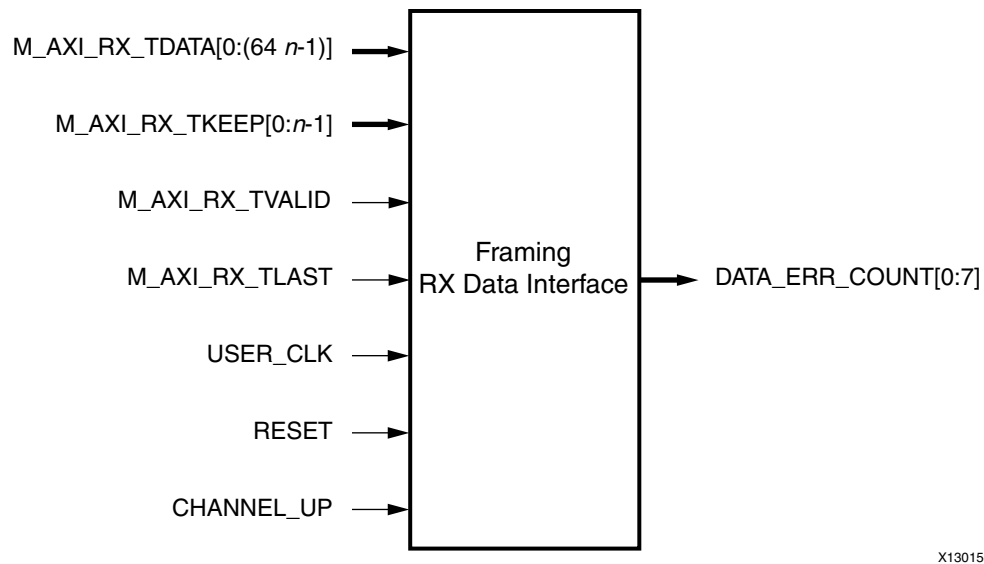


Figure 6-7: Aurora 64B/66B Core Framing RX Data Interface (FRAME_CHECK)

Table 6-7 lists the FRAME_CHECK framing RX data ports and their descriptions.

Table 6-7: FRAME_CHECK Framing User I/O Ports (RX)

Name	Direction	Description
M_AXI_RX_TDATA[0:(64n-1)]	Input	Incoming frame data from channel partner (Ascending bit order).
M_AXI_RX_TKEEP[0:n-1]	Input	Specifies the number of valid bytes in the last data beat. Valid only when M_AXI_RX_TLAST is asserted.
M_AXI_RX_TVALID	Input	Asserted (active-High) when data and control signals from an Aurora core are valid. Deasserted (Low) when data and/or control signals from an Aurora core should be ignored.
M_AXI_RX_TLAST	Input	Signals the end of the incoming frame (active-High, asserted for a single USER_CLK cycle).
DATA_ERR_COUNT[0:7]	Output	Count of the number of RX frame data words received by the frame checker that did not match the expected value.
CHANNEL_UP	Input	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.

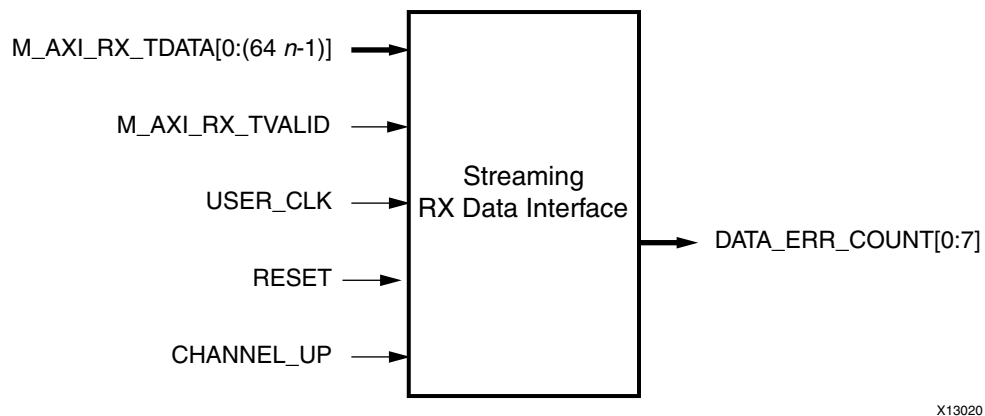
Table 6-7: FRAME_CHECK Framing User I/O Ports (RX) (Cont'd)

Name	Direction	Description
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

Streaming RX Data Interface

- In streaming mode, the incoming RX data is compared against calculated RX data.
- The RX data is compared only when M_AXI_RX_TVALID is asserted.

Figure 6-8 shows the FRAME_CHECK streaming user interface of the Aurora 64B/66B core ports for RX data.



X13020

Figure 6-8: Aurora 64B/66B Core Streaming RX Data Interface (FRAME_CHECK)

Table 6-8 lists the FRAME_CHECK streaming RX data ports and their descriptions.

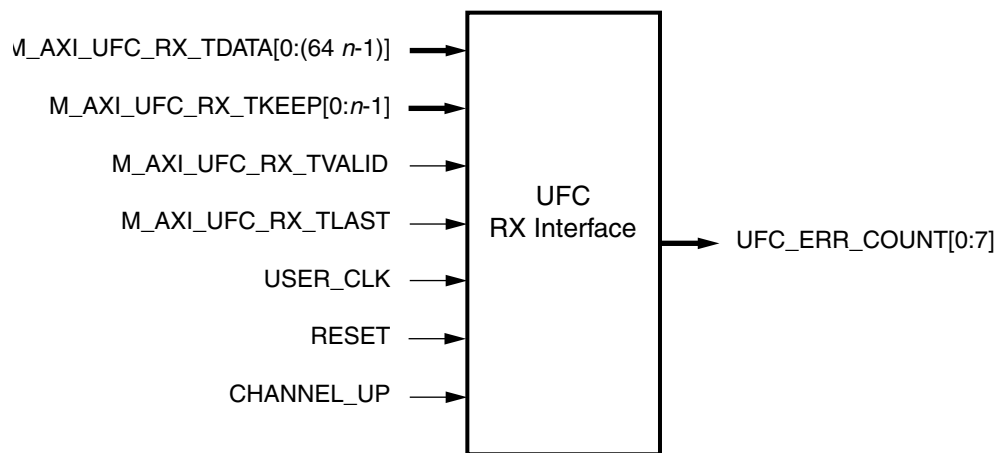
Table 6-8: FRAME_CHECK Streaming User I/O Ports (RX)

Name	Direction	Description
M_AXI_RX_TDATA[0:(64n-1)]	Input	Incoming frame data from channel partner (ascending bit order).
M_AXI_RX_TVALID	Input	Asserted (active-High) when data and control signals from an Aurora core are valid. Deasserted (Low) when data and/or control signals from an Aurora core should be ignored.
DATA_ERR_COUNT[0:7]	Output	Count of the number of RX data words received by the frame checker that did not match the expected value.
CHANNEL_UP	Input	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

UFC RX Interface

- Expected UFC RX data is computed by LFSR.
- Error checking and counter logic is similar to that of [Framing RX Data Interface](#).
- If the incoming `M_AXI_UFC_RX_TDATA` does not match with expected RX UFC data, an 8-bit error counter is incremented.
- The error counter is indicated to the user application through the `UFC_ERR_COUNT` port.

Figure 6-9 shows the `FRAME_CHECK` UFC RX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for UFC RX data.



X13025

Figure 6-9: Aurora 64B/66B Core UFC RX Interface (FRAME_CHECK)

Table 6-9 lists the FRAME_CHECK UFC RX data ports and their descriptions.

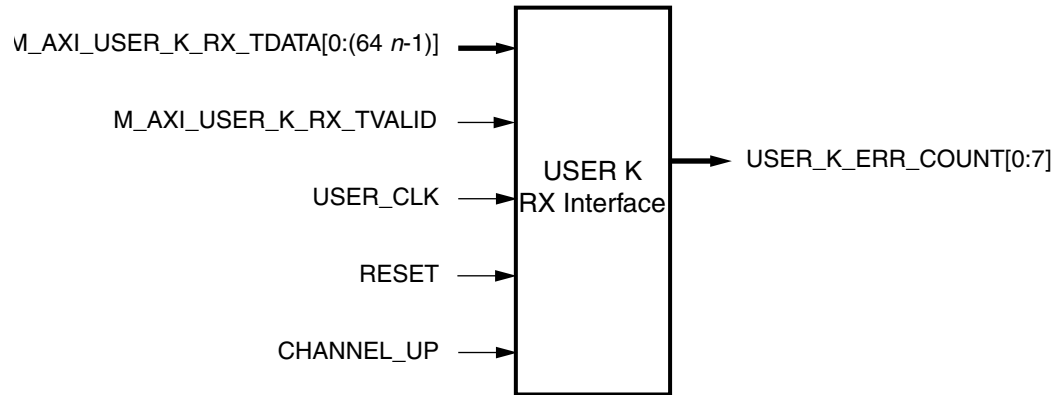
Table 6-9: FRAME_CHECK UFC User I/O Ports (RX)

Name	Direction	Description
M_AXI_UFC_RX_TDATA [0: (64n-1)]	Input	Incoming UFC message data from the channel partner.
M_AXI_UFC_RX_TKEEP [0: n-1]	Input	Specifies the number of valid bytes of data presented on the M_AXI_UFC_RX_TDATA port on the last word of a UFC message. Valid only when M_AXI_UFC_RX_TLAST is asserted. n = 256 bytes maximum.
M_AXI_UFC_RX_TVALID	Input	Asserted (active-High) when the values on the M_AXI_UFC_RX_TDATA port is valid. When this signal is not asserted, all values on the M_AXI_UFC_RX_TDATA port should be ignored.
M_AXI_UFC_RX_TLAST	Input	Signals the end of the incoming UFC message.
UFC_ERR_COUNT[0:7]	Output	Count of the number of RX UFC data words received by the frame checker that did not match the expected value.
CHANNEL_UP	Input	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

User K RX Interface

- M_AXI_USER_K_RX_TVALID to be asserted during comparison of expected to actual User K data
- Incoming M_AXI_USER_K_RX_TDATA is compared against predefined User K data.
- 8-bit USER_K_ERR_COUNT is incremented if the comparison fails.
- The error counter is indicated to the user application through USER_K_ERR_COUNT port.

Figure 6-10 shows the FRAME_CHECK User K RX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for User K RX data.



X13029

Figure 6-10: Aurora 64B/66B Core User K RX Interface (FRAME_CHECK)

Table 6-10 lists the FRAME_CHECK User K RX data ports and their descriptions.

Table 6-10: FRAME_CHECK User K User I/O Ports (RX)

Name	Direction	Description
M_AXI_USER_K_RX_TVALID	Input	Asserted (active-High) when User K data on M_AXI_USER_K_RX_TDATA port is valid.
M_AXI_USER_K_RX_TDATA [0:(n*64-1)]	Input	Receive User K-blocks from the Aurora lane. Signal Mapping per lane: M_AXI_USER_K_RX_TDATA = {4'h0, User K Block No, User K Data}
USER_K_ERR_COUNT[0:7]	Output	Count of the number of RX User K data words received by the frame checker that did not match the expected value.
CHANNEL_UP	Input	Asserted when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-High).

The Aurora 64B/66B example design has been tested with XST for synthesis and Mentor Graphics ModelSim for simulation.

Generating the Core

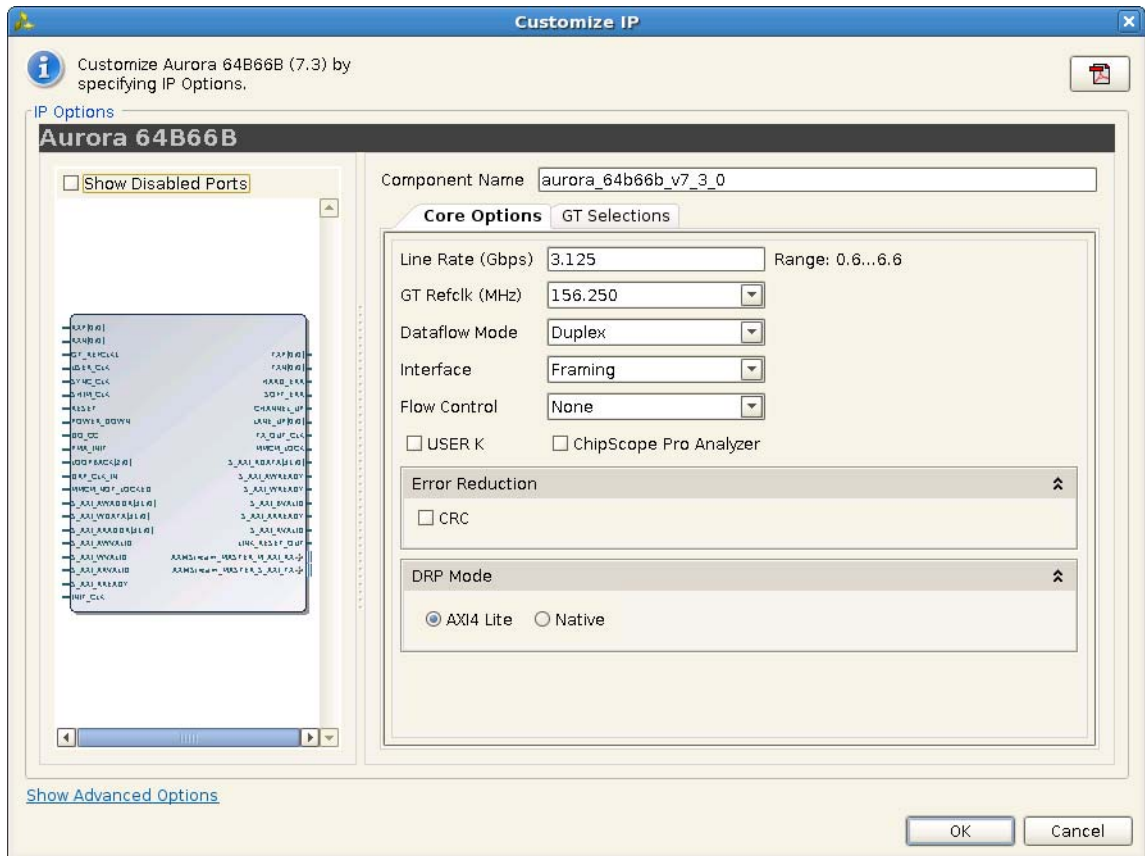


Figure 6-11: Aurora 64B/66B IP Catalog Customization Screen - Page 1

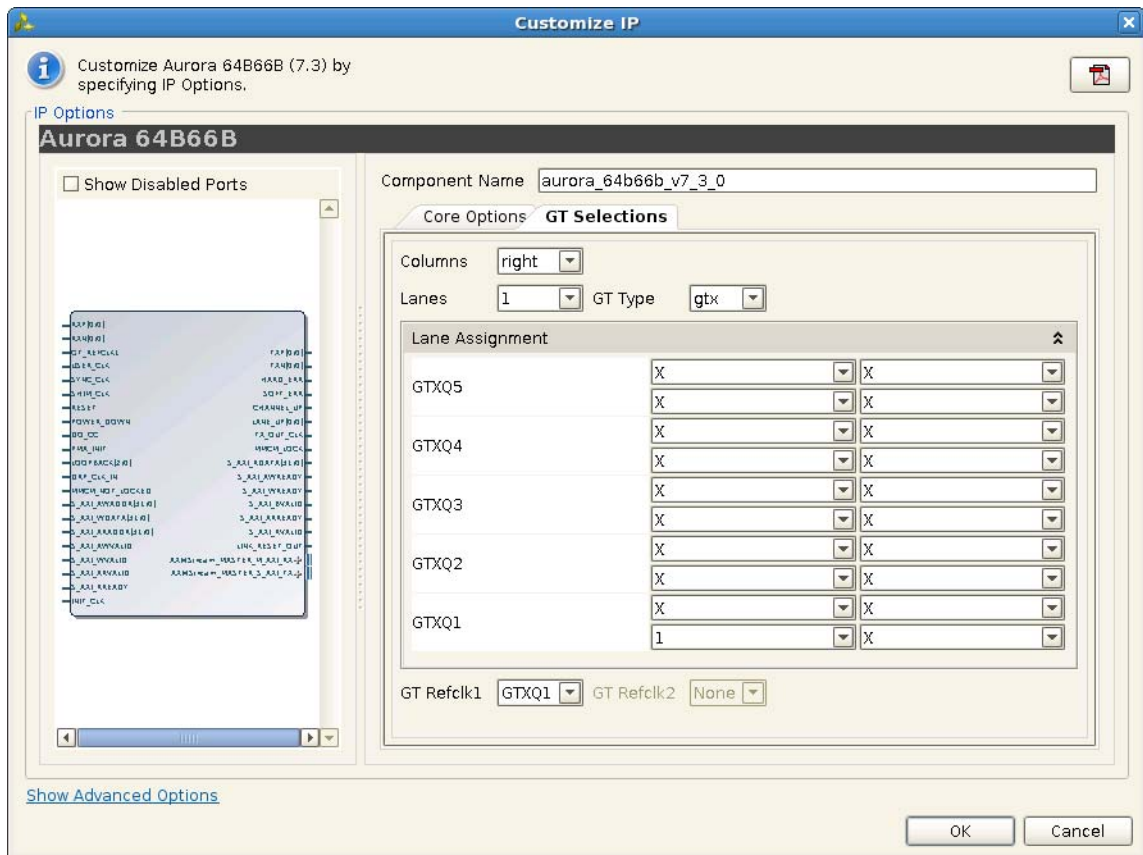


Figure 6-12: Aurora 64B/66B IP Catalog Customization Screen - Page 2

Implementing the Example Design

The example design needs to be generated from the IP core. To do that, right-click the generated IP. Click **Open Example Design** on the menu displayed for the right-click operation. This action opens an example design for the generated IP core. You can click **Run Implementation** to run the Synthesis followed by implementation. Additionally you can also generate a bitstream by clicking **Generate Bitstream**.

SECTION III: ISE DESIGN SUITE

Customizing and Generating the Core

Constraining the Core

Detailed Example Design

Customizing and Generating the Core

This chapter includes information on using ISE® Design Suite tools to customize and generate the LogiCORE™ IP Aurora 64B/66B core.

GUI

The Aurora 64B/66B core can be customized to suit a wide variety of requirements using the CORE Generator™ tool. This chapter details the customization parameters available to the user application and how these parameters are specified within the IP Customizer interface.

Using the IP Customizer

The Aurora 64B/66B IP Customizer is presented when you select the Aurora 64B/66B core in the CORE Generator tool. For help starting and using the CORE Generator tool, see the *CORE Generator Guide* in the ISE tool documentation. [Figure 7-1, page 101](#), [Figure 7-2, page 102](#), [Figure 7-3, page 103](#), and [Figure 7-4, page 104](#) show features that are described in corresponding sections.

Note: The options shown in [Figure 7-3](#) and [Figure 7-4](#) are only available for the Virtex®-6 HXT devices.

IP Customizer

[Figure 7-1](#) and [Figure 7-2](#) show the customizer. The left side displays a representative block diagram of the Aurora 64B/66B core as currently configured. The right side consists of user-configurable parameters. Details on the customizing options are provided in the following subsections.

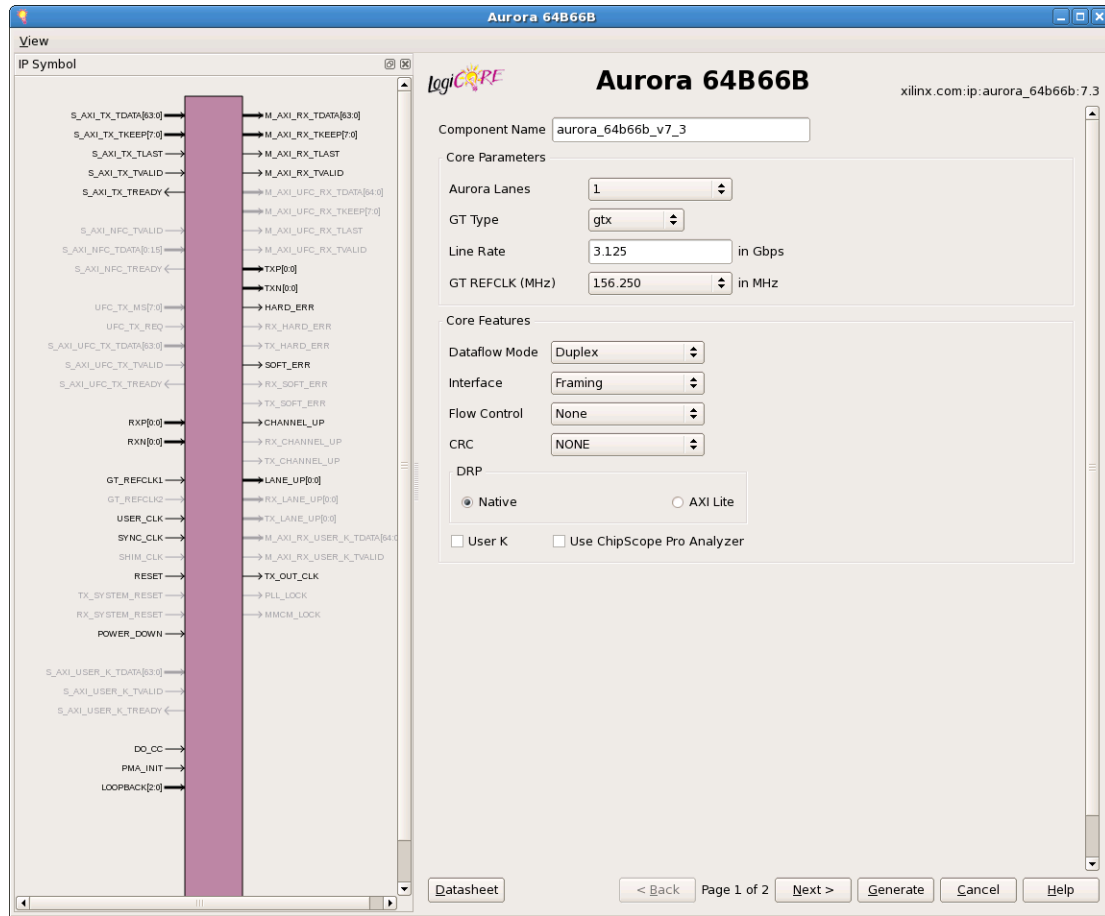


Figure 7-1: Aurora 64B/66B IP Customizer Page 1

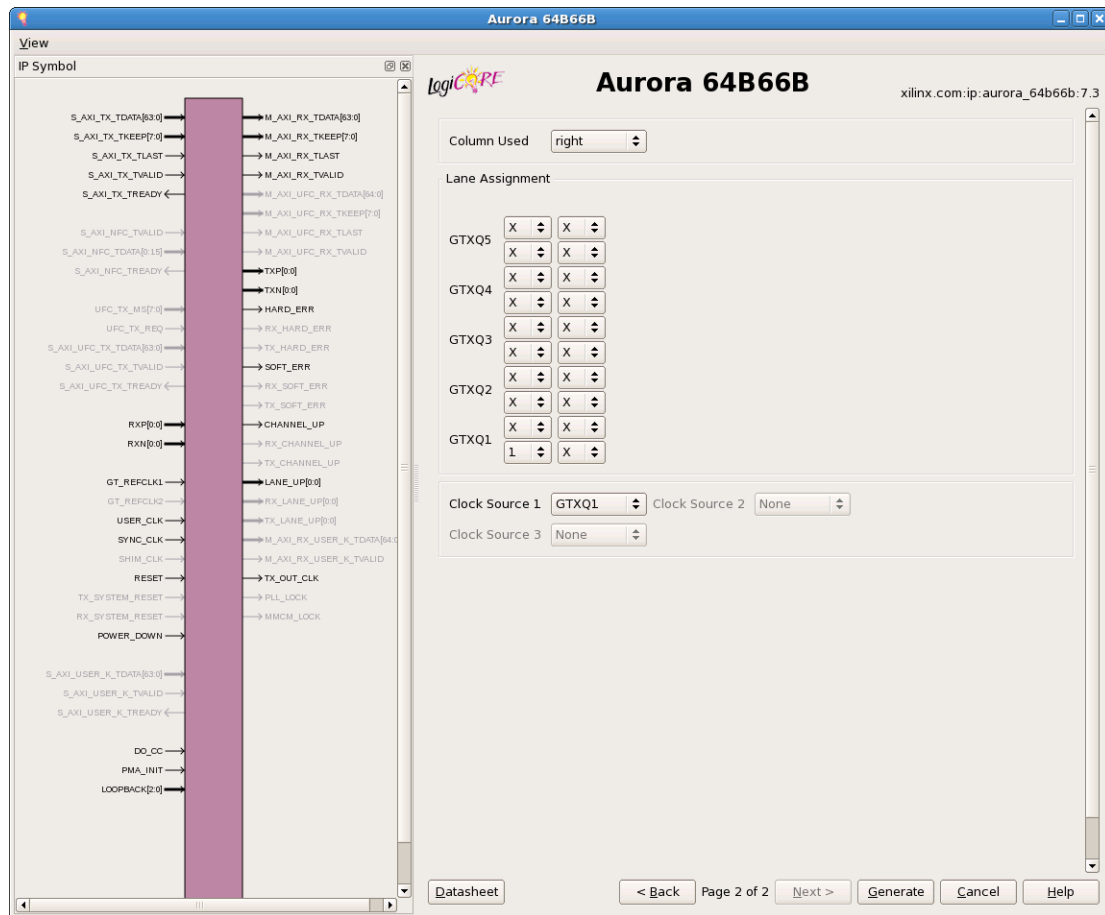


Figure 7-2: Aurora 64B/66B IP Customizer Page 2

The options shown in Figure 7-3 and Figure 7-4 are available only for the Virtex-6 HXT devices.

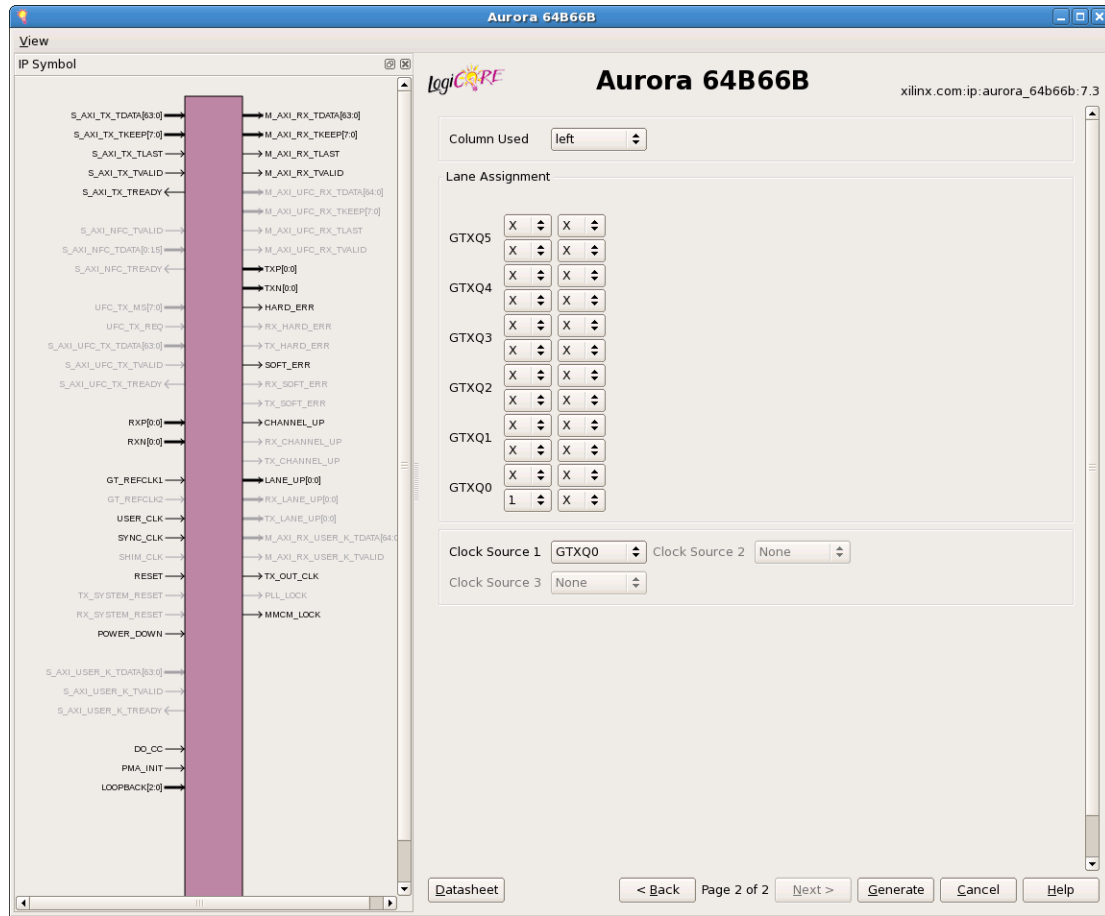


Figure 7-3: Aurora 64B/66B IP Customizer Page 2 (Virtex-6 HXT Devices for GTX Transceivers Only)

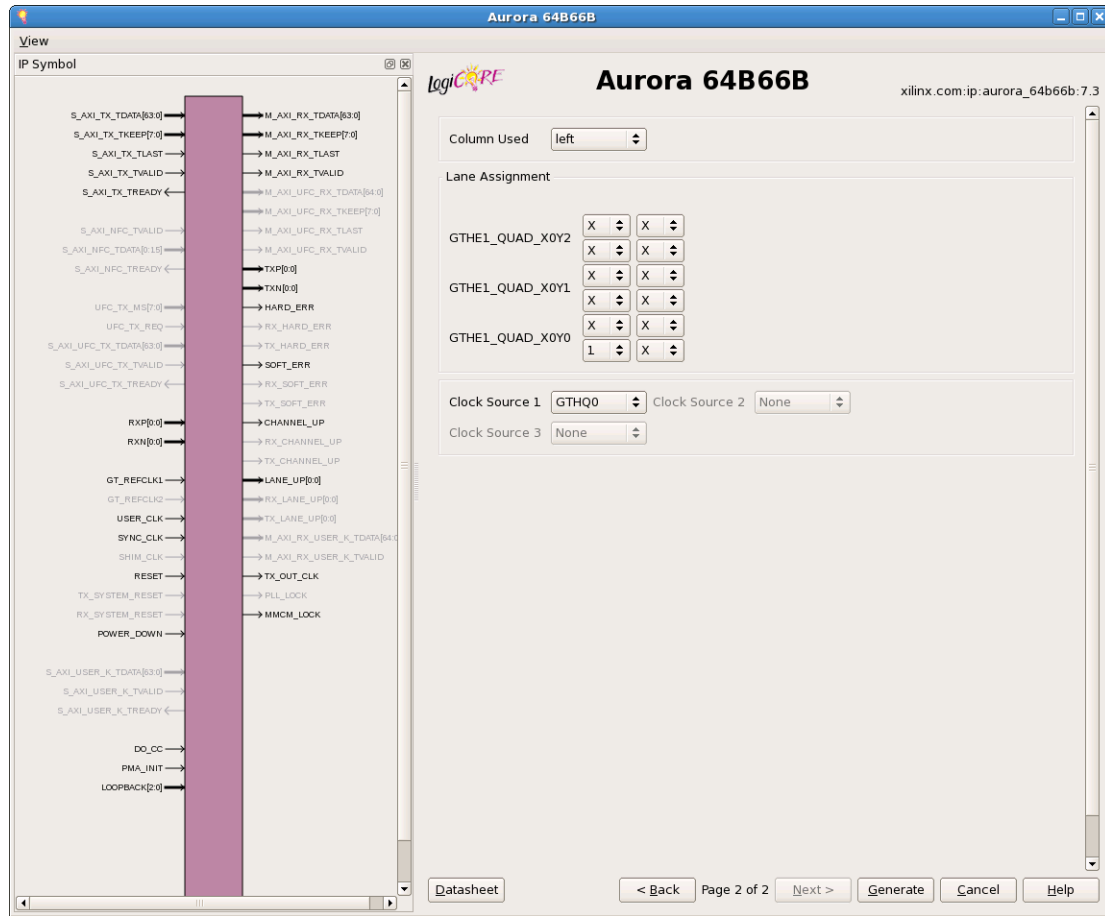


Figure 7-4: Aurora 64B/66B IP Customizer Page 2 (Virtex-6 HXT Devices for GTH Transceivers Only)

Component Name

Enter the top-level name for the core in this text box. Illegal names are highlighted in red until they are corrected. All files for the generated core are placed in a subdirectory using this name. The top-level module for the core also use this name.

Default: `aurora_64b66b_v7_3`

Lane Assignment

See the diagram in the information area in [Figure 7-1, page 101](#). Each numbered row represents a GT transceiver tile and each active box represents an available GTX/GTH transceiver. For each Aurora lane in the core, starting with Lane 1, select a GTX/GTH transceiver and place the lane by selecting its number in the GTX/GTH placement box.

Aurora Lanes

Select the number of lanes (GTX/GTH transceivers) to be used in the core. The valid range depends on the target device selected.

Default: 1

Interface

Select the type of datapath interface used for the core. Select Framing to use a complete AXI4-Stream interface that allows encapsulation of data frames of any length. Select Streaming to use a simple word-based interface with a data valid signal to stream data through the Aurora 64B/66B channel. See [User Interface in Chapter 2](#) for more information.

Default: Framing

Data Flow Mode

Select the options for the direction of the channel that the Aurora 64B/66B core will support. Simplex Aurora 64B/66B cores have a single, unidirectional serial port that connects to a complementary simplex Aurora 64B/66B core. Available options are **RX-only Simplex**, **TX-only Simplex**, and **Duplex**. See [Status, Control, and the Transceiver Interface, page 60](#) for more information.

Default: Duplex

Flow Control

Select the required option to add flow control to the core. *User* flow control (UFC) allows applications to send each other brief, high-priority messages through the Aurora channel. *Native* flow control (NFC) allows full-duplex receivers to regulate the rate of the data sent to them. Immediate mode allows idle codes to be inserted within data frames while completion mode only inserts idle codes between complete data frames.

Available options are:

- None
- UFC only
- Immediate Mode - NFC
- Completion Mode - NFC
- UFC + Immediate Mode - NFC
- UFC + Completion Mode - NFC

For the streaming interface, only immediate mode is available. For the framing interface, both immediate and completion modes are available.

Default: None

CRC

Select the option to insert CRC32 in the data stream.

Available options are:

- None
- CRC32

Default: None

User K

Select to add User K interface to the core. User K-blocks are special single-block codes passed directly to the user application. These blocks are used to implement application-specific control functions.

Default: Unchecked

DRP

Select the required interface to control or monitor the Transceiver interface using the Dynamic Reconfiguration Port (DRP).

Available options are:

- Native
- AXI4_Lite

Default: Native

GT_TYPE

Select the type of serial transceiver from the drop-down list. This option is applicable only for Virtex-6 HXT or Virtex-7 XT devices. For other devices, the drop-down box is not visible.

Available options are:

- GTX
- GTH
- V7GTH

Default: gtx

Line Rate

Enter a floating-point value in gigabits per second. The value entered must be within the valid range shown. This determines the unencoded bit rate at which data is transferred over the serial link.

Default: 3.125 Gb/s for GTX transceivers and Virtex-7 FPGA GTH transceivers, and 10.3125 Gb/s for Virtex-6 GTH transceivers

GT Reference Clock Frequency

Select a reference clock frequency from the drop-down list. Reference clock frequencies are given in megahertz, and depend on the line rate selected. For best results, select the highest rate that can be practically applied to the reference clock input of the target device.

Default: 156.250 MHz

Clock Source 1, Clock Source 2, and Clock Source 3

Select reference clock sources for the GTX/GTH tiles from the drop-down list in this section.

Default: Clock Source 1: GTXQn/ GTHQn; Clock Source 2: None; Clock Source 3: None

Note:

- Clock Source 3 is enabled only for Virtex-6 FPGA GTH transceivers depending on the number of lanes and the line rate.
- n depends on the serial transceiver (GTX/GTH) position.

Column Used

Select appropriate column from the drop-down list. This option is applicable only for Virtex-7, Kintex™-7, Virtex-6 HXT and devices. For other devices, the drop-down box is not visible.

Default: left

ChipScope Pro Analyzer

Select to add ChipScope™ Pro IP cores to the Aurora 64B/66B core. (See [Quick Start Example Design, page 118](#).) This option provides users a debugging interface that shows the core status signals in the ChipScope Pro analyzer tool.

Default: Unchecked

Generate

Click **Generate** to generate the core. (See [Generating the Core, page 121](#).) The modules for the Aurora 64B/66B core are written to the CORE Generator tool project directory using the same name as the top level of the core.

Output Generation

The customized Aurora 64B/66B core is delivered as a set of HDL source modules in the language selected in the CORE Generator™ tool project with supporting script and documentation files. These files are arranged in a predetermined directory structure under the project directory name provided to the CORE Generator tool when the project is created as shown in this section.

Directory and File Structure

<project directory>

Top-level project directory; name is user-defined.

<project directory>/<component name>

Core readme file

<component name>/doc

Product documentation

<component name>/example_design

Example design files

/example_design/cc_manager

Verilog/VHDL design files for the clock management block

/example_design/clock_module

Verilog/VHDL design files for the clocking blocks

/example_design/gt

Verilog/VHDL design files for the GTX/GTH transceiver

/example_design/traffic_gen_and_check

Verilog/VHDL design files for the frame generator and checker

<component name>/implement

Implementation scripts and support files

/implement/results


Implement script results

<component name>/simulation

Simulation test bench and simulation script files

/simulation/functional

Functional simulation files

 `<component name>/src`
Verilog/VHDL files for the core

Directory and File Contents

The Aurora 64B/66B core directories and their associated files are defined in the following sections.

<project directory>

The `project` directory contains the CORE Generator tool project files.

Table 7-1: **project Directory**

Name	Description
<project directory>	
<coregen project filename>.cgp	CORE Generator tool project file
<component name>.v[hd]	Aurora 64B/66B core top-level file
<component name>.ucf	Aurora 64B/66B core design constraints
<component name>.xdc	Aurora 64B/66B core design constraints (only for 7 series devices)

[Back to Top](#)

<project directory>/<component name>

The `component name` directory contains the core file.

Table 7-2: **component name Directory**

Name	Description
<project directory>/<component name>	
aurora_64b66b_v7_3_readme.txt	Readme file

[Back to Top](#)

<component name>/doc

The `doc` directory contains a Xilinx website redirect to the product documentation.

Table 7-3: **doc Directory**

Name	Description
<component name>/doc	
pg074-aurora-64b66b.pdf	<i>LogiCORE IP Aurora 64B/66B v7.3 Product Guide</i>
aurora_64b66b_v7_3_vinfo.html	Version information file

[Back to Top](#)

<component name>/example_design

The `example_design` directory contains the example design files provided with the core.

Table 7-4: **example_design Directory**

Name	Description
<component name>/example_design	
<component name>_exdes.v[hd]	Example design source file
v6_icon.ngc v6_ila.ngc v6_vio.ngc k7_icon.ngc k7_ila.ngc k7_vio.ngc	NGC files for the debug cores compatible with the ChipScope Pro Analyzer tool
<component name>_exdes.ucf	Aurora 64B/66B example design constraints
<component name>_exdes.xdc	Aurora 64B/66B example design constraints (only for 7 series devices)
<component name>_reset_logic.v[hd]	Aurora 64B/66B reset logic

[Back to Top](#)

/example_design/cc_manager

The `cc_manager` directory contains the clock compensation source file.

Table 7-5: **cc_manager Directory**

Name	Description
<component name>/example_design/cc_manager	
<component name>_standard_cc_module.v[hd]	Clock compensation module source file

[Back to Top](#)

/example_design/clock_module

The `clock_module` directory contains the clock module source file.

Table 7-6: **clock_module Directory**

Name	Description
<component name>/example_design/clock_module	
<component name>_clock_module.v[hd]	Clock module source file
<component name>_enable_generator.v[hd]	Clock enable generator

[Back to Top](#)

/example_design/gt

The `gt` directory contains the Verilog/VHDL wrapper files for the GTX/GTH transceiver.

Table 7-7: `gt` Directory

Name	Description
<component name>/example_design/gt	
<component name>_gt_wrapper.v[hd] <component name>_multi_wrapper.v[hd] <component name>_gth_init.v[hd] ⁽²⁾ <component name>_gtx.v[hd] ⁽¹⁾ <component name>_quad.v[hd] ⁽³⁾ <component name>_gth_reset.v[hd] ⁽²⁾ <component name>_gth_rx_pcs_cdr_reset.v[hd] ⁽²⁾ <component name>_gth_tx_pcs_cdr_reset.v[hd] ⁽²⁾	Verilog/VHDL wrapper files for the GTX/GTH transceiver

1. For Virtex-6 FPGA GTX transceivers or Virtex-7/Kintex-7 FPGA GTX/GTH transceivers.
2. For Virtex-6 FPGA GTH transceivers.
3. For Virtex-6 FPGA GTH transceivers.

[Back to Top](#)

/example_design/traffic_gen_and_check

The `traffic_gen_and_check` directory contains frame generator and frame checker modules for Aurora 64B/66B core.

Table 7-8: `traffic_gen_and_check` Directory

Name	Description
<component name>/example_design/traffic_gen_and_check	
<component name>_frame_check.v[hd] <component name>_frame_gen.v[hd]	Example design traffic generation and checker files

[Back to Top](#)

<component name>/implement

The `implement` directory contains scripts and support files for both Linux and Windows operating systems. These scripts automate the process of synthesizing and implementing the files needed for the example design.

Table 7-9: **implement Directory**

Name	Description
<component name>/implement	
implement.bat	Windows batch file that processes the example design through the Xilinx tool flow
implement.sh	Linux shell script that processes the example design through the Xilinx tool flow
xst.scr	XST script file for the example design
xst.prj	XST project file for the example design
Chipscope_prj.cpj	ChipScope™ Pro tool project file
planAhead_ise.tcl	PlanAhead™ tool script files for the example design using ISE tools flow
synplify.prj implement_synplify.sh implement_synplify.bat	Synplify Pro script files for Aurora 64B/66B example design

[Back to Top](#)

/implement/results

The `results` directory is created by the `implement` script, after which the `implement` script results are placed in the `results` directory.

 Table 7-10: **results Directory**

Name	Description
<component name>/implement/results	
Implement script result files	

[Back to Top](#)

<component name>/simulation

The `simulation` directory contains the test bench files for the example design.

 Table 7-11: **simulation Directory**

Name	Description
<component name>/simulation	
<component name>_v[hd]	Test bench file for simulating the example design

[Back to Top](#)

/simulation/functional

The `functional` directory contains functional simulation scripts provided with the core.

Table 7-12: **functional Directory**

Name	Description
<component name>/simulation/functional	
<code>simulate_mti.do</code>	ModelSim macro file that compiles the example design sources, the structural simulation model, and the demonstration test bench then runs the functional simulation to completion
<code>wave_mti.do</code>	ModelSim macro file that opens a Wave window
<code>simulate_mti.sh</code>	Linux shell script to invoke ModelSim and run example design
<code>simulate_mti.bat</code>	Windows batch file to invoke ModelSim and run example design
<code>simulate_isim.sh</code>	ISim macro file that compiles the example design sources and the structural simulation model. The demonstration test bench then runs the functional simulation to completion in the Linux operating system.
<code>simulate_isim.bat</code>	ISim macro file that compiles the example design sources and the structural simulation model. The demonstration test bench then runs the functional simulation to completion in the Windows operating system.
<code>wave_isim.tcl</code>	ISim macro file that opens a Wave window with top-level signals.

[Back to Top](#)

/simulation/timing

The `timing` directory contains timing simulation scripts provided with the core.

Table 7-13: **timing Directory**

Name	Description
<component name>/simulation/timing	
<code>simulate_mti.do</code>	ModelSim macro file that compiles the SDF files of the core and the demonstration test bench then runs the timing simulation to completion
<code>wave_mti.do</code>	ModelSim macro file that opens a Wave window

[Back to Top](#)

<component name>/src

The `src` directory contains the source files related to the Aurora example design.

Table 7-14: `src` Directory

Name	Description
<component name>/src	
<component name>_64B66B.v[hd] <component name>_64B66B_descrambler.v[hd] <component name>_64B66B_scrambler.v[hd] <component name>_aurora_lane.v[hd] <component name>_aurora_pkg.vhd (VHDL Only) <component name>_aurora_to_gtx.v[hd] <component name>_block_sync_sm.v[hd] <component name>_cbcc_gtx_6466.v[hd] <component name>_ch_bond_code_gen.v[hd] <component name>_channel_err_detect.v[hd] <component name>_channel_init_sm.v[hd] <component name>_err_detect.v[hd] <component name>_global_logic.v[hd] <component name>_gtx_to_aurora.v[hd] <component name>_lane_init_sm.v[hd] <component name>_rx_ll.v[hd] <component name>_rx_ll_datapath.v[hd] <component name>_sym_dec.v[hd] <component name>_sym_gen.v[hd] <component name>_tx_ll.v[hd] <component name>_tx_ll_control_sm.v[hd] <component name>_tx_ll_datapath.v[hd] <component name>_tx_gearbox.v[hd] <component name>_rx_gearbox.v[hd] <component name>_ll_to_axi.v[hd] <component name>_axi_to_ll.v[hd]	Aurora 64B/66B source files

[Back to Top](#)

Constraining the Core

This chapter is relevant to the ISE® Design Suite.

Device, Package, and Speed Grade Selections

Not Applicable

Clock Frequencies

Aurora 64B/66B example design clock constraints can be grouped into following three categories:

- GT reference clock constraint

The Aurora 64B/66B core uses one minimum reference clock and two maximum reference clocks for the design. The number of GT reference clocks is derived based on transceiver selection (that is, lane assignment in the second page GUI). The GT REFCLK value selected in the first page of the GUI is used to constrain the GT reference clock. TNM_NET of the GT reference clock with TIMESPEC is used to constrain GT reference clocks.

- CORECLK clock constraint

CORECLKs are the clock based on which the core functions. CORECLKS such as USER_CLK and SYNC_CLK are derived out of TXOUTCLK generated by the GT transceiver based on the applied reference clock and the divider settings of the GT transceiver. The Aurora 64B/66B core calculates the USER_CLK/SYNC_CLK frequency based on the line rate and GT interface width. RXRECCLK_32 and RXRECCLK_64 are the received recovered clock constraint derived out of RXRECCLK for capturing the receive data from GT transceiver. TNM_NET of the CORECLKs with TIMESPEC is used to constrain all CORECLKs.

- System clock constraint



RECOMMENDED: *The Aurora 64B/66B example design uses a debounce circuit to sample PMA_INIT asynchronously clocked by the system clock. It is recommended to have the system clock frequency lower than the GT reference clock frequency. TNM_NET on the System clock with TIMESPEC is used to constrain the system clock.*

- GT location constraint

LOC on INST (that is, module that contains GT instantiation) is used to constrain the GT transceiver location. This is provided as either a tool tip or displayed adjacent to lane selection on the second page of the GUI

False Paths

The system clock and user clock are not related to one another. No phase relationship exists between those two clocks. Those two clocks domains need to be set as false paths. TIG command is used to constrain the false paths.

Example Design

The generated example design is a 10.3125 Gb/s line rate, and a 156.25 MHz reference clock. The UCF generated for the XC7K325T-FFG900-2 device follows:

```
# User Clock Constraint: the value is selected based on the line rate of the module
NET "user_clk_i" TNM_NET = "user_clk_i";
TIMESPEC "TS_user_clk_i" = PERIOD "user_clk_i" 161.125 MHz HIGH 50%;
# SYNC Clock Constraint
NET "sync_clk_i" TNM_NET = "sync_clk_i";
TIMESPEC "TS_sync_clk_i" = PERIOD "sync_clk_i" 322.25 MHz HIGH 50%;
# Constraints for RX Recovered clocks
NET "gt_wrapper_i/rxrecclk_to_pll_i" TNM_NET = "rxrecclk_32";
TIMESPEC "TS_rxrecclk_32" = PERIOD "rxrecclk_32" 322.25 MHz HIGH 50%;
# Constraints for Clock Enables
NET "gt_wrapper_i/enable_32_i" TNM_NET = FFS "enable_32";
TIMESPEC "TS_enable_32_multiclk" = FROM "enable_32" to "enable_32" TS_rxrecclk_32/2;

# 156.25MHz GTX Reference clock constraint
NET "GTXQ0_left_i" TNM_NET = "GTXQ0_left_i";
TIMESPEC "TS_GTXQ0_left_i" = PERIOD "GTXQ0_left_i" 156.25 MHz HIGH 50%;

# 50 MHz Board Clock Constraint
NET "INIT_CLK_i" TNM_NET = INIT_CLK;
TIMESPEC TS_INIT_CLK = PERIOD "INIT_CLK" 20 ns HIGH 50%;

NET INIT_CLK_P LOC=C25;
NET INIT_CLK_N LOC=B25;

NET RESET LOC=G19; #BUTTON
NET RESET PULLUP;
NET PMA_INIT LOC=K18; #BUTTON
```

```
NET CHANNEL_UP      LOC=A20;    #LED
NET LANE_UP[0]      LOC=A17;    #LED
NET LANE_UP[1]      LOC=A16;    #LED
NET HARD_ERR        LOC=G17;    #LED
NET SOFT_ERR        LOC=F17;    #LED

##### No cross clock domain analysis. Domains are not related #####
TIMESPEC "TS_TIG1" = FROM "INIT_CLK" TO "user_clk_i" TIG;
NET "gt_wrapper_i/cbcc_gtx0_i/fifo_reset_i" TIG;

##### GT CLOCK Locations #####
# Differential SMA Clock Connection
NET GTXQ0_P LOC=R8;
NET GTXQ0_N LOC=R7;

##### GT LOC #####
INST gt_wrapper_i/gt_multi_gt_i/GTX_INST/gtxe2_i LOC=GTXE2_CHANNEL_X0Y0;
```

Detailed Example Design

This chapter describes the detailed example design that is delivered in the ISE® Design Suite environment.

Directory and File Contents

See [Output Generation in Chapter 7](#) for the directory structure and file contents of the example design.

Quick Start Example Design

The quick start instructions provide a step-by-step procedure for generating an Aurora 64B/66B core, implementing the core in hardware using the accompanying example design, and simulating the core with the provided demonstration test bench (<component name>_tb). For detailed information about the example design provided with the Aurora 64B/66B core, see [Detailed Example Design](#).

The quick start example design consists of these components:

- An instance of the Aurora 64B/66B core generated using the default parameters
 - Full-duplex with a single GTX transceiver
 - AXI4-Stream user interface
- A demonstration test bench to simulate two instances of the example design

Detailed Example Design

Each Aurora 64B/66B core includes an example design (`aurora_example`) that uses the core in a simple data transfer system. For more details about the `example_design` directory, see [Output Generation in Chapter 7](#).

The example design consists of two main components:

- Frame generator ([FRAME_GEN, page 85](#)) connected to the TX interface
- Frame checked ([FRAME_CHECK, page 91](#)) connected to the RX user interface

[Figure 9-1](#) shows a block diagram of the example design for a full-duplex core. [Table 9-1, page 120](#) describes the ports of the example design.

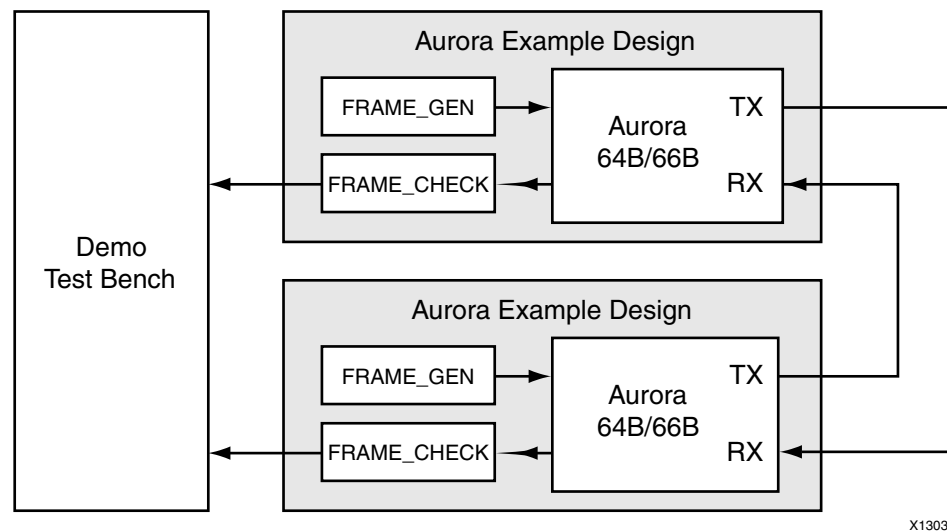


Figure 9-1: Example Design

The example design uses all the interfaces of the core. There are separate AXI4-Stream interfaces for optional flow control. Simplex cores without a TX or RX interface have no `FRAME_GEN` or `FRAME_CHECK` block, respectively. The frame generator produces a random stream of data for cores with a streaming/framing interface.

The scripts provided in the `implement` and `functional` subdirectories can be used to quickly get an Aurora 64B/66B design up and running on a board, or perform a quick simulation of the module. The design can also be used as a reference for connecting the trickier interfaces on the Aurora 64B/66B core, such as the clocking interface.

When using the example design on a board, be sure to edit the `<component name>_example_design.ucf` in the `ucf` subdirectory to supply the correct pins and clock constraints. [Table 9-1](#) describes the ports available in the example design.

Table 9-1: Example Design I/O Ports

Port	Direction	Description
RXN[0:m-1]	Input	Negative differential serial data input pin.
RXP[0:m-1]	Input	Positive differential serial data input pin.
TXN[0:m-1]	Output	Negative differential serial data output pin.
TXP[0:m-1]	Output	Positive differential serial data output pin.
RESET	Input	Reset signal for the example design. The active-High reset is debounced using a USER_CLK signal generated from the reference clock input.
<reference clock(s)>	Input	The reference clocks for the Aurora 64B/66B core are brought to the top level of the example design. See Clock Interface and Clocking, page 30 for details about the reference clocks.
<core error signals>	Output	The error signals from the Aurora 64B/66B core' Status and Control interface are brought to the top level of the example design and registered. See Status, Control, and the Transceiver Interface, page 60 for details.
<core channel up signals>	Output	The channel up status signals for the core are brought to the top level of the example design and registered. See Status, Control, and the Transceiver Interface, page 60 for details.
<core lane up signals>	Output	The lane up status signals for the core are brought to the top level of the example design and registered. Cores have a lane up signal for each GTX/GTH transceiver they use. See Status, Control, and the Transceiver Interface, page 60 for details.
PMA_INIT	Input	The reset signal for the PCS and PMA modules in the GTX/GTH transceivers is connected to the top level through a debouncer. The signal is debounced using the INIT_CLK. See the Reset section in the <i>Virtex-6 FPGA GTX Transceivers User Guide</i> , the <i>Virtex-6 FPGA GTH Transceivers User Guide</i> , or the <i>7 Series FPGAs GTX/GTH Transceivers User Guide</i> for further details on GT RESET.
INIT_CLK	Input	INIT_CLK is used to register and debounce the PMA_INIT signal. INIT_CLK must not come from a GTX/GTH transceiver, and should be set to a slow rate, preferably slower than the reference clock.
DATA_ERR_COUNT[0:7]	Output	Count of the number of frame data words received by the FRAME_CHECK that did not match the expected value.
UFC_ERR	Output	Asserted (active-High) when UFC data words received by the FRAME_CHECK that did not match the expected value.
USER_K_ERR	Output	Asserted (active-High) when User K data words received by the FRAME_CHECK that did not match the expected value.

FRAME_GEN and FRAME CHECK

See [FRAME_GEN](#) and [FRAME_CHECK](#) in [Chapter 6, Detailed Example Design](#).

Generating the Core

To generate an Aurora 64B/66B core with default values using the CORE Generator™ tool:

1. Start the CORE Generator tool from a required directory.

For help starting and using the CORE Generator tool, see CORE Generator Help in the ISE [tool documentation](#).

2. Choose **File > New Project**.
3. Type a project name.
4. To set project options:

On the Part tab, for Family select **Virtex6**. For Device, select an appropriate device that supports GTX transceivers, such as **xc6v1x240t**.

Note: If an unsupported silicon family is selected, the Aurora 64B/66B appears light gray in the taxonomy tree and cannot be customized. Only devices containing GTX/GTH transceivers are supported by the core. For a list of supported architectures, see [IP Facts](#).

No further project options need to be set.

Optionally, on the Generation tab, set the Design Entry pull-down to **Verilog**.

5. After creating the project, locate the Aurora 64B/66B core v7.3 in the taxonomy tree under:

```
/Communication_&_Networking/Serial_Interfaces
```

6. Double-click the core for generation.

The customization screens are shown in [Figure 9-2](#) and [Figure 9-3](#).

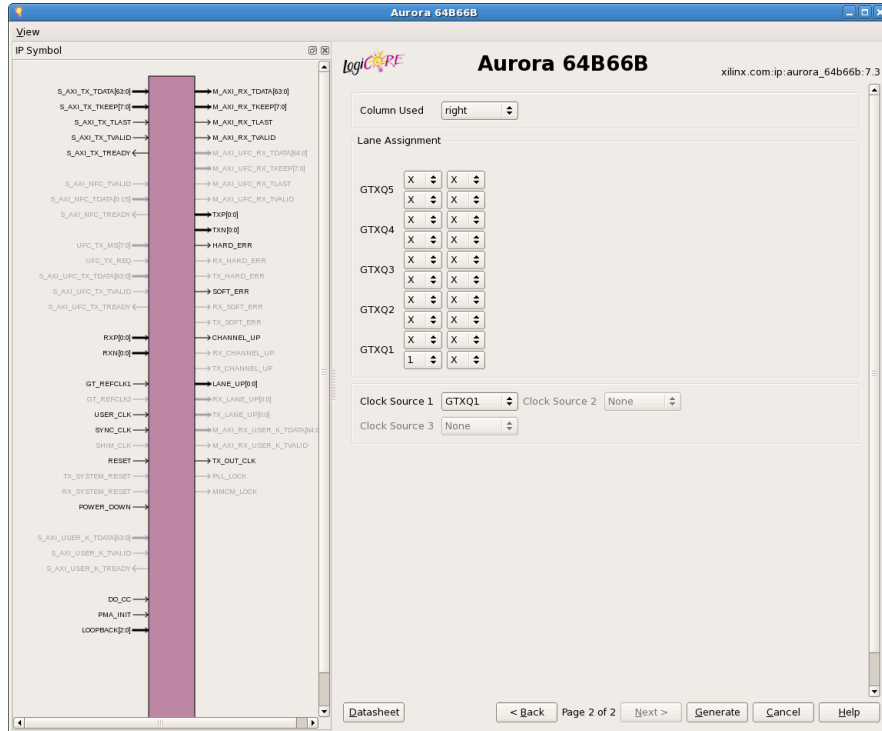


Figure 9-2: CORE Generator Tool Aurora 64B/66B Customization Screen - Page 1

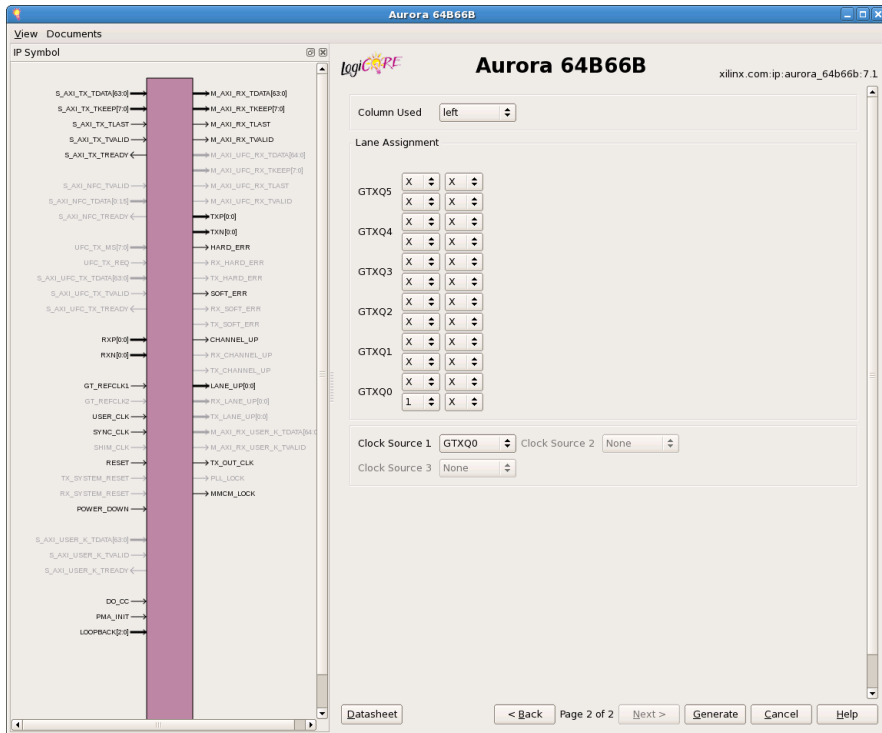


Figure 9-3: CORE Generator Tool Aurora 64B/66B Customization Screen - Page 2

7. In the Component Name field, enter a name for the core instance. This example uses the name **aurora_64b66b_v7_3**.
8. Click **Generate**.

The core and its supporting files, including the example design, are generated in the project directory. For detailed information about the example design files and directories, see [Output Generation in Chapter 7](#).

Simulating the Example Design

The Aurora 64B/66B core provides a quick way to simulate and observe the behavior of the core using the provided example design. Prior to simulating the core, the functional (gate-level) simulation models must be generated. You must compile all source files in the following directories to a single library as shown in [Table 9-2](#). See the *Synthesis and Simulation Design Guide* ([UG626](#)) for the ISE v14.4 tool for instructions for compiling ISE tool simulation libraries.

Table 9-2: Required Simulation Libraries

HDL	Library	Source Directories
Verilog	UNISIMS_VER	<Xilinx dir>/verilog/src/unisims <Xilinx dir>/secureip/<SIMULATOR>
VHDL	UNISIM	<Xilinx dir>/vhdl/src/unisims <Xilinx dir>/secureip/<SIMULATOR>

Notes:

1. SIMULATOR can be ModelSim.

The Aurora 64B/66B core provides a command line script to simulate the example design. To run a VHDL or Verilog ModelSim simulation of the Aurora 64B/66B core, use these instructions:

1. Launch the ModelSim simulator and set the current directory to:

```
<project directory>/aurora_64b66b_v7_3/simulation/functional
```

2. Set the MTI_LIBS variable:

```
modelsim> setenv MTI_LIBS <path to compiled libraries>
```

3. Launch the simulation script:

```
modelsim> do simulate_mti.do
```

The ModelSim script compiles the example design and test bench, and adds the relevant signals to the wave window. After the design is compiled and the wave window is displayed, run the simulation to see the Aurora 64B/66B core power up, followed by Aurora 64B/66B channel initialization and data transfer. Data transfer begins after the `CHANNEL_UP` signal goes High.

Simplex cores need to be generated one after the other. In addition, simulating simplex cores requires additional steps. To simulate a simplex TX or simplex RX core, perform the following steps:

1. Generate the simplex core.
2. Generate a complementary simplex core.
3. Go to the simulation directory of the first core generated.
4. Set the environment variable `SIMPLEX_PARTNER` to point to the directory of the complementary core.
5. Run the script as explained previously.

Note: The top-level module name of the simplex design and simplex partner design should be similar. For example, if the top-level module name of the TX simplex design is `aurora_64b66b_simplex`, the top-level module name of the simplex partner should be `rx_aurora_64b66b_simplex`.

Implementing the Example Design

After the core is generated, the design can be processed by the Xilinx implementation tools. The generated output files include several scripts to assist you in running the Xilinx tools.

From the command prompt, navigate to the project directory and type the following:

For Windows

```
ms-dos> cd aurora_64b66b_v7_3\implement
```

```
ms-dos> .\implement.bat
```

For Linux

```
% cd aurora_64b66b_v7_3/implement
```

```
% ./implement.sh
```

These commands execute a script that synthesizes, translates, maps, place-and-routes the example design and produces a bitmap file. The resulting files are placed in the results directory created within the `implement` directory.

SECTION IV: APPENDICES

Verification, Compliance, and Interoperability

Migrating

Debugging

Generating a GT Wrapper File from the
Transceiver Wizard

Additional Resources

Verification, Compliance, and Interoperability

Aurora 64B/66B cores are verified for protocol compliance using an array of automated hardware and simulation tests. The core comes with an example design implemented using a linear feedback shift register (LFSR) for understanding and verification of the core features.

The Aurora 64B/66B core is verified using the Aurora 64B/66B Bus Functional Model (BFM) and proprietary custom test benches. The Aurora 64B/66B BFM verifies the protocol compliance along with interface level checks and error scenarios. An automated test system runs a series of simulation tests on the most widely used set of design configurations chosen at random. Aurora 64B/66B cores are also tested in hardware for functionality, performance, and reliability using Xilinx GTX transceiver demonstration boards. Aurora verification test suites for all possible modules are continuously being updated to increase test coverage across the range of possible parameters for each individual module.

Two boards can be used for verification:

- ML623
- KC724

Migrating

Introduction

This appendix explains about migrating legacy (LocalLink based) Aurora cores to the AXI4-Stream Aurora core.

For information on migrating to the Vivado™ Design Suite, see *Vivado Design Suite Migration Methodology Guide* ([UG911](#)).

Prerequisites

- ISE® 14.4/Vivado 2012.4 tool build containing the Aurora 64B/66B v7.3 core supporting the AXI4-Stream protocol
- Familiarity with the Aurora directory structure
- Familiarity with running the Aurora example design
- Basic knowledge of the AXI4-Stream and LocalLink protocols
- Latest product guide (PG074) of the core with the AXI4-Stream updates
- Legacy data sheet (DS528), getting started guide (UG238), and user guide (UG237) for reference
- Migration guide (this Appendix)

Overview of Major Changes

The major change to the core is the addition of the AXI4-Stream interface:

- The user interface is modified from the legacy LocalLink (LL) to AXI4-Stream
- All AXI4-Stream signals are active-High, whereas LocalLink signals are active-Low
- The user interface in the example design and design top file is AXI4-Stream
- A new shim module is introduced in the AXI4-Stream Aurora core to convert AXI4-Stream signals to LL and LL back to AXI4-Stream
 - The AXI4-Stream to LL shim on the transmit converts all AXI4-Stream signals to LL
 - The shim deals with active-High to active-Low conversion of signals between AXI4-Stream and LocalLink
 - Generation of SOF_N and REM bits mapping are handled by the shim
 - The LL to AXI4-Stream shim on the receive converts all LL signals to AXI4-Stream
- Each interface (PDU, UFC, and NFC) has a separate AXI4-Stream to LL and LL to AXI4-Stream shim instantiated from the design top file
- Frame generator and checker have respective LL to AXI4-Stream and AXI4-Stream to LL shim instantiated in the Aurora example design to interface with the generated AXI4-Stream design

Block Diagrams

Figure B-1 shows an example Aurora design using the legacy LocalLink interface. Figure B-2 shows an example Aurora design using the AXI4-Stream interface.

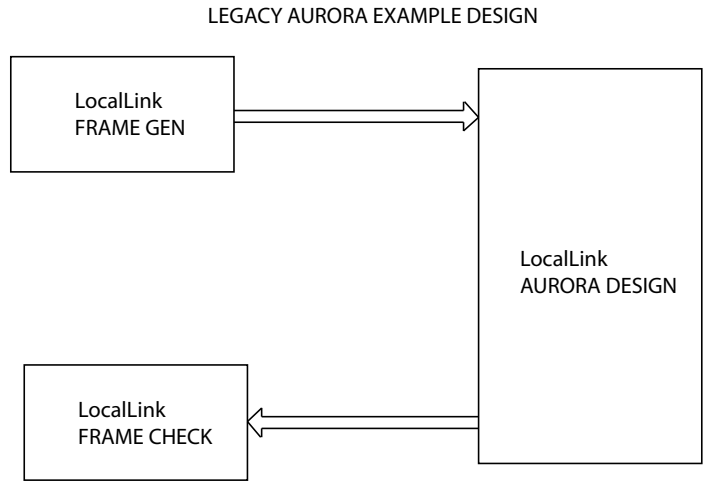


Figure B-1: Legacy Aurora Example Design

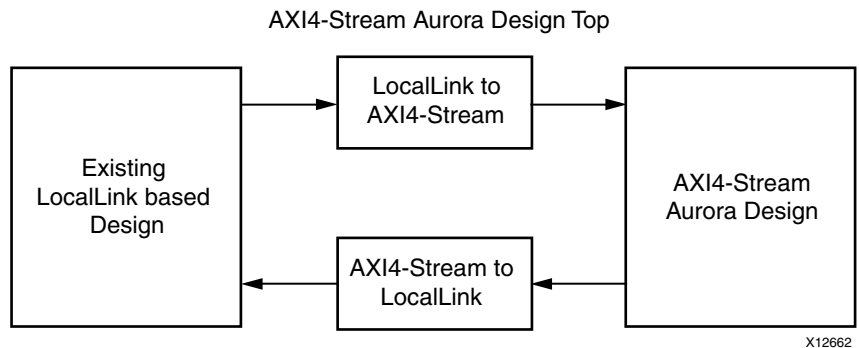


Figure B-2: AXI4-Stream Aurora Example Design

Signal Changes

Table B-1: Interface Changes

LocalLink Name	AXI4-S Name	Difference
TX_D	S_AXI_TX_TDATA	Name change only
TX_REM	S_AXI_TX_TKEEP	Name change. For functional differences, see Table 2-7, page 19
TX_SOF_N		Generated Internally
TX_EOF_N	S_AXI_TX_TLAST	Name change; Polarity
TX_SRC_RDY_N	S_AXI_TX_TVALID	Name change; Polarity
TX_DST_RDY_N	S_AXI_TX_TREADY	Name change; Polarity
UFC_TX_REQ_N	UFC_TX_REQ	Name change; Polarity
UFC_TX_MS	UFC_TX_MS	No Change
UFC_TX_D	S_AXI_UFC_TX_TDATA	Name change only
UFC_TX_SRC_RDY_N	S_AXI_UFC_TX_TVALID	Name change; Polarity
UFC_TX_DST_RDY_N	S_AXI_UFC_TX_TREADY	Name change; Polarity
NFC_TX_REQ_N	S_AXI_NFC_TX_TVALID	Name change; Polarity
NFC_TX_ACK_N	S_AXI_NFC_TX_TREADY	Name change; Polarity
NFC_PAUSE	S_AXI_NFC_TX_TDATA	Name change. For signal mapping, see Table 2-11, page 22
NFC_XOFF		
USER_K_DATA	S_AXI_USER_K_TDATA	Name change. For signal mapping, see Table 2-12, page 23
USER_K_BLK_NO		
USER_K_TX_SRC_RDY_N	S_AXI_USER_K_TX_TVALID	Name change; Polarity
USER_K_TX_DST_RDY_N	S_AXI_USER_K_TX_TREADY	Name change; Polarity
RX_D	M_AXI_RX_TDATA	Name change only
RX_REM	M_AXI_RX_TKEEP	Name change. For functional difference, see Table 2-7, page 19
RX_SOF_N		Removed
RX_EOF_N	M_AXI_RX_TLAST	Name change; Polarity
RX_SRC_RDY_N	M_AXI_RX_TVALID	Name change; Polarity
UFC_RX_DATA	M_AXI_UFC_RX_TDATA	Name change only
UFC_RX_REM	M_AXI_UFC_RX_TKEEP	Name change For functional difference, see Table 2-10, page 20
UFC_RX_SOF_N		Removed
UFC_RX_EOF_N	M_AXI_UFC_RX_TLAST	Name change; Polarity
UFC_RX_SRC_RDY_N	M_AXI_UFC_RX_TVALID	Name change; Polarity

Table B-1: Interface Changes (Cont'd)

LocalLink Name	AXI4-S Name	Difference
RX_USER_K_DATA	M_AXI_USER_K_RX_TDATA	Name change
RX_USER_K_BLK_NO		For functional difference, see Table 2-12, page 23
RX_USER_K_SRC_RDY_N	M_AXI_USER_K_RX_TVALID	Name change; Polarity

Migration Steps

Generate an AXI4-Stream Aurora core from the CORE Generator™ tool using the ISE v14.4 tool or Vivado v2012.4 design tools.

Simulate the Core

1. Run the `vsim -do simulate_mti.do` file from the `/simulation/functional` directory.
2. ModelSim GUI launches and compiles the modules.
3. The `wave_mti.do` file loads automatically and populates AXI4-Stream signals.
4. Allow the simulation to run. This might take some time.
 - a. Initially lane up is asserted.
 - b. Channel up is then asserted and the data transfer begins.
 - c. Data transfer from all flow control interfaces now begins.
 - d. Frame checker continuously checks the received data and reports for any data mismatch.
5. A 'TEST PASS' or 'TEST FAIL' status is printed on the ModelSim console providing the status of the test.

Implement the Core

1. Run `./implement.sh` (for Linux) from the `/implement` directory.
2. The `implement` script compiles the core and runs through the ISE tool and generates a bit file and netlist for the core.

Integrate to an Existing LocalLink-based Aurora Design

1. The Aurora core provides a lightweight 'shim' to interface to any existing LL based interface. The shims are delivered along with the core from the `aurora_64b66b_v7_3` version of the core.
2. See [Figure B-2, page 129](#) for the emulation of a LL Aurora core from a AXI4-Stream Aurora core.
3. Two shims `<user_component_name>_ll_to_axi.v[hd]` and `<user_component_name>_axi_to_ll.v[hd]` are provided in the 'src' directory of the AXI4-Stream Aurora core.
4. Instantiate both the shims along with `<user_component_name>.v[hd]` in the existing LL based design top.
5. Connect the shim and AXI4-Stream Aurora design as shown in [Figure B-2, page 129](#).
6. The latest AXI4-Stream Aurora core can be plugged into any existing LL design environment.

GUI Changes

Figure B-3 shows the AXI4-Stream signals in the IP Symbol diagram.

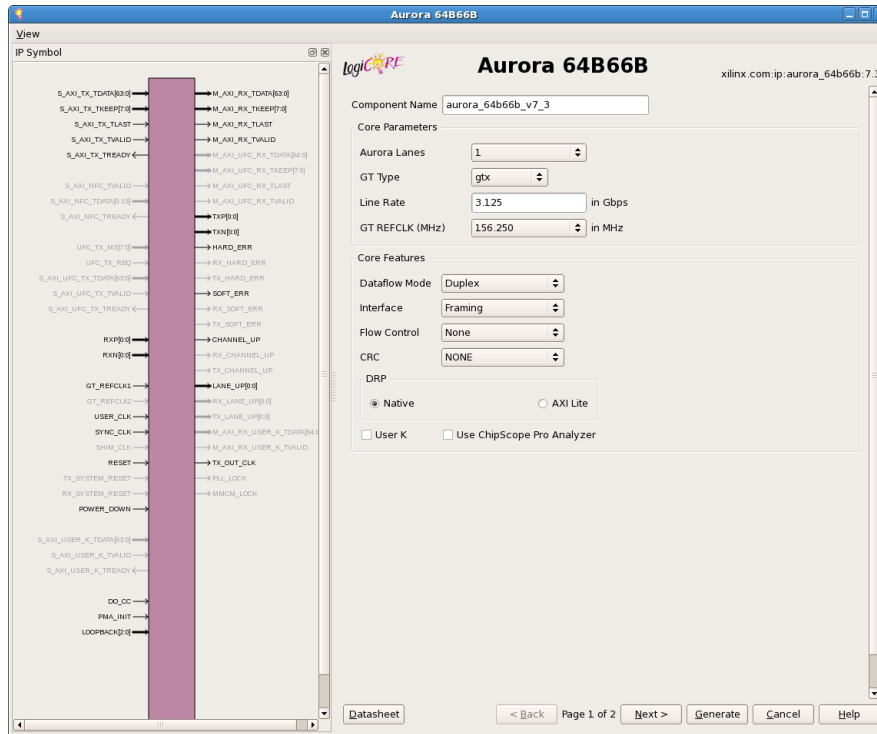


Figure B-3: AXI4-Stream Signals

Limitations

This section outlines the limitations of the Aurora 64B/66B core for AXI4-Stream support.



IMPORTANT: *Be aware of the following limitations while interfacing the Aurora 64B/66B core with the AXI4-Stream compliant interface core.*

Limitation 1:

The AXI4-Stream specification supports four types of data stream:

- Byte stream
- Continuous aligned stream
- Continuous unaligned stream
- Sparse stream

The Aurora 64B/66B core supports only continuous aligned stream and continuous unaligned stream. The position bytes are valid only at the end of packet.

Limitation 2:

The AXI4-Stream protocol supports transfer with zero data at the end of packet, but the Aurora 64B/66B core expects at least one byte should be valid at the end of packet.

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools. In addition, this appendix provides a step-by-step debugging process and a flow diagram to guide you through debugging the Aurora 64B/66B core.

The following topics are included in this appendix:

- [Finding Help on Xilinx.com](#)
- [Debug Tools](#)
- [Simulation Debug](#)
- [General Checks](#)
- [Interface Debug](#)

Finding Help on Xilinx.com

To help in the design and debug process when using the Aurora 64B/66B core, the [Xilinx Support web page](#) (www.xilinx.com/support) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for opening a Technical Support WebCase. Also see the [Aurora home page](#).

Documentation

This product guide is the main document associated with the Aurora 64B/66B core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page (www.xilinx.com/support) or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Design Tools tab on the Downloads page (www.xilinx.com/download). For more information about this tool and the features available, open the online help after installation.

Release Notes

Known issues for all cores, including the Aurora 64B/66B core are described in the [IP Release Notes Guide \(XTP025\)](#).

Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Known Issues

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core are listed below, and can also be located by using the Search Support box on the main [Xilinx support web page](#). To maximize your search results, use proper keywords such as

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Answer Records for the Aurora 64B/66B Core

- [42552](#) -- Aurora 64b/66b Issues and Answer Record List
- [52313](#) -- Release Notes and Known Issues

Xilinx provides premier technical support for customers encountering issues that require additional assistance.

To contact Xilinx Technical Support:

1. Navigate to www.xilinx.com/support.
2. Open a WebCase by selecting the [WebCase](#) link located under Support Quick Links.

When opening a WebCase, include:

- Target FPGA including package and speed grade.
- All applicable Xilinx Design Tools and simulator software versions.

- Additional files based on the specific issue might also be required. See the relevant sections in this debug guide for guidelines about which file(s) to include with the WebCase.

Debug Tools

There are many tools available to address Aurora 64B/66B core design issues. It is important to know which tools are useful for debugging various situations.

ChipScope Pro Tool

The ChipScope™ Pro debugging tool inserts logic analyzer, bus analyzer, and virtual I/O cores directly into your design. The ChipScope Pro debugging tool allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed through the ChipScope Pro logic analyzer tool. For detailed information for using the ChipScope Pro debugging tool, see www.xilinx.com/tools/cspro.htm.

Reference Boards

Various Xilinx development boards support the Aurora 64B/66B core. These boards can be used to prototype designs and establish that the core can communicate with the system.

- 7 series FPGA evaluation boards
 - KC705
 - KC724

Simulation Debug

Lanes and Channel do not come up in simulation

- The quickest way to debug these problems is to view the signals from one of the GTX/GTH instances that are not working.
- Make sure that the reference clock and user clocks are all toggling.
Note: Only one of the reference clocks should be toggling, The rest will be tied low.
- Check to see that RECCLK and TXOUTCLK are toggling. If they are not toggling, you might have to wait longer for the PMA to finish locking. You should typically wait about 6-9 microseconds for lane up and channel up. You might need to wait longer for simplex/ 7 series designs.

- Make sure that TXN and TXP are toggling. If they are not, make sure you have waited long enough (see the previous bullet) and make sure you are not driving the TX signal with another signal.
- Check the PLL/MMCM_NOT_LOCKED signal and the RESET signals on your design. If these are being held active, your Aurora module will not be able to initialize.
- Be sure you do not have the POWER_DOWN signal asserted
- Make sure the TXN and TXP signals from each GTX/GTH are connected to the appropriate RXN and RXP signals from the corresponding GTX/GTH on the other side of the channel
- If you are simulating Verilog, you will need to instantiate the "glbl" module and use it to drive the power_up reset at the beginning of the simulation to simulate the reset that occurs after configuration. You should hold this reset for a few cycles. The following code can be used as an example:

```
//Simulate the global reset that occurs after configuration at the beginning
//of the simulation.
assign glbl.GSR = gsr_r;
assign glbl.GTS = gts_r;

initial
  begin
    gts_r = 1'b0;
    gsr_r = 1'b1;
    #(16*CLOCKPERIOD_1);
    gsr_r = 1'b0;
  end
```

- If you are using a multilane channel, make sure all the GTs on each side of the channel are connected in the correct order.

Channel comes up in simulation but S_AXI_TX_TREADY is never asserted (never goes high)

- If your module includes flow control but you are not using it, make sure the request signals are not currently driven high. S_AXI_NFC_TX_TVALID and UFC_TX_REQ are active-High: if they are high, S_AXI_TX_TREADY will stay low because the channel will be allocated for flow control.
- Make sure DO_CC is not being driven high continuously. Whenever DO_CC is high on a positive clock edge, the channel is used to send clock correction characters, so S_AXI_TX_TREADY is deasserted.
- If your module includes USER K Blocks but you are not using it, make sure the S_AXI_USER_K_TX_TVALID is not driven high. If it is high, S_AXI_TX_TREADY will stay low as channel will be allocated for USER K Blocks.
- If you have NFC enabled, make sure the design on the other side of the channel did not send an NFC XOFF message. This will cut off the channel for normal data until the other side sends an NFC XON message to turn the flow on again. See ug775.pdf for more details.

Bytes and words are being lost as they travel through the Aurora channel

- If you are using the AXI4-Stream interface, make sure you are writing data correctly. The most common mistake is to assume words are written without looking at S_AXI_TX_TREADY. Also remember that the S_AXI_TX_TKEEP signal must be used to indicate which bytes are valid when S_AXI_TX_TLAST is asserted.
- Make sure you are reading correctly from the RX interface. Data and framing signals are only valid when M_AXI_RX_TVALID is asserted.

Problems while compiling the design

- Make sure you include all the files from the src directory when compiling.
- If you are using VHDL, make sure to include the `aurora_pkg.vhd` file in your synthesis.

General Checks

Ensure that all the timing constraints for the core were properly incorporated from the example design and that all constraints were met during implementation.

- Does it work in post-place and route timing simulation? If problems are seen in hardware but not in timing simulation, this could indicate a PCB issue. Ensure that all clock sources are active and clean.
- If using MMCMs in the design, ensure that all MMCMs have obtained lock by monitoring the `LOCKED` port.
- If your outputs go to 0, check your licensing.

Interface Debug

AXI4-Stream Interfaces

If data is not being transmitted or received, check the following conditions:

- If transmit `s_axi_tx_tready` is stuck low following the `s_axi_tx_tvalid` input being asserted, the core cannot send data.
- If the receive `s_axi_tx_tvalid` is stuck low, the core is not receiving data.
- Check that the `USER_CLK` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed. See [Figure 3-12](#).
- Check core configuration.
- Add appropriate core specific checks.

Generating a GT Wrapper File from the Transceiver Wizard

The transceiver attributes play a vital role in the functionality of the Aurora 64B/66B core. Use the latest transceiver wizard to generate the transceiver wrapper file.



RECOMMENDED: *Xilinx strongly recommends that you update the transceiver wrapper file in the Design Suite tool releases when the transceiver wizard has been updated but the Aurora core has not.*

This appendix provides instructions to generate these transceiver wrapper files:

- [Case 1: Virtex-7/Kintex-7 FPGA Wrapper Compatibility](#)
- [Case 2: Virtex-6 GTX FPGA Wrapper Compatibility](#)
- [Case 3: Virtex-6 GTH FPGA Wrapper Compatibility](#)

Case 1: Virtex-7/Kintex-7 FPGA Wrapper Compatibility

Use these steps to generate the transceiver wrapper file using the 7 series FPGAs Transceivers Wizard:

1. Using the IP catalog, run the latest version of the 7 series FPGAs Transceivers Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 64B/66B core.
2. Select the protocol template: Aurora 64B/66B
3. Change the Line Rate in both TX and RX based on the application requirement.
4. Select the Reference Clock from the drop-down box menu in both TX and RX based on the application requirement.
5. Select transceiver(s) and the clock source(s) based on the application requirement.
6. On Page 3, select External Data Width of RX to be 32 Bits and Internal Data Width to be 16 Bits
7. Keep all other settings as default.

8. Generate the core.
9. Replace the `<component name>_gtx.v[hd]` file in the `example_design/gt/` directory available in the Aurora 64B/66B core with the generated `<component name>_gt.v[hd]` file generated from the 7 series FPGAs Transceivers Wizard.

The transceiver settings for the Aurora 64B/66B core are up to date now.

Case 2: Virtex-6 GTX FPGA Wrapper Compatibility

Use these steps to generate the transceiver wrapper file using the Virtex®-6 FPGAs Transceivers Wizard:

1. Using the ISE® CORE Generator™ IP catalog, run the latest version of the Virtex-6 GTX FPGAs Transceivers Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 64B/66B core.
2. Select the protocol template: Aurora 64B/66B
3. Change the Line Rate in both TX and RX based on the application requirement.
4. Select the Reference Clock from the drop-down box menu in both TX and RX based on the application requirement.
5. Select Data Path Width under RX to 16. Ensure Decoding is set to 64B/66B
6. Select transceiver(s) and the clock source(s) based on the application requirement.
7. Keep all other settings as default.
8. Generate the core.
9. Replace the `<component name>_gtx.v[hd]` file in the `example_design/gt/` directory available in the Aurora 64B/66B core with the generated `<component name>_gt.v[hd]` file generated from the Virtex-6 GTX FPGAs Transceivers Wizard.

The transceiver settings for the Aurora 64B/66B core are up to date now.

Case 3: Virtex-6 GTH FPGA Wrapper Compatibility

Use these steps to generate the transceiver wrapper file using the Virtex-6 GTH FPGAs Transceivers Wizard:

1. Using the CORE Generator IP catalog, run the latest version of the Virtex-6 GTH FPGAs Transceivers Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 64B/66B core.
2. Select the protocol template: Aurora 64B/66B
3. Select the Reference Clock from the drop-down box menu in both TX and RX based on the application requirement.
4. Select transceiver(s) and the clock source(s) based on the application requirement
5. Go to page 2 and select the Line Rate based on the application requirement. GTH0 Line Rate can be set to $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ of the base line rate.
6. Keep all other settings as default.
7. Generate the core.
8. Compare and update the `<component name>_quad.v[hd]` file in the `example_design/gt/` directory available in the Aurora 64B/66B core with the generated `<component name>_quad.v[hd]` file generated from the Virtex-6 GTH FPGAs Transceivers Wizard.

The transceiver settings for the Aurora 64B/66B core are up to date now.

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

www.xilinx.com/company/terms.htm.

References

Unless otherwise noted, IP references are for the product documentation page. These documents provide supplemental material useful with this product guide:

1. Xilinx Aurora website: www.xilinx.com/aurora
Aurora 64B/66B Protocol Specification ([SP011](#))
2. [AMBA AXI4-Stream Protocol Specification](#)
3. Vivado™ Design Suite documentation: www.xilinx.com/cgi-bin/docs/rdoc?v=2012.4;t=vivado+userguides
4. These Xilinx documents can be located from the [Xilinx Support website](#):
 - *AXI Reference Guide* ([UG761](#))
 - *Vivado Design Suite Migration Methodology Guide* ([UG911](#))
 - *7 Series FPGAs GTX/GTH Transceivers User Guide* ([UG476](#))
 - *7 Series FPGAs Overview* ([DS180](#))
 - *Virtex-7 FPGAs Data Sheet: DC and Switching Characteristics* ([DS183](#))
 - *Kintex-7 FPGAs Data Sheet: DC and Switching Characteristics* ([DS182](#))
 - *Virtex-6 Family Overview* ([DS150](#))

- *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics* ([DS152](#))
 - *Virtex-6 FPGA GTX Transceivers User Guide* ([UG366](#))
 - *Virtex-6 FPGA GTH Transceivers User Guide* ([UG371](#))
5. *Synthesis and Simulation Design Guide* ([UG626](#))

Technical Support

Xilinx provides technical support at www.xilinx.com/support for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled DO NOT MODIFY.

See the IP Release Notes Guide ([XTP025](#)) for more information on this core. For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Resolved Issues
- Known Issues

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/16/12	1.0	Initial Xilinx release as a product guide. This document replaces UG775, <i>LogiCORE IP Aurora 64B/66B User Guide</i> and DS815, <i>LogiCORE IP Aurora 64B/66B Data Sheet</i> . <ul style="list-style-type: none"> • Added section explaining constraining of the core. • Added section explaining core debugging.
12/18/12	1.0.1	Updated for 14.4 and 2012.4 release. Added TKEEP description Updated Debugging appendix.

Notice of Disclaimer

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at <http://www.xilinx.com/warranty.htm>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: <http://www.xilinx.com/warranty.htm#critapps>.

© Copyright 2012 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, and PrimeCell are trademarks of ARM in the EU and other countries. All other trademarks are the property of their respective owners.