LogiCORE IP Aurora 8B/10B v8.2

Product Guide

PG046 July 25, 2012

XILINX®

Table of Contents

Introduction

The LogiCORE™ IP Aurora 8B/10B core supports the AMBA® protocol AXI4-Stream user interface. The core implements the Aurora 8B/10B protocol using the high-speed serial transceivers on the Virtex®-7 and Kintex™-7 families (including the -2L lower power devices), Virtex-6 LXT, SXT, CXT, HXT, and lower power families, and the Spartan®-6 LXT family.

The Aurora 8B/10B core is a scalable, lightweight, link-layer protocol for high-speed serial communication. The protocol is open and can be implemented using Xilinx® FPGA technology. The protocol is typically used in applications requiring simple, low-cost, high-rate, data channels.

The Vivado™ Design Suite tool produces source code for Aurora 8B/10B cores with variable datapath width. The cores can be simplex or full-duplex, and feature one of two simple user interfaces and optional flow control.

Features

- General-purpose data channels with throughput range from 480 Mb/s to 84.48 Gb/s
- Supports up to any 16 of 56 Virtex-7/Kintex-7 FPGA GTX/GTH transceivers, 16 of 36 Virtex-6 FPGA GTX transceivers or 4 of 8 Spartan-6 FPGA GTP transceivers
- Aurora 8B/10B protocol specification v2.2 compliant
- Low resource cost (see Resource Utilization, page 10)
- Easy-to-use framing and flow control
- Automatically initializes and maintains the channel
- Full-duplex or simplex operation
- AXI4-Stream (framing) or streaming user interface
- 16-bit additive scrambler/descrambler
- 16-bit or 32-bit CRC for user data
- Hot-plug logic

LogiCORE IP Facts Table					
Core Specifics					
Supported Device Family[1][2]	Virtex-7, Kintex-7, Virtex-6, Spartan-6				
Supported User Interfaces	AXI4-Stream				
Resources[3]	**LUTs**	**FFs**	**DSP Slices**	**Block RAMs**	**Max. Frequency[4]**
Config1	342	463	0	0	330 MHz
Provided with Core					
Design Files	Verilog and VHDL				
Example Design	Verilog and VHDL				
Test Bench	Verilog and VHDL				
Constraints File	XDC				
Simulation Model	Not Provided				
Supported S/W Driver	N/A				
Tested Design Flows[5]					
Design Entry	Vivado™ Design Suite[6]				
Simulation	Vivado Simulator				
Synthesis	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. This core release supports only 7 series devices. Spartan-6 and Virtex-6 families are listed for backward compatibility with previous core releases. GTP references are associated with GTP transceivers in Spartan-6 devices.
3. For device performance numbers, see Table 2-1 through Table 2-12.
4. For more complete performance data, see Performance, page 9.
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 Kintex™-7 and Virtex™-7 devices. Spartan®-6 and Virtex-6 families are listed for backward compatibility with previous core releases.

This product guide describes how to generate a LogiCORE™ IP Aurora 8B/10B core using Virtex-7/Kintex-7 FPGA GTX/GTH transceivers, Virtex-6 FPGA GTX transceivers, and Spartan-6 FPGA GTP transceivers. The core implements the Aurora 8B/10B protocol using the high-speed serial transceivers on the Virtex-7/Kintex-7 families (including lower power), Virtex-6 LXT, SXT, CXT, HXT families (including lower power) and the Spartan-6 LXT family. The LogiCORE IP Aurora 8B/10B v8.2 core supports the AMBA® protocol AXI4-Stream user interface.

The Aurora 8B/10B core is a lightweight, serial communications protocol for multi-gigabit links. It is used to transfer data between devices using one or many transceivers. Connections can be *full-duplex* (data in both directions) or *simplex* (Figure 1-1).

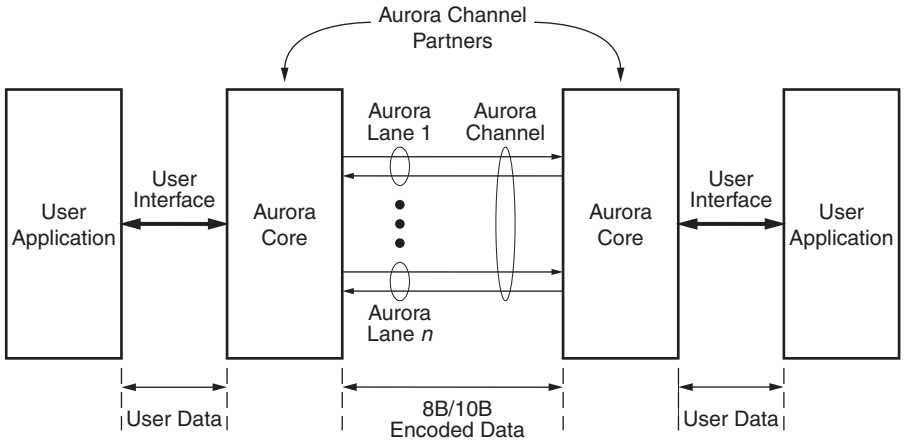

Figure 1-1: **Aurora 8B/10B Channel Overview**

Aurora 8B/10B cores automatically initialize a channel when they are connected to an Aurora channel partner. After initialization, applications can pass data freely across the channel as *frames* or *streams* of data. Aurora *frames* can be any size, and can be interrupted at any time. Gaps between valid data bytes are automatically filled with *idles* to maintain lock and prevent excessive electromagnetic interference. *Flow control* is optional in Aurora. It can be used to reduce the rate of incoming data or to send brief, high-priority messages through the channel.

Streams are implemented in the Aurora 8B/10B core as a single, unending frame. Whenever data is not being transmitted, idles are transmitted to keep the link alive. The Aurora 8B/10B core detects single-bit and most multi-bit errors using 8B/10B coding rules. Excessive bit errors, disconnections, or equipment failures cause the core to reset and attempt to re-initialize a new channel.

Although the Aurora 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, experience building high-performance, pipelined FPGA designs using Xilinx implementation tools and Xilinx Design Constraints files (XDC) with the Vivado Design Suite is recommended. Read Status, Control, and the Transceiver Interface, page 56, carefully.

Consult the PCB design requirements information in:

- *7 Series FPGAs GTX/GTH Transceivers User Guide* [Ref 3]
- *Virtex-6 FPGA GTX Transceivers User Guide* [Ref 4]
- *Spartan-6 FPGA GTP Transceivers User Guide* [Ref 5]

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

Feature Summary

Aurora 8B/10B is a scalable, lightweight, link-layer protocol for high-speed serial communication. The LogiCORE IP Aurora 8B/10B core provides a user interface from which designers can develop serial links. The core performs data transfers between devices using Xilinx GTX, GTP, and GTH transceivers. Up to 16 transceivers can be implemented, running at any supported line rate. The throughput is scalable from 480 Mb/s to over 84.48 Gb/s. Data channels can be full-duplex or simplex.

The Aurora 8B/10B core is compliant with the *Aurora 8B/10B Specification v2.2*. It is delivered as Verilog or VHDL source code.

Applications

Aurora 8B/10B 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 8B/10B 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 core provides the logic needed to use GTP/GTX/GTH transceivers, with minimal FPGA resource cost.

- **Board-to-board and backplane links**: The Aurora 8B/10B core uses standard 8B/10B encoding, making it compatible with many existing hardware standards for cables and backplanes. Aurora 8B/10B cores 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 protocol provides several ways to perform unidirectional channel initialization, making it possible to use the GTP/GTX/GTH transceivers when a back channel is not available, and to reduce costs due to unused full-duplex resources.
- **ASIC applications**: The Aurora protocol 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 protocol leads to low resource costs in ASICs as well as in FPGAs, and design resources like the Aurora bus functional model (ABFM 8B/10B) with compliance testing make it easy to get an Aurora channel up and running.

  Note: Contact Xilinx Sales or Auroramkt@xilinx.com for information on licensing the Aurora 8B/10B core for ASIC applications.

Licensing and Ordering Information

This Xilinx® LogiCORE IP module is provided at no additional cost with the Xilinx Vivado™ 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 8B/10B 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.

Product Specification

Figure 2-1 shows a block diagram of the implementation of the Aurora 8B/10B core.

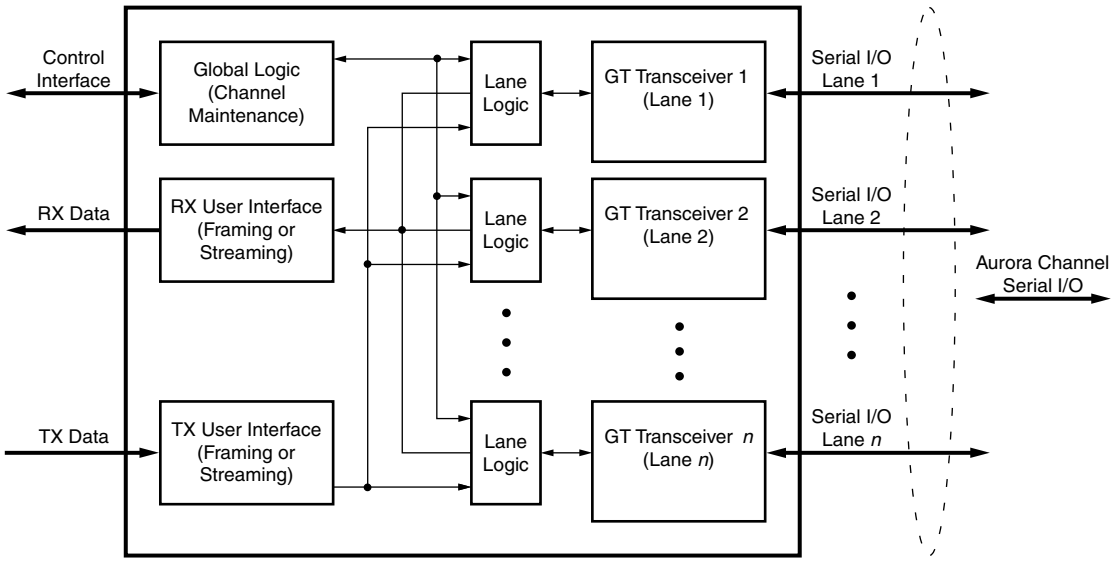

Figure 2-1: **Aurora 8B/10B Core Block Diagram**

The major functional modules of the Aurora 8B/10B core are:

- **Lane logic**: Each GTP/GTX/GTH transceiver is driven by an instance of the lane logic module, which initializes each individual GTP/GTX/GTH transceiver and handles the encoding and decoding of control characters and error detection.
- **Global logic**: The global logic module in each Aurora 8B/10B core performs the bonding and verification phases of channel initialization. While the channel is operating, the module generates the random idle characters required by the Aurora protocol and monitors all the lane logic modules for errors.
- **RX user interface**: The 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 a data valid signal. Frames are presented using a standard AXI4-Stream interface. This module also performs flow control functions.
- **TX user interface**: The TX user interface moves data from the application to the channel. A stream interface with a data valid and a ready signal is 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 8B/10B cores). This interface is normally driven by a standard clock compensation manager module provided with the Aurora 8B/10B core, but it can be turned off, or driven by custom logic to accommodate special needs.

Standards Compliance

The Aurora 8B/10B core is compliant with the *Aurora 8B/10B Protocol Specification v2.2*.

Performance

Maximum Frequencies

Config1 cited in the LogiCORE™ IP Facts table on page 4 runs at 330 MHz in a Virtex®-7 VX690T-FFG1761 device with -2 speed grade. Config1 is a single-lane Aurora 8B/10B core with Streaming interface, 2-byte lane width, Duplex dataflow, targeting a 6.6 Gb/s line rate.

The Aurora 8B/10B cores listed in Table 2-1, page 10 through Table 2-12, page 15 run at 156.25 MHz in devices with speed grades ranging from -1 to -3.

Latency

Latency through an Aurora 8B/10B core is caused by pipeline delays through the protocol engine (PE) and through the GTP/GTX/GTH transceivers. The PE pipeline delay increases as the AXI4-Stream interface width increases. The GTP/GTX/GTH transceivers delays are fixed per the features of the GTP/GTX/GTH transceivers.

This section outlines expected latency for the Aurora 8B/10B core's AXI4-Stream user interface in terms of `USER_CLK` cycles for 2-byte-per-lane and 4-byte-per-lane designs. For the purposes of illustrating latency, the Aurora 8B/10B modules partitioned into GTP/GTX/GTH transceivers logic and protocol engine (PE) logic implemented in the FPGA fabric.

Note: These figures do not include the latency incurred due to the length of the serial connection between each side of the Aurora 8B/10B channel.

Latency of the Frame Path

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

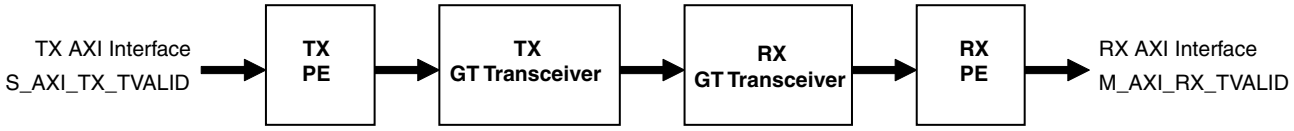

Figure 2-2: **Latency of the Frame Path**

Maximum latency for a 2-byte designs from TX_SOF_N to RX_SOF_N is approximately 52 USER_CLK cycles in simulation.

Maximum latency for a 4-byte designs from TX_SOF_N to RX_SOF_N is approximately 61 USER_CLK cycles in simulation.

The pipeline delays are designed to maintain the clock speed.

Throughput

Aurora core throughput depends on the number of the transceivers and the target line rate of the transceivers selected. Throughput varies from 0.4 Gb/s to 84.48 Gb/s for single lane design to 16 lane design, respectively. The throughput was calculated using 25% overhead of Aurora 8B/10B protocol encoding and 0.5 Gb/s to 6.6 Gb/s line rate range.

Resource Utilization

Table 2-1 through Table 2-12 show the number of look-up tables (LUTs) and flip-flops (FFs) used in selected Aurora configurations. The Aurora 8B/10B core is also available in configurations not shown in the tables. The estimated resource usage for other configurations can be extrapolated from the tables. These tables do not include the additional resource usage for flow control. They also do not include the additional resource usage for the example design modules, such as FRAME_GEN and FRAME_CHECK.

Table 2-1: **Virtex-7 and Kintex-7 Family Resource Usage for Streaming with 2-Byte Lane Width**

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Families			Streaming		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full Duplex	TX Only	RX Only
1	2	FFs	245	158	123
		LUTs	190	120	84
2	2	FFs	391	199	241
		LUTs	338	158	176
4	2	FFs	633	262	417
		LUTs	551	235	298

Table 2-1: **Virtex-7 and Kintex-7 Family Resource Usage for Streaming with 2-Byte Lane Width**

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Families			Streaming		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full Duplex	TX Only	RX Only
8	2	FFs	1119	386	769
		LUTs	1029	397	552
16	2	FFs	2086	642	1473
		LUTs	1922	670	1028

Table 2-2: **Virtex-7 and Kintex-7 Family Resource Usage for Framing with 2-Byte Lane Width**

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Families			Framing		
Lanes	Lane Width	Resource Type	Duplex	Simplex	
			Full Duplex	TX Only	RX Only
1	2	FFs	266	168	136
		LUTs	208	137	94
2	2	FFs	439	204	283
		LUTs	352	164	195
4	2	FFs	711	267	489
		LUTs	587	223	335
8	2	FFs	1253	393	899
		LUTs	1052	373	622
16	2	FFs	2369	649	1752
		LUTs	2003	608	1191

Table 2-3: **Virtex-7 and Kintex-7 Family Resource Usage for Streaming with 4-Byte Lane Width**

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Family			Streaming		
Lanes	Lane Width	Resource Type	Duplex	Simplex	
			Full-Duplex	TX Only	RX Only
1	4	FFs	308	158	180
		LUTs	270	126	119
2	4	FFs	543	211	365
		LUTs	492	191	272
4	4	FFs	940	294	665
		LUTs	880	307	493
8	4	FFs	1734	452	1265
		LUTs	1593	542	902

Table 2-3: **Virtex-7 and Kintex-7 Family Resource Usage for Streaming with 4-Byte Lane Width**

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Family			Streaming		
Lanes	Lane Width	Resource Type	Duplex	Simplex	
			Full-Duplex	TX Only	RX Only
16	4	FFs	3325	780	2465
		LUTs	3144	979	1723

Table 2-4: **Virtex-7 and Kintex-7 Family Resource Usage for Framing with 4-Byte Lane Width**

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Family			Framing		
Lanes	Lane Width	Resource Type	Duplex	Simplex	
			Full Duplex	TX Only	RX Only
1	4	FFs	361	166	223
		LUTs	279	137	148
2	4	FFs	620	215	439
		LUTs	497	175	315
4	4	FFs	1074	299	799
		LUTs	925	250	580
8	4	FFs	2013	453	1552
		LUTs	1714	499	1125
16	4	FFs	3863	773	3027
		LUTs	3334	822	2176

Table 2-5: **Virtex-6 LXT/SXT/CXT/HXT Family Resource Usage for Streaming with 2-Byte Lane Width**

Virtex-6 LXT/SXT/CXT/HXT Family			Streaming		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	2	FFs	243	162	131
		LUTs	209	134	102
2	2	FFs	405	218	262
		LUTs	345	181	196
4	2	FFs	678	319	438
		LUTs	579	284	320
8	2	FFs	1219	516	789
		LUTs	1112	505	573
16	2	FFs	2307	916	1493
		LUTs	2070	820	1073

Table 2-6: **Virtex-6 LXT/SXT/CXT/HXT Family Resource Usage for Framing with 2-Byte Lane Width**

Virtex-6 LXT/SXT/CXT/HXT Family			Framing		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	2	FFs	265	170	145
		LUTs	225	140	111
2	2	FFs	457	227	305
		LUTs	375	191	227
4	2	FFs	765	333	481
		LUTs	641	269	335
8	2	FFs	1373	538	890
		LUTs	1124	459	628
16	2	FFs	2627	954	1744
		LUTs	2200	750	1240

Table 2-7: **Virtex-6 LXT/SXT/CXT/HXT Family Resource Usage for Streaming with 4-Byte Lane Width**

Virtex-6 LXT/SXT/CXT/HXT Family			Streaming		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	4	FFs	321	176	173
		LUTs	271	148	123
2	4	FFs	579	258	356
		LUTs	508	230	256
4	4	FFs	1034	407	656
		LUTs	927	376	469
8	4	FFs	1945	706	1255
		LUTs	1659	626	892
16	4	FFs	3768	1305	2455
		LUTs	3273	1153	1759

Table 2-8: **Virtex-6 LXT/SXT/CXT/HXT Family Resource Usage for Framing with 4-Byte Lane Width**

Virtex-6 LXT/SXT/CXT/HXT Family			Framing		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	4	FFs	366	181	217
		LUTs	303	146	155

Table 2-8: **Virtex-6 LXT/SXT/CXT/HXT Family Resource Usage for Framing with 4-Byte Lane Width**

Virtex-6 LXT/SXT/CXT/HXT Family			Framing		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
2	4	FFs	663	269	431
		LUTs	549	209	314
4	4	FFs	1180	422	791
		LUTs	960	308	580
8	4	FFs	2249	729	1544
		LUTs	1778	531	1130
16	4	FFs	4338	1344	3001
		LUTs	3543	927	2233

Table 2-9: **Spartan-6 LXT Family Resource Usage for Streaming with 2-Byte Lane Width**

Spartan-6 LXT Family			Streaming		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	2	FFs	243	157	126
		LUTs	198	122	96
2	2	FFs	406	206	259
		LUTs	340	171	191
4	2	FFs	677	299	435
		LUTs	601	263	308

Table 2-10: **Spartan-6 LXT Family Resource Usage for Framing with 2-Byte Lane Width**

Spartan-6 LXT Family			Framing		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	2	FFs	264	166	142
		LUTs	217	133	105
2	2	FFs	454	217	302
		LUTs	362	181	220
4	2	FFs	762	313	508
		LUTs	648	266	363

Table 2-11: **Spartan-6 LXT Family Resource Usage for Streaming with 4-Byte Lane Width**

Spartan-6 LXT Family			Streaming		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	4	FFs	318	171	170
		LUTs	263	137	117
2	4	FFs	583	246	383
		LUTs	516	211	284
4	4	FFs	1035	393	683
		LUTs	947	374	493

Table 2-12: **Spartan-6 LXT Family Resource Usage for Framing with 4-Byte Lane Width**

Spartan-6 LXT Family			Framing		
			Duplex	Simplex	
Lanes	Lane Width	Resource Type	Full-Duplex	TX Only	RX Only
1	4	FFs	369	175	214
		LUTs	312	139	149
2	4	FFs	666	256	458
		LUTs	553	199	351
4	4	FFs	1183	401	818
		LUTs	1004	300	621

Port Descriptions

The parameters used to generate each Aurora 8B/10B core determine the interfaces available (Figure 2-3) for that specific core. The Aurora 8B/10B cores have four to six interfaces:

- User Interface, page 16
- User Flow Control Interface, page 18
- Native Flow Control Interface, page 19
- Transceiver Interface, page 23
- Clock Interface, page 24
- Clock Compensation Interface, page 25

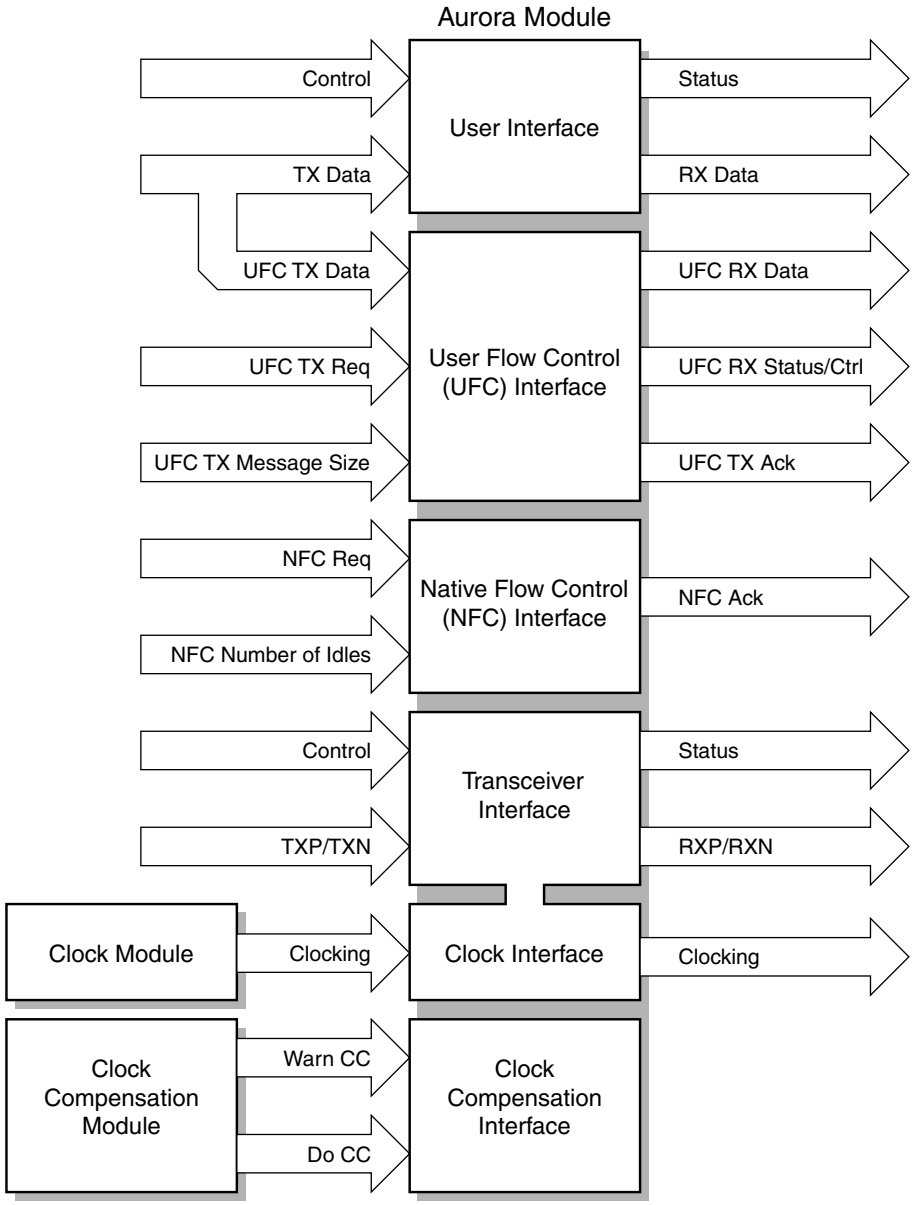

Figure 2-3: **Top-Level Interface**

User Interface

This interface includes all the ports needed to read and write streaming or framed data to and from the Aurora 8B/10B core. AXI4-Stream ports are used if the Aurora 8B/10B core is generated with a framing interface; for streaming modules, the interface consists of a simple set of data ports and data valid ports. Full-duplex cores include ports for both transmit and receive; simplex cores use only the ports they require to send data in the direction they support. The width of the data ports in all interfaces depends on the number of GTP/GTX transceivers in the core, and on the width selected for these transceivers.

Framing Interface Ports

Table 2-13 lists port descriptions for AXI4-Stream TX data ports. These ports are included on full-duplex and simplex TX framing cores.

Table 2-13: **Framing User I/O Ports (TX)**

Name	Direction	Description
S_AXI_TX_TDATA[0:(8n-1)]	Input	Outgoing data (Ascending bit order). • *n* is the number of bytes
S_AXI_TX_TREADY	Output	Asserted (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.
S_AXI_TX_TLAST	Input	Signals the end of the frame (active-High).
S_AXI_TX_TKEEP[0:(n-1)]	Input	Specifies the number of valid bytes in the last data beat; valid only while S_AXI_TX_TLAST is asserted. S_AXI_TX_TKEEP is the byte qualifier that indicates whether the content of the associated byte of S_AXI_TX_TDATA is valid or not. The Aurora core 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 (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.

Table 2-14 lists port descriptions for Framing RX data ports. These ports are included on full-duplex and simplex RX framing cores.

Table 2-14: **Framing User I/O Ports (RX)**

Name	Direction	Description
M_AXI_RX_TDATA[0:8(n-1)]]	Output	Incoming data from channel partner (Ascending bit order).
M_AXI_RX_TLAST	Output	Signals the end of the incoming frame (active-High, asserted for a single user clock cycle). Ignored when M_AXI_RX_TVALID is deasserted (Low).
M_AXI_RX_TKEEP[0:(n-1)]	Output	Specifies the number of valid bytes in the last data beat; valid only when M_AXI_RX_TLAST is asserted.
M_AXI_RX_TVALID	Output	Asserted (High) when data and control signals from an Aurora 8B/10B core are valid. Deasserted (Low) when data and/or control signals from an Aurora 8B/10B core should be ignored.

See Framing Interface, page 39 for more information.

Streaming Interface Ports

Table 2-15 lists the streaming TX data ports. These ports are included on full-duplex and simplex TX framing cores.

Table 2-15: **Streaming User I/O Ports (TX)**

Name	Direction	Description
S_AXI_TX_TDATA[0:(8n-1)]]	Input	Outgoing data (ascending bit order).
S_AXI_TX_TREADY	Output	Asserted (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.
S_AXI_TX_TVALID	Input	Asserted (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.

Table 2-16 lists the streaming RX data ports. These ports are included on full-duplex and simplex RX framing cores.

Table 2-16: **Streaming User I/O Ports (RX)**

Name	Direction	Description
M_AXI_RX_TDATA[0:(8n-1)]	Output	Incoming data from channel partner (Ascending bit order).
M_AXI_RX_TVALID	Output	Asserted (High) when data and control signals from an Aurora 8B/10B core are valid. Deasserted (Low) when data from an Aurora 8B/10B core should be ignored.

See Streaming Interface, page 47 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 and an acknowledge port that are used to start a UFC message, and a 3-bit port to specify the length of the message. The user supplies the message data to the data port of the user interface; immediately after a UFC request is acknowledged, the user interface indicates it is no longer ready for normal data, thereby allowing UFC data to be written to the 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 2-17 describes the ports for the UFC interface.

Table 2-17: **UFC I/O Ports**

Name	Direction	Description
S_AXI_UFC_TX_REQ	Input	Asserted to request a UFC message be sent to the channel partner (active-High). Must be held until S_AXI_UFC_TX_ACK is asserted. Do not assert this signal unless the entire UFC message is ready to be sent; a UFC message cannot be interrupted after it has started.
S_AXI_UFC_TX_MS[0:2]	Input	Specifies the size of the UFC message that is sent. The SIZE encoding is a value between 0 and 7. See Table 3-9, page 52.
S_AXI_UFC_TX_ACK	Output	Asserted when an Aurora 8B/10B core is ready to read the contents of the UFC message (active-High). On the cycle after the S_AXI_UFC_TX_ACK signal is asserted, data on the S_AXI_TX_TDATA port is treated as UFC data. S_AXI_TX_TDATA data continues to be used to fill the UFC message until enough cycles have passed to send the complete message. Unused bytes from a UFC cycle are discarded.
M_AXI_UFC_RX_TDATA[0:(8n-1)]	Output	Incoming UFC message data from the channel partner (n = 16 bytes maximum).
M_AXI_UFC_RX_TVALID	Output	Asserted when the values on the M_AXI_UFC_RX ports are valid. When this signal is not asserted, all values on the M_AXI_UFC_RX ports should be ignored (active-High).
M_AXI_UFC_RX_TLAST	Output	Signals the end of the incoming UFC message (active-High).
M_AXI_UFC_RX_TKEEP[0:(n-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 (*n* = 16 bytes maximum).

See User Flow Control, page 51 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, and a 4-bit port to specify the number of idle cycles requested.

Table 2-18 lists the ports for the NFC interface available only in full-duplex Aurora 8B/10B cores.

Table 2-18: **NFC I/O Ports**

Name	Direction	Description
S_AXI_NFC_ACK	Output	Asserted when an Aurora 8B/10B core accepts an NFC request (active-High).
S_AXI_NFC_NB[0:3]	Input	Indicates the number of PAUSE idles the channel partner must send when it receives the NFC message. Must be held until S_AXI_NFC_ACK is asserted.

Table 2-18: **NFC I/O Ports *(Cont'd)***

Name	Direction	Description
S_AXI_NFC_REQ	Input	Asserted to request an NFC message be sent to the channel partner (active-High). Must be held until S_AXI_NFC_ACK is asserted.
M_AXI_RX_SNF	Output	Indicates an NFC message is received from the partner. This port is asserted for one USER_CLK cycle.
M_AXI_RX_FC_NB[0:3]	Output	Indicates the PAUSE value of the received NFC message. This port should be sampled with M_AXI_RX_SNF.

See Native Flow Control, page 49 for more information.

Status and Control Ports for Full-Duplex Cores

Table 2-19 describes the function of each of the status and control ports for full-duplex cores.

Table 2-19: **Status and Control Ports for Full-Duplex Cores**

Name	Direction	Description
CHANNEL_UP	Output	Asserted when Aurora 8B/10B channel initialization is complete and channel is ready to send data. The Aurora 8B/10B core cannot receive data before CHANNEL_UP.
LANE_UP[0:*m*-1](1)	Output	Asserted for each lane upon successful lane initialization, with each bit representing one lane (active-High). The Aurora 8B/10B core can only receive data after all LANE_UP signals are High.
FRAME_ERR	Output	Channel frame/protocol error detected. This port is active-High and is asserted for a single clock.
HARD_ERR	Output	Hard error detected. (Active High, asserted until Aurora 8B/10B core resets). See Error Signals in Full-Duplex Cores, page 57 for more details.
LOOPBACK[2:0]	Input	The LOOPBACK[2:0] port selects between the normal operation mode and the different loopback modes. See the *7 Series FPGAs GTX Transceivers User Guide*, *Virtex-6 FPGA GTX Transceivers User Guide*, or the *Spartan-6 FPGA GTP Transceivers User Guide* for details about loopback.
POWER_DOWN	Input	Drives the power-down input of the GTP/GTX transceiver (active-High).
RESET	Input	Resets the Aurora 8B/10B core (active High). This signal must be synchronous to USER_CLK and must be asserted for at least one USER_CLK cycle.
SOFT_ERR	Output	Soft error detected in the incoming serial stream. See Error Signals in Full-Duplex Cores, page 57 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.

Table 2-19: **Status and Control Ports for Full-Duplex Cores *(Cont'd)***

Name	Direction	Description
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
TXN[0:*m*-1]	Output	Negative differential serial data output pin.
GT_RESET	Input	The reset signal for the PMA modules in the transceivers is connected to the top level through a debouncer. The GT_RESET port should be asserted (active-High) when the module is first powered up in hardware. This systematically resets all PCS and PMA subcomponents of the transceiver. The signal is debounced using INIT_CLK_IN. See the Reset section in the respective transceiver user guide for further details.
INIT_CLK_IN	Input	INIT_CLK_IN is used to register and debounce the GT_RESET signal. INIT_CLK_IN is required because USER_CLK stops when GT_RESET is asserted. INIT_CLK_IN should be set to a slow rate, preferably slower than the reference clock. INIT_CLK_IN is a board clock. It is recommended to set this frequency lower than the GT reference clock frequency. For example, the KC724 board has a 200 MHz crystal oscillator, and it is constrained for the 50 MHz frequency by default in <component name>_exdes.xdc. Users need to update this clock constraint with respect to their board clock frequency.

Notes:

1. *m* is the number of GTP/GTX/GTH transceivers.

See Full-Duplex Cores, page 56 for more information.

Status and Control Ports for Simplex Cores

Table 2-20 describes the function of each of the status and control ports in the simplex TX interface.

Table 2-20: **Status and Control Ports for Simplex TX Cores**

Name	Direction	Description
TX_ALIGNED	Input	Asserted when RX channel partner has completed lane initialization for all lanes. Typically connected to RX_ALIGNED.
TX_BONDED	Input	Asserted when RX channel partner has completed channel bonding. Not needed for single-lane channels. Typically connected to RX_BONDED.
TX_VERIFY	Input	Asserted when RX channel partner has completed verification. Typically connected to RX_VERIFY.

Table 2-20: **Status and Control Ports for Simplex TX Cores *(Cont'd)***

Name	Direction	Description
TX_RESET	Input	Asserted when reset is required because of initialization status of RX channel partner. This signal must be synchronous to USER_CLK and must be asserted for at least one USER_CLK cycle. Typically connected to RX_RESET.
TX_CHANNEL_UP	Output	Asserted when Aurora 8B/10B channel initialization is complete and channel is ready to send data. The Aurora 8B/10B core cannot receive data before TX_CHANNEL_UP.
TX_LANE_UP[0:*m*-1]	Output	Asserted for each lane upon successful lane initialization, with each bit representing one lane (active-High).
TX_HARD_ERR	Output	Hard error detected. (Active-High, asserted until Aurora 8B/10B core resets). See Error Signals in Simplex Cores, page 59 for more details.
POWER_DOWN	Input	Drives the powerdown input of the GTP/GTX transceiver (active-High).
TX_SYSTEM_RESET	Input	Resets the Aurora 8B/10B core (active-High).
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 GTP/GTX transceivers.

Table 2-21 describes the function of each of the status and control ports in the simplex RX interface.

Table 2-21: **Status and Control Ports for Simplex RX Cores**

Name	Direction	Description
RX_ALIGNED	Output	Asserted when RX module has completed lane initialization. Typically connected to TX_ALIGNED.
RX_BONDED	Output	Asserted when RX module has completed channel bonding. Not used for single-lane channels. Typically connected to TX_BONDED.
RX_VERIFY	Output	Asserted when RX module has completed verification. Typically connected to TX_VERIFY.
RX_RESET	Output	Asserted when the RX module needs the TX module to restart initialization. Typically connected to TX_RESET.
RX_CHANNEL_UP	Output	Asserted when Aurora 8B/10B channel initialization is complete and channel is ready to send data. The Aurora 8B/10B core cannot receive data before RX_CHANNEL_UP.

Table 2-21: **Status and Control Ports for Simplex RX Cores *(Cont'd)***

Name	Direction	Description
RX_LANE_UP[0:*m*-1]	Output	Asserted for each lane upon successful lane initialization, with each bit representing one lane (active-High). The Aurora 8B/10B core can only receive data after all RX_LANE_UP signals are High.
FRAME_ERR	Output	Channel frame/protocol error detected. This port is active-High and is asserted for a single clock.
RX_HARD_ERR	Output	Hard error detected. (Active-High, asserted until Aurora 8B/10B core resets). See Error Signals in Simplex Cores, page 59 for more details.
POWER_DOWN	Input	Drives the power-down input of the GTP/GTX transceiver (active-High).
RX_SYSTEM_RESET	Input	Resets the Aurora 8B/10B core (active-High).
SOFT_ERR	Output	Soft error detected in the incoming serial stream. See Error Signals in Simplex Cores, page 59 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 GTP/GTX transceivers.
2. RX_ALIGNED, RX_BONDED, RX_VERIFY, and RX_RESET are available as output signals even when the simplex partner is timer based, but functionally these signals are not required.

See Simplex Cores, page 59 for more information.

Transceiver Interface

This interface includes the serial I/O ports of the GTP/GTX/GTH transceivers, and the control and status ports of the Aurora 8B/10B core. This interface is the user's access to control functions such as reset, loopback, channel bonding, clock correction, and power down. Status information about the state of the channel, and error information is also available here.

Table 2-22 describes the transceiver ports.

Table 2-22: **Transceiver Ports**

Name	Direction	Description
RXP[0:*m*-1](1)	Input	Positive differential serial data input pin.
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.

Table 2-22: **Transceiver Ports *(Cont'd)***

Name	Direction	Description
POWER_DOWN	Input	Drives the power-down input of the GTP/GTX transceiver (active-High).

Notes:
1. *m* is the number of GTP/GTX transceivers.

Clock Interface

This interface is most critical for correct Aurora 8B/10B core operation. The clock interface has ports for the reference clocks that drive the GTP/GTX/GTH transceivers, and ports for the parallel clocks that the Aurora 8B/10B core shares with application logic.

Table 2-23 describes the Aurora 8B/10B core clock ports.

Table 2-23: **Clock Ports for a GTP/GTX Aurora 8B/10B Core**

Clock Ports	Direction	Description
PLL_NOT_LOCKED	Input	If a PLL is used to generate clocks for the Aurora 8B/10B core, the PLL_NOT_LOCKED signal should be connected to the inverse of the PLL's LOCKED signal. The clock module provided with the Aurora 8B/10B core uses the PLL for clock division. The PLL_NOT_LOCKED signal from the clock module should be connected to the PLL_NOT_LOCKED signal on the Aurora 8B/10B core. If the PLL is not used to generate clock signals for the Aurora 8B/10B core, tie PLL_NOT_LOCKED to ground.
USER_CLK	Input	Parallel clock shared by the Aurora 8B/10B core and the user application. In Aurora 8B/10B cores, USER_CLK and SYNC_CLK are the outputs of a PLL or BUFG whose input is derived from TX_OUT_CLK. These clock generations are available in <component name>_clock_module file. The Spartan-6 FPGA uses the GTPCLKOUT port to derive USER_CLK and SYNC_CLK outputs. USER_CLK goes as the TXUSRCLK2 input to the transceiver. See the respective transceiver user guide for more information.
SYNC_CLK	Input	Parallel clock used by the internal synchronization logic of the GTP/GTX/GTH transceivers in the Aurora 8B/10B core. SYNC_CLK goes as the TXUSRCLK input to the transceiver. See the respective transceiver user guide for more information.
GT_REFCLK	Input	GT_REFCLK (CLKP/CLKN) is a dedicated external clock generated from an oscillator. This clock is fed through IBUFDS. To minimize the number of oscillators, the GTP/GTX/GTH transceiver architecture has a NORTH/SOUTH clock routing matrix using CLKP/CLKN.

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. For modules with UFC and NFC, the WARN_CC port prevents UFC messages and CC sequences from colliding. Each Aurora 8B/10B core is accompanied by a clock compensation management module that is used to drive the clock compensation interface in accordance with the *Aurora 8B/10B Protocol Specification*. When the same physical clock is used on both sides of the channel, WARN_CC and DO_CC should be tied Low.

Table 2-24 describes the function of the clock compensation interface ports.

Table 2-24: **Clock Compensation I/O Ports**

Name	Direction	Description
DO_CC	Input	The Aurora 8B/10B 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.
WARN_CC	Input	The Aurora 8B/10B core does not acknowledge UFC requests while this signal is asserted. It is used to prevent UFC messages from starting too close to CC events. Connects to the `WARN_CC` output on the CC module.

See Clock Compensation Interface, page 34 for more information.

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

This section describes the steps required to turn an Aurora 8B/10B core into a fully functioning design with user-application logic. Not all implementations require all of the design steps listed here. Follow the logic design guidelines in this manual carefully.

Use the Example Design as a Starting Point

Each instance of an Aurora 8B/10B core created by the Vivado™ IP catalog is delivered with an example design that can be simulated and implemented in FPGA. This design can be used as a starting point for your own design or can be used to troubleshoot the user application, if necessary.

Know the Degree of Difficulty

Aurora 8B/10B design is challenging to implement in any technology, and the degree of difficulty is further influenced by:

- Maximum system clock frequency
- Targeted device architecture
- Nature of the user application

All Aurora 8B/10B 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. Registering signals might not be possible for all paths, but doing so simplifies timing analysis and makes it easier for the Xilinx tools to place-and-route the design.

Recognize Timing Critical Signals

The 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 or VHDL source code. The example implementation scripts provided currently use the Vivado synthesis tool for the example design that is delivered with the core. Other synthesis tools can also be used.

Make Only Allowed Modifications

The Aurora 8B/10B core is not user modifiable. Any modifications might have adverse effects on the system timings and protocol compliance. Supported user configurations of the Aurora 8B/10B core can only be made by selecting options from the GUI.

Clocking

Clock Interface and Clocking

Good clocking is critical for the correct operation of the Aurora 8B/10B core. The core requires a high-quality, low-jitter reference clock to drive the high-speed TX clock and clock recovery circuits in the GTP/GTX/GTH transceivers. It also requires at least one frequency locked parallel clock for synchronous operation with the user application.

The Virtex®-7/Kintex™-7/Virtex-6 FPGA has four GTX/GTH transceivers in a Quad. The Spartan®-6 FPGA GTP architecture has a pair of transceivers in each GTPA1_DUAL tile. Virtex-7/Kintex-7 FPGA GTX/GTH transceivers have a channel PLL (CPLL) per transceiver and a Quad PLL (QPLL) per quad. The Virtex-6 FPGA GTX transceiver has individual PLLs for both TX and RX portion of the transceivers. The Spartan-6 FPGA has individual PLLs for each transceiver in a GTPA1_DUAL tile. The reference clock is used to produce the PLL clock, which is divided to make individual TX and RX serial clocks and parallel clocks in each GTP/GTX/GTH transceiver.

Each Aurora 8B/10B core is generated in the `example_design` directory that includes a design called aurora_example. This design by instantiating the generated Aurora 8B/10B core, demonstrates a working clock configuration of the core. First-time users should examine the Aurora example design and use it as a template when connecting the clock interface.

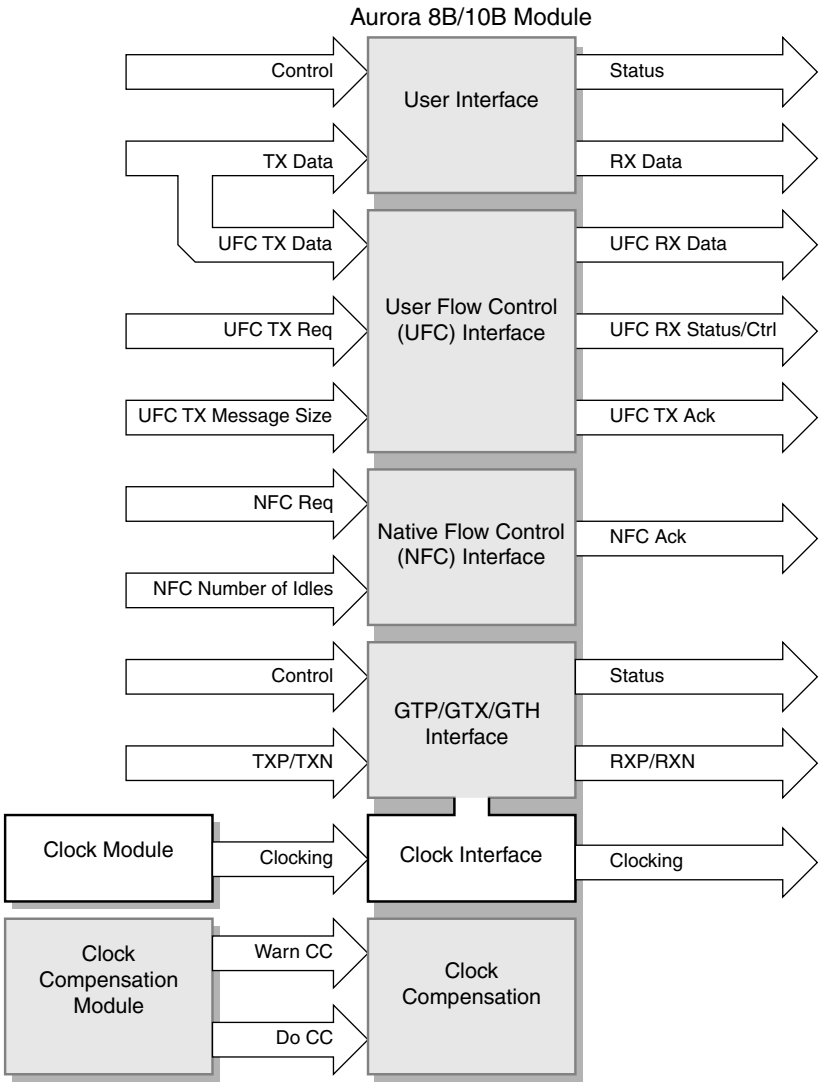

Figure 3-1: **Top-Level Clocking**

Clock Interface Ports for the Aurora Core

See Table 2-23, page 24 for descriptions of the transceiver ports on the clock interface.

Clocking from a Neighboring GTX/GTH Transceiver for Virtex-7/Kintex-7 FPGA Designs

The Xilinx implementation tools make necessary adjustments to the north-south routing shown in Figure 3-2, page 31 as well as pin swapping necessary to GTXE2/GTHE2 clock inputs to route clocks from one Quad to another when required.

The following rules must be observed when sharing a reference clock to ensure that jitter margins for high-speed designs are met:

1. The number of GTX/GTH Quads above the sourcing Quad must not exceed one.
2. The number of GTX/GTH Quads below the sourcing Quad must not exceed one.
3. The total number of GTX/GTH Quads sourced by an external clock pin pair (MGTREFCLKN/MGTREFCLKP) must not exceed three or 12 GTXE2_CHANNEL/GTHE2_CHANNEL transceivers.

The maximum number of GTX/GTH transceivers that can be sourced by a single clock pin pair is 12. Designs with more than 12 transceivers require the use of multiple external clock pins to ensure that the rules for controlling jitter are followed. When multiple clock pins are used, an external buffer can be used to drive them from the same oscillator.

Note: Virtex-6 FPGA reference clock sharing guidelines are the same as the guidelines for Virtex-7/Kintex-7 FPGAs.

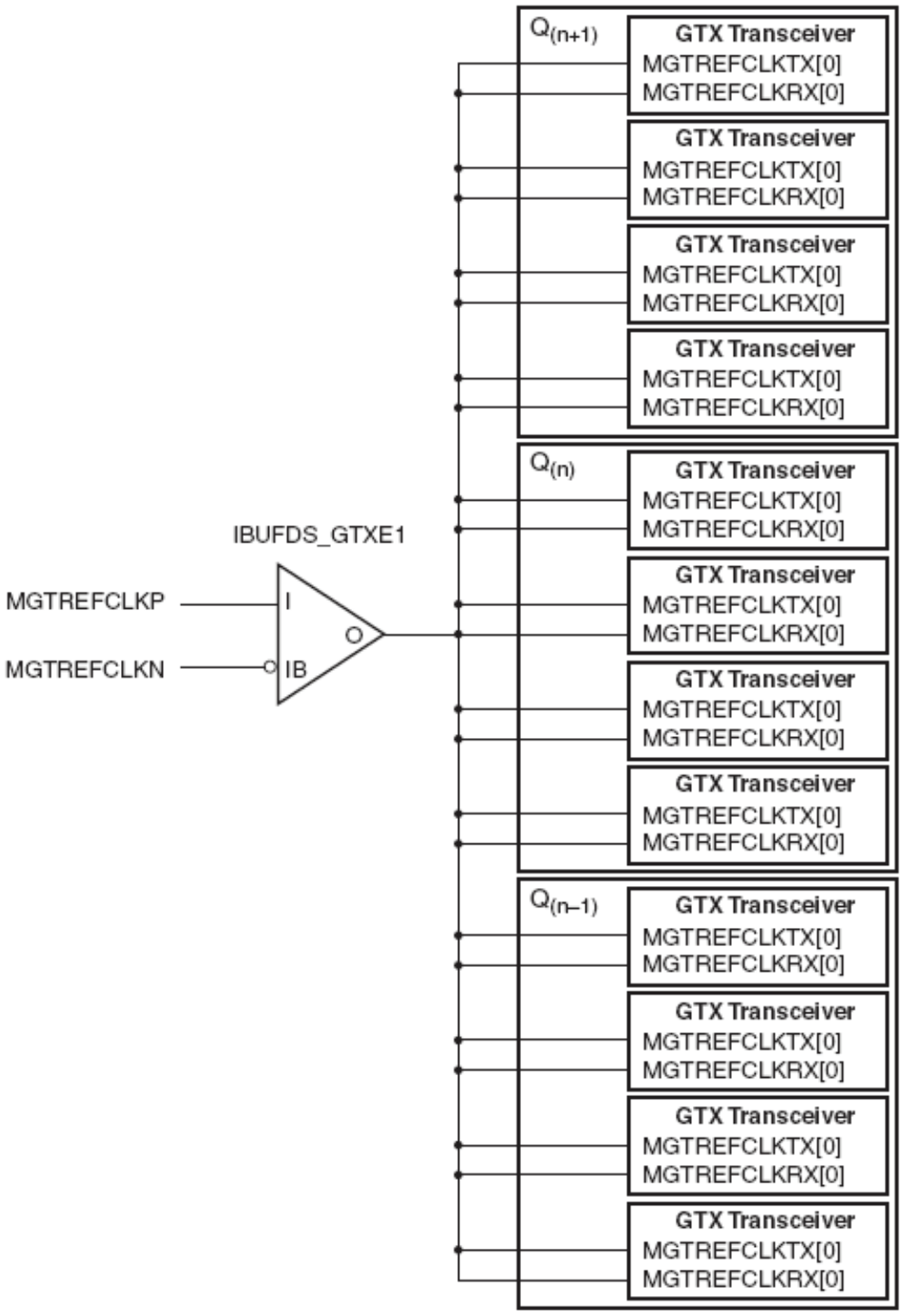

Figure 3-2: **North-South Routing Adjustments in Virtex-7/Kintex-7 FPGAs**

Reference Clocks for Spartan-6 FPGA GTP Transceiver Designs

In Spartan-6 FPGA transceiver designs, the reference clock is GTPD, which is a differential input clock for each GTPA1_DUAL. The reference clock for GTPA1_DUAL is provided through

the CLK00 and CLK01 ports. The two possible use models for distributing a reference clock to drive the CLK00 and CLK01 ports are:

- Clocking from an External Source
- Clocking from a Neighboring GTPA1_DUAL Tile

Clocking from an External Source

Each GTPA1_DUAL tile has a pair of dedicated pins that can be connected to an external clock source. To use these pins, a IBUFDS primitive is instantiated. In the constraints file (XDC), the IBUFDS input pins are set to the dedicated clock pins for the tile. In the design, the output of the IBUFDS is connected to the CLK00 and CLK01 ports. Each GTPA1_DUAL takes differential GTPD clock inputs, which are directly bonded to the FPGA pins. For multilane Aurora 8B/10B designs, GTPD clock of any GTPA1_DUAL can be used as the reference clock for the Aurora 8B/10B design. Using a low-jitter oscillator delivers a high-quality clock suitable for top-speed operation. Figure 3-3 shows a differential GTPA1_DUAL clock pin pair sourced by an external oscillator on the board. This clocking mechanism is used for Spartan-6 FPGA single lane and 2-lane designs.

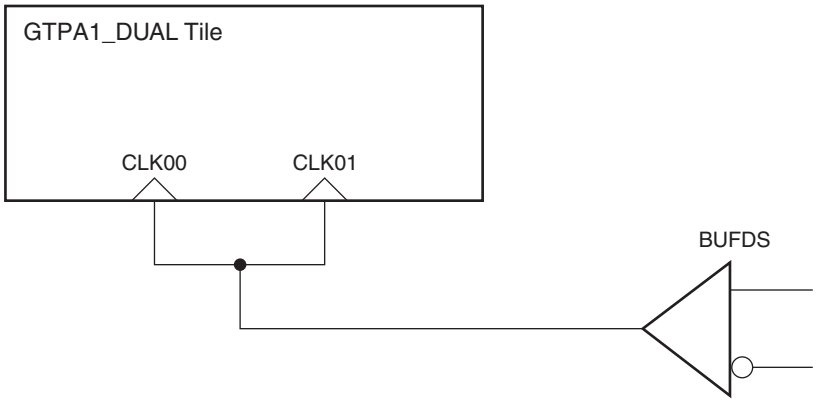

Figure 3-3: **Single GTPA1_DUAL Tile Clocked Externally**

Clocking from a Neighboring GTPA1_DUAL Tile

The external clock from one tile can be used to drive the CLK00 and CLK01 ports of neighboring tiles.

The example in Figure 3-4 uses the clock from one GTPA1_DUAL tile to clock neighboring tiles. A GTPA1_DUAL tile shares its clock with its neighbors using dedicated clock routing resources. This clocking mechanism is used for Spartan-6 FPGA 4-lane design.

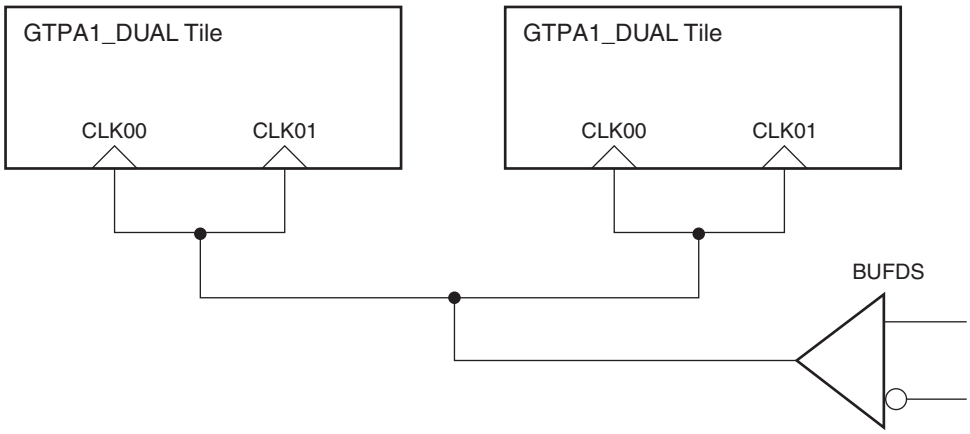

Figure 3-4: **GTPA1_DUAL Tiles with Shared Reference Clock**

Clock Rates for GTP/GTX/GTH Transceiver Designs

GTP/GTX/GTH transceivers support a wide range of serial rates. The attributes used to configure the GTP/GTX/GTH transceivers in the Aurora 8B/10B core for a specific line rate are kept in the transceiver_wrapper module for simulation. These attributes are set automatically by the IP catalog in response to the line rate and reference clock selections made in the Configuration GUI window for the core. Manual edits of the attributes are not recommended, but are possible using the recommendations in the *7 Series FPGAs GTX/GTH Transceivers User Guide*, *Virtex-6 FPGA GTX Transceivers User Guide*, and the *Spartan-6 FPGA GTP Transceivers User Guide*.

Clock Compensation

Clock compensation is a feature that allows up to ± 100 ppm difference in the reference clock frequencies used on each side of an Aurora 8B/10B 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.

The Aurora 8B/10B core's clock compensation interface enables full control over the core's clock compensation features. A standard clock compensation module is generated with the Aurora 8B/10B core to provide Aurora 8B/10B-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.

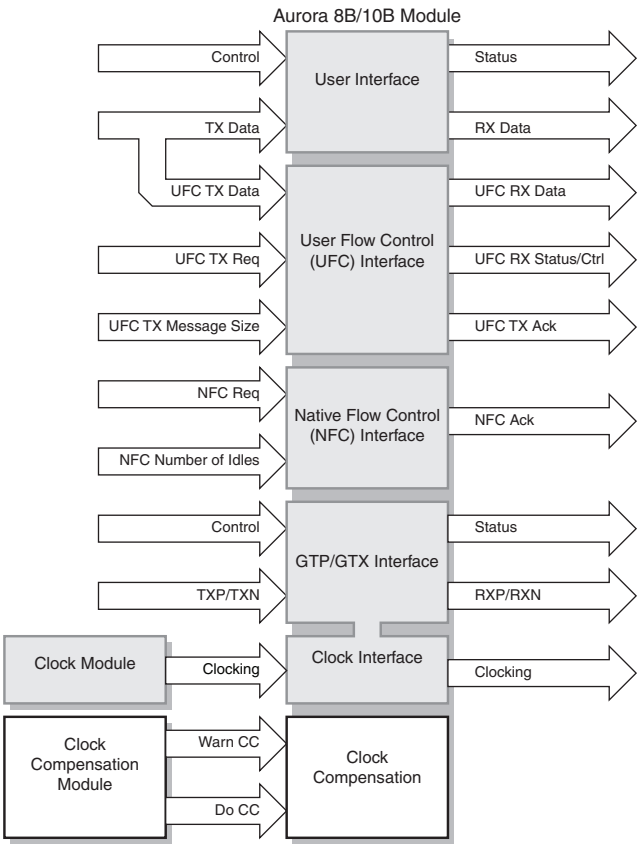

Figure 3-5: **Top-Level Clock Compensation**

Clock Compensation Interface

All Aurora 8B/10B cores include a clock compensation interface for controlling the transmission of clock compensation sequences.

Figure 3-6 and Figure 3-7 are waveform diagrams showing how the DO_CC signal works.

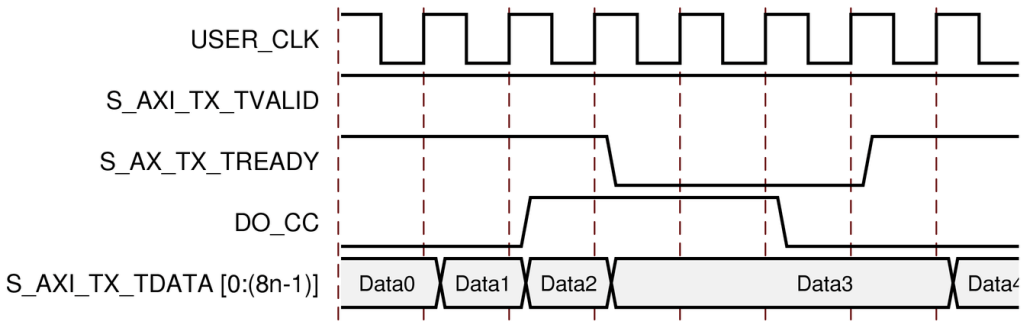

Figure 3-6: **Streaming Data with Clock Compensation Inserted**

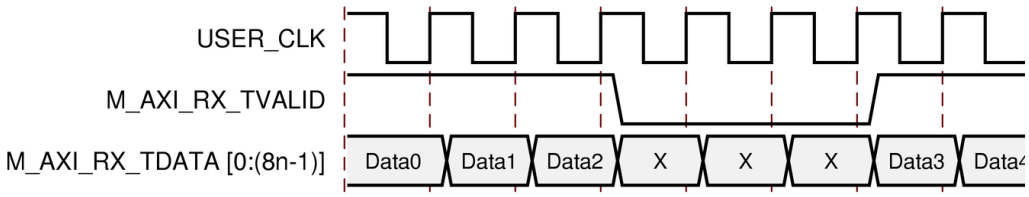

Figure 3-7: **Data Reception Interrupted by Clock Compensation**

The Aurora 8B/10B protocol specifies a clock compensation mechanism that allows up to ± 100 ppm difference between reference clocks on each side of an Aurora 8B/10B channel. To perform Aurora 8B/10B-compliant clock compensation, DO_CC must be asserted for several cycles in every clock compensation period. The duration of the DO_CC assertion and the length of time between assertions is determined based on the width of the GTP/GTX transceiver data interface. While DO_CC is asserted, S_AXI_TX_TREADY on the user interface for modules with TX while the channel is being used to transmit clock compensation sequences. Table 3-1 shows the required durations and periods for 2-byte and 4-byte wide lanes.

Table 3-1: **Clock Compensation Cycles**

Lane Width	USER_CLK Cycles Between DO_CC	DO_CC Duration (USER_CLK cycles)
2	5000	6
4	3000	3

WARN_CC is for cores with user flow control (UFC) and/or native flow control (NFC). Driving this signal before DO_CC is asserted prevents the UFC interface from acknowledging and sending UFC messages too close to a clock correction sequence. This precaution is necessary because data corruption occurs when CC sequences and UFC messages overlap. The number of lookahead cycles required to prevent a 16-byte UFC message from colliding with a clock compensation sequence depends on the number of lanes in the channel and the width of each lane. Table 3-2 shows the number of lookahead cycles required for each combination of lane width, channel width, and maximum UFC message size.

Table 3-2: **Lookahead Cycles**

Data Interface Width	Max UFC Size	WARN_CC Lookahead
2	2	3
2	4	4
2	6	5
2	8	6
2	10	7
2	12	8
2	14	9

Table 3-2: **Lookahead Cycles *(Cont'd)***

Data Interface Width	Max UFC Size	WARN_CC Lookahead
2	16	10
4	2-4	3
4	6-8	4
4	10-12	5
4	14-16	6
6	2-6	3
6	8-12	4
6	14-16	5
8	2-8	3
8	10-16	4
10	2-10	3
10	12-16	4
12	2-12	3
12	14-16	4
14	2-14	3
14	16	4
≥16	2-16	3

Native flow control message requests are not acknowledged during assertion of WARN_CC and DO_CC signals. This helps to prevent the collision of an NFC message and the clock compensation sequence.

To make Aurora 8B/10B compliance easy, a standard clock compensation module is generated along with each Aurora 8B/10B core from the Vivado IP catalog in the cc_manager subdirectory. It automatically generates pulses to create Aurora 8B/10B compliant clock compensation sequences on the DO_CC port and sufficiently early pulses on the WARN_CC port to prevent UFC collisions with maximum-sized UFC messages. This module must always be connected to the clock compensation port on the Aurora 8B/10B module, except in special cases. Table 3-3 shows the port description for the standard CC module.

Table 3-3: **Standard CC I/O Port**

Name	Direction	Description
WARN_CC	Output	Connect this port to the WARN_CC input of the Aurora 8B/10B core when using UFC.
DO_CC	Output	Connect this port to the DO_CC input of the Aurora 8B/10B 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 simplex TX port.

Clock compensation is not needed when both sides of the Aurora 8B/10B channel are being driven by the same clock (see Figure 3-7, page 35) because the reference clock frequencies on both sides of the module are locked. In this case, WARN_CC and DO_CC should both be tied to ground. Additionally, the CLK_CORRECT_USE attribute can be set to false in the transceiver interface module for the core. This can result in lower latencies for single lane modules.

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. 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 the following guidelines:

- Clock compensation sequences should last at least two cycles to ensure they are recognized by all receivers
- Be sure the duration and period selected is sufficient to correct for the maximum difference between the frequencies of the clocks that are used
- Do not perform multiple clock correction sequences within eight cycles of one another
- Replacing long sequences of idles (>12 cycles) with CC sequences results in increased EMI
- DO_CC has no effect until after CHANNEL_UP; DO_CC should be asserted immediately after CHANNEL_UP because no clock compensation can occur during initialization

User Interface

An Aurora 8B/10B core can be generated with either a *framing* or *streaming* user data interface. In addition, flow control options are available for designs with framing interfaces. See Flow Control, page 49.

The framing user interface complies with the *AXI4-Stream Protocol Specification*. It comprises the signals necessary for transmitting and receiving framed user data. The streaming interface allows users to send data without special frame delimiters. It is simple to operate and uses fewer resources than framing.

Top-Level Architecture

Aurora 8B/10B top level (block level) file instantiates Aurora 8B/10B lane module, TX and RX AXI4-Stream modules, global logic module, and 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 Aurora 8B/10B top level for a duplex configuration. The top-level file is the starting point for a user design.

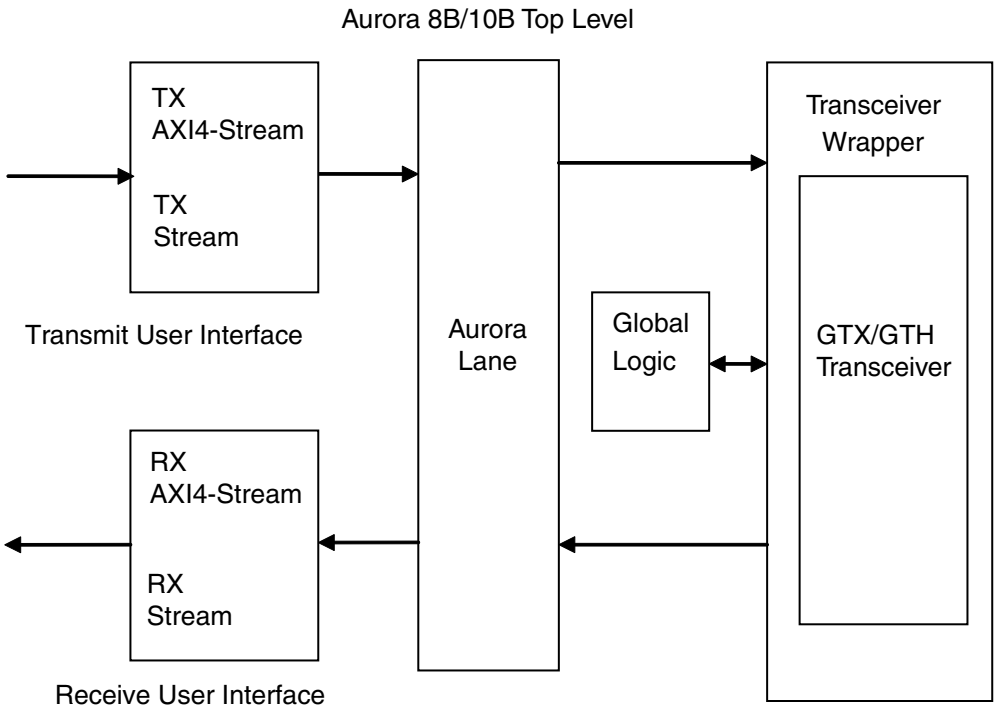

Figure 3-8: **Top-Level Architecture**

The following sections describe the streaming and framing interface in details. User interface logic should be designed to comply with the timing requirement of the respective interface as explained in the subsequent sections.

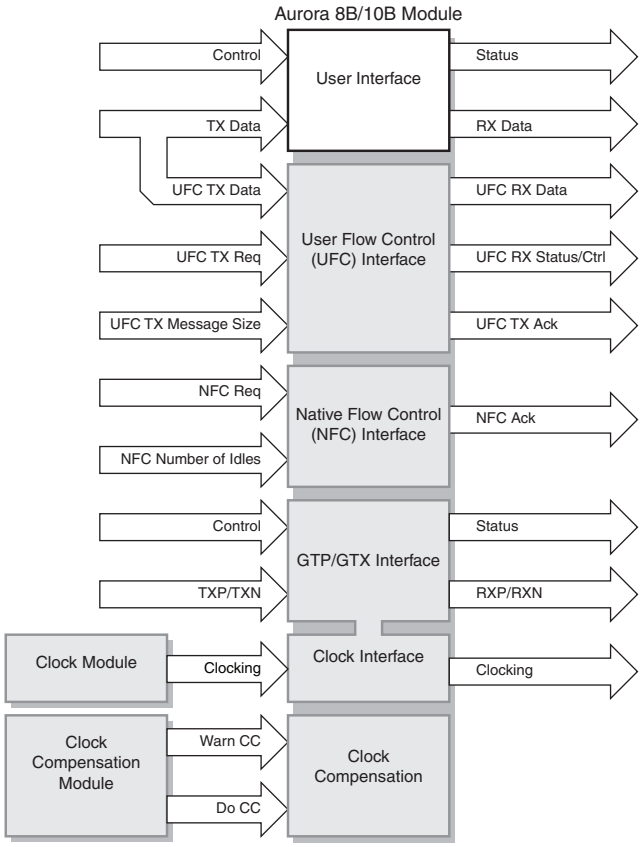

Figure 3-9: **Top-Level User Interface**

Note: The user interface signals vary depending upon the selections made when generating an Aurora 8B/10B core in the Vivado IP catalog.

Framing Interface

Figure 3-10 shows the framing user interface of the Aurora 8B/10B core, with AXI4-Stream compliant ports for TX and RX data.

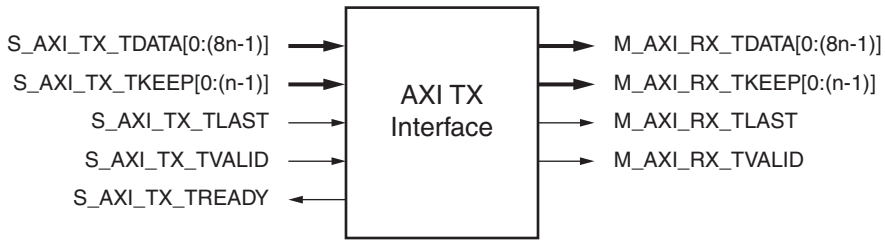

Figure 3-10: **Aurora 8B/10B Core Framing Interface (AXI4-Stream)**

To transmit data, the user manipulates control signals to cause the core to do the following:

- Take data from the user on the `S_AXI_TX_TDATA` bus

- Encapsulate and stripe the data across lanes in the Aurora 8B/10B channel (S_AXI_TX_TVALID, 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, SCP, ECP)
- Asserts framing signal (M_AXI_RX_TLAST)
- Recovers data from the lanes
- Assembles data for presentation to the user on the M_AXI_RX_TDATA bus

AXI4-Stream Bit Ordering

Aurora 8B/10B cores use ascending ordering. They transmit and receive the most significant bit of the most significant byte first. Figure 3-11 shows the organization of an *n*-byte example of the AXI4-Stream data interfaces of an Aurora 8B/10B core.

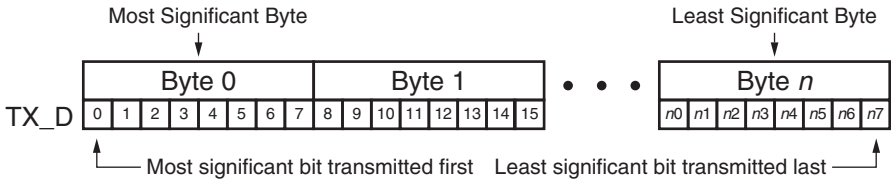

Figure 3-11: **AXI4-Stream Interface Bit Ordering**

Transmitting Data

AXI4-Stream is a synchronous interface. The Aurora 8B/10B 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 (High).

When AXI4-Stream signals are sampled, they are only considered valid if S_AXI_TX_TVALID is asserted. The user application can deassert S_AXI_TX_TVALID on any clock cycle; this causes the Aurora 8B/10B 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 8B/10B 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 start a frame, assert S_AXI_TX_TVALID while the first word of data is on the S_AXI_TX_TDATA port. 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.

Note: In the case of frames that are a single word long or less, S_AXI_TX_TVALID and S_AXI_TX_TLAST are asserted simultaneously.

Data Remainder

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.

Aurora 8B/10B Frames

The TX submodules translate each user frame that it receives through the TX interface to an Aurora 8B/10B frame. The 2-byte SCP code group is added to the beginning of the frame data to indicate the start of frame, and a 2-byte ECP set is sent after the frame ends to indicate the end of frame. Idle code groups are inserted whenever data is not available. Code groups are 8B/10B encoded byte pairs. All data in Aurora 8B/10B is sent as code groups, so user frames with an odd number of bytes have a control character called PAD appended to the end of the frame to fill out the final code group. Table 3-4 shows a typical Aurora 8B/10B frame with an even number of data bytes.

Length

The user controls the channel frame length by manipulation of the S_AXI_TX_TVALID and S_AXI_TX_TLAST signals. The Aurora 8B/10B core responds with start-of-frame and end-of-frame ordered sets, /SCP/ and /ECP/ respectively, as shown in Table 3-4.

Table 3-4: **Typical Channel Frame**

$/SCP/_1$	$/SCP/_2$	Data Byte 0	Data Byte 1	Data Byte 2	...	Data Byte *n* -1	Data Byte *n*	$/ECP/_1$	$/ECP/_2$

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 3*n* 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 8B/10B core is not sending data, it sends idle sequences.

To begin the data transfer, the user asserts S_AXI_TX_TVALID and the first n bytes of the user frame. Because S_AXI_TX_TREADY is already asserted, data transfer begins on the next clock edge. An /SCP/ ordered set is placed on the first two bytes of the channel to indicate the start of the frame. Then the first *n*-2 data bytes are placed on the channel. Because of the offset required for the /SCP/, the last two bytes in each data beat are always delayed one cycle and transmitted on the first two bytes of the next beat of the channel.

To end the data transfer, the user asserts S_AXI_TX_TLAST, 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 N (in the waveform for demonstration) to indicate that all bytes are valid in the last data beat. One clock cycle after S_AXI_TX_TLAST is asserted, the AXI4-Stream interface

deasserts S_AXI_TX_TREADY and uses the gap in the data flow to send the final offset data bytes and the /ECP/ ordered set, indicating 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 8B/10B core sends idles.

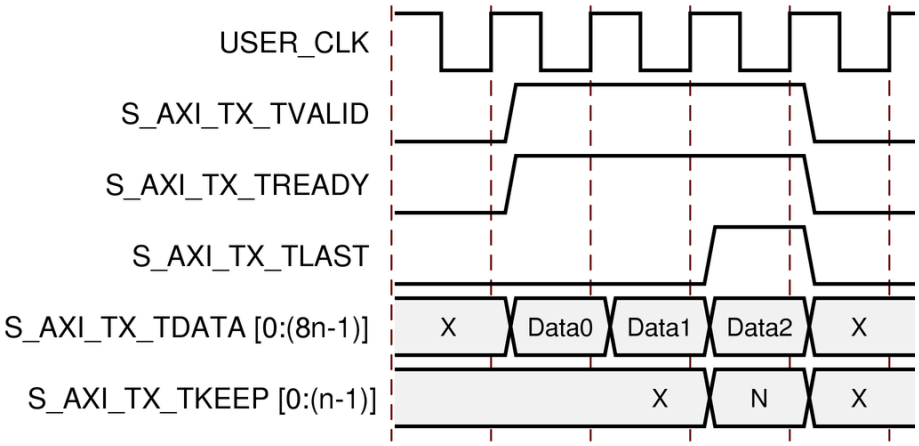

Figure 3-12: **Simple Data Transfer**

Example B: Data Transfer with Pad

Figure 3-13 shows an example of a (3*n*-1)-byte data transfer that requires the use of a pad. Because there is an odd number of data bytes, the Aurora 8B/10B core appends a pad character at the end of the Aurora 8B/10B frame, as required by the protocol. A transfer of 3*n*-1 data bytes requires two full *n*-byte data words and one partial data word. In this example, S_AXI_TX_TKEEP is set to N-1 to indicate *n*-1 valid bytes in the last data word.

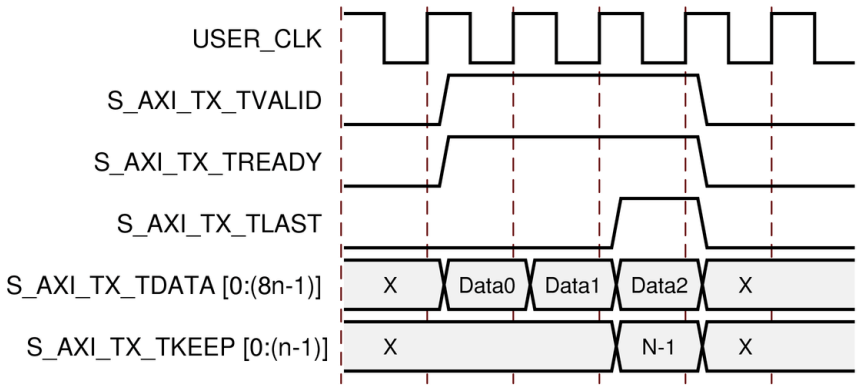

Figure 3-13: **Data Transfer with Pad**

Example C: Data Transfer with Pause

Figure 3-14 shows how a user can pause data transmission during a frame transfer. In this example, the user is sending 3*n* bytes of data, and pauses the data flow after the first

n bytes. After the first data word, the user deasserts S_AXI_TX_TVALID, causing the TX Aurora 8B/10B core to ignore all data on the bus and transmit idles instead. The offset data from the first data word in the previous cycle still is transmitted on lane 0, but the next data word is replaced by idle characters. The pause continues until S_AXI_TX_TVALID is deasserted.

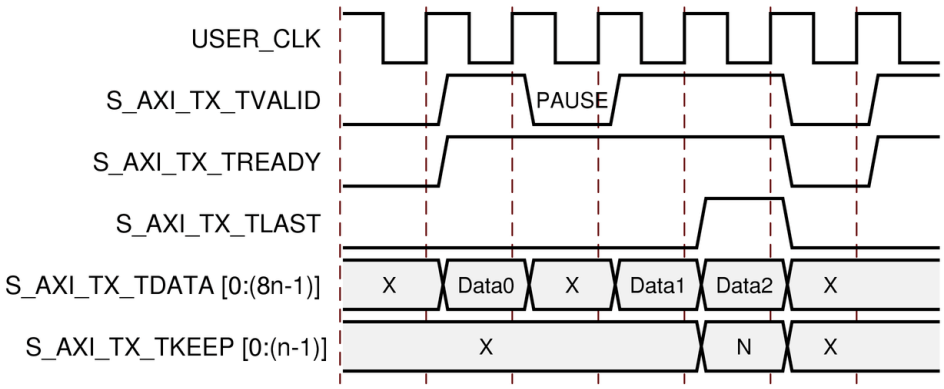

Figure 3-14: **Data Transfer with Pause**

Example D: Data Transfer with Clock Compensation

The Aurora 8B/10B core automatically interrupts data transmission when it sends clock compensation sequences. The clock compensation sequence imposes 12 bytes of overhead per lane every 10,000 bytes.

Figure 3-15 shows how the Aurora 8B/10B core pauses data transmission during the clock compensation (1) sequence.

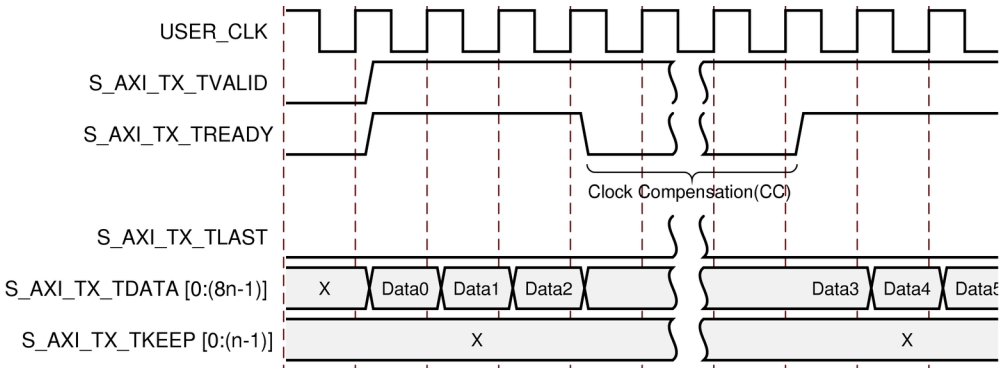

Figure 3-15: **Data Transfer Paused by Clock Compensation**

1. Because of the need for clock compensation every 10,000 bytes per lane (5,000 clocks for 2-byte per lane designs; 2,500 clocks for 4-byte per lane designs), a user cannot continuously transmit data nor can data be continuously received. During clock compensation, data transfer is suspended for six clock periods.

Receiving Data

When the Aurora 8B/10B core receives an Aurora 8B/10B frame, it presents it to the user through the RX AXI4-Stream interface after discarding the framing characters, idles, and clock compensation sequences.

The RX submodules has no built in elastic 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 8B/10B channel is to use one of the core's 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 8B/10B 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 (Low).

M_AXI_RX_TVALID is asserted concurrently with the first word of each frame from the Aurora 8B/10B 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.M_AXI_RX_TKEEP is only valid when M_AXI_RX_TLAST is asserted.

The Aurora 8B/10B core can deassert M_AXI_RX_TVALID anytime, even during a frame. The timing of the M_AXI_RX_TVALID deassertions is independent of the way the data was transmitted. The core can occasionally deassert M_AXI_RX_TVALID even if the frame was originally transmitted without pauses. These pauses are a result of the framing character stripping and left alignment process, as the core attempts to process each frame with as little latency as possible.

Example A: Data Reception with Pause shows the reception of a typical Aurora 8B/10B frame.

Example A: Data Reception with Pause

Figure 3-16 shows an example of 3*n* 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, M_AXI_RX_TVALID is asserted to indicate that data is ready for the user. On the clock cycle following the first data beat, the core deasserts M_AXI_RX_TVALID, indicating to the user 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 based on the total number of valid bytes in the final word of the frame.

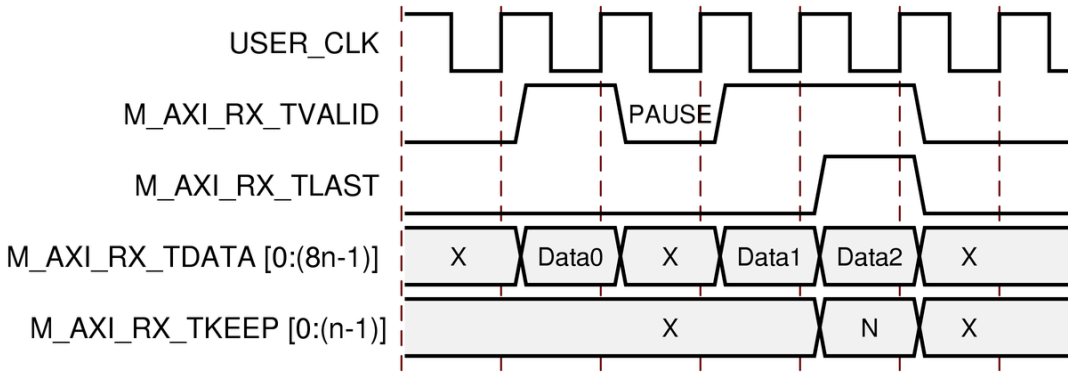

Figure 3-16: **Data Reception with Pause**

Framing Efficiency

There are two factors that affect framing efficiency in the Aurora 8B/10B core:

- Size of the frame
- Width of the datapath

The CC sequence, which uses 12 bytes on every lane every 10,000 bytes, consumes about 0.12% of the total channel bandwidth.

All bytes in Aurora 8B/10B are sent in 2-byte code groups. Aurora 8B/10B frames with an even number of bytes have four bytes of overhead, two bytes for SCP (start of frame) and two bytes for ECP (end of frame). Aurora 8B/10B frames with an odd number of bytes have five bytes of overhead, four bytes of framing overhead plus an additional byte for the pad byte that is sent to fill the second byte of the code group carrying the last byte of data in the frame.

The core transmits frame delimiters only in specific lanes of the channel. SCP is only transmitted in the left-most (most-significant) lane, and ECP is only transmitted in the right-most (least-significant) lane. Any space in the channel between the last code group with data and the ECP code group is padded with idles. The result is reduced resource cost for the design, at the expense of a minimal additional throughput cost. Though SCP and ECP could be optimized for additional throughput, the single frame per cycle limitation imposed by the user interface would make this improvement unusable in most cases.

Use the formula shown in Figure 3-17 to calculate the efficiency for a design of any number of lanes, any width of interface, and frames of any number of bytes.

Note: This formula includes the overhead for clock compensation.

$$E = \frac{100n}{n + 4 + 0.5 + IDLEs + \frac{12n}{9,988}}$$

Where:

E = The average efficiency of a specified PDU
n = Number of user data bytes
12*n*/9,988 = Clock correction overhead
4 = The overhead of SCP + ECP
0.5 = Average PAD overhead
IDLEs = The overhead for IDLEs = (*w*/2)-1
(*w* = The interface width)

Figure 3-17: **Formula for Calculating Overhead**

Example

Table 3-5 is an example calculated from the formula given in Figure 3-17. It shows the efficiency for an 8-byte, 4-lane channel and illustrates that the efficiency increases as the length of channel frames increases.

Table 3-5: **Efficiency Example**

User Data Bytes	Efficiency
100	92.92%
1,000	99.14%
10,000	99.81%

Table 3-6 shows the overhead in an 8-byte, 4-lane channel when transmitting 256 bytes of frame data across the four lanes. The resulting data unit is 264 bytes long due to start and end characters, and due to the idles necessary to fill out the lanes. This amounts to 3.03% of overhead in the transmitter. In addition, a 12-byte clock compensation sequence occurs on each lane every 10,000 bytes, which adds a small amount more to the overhead. The receiver can handle a slightly more efficient data stream because it does not require any idle pattern.

Table 3-6: **Typical Overhead for Transmitting 256 Data Bytes**

Lane	Clock	Function	Character or Data Byte	
			Byte 1	Byte 2
0	1	Start of channel frame	/SCP/$_1$	/SCP/$_2$
1	1	Channel frame data	D0	D1
2	1	Channel frame data	D2	D3
3	1	Channel frame data	D4	D5
. . .				
0	33	Channel frame data	D254	D255
1	33	Transmit idles	/I/	/I/

Table 3-6: **Typical Overhead for Transmitting 256 Data Bytes *(Cont'd)***

Lane	Clock	Function	Character or Data Byte	
			Byte 1	Byte 2
2	33	Transmit idles	/I/	/I/
3	33	End of channel frame	$/ECP/_1$	$/ECP/_2$

Table 3-7 shows the overhead that occurs with each value of S_AXI_TX_TKEEP.

Table 3-7: **S_AXI_TX_TKEEP Value and Corresponding Bytes of Overhead**

S_AXI_TX_TKEEP Bus Value (in Binary)	SCP	Pad	ECP	Idles	Total
1000_0000	2	1	2	6	11
1100_0000		0			10
1110_0000		1		4	9
1111_0000		0			8
1111_1000		1		2	7
1111_1100		0			6
1111_1110		1		0	5
1111_1111		0			4

Streaming Interface

Figure 3-18 shows an example of an Aurora 8B/10B core configured with a streaming user interface.

Figure 3-18: **Aurora 8B/10B Core Streaming User Interface**

Transmitting and Receiving Data

The streaming interface allows the Aurora 8B/10B 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 High). The Aurora 8B/10B core deasserts S_AXI_TX_TREADY (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 GTP/GTX transceiver to make up for frequency differences between the two sides of the Aurora 8B/10B 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 8B/10B 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 is 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 streaming data. The example begins with neither of the ready signals asserted, indicating that both the user logic and the Aurora 8B/10B core are not ready to transfer data. During the next clock cycle, the Aurora 8B/10B 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_TDATA bus and the S_AXI_TX_TVALID signal. Because both ready signals are now asserted, data D0 is transferred from the user logic to the Aurora 8B/10B core. Data D1 is transferred on the following clock cycle. In this example, the Aurora 8B/10B 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 deasserts S_AXI_TX_TVALID on the next clock cycle, and no data is transferred until both ready signals are asserted.

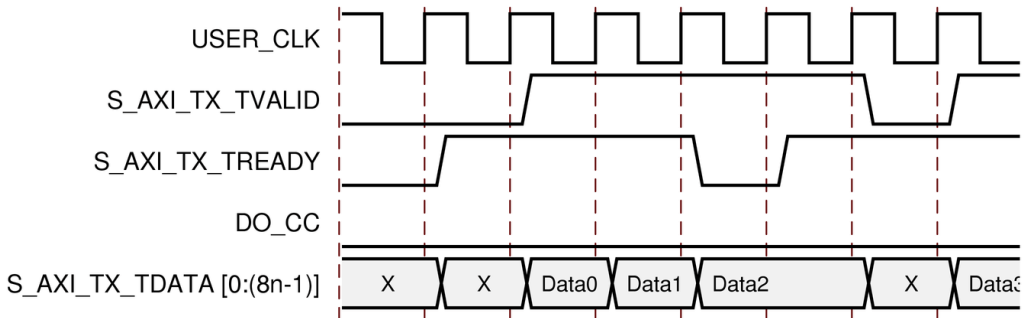

Figure 3-19: **Typical Streaming Data Transfer**

Figure 3-20 shows the receiving end of the data transfer that is shown in Figure 3-19.

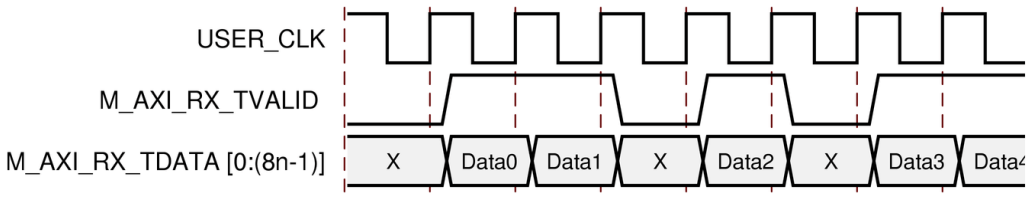

Figure 3-20: **Typical Data Reception**

Flow Control

This section explains how to use Aurora 8B/10B flow control. Two flow control interfaces are available as options on cores that use a framing interface. *Native flow control* (NFC) is used for regulating the data transmission rate at the receiving end a full-duplex channel. *User flow control* (UFC) is used to accommodate high priority messages for control operations.

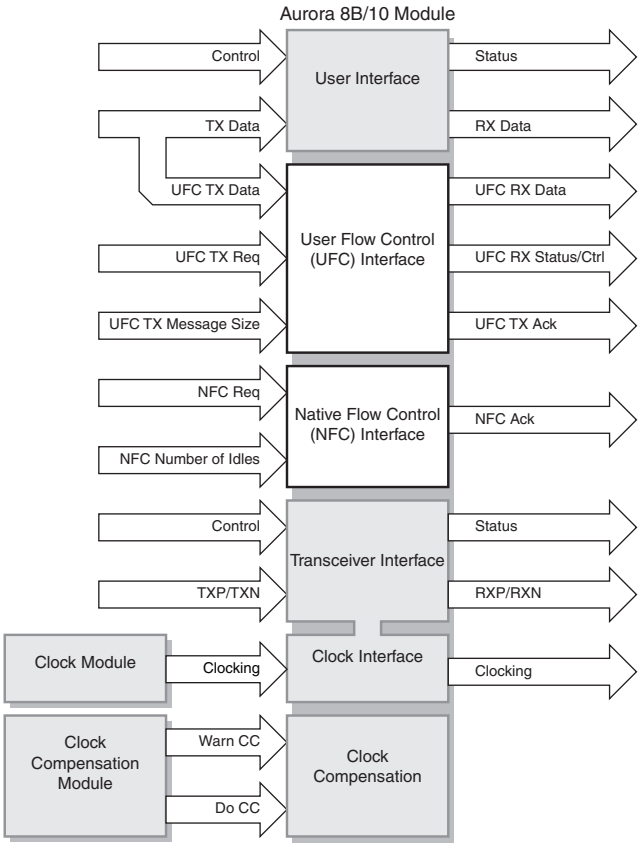

Figure 3-21: **Top-Level Flow Control**

Native Flow Control

Table 3-8 shows the codes for native flow control (NFC).

Table 3-8: **NFC Codes**

S_AXI_NFC_NB	Idle Cycles Requested
0000	0 (XON)
0001	2
0010	4
0011	8

Table 3-8: **NFC Codes**

S_AXI_NFC_NB	Idle Cycles Requested
0100	16
0101	32
0110	64
0111	128
1000	256
1001 to 1110	Reserved
1111	Infinite (XOFF)

The Aurora 8B/10B protocol includes native flow control (NFC) to allow receivers to control the rate at which data is sent to them by specifying several idle data beats that must be placed into the data stream. 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 and NFC codes, see the *Aurora 8B/10B Protocol Specification*.

To send an NFC message to a channel partner, the user application asserts S_AXI_NFC_REQ and writes an NFC code to S_AXI_NFC_NB. The NFC code indicates the minimum number of idle cycles the channel partner should insert in its TX data stream. The user application must hold S_AXI_NFC_REQ and S_AXI_NFC_NB until S_AXI_NFC_ACK is asserted on a positive USER_CLK edge, indicating the Aurora 8B/10B core will transmit the NFC message. Aurora 8B/10B cores cannot transmit data while sending NFC messages. S_AXI_TX_TREADY is always deasserted on the cycle following an S_AXI_NFC_ACK assertion.

Example A: Transmitting an NFC Message

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

Note: S_AXI_TX_TREADY is deasserted for one cycle (assumes that *n* is at least 2) to create the gap in the data flow in which the NFC message is placed.

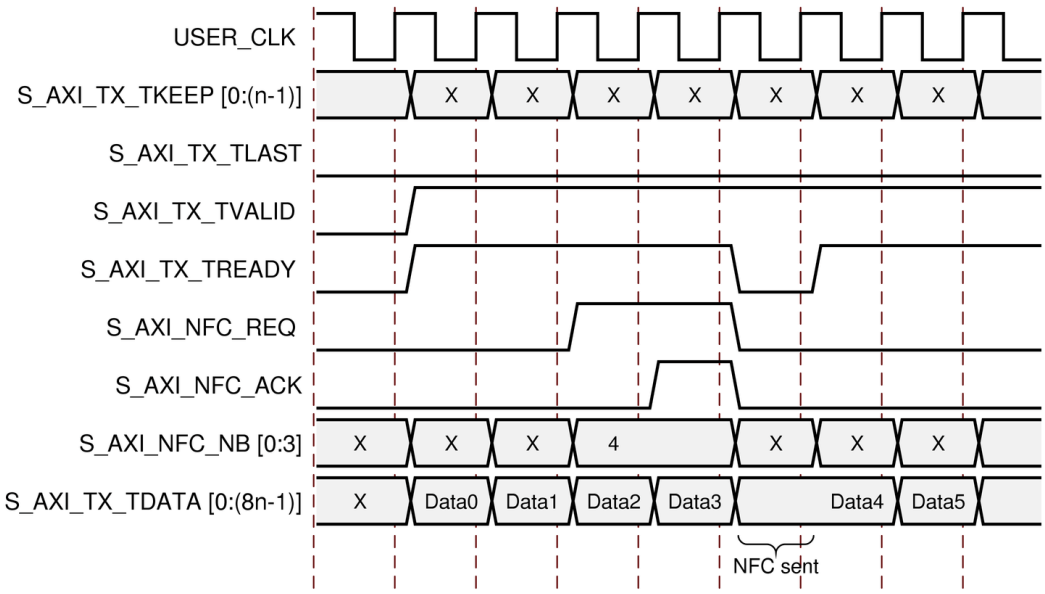

Figure 3-22: **Transmitting an NFC Message**

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 has a code of `0001`, requesting two data beats of idles. The core deasserts `S_AXI_TX_TREADY` on the user interface until enough idles have been sent to satisfy the request. In this example, the core is operating in immediate NFC mode. Aurora 8B/10B cores can also operate in completion mode, where NFC idles are only inserted between frames. 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.

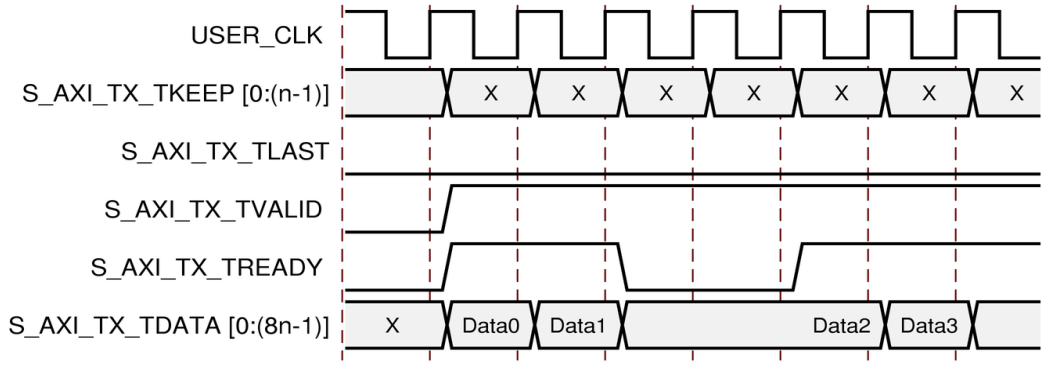

Figure 3-23: **Transmitting a Message with NFC Idles Inserted**

User Flow Control

The Aurora 8B/10B protocol includes user flow control (UFC) to allow channel partners to send control information using a separate in-band channel. The user 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.

Transmitting UFC Messages

UFC messages can carry an even number of data bytes from 2 to 16. The user application specifies the length of the message by driving a SIZE code on the S_AXI_UFC_TX_MS port. Table 3-9 shows the legal SIZE code values for UFC.

Table 3-9: **SIZE Encoding**

SIZE Field Contents	UFC Message Size
000	2 bytes
001	4 bytes
010	6 bytes
011	8 bytes
100	10 bytes
101	12 bytes
110	14 bytes
111	16 bytes

To send a UFC message, the user application asserts S_AXI_UFC_TX_REQ while driving the S_AXI_UFC_TX_MS port with the desired SIZE code. S_AXI_UFC_TX_REQ must be held until the Aurora 8B/10B core asserts the S_AXI_UFC_TX_ACK signal, indicating that the core is ready to send the UFC message. The data for the UFC message must be placed on the S_AXI_TX_TDATA port of the data interface, starting on the first cycle after S_AXI_UFC_TX_ACK is asserted. The core deasserts S_AXI_TX_TREADY while the S_AXI_TX_TDATA port is being used for UFC data.

Figure 3-24 shows a useful circuit for switching TX_D from sending regular data to UFC data.

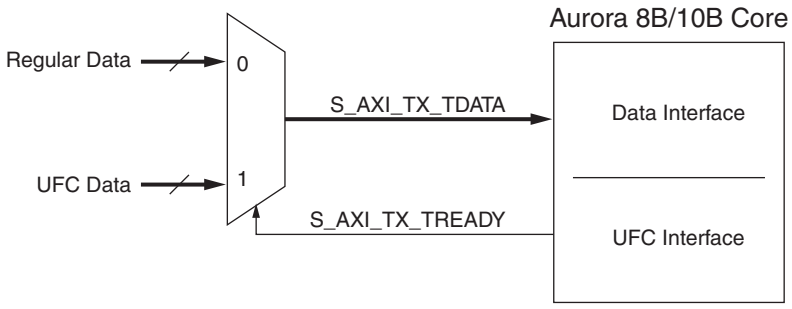

Figure 3-24: **Data Switching Circuit**

Table 3-10, page 53 shows the number of cycles required to transmit UFC messages of different sizes based on the width of the AXI4-Stream data interface. UFC messages should

never be started until all message data is available. Unlike regular data, UFC messages cannot be interrupted after S_AXI_UFC_TX_ACK has been asserted.

Table 3-10: **Number of Data Beats Required to Transmit UFC Messages**

UFC Message	S_AXI_UFC_TX_MS Value	AXI4 Interface Width	Number of Data Beats	AXI4 Interface Width	Number of Data Beats
2 Bytes	0	2 Bytes	1	10 Bytes	1
4 Bytes	1		2		
6 Bytes	2		3		
8 Bytes	3		4		
10 Bytes	4		5		
12 Bytes	5		6		2
14 Bytes	6		7		
16 Bytes	7		8		
2 Bytes	0	4 Bytes	1	12 Bytes	1
4 Bytes	1				
6 Bytes	2		2		
8 Bytes	3				
10 Bytes	4		3		
12 Bytes	5				
14 Bytes	6		4		2
16 Bytes	7				
2 Bytes	0	6 Bytes	1	14 Bytes	1
4 Bytes	1				
6 Bytes	2				
8 Bytes	3		2		
10 Bytes	4				
12 Bytes	5				
14 Bytes	6		3		
16 Bytes	7				2
2 Bytes	0	8 Bytes	1	16 Bytes or more	1
4 Bytes	1				
6 Bytes	2				
8 Bytes	3				
10 Bytes	4		2		
12 Bytes	5				
14 Bytes	6				
16 Bytes	7				

Example A: Transmitting a Single-Cycle UFC Message

The procedure for transmitting a single cycle UFC message is shown in Figure 3-25. In this case, a 4-byte message is being sent on a 4-byte interface.

Note: S_AXI_TX_TREADY is deasserted for two cycles. Aurora 8B/10B cores use this gap in the data flow to transmit the UFC header and message data.

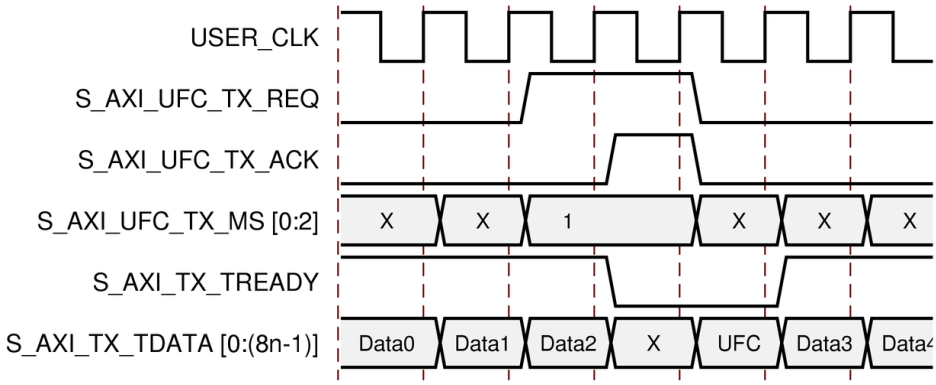

Figure 3-25: **Transmitting a Single-Cycle UFC Message**

Example B: Transmitting a Multicycle UFC Message

The procedure for transmitting a two-cycle UFC message is shown in Figure 3-26. In this case the user application is sending a 4-byte message using a 2-byte interface. S_AXI_TX_TREADY is asserted for three cycles: one cycle for the UFC header which is sent during the S_AXI_UFC_TX_ACK cycle, and two cycles for UFC data.

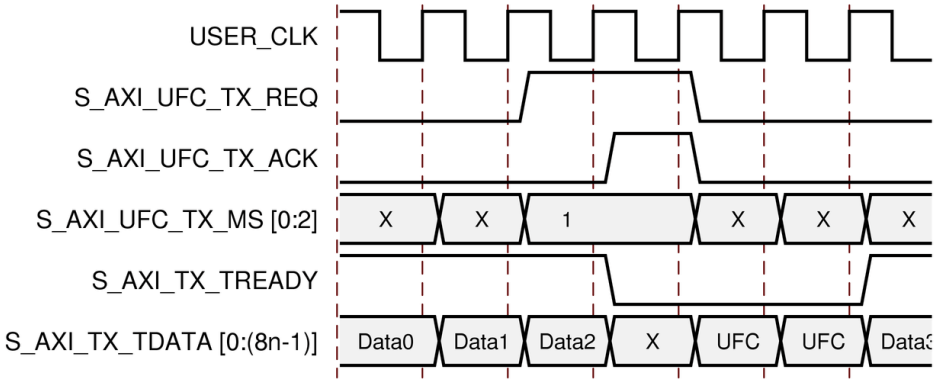

Figure 3-26: **Transmitting a Multicycle UFC Message**

Receiving User Flow Control Messages

When the Aurora 8B/10B 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; 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 M_AXI_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 3-27 shows an Aurora 8B/10B core with a 4-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. M_AXI_UFC_RX_TKEEP is set to 4'hF, indicating only the four most significant bytes of the interface are valid.

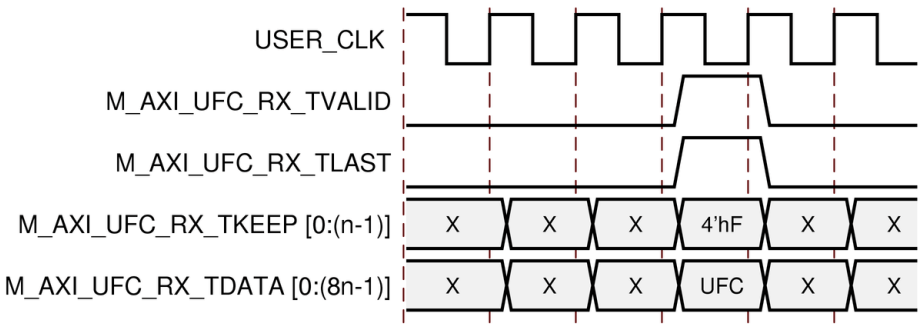

Figure 3-27: **Receiving a Single-Cycle UFC Message**

Example D: Receiving a Multicycle UFC Message

Figure 3-28 shows an Aurora 8B/10B core with a 4-byte interface receiving an 8-byte message.

Note: The resulting frame is two cycles long, with M_AXI_UFC_RX_TKEEP set to 4'hF on the second cycle indicating that all four bytes of the data are valid.

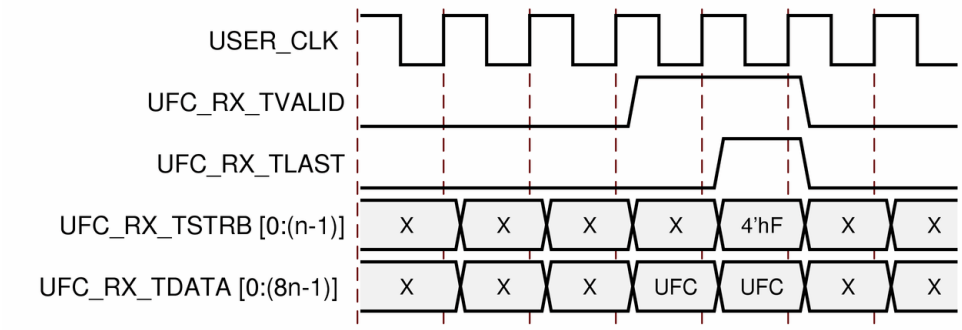

Figure 3-28: **Receiving a Multicycle UFC Message**

Status, Control, and the Transceiver Interface

The status and control ports of the Aurora 8B/10B core allow user applications to monitor the Aurora 8B/10B channel and use built-in features of the GTP/GTX transceivers. Aurora 8B/10B cores can be configured as full-duplex or simplex modules. Full-duplex modules provide high-speed TX and RX links. Simplex modules provide a link in only one direction and are initialized using sideband ports or with a built-in timer. This section provides diagrams and port descriptions for the Aurora 8B/10B core's status and control interface, along with the GTP/GTX transceiver serial I/O interface and the sideband initialization ports that are used exclusively for simplex modules.

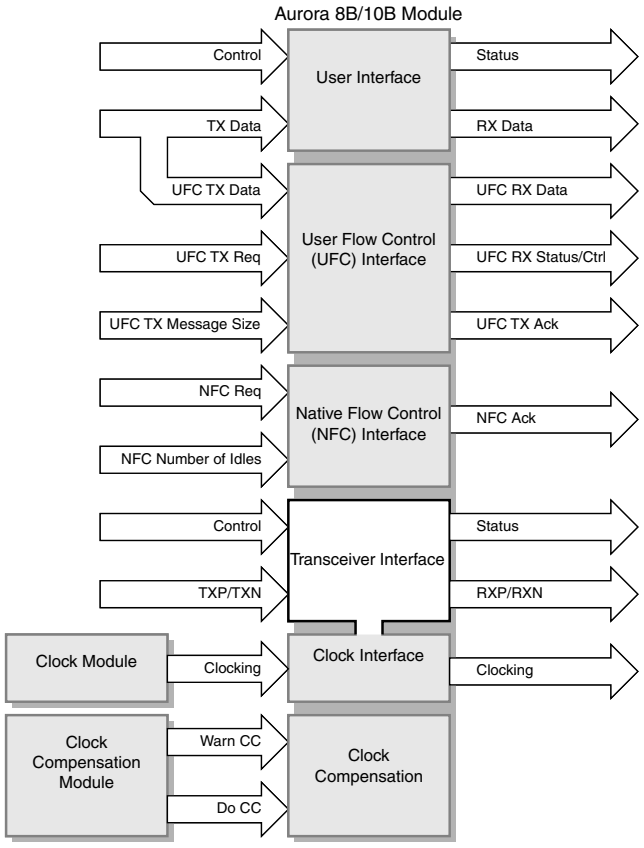

Figure 3-29: **Top-Level Transceiver Interface**

Full-Duplex Cores

Full-Duplex Status and Control Ports

Full-duplex cores provide a TX and an RX Aurora 8B/10B channel connection. Figure 3-30 shows the status and control interface for a full-duplex Aurora 8B/10B core.

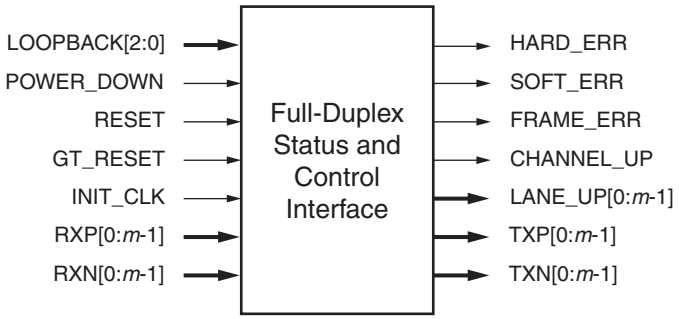

Figure 3-30: **Status and Control Interface for Full-Duplex Cores**

Error Signals in Full-Duplex Cores

Equipment problems and channel noise can cause errors during Aurora 8B/10B channel operation. 8B/10B encoding allows the Aurora 8B/10B core to detect all single bit errors and most multi-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 GTP/GTX transceiver for hardware errors such as buffer overflow/underflow 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 leaky bucket algorithm described in the *Aurora 8B/10B Protocol Specification* to detect large numbers of soft errors occurring in a short period of time, and asserts the HARD_ERR signal when it detects them.

Whenever a hard error is detected, the Aurora 8B/10B core automatically resets itself and attempts to reinitialize. In most cases, this allows the Aurora 8B/10B 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 Aurora 8B/10B leaky bucket algorithm.

Aurora 8B/10B cores with a AXI4-Stream data interface can also detect errors in Aurora 8B/10B frames. Errors of this type include frames with no data, consecutive Start of Frame symbols, and consecutive End of Frame symbols. When the core detects a frame problem, it asserts the `FRAME_ERR` signal. This signal is usually asserted close to a `SOFT_ERR` assertion, with soft errors being the main cause of frame errors.

Table 3-11 summarizes the error conditions the Aurora 8B/10B core can detect and the error signals used to alert the user application.

Table 3-11: **Error Signals in Full-Duplex Cores**

Signal	Description
HARD_ERR	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. RX Overflow/Underflow: The elastic buffer for RX data overflows or underflows. This can occur when the clock source frequencies for the two channel partners are not within ± 100 ppm. Bad Control Character: The protocol engine attempts to send a bad control character. This is an indication of design corruption or catastrophic failure. Soft Errors: There are too many soft errors within a short period of time. The Aurora 8B/10B protocol defines a leaky bucket algorithm for determining the acceptable number of soft errors within a given time period. When this number is exceeded, the physical connection might be too poor for communication using the current voltage swing and pre-emphasis settings.
SOFT_ERR	Invalid Code: The 10-bit code received from the channel partner was not a valid code in the 8B/10B table. This usually means a bit was corrupted in transit, causing a good code to become unrecognizable. Typically, this also results in a frame error or corruption of the current channel frame. Disparity Error: The 10-bit code received from the channel partner did not have the correct disparity. This error is also usually caused by corruption of a good code in transit, and can result in a frame error or bad data if it occurs while a frame is being sent.
FRAME_ERR	Truncated Frame: A channel frame is started without ending the previous channel frame, or a channel frame is ended without being started. Invalid Control Character: The protocol engine receives a control character that it does not recognize. No Data in Frame: A channel frame is received with no data.

Full-Duplex Initialization

Full-duplex cores initialize automatically after power up, reset, or hard error. Full-duplex modules on each side of the channel perform the Aurora 8B/10B 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.

Aurora 8B/10B cores cannot 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.

Simplex Cores

Simplex TX Status and Control Ports

Simplex TX cores allow user applications to transmit data to a simplex RX core. They have no RX connection. Figure 3-31 shows the status and control interface for a simplex TX core.

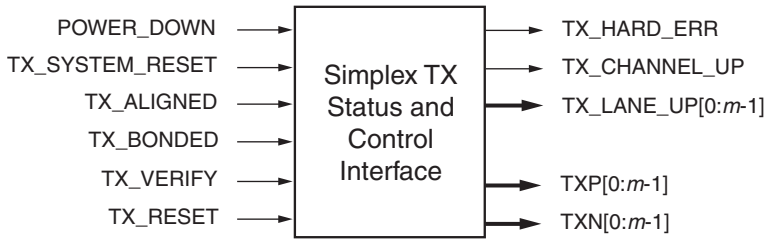

Figure 3-31: **Status and Control Interface for Simplex TX Core**

Simplex RX Status and Control Ports

Simplex RX cores allow user applications to receive data from a simplex TX core. Figure 3-32 shows the status and control interface for a simplex RX core.

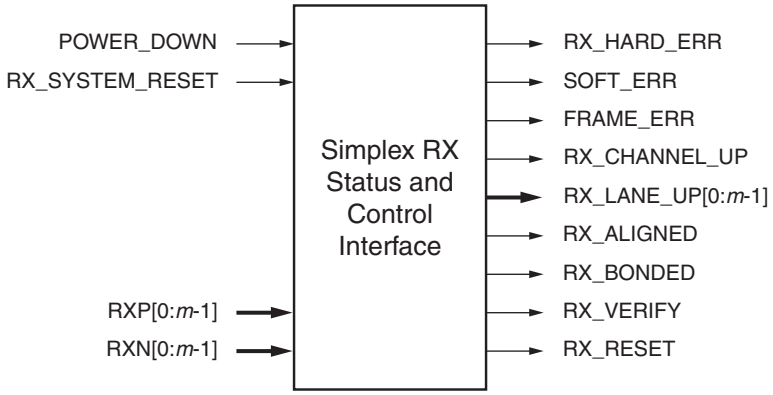

Figure 3-32: **Status and Control Interface for Simplex RX Core**

Error Signals in Simplex Cores

The 8B/10B encoding allows RX simplex cores to detect all single bit errors and most multi-bit errors in a simplex channel. The cores report these errors by asserting the SOFT_ERR signal on every cycle an error is detected. The TX simplex cores do not include a SOFT_ERR port. All transmit data is assumed correct at transmission unless there is an equipment problem.

All simplex cores monitor their GTP/GTX transceivers for hardware errors such as buffer overflow/underflow and loss of lock. Hardware errors on the TX side of the channel are reported by asserting the TX_HARD_ERR signal; RX side hard errors are reported using the RX_HARD_ERR signal. Simplex RX cores use the Aurora 8B/10B protocol's leaky bucket

algorithm to evaluate bursts of soft errors. If too many soft errors occur in a short span of time, RX_HARD_ERR is asserted.

Whenever a hard error is detected, the Aurora 8B/10B core automatically resets itself and attempts to re-initialize. Resetting allows the Aurora 8B/10B channel to be re-established as soon as the hardware issue that caused the hard error is resolved in most cases. Soft errors do not lead to a reset unless enough of them occur in a short period of time to trigger the Aurora 8B/10B leaky bucket algorithm.

Simplex RX cores with a AXI4-Stream data interface can also detect errors in Aurora 8B/10B frames when they are received. Errors of this type include frames with no data, consecutive Start of Frame symbols, and consecutive End of Frame symbols. When the core detects a frame problem, it asserts the FRAME_ERR signal. This signal is usually asserted close to a SOFT_ERR assertion, as soft errors are the main cause of frame errors. Simplex TX modules do not use the FRAME_ERR port.

Table 3-12 summarizes the error conditions simplex Aurora 8B/10B cores can detect and the error signals uses to alert the user application.

Table 3-12: **Error Signals in Simplex Cores**

Signal	Description	TX	RX
HARD_ERR	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.	x	
	RX Overflow/Underflow: The elastic buffer for RX data overflows or underflows. This can occur when the clock source frequencies for the two channel partners are not within ± 100 ppm.		x
	Bad Control Character: The protocol engine attempts to send a bad control character. This is an indication of design corruption or catastrophic failure.	x	
	Soft Errors: There are too many soft errors within a short period of time. The Aurora 8B/10B protocol defines a leaky bucket algorithm for determining the acceptable number of soft errors within a given time period. When this number is exceeded, the physical connection might be too poor for communication using the current voltage swing and pre-emphasis settings.		x
SOFT_ERR	Invalid Code: The 10-bit code received from the channel partner was not a valid code in the 8B/10B table. This usually means a bit was corrupted in transit, causing a good code to become unrecognizable. Typically, this also results in a frame error or corruption of the current channel frame.		x
	Disparity Error: The 10-bit code received from the channel partner did not have the correct disparity. This error is also usually caused by corruption of a good code in transit, and can result in a frame error or bad data if it occurs while a frame is being sent.		x
	No Data in Frame: A channel frame is received with no data.		x

Table 3-12: **Error Signals in Simplex Cores *(Cont'd)***

Signal	Description	TX	RX
FRAME_ERR	Truncated Frame: A channel frame is started without ending the previous channel frame, or a channel frame is ended without being started.	x	
	Invalid Control Character: The protocol engine receives a control character that it does not recognize.		x
	Invalid UFC Message Length: A UFC message is received with an invalid length.		x

Simplex Initialization

Simplex cores do not depend on signals from an Aurora 8B/10B channel for initialization. Instead, the TX and RX sides of simplex channels communicate their initialization state through a set of sideband initialization signals. The initialization ports are called ALIGNED, BONDED, VERIFY, and RESET; one set for the TX side with a TX_ prefix, and one set for the RX side with an RX_ prefix. The bonded port is only used for multi-lane cores.

There are two ways to initialize a simplex module using the sideband initialization signals:

- Send the information from the RX sideband initialization ports to the TX sideband initialization ports
- Drive the TX sideband initialization ports independently of the RX sideband initialization ports using timed initialization intervals

Both initialization methods are described in the following sections.

Using a Back Channel

If there is no communication channel available from the RX side of the connection to the TX side, using a back channel is the safest way to initialize and maintain a simplex channel. There are very few requirements on the back channel; it need only deliver messages to the TX side to indicate which of the sideband initialization signals is asserted when the signals change.

The Aurora example design included in the example_design directory with simplex Aurora 8B/10B cores shows a simple side channel that uses three or four I/O pins on the device.

Using Timers

For some systems a back channel is not possible. In these cases, serial channels can be initialized by driving the TX simplex initialization with a set of timers. The timers must be designed carefully to meet the needs of the system because the average time for initialization depends on many channel specific conditions such as clock rate, channel latency, skew between lanes, and noise. C_ALIGNED_TIMER, C_BONDED_TIMER, and C_VERIFY_TIMER are timers used for assertion of TX_ALIGNED, TX_BONDED, and TX_VERIFY signals, respectively. These timers use worst-case values obtained from corner case functional simulations and implemented in the <component name> module.

Some of the initialization logic in the Aurora 8B/10B module uses watchdog timers to prevent deadlock. These watchdog timers are used on the RX side of the channel, and can interfere with the proper operation of TX initialization timers. If the RX simplex module goes from ALIGNED, BONDED or VERIFY, to RESET, make sure that it is not because the TX logic spend too much time in one of those states. If a particularly long timer is required to meet the needs of the system, the watchdog timers can be adjusted by editing the lane_init_sm module and the channel_init_sm module. For most cases, this should not be necessary and is not recommended.

Aurora 8B/10B channels normally re-initialize only in the case of failure. When there is no back channel available, event-triggered re-initialization is impossible for most errors because it is usually the RX side that detects a failure and the TX side that must handle it. The solution for this problem is to make timer-driven TX simplex modules re-initialize on a regular basis. If a catastrophic error occurs, the channel is reset and running again after the next re-initialization period arrives. System designers should balance the average time required for re-initialization against the maximum time their system can tolerate an inoperative channel to determine the optimum re-initialization period for their systems.

Reset and Power Down

Reset

The reset signals on the control and status interface are used to set the Aurora 8B/10B 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`. On simplex modules, the resets for the TX and RX channels are separate. `TX_SYSTEM_RESET` resets TX channels; `RX_SYSTEM_RESET` resets RX channels. The `TX_SYSTEM_RESET` is separate from the `TX_RESET` and `RX_RESET` signals used on the simplex sideband interface.

Power Down

This is an active-High signal. When `POWER_DOWN` is asserted, the GTP/GTX transceivers in the Aurora 8B/10B 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 the Clock Interface and Clocking, page 28). `TX_OUT_CLK` stops when the GTP/GTX transceivers are powered down. See the *7 Series FPGAs GTX Transceivers User Guide*, the *Virtex-6 FPGA GTX Transceivers User Guide*, or the *Spartan-6 FPGA GTP Transceivers User Guide* for the device being used for details about powering down GTP/GTX transceivers.

Timing

Figure 3-33 shows the timing for the RESET and POWER_DOWN signals. In a quiet environment, t_{CU} is generally less than 800 clocks; in a noisy environment, t_{CU} can be much longer.

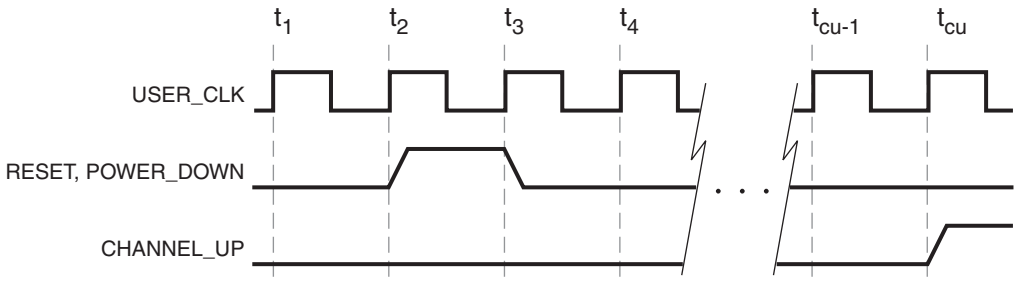

Figure 3-33: **Reset and Power Down Timing**

Core Features

Using the Scrambler/Descrambler

A 16-bit additive scrambler/descrambler, implemented for data, is available in the `<component name>_scrambler.v[hd]` module. It ensures non-occurrence of repetitive data over long periods of time. The scrambler and descrambler are synchronized based on reception of the clock compensation characters. DO_CC must be transmitted to load the seed value of the scrambler and descrambler simultaneously. Thus, the `standard_cc_module` that comes with the Aurora example design should always be used, if the **Use Scrambler/Descrambler** option is selected in the GUI.

Using CRC

A 16-bit or 32-bit CRC, implemented for user data, is available in the `<component name>_crc_top.v[hd]` module. CRC16 is generated for 2-byte designs, and CRC32 is generated for 4-byte designs. The `CRC_VALID` and `CRC_PASS_FAIL_N` signals indicate the result of a received CRC with a transmitted CRC (see Table 4-1).

Table 4-1: **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 Analyzer Cores

The ICON and VIO cores in the ChipScope™ Pro Analyzer help to debug and validate the design in boards. These cores are provided with the Aurora 8B/10B core. Select the **Use ChipScope Pro Analyzer** checkbox from the core GUI to include it as a part of the example

design. Alternatively, the USE_CHIPSCOPE parameter in the <component name>_exdes module can be set to 1 before running implementation.

Hot-Plug Logic

Hot-plug logic in Aurora 8B/10B designs with Virtex®-7, Kintex™-7, and Virtex-6 FPGAs is based on the received clock compensation characters. 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 8B/10B designs with Virtex-7, Kintex-7, and Virtex-6 FPGAs. To disable hot-plug logic, set the ENABLE_HOTPLUG parameter to 0 in the <component name>/<component name>_hotplug.v[hd] module. Hot-plug logic then does not repeatedly reset the core when looking for clock compensation characters in the received data.

Customizing and Generating the Core

This chapter includes information on using Vivado™ Design Suite tools to customize and generate the core.

GUI

The Aurora 8B/10B 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 8B/10B IP Customizer is presented when you select the Aurora 8B/10B core in the Vivado IP Catalog. Each numbered item in Figure 5-1 corresponds to its respective section that describes the purpose of the feature.

IP Customizer

Figure 5-1 shows the customizer. The left side displays a representative block diagram of the Aurora 8B/10B core as currently configured. The right side consists of user-configurable parameters.

The second page of the GUI is shown in Figure 5-2, page 68 for Virtex®-7/Kintex™-7 FPGA GTX/GTH transceivers.

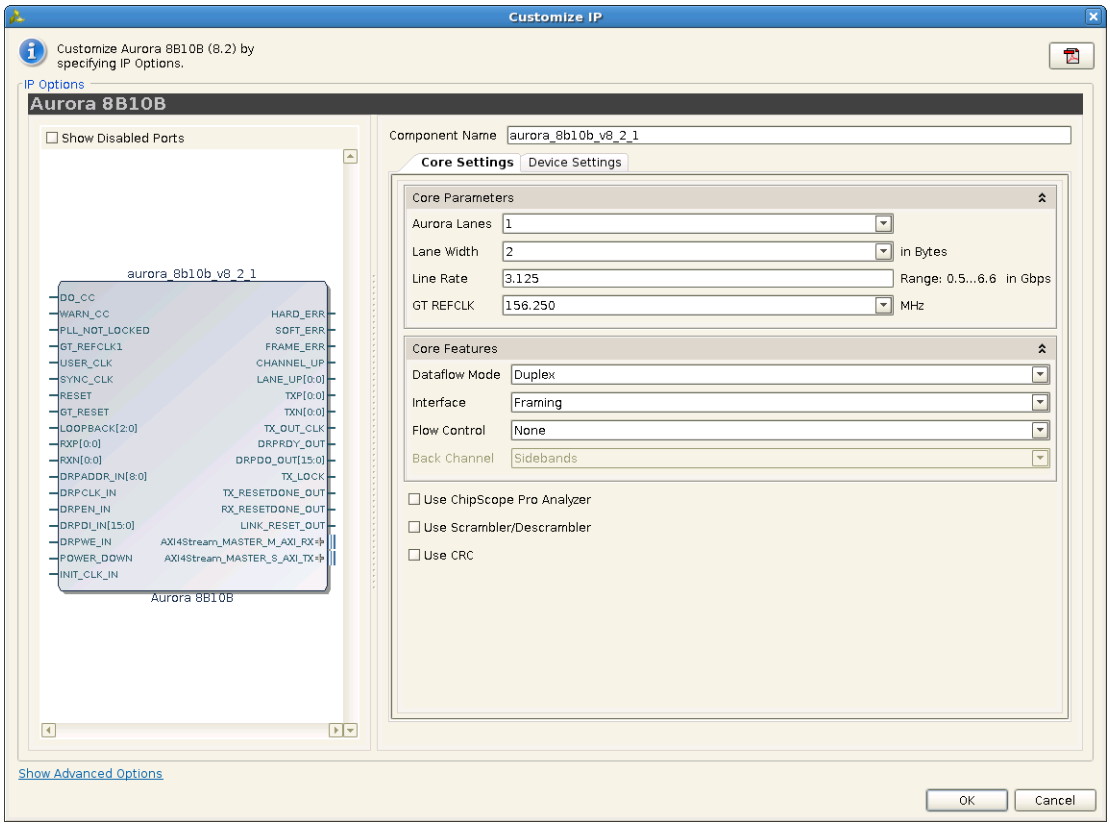

Figure 5-1: **Aurora 8B/10B IP Customizer**

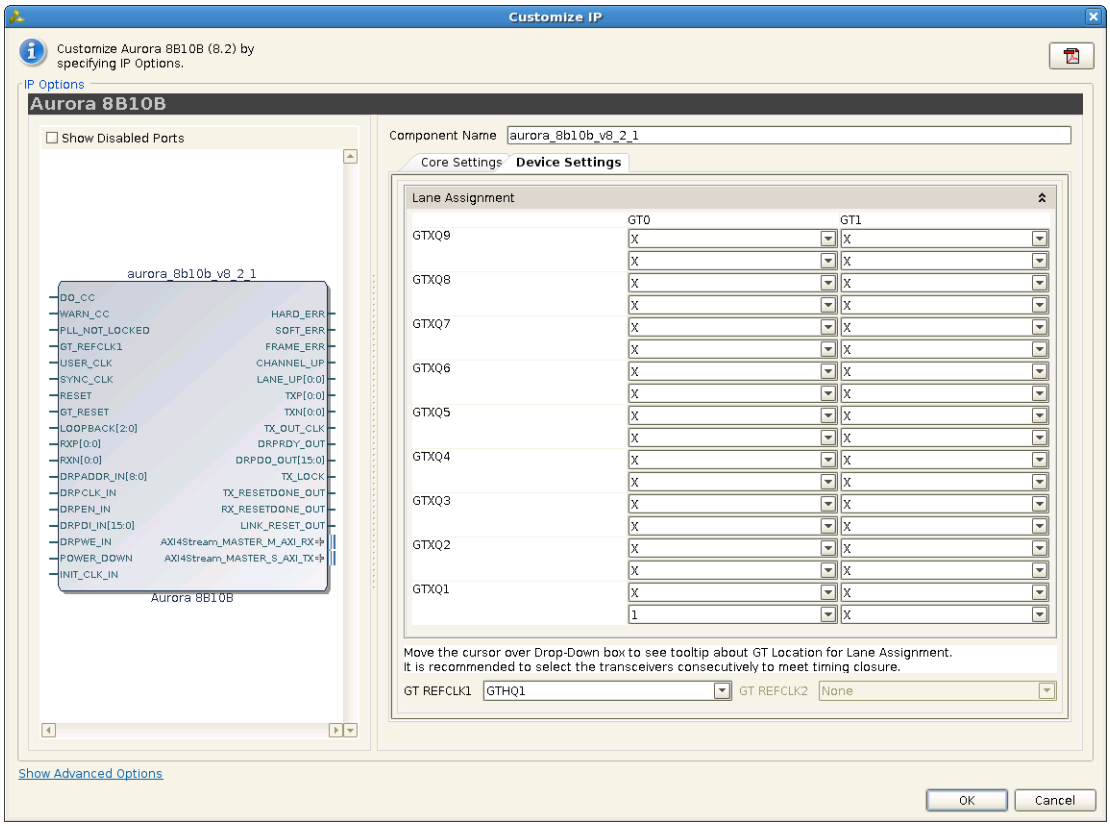

Figure 5-2: **Second GUI Page for Virtex-7/Kintex-7 FPGA GTX/GTH Transceivers**

Component Name

Enter the top-level name for the core in this text box. Illegal names are highlighted in red until they are corrected. The core uses `<Component Name>_exdes` as the top-level module.

Default: aurora_8b10b_v8_2

Lane Assignment

Refer to the diagram in the information area in Figure 5-2. Two rows or four boxes represent a GTX/GTH Quad in Virtex-7 and Kintex-7 FPGAs. Each active box represents an available GTX or GTH transceiver.

Aurora Lanes

Select the number of lanes (GTX/GTH transceivers) to be used in the core. The valid range is from 1 to 16 and depends on the target device selected.

Default: 1

Lane Width

Select the byte width of the GTX/GTH transceivers used in the core. This parameter defines the TXDATA/RXDATA width of the transceiver and the user interface data bus width as well. Valid values are 2 and 4.

Default: 2

Interface

Select the type of datapath interface used for the core. Select Framing to use an 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 8B/10B channel. See User Interface, page 37 for more information.

Default: Framing

Dataflow Mode

Select the options for direction of the channel the Aurora 8B/10B core supports. Simplex Aurora 8B/10B cores have a single, unidirectional serial port that connects to a complementary simplex Aurora 8B/10B core. Available options are *RX-only Simplex*, *TX-only Simplex*, and *Duplex*. See Status, Control, and the Transceiver Interface, page 56 for more information.

Default: Duplex

Back Channel

Select the options for Back Channel only for Simplex Aurora cores; Duplex Aurora cores do not require this option. The available options are:

- Sidebands
- Timer

Default: Sidebands

Note: There is no functionality difference between RX-only Simplex design with Sidebands option and RX-only Simplex design with Timer option.

Flow Control

Select the required option to add flow control to the core. User flow control (UFC) allows applications to send a brief, high-priority message through the Aurora 8B/10B channel. Native flow control (NFC) allows full duplex receivers to regulate the rate of the data send 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 listed below (see Flow Control, page 69 for more information):

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

Default: None

Line Rate

Enter a floating-point value in gigabits per second within the valid range. This determines the unencoded bit rate at which data is transferred over the serial link. The aggregate data rate of the core is (0.8 x line rate) x Aurora 8B/10B lanes.

Default: 3.125 Gbps

GT REFCLK (MHz)

Select a reference clock frequency for the transceiver from the drop-down list. Reference clock frequencies are given in megahertz (MHz), 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

GT REFCLK1 and GT REFCLK2

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

- Default: GT REFCLK Source1 - GTXQ0; GT REFCLK Source2 - None for Virtex-7/Kintex-7 FPGA GTX transceivers
- Default: GT REFCLK Source1 - GTHQ0; GT REFCLK Source2 - None for Virtex-7/Kintex-7 FPGA GTH transceivers
- GTXQ0 and GTHQ0 change based on the selected device and package.

Column Used

Select the appropriate column of transceivers used from the drop-down list. The column used is enabled only for Virtex-7 and Kintex-7 devices and is disabled for all other devices.

Default: left

Use Scrambler/Descrambler

Select to include the 16-bit additive scrambler/descrambler to the Aurora 8B/10B design. See Using the Scrambler/Descrambler in Chapter 4 for more information.

Default: Unchecked

Use CRC

Select to include the CRC for user data. See Using CRC in Chapter 4 for more information.

Default: Unchecked

Use ChipScope Pro Analyzer

Select to add ChipScope™ Pro cores to the Aurora 8B/10B core. This option provides a debugging interface that shows the core status signals in the ChipScope Pro analyzer tool.

Default: Unchecked

Generate

Click Generate to generate the core. The modules for the Aurora 8B/10B core are written to the Vivado tool project directory using the same name as the top level of the core. See Output Generation, page 71 for details about the example_design directory and files.

Output Generation

The customized Aurora 8B/10B core is delivered as a set of HDL source modules in the language selected in the Vivado™ 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 Vivado 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>
Top-level core directory; contains the core deliverables and the readme file

<component name>/doc

Product documentation

- <component name>/example_design
  Example design and user constraints files
  - /example_design/cc_manager
    Verilog/VHDL design file for the clock compensation block
  - /example_design/clock_module
    Verilog/VHDL design file for the clocking blocks
  - /example_design/gt
    Verilog/VHDL wrapper files for the GTP/GTX transceivers
  - /example_design/traffic_gen_check
    Verilog/VHDL design files for the frame generator and checker
- <component name>/implement
  Implementation scripts and support files
- <component name>/simulation
  Simulation test bench and simulation script files
  - /simulation/functional
    Functional simulation files
  - /simulation/timing
    Timing simulation file
- <component name>/src
  Verilog/VHDL files for the core

Directory and File Contents

The Aurora 8B/10B core directories and their associated files are defined in the following sections.

<project directory>

The `project` directory contains the Vivado tool project files.

Table 5-1: **project Directory**

Name	Description
<project directory>	
<component name>.v[hd]	Aurora core top-level module

Back to Top

<project directory>/<component name>

This top-level core directory contains the core deliverables and the readme file.

Table 5-2: **component name Directory**

Name	Description
<project directory>/<component name>	
aurora_8b10b_v8_2_readme.txt	Release notes file

Back to Top

<component name>/doc

The doc directory contains the product documentation.

Table 5-3: **doc Directory**

Name	Description
<component name>/doc	
aurora_8b10b_v8_2_vinfo.html	Version information file

Back to Top

<component name>/example_design

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

Table 5-4: **example_design Directory**

Name	Description
<component name>/example_design	
<component name>_exdes.v[hd]	Example design top-level file
<component name>_exdes.xdc	Aurora 8B/10B example design XDC constraints
<component name>_reset_logic.v[hd]	Aurora 8B/10B reset logic
v7_icon.ngc v7_vio.ngc	Virtex-7/Kintex-7 FPGA NGC files for the debug cores compatible with the ChipScope™ Pro Analyzer tool

Back to Top

/example_design/cc_manager

The cc_manager directory contains the clock compensation source file.

Table 5-5: **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 5-6: **clock_module Directory**

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

Back to Top

/example_design/gt

The gt directory contains the Verilog/VHDL wrapper files for the GTP/GTX transceivers.

Table 5-7: **gt Directory**

Name	Description
<component name>/example_design/gt	
<component name>_gtx.v[hd] <component name>_transceiver_wrapper.v[hd]	Verilog/VHDL wrapper files for the transceiver

Back to Top

/example_design/traffic_gen_check

The traffic_gen_check directory contains frame generator and frame checker modules for Aurora 8B/10B core.

Table 5-8: **traffic_gen_check Directory**

Name	Description
<component name>/example_design/traffic_gen_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 5-9: **implement Directory**

Name	Description
<component name>/implement	
synplify_pro.prj implement_synplify.sh implement_synplify.bat	Synplify Pro script files for Aurora 8B/10B example design

Back to Top

<component name>/simulation

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

Table 5-10: **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 5-11: **functional Directory**

Name	Description
<component name>/simulation/functional	
simulate_isim.bat	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 Microsoft Windows operating system.
simulate_isim.sh	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.
wave_isim.tcl	ISim macro file that opens a Wave window with top-level signals.

Table 5-11: **functional Directory**

Name	Description
simulate_mti.do simulate_mti.sh simulate_mti.bat	ModelSim macro files that compile the example design sources and the structural simulation model. The demonstration test bench then runs the functional simulation to completion.
wave_mti.do	ModelSim macro file that opens a Wave window.
simulate_ncsim.sh simulate_ncsim.bat wave_ncsim.sv	Cadence IES simulator scripts that run functional simulation of the example design.

Back to Top

/simulation/timing

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

Table 5-12: **timing Directory**

Name	Description
<component name>/simulation/timing	
simulate_mti.do simulate_mti.sh simulate_mti.bat	ModelSim macro files that compile the post place and route netlist of the example design along with standard delay format (SDF) back annotation then runs timing simulation to completion

Back to Top

<component name>/src

The src directory contains the source files related to the Aurora 8B/10B example design.

Table 5-13: **src Directory**

Name	Description
<component name>/src	
<component name>_aurora_lane.v[hd] <component name>_aurora_pkg.vhd (VHDL Only) <component name>_axi_to_ll.v[hd] <component name>_channel_err_detect.v[hd] <component name>_channel_init_sm.v[hd] <component name>_chbond_count_dec.v[hd] <component name>_err_detect.v[hd] <component name>_global_logic.v[hd] <component name>_idle_and_ver_gen.v[hd] <component name>_lane_init_sm.v[hd] <component name>_ll_to_axi.v[hd] <component name>_rx_ll.v[hd] <component name>_rx_ll_pdu_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.v[hd] <component name>_tx_ll_datapath.v[hd]	Aurora 8B/10B source files

Back to Top

Constraining the Core

Design Constraints

Aurora 8B/10B example design constraints can be grouped into any of these six categories:

1. GT reference clock constraint

   The Aurora 8B/10B 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 XGUI). The GT REFCLK value selected in the first page of the XGUI is used to constrain the GT reference clock. The create_clock XDC command is used to constrain GT reference clocks.

2. TXOUTCLK clock constraint

   TXOUTCLK is generated by the GT transceiver based on the applied reference clock and the divider settings of the GT transceiver. The Aurora 8B/10B core calculates the TXOUTCLK frequency based on the line rate and lane width. The create_clock XDC command is used to constrain TXOUTCLK.

3. GT reference clock pins constraints

   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.

4. System clock constraint

   The Aurora 8B/10B example design uses a debounce circuit to sample GT_RESET 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.

5. GT location constraint

   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.

6. 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.

The generated example design is configured with a 2-byte lane width, 6.6 Gb/s line rate, and a 660.0 MHz reference clock. The XDC file generated for the XC7VX690T-FFG1761-2 device is given below:

```
-----------------------
## XDC generated for xc7vx690t-ffg1761-2 device
# 660.0MHz GT Reference clock constraint
create_clock -name GT_REFCLK1 -period 1.515 [get_pins IBUFDS_GTE2_CLK1/O]

######################## GT reference clock LOC ########################
set_property LOC AW9 [get_ports GTHQ1_N]
set_property LOC AW10 [get_ports GTHQ1_P]

# TXOUTCLK Constraint: Value is selected based on the line rate (6.6 Gbps) and lane
width (2-Byte)
create_clock -name tx_out_clk_i -period 3.03 [get_pins aurora_module_i/
gt_wrapper_i/GTE2_INST/gthe2_i/TXOUTCLK]

# USER_CLK Constraint : Value is selected based on the line rate (6.6 Gbps) and lane
width (2-Byte)
create_clock -name user_clk_i -period 3.03 [get_pins clock_module_i/user_clk_buf_i/
O]

# 50 MHz Board Clock Constraint
create_clock -name init_clk_i -period 20.000 [get_pins reset_logic_i/
init_clk_ibufg_i/O]

###### No cross clock domain analysis. Domains are not related ##############
set_false_path -from [get_clocks init_clk_i] -to [get_clocks user_clk_i]
set_false_path -from [get_clocks user_clk_i] -to [get_clocks init_clk_i]
set_false_path -from [get_clocks init_clk_i] -to [get_clocks tx_out_clk_i]
set_false_path -from [get_clocks tx_out_clk_i] -to [get_clocks init_clk_i]


############################### GT LOC #####################################
set_property LOC GTHE2_CHANNEL_X1Y4 [get_cells aurora_module_i/gt_wrapper_i/
GTE2_INST/gthe2_i]

----------------
```

Detailed Example Design

Directory and File Contents

See Output Generation, page 71 for the directory structure and file contents of the example design.

Example Design

Each Aurora 8B/10B 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 71.

Figure 7-1 illustrates the block diagram of the example design for a full-duplex core. Table 7-1 describes the ports of the example design.

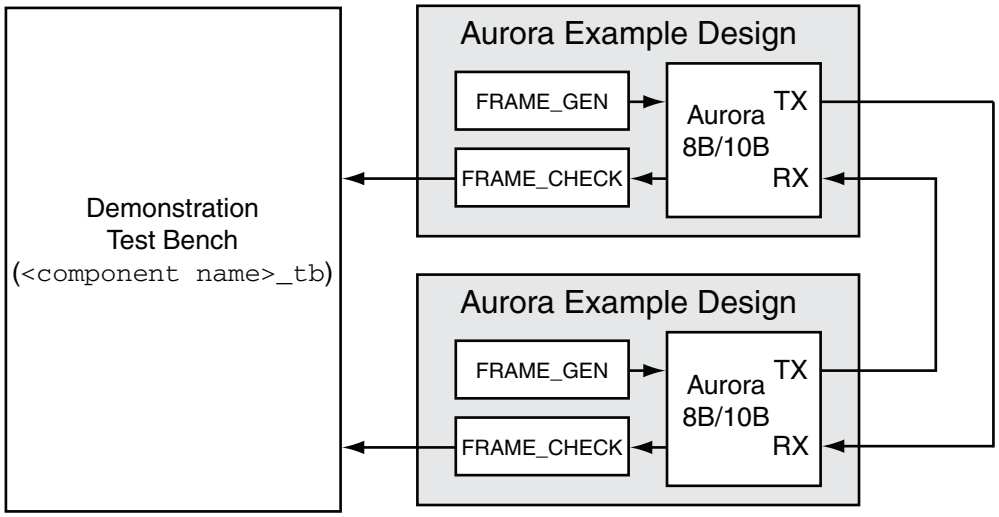

Figure 7-1: **Example Design**

The example designs uses all the interfaces of the core. Simplex cores without a TX or RX interface have no FRAME_GEN or FRAME_CHECK block, respectively. The frame generator produces a constant stream of data for cores with a streaming interface.

Using the scripts provided in the `implement` subdirectory, the example design can be used to quickly get an Aurora 8B/10B 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 the connecting the trickier interfaces on the Aurora 8B/10B core, such as the clocking interface.

When using the example design on a board, be sure to edit the `<component name>_exdes.xdc` file in the example_design subdirectory to supply the correct pins and clock constraints.

Table 7-1: **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.
ERR_COUNT[0:7]	Output	Count of the number of data words received by the frame checker that did not match the expected value.
RESET	Input	Reset signal for the example design. The reset is debounced using a USER_CLK signal generated from the reference clock input.
<reference clock(s)>	Input	The reference clocks for the Aurora 8B/10B core are brought to the top level of the example design. See Clock Interface and Clocking, page 28 for details about the reference clocks.
<core error signals>	Output	The error signals from the Aurora 8B/10B core's Status and Control interface are brought to the top level of the example design and registered. See Status, Control, and the Transceiver Interface, page 56 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. Full-duplex cores have a single channel up signal; simplex cores have one for each channel direction supported. See Status, Control, and the Transceiver Interface, page 56 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 GTP/GTX transceiver they use. Simplex cores have a separate lane up signal per GTP/GTX transceiver they use for each channel direction supported. See Status, Control, and the Transceiver Interface, page 56 for details.
<simplex initialization signals>	Input/ Output	If the core is a simplex core, its sideband initialization ports are registered and brought to the top level of the example design. See Status, Control, and the Transceiver Interface, page 56 for details.

Implementation

Overview

The quick start example consists of the following components:

- An instance of the Aurora 8B/10B core generated using the default parameters
  - Full-duplex with a single GTP/GTX transceiver
  - AXI4-Stream interface
- A demonstration test bench to simulate two instances of the example design

The Aurora 8B/10B example design has been tested with the Vivado™ Design Suite for synthesis and Mentor Graphics ModelSim for simulation.

XILINX®

Appendix A

Verification, Compliance, and Interoperability

Aurora 8B/10B 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/verification of the core features.

The Aurora 8B/10B core is verified using the Aurora 8B/10B Bus Functional Model (BFM) and proprietary custom test benches. The Aurora 8B/10B 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 8B/10B 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.

The test board used for verification is KC724.

Simulation

The Aurora 8B/10B 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 A-1. Refer to the *Synthesis and Verification Design Guide* for the Vivado tool for instructions on how to compile simulation libraries.

Table A-1: **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 8B/10B core provides a command line script to simulate the example design. To run a VHDL or Verilog ModelSim simulation of the Aurora 8B/10B core, use the following instructions:

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

   `<project directory>/aurora_8b10b_v8_2/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 8B/10B core power up, followed by Aurora 8B/10B channel initialization and data transfer. Data transfer begins after the `CHANNEL_UP` signal goes High.

Because cores are generated one at a time, simulating simplex cores requires additional steps. To simulate a simplex TX or simplex RX core, perform these steps:

1. Generate the core for simulation.
2. Generate a complementary simplex core. Go to the implement directory of the first core generated.
3. Set the environment variable SIMPLEX_PARTNER to point to the directory for the complementary core.
4. Run the script according to the instructions in this section.

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 `simplex_201_tx`, then the top-level module name of the simplex partner should be `rx_simplex_201_tx`.

Appendix B

Migrating

Introduction

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

For information on migrating to the Vivado™ Design Suite, see UG911, *Vivado Design Suite Migration Methodology Guide* [Ref 6].

Prerequisites

- Vivado v2012.2 tool build containing the Aurora 8B/10B v8.2 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 (PG046) of the core with the AXI4-Stream updates
- Legacy data sheet (DS637) and data sheet (UG353) for reference
- Migration guide (this appendix)

Limitations

This section outlines the limitations of the Aurora 8B/10B core for AXI4-Stream support. The user has to take care of two limitations while interfacing the Aurora 8B/10B core with the AXI4-Stream compliant interface core:

1. The Aurora 8B/10B core supports only continuous aligned streams and continuous unaligned streams. The position bytes are valid only at the end of packet. In other words, TKEEP is sampled only at TLAST assertion.
2. The AXI4-Stream protocol supports transfers with zero data at the end of packet, but the Aurora 8B/10B core expects at least one byte to be valid at the end of packet. In other words, TKEEP should contain a non-zero value during TLAST assertion.

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 conversions of signals between AXI4-Stream and LocalLink
  - Generation of SOF_N and REM bits mapping is 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 has 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 Diagram

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.

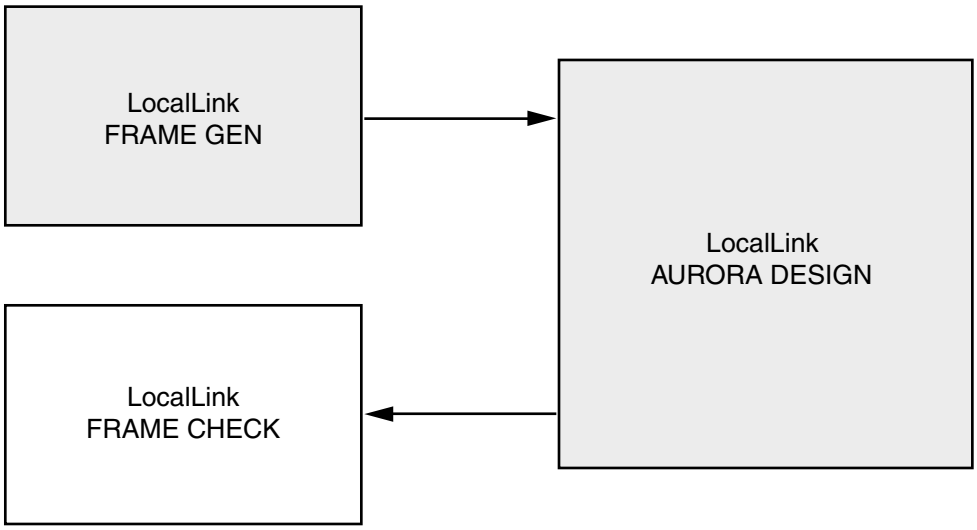

Figure B-1: **Legacy Aurora Example Design**

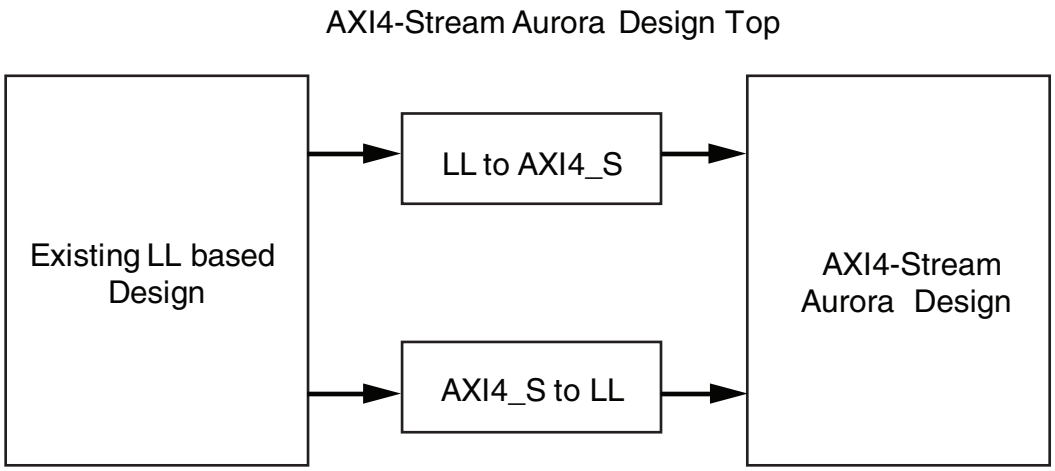

Figure B-2: **AXI4-Stream Aurora Example Design**

Migration Steps

Generate an AXI4-Stream Aurora core from the Vivado Design Suite v2012.2.

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, runs through the Vivado 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 light-weight 'shim' to interface to any existing LL based interface. The shims are delivered along with the core from the aurora_8b10b_v8_2 version of the core.
2. See Figure B-2, page 87 for the emulation of an LL Aurora core from an AXI4-Stream Aurora core.
3. Two shims `<component name>_ll_to_axi.v[hd]` and `<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 `<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 87.
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.

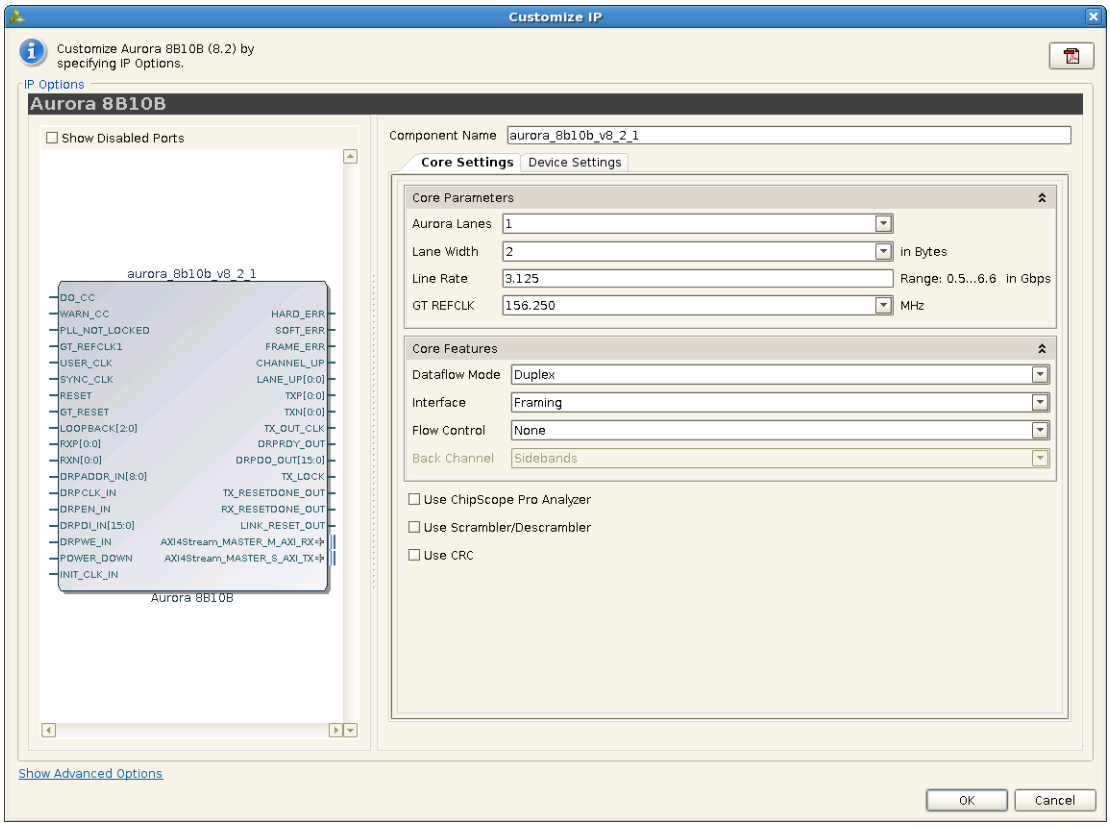

Figure B-3: **AXI4-Stream Signals**

Debugging

This appendix provides some useful debugging tips.

See Additional Resources in Appendix F for additional information helpful to the debugging progress.

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 GT instances that is not working.
- Make sure that the GT reference clock and user clocks are all toggling.
- Check that TXOUTCLK from the GT wrapper is toggling. If it is not toggling, you might have to wait longer for the PMA to finish locking. You should typically wait about 6 µs to 9 µs for lane up and channel up. The wait time is longer for simplex designs.
- Make sure that TXN and TXP are toggling. If they are not, make sure that your wait time is long enough and that you are not driving the TX signal with another signal.
- Check the `PLL_NOT_LOCKED` signal on your design. If it is held active-High, your Aurora module cannot initialize.
- Be sure the POWER_DOWN signal is not asserted.
- Make sure the TXN and TXP signals from each GT transceiver are connected to the appropriate RXN and RXP signals from the corresponding GT transceiver on the other side of the channel.
- When simulating Verilog, you 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. Hold this reset for a few cycles.

  The following code provides 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 GT transceivers on each side of the channel are connected in the correct order.

Channel Comes Up in Simulation But Not in Hardware

- Make sure the REFCLK, INIT CLK, RESETs (GT transceiver reset and Aurora reset), and transceivers are constrained exactly matching to the hardware connections in the XDC file.
- Both RESET inputs are active-Low. Make sure the hardware takes care of the RESET polarity.
- Make sure the REFCLK frequency is exactly the same as for what the Aurora core is generated.
- If the REFCLK is driven from a synthesizer, make sure the synthesizer is stable (locked).
- Make sure the cable connection from TXP/TXN to RXP/RXN is proper.
- If there are RXNOTINTABLE errors observed from the GT transceiver, validate the link using IBERT. Make sure there is no BER in the channel. Use the sweep test in the IBERT tool and use the same GT transceiver attributes that provide "Zero" bit error rate (BER) in IBERT.

Channel Comes Up in Simulation but S_AXI_TX_TVALID 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 driven Low. `S_AXI_NFC_REQ` and `S_AXI_UFC_TX_REQ` are active-High. If they are High, `S_AXI_TX_TVALID` stays Low because the channel is allocated for flow control.
- Make sure `WARN_CC` and `DO_CC` are not being driven High continuously. When `DO_CC` is High on a positive clock edge, the channel is used to send clock correction characters, and S_AXI_TX_TVALID is deasserted.
- If NFC is enabled, make sure the design on the other side of the channel does not send an NFC XOFF message. This cuts off the channel for normal data until the other side sends an NFC XON message to turn the flow on again.

Bytes and Words Are Lost While Traveling Through the Aurora Channel

- If the AXI4-Stream interface is used, make sure data is written correctly. The most common mistake is to assume words are written without looking at TVALID. Also remember that the TKEEP signal must be used to indicate which bytes are valid when TLAST is asserted. TKEEP is ignored when TLAST is not asserted (active-High).

- Make sure you are reading correctly from the Rx interface. Data and framing signals are only valid when TVALID is asserted.

Problems Occur While Compiling the Design

- Be sure to include all files from the `src` directory when compiling.
- When using VHDL, include the `aurora_pkg.vhd` file in your synthesis.

XILINX®

Appendix D

Generating a Wrapper File from the Transceiver Wizard

The transceiver attributes play a vital role in the functionality of the Aurora 8B/10B core. Use the latest Transceiver Wizard to generate the transceiver wrapper file.

Xilinx strongly recommends that you update the transceiver wrapper file in 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

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

1. Using the Vivado 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 8B/10B core.
2. Select the protocol template from the following based on the number of lane(s) and lane width:
   - Aurora 8B/10B single lane 2 byte
   - Aurora 8B/10B single lane 4 byte
   - Aurora 8B/10B multi lane 2 byte
   - Aurora 8B/10B multi lane 4 byte
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. Keep all other settings as default.
7. Generate the core.
8. Replace the `<component name>_gt.v[hd]` file in the `example_design/gt` directory available in the Aurora 8B/10B core with the generated `<component name>_gt.v[hd]` file generated from the 7 Series FPGAs Transceivers Wizard.

The transceiver settings for the Aurora 8B/10B core are up to date now.

Case 2: Virtex-6 FPGA GTX Wrapper

Use these steps to generate the transceiver wrapper file using the Virtex®-6 FPGA GTX Transceiver Wizard.

1. Using the Vivado IP catalog, run the latest version of the Virtex-6 FPGA GTX Transceiver Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 8B/10B core.
2. Select the protocol template from the following based on number of lane(s) and lane width:
   - Aurora 2-byte single lane
   - Aurora 4-byte single lane
   - Aurora 2-byte multi-lane
   - Aurora 4-byte multi-lane
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. Keep all other settings as default.
7. Generate the core.
8. Replace the `<component name>_gtx.v[hd]` file in the `example_design/gt` directory available in the Aurora 8B/10B core with the generated `<component name>_gtx.v[hd]` file generated from the Virtex-6 FPGA GTX Transceiver Wizard.

The transceiver settings for the Aurora 8B/10B core are up to date now.

Case 3: Spartan-6 FPGA GTP Wrapper

Use these steps to generate the transceiver wrapper file using the Spartan®-6 FPGA GTP Transceiver Wizard.

1. Using the Vivado IP catalog, run the latest version of Spartan-6 FPGA GTP Transceiver Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 8B/10B core.
2. Select transceiver(s) and the clock source(s) based on application requirement.
3. Select the protocol template to either aurora single lane or aurora multi lane based on the number of lane(s).
4. Change the Target Line Rate in Gbps based on application requirement.
5. Select the Reference Clock from the drop-down box menu based on application requirement.
6. Select RXCHARISCOMMA and RXCHARISK ports in 8B/10B Optional Ports, if not selected by default.
7. Select TXBUFSTATUS port in Synchronization and Clocking, if not selected by default.
8. Keep all other settings as default.
9. Generate the core.
10. Replace the `<component name>_tile.v[hd]` file in the `example_design/gt` directory available in the Aurora 8B/10B core with the generated `<component name>_tile.v[hd]` file generated from the Spartan-6 FPGA GTP Transceiver Wizard.

The transceiver tile settings for the Aurora 8B/10B core are up to date now.

Appendix E

Handling Timing Errors

This appendix describes how to handle timing errors resulting from transceivers that are far apart.

The Aurora 8B/10B core allows the user to select any combination of transceiver(s) during core generation. The design parameters that affect the timing performance are:

- Line rate
- Transceiver datapath width (2/4 bytes) and
- Number of unused transceivers between two selected transceivers

As a result of one or more of these parameters, timing errors can occur because:

- CHBONDO does not meet timing
- RXCHARISCOMMA, RXCHARISK, and RXCHANISALIGNED do not meet timing

The following suggestions can be attempted to meet timing:

1. Select the transceivers consecutively.

   Use the Lane Assignment in the Aurora 8B/10B GUI to select the transceivers during core generation.

   Note: Most of the timing errors are due to unused transceivers and channel bonding signals connections among transceivers.

2. Use the Vivado™ tool Strategies options provided for implementation in the Vivado™ Design Suite.

   See the Vivado tool documentation [Ref 6] for instructions on how to use Vivado Strategies.

Appendix F

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.

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.

See the Aurora Solutions Center for support specific to the Aurora 8B/10B core.

References

For detailed information and updates about the Aurora core, see the following document, located on the Aurora product page at www.xilinx.com/aurora:

- UG058, *Aurora 8B/10B Bus Functional Model User Guide*
  (Contact: auroramkt@xilinx.com)

These documents provide supplemental material useful with this product guide. Users should be familiar with these documents prior to generating an Aurora 8B/10B core:

1. SP002, *Aurora 8B/10B Protocol Specification*
2. AMBA AXI4-Stream Protocol Specification

3. UG476, *7 Series FPGAs GTX/GTH Transceivers User Guide*
4. UG366, *Virtex-6 FPGA GTX Transceivers User Guide*
5. UG386, *Spartan-6 FPGA GTP Transceivers User Guide*
6. Vivado™ tool documentation: www.xilinx.com/cgi-bin/docs/rdoc?v=2012.2;t=vivado+docs

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
07/25/12	1.0	Initial Xilinx release. This release supports core version 8.2 with Vivado Design Suite v2012.2. This document replaces UG766 and DS797.

Notice of Disclaimer

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.