LogiCORE IP Aurora 8B/10B v10.0

Product Guide for Vivado Design Suite

PG046 October 2, 2013

Table of Contents

Introduction

The Xilinx 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 Zynq®-7000 All Programmable SoCs, Virtex®-7, Kintex®-7, and Artix®-7 families.

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 and Kintex-7 FPGA GTX and GTH transceivers, 8 of 16 Artix-7 FPGA GTP transceivers
- Aurora 8B/10B protocol specification v2.2 compliant
- Low resource cost (see Resource Utilization, page 11)
- 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 Cyclic Redundancy Check (CRC) for user data
- Hot-plug logic

LogiCORE IP Facts Table					
Core Specifics					
Supported Device Family[1]	Zynq-7000, Virtex-7, Kintex-7, Artix-7				
Supported User Interfaces	AXI4-Stream				
Resources[2]	**LUTs**	**FFs**	**DSP Slices**	**Block RAMs**	**Max. Frequency**[3]
Config1	342	463	0	0	330 MHz
Provided with Core					
Design Files	RTL				
Example Design	Verilog and VHDL				
Test Bench	Verilog and VHDL				
Constraints File	Xilinx Design Constraints (XDC)				
Simulation Model	Not Provided				
Supported S/W Driver	N/A				
Tested Design Flows[4]					
Design Entry	Vivado® Design Suite Vivado IP Integrator				
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.				
Synthesis	Vivado Synthesis				
Support					
Provided by Xilinx @ www.xilinx.com/support					

1. For a complete list of supported devices, see the Vivado IP catalog.
2. For device performance numbers, see Table 2-1 through Table 2-4.
3. For more complete performance data, see Performance, page 10.
4. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

This guide describes how to generate a LogiCORE™ IP Aurora 8B/10B core using Virtex®-7, Kintex®-7 FPGA GTX and GTH transceivers and Artix®-7 GTP transceivers, The core implements the Aurora 8B/10B protocol using the high-speed serial transceivers on the Virtex-7 and Kintex-7 families (including lower power), The Aurora 8B/10B v10.0 core supports the AMBA® protocol AXI4-Stream user interface.

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

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. 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).

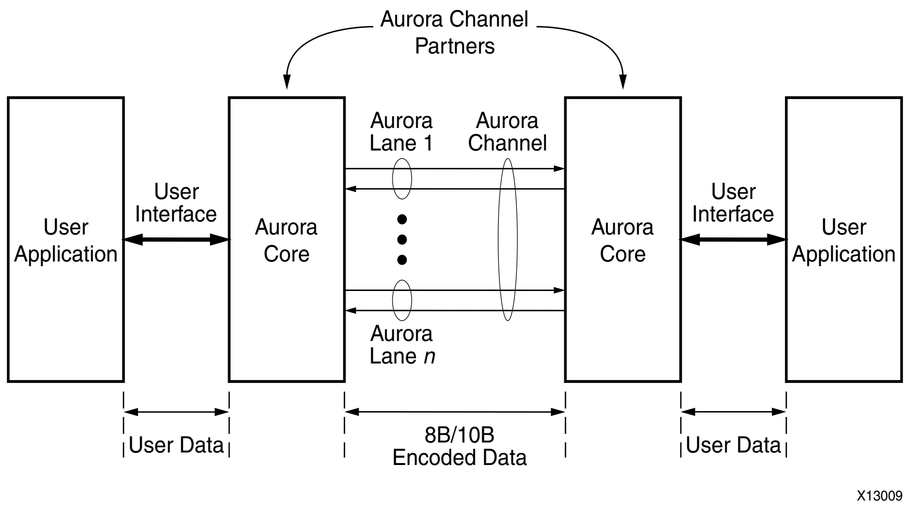


Figure 1-1: **Aurora 8B/10B Channel Overview**

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.

RECOMMENDED: *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 constraints files (XDC) with the Vivado Design Suite is recommended. Read Status, Control, and the Transceiver Interface, carefully.*

Consult the PCB design requirements information in:

- *7 Series FPGAs GTP Transceivers User Guide* (UG482) [Ref 1]
- *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476) [Ref 2]

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

Feature Summary

The Aurora 8B/10B core is a scalable, lightweight, link-layer protocol for high-speed serial communication. The 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* (SP002) [Ref 3]. 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 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, and GTH transceivers, with minimal FPGA resource cost.
- **Board-to-board and backplane links**: The 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, and 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 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.

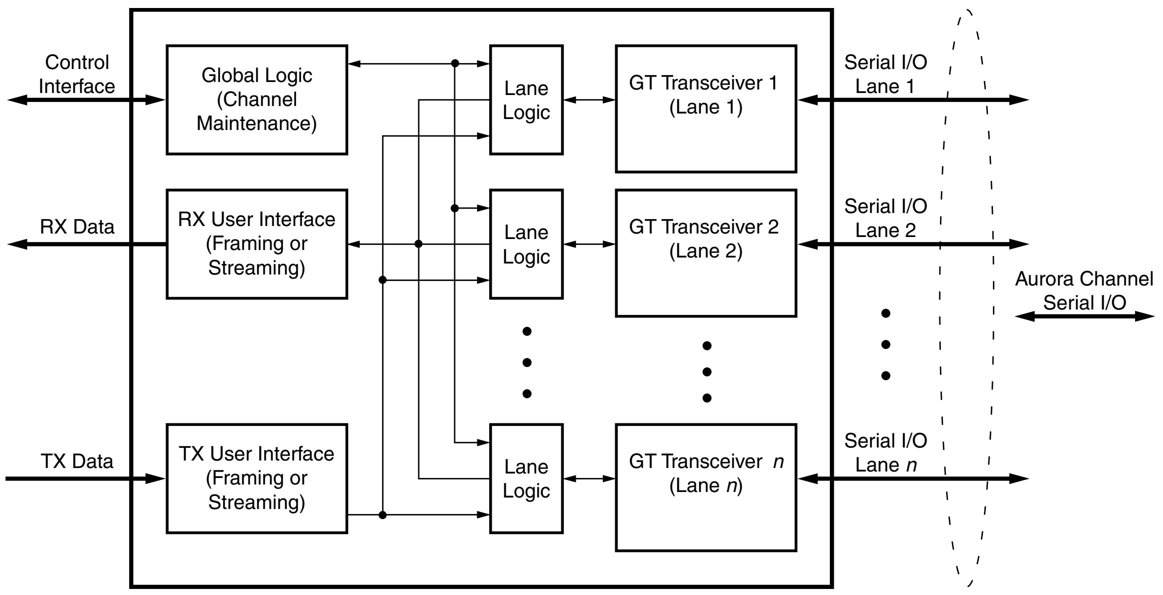

Control Interface
RX Data
TX Data
Global Logic (Channel Maintenance)
RX User Interface (Framing or Streaming)
TX User Interface (Framing or Streaming)
Lane Logic
Lane Logic
Lane Logic
GT Transceiver 1 (Lane 1)
GT Transceiver 2 (Lane 2)
GT Transceiver n (Lane n)
Serial I/O Lane 1
Serial I/O Lane 2
Serial I/O Lane n
Aurora Channel Serial I/O

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, or GTH transceiver is driven by an instance of the lane logic module, which initializes each individual GTP, GTX, or 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

The Aurora 8B/10B core is compliant with the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 3].

Performance

Maximum Frequencies

Config1 cited in the LogiCORE™ IP Facts table on page 5 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 a 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 12 through Table 2-4, page 13 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, or GTH transceivers. The PE pipeline delay increases as the AXI4-Stream interface width increases. The GTP, GTX, or GTH transceiver delays are dependent on the features and attributes of the selected GTP, GTX, or GTH transceivers.

This section outlines expected latency for the Aurora 8B/10B core 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 are partitioned into GTP, GTX, or GTH transceiver logic and protocol engine (PE) logic is implemented in the FPGA logic.

Note: These numbers 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. This latency information is provided for a Virtex-7 VX690T-FFG1761 device with –2 speed grade. Latency can vary based on the transceiver(s) used in the design.

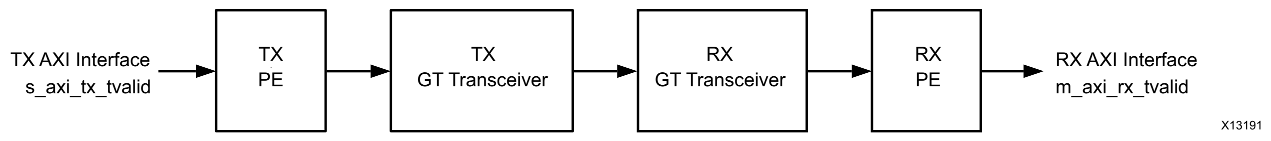


Figure 2-2: **Latency of the Frame Path**

Minimum latency for a two-byte framing design from `s_axi_tx_tvalid` to `m_axi_rx_tvalid` is approximately 37 `user_clk` cycles in functional simulation.

Minimum latency for a four-byte framing design from `s_axi_tx_tvalid` to `m_axi_rx_tvalid` is approximately 41 `user_clk` cycles in functional simulation.

Maximum latency varies based on the IP configuration.

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 0.5 Gb/s to 84.48 Gb/s for a single lane design to a 16 lane design, respectively. The throughput was calculated using 25% overhead of the 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 2-4 show the number of look-up tables (LUTs) and flip-flops (FFs) used in selected Aurora core configurations in the Vivado® design tools.

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/scrambler/crc. 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 and Virtex-7 Lower Power Kintex-7 and Kintex-7 Lower Power Families		Streaming		
		Duplex	Simplex	
Lanes	Resource Type	Full Duplex	TX-Only Simplex	RX-Only Simplex
1	LUTs	379	166	236
	FFs	582	275	355
2	LUTs	520	210	324
	FFs	798	329	526
4	LUTs	760	316	470
	FFs	1189	433	805
8	LUTs	1258	478	757
	FFs	1970	656	1361
16	LUTs	2229	841	1345
	FFs	3534	1092	2473

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

Virtex-7 and Virtex-7 Lower Power Kintex-7 and Kintex-7 Lower Power Families		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full Duplex	TX-Only Simplex	RX-Only Simplex
1	LUTs	388	163	244
	FFs	596	273	371
2	LUTs	553	213	356
	FFs	843	329	572
4	LUTs	827	297	530
	FFs	1271	438	885
8	LUTs	1374	475	867
	FFs	2145	662	1507
16	LUTs	2448	903	1545
	FFs	3907	1153	2785

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

Virtex-7 and Virtex-7 Lower Power Kintex-7 and Kintex-7 Lower Power Families		Streaming		
		Duplex	Simplex	
Lanes	**Resource Type**	**Full Duplex**	**TX-Only Simplex**	**RX-Only Simplex**
1	LUTs	447	182	277
	FFs	651	285	415
2	LUTs	684	251	434
	FFs	964	367	652
4	LUTs	1091	376	687
	FFs	1536	530	1057
8	LUTs	1877	625	1169
	FFs	2678	852	1865
16	LUTs	3471	1124	2148
	FFs	4962	1496	3481

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

Virtex-7 and Virtex-7 Lower Power Kintex-7 and Kintex-7 Lower Power Families		Framing		
		Duplex	Simplex	
Lanes	**Resource Type**	**Full Duplex**	**TX-Only Simplex**	**RX-Only Simplex**
1	LUTs	490	186	309
	FFs	695	283	461
2	LUTs	750	259	488
	FFs	1044	368	732
4	LUTs	1210	398	795
	FFs	1707	532	1203
8	LUTs	2108	680	1382
	FFs	3043	905	2177
16	LUTs	3859	1212	2545
	FFs	5369	1605	3922

Port Descriptions

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

- User Interface
- User Flow Control Interface
- Native Flow Control Interface
- Transceiver Interface
- Clock Interface
- Clock Compensation Interface

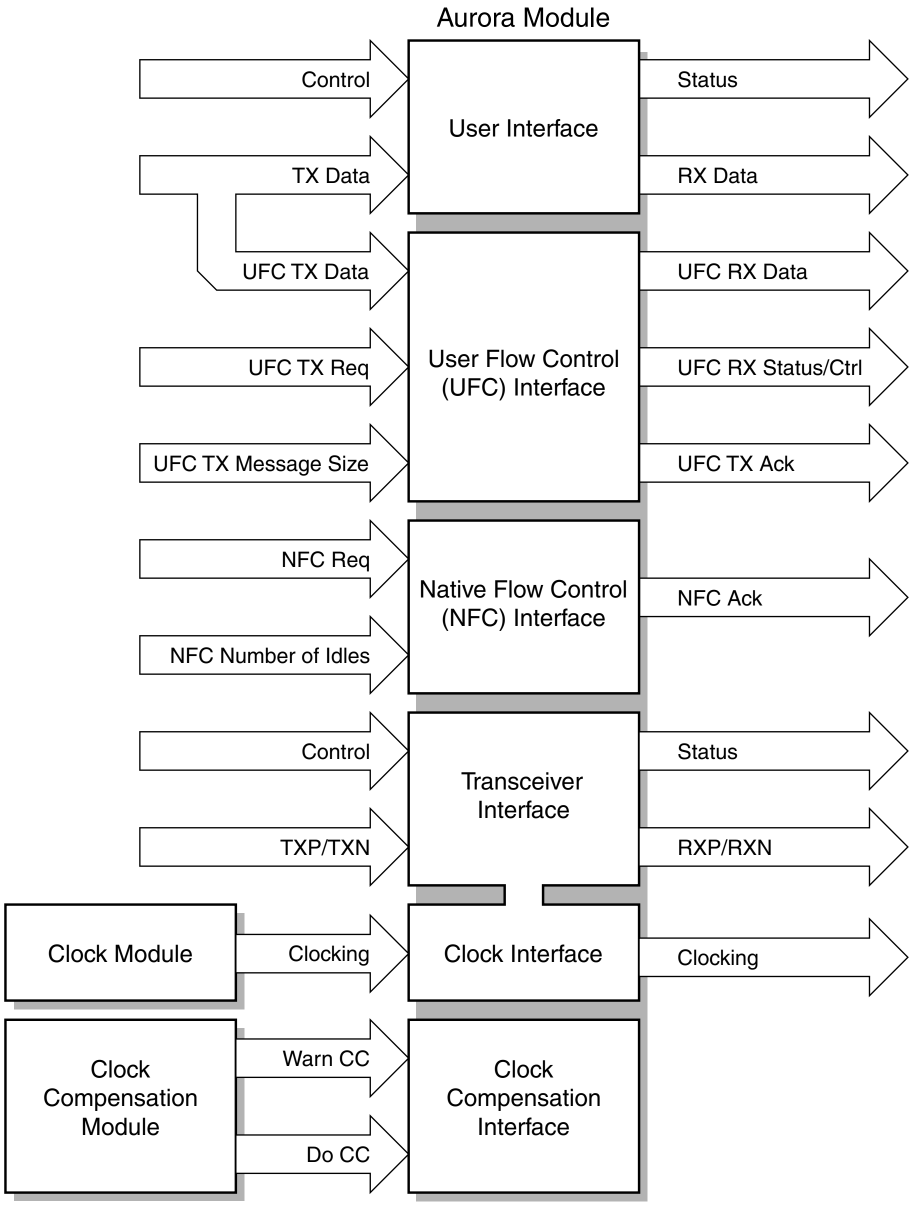


Figure 2-3: **Top-Level Interface**

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 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 or GTX transceivers in the core, and on the width selected for these transceivers.

Framing Interface Ports

Table 2-5 lists port descriptions for AXI4-Stream TX data ports. These ports are included on full-duplex and simplex TX framing cores.

Table 2-5: **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 2-6 lists port descriptions for Framing RX data ports. These ports are included on full-duplex and simplex RX framing cores.

Table 2-6: **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 45 for more information.

Streaming Interface Ports

Table 2-7 lists the streaming TX data ports. These ports are included on full-duplex and simplex TX framing cores.

Table 2-7: **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 2-8 lists the streaming RX data ports. These ports are included on full-duplex and simplex RX framing cores.

Table 2-8: **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 53 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. You supply 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 2-9 describes the ports for the UFC interface.

Table 2-9: **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 3-10, page 58.
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 58 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 2-10 lists the ports for the NFC interface available only in full-duplex Aurora 8B/10B cores.

Table 2-10: **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.
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 56 for more information.

Status and Control Ports for Full-Duplex Cores

Table 2-11 describes the function of each of the status and control ports for full-duplex cores.

Table 2-11: **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 64 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/GTH Transceivers User Guide* (UG476) [Ref 2].
power_down	Input	Drives the power-down input of the GTP, GTX, and GTH 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 six `user_clk` cycles.

Table 2-11: **Status and Control Ports for Full-Duplex Cores *(Cont'd)***

Name	**Direction**	**Description**
soft_err	Output	Soft error detected in the incoming serial stream. See Error Signals in Full-Duplex Cores, page 64 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] (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.
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 Physical Coding Sublayer (PCS) and Physical Medium Attachment (PMA) subcomponents of the transceiver. The signal is debounced using `init_clk_in` and must be asserted for six `init_clk_in` cycles. See the Reset section in the respective transceiver user guide for further details.
init_clk_in	Input	The `init_clk_in` port is used to register and debounce the `gt_reset` signal. The `init_clk_in` port is required because `user_clk` stops when gt_reset is asserted. The `init_clk_in` port should be set to a slow rate, preferably slower than the reference clock. It is constrained for 50 MHz frequency by default in `<component name>_exdes.xdc`. Update this clock constraint with respect to your clock board frequency.

Notes:

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

See Full-Duplex Cores, page 63 for more information.

Status and Control Ports for Simplex Cores

Table 2-12 describes the function of each of the status and control ports in the simplex TX interface.

Table 2-12: **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`.
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 66 for more details.
power_down	Input	Drives the powerdown input of the GTP or GTX transceiver (active-High).
tx_system_reset	Input	Resets the Aurora 8B/10B core (active-High). Reset should be asserted for a minimum of six cycles of `user_clk`.
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, or GTH transceivers.

Table 2-13 describes the function of each of the status and control ports in the simplex RX interface.

Table 2-13: **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`.
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 66 for more details.
power_down	Input	Drives the power-down input of the GTP or GTX transceiver (active-High).
rx_system_reset	Input	Resets the Aurora 8B/10B core (active-High). Reset should be asserted for a minimum of six cycles of `user_clk`.
soft_err	Output	Soft error detected in the incoming serial stream. See Error Signals in Simplex Cores, page 66 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 or GTX transceivers.
2. The `rx_aligned`, `rx_bonded`, `rx_verify`, and `rx_reset` signals are available as output signals even when the simplex partner is timer based, but functionally these signals are not required.

See Simplex Cores, page 66 for more information.

Transceiver Interface

This interface includes the serial I/O ports of the GTP, GTX, or GTH transceivers, and the control and status ports of the Aurora 8B/10B core. This interface is the user 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 2-14 describes the transceiver ports.

Table 2-14: **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.
power_down	Input	Drives the power-down input of the GTP, GTX, or GTH transceiver (active-High). See the relevant transceiver user guide for more information
loopback[2:0]	Input	Loopback port of the transceiver. See the applicable transceiver guide for loopback test mode configurations.
gt_reset	Input	Asynchronous reset signal for the transceiver. See the applicable transceiver guide for more information.
tx_resetdone_out	Output	The `txresetdone` signal of the 7 series FPGA GTP, GTX, or GTH transceiver. See the applicable transceiver guide for more information.
rx_resetdone_out	Output	The `rxresetdone` signal of the 7 series FPGA GTP, GTX, or GTH transceiver. See the applicable transceiver guide for more information.
tx_lock	Output	Indicates incoming serial transceiver refclk is locked by the transceiver PLL. See the applicable transceiver guide for more information.
	7 Series FPGA Transceiver DRP Ports	
drpaddr_in	Input	DRP address bus. See the applicable transceiver guide for more information.
drpclk_in	Input	DRP interface clock. See the applicable transceiver guide for more information.
drpdi_in	Input	Data bus for writing configuration data from the FPGA logic resources to the transceiver. See the applicable transceiver guide for more information.
drpdo_out	Output	Data bus for reading configuration data from the transceiver to the FPGA logic resources. See the applicable transceiver guide for more information.
drpen_in	Input	DRP enable signal. See the applicable transceiver guide for more information.

Table 2-14: **Transceiver Ports *(Cont'd)***

Name	Direction	Description
drprdy_out	Output	Indicates operation is complete for write operations and data is valid for read operations. See the applicable transceiver guide for more information.
drpwe_in	Input	DRP write enable. See the applicable transceiver guide for more information.
7 Series Transceiver Debug Ports		
gt<lane>_txpostcursor_in	Input	Transmitter post-cursor TX pre-emphasis control. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_txprecursor_in	Input	Transmitter pre-cursor TX pre-emphasis control. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_txchardispmode_in	Input	Set High to work with `txchardispval` to force running disparity negative or positive when encoding TXDATA. Set Low to use normal running disparity. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_txchardispval_in	Input	Work with `txchardispmode` to provide running disparity control. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_txdiffctrl_in	Input	Driver Swing Control. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_txmaincursor_in	Input	Allows the main cursor coefficients to be directly set if the TX_MAINCURSOR_SEL attribute is set to 1'b1. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_txpolarity_in	Input	The `txpolarity` port is used to invert the polarity of outgoing data. • 0: Not inverted. TXP is positive, and TXN is negative. • 1: Inverted. TXP is negative, and TXN is positive. Available for Duplex and TX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_tx_buf_err_out	Output	TX buffer status. `txbufstatus[1]` is connected to this port. Available for Duplex and TX-Only Simplex configuration

Table 2-14: **Transceiver Ports *(Cont'd)***

Name	Direction	Description
gt<lane>_rxlpmhfhold_in	Input	When set to 1'b1, the current value of the high-frequency boost is held. When set to 1'b0, the high-frequency boost is adapted. Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTP transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_rxlpmlfhold_in	Input	When set to 1'b1, the current value of the low-frequency boost is held. When set to 1'b0, the low-frequency boost is adapted. Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTP transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_rxlpmen_in	Input	RX datapath • 0: DFE • 1: LPM Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTX and GTH transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_rxcdrovrden_in	Input	Reserved. Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTX and GTH transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_rxcdrhold_in	Input	Hold the CDR control loop frozen. Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTX and GTH transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_rxdfelpmreset_in	Input	This port is driven High and then deasserted to start the DFE reset process. Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTX and GTH transceivers only. See the relevant transceiver user guide for more information.

Table 2-14: **Transceiver Ports *(Cont'd)***

Name	Direction	Description
gt<lane>_rxmonitorout_out	Output	GTX transceiver: • RXDFEVP[6:0] = RXMONITOROUT[6:0] • RXDFEUT[6:0] = RXMONITOROUT[6:0] • RXDFEAGC[4:0] = RXMONITOROUT[4:0] GTH transceiver: • RXDFEVP[6:0] = RXMONITOROUT[6:0] • RXDFEUT[6:0] = RXMONITOROUT[6:0] • RXDFEAGC[3:0] = RXMONITOROUT[4:1] Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTX and GTH transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_rxmonitorsel_in	Input	Select signal for `rxmonitorout[6:0]` • `2'b00`: Reserved • `2'b01`: Select AGC loop • `2'b10`: Select UT loop • `2'b11`: Select VP loop Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTX and GTH transceivers only. See the relevant transceiver user guide for more information.
gt<lane>_eyescanreset_in	Input	This port is driven High and then deasserted to start the EYESCAN reset process. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_eyescandataerror_out	Output	Asserts High for one `rec_clk` cycle when an (unmasked) error occurs while in the COUNT or ARMED state. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_eyescantrigger_in	Input	Causes a trigger event. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_rxbyteisaligned_out	Output	This signal from the comma detection and realignment circuit is High to indicate that the parallel data stream is properly aligned on byte boundaries according to comma detection. • 0: Parallel data stream not aligned to byte boundaries • 1: Parallel data stream aligned to byte boundaries Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.

Table 2-14: **Transceiver Ports *(Cont'd)***

Name	Direction	Description
gt<lane>_rxcommadet_out	Output	This signal is asserted when the comma alignment block detects a comma. The assertion occurs several cycles before the comma is available at the FPGA RX interface. • 0: Comma not detected • 1: Comma detected Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_rx_disp_err_out	Output	Active-High indicates the corresponding byte shown on `rxdata` has a disparity error. The `rxdisperr` pin of the transceiver is connected to this port. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_rx_not_in_table_out	Output	Active-High indicates the corresponding byte shown on `rxdata` was not a valid character in the 8B/10B table. `rxnotintable` pin of transceiver is connected to this port. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_rx_realign_out	Output	This signal from the comma detection and realignment circuit indicates that the byte alignment within the serial data stream has changed due to comma detection. • 0: Byte alignment has not changed • 1: Byte alignment has changed Data can be lost or repeated when alignment occurs, which can cause data errors (and disparity errors when the 8B/10B decoder is used). The `rxbyterealign` pin of the transceiver is connected to this port. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.
gt<lane>_rx_buf_err_out	Output	RX buffer status. `rxbufstatus[2]` is connected to this port. Available for Duplex and RX-Only Simplex configuration. See the relevant transceiver user guide for more information.

Notes:

1. *m* is the number of GTP, GTX, or GTH transceivers.
2. The 7 series FPGA transceiver debug ports will get enabled if you select the A**dditional transceiver control and status ports** check-box option in the Vivado IDE.
3. <lane> takes values from 0 to AURORA_LANES.

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, or GTH transceivers, and ports for the parallel clocks that the Aurora 8B/10B core shares with application logic.

Table 2-15 describes the Aurora 8B/10B core clock ports.

Table 2-15: **Clock Ports for a GTP or 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 locked signal of the PLL. 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 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, or 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	The `gt_refclk (clkp/clkn)` port is a dedicated external clock generated from an oscillator. This clock is fed through IBUFDS. To minimize the number of oscillators, the GTP, GTX, or 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 clock compensation (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 v2.2* (SP002) [Ref 3]. When the same physical clock is used on both sides of the channel, `warn_cc` and `do_cc` should be tied Low.

Table 2-16 describes the function of the clock compensation interface ports.

Table 2-16: **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 40 for more information.

Designing with the Core

This chapter includes guidelines and additional information to make designing with the core easier.

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 that is created is delivered with an example design that can be simulated and implemented in an 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 core 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 core 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 and supports the Vivado® synthesis tool. No scripts are provided by 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 Vivado Integrated Design Environment (IDE).

Shared Logic

Up to version 9.1 of the core, the RTL hierarchy for the core was fixed. This resulted in some difficulty because shareable clocking and reset logic needed to be extracted from the core example design for use with a single instance or multiple instances of the core.

Shared logic is a new feature that provides a more flexible architecture that works both as a standalone core and as part of a larger design with one or more core instances. This minimizes the amount of HDL modifications required, but at the same time retains the flexibility to address more use cases.

The new level of hierarchy is called <component_name>_support. Figure 3-1 and Figure 3-2 show two hierarchies where the shared logic block is contained either in the core or in the example design. In these figures, <component_name> is the name of the generated core. The difference between the two hierarchies is the boundary of the core. It is controlled using the **Shared Logic** option in the Vivado IDE (see Figure 5-2).

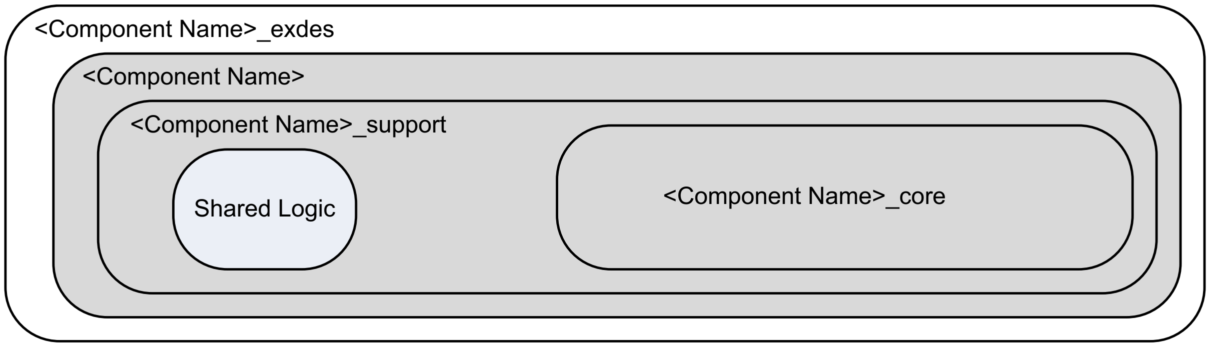


Figure 3-1: **Shared Logic Included in Core**

Note: Grayed blocks in Figure 3-1 and Figure 3-2 refer to the IP core.

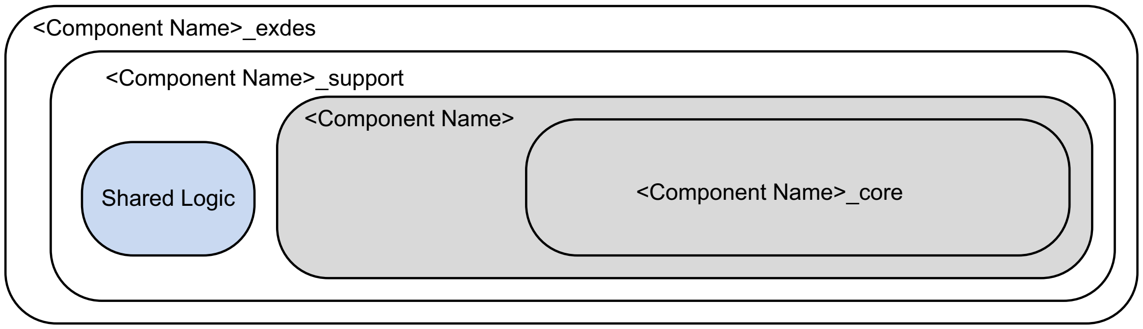


Figure 3-2: **Shared Logic Included in Example Design**

The contents of the shared logic depend upon the physical interface and the target device. Shared logic will contain instance(s) of GT differential buffer (IBUFDS_GTE2), support reset logic and instantiation of <=: USER_COMPONENT_NAME:>_CLOCK_MODULE. In addition to these blocks, shared logic will also contain instance of transceiver common. This transceiver common is instantiated based selected transceiver type and GTPE2_COMMON, GTXE2_COMMON and GTHE2_COMMON is instantiated for 7 series FPGA GTP, GTX and GTH transceivers respectively. Support reset logic contains the de-bouncer logic for the `reset` and `gt_reset` ports.

Note: The Aurora 8B/10B core uses CPLL and does not use QPLL (that is, GTXE2_COMMON/GTHE2_COMMON). QPLL is brought out and instantiated in shared logic for uniformity with other Xilinx serial connectivity cores.

The following table provides the details about the port changes due to **Shared Logic** option.

Table 3-1: **Port Changes Due to Shared Logic Option**

Name	Direction	Description	Remarks
gt_refclk1_p gt_refclk1_n	Input	Transceiver reference clock 1	Enabled when Shared Logic is in the example design
gt_refclk2_p gt_refclk2_n	Input	Transceiver reference clock 2	Enabled when Shared Logic is in example design
gt_refclk1_out	Output	Output of IBUFDS_GTE2 for transceiver reference clock 1	Enabled when **Shared Logic in Core** is selected
gt_refclk2_out	Output	Output of IBUFDS_GTE2 for transceiver reference clock 2	Enabled when **Shared Logic in Core** is selected
user_clk_out	Output	Parallel clock shared by Aurora 8B1/0B core	Enabled when **Shared Logic in Core** is selected
sync_clk_out	Output	TXUSRCLK for Artix-7 device GTP transceiver designs	Enabled when **Shared Logic in Core** is selected
sys_reset_out	Output	Output of de-bouncer for RESET	Enabled when **Shared Logic in Core** is selected
gt_reset_out	Output	Output of de-bouncer for GT_RESET	Enabled when **Shared Logic in Core** is selected
init_clk_p init_clk_n	Input	Free running system/board clock	Enabled when **Shared Logic in Core** is selected
init_clk_out	Output	Output of system clock differential buffer	Enabled when **Shared Logic in Core** is selected
gt0_pll0refclklost_out gt1_pll0refclklost_out	Output	Indicates `refclklost` port of the GTPE2_COMMON	Enabled when **Shared Logic in Core** is selected. Applicable for Artix-7 FPGA GTP transceiver designs.
quad1_common_lock_out quad2_common_lock_out	Output	Indicates PLL of the GTPE2_COMMON is achieved lock	Enabled when **Shared Logic in Core** is selected. Applicable for Artix-7 FPGA GTP transceiver designs.
gt0_pll0outclk_out gt0_pll1outclk_out gt0_pll0outrefclk_out gt0_pll1outrefclk_out gt1_pll0outclk_out gt1_pll1outclk_out gt1_pll0outrefclk_out gt1_pll1outrefclk_out	Output	Clock outputs generated by GTPE2_COMMON	Enabled when **Shared Logic in Core** is selected. Applicable for Artix-7 FPGA GTP transceiver designs.

Table 3-1: **Port Changes Due to Shared Logic Option *(Cont'd)***

Name	Direction	Description	Remarks
gt<quad>_qplllock_out	Output	Indicates PLL of the GTXE2_COMMON/GTHE2_COMMON has achieved lock	Enabled when **Shared Logic in Core** is selected. Applicable for 7 series FPGA GTX/GTH transceiver designs. These ports are enabled for each quad that you select in the Vivado IDE during core configuration in the Vivado Design Suite.
gt<quad>_qpllrefclklost_out	Output	Indicates REFCLKLOST port of the GTXE2_COMMON/GTHE2_COMMON	Enabled when **Shared Logic in Core** is selected. Applicable for 7 series FPGA GTX/GTH transceiver designs. These ports are enabled for each quad that you select in the Vivado IDE during core configuration in the Vivado Design Suite.
gt_qpllclk_quad<quad>_out gt_qpllclk_quad<quad>_out	Output	Clock outputs generated by GTXE2_COMMON/GTHE2_COMMON	Enabled when **Shared Logic in Core** is selected. Applicable for 7 series FPGA GTX/GTH transceiver designs. These ports are enabled for each quad that you select in the Vivado IDE during core configuration in the Vivado Design Suite.

Note: <quad> refers to the transceiver quad and starts from 1 to 12.

The `gt_refclk1_out` and `gt_refclk2_out` signals can be shared by other transceivers in the design and should follow the transceiver clocking guidelines for connectivity and transceiver quad proximity.

GTPE2_COMMON of the Artix®-7 device design ports is used by the core and can be shared by other cores that belong to same quad. The `init_clk_out` clock can be used by other core in the system. `user_clk_out` should be used in modules on the example design (that is, frame_gen, frame_check and standard_cc_module).

Serial Transceiver Reference Clock Interface

Functional Description

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, or GTH transceivers. It also requires at least one frequency locked parallel clock for synchronous operation with the user application.

The Virtex®-7, Kintex®-7, and Artix-7 FPGAs have four GTP, GTX, and GTH transceivers in a Quad. Virtex-7 and Kintex-7 FPGA GTX or GTH transceivers have a channel PLL (CPLL) per transceiver and a Quad PLL (QPLL) per quad and core configures PLL0 for Artix-7 devices. The Artix-7 FPGA GTP transceiver has two PLLs (PLL0 and PLL1) per quad. Aurora 8B/10B core configures CPLL in Virtex-7, Kintex-7, and Zynq®-7000 family designs.

Each Aurora 8B/10B core is generated in the `<component name>_example` 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 core example design and use it as a template when connecting the clock interface.

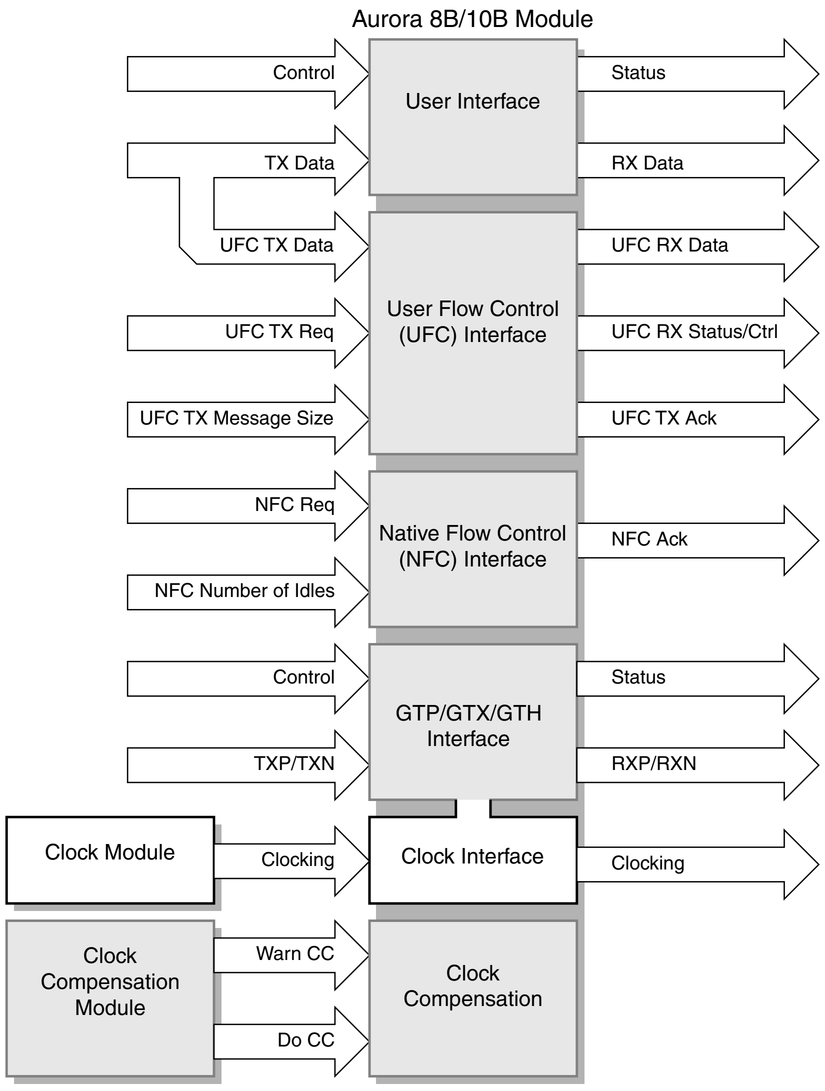


Figure 3-3: **Top-Level Clocking**

Clock Interface Ports for the Aurora Core

See Table 2-15, page 28 for descriptions of the transceiver ports on the clock interface.

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

The Xilinx implementation tools make necessary adjustments to the north-south routing (shown in Figure 3-4, page 38) and to the pin swapping necessary to GTXE2 or GTHE2 transceiver clock inputs to route clocks from one quad to another, when required.

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

- The number of GTX or GTH transceiver quads above the sourcing quad must not exceed one.
- The number of GTX or GTH transceiver quads below the sourcing quad must not exceed one.
- The total number of GTX or GTH transceiver 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 or 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.

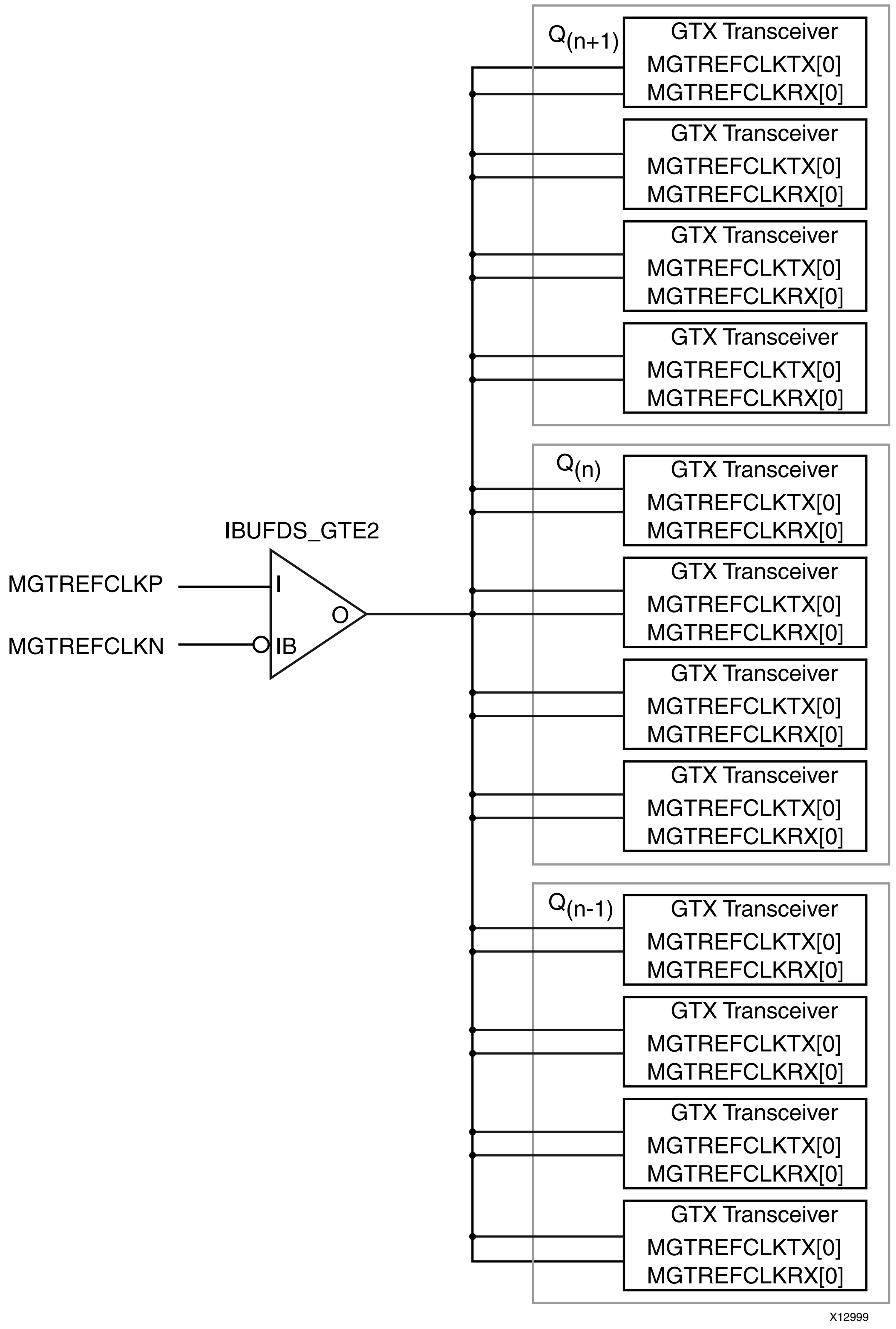


Figure 3-4: **North-South Routing Adjustments in Virtex-7 and Kintex-7 FPGAs**

Clock Rates for GTP, GTX, and GTH Transceiver Designs

GTP, GTX, and GTH transceivers support a wide range of serial rates. The attributes used to configure the GTP, GTX, and GTH transceivers in the Aurora 8B/10B core for a specific line rate are kept in the <component name>_gt 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 Vivado IDE window for the core.

IMPORTANT: *Manual edits of the attributes are not recommended, but are possible using the recommendations in the 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 2].*

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 clock compensation interface enables full control over the core clock compensation features. A standard clock compensation module is generated with the 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.

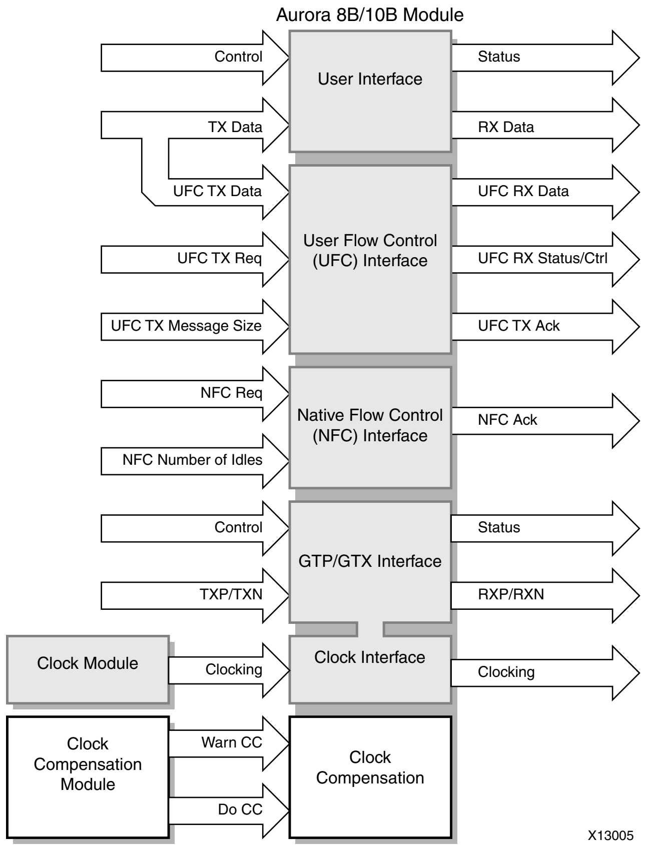


Figure 3-5: **Top-Level Clock Compensation**

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 clock compensation signal works.

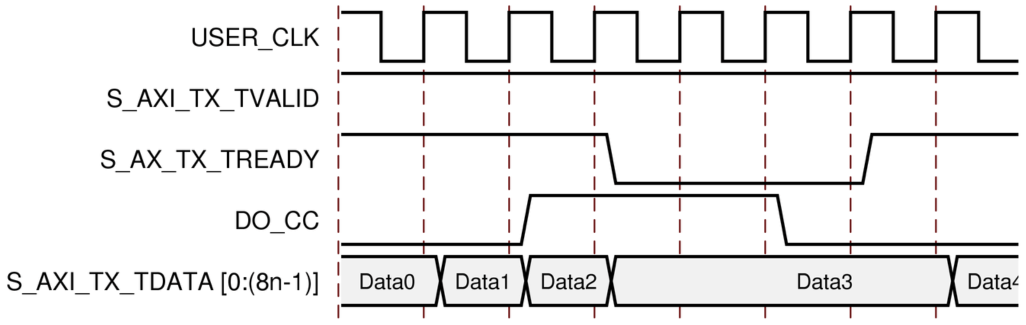


Figure 3-6: **Streaming Data with Clock Compensation Inserted**

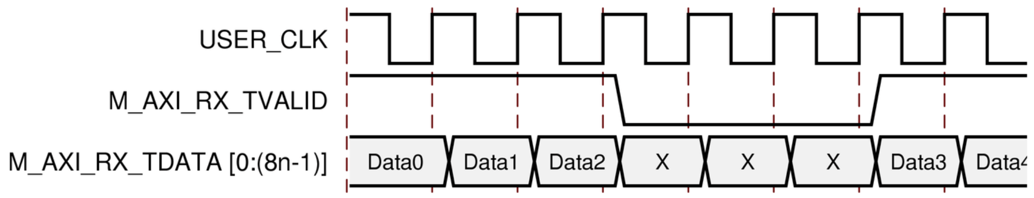


Figure 3-7: **Data Reception Interrupted by Clock Compensation**

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 or 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 3-2 shows the required durations and periods for 2-byte and 4-byte wide lanes.

Table 3-2: **Clock Compensation Cycles**

Lane Width	USER_CLK Cycles Between DO_CC	DO_CC Duration (USER_CLK cycles)
2	5000	6
4	2500	3

The `warn_cc` signal 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 3-3 shows the number of lookahead cycles required for each combination of lane width, channel width, and maximum UFC message size.

Table 3-3: **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

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

Data Interface Width	Max UFC Size	WARN_CC Lookahead
2	14	9
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 design tool 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 3-4 shows the port description for the standard CC module.

Table 3-4: **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 41) 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.

IMPORTANT: *In general, customizing the clock compensation logic is not recommended, and when it is attempted, it should be performed with careful analysis, testing, and consideration of 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 `lane_up`; `do_cc` should be asserted immediately after `lane_up` because no clock compensation can occur during initialization

User Data 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 55.

The framing user interface complies with the *AMBA AXI4-Stream Protocol Specification* [Ref 4]. It comprises the signals necessary for transmitting and receiving framed user data. The streaming interface allows you 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 or GTH transceiver. This top-level wrapper file is instantiated in the example design file together with clock, reset circuit and frame generator and checker modules.

Figure 3-8 shows the Aurora 8B/10B top level core for a duplex configuration. The top-level file is the starting point for a user design.

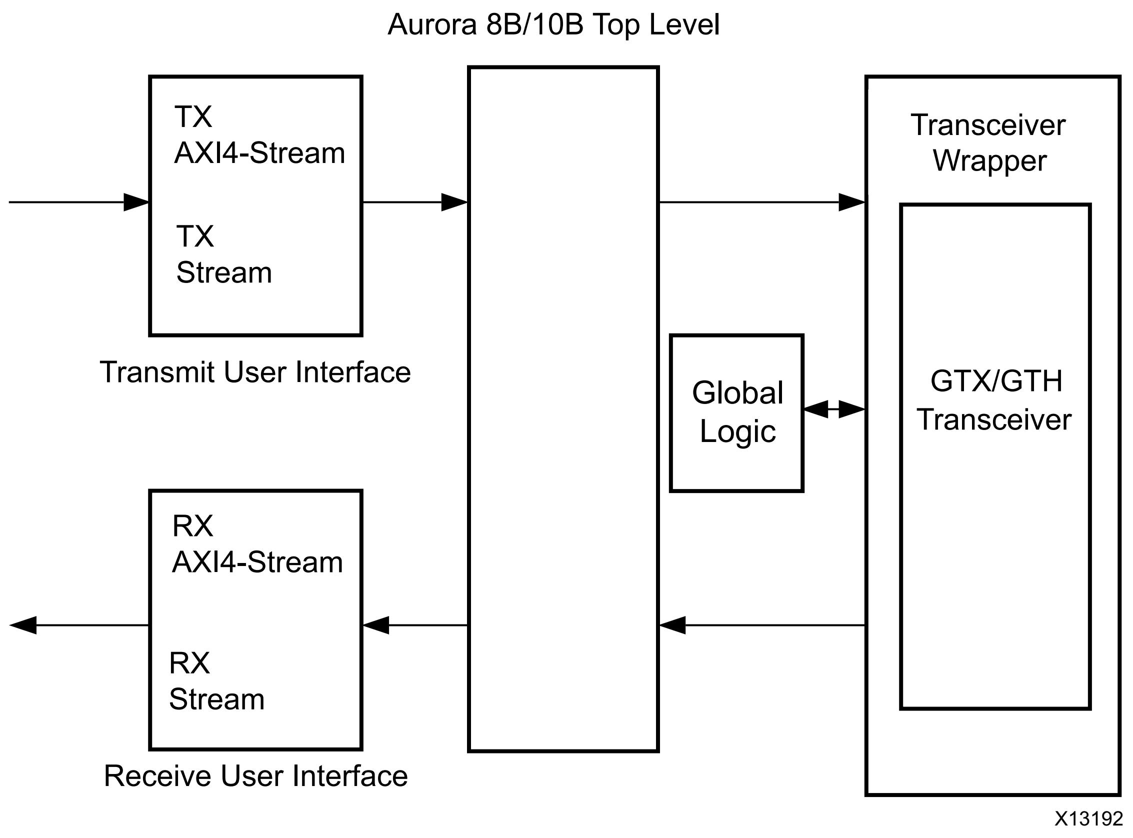


Figure 3-8: **Top-Level Architecture**

This section provides 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 here.

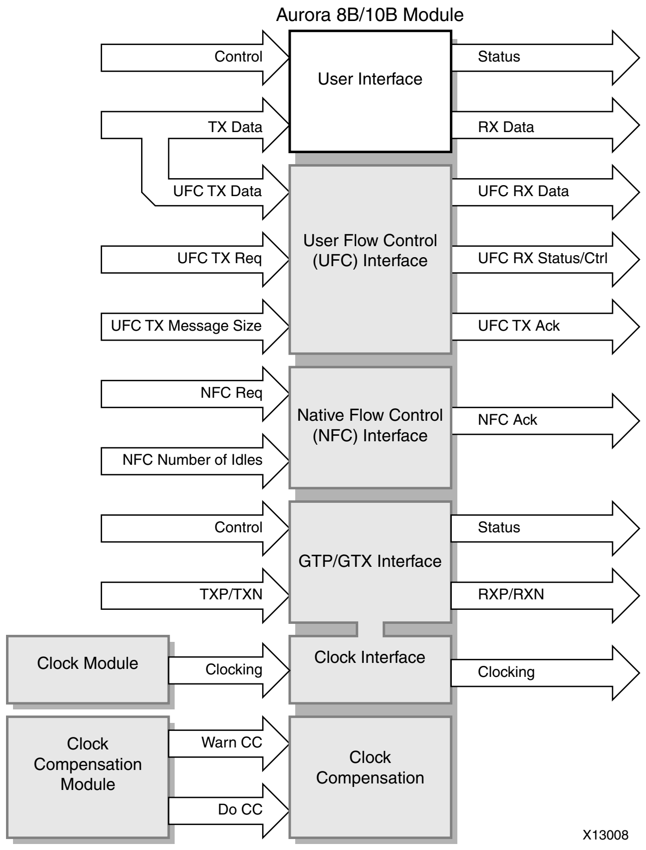


Figure 3-9: **Top-Level User Interface**

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

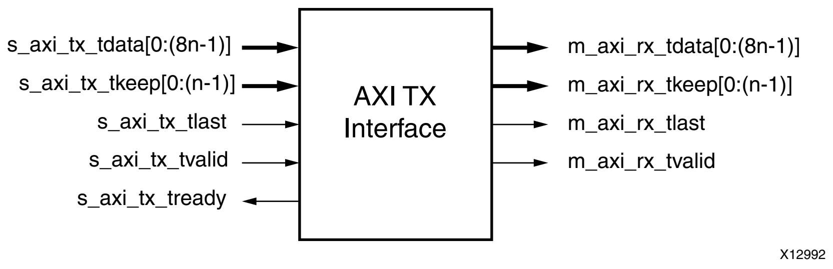


Figure 3-10: **Aurora 8B/10B Core Framing Interface (AXI4-Stream)**

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

- Take data from the user interface 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, Start of Channel PDU (SCP), End of Channel PDU (ECP))
- Asserts framing signal (`m_axi_rx_tlast`)
- Recovers data from the lanes
- Assembles data for presentation to the user interface 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.

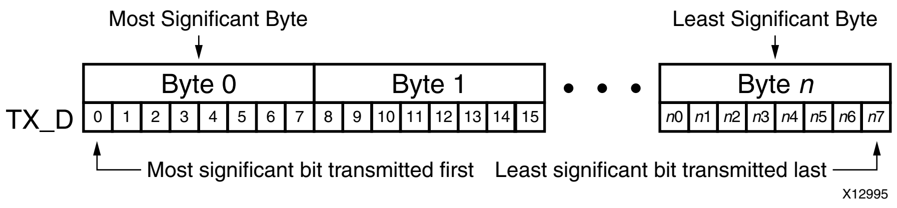


Figure 3-11: **AXI4-Stream Interface Bit Ordering**

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 two-byte SCP code group is added to the beginning of the frame data to indicate the start of frame, and a two-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 the Aurora 8B/10B core 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 3-5 shows a typical Aurora 8B/10B frame with an even number of data bytes.

Length

The user application 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 3-5.

Table 3-5: **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 application 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 application 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.

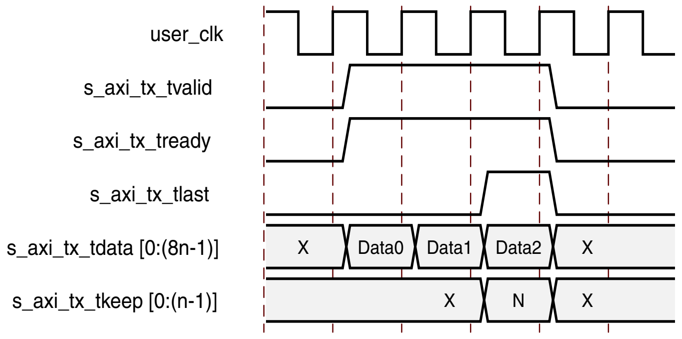


Figure 3-12: **Simple Data Transfer**

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.

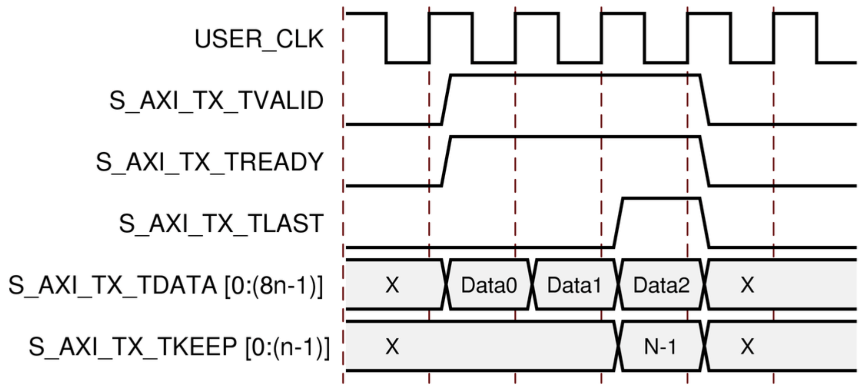


Figure 3-13: **Data Transfer with Pad**

Example C: Data Transfer with Pause

Figure 3-14 shows how a user interface can pause data transmission during a frame transfer. In this example, the user application is sending 3*n* bytes of data, and pauses the data flow after the first *n* bytes. After the first data word, the user application deasserts `s_axi_tx_tvalid`, causing the TX Aurora 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.

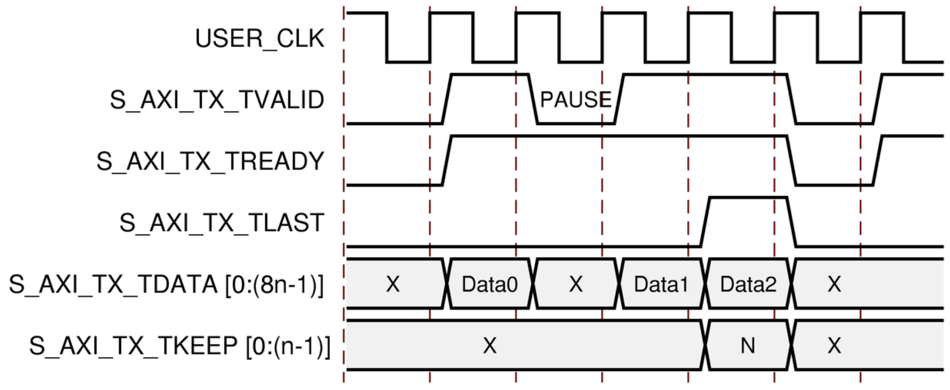


Figure 3-14: **Data Transfer with Pause**

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 sequence.

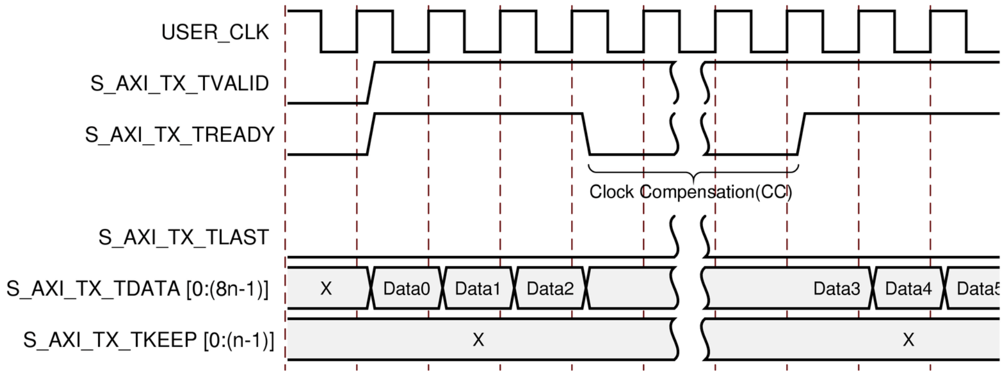


Figure 3-15: **Data Transfer Paused by Clock Compensation**

Note: 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 application 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 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).

The `m_axi_rx_tvalid` signal 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.The `m_axi_rx_tkeep` signal 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 application. On the clock cycle following the first data beat, the core deasserts `M_AXI_RX_TVALID`, indicating to the user application that there is a pause in the data flow.

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

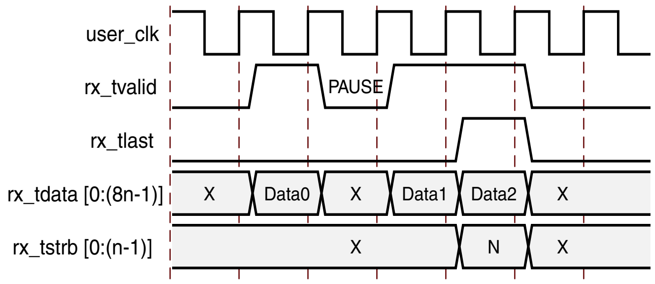


Figure 3-16: **Data Reception with Pause**

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 the Aurora 8B/10B core are sent in two-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 Equation 3-1 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.

$$E = \frac{100n}{n + 4 + 0.5 + IDLEs + \frac{12n}{9988}} \qquad \textit{Equation 3-1}$$

Where:

- E = The average efficiency of a specified PDU
- n = Number of user data bytes
- 12n/9988 = Clock correction overhead
- 4 = Overhead of SCP + ECP
- 0.5 = Average PAD overhead
- IDLEs = Overhead for IDLEs = (w/2) – 1
- w = Interface width

Example

Table 3-6 is an example calculated from Equation 3-1. 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 3-6: **Efficiency Example**

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

Table 3-7 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 3-7: **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/
2	33	Transmit idles	/I/	/I/
3	33	End of channel frame	$/ECP/_1$	$/ECP/_2$

Table 3-8 shows the overhead that occurs with each value of `S_AXI_TX_TKEEP`.

Table 3-8: **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 3-17 shows an example of an Aurora 8B/10B core configured with a streaming user interface.

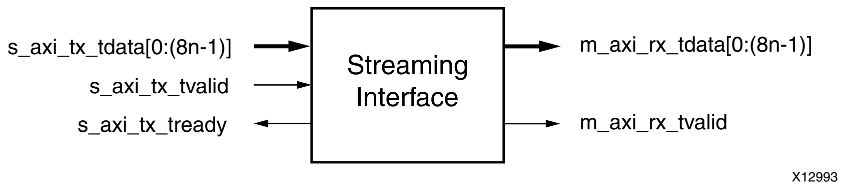


Figure 3-17: **Aurora 8B/10B Core Streaming User Interface**

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 39.

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 3-18 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.

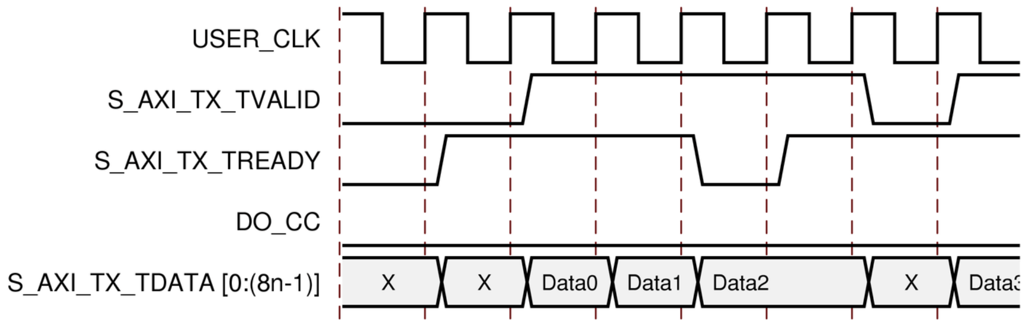


Figure 3-18: **Typical Streaming Data Transfer**

Figure 3-19 shows the receiving end of the data transfer that is shown in Figure 3-18.

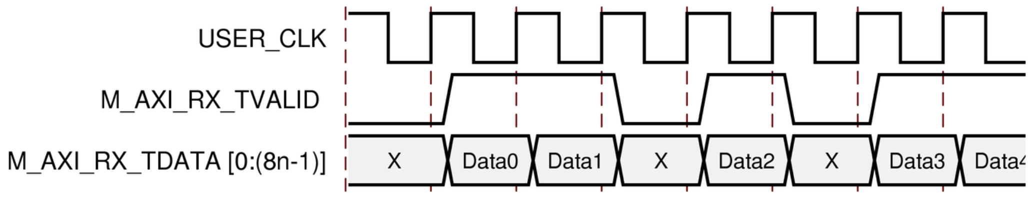


Figure 3-19: **Typical Data Reception**

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.

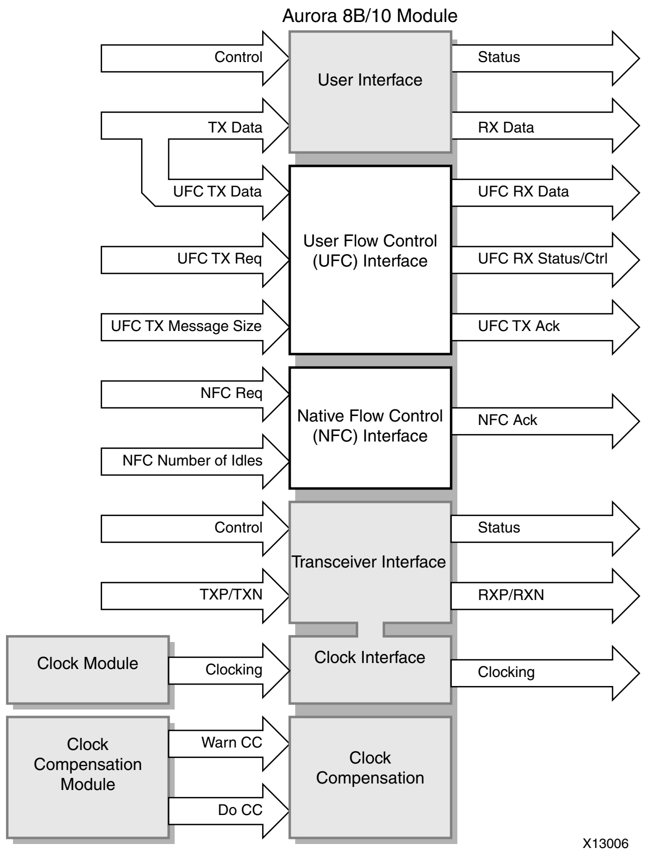


Figure 3-20: **Top-Level Flow Control**

Native Flow Control

Table 3-9 shows the codes for native flow control (NFC).

Table 3-9: **NFC Codes**

S_AXI_NFC_NB	Idle Cycles Requested
`0000`	0 (XON)
`0001`	2
`0010`	4
`0011`	8
`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 v2.2* [Ref 3].

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 3-21 shows an example of the transmit timing when the user application sends an NFC message to a channel partner.

Note: The `s_axi_tx_tready` signal 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.

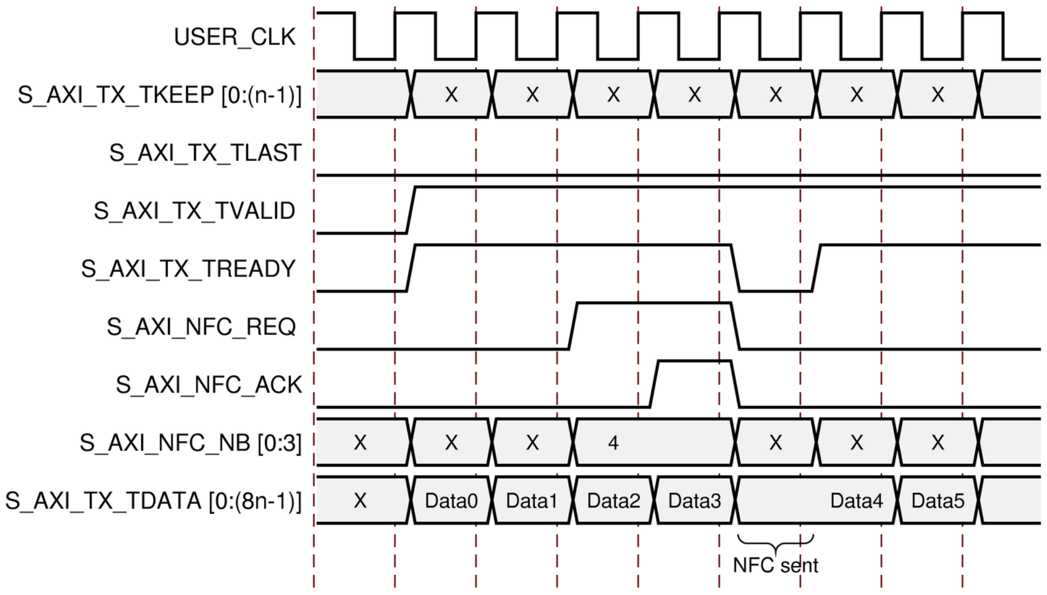


Figure 3-21: **Transmitting an NFC Message**

Example B: Receiving a Message with NFC Idles Inserted

Figure 3-22 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.

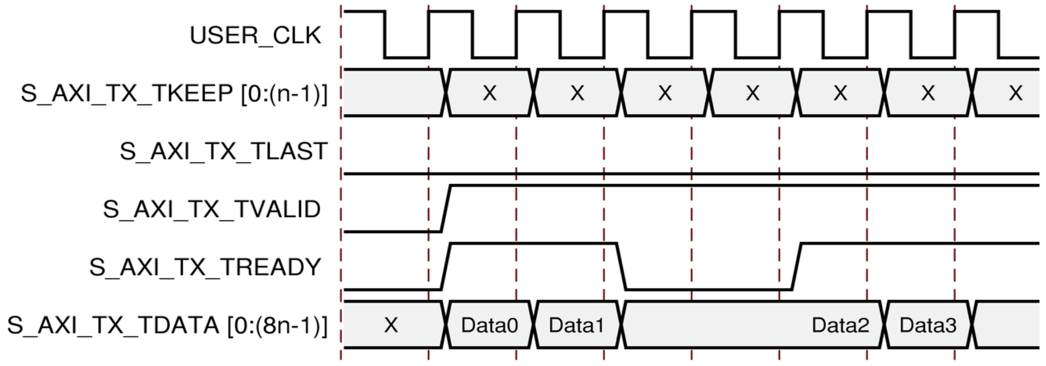


Figure 3-22: **Transmitting a Message with NFC Idles Inserted**

User Flow Control

The Aurora 8B/10B core protocol includes user flow control (UFC) to allow channel partners to send control information using a separate in-band channel. You can send short UFC messages to the core 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 3-10 shows the legal SIZE code values for UFC.

Table 3-10: **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. The `s_axi_ufc_tx_req` signal 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 3-23 shows a useful circuit for switching `TX_D` from sending regular data to UFC data.

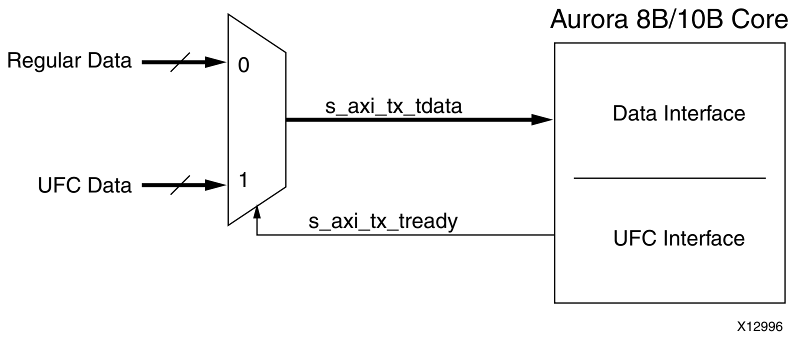


Figure 3-23: **Data Switching Circuit**

Table 3-11, page 59 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 3-11: **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				

Table 3-11: **Number of Data Beats Required to Transmit UFC Messages *(Cont'd)***

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	6 Bytes	1	14 Bytes	1
4 Bytes	1	6 Bytes	1	14 Bytes	1
6 Bytes	2	6 Bytes	1	14 Bytes	1
8 Bytes	3	6 Bytes	2	14 Bytes	1
10 Bytes	4	6 Bytes	2	14 Bytes	1
12 Bytes	5	6 Bytes	2	14 Bytes	1
14 Bytes	6	6 Bytes	3	14 Bytes	1
16 Bytes	7	6 Bytes	3	14 Bytes	2
2 Bytes	0	8 Bytes	1	16 Bytes or more	1
4 Bytes	1	8 Bytes	1	16 Bytes or more	1
6 Bytes	2	8 Bytes	1	16 Bytes or more	1
8 Bytes	3	8 Bytes	1	16 Bytes or more	1
10 Bytes	4	8 Bytes	2	16 Bytes or more	1
12 Bytes	5	8 Bytes	2	16 Bytes or more	1
14 Bytes	6	8 Bytes	2	16 Bytes or more	1
16 Bytes	7	8 Bytes	2	16 Bytes or more	1

Example A: Transmitting a Single-Cycle UFC Message

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

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

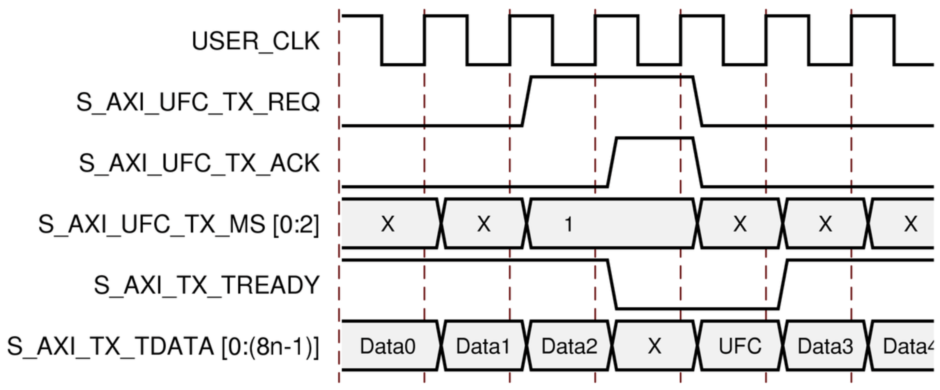


Figure 3-24: **Transmitting a Single-Cycle UFC Message**

Example B: Transmitting a Multicycle UFC Message

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

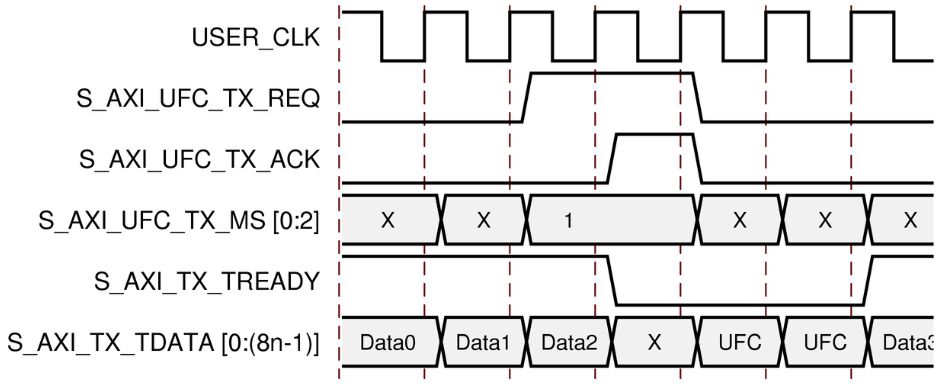


Figure 3-25: **Transmitting a Multicycle UFC Message**

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 3-26 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.

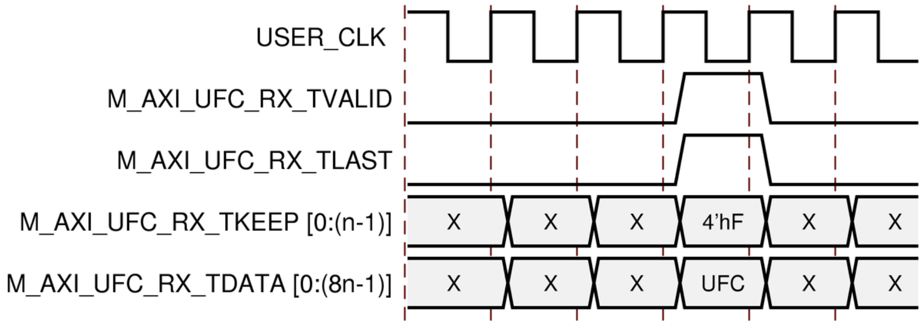


Figure 3-26: **Receiving a Single-Cycle UFC Message**

Example D: Receiving a Multicycle UFC Message

Figure 3-27 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.

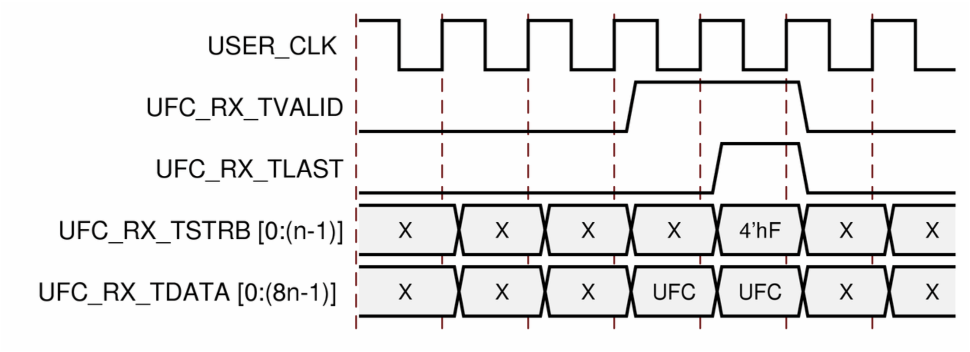


Figure 3-27: **Receiving a Multicycle UFC Message**

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, and GTH 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 status and control interface, along with the GTP, GTX, and GTH transceiver serial I/O interface and the sideband initialization ports that are used exclusively for simplex modules.

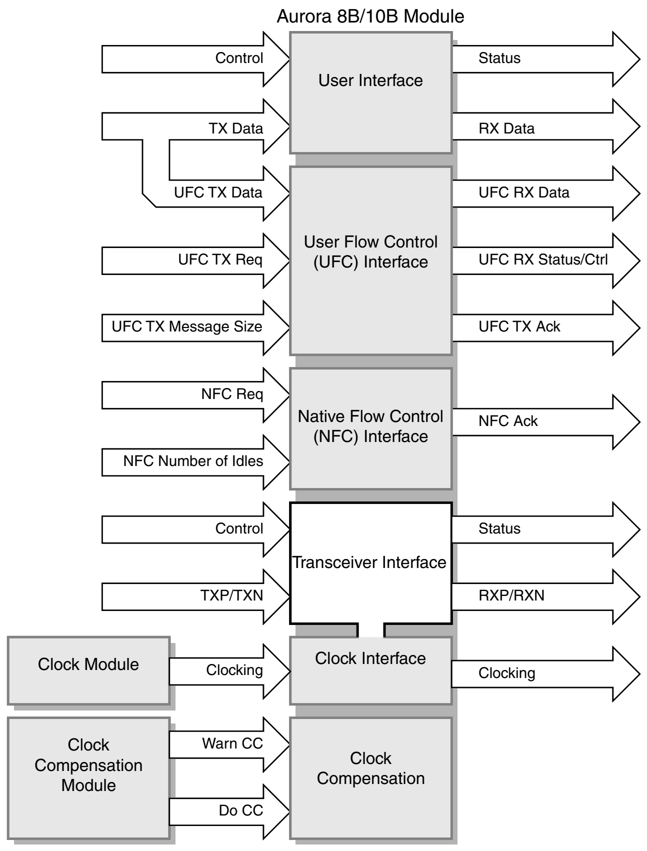


Figure 3-28: **Top-Level Transceiver Interface**

Full-Duplex Cores

Full-Duplex Status and Control Ports

Full-duplex cores provide a TX and an RX Aurora 8B/10B channel connection. Figure 3-29 shows the status and control interface for a full-duplex Aurora 8B/10B core.

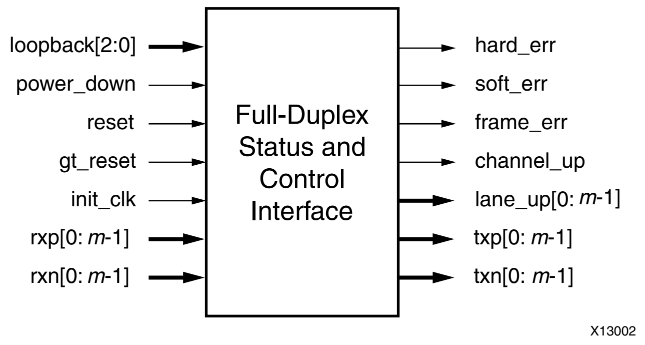


Figure 3-29: **Status and Control Interface for Full-Duplex Cores**

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 and 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 re-initialize. 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 3-12 summarizes the error conditions the Aurora 8B/10B core can detect and the error signals used to alert the user application.

Table 3-12: **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.

Note: The WATCHDOG_TIMEOUT parameter is available in the channel_init_sm module to control the watchdog timers present in the channel initialization process.

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 3-30 shows the status and control interface for a simplex TX core.

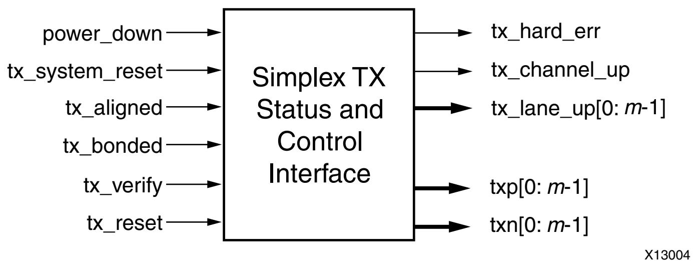


Figure 3-30: **Status and Control Interface for Simplex TX Core**

Simplex RX Status and Control Ports

Simplex RX cores allow user applications to receive data from a simplex TX core. Figure 3-31 shows the status and control interface for a simplex RX core.

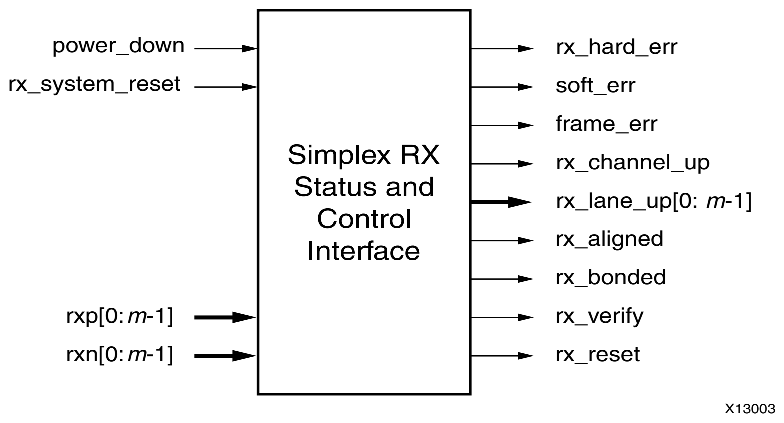


Figure 3-31: **Status and Control Interface for Simplex RX Core**

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 and 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 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 3-13 summarizes the error conditions simplex Aurora 8B/10B cores can detect and the error signals uses to alert the user application.

Table 3-13: **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 3-13: **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>_core 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.

Note: The WATCHDOG_TIMEOUT parameter is available in the tx_channel_init_sm/rx_channel_init_sm module to control the watchdog timers present in the channel initialization process.

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. The `gt_reset` signal resets the transceivers which eventually resets the Aurora core.

Reset

Use Case 1: Assertion of reset in duplex core

The `reset` assertion in the duplex core should be a minimum of six `user_clk` cycles. In effect to this, `channel_up` will be deasserted after three `user_clk` cycles as shown in the Figure 3-32.

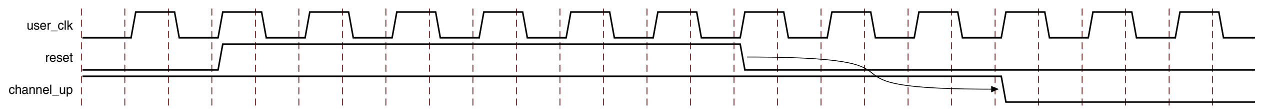


Figure 3-32: **RESET assertion in duplex core**

Use Case 2: gt_reset assertion in duplex core

Figure 3-33 shows the `gt_reset` assertion in the duplex core and should be a minimum of six `init_clk` cycles. As a result, `user_clk` is stopped after a few clock cycles because there is no `txoutclk` from the transceiver and `channel_up` will be deasserted.

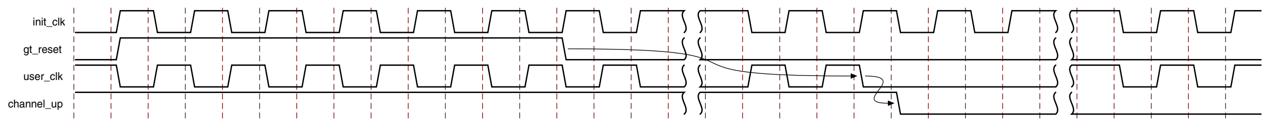


Figure 3-33: **GT_RESET assertion in duplex core**

Use Case 3: tx_system_reset and rx_system_reset assertion in simplex core

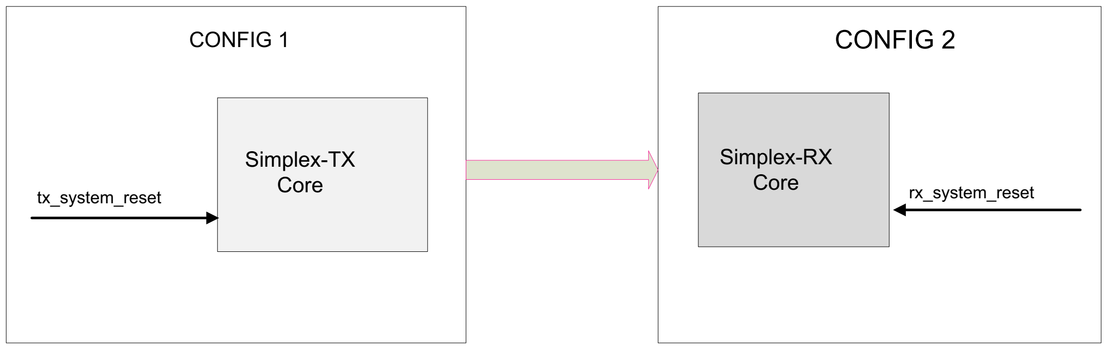


Figure 3-34: **System with Simplex Cores**

Figure 3-34 shows the Simplex-TX core and Simplex-RX core connected in a system. CONFIG1 and CONFIG2 could be in same or multiple device(s).

Figure 3-35 shows the recommended procedure of `tx_system_reset` and `rx_system_reset` assertion in the simplex core.

1. `tx_system_reset` and `rx_system_reset` are asserted for at least six clock cycles of `USER_CLK`.
2. `tx_channel_up` and `rx_channel_up` are deasserted after three clock cycles
3. `rx_system_reset` is deasserted (or) released after `tx_system_reset` is deasserted. This ensures that the transceiver in Simplex-TX core starts transmitting initialization data much earlier and it enhances the likelihood of the Simplex-RX core aligning to correct data sequence.
4. `rx_channel_up` is asserted before `tx_channel_up` assertion. This condition must be satisfied by the Simplex-RX core and simplex timer parameters (C_ALIGNED_TIMER, C_BONDED_TIMER and C_VERIFY_TIMER) in Simplex-TX core needs to be adjusted to meet this criteria.
5. `tx_channel_up` is asserted when the Simplex-TX core completes the Aurora protocol channel initialization sequence transmission for the configured time. Assertion of `tx_channel_up` last ensures that the Simplex-TX core will transmit Aurora initialization sequence when the Simplex-RX core is ready.

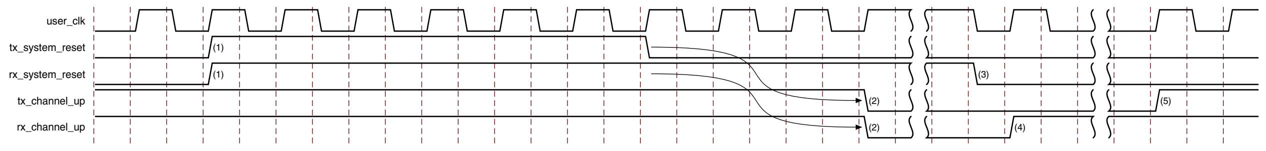


Figure 3-35: **tx_system_reset and rx_system_reset assertion in simplex core**

Recommended Reset Sequence for Aurora Core:

Figure 3-36 shows the recommended reset sequence to be followed.

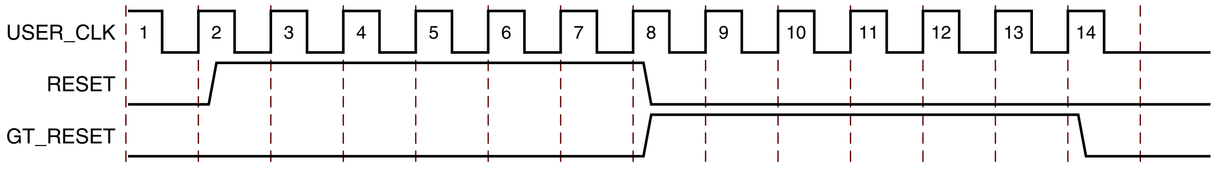


Figure 3-36: **Recommended Reset Sequence**

1. The `reset` signal is asserted for at least six clock cycles of `user_clk`.
2. The `gt_reset` signal is asserted after deassertion of `reset`.

This sequence will ensure that reset will be applied to fabric logic before `user_clk` is lost.

Power Down

This is an active-High signal. When `power_down` is asserted, the GTP and 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.

CAUTION! *Be careful when asserting this signal on cores that use `tx_out_clk` (see the Serial Transceiver Reference Clock Interface, page 35). `tx_out_clk` stops when the GTP and GTX transceivers are powered down. See the 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 2].*

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. The scrambler implements the following polynomial: G(x) = X16 + X5 + X4 + X3 + 1.

It ensures non-occurrence of repetitive data over long periods of time. The scrambler and descrambler are synchronized based on transmission and reception of the clock compensation characters respectively. 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 Vivado® Integrated Design Environment.

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	The `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 Vivado Lab Tools

The ICON and Virtual Input Output (VIO) cores in the Vivado® lab tools help to debug and validate the design in boards. These cores are provided with the Aurora 8B/10B core. Select the **Vivado Lab Tools** checkbox from the core Vivado IDE 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 core designs with Virtex®-7, Kintex®-7, and Artix®-7 FPGAs is based on the received clock compensation characters. Reception of clock compensation characters at RX interface of Aurora implies communication channel is alive and not broken. If clock compensation characters are not received in a predetermined time, the hot-plug logic resets the core and the transceiver. The clock compensation module must be used for Aurora 8B/10B designs with Virtex-7, Kintex-7, and Artix-7 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.

The hot-plug circuit is clocked by `init_clk` which is a free running clock; the frequency of the `init_clk` depends on the system/board design. RX_CC is generated in the USER_CLK domain and the USER_CLK frequency is dependent on the line rate and lane width of the Aurora core configuration. Hot-plug logic extends the RX_CC by four clock cycles in the USER_CLK domain to handle this frequency difference.

Customizing and Generating the Core

This chapter includes information about using Xilinx tools to customize and generate the Aurora 8B/10B core in the Vivado® Design Suite.

Note: This core supports the IP integrator but no parameters are grayed out. All dialog box options are visible and user-selectable in the IP integrator.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or popup menu.

For details, see the sections, "Working with IP" and "Customizing IP for the Design" in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5] and the "Working with the Vivado IDE" section in the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 6].

Note: Figures in this chapter are illustrations of the Vivado IDE. This layout might vary from the current version.

Vivado Integrated Design Environment

The Aurora 8B/10B core can be customized to suit a wide variety of requirements using the IP catalog tool. This chapter details the available customization parameters and how these parameters are specified within the IP Customizer interface.

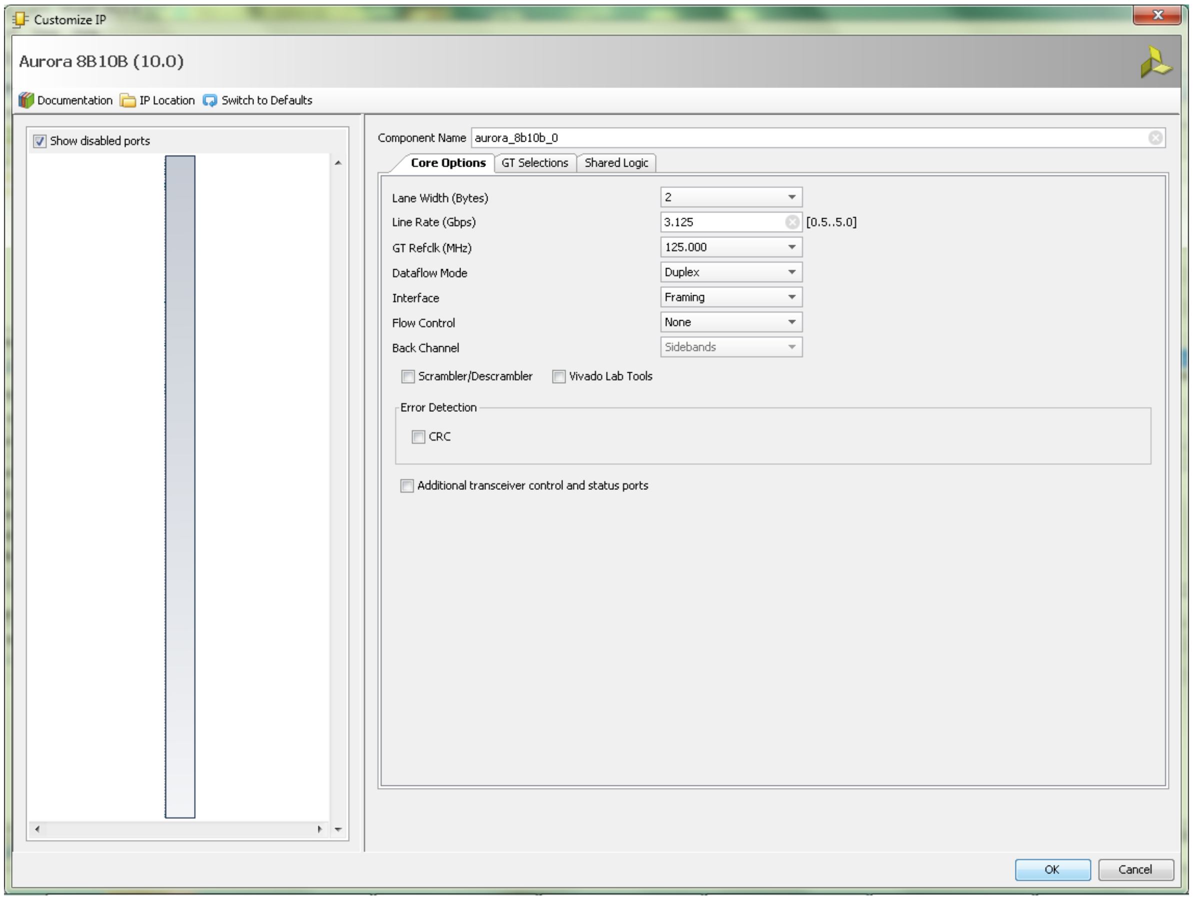
Using the IP Catalog

The Aurora 8B/10B IP Customizer is presented when you select the Aurora 8B/10B core in the IP catalog. For help starting and using the IP catalog, see the Vivado design tools documentation.

IP Customizer

Figure 5-1 shows the customizer with the default options. 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 tab of the Vivado Integrated Design Environment (IDE) – GT Selections is shown in Figure 5-3, page 81 for Virtex®-7 and Kintex®-7 FPGA GTX and GTH transceivers.



Component Name aurora_8b10b_0

Lane Width (Bytes)	2
Line Rate (Gbps)	3.125 [0.5..5.0]
GT Refclk (MHz)	125.000
Dataflow Mode	Duplex
Interface	Framing
Flow Control	None
Back Channel	Sidebands

☐ Scrambler/Descrambler ☐ Vivado Lab Tools

Error Detection

☐ CRC

☐ Additional transceiver control and status ports

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

Component Name

Enter the top-level name for the core in this text box. Illegal names are highlighted in red until they are corrected.

Default: aurora_8b10b_0

Lane Width

Select the byte width of the GTX, GTH, or GTP 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

Line Rate

Enter a floating-point value in gigabits per second within the valid range from 0.5 (Gb/s) to 6.6 (Gb/s).

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. Line rate is limited based on the speed grade and package of the selected device.

See the respective family data sheet for the limits. Aurora 8B/10B core supports up to 6.6 Gb/s line rate.

Default: 3.125 Gb/s

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: 125.000 MHz

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 62 for more information.

Default: Duplex

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 Data Interface, page 43 for more information.

Default: Framing

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 follow (see Flow Control, page 78 for more information):

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

Default: None

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.

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

Vivado Lab Tools

Select to add Vivado lab tool cores to the Aurora 8B/10B core. See Using Vivado Lab Tools in Chapter 4. This option provides a debugging interface that shows the core status signals in the Vivado Logic Analyzer.

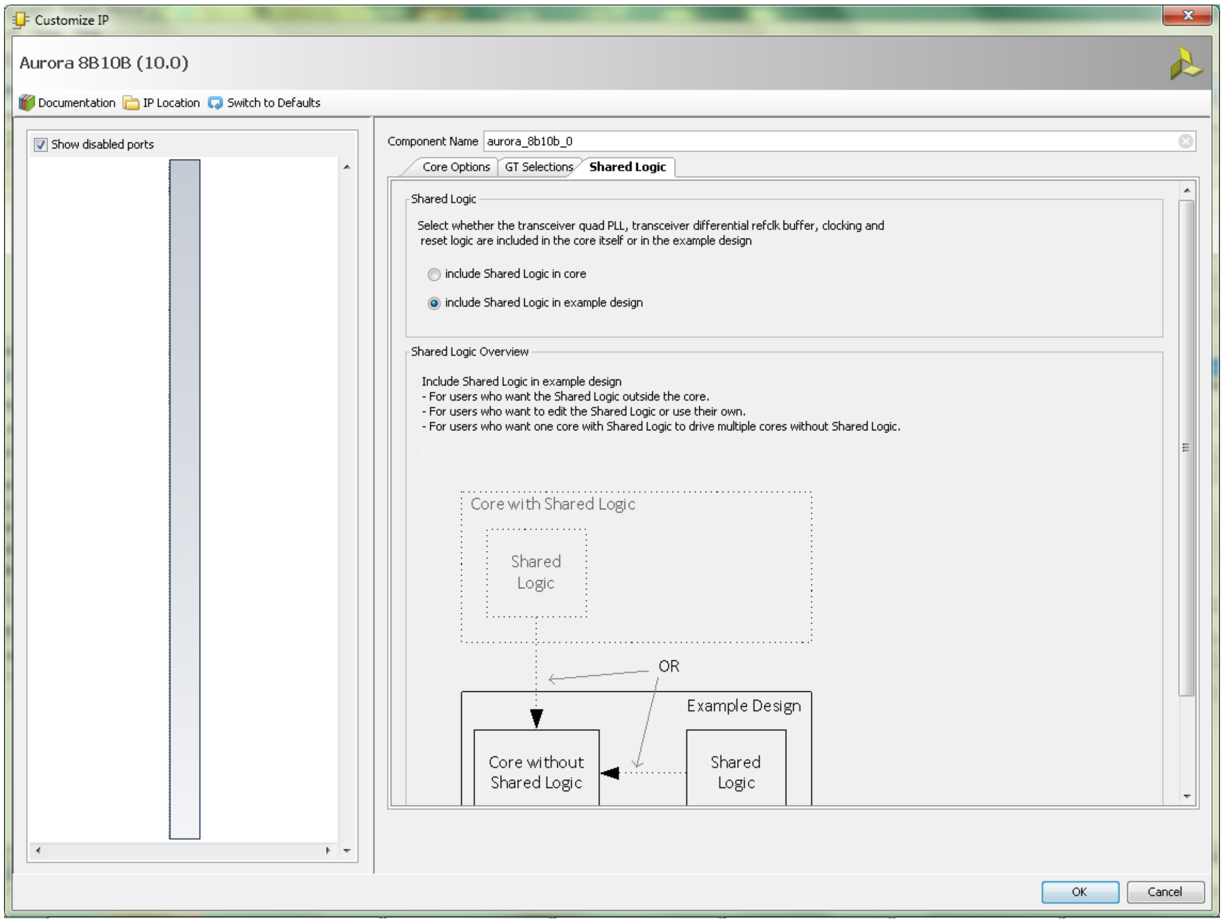
Default: Unchecked

Use CRC

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

Default: Unchecked

Shared Logic



Show disabled ports [✓]

Component Name aurora_8b10b_0

Shared Logic

Select whether the transceiver quad PLL, transceiver differential refclk buffer, clocking and reset logic are included in the core itself or in the example design

include Shared Logic in core [✓ include Shared Logic in example design]

Shared Logic Overview

Include Shared Logic in example design
- For users who want the Shared Logic outside the core.
- For users who want to edit the Shared Logic or use their own.
- For users who want one core with Shared Logic to drive multiple cores without Shared Logic.

Figure 5-2: **Shared Logic Tab**

Select to include transceiver common PLL and its logic in IP core or in the example design

Available options:

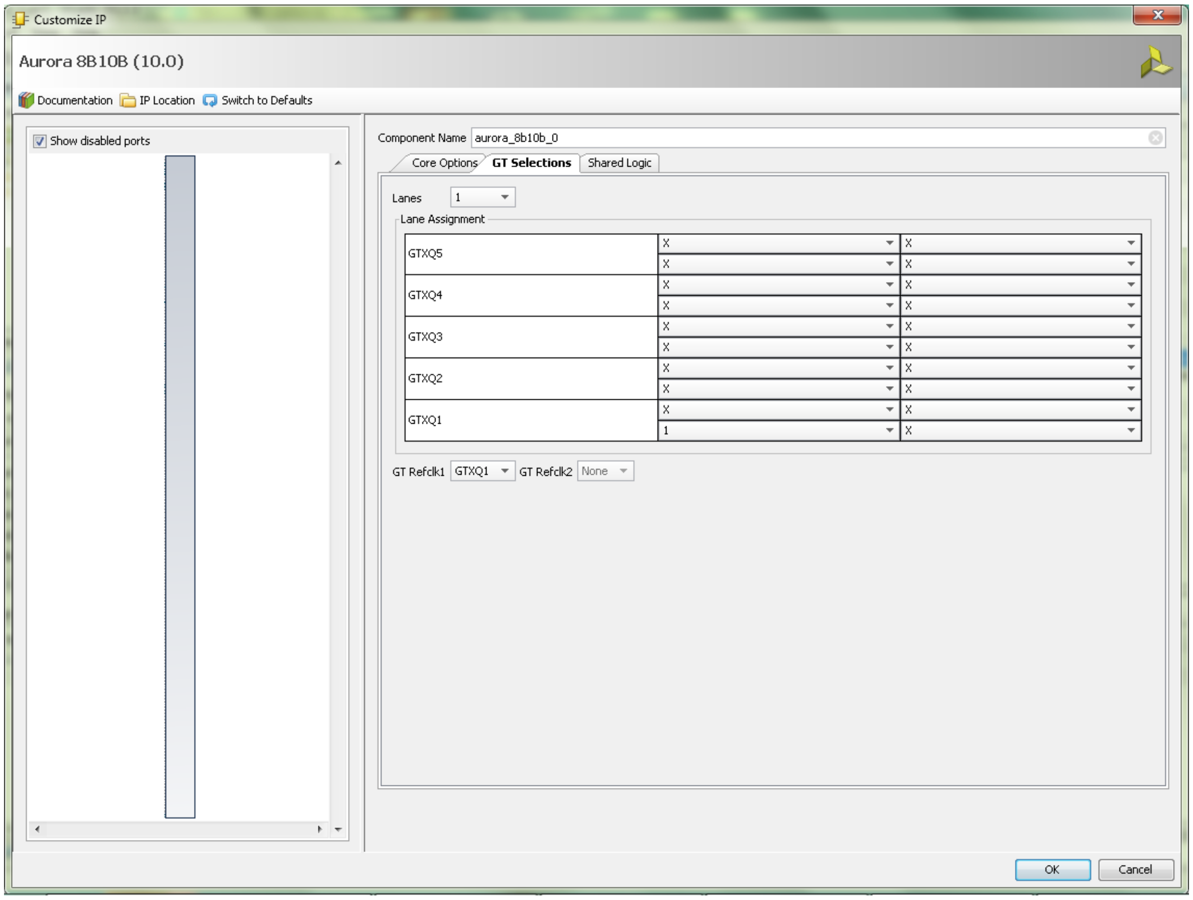
- include shared logic in core
- include shared logic in example design

Default: include shared logic in example design

Additional Transceiver Control and Status Ports

Select to include transceiver control and status ports to core top level

Default: Unchecked



[✓] Show disabled ports

Component Name aurora_8b10b_0

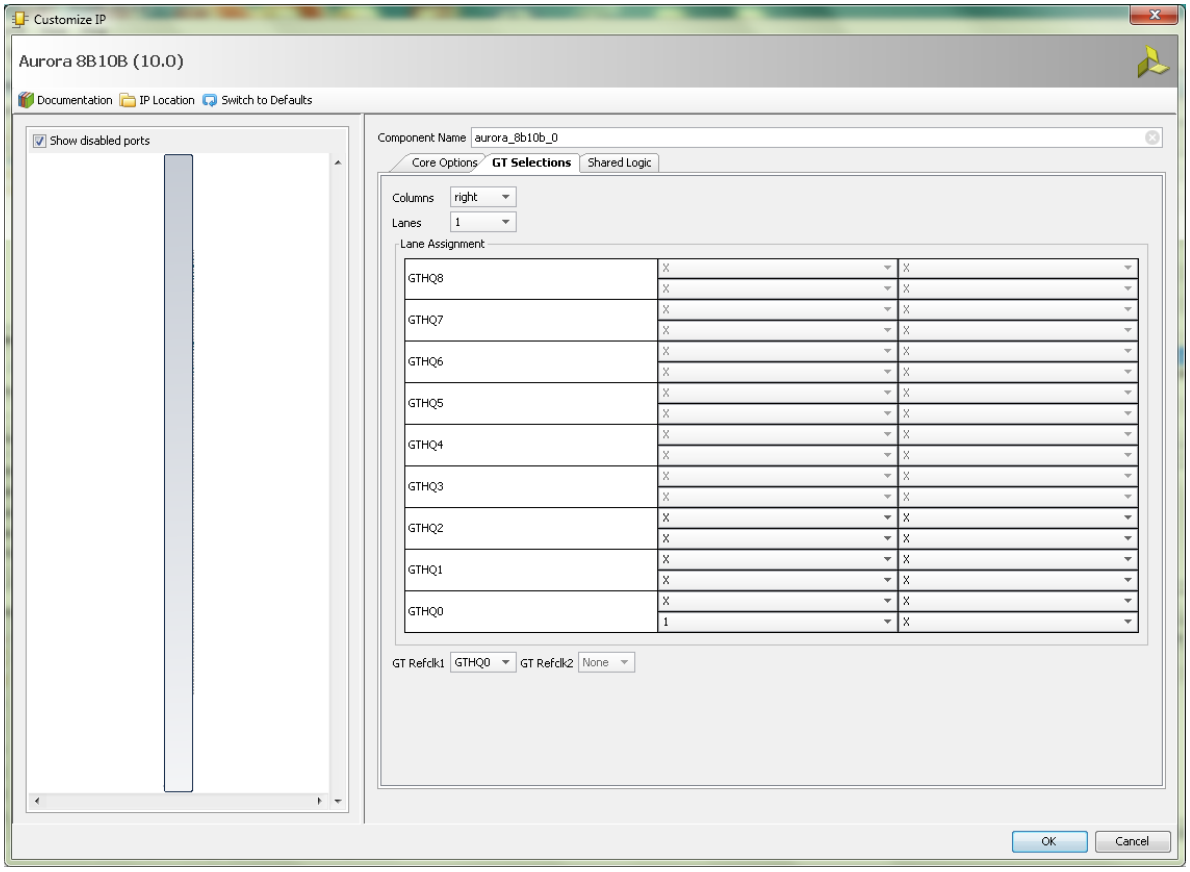
Lanes 1

Lane Assignment

GTXQ5	X	X
	X	X
GTXQ4	X	X
	X	X
GTXQ3	X	X
	X	X
GTXQ2	X	X
	X	X
GTXQ1	X	X
	1	X

GT Refclk1 GTXQ1 GT Refclk2 None

Figure 5-3: **Virtex-7 and Kintex-7 FPGA GTX Transceivers, GT Selections Tab**



Component Name aurora_8b10b_0

Columns right

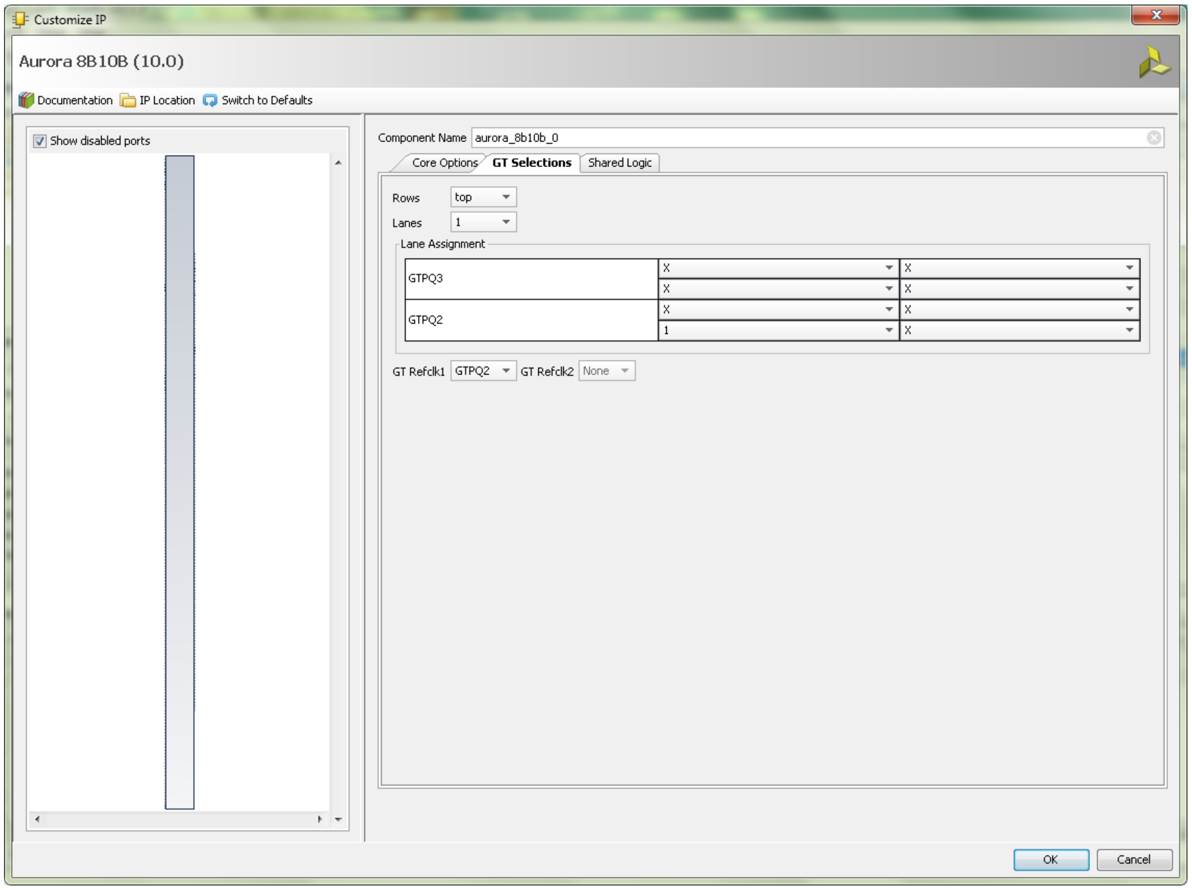
Lanes 1

Lane Assignment

GTHQ8	X	X
	X	X
GTHQ7	X	X
	X	X
GTHQ6	X	X
	X	X
GTHQ5	X	X
	X	X
GTHQ4	X	X
	X	X
GTHQ3	X	X
	X	X
GTHQ2	X	X
	X	X
GTHQ1	X	X
	X	X
GTHQ0	X	X
	1	X

GT Refclk1 GTHQ0 GT Refclk2 None

Figure 5-4: **Virtex-7 FPGA GTH Transceivers, GT Selections Tab**



Component Name aurora_8b10b_0

Rows top

Lanes 1

Lane Assignment

GTPQ3	X	X
	X	X
GTPQ2	X	X
	1	X

GT Refclk1 GTPQ2 GT Refclk2 None

Figure 5-5: **Artix-7 FPGA GTP Transceivers, GT Selections Tab**

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

Row Used

Select the appropriate row of transceivers used from the drop-down list. The row used is enabled only for Artix®-7 devices and is disabled for all other devices.

Default: top

Lanes

Select the number of lanes (GTP, GTX, or 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 Assignment

See the diagram in the information area in Figure 5-3. Two rows or four boxes represent a quad in Virtex-7, Kintex-7 FPGAs, and Artix-7 FPGAs. Each active box represents an available GTX, GTH or GTP transceiver. A tooltip is provided to specify which transceiver (for example, GTXE2_CHANNEL_X0Y0) is being implemented in hardware.

GT REFCLK1 and GT REFCLK2

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

- Default: GT REFCLK Source1 – GTPQ0; GT REFCLK Source2 - None for Artix-7 FPGA GTP transceivers
- Default: GT REFCLK Source1 – GTXQ0; GT REFCLK Source2 - None for Virtex-7 and Kintex-7 FPGA GTX transceivers
- Default: GT REFCLK Source1 – GTHQ0; GT REFCLK Source2 - None for Virtex-7 and Kintex-7 FPGA GTH transceivers
- GTXQ<n>/GTHQ<n>/GTPQ<n> change based on the selected device and package.

Core Generation

Click **OK** to generate the core. The modules for the Aurora 8B/10B core are written to the Vivado design tools project directory using the same name as the top level of the core. See Output Generation, page 85 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 design tools project with supporting files. These files are arranged in a predetermined directory structure under the project directory name provided to the IP catalog when the project is created as shown in this section.

For details, see "Generating IP Output Products" in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5].

Table 5-1 shows the files associated with the core.

Table 5-1: **Core Files**

Name	Description
<project_name>/<project_name>.srcs/sources_1/ip <instance_name>/<instance_name>.xci	Instance configuration file. Contains the parameters values of the instance. This is the source file for importing to another project.
<project_name>/<project_name>.srcs/sources_1/ip/ <instance_name>/<instance_name>.veo	Verilog instantiation wrapper
<project_name>/<project_name>.srcs/sources_1/ip/ <instance_name>/<instance_name>.vho	VHDL instantiation wrapper.
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_aurora_pkg.vhd (VHDL only) <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>_core.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_aurora_lane.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_axi_to_ll.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_channel_err_detect.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_channel_init_sm.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_chbond_count_dec.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_err_detect.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_global_logic.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name >/<instance_name>/src/<instance_name>_hotplug.[v]hd	Aurora 8B/10B source files

Table 5-1: **Core Files *(Cont'd)***

Name	Description
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_idle_and_ver_gen.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_lane_init_sm.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_ll_to_axi.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_rx_ll_pdu_datapath.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_rx_ll.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_sym_dec.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_sym_gen.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_sync_block.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_tx_ll_control.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_tx_ll_datapath.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/src/<instance_name>_tx_ll.[v]hd	Aurora 8B/10B source files (Continued)
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/gt/<instance_name>_tx_startup_fsm.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/gt/<instance_name>_rx_startup_fsm.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/gt/<instance_name>_gt.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/gt/<instance_name>_multi_gt.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/gt/<instance_name>_transceiver_wrapper.[v]hd	Verilog/VHDL wrapper files for the transceiver
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/<instance_name>_exdes.[v]hd	Example design top-level file
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/<instance_name>_reset_logic.[v]hd	Aurora 8B/10B reset logic
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/clock_module/<instance_name>_clock_module.[v]hd	Clock module source file

Table 5-1: **Core Files *(Cont'd)***

Name	Description
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/traffic_gen_check/<instance_name>_frame_check.[v]hd <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/traffic_gen_check/<instance_name>_frame_gen.[v]hd	Example design traffic generation and checker files
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/cc_manager/<instance_name>_standard_cc_module.[v]hd	Clock compensation module source file
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/simulation/<instance_name>_tb.[v]hd	Test bench file for example design
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>.[v]hd	Aurora 8B/10B IP core top level file
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>/example_design/<instance_name>_exdes.xdc	Aurora 8B/10B example design XDC constraints
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>_clocks.xdc	Aurora 8B/10B IP core xdc constraints
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/<instance_name>_ooc.xdc	XDC constraints file for Out-Of-Context (OOC) flow
<project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/xci/vio_7series.xci <project_name>/<project_name>.srcs/sources_1/ip/<instance_name>/xci/ila_7series.xci	Vivado lab tools xci files

Aurora IP core delivers the demonstration test bench for the example design. Simulation status is reported through messages. The TEST COMPLETED SUCCESSFULLY message signifies the completion of the example design simulation.

Note: The **Reached max. simulation time limit** message means that simulation was not successful See Appendix C, Debugging for more information.

Simulating the Duplex core is a single-step process after generating example design. Additional steps are required to simulate the simplex core. The following steps illustrate the simplex core partner generation and functional simulation.

1. Generate the Simplex IP core (IP1). The Dataflow Mode could be either TX-only Simplex or RX-only Simplex.
2. Generate the simplex partner IP core (IP2). The Dataflow Mode will be TX_only Simplex if RX_only Simplex is selected in step 1 or vice versa. The component name should be tx_*IP1 component name* or rx_*IP1 component name* based on the IP2 Dataflow Mode selection.
3. Generate an example design for IP1 (IP1_EXDES).
4. Generate an example design for IP2 (IP2_EXDES).
5. Add the IP2 core example design source files (IP2_EXDES) to the IP1 example design (IP1_EXDES). The source files to be included are IP core files, transceiver modules, top module, and the IP core top level wrapper. Make sure all the required files are added. This can be verified by checking each hierarchy and no questions mark is placed to any of the IP2 source files. The test bench of IP2_EXDES should not be included.
6. Click **Run Simulation**.
7. Perform the following steps:
   a. Vivado simulator: Enter the **`run all`** command in Vivado Tcl console to run the simulation.
   b. Questa® SIM simulator: Enter the **`run -all`** command in Questa SIM console to run the simulation.

Constraining the Core

This chapter provides information for constraining the Aurora core in to the Vivado® Design Suite.

Clock Frequencies

The Aurora 8B/10B core example design clock constraints can be grouped into following three categories:

- 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 of the Vivado IDE). The GT REFCLK value selected in the first page of the Vivado IDE is used to constrain the GT reference clock. The create_clock XDC command is used to constrain GT reference clocks.

- 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.

- init_clk 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.

Clock Management

Not Applicable

Clock Placement

Not Applicable

Banking

Not Applicable

Transceiver Placement

The set_property XDC command is used to constrain the GT transceiver location. This is provided as a tool tip on the second page of the Vivado IDE. A sample XDC is provided for reference.

I/O Standard and Placement

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

False Paths

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

Example Design XDC

The generated verilog example design is configured with a two-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 follows:

```
-----------------------
## 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 Gb/s) 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 Gb/s) 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/inst/gt_wrapper_i/aurora_8b10b_v9_0_0_multi_gt_i/gt0_aurora_8b10b_v
9_0_0_i/gthe2_i]
```

The preceding XDC is provided for reference. The example design XDC is created automatically when the core is generated from the Vivado design tools.

Simulation

For comprehensive information about Vivado® simulation components, as well as information about using supported third party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 7]

Aurora IP core delivers a demonstration test bench for the example design. The simulation status is reported through messages. The TEST COMPLETED SUCCESSFULLY message signifies the completion of the example design simulation.

Note: The `Reached max. simulation time limit` message means that simulation was not successful. See Appendix C, Debugging for more information.

Simulating the Duplex core is a single step process after generating an example design. Additional steps are required to simulate the simplex core. The following steps illustrate simplex core partner generation and functional simulation.

1. Generate the Simplex IP core (IP1). The Dataflow Mode can be either TX-only Simplex or RX-only Simplex.
2. Generate the Simplex partner IP core (IP2). The Dataflow Mode will be TX_only Simplex if RX_only Simplex is selected in step 1 or vice versa. The component name should be tx_<IP1 component name> or rx_<IP1 component name> based on the IP2 Dataflow Mode selection.
3. Generate an example design for IP1 (IP1_EXDES).
4. Generate an example design for IP2 (IP2_EXDES).
5. Add the IP2 core example design source files (IP2_EXDES) to the IP1 example design (IP1_EXDES). The source files to be included are the IP core files, transceiver modules, a top module, and an IP core top-level wrapper. Make sure all the required files are added. This can be verified by checking each hierarchy to see that no question mark is placed on any of the IP2 source files. The test bench of IP2_EXDES should not be included.
6. Click **Run Simulation**
   a. Vivado simulator: Enter the **run all** command in the Vivado design tools Tcl Console to run the simulation.
   a. Questa® SIM simulator: Enter the **run -all** command in the Questa SIM console for to run the simulation.

For more information, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5].

Synthesis and Implementation

For more details about synthesis and implementation, see “Synthesizing IP” and “Implementing IP” in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5].

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 or GTX transceiver
  - AXI4-Stream interface
- A demonstration test bench to simulate two instances of the example design

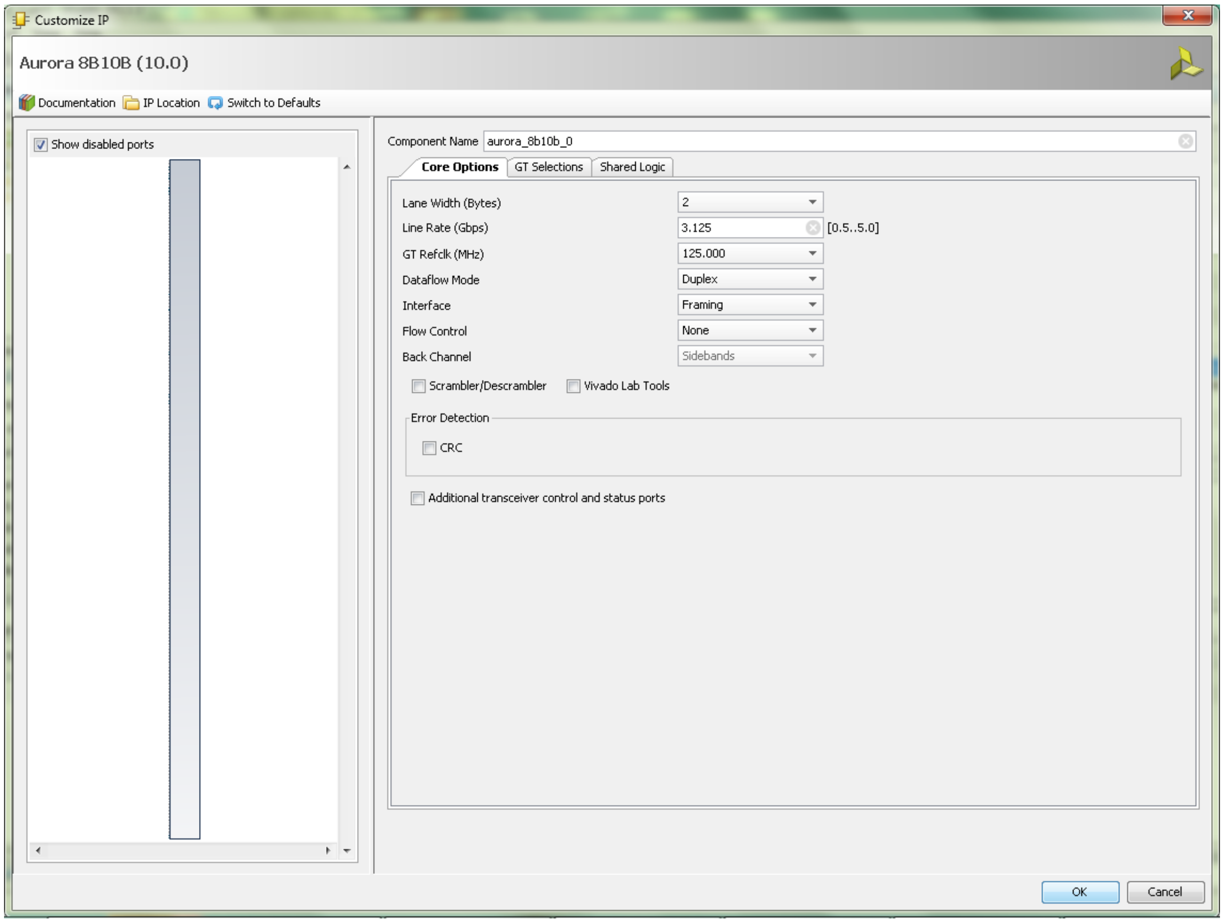
The Aurora 8B/10B example design has been tested with the Vivado® Design Suite for synthesis and Mentor Graphics Questa® SIM for simulation.

Generating the Core

To generate an Aurora 8B/10B core with default values using the Vivado design tools:

1. Start the Vivado tools from a required directory. For help starting and using the Vivado design tools, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5].
2. Choose **Create New Project New > Project > Next**.
3. Type the new project name and enter the project location.
4. Select **Project Type** as **RTL Project** and click **Next**.
5. Select the part as xc7vx485tffg1157-1
6. After creating the project, click **IP catalog** in the Project Manager panel.
7. Locate the Aurora 8B/10B v10.0 core in the IP catalog taxonomy tree under: / Communication_&_Networking/Serial_Interfaces

8. Double-click the core.
9. Click **OK**.



Component Name aurora_8b10b_0

Lane Width (Bytes) 2
Line Rate (Gbps) 3.125 [0.5..5.0]
GT Refclk (MHz) 125.000
Dataflow Mode Duplex
Interface Framing
Flow Control None
Back Channel Sidebands

☐ Scrambler/Descrambler ☐ Vivado Lab Tools

Error Detection
☐ CRC

☐ Additional transceiver control and status ports

Figure 8-1: **Customize IP Vivado IDE**

Implementing the Example Design

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

Note: You need to specify LOC and IO standards in XDC for all input and output ports of the design.

Detailed Example Design

This chapter contains information about the provided example design in the Vivado® Design Suite.

Directory and File Contents

See Output Generation, page 85 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 85.

Figure 9-1 illustrates the block diagram of the example design for a full-duplex core. Table 9-1 describes the ports of the example design.

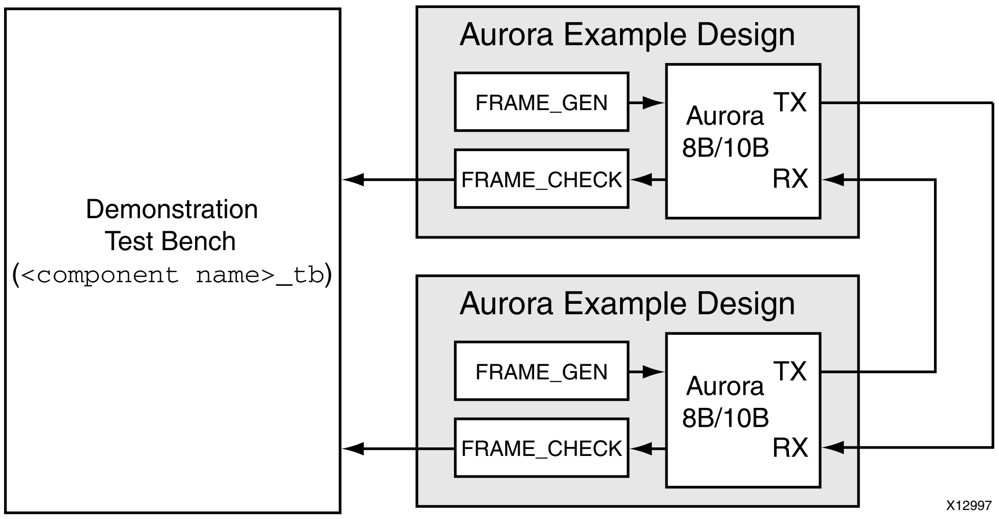


Figure 9-1: **Example Design**

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 9-1: **Example Design I/O Ports**

Port	Direction	Description
rxn[0:*m*–1]	Input	Negative differential serial data input pin.
rxp[0:*m*–1]	Input	Positive differential serial data input pin.
txn[0:*m*–1]	Output	Negative differential serial data output pin.
txp[0:*m*–1]	Output	Positive differential serial data output pin.
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 Functional Description, page 35 for details about the reference clocks.
<core error signals>	Output	The error signals from the Aurora 8B/10B core Status and Control interface are brought to the top level of the example design and registered. See Status, Control, and the Transceiver Interface, page 62 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 62 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 or GTX transceiver they use. Simplex cores have a separate lane up signal per GTP or GTX transceiver they use for each channel direction supported. See Status, Control, and the Transceiver Interface, page 62 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 62 for details.

Test Bench

The Aurora core delivers a demonstration test bench for the example design. This chapter describes the Aurora test bench and its functionality. The test bench consist of the following modules:

- Device Under Test (DUT)
- Clock and reset generator
- Status monitor

The Aurora test bench components can change based on the selected Aurora core configurations, but the basic functionality remains the same for all of the core configurations.

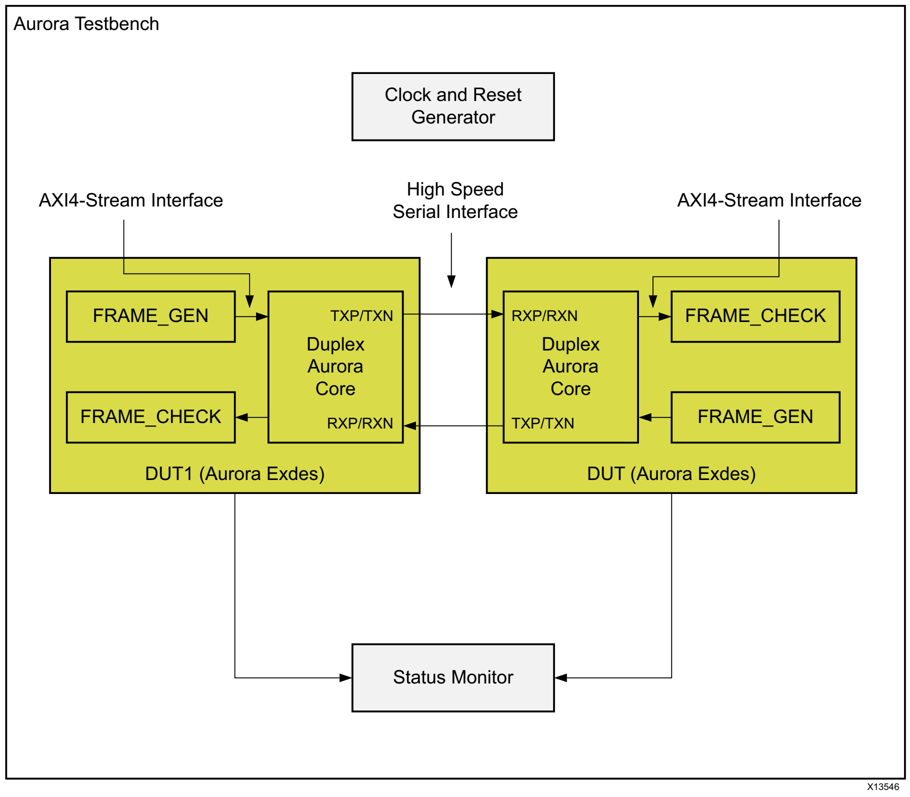


Figure 10-1: **Aurora Test Bench for Duplex Configuration**

The Aurora test bench environment connects the Aurora Duplex core in loopback using a high-speed serial interface. Figure 10-1 shows the Aurora test bench for the Duplex configuration.

The test bench looks for the state of the channel, then integrity of the user data and UFC data for a predetermined simulation time. The `channel_up` assertion message indicates that link training and channel bonding (in case of multi-lane designs) are successful. The counter is maintained in the FRAME_CHECK module to track the reception of the erroneous data. The test bench flags an error when erroneous data is received.

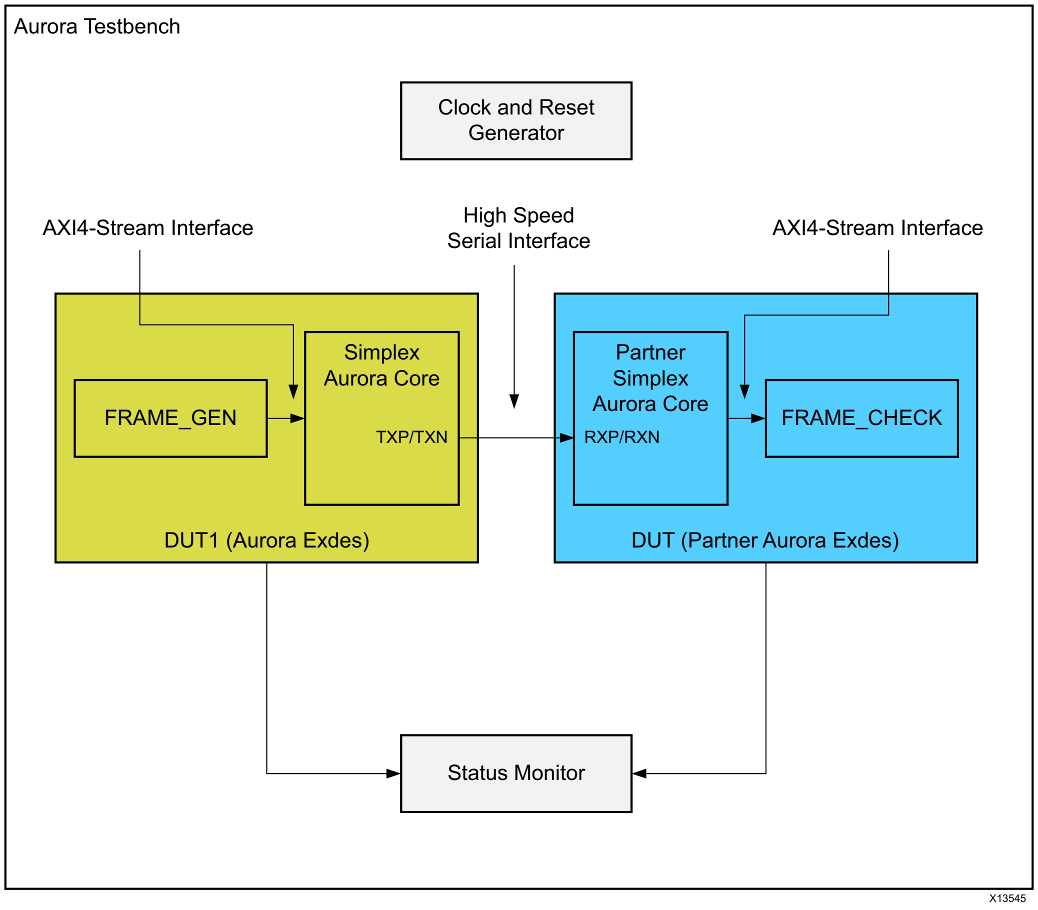


Figure 10-2: **Aurora Test Bench for Simplex Configuration**

The Aurora test bench environment connects the Aurora Simplex core to the partner Simplex Aurora core using the high-speed serial interface. Figure 10-2 shows the Aurora test bench for the Simplex configuration where DUT1 is configured as TX-only Simplex and DUT2 is configured as RX-only Simplex.

The test bench looks for the state of the transmitter channel and receiver channel and then the integrity of the user data for a predetermined simulation time. The `tx_channel_up` and `rx_channel_up` assertion messages indicate that link training and channel bonding (in case of multi-lane designs) are successful.

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 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 KC724, VC7203 and AC722 boards are used for Aurora 8B/10B core verification on hardware.

Migrating and Upgrading

This appendix contains information about migrating a design from ISE® to the Vivado® Design Suite, and for upgrading to a more recent version of the IP core. For customers upgrading in the Vivado Design Suite, important details (where applicable) about any port changes and other impact to user logic are included.

Migrating to the Vivado Design Suite

For information about migrating to the Vivado Design Suite, *see the ISE to Vivado Design Suite Migration Guide* (UG911) [Ref 8].

Upgrading in the Vivado Design Suite

In the latest revision of the core, there have been several changes that make the core pin-incompatible with the previous versions. These changes were required as part of the general one-off hierarchical changes to enhance the customer experience and are not likely to occur again.

Shared Logic

As part of the hierarchical changes to the core, it is now possible to have the core itself include all of the logic which can be shared between multiple cores, which was previously exposed in the example design for the core.

If you are updating a previous version to a new one with shared logic, there is no simple upgrade path and it is recommended to consult the Shared Logic sections of this document for more guidance.

Migrating LocalLink-based Aurora Cores to the AXI4-Stream Aurora Core

Introduction

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

Prerequisites

- Vivado design tools build containing the Aurora 8B/10B v10.0 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 *LogiCORE IP Aurora 8B/10B v5.3 Data Sheet* (DS637) [Ref 9] and *LogiCORE IP Aurora 8B/10B v5.3 User Guide* (UG353) [Ref 10] for reference.
- Migration guide (this appendix)

Limitations

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

- 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.
- 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.

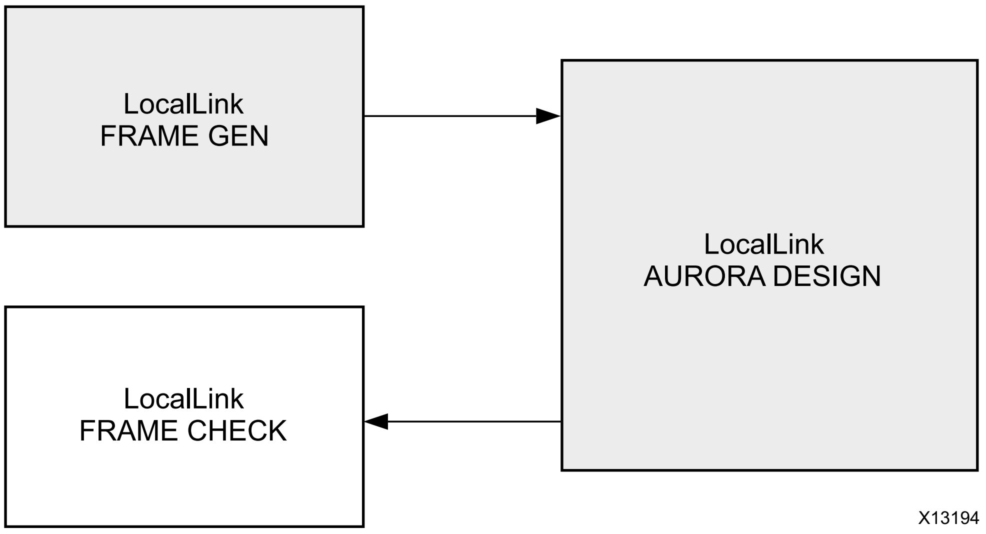


Figure B-1: **Legacy Aurora Example Design**

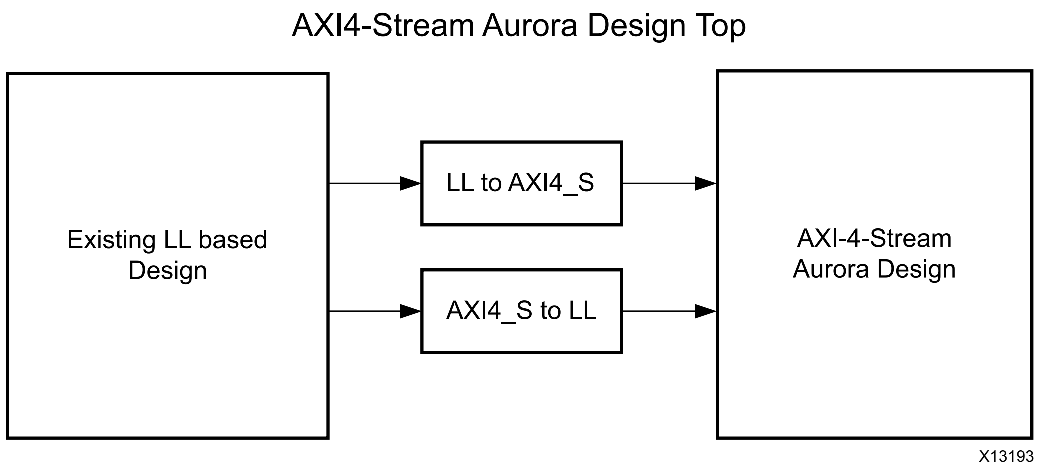


Figure B-2: **AXI4-Stream Aurora Example Design**

Migration Steps

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

Simulate the Core

1. Run the `vsim -do simulate_mti.do` file from the `/simulation/functional` directory.
2. Questa® SIM 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 Questa SIM console providing the status of the test.

Implement the Core

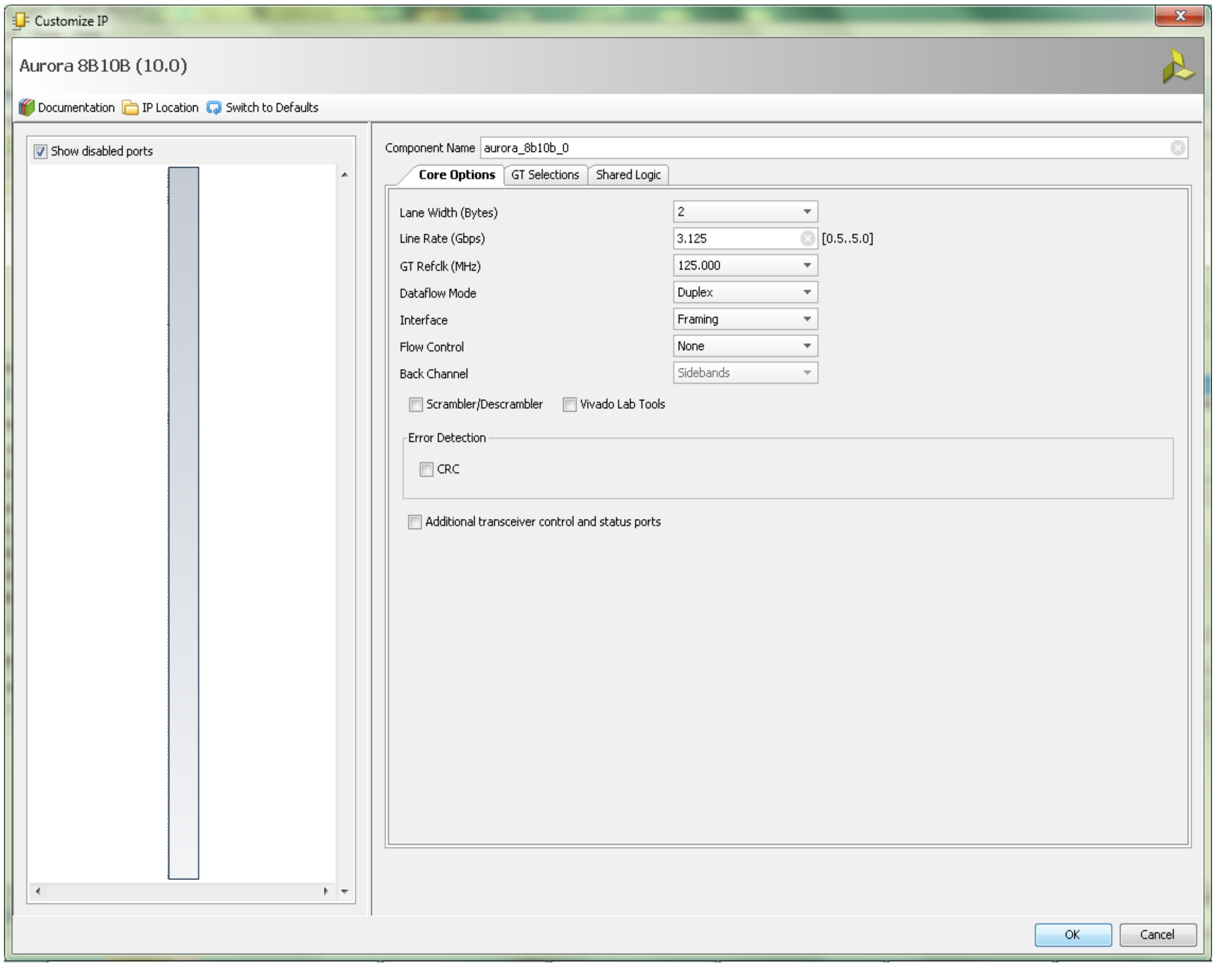
1. Run `./implement.sh` (for Linux) from the `/implement` directory.
2. The implement script compiles the core, runs through the Vivado design tools, and generates a bit file and netlist for the core.

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_3 version of the core.
2. See Figure B-2, page 103 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 103.
6. The latest AXI4-Stream Aurora core can be plugged into any existing LL integrated design environment.

Vivado IDE Changes

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



Component Name aurora_8b10b_0

Lane Width (Bytes) 2
Line Rate (Gbps) 3.125 [0.5..5.0]
GT Refclk (MHz) 125.000
Dataflow Mode Duplex
Interface Framing
Flow Control None
Back Channel Sidebands

Scrambler/Descrambler
Vivado Lab Tools

Error Detection
CRC

Additional transceiver control and status ports

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

Debugging

This appendix provides information on using resources available on the Xilinx Support website, available debug tools, and a step-by-step process for debugging designs that use the Aurora 8B/10B core. This appendix uses a flow diagram to guide you through the debug process.

Finding Help on Xilinx.com

To help in the design and debug process when using the Aurora 8B/10B core, the Xilinx Support web page (www.xilinx.com/support) contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the Aurora 8B/10B core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page (www.xilinx.com/support) or by using the Xilinx Documentation Navigator.

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

To see the available documentation by device family, navigate to www.xilinx.com/support.

To see the available documentation by solution:

1. Navigate to www.xilinx.com/support.
2. Select the Documentation tab located at the top of the web page.

   This is the Documentation Center where Xilinx documentation is sorted by Devices, Boards, IP, Design Tools, Doc Type, and Topic.

Solution Centers

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

Answer Records

Answer Records include information on commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a product. Answer Records are created and maintained daily ensuring users have access to the most up-to-date information on Xilinx products. Answer Records can be found by searching the Answers Database.

Answer Records for this can also be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as

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

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

To use the Answers Database Search:

1. Navigate to www.xilinx.com/support. The Answers Database Search is located at the top of this web page.
2. Enter keywords in the provided search field and select **Search**.
    - Examples of searchable keywords are product names, error messages, or a generic summary of the issue encountered.
    - To see all answer records directly related to the Aurora 8B/10B core, search for the phrase "Aurora 8B10B"

Master Answer Record for the Aurora 8B/10B Core

AR: 54367

Contacting Xilinx 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.

To contact Technical Support:

1. Navigate to www.xilinx.com/support.
2. Open a WebCase by selecting the WebCase link located under Additional Resources.

When opening a WebCase, include:

- Target FPGA including package and speed grade
- All applicable Vivado® design tools, synthesis (if not XST), and simulator software versions
- The XCI file created during Aurora 8B/10B core generation

  This file is located in the directory targeted for the Vivado design tools project. Additional files might be required based on the specific issue. See the relevant sections in this debug guide for further information on specific files to include with the WebCase.

Note: Access to WebCase is not available in all cases. Login to the WebCase tool to see your specific support options.

Debug Tools

There are many tools available to address Aurora 8B/10B core design issues. It is important to know which tools are useful for debugging various situations.

Transceiver Wizard

Serial transceiver attributes play a vital role in Aurora 8B/10b core functionality and performance. See Appendix D, Generating a Wrapper File from the Transceiver Wizard to get the latest attribute updates for the core.

Vivado Lab Tools

Vivado lab tools insert logic analyzer and virtual I/O cores directly into your design. Vivado lab tools allow you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature represents the functionality in the Vivado IDE that is used for logic debugging and validation of a design running in Xilinx devices in hardware.

The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 3.0 (and later versions)
- VIO 3.0 (and later versions)

See *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 11].

Reference Boards

Various Xilinx development boards support the Aurora 8B/10B core. These boards can be used to prototype designs and establish that the core can communicate with the system.

- 7 series FPGA evaluation boards
  - KC705
  - KC724
  - VC7203
  - AC722
  - VC707

Simulation Debug

Lanes and Channel do not come up in simulation

- The quickest way to debug these problems is to view the signals from one of the serial transceivers instances that are not working.
- Make sure that the serial transceiver reference clock and user clocks are all toggling.
- Check to see that `txoutclk` from the serial transceiver 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 6–9 µs for lane up and channel up. You might need to wait longer for simplex designs.
- Make sure that `txn` and `txp` are toggling. If they are not, make sure you have waited long enough and make sure 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 will not be able to initialize.
- Be sure you do not have the `power_down` signal asserted.
- Make sure the `txn` and `txp` signals from each serial transceiver are connected to the appropriate `rxn` and `rxp` signals from the corresponding serial transceiver on the other side of the channel.
- If you are 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. You should hold this reset for a few cycles.

  The following code can be used 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 serial transceivers on each side of the channel are connected in the correct order.

Channel comes up in simulation but not in hardware

- Both `reset` inputs are active-High. Make sure the `reset` polarity is taken care in the hardware.
- Make sure the `refclk` frequency is exactly same as the aurora core generated for.
- 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 serial 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 serial transceiver attributes which provide "Zero" BER in IBERT.
- Burst of soft errors results in hard error and re-initializes the channel. Set ENABLE_SOFT_ERR_MONITOR to 0 in <component name>_err_detect module to disable hard error assertion from soft errors.

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 currently driven Low. `s_axi_nfc_req` and `s_axi_ufc_tx_req` are active-High: if they are High, `s_axi_tx_tvalid` will stay Low because the channel will be allocated for flow control.
- Make sure `warn_cc` and `do_cc` are not being driven High continuously. Whenever `do_cc` is High on a positive clock edge, the channel is used to send clock correction characters, so `s_axi_tx_tvalid` is deasserted.
- If you have NFC enabled, make sure the design on the other side of the channel did 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. See Native Flow Control Interface in Chapter 2 for more details.

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

- If you are using the AXI4-Stream interface, make sure you are writing data correctly. The most common mistake is to assume words are written without looking at `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 while compiling the design

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

Next Step

If the debug suggestions listed previously do not resolve the issue, open a support case to have the appropriate Xilinx expert assist with the issue.

To create a technical support case in WebCase, see the Xilinx website at:

www.xilinx.com/support/clearexpress/websupport.htm

Items to include when opening a case:

- Detailed description of the issue and results of the steps listed previously.
- Attach a VCD or WLF dump of the simulation.

To discuss possible solutions, use the Xilinx User Community: forums.xilinx.com/xlnx/

Hardware Debug

Aurora 8B/10B core has an option to use the Vivado lab tools in the example design. The example design has a VIO core instantiated and connected with important status and control signals.

RECOMMENDED: *Ensure that the serial transceiver attributes are updated. See Appendix D, Generating a Wrapper File from the Transceiver Wizard for instructions and information about updating the serial transceiver attribute settings. This section provides a debug flow diagram for resolving some of the most common issues.*

Figure C-1 shows the various steps for performing a hardware debug.

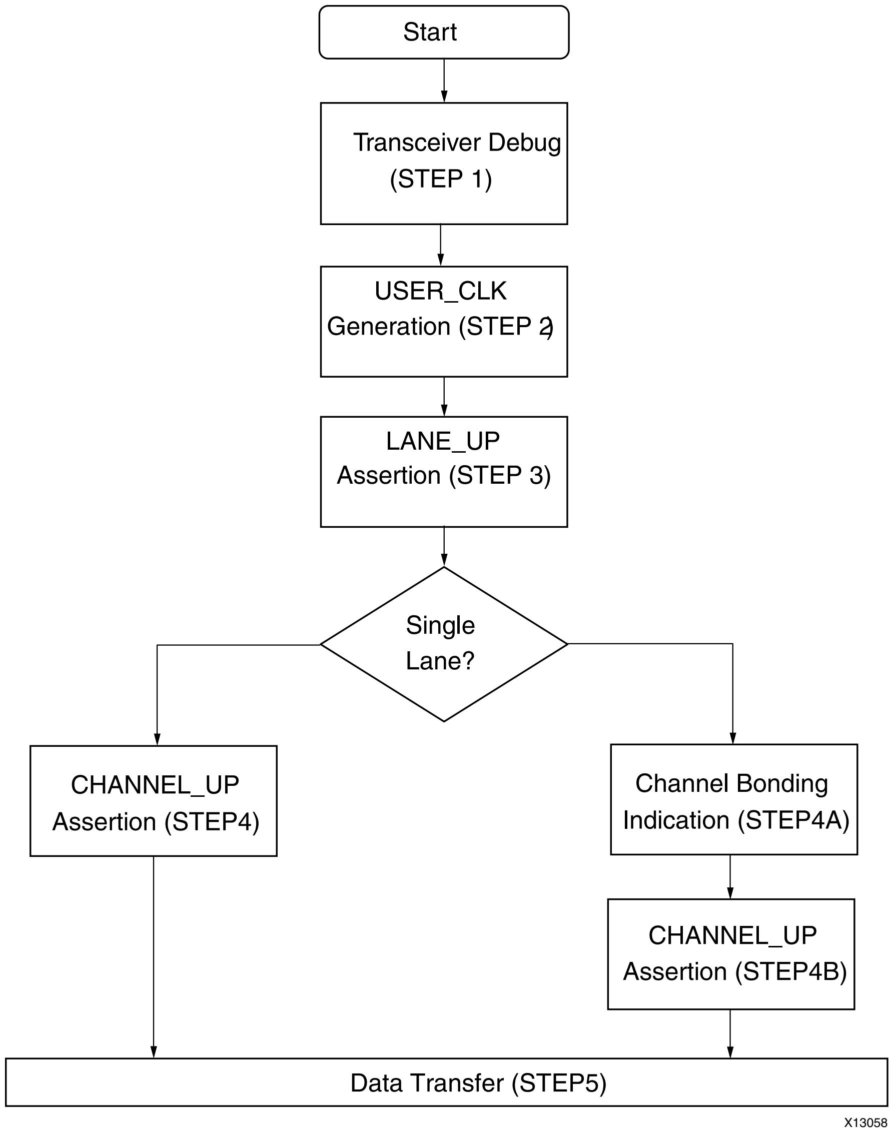

Start
Transceiver Debug (STEP 1)
USER_CLK Generation (STEP 2)
LANE_UP Assertion (STEP 3)
Single Lane?
CHANNEL_UP Assertion (STEP4)
Channel Bonding Indication (STEP4A)
CHANNEL_UP Assertion (STEP4B)
Data Transfer (STEP5)
X13058

Figure C-1: **Flow Chart**

STEP 1: Transceiver Debug

Transceiver being the critical building block in aurora core, debugging and ensuring proper operation of transceiver attains utmost importance. Figure C-2 shows the steps involved for debugging transceiver related issues.

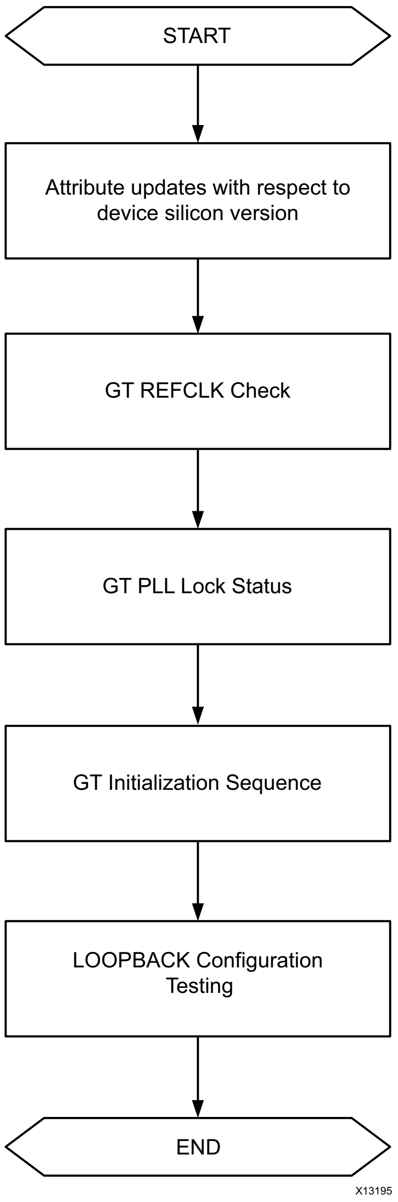

START
Attribute updates with respect to device silicon version
GT REFCLK Check
GT PLL Lock Status
GT Initialization Sequence
LOOPBACK Configuration Testing
END
X13195

Figure C-2: **GT Debug Flow Chart**

1. Attribute updates with respect to device silicon version

   Transceiver attributes must match with the silicon version of the device being used in the board. Apply all the applicable workaround and Answer Records given for the respective silicon version.

2. GT REFCLK Check

   A low jitter differential clock must be provided to the transceiver reference clock. Connecting onboard differential clock to the transceiver will narrow down to the issue with external clock generation and/or external clock cables connected to transceiver.

3. GT PLL Lock Check

   Transceiver locks into incoming GT REFCLK and asserts the `plllock` signal. This signal is available as `tx_lock` signal in Aurora example design. Make sure that GT PLL attributes are set correctly and transceiver generates the `txoutclk` with expected frequency for the given line rate and datapath width options. It must be noted that aurora core uses Channel PLL (CPLL) in the generated core for Virtex®-7 and Kintex®-7 FPGA GTX and GTH transceivers and PLL0/PLL1 for Artix®-7 FPGA GTP transceivers.

4. GT Initialization Sequence

   Aurora core uses sequential mode as reset mode and all of the transceiver components are being reset sequentially one after another. The `txresetdone` and `rxresetdone` signals are asserted at the end of the transceiver initialization. In general, `rxresetdone` assertion will take longer time compare to TXRESETDONE assertion. Make sure, `gt_reset` signal pulse width duration matches with respective transceiver guideline. The `txresetdone` and `rxresetdone` signals are available in the Aurora example design to monitor.

5. LOOPBACK Configuration Testing

   Loopback modes are specialized configurations of transceiver datapath. The `loopback` port at Aurora example design will control the loopback modes. Four loopback modes are available and refer respective transceiver UG for guidelines and more information. Figure C-3 illustrates a loopback test configuration with four different loopback modes.

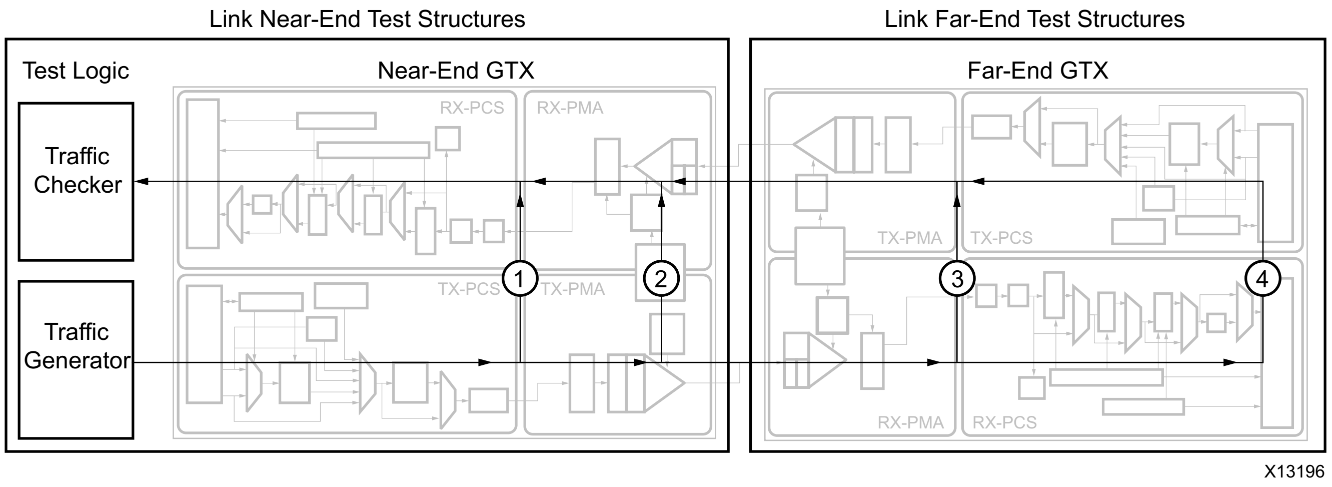


Figure C-3: **Loopback Testing Overview**

STEP 2: USER_CLK Generation

GT generates `txoutclk` based on the line rate and lane-width parameters. The `user_clk` is generated from `txoutclk` and the Aurora 8B/10B core uses as an FPGA logic clock.

Therefore, you must check that `user_clk` is generated properly with the expected frequency from `txoutclk`. If `user_clk` frequency is not in the expected range, you must check the frequency of the GT reference clock being applied. Additionally, you also need to check GT PLL attributes to make sure the generated `txoutclk` frequency is correct.

STEP 3: LANE_UP Assertion

The `lane_up` assertion indicates the communication between GT and its channel partner is established and link training is successful. This signal is connected to VIO for monitoring. You must bring LANE_INIT_SM module FSM state signals to debug if `lane_up` is not asserted. See the Lane Initialization Procedure in the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 3] for `lane_up` assertion.

STEP 4: CHANNEL_UP Assertion

The verification sequence defined in the Aurora 8B/10B protocol being transferred between channel partners and successful reception of four verification sequences are the criteria for `channel_up` assertion. This is connected to VIO as channel_up_i. You must bring CHANNEL_INIT_SM module FSM state signals to debug if `channel_up` is not asserted. See the Channel Verification Procedure in the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 3] for `channel_up` assertion.

STEP 4A: Channel Bonding Assertion

Channel bonding is necessary for a multi-lane Aurora design. Channel bonding is performed by GT and the required logic is present in the transceiver_wrapper module. Make sure that channel bonding level, master and slave connections are correct. See the Channel Bonding Procedure in the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 3] for `channel_up` assertion.

STEP 4B: CHANNEL_UP Assertion

This step is same as STEP 4 described previously.

STEP 5: Data Transfer

After `channel_up` is asserted, the Aurora 8B/10B core is ready to do a data transfer. Data errors can be monitored as `err_count_r` signal in VIO. The `tx_d` and `rx_d` signals are connected to monitor the data transfer. Apart from this, `soft_err`, `hard_err` and `frame_err` are also connected to VIO. FIFO is used by GT for clock correction and channel bonding. Overflow and underflow of this GT FIFO results in `hard_err` (HARD_ERR). You need to tune CLK_COR_MIN_LAT and CLK_COR_MAX_LAT attributes of the GT to correct GT FIFO overflow/underflow errors.

Note: The ENABLE_SOFT_ERR_MONITOR parameter is available in the err_detect module under the src directory to control the leaky bucket algorithm. This parameter can be to set to 0 to disable the leaky bucket algorithm for debug purposes.

Additional Assistance

If the debug suggestions listed previously do not resolve the issue, open a support case to have the appropriate Xilinx expert assist with the issue.

To create a technical support case in WebCase, see the Xilinx website at:

www.xilinx.com/support/clearexpress/websupport.htm

Items to include when opening a case:

- Detailed description of the issue and results of the steps listed previously.
- Attach Vivado lab tools captures taken in the previous steps.

To discuss possible solutions, use the Xilinx User Community: forums.xilinx.com/xlnx/

Interface Debug

AXI4-Stream Interfaces

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

- If transmit `s_axi_tx_tready` is stuck Low following the `s_axi_rx_tvalid` input being asserted, the core cannot send data.
- If the receive `m_axi_rx_tvalid` is stuck Low, the core is not receiving data.
- Check that the `user_clk` input is connected and toggling.
- Check that the AXI4-Stream waveforms are being followed. See Figure 3-12 for data transfer and Figure 3-16 for data reception.
- Check core configuration.

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.

RECOMMENDED: *Xilinx strongly recommends that you update the transceiver wrapper file in the Xilinx Vivado® 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.

Virtex-7, Kintex-7, and Artix-7 FPGA Wrapper Compatibility

Use these steps to generate the transceiver wrapper file using the 7 series FPGAs transceivers wizard:

1. Using the IP catalog, run the latest version of the 7 Series FPGAs Transceivers Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 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 `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.

Handling Timing Errors

This appendix describes how to handle timing errors resulting from transceivers that are far apart. The Aurora 8B/10B core allows you 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)
- 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:

- Select the transceivers consecutively.

  Use the Lane Assignment in the Aurora 8B/10B Vivado® IDE 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.
- Use the Strategies options provided for implementation in the Vivado Design Suite. See the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 5] for instructions on how to use Vivado strategies.

Additional Resources

Xilinx Resources

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

www.xilinx.com/support.

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

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

References

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

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

1. *7 Series FPGAs GTP Transceivers User Guide* (UG482)
2. *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476)
3. *Aurora 8B/10B Protocol Specification v2.2* (SP002)
4. *AMBA AXI4-Stream Protocol Specification* (v1.0)
5. *Vivado Design Suite User Guide: Designing with IP* (UG896)
6. *Vivado Design Suite User Guide: Getting Started* (UG910)
7. *Vivado Design Suite User Guide - Logic Simulation* (UG900)
8. *ISE to Vivado Design Suite Migration Guide* (UG911)
9. *LogiCORE IP Aurora 8B/10B v5.3 Data Sheet* (DS637)
10. *LogiCORE IP Aurora 8B/10B v5.3 User Guide* (UG353)
11. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)

12. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)

To search for Xilinx documentation, go to www.xilinx.com/support.

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.
10/16/12	2.0	This release supports core version 8.3 with Vivado Design Suite v2012.3 and ISE® Design Suite v14.3. Major changes include: • Updated screen captures for Figures 5-1, 5-2, 7-2, 8-1, 8-2, 8-3, 8-4, 10-2, and B-3. • Added steps for Generating the Core in Chapter 7. • Added Artix®-7 device support. • Added GTH transceiver support. • Added LOOPBACK[2:0] and GT_RESET ports to Table 2-22. • Replaced IBUFDS_GTXE1 to IBUFDS_GTE2 in Figure 3-2. • Removed Design Constraints section in Chapter 6. • Added Clock Frequencies, I/O Placement, and I/O Standard and Placement sections.
12/18/12	2.0.1	• Updated for Vivado Design Suite v2012.4 and ISE Design Suite v14.4. • Modified maximum and minimum latency. • Added many new signals to Table 2-22, Transceiver Ports. • Updated screen captures in Chapter 5, Chapter 7, and Appendix B. • Modified Appendix C, Debugging
03/20/13	3.0	• Updated for Vivado Design Suite and core version 11.0 • Modified Appendix C, Debugging with transceiver debug details. • Updated screen captures in Chapter 5, Chapter 7, and Appendix B. • Removed ISE, CORE Generator™, UCF, Virtex®-6, and Spartan®-6 material. • Updated Reset waveforms. • Updated Directory and File Structure. • Created lowercase ports for Verilog.

Date	Version	Revision
06/19/2013	9.1	• Revision number advanced to 9.1 to align with core version number. • Updated for Vivado Design Suite v2013.2 and ISE Design Suite v14.6. • Aurora 8B10B v9.0 core is updated to Aurora 8B10B v9.1 based on revision guidelines.
10/02/2013	10.0	• Added new chapters: Simulation, Test Bench and Synthesis and Implementation. • Added shared logic and transceiver debug features. • Updated directory and file structure. • Updated resource utilization tables. • Added information about hot-plug logic. • Updated screen captures for Figures 5-1, 5-2, 5-3, 5-4, 5-5, 8-1 and B-3. • Changed all upper case signal names to lowercase. • Updated Migrating and Upgrading appendix.

Notice of Disclaimer