Aurora 8B/10B v10.3

LogiCORE IP Product Guide

Vivado Design Suite

PG046 October 1, 2014

XILINX
ALL PROGRAMMABLE™

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 UltraScale™ architecture, Zynq®-7000 All Programmable SoC, Artix®-7, Kintex®-7 and Virtex®-7 families.

Features

- General-purpose data channels with throughput range from 480 Mb/s to 84.48 Gb/s
- Supports any of up to 16 bonded Virtex-7 and Kintex-7 FPGA GTX and GTH transceivers, 8 bonded Artix-7 FPGA GTP transceivers and 16 bonded UltraScale architecture GTH transceivers
- Aurora 8B/10B protocol specification v2.3 compliant
- Low resource cost (see Resource Utilization, page 10)
- Easy-to-use framing and flow control
- Automatically initializes and maintains the channel
- Full-duplex or simplex operation
- AXI4-Stream (framing) or streaming user interface
- 16-bit additive scrambler/descrambler
- 16-bit or 32-bit Cyclic Redundancy Check (CRC) for user data
- Hot-plug logic
- Configurable DRP/INIT clock

LogiCORE IP Facts Table					
Core Specifics					
Supported Device Family(1)	UltraScale architecture, Zynq-7000, 7 Series				
Supported User Interfaces	AXI4-Stream				
Resources(2)	**LUTs**	**FFs**	**DSP Slices**	**Block RAMs**	**Max. Frequency(3)**
Config1(3)	342	463	0	0	330 MHz
Provided with Core					
Design Files	RTL				
Example Design	Verilog and VHDL(4)				
Test Bench	Verilog and VHDL(4)				
Constraints File	Xilinx Design Constraints (XDC)				
Simulation Model	Not Provided				
Supported S/W Driver	N/A				
Tested Design Flows(5)					
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 9.
4. Verilog only source code delivered for UltraScale architecture.
5. 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 UltraScale™ architecture GTH transceivers, Kintex®-7, Virtex®-7 FPGA GTX and GTH transceivers, Artix®-7 FPGA GTP transceivers, and Zynq®-7000 device GTX and GTP transceivers. The Aurora 8B/10B 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 (Figure 1-1) 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 and is used to transfer data between devices using one or many transceivers.

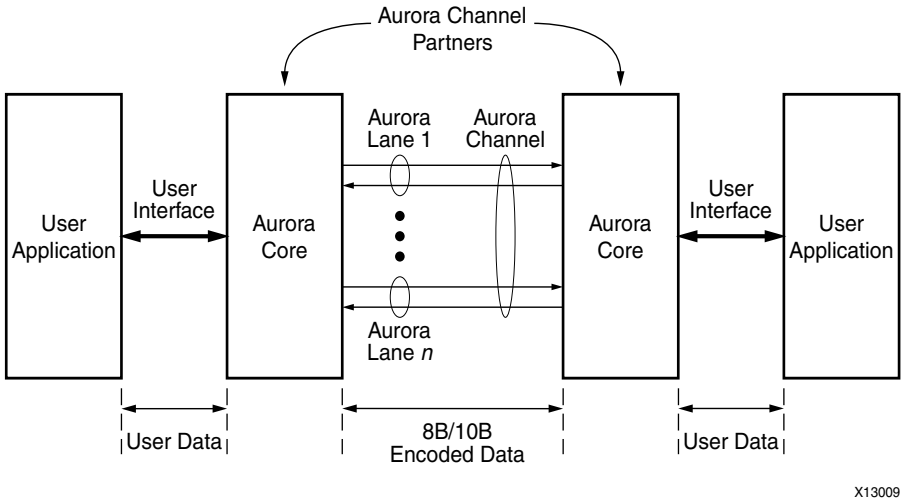


Figure 1-1: **Aurora 8B/10B Channel Overview**

Aurora 8B/10B cores automatically initialize a channel when they are connected to an Aurora channel partner and 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* can be used to reduce the rate of incoming data or to send brief, high-priority messages through the channel.

Streams are single, unending frames. In the absence of data, 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 8B/10B 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, page 36, carefully.*

Consult the PCB design requirements information in:

- *UltraScale FPGAs GTH Transceivers User Guide* (UG576) [Ref 1]
- *7 Series FPGAs GTP Transceivers User Guide* (UG482) [Ref 2]
- *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476) [Ref 3]

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

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)**: The Aurora protocol provides alternate ways to perform unidirectional channel initialization making possible the use of the GTP, GTX, and GTH transceivers in the absence of a back channel 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 8B/10B 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 Aurora 8B/10B core licensing 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.

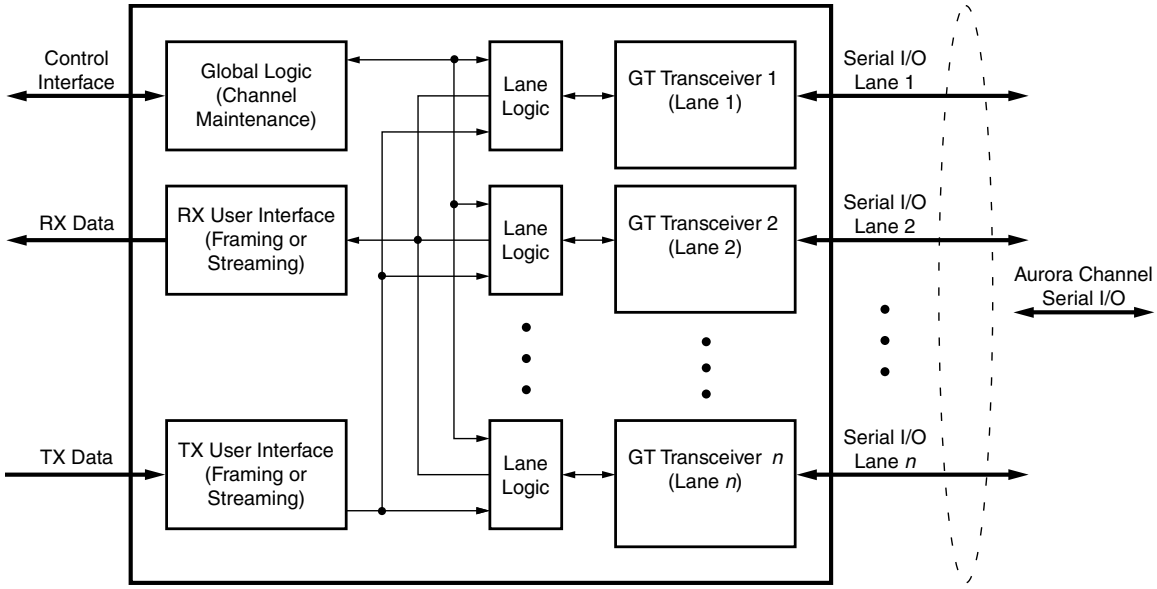


Figure 2-1: **Aurora 8B/10B Core Block Diagram**

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

- **Lane logic**: Each GTP, GTX, or GTH transceiver (hereinafter called transceiver) is driven by an instance of the lane logic module, which initializes each individual transceiver and handles the encoding and decoding of control characters and error detection.
- **Global logic**: The global logic module performs the bonding and verification phases of channel initialization. During operation, 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 AXI4-Stream RX user interface moves data from the channel to the application and performs flow control functions.

- **TX user interface**: The AXI4-Stream TX user interface moves data from the application to the channel and 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 channel partners). This interface is normally driven by a standard clock compensation manager module provided with the Aurora 8B/10B core. This module can be turned off, or the interface can be driven by custom logic.

Performance

Maximum Frequencies

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

The core GT reference clock for the Aurora 8B/10B cores listed in Table 2-1, page 11 through Table 2-4, page 12 runs 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 transceivers. The PE pipeline delay increases as the AXI4-Stream interface width increases. The transceiver delays are dependent on the features and attributes of the selected 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 transceiver logic and protocol engine (PE) logic which is implemented in the FPGA programmable 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 Datapath

Figure 2-2 illustrates the latency of the datapath. Latency can vary based on the transceiver(s) used in the design and the IP configuration.

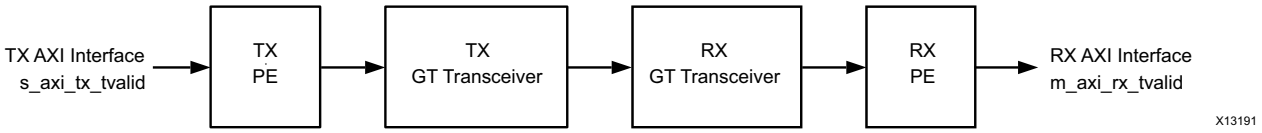


Figure 2-2: **Latency of the Data Path**

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 for the default core configuration (see Figure 2-3).

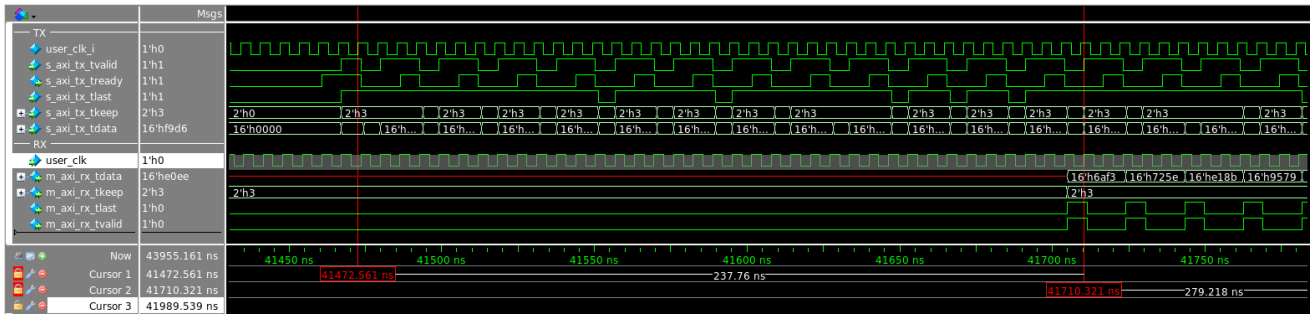


Figure 2-3: **Aurora 8B/10B 2-Byte Latency**

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.

The pipeline delays are designed to maintain the clock speed.

Throughput

Aurora 8B/10B core throughput depends on the number of the transceivers and the targeted line rate. Throughput varies from 0.4 Gb/s to 84.48 Gb/s for a single lane design to a 16 lane design, respectively. The throughput was calculated using 20% 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 8B/10B 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 example design modules such as FRAME_GEN and FRAME_CHECK.

Table 2-1: **7 Series and Zynq-7000 Family Resource Usage for Streaming with 2-Byte Lane Width**

7 Series and Zynq-7000 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	1,189	433	805
8	LUTs	1,258	478	757
	FFs	1,970	656	1,361
16	LUTs	2,229	841	1,345
	FFs	3,534	1,092	2,473

Table 2-2: **7 Series and Zynq-7000 Family Resource Usage for Framing with 2-Byte Lane Width**

7 Series and Zynq-7000 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	1,271	438	885
8	LUTs	1,374	475	867
	FFs	2,145	662	1,507
16	LUTs	2,448	903	1,545
	FFs	3,907	1,153	2,785

Table 2-3: **7 Series and Zynq-7000 Resource Usage for Streaming with 4-Byte Lane Width**

7 Series and Zynq-7000 Families		Streaming		
		Duplex	Simplex	
Lanes	Resource Type	Full Duplex	TX-Only Simplex	RX-Only Simplex

Table 2-3: **7 Series and Zynq-7000 Resource Usage for Streaming with 4-Byte Lane Width**

7 Series and Zynq-7000 Families		Streaming		
1	LUTs	447	182	277
	FFs	651	285	415
2	LUTs	684	251	434
	FFs	964	367	652
4	LUTs	1,091	376	687
	FFs	1,536	530	1,057
8	LUTs	1,877	625	1,169
	FFs	2,678	852	1,865
16	LUTs	3,471	1,124	2,148
	FFs	4,962	1,496	3,481

Table 2-4: **7 Series and Zynq-7000 Family Resource Usage for Framing with 4-Byte Lane Width**

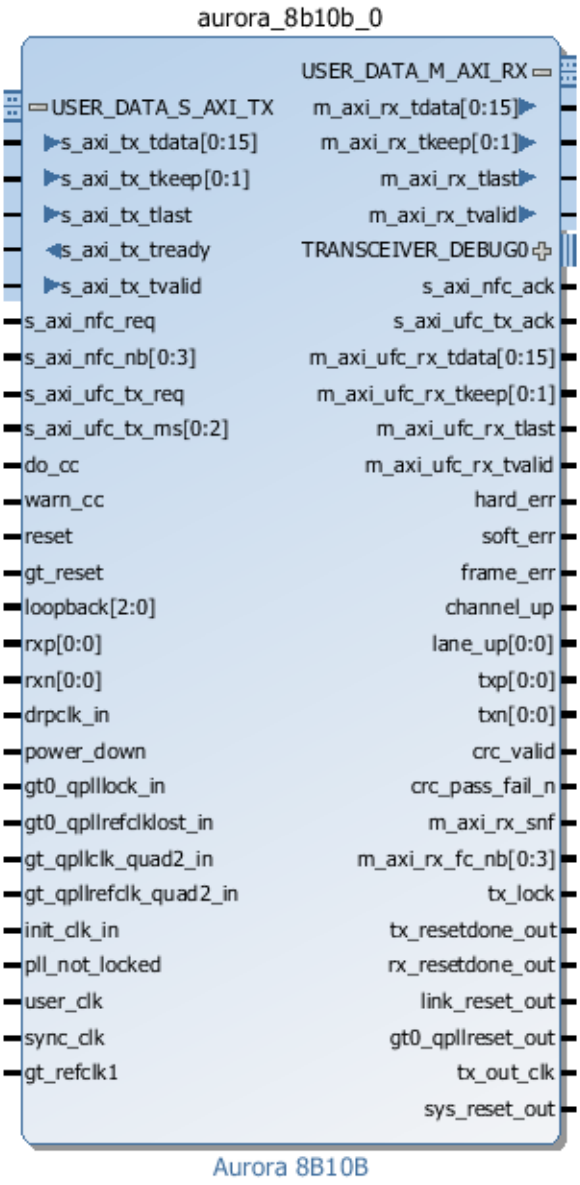
7 Series and Zynq-7000 Families		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full Duplex	TX-Only Simplex	RX-Only Simplex
1	LUTs	490	186	309
	FFs	695	283	461
2	LUTs	750	259	488
	FFs	1,044	368	732
4	LUTs	1,210	398	795
	FFs	1,707	532	1,203
8	LUTs	2,108	680	1,382
	FFs	3,043	905	2,177
16	LUTs	3,859	1,212	2,545
	FFs	5,369	1,605	3,922

Note: UltraScale™ device implementation results are expected to be similar to 7 series devices provided in the preceding tables.

Port Descriptions

The parameters used to generate each Aurora 8B/10B core determine the interfaces available for that specific core (see Figure 2-4). The cores have four to six interfaces:

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



Notes:

1. [*n*:0] bus format is used when the **Little Endian Support** option is selected.
2. [0:*n*] bus format is used when the **Big Endian Support** option is selected.
3. Ports are active-High unless otherwise specified.

Figure 2-4: **Top-Level Interface**

User Interface

The Aurora 8B/10B core can be generated with either a *framing* or *streaming* user data interface. This interface includes all the ports needed for streaming or framed data transfer. The framing user interface complies with the AMBA® AXI4-Stream Protocol Specification [Ref 5] and comprises the signals necessary for transmitting and receiving framed user data. The streaming interface allows data to be sent without frame delimiters, is simpler to operate and uses fewer resources than a framing interface. The data port width depends on the lane width and the number of lanes selected.

Top-Level Architecture

The Aurora 8B/10B core top level (block level) file instantiates the lane logic module, TX and RX AXI4-Stream modules, the global logic module, and the wrapper for the GTX or GTH transceiver. Also instantiated are the clock, reset circuit, frame generator and checker modules.

Figure 2-5 shows the Aurora 8B/10B core top level for a duplex configuration. The top-level file is the starting point for a user design.

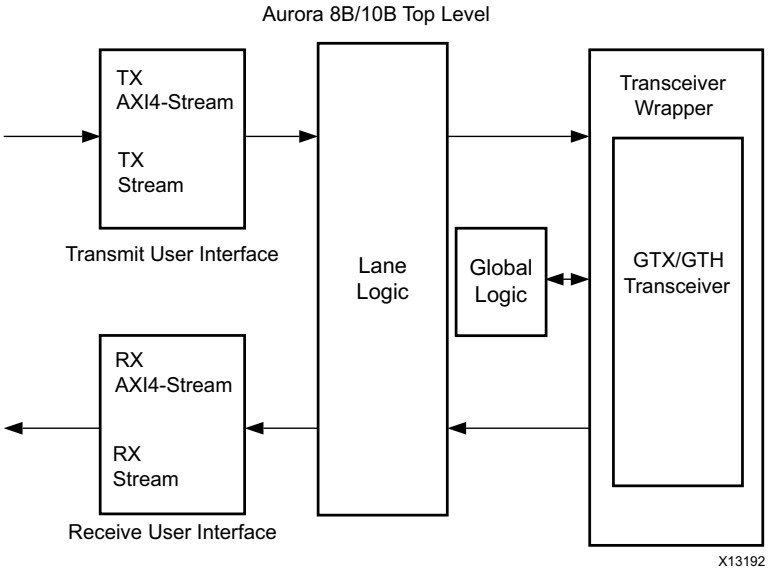


Figure 2-5: **Top-Level Architecture**

This section provides the streaming and framing interface details. User interface logic should be designed to comply with the timing requirement of the respective interface as explained here.

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 2-6 shows the organization of an *n*-byte example of the AXI4-Stream data interfaces of an Aurora 8B/10B core.

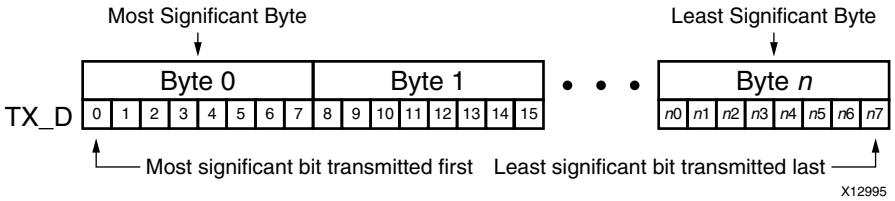


Figure 2-6: **AXI4-Stream Interface Bit Ordering**

User Interface Ports

Table 2-5 and Table 2-6 list duplex and simplex core AXI4-Stream TX and RX data port descriptions.

Table 2-5: **User I/O Ports (TX)**

Name	Direction	Clock Domain	Description
USER_DATA_S_AXI_TX			
s_axi_tx_tdata[0:(8n–1)] or s_axi_tx_tdata[(8n–1):0]	Input	user_clk	Outgoing data. *n* is the number of bytes computed as Number of lanes x Lane Width.
s_axi_tx_tready	Output	user_clk	Asserted when signals from the source are accepted and when outgoing data is ready to send.
s_axi_tx_tlast(1)	Input	user_clk	Signals the end of the frame.
s_axi_tx_tkeep[0:(n–1)] or s_axi_tx_tkeep[(n–1):0](1)	Input	user_clk	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 8B/10B 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	user_clk	Asserted when outgoing AXI4-Stream signals or signals from the source are valid.

Notes:

1. This port is not available if the **Streaming interface** option is chosen.

Table 2-6: **User I/O Ports (RX)**

Name	Direction	Clock Domain	Description
USER_DATA_M_AXI_RX			
m_axi_rx_tdata[0:8(n–1)] or m_axi_rx_tdata[8(n–1):0]	Output	user_clk	Incoming data from channel partner (Ascending bit order).
m_axi_rx_tlast(1)	Output	user_clk	Signals the end of the incoming frame (asserted for a single user clock cycle).
m_axi_rx_tkeep[0:(n–1)] or m_axi_rx_tkeep[(n–1):0](1)	Output	user_clk	Specifies the number of valid bytes in the last data beat.
m_axi_rx_tvalid	Output	user_clk	Asserted when outgoing data and control signals or data and control signals from an Aurora 8B/10B core are valid.

Notes:

1. This port is not available if the **Streaming interface** option is chosen.

Framing Interface

Figure 2-7 shows the framing user interface of the Aurora 8B/10B core, with AXI4-Stream compliant ports for TX and RX data.

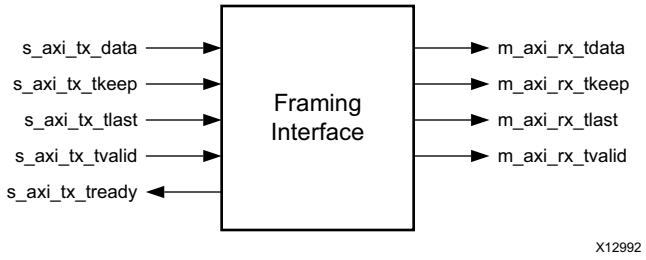


Figure 2-7: **Aurora 8B/10B Core Framing Interface (AXI4-Stream)**

Transmitting Data

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 when `s_axi_tx_tvalid` and `s_axi_tx_tready` signals are asserted.
- Stripe the data across lanes in the Aurora 8B/10B channel.
- Use the `s_axi_tx_tvalid` signal to transmit data. The user application can deassert `s_axi_tx_tvalid` to insert idles on the line (introduce stalls or pause).
- Pause data (that is, insert idles) (`s_axi_tx_tvalid` is deasserted)
- Encapsulate end of the frame when `s_axi_tx_tlast` is asserted.

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 by asserting of the `m_axi_rx_tvalid` signal.

The Aurora 8B/10B core samples data only when both `s_axi_tx_tready` and `s_axi_tx_tvalid` are asserted (High).

AXI4-Stream signals are sampled only if `s_axi_tx_tvalid` is asserted. The user application can deassert `s_axi_tx_tvalid` on any clock cycle 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.

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.

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 Strobe

The AXI4-Stream protocol allows the last word of a frame to be a partial word. 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 received user frame through the TX interface to an Aurora 8B/10B frame. Start of frame is indicated by the addition of a 2-byte SCP code group at the beginning of the frame. End of frame is indicated by the addition of a 2-byte ECP code group at the end of the frame. Idle code groups are inserted whenever data is not available. Code groups are 8B/10B encoded byte pairs and all data are 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 2-7 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 2-7.

Table 2-7: **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 2-8 shows an example of a simple data transfer on an 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.

The user application asserts `s_axi_tx_tvalid` during the first *n* bytes to begin data transfer. 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. When `s_axi_tx_tlast` is asserted, `s_axi_tx_tready` is deasserted in the next clock cycle and the core 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 to allow data transfers to continue.

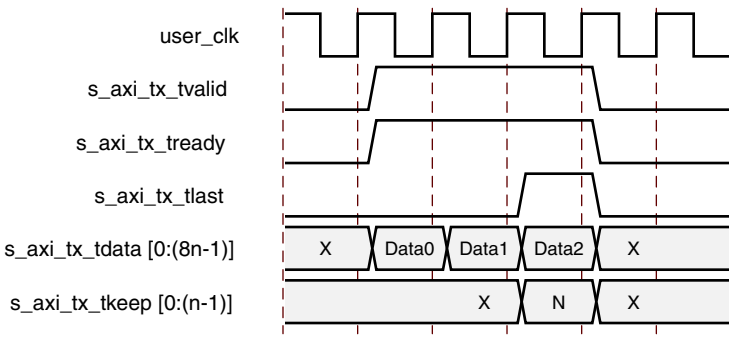


Figure 2-8: **Simple Data Transfer**

Example B: Data Transfer with Pad

Figure 2-9 shows an example of a (3*n*–1)-byte data transfer that requires the use of a pad. The Aurora 8B/10B core appends a pad character for a frame with an odd number of bytes as per the protocol requirement. 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.

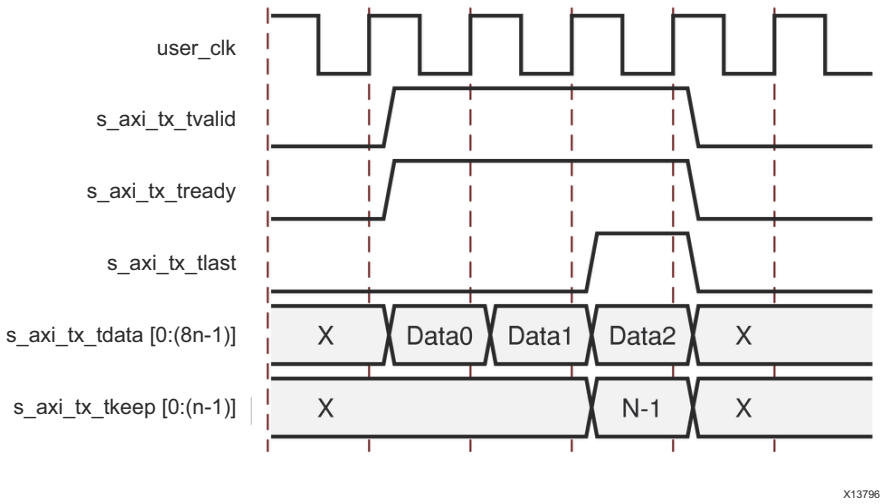


Figure 2-9: **Data Transfer with Pad**

Example C: Data Transfer with Pause

Figure 2-10 shows how a user interface can pause data transmission during a frame transfer. In this example, the user application pauses the data flow after the first *n* bytes by deasserting `s_axi_tx_tvalid`, and transmitting idles instead. The pause continues until `s_axi_tx_tvalid` is deasserted.

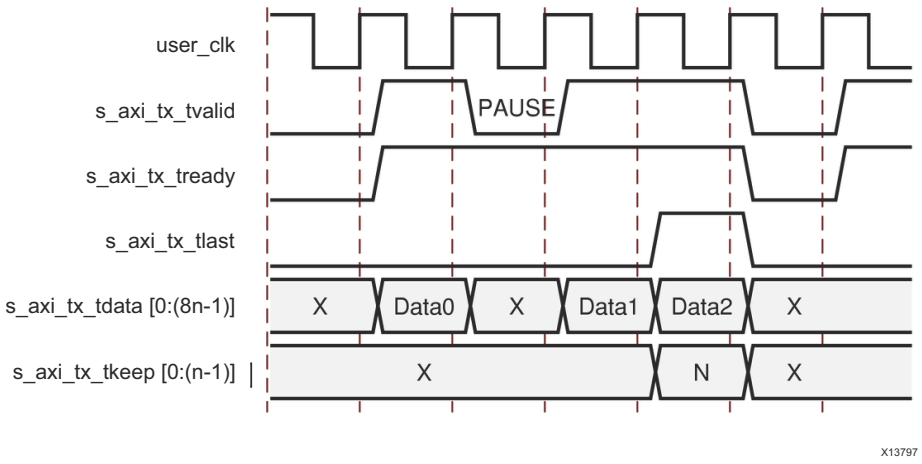


Figure 2-10: **Data Transfer with Pause**

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 2-11 shows how the Aurora 8B/10B core pauses data transmission during the clock compensation sequence.

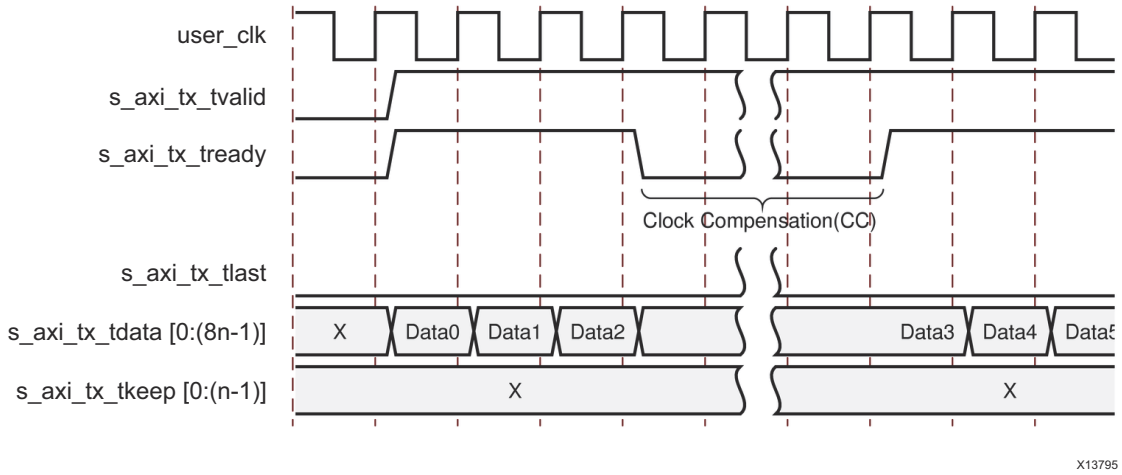


Figure 2-11: **Data Transfer Paused by Clock Compensation**

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 or three clock periods.

Receiving Data

The RX submodules have 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.

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 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.

Example A: Data Reception with Pause

Figure 2-12 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. The core deasserts `m_axi_rx_tvalid` on the clock cycle following the first data beat to indicate 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.

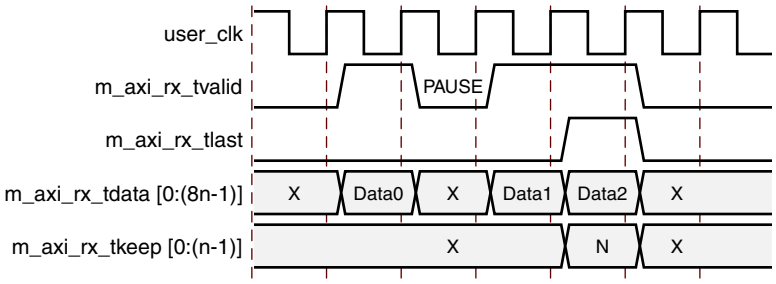


Figure 2-12: **Data Reception with Pause**

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.

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 the 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 2-1 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.

$$E = \frac{100n}{n + 4 + 0.5 + IDLEs + \frac{12n}{9988}} \qquad \textit{Equation 2-1}$$

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 2-8 is an example calculated from Equation 2-1. 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 2-8: **Efficiency Example**

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

Table 2-9 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 2-9: **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 2-10 shows the overhead that occurs with each value of s_axi_tx_tkeep.

Table 2-10: **Value of s_axi_tx_tkeep 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

Note: When the **Little Endian** option is selected in the Vivado IDE, s_axi_tx_tkeep bit ordering changes from MSB to LSB.

Streaming Interface

Figure 2-13 shows an example of an Aurora 8B/10B core configured with a streaming user interface.

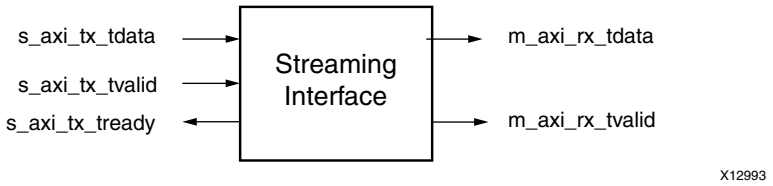


Figure 2-13: **Aurora 8B/10B Core Streaming User Interface**

Streaming Interface Ports

Table 2-5, page 16 and Table 2-6, page 17 list duplex and simplex core AXI4-Stream TX and RX data port descriptions.

Transmitting and Receiving Data

The streaming interface allows the Aurora 8B/10B channel to be used as a pipe. After initialization, the channel is always available for writing, except when the do_cc signal is asserted to send clock compensation sequences. Core data transfer is compliant with the AXI4-Stream protocol.

When s_axi_tx_tvalid is deasserted, gaps are created between words and the gaps are preserved, except when clock compensation sequences are being transmitted. For details on the do_cc signal, see Clock Compensation, page 50.

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.

Example A: TX Streaming Data Transfer

Figure 2-14 shows a typical example of streaming data. 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, again, the `s_axi_tx_tready` signal is asserted. Then the user logic deasserts `s_axi_tx_tvalid` on the next clock cycle, and no data is transferred until both ready signals are asserted.

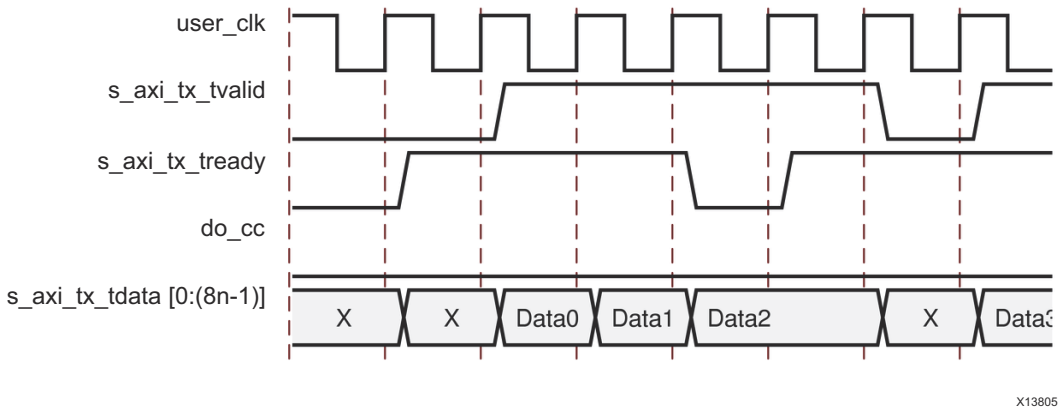


Figure 2-14: **Typical Streaming Data Transfer**

Example B: RX Streaming Data Transfer

Figure 2-15 shows the receiving end of the data transfer that is shown in Figure 2-14.

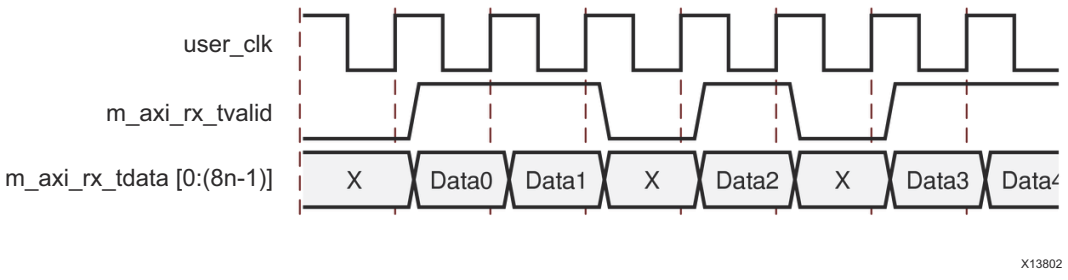


Figure 2-15: **Typical Data Reception**

Flow Control

This section explains how to use Aurora 8B/10B flow control. Two optional flow control interfaces are available on cores using a framing interface. Native flow control (NFC) regulates the data transmission rate at the receiving end of a full-duplex channel. User flow control (UFC) accommodates high-priority messages for control operations.

User Flow Control Interface

The UFC interface is created when the core is generated with UFC enabled (Figure 2-16). UFC request and acknowledge ports on the TX side start the UFC message and a 3-bit port specifies the length of the message. With the UFC request acknowledged, the UFC message can be supplied to the data port.

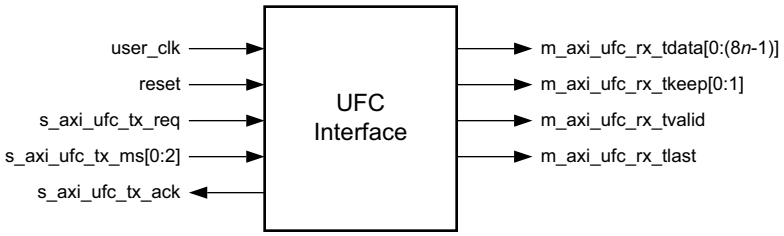


Figure 2-16: **Aurora 8B/10B Core UFC Interface**

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. Simplex modules retain only the interface needed to send data in the supported direction.

Table 2-11 describes the ports for the UFC interface.

Table 2-11: **UFC I/O Ports**

Name	Direction	Clock Domain	Description
s_axi_ufc_tx_req	Input	user_clk	Asserted to request a UFC message be sent to the channel partner. 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] or s_axi_ufc_tx_ms[2:0]	Input	user_clk	Specifies the size of the UFC message that is sent. The SIZE encoding is a value between 0 and 7. See Table 2-12.

Table 2-11: **UFC I/O Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
s_axi_ufc_tx_ack	Output	user_clk	Asserted when an Aurora 8B/10B core is ready to read the contents of the UFC message. 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)] or m_axi_ufc_rx_tdata[(8n–1):0]	Output	user_clk	Incoming UFC message data from the channel partner (n = 16 bytes maximum).
m_axi_ufc_rx_tvalid	Output	user_clk	Asserted when the values on the m_axi_ufc_rx ports are valid.
m_axi_ufc_rx_tlast	Output	user_clk	Signals the end of the incoming UFC message.
m_axi_ufc_rx_tkeep[0:(n–1)] or m_axi_ufc_rx_tkeep[(n–1):0]	Output	user_clk	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).

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 2-12 shows the legal SIZE code values for UFC messages.

Table 2-12: **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. The data for the UFC message must be placed on the s_axi_tx_tdata port, 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.

Note: A UFC request should be given only after completion of the current UFC request and back-to-back UFC requests might not honored by IP.

Figure 2-17 shows a useful circuit for switching `TX_D` from sending regular data to UFC data.

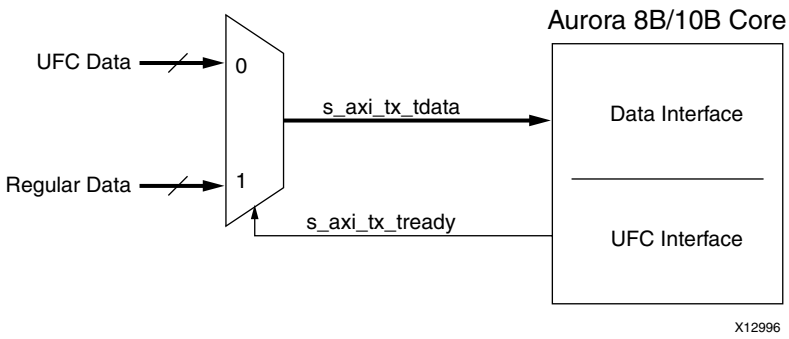


Figure 2-17: **Data Switching Circuit**

Table 2-13 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 until completion of the current UFC message.

Table 2-13: **Number of Data Beats Required to Transmit UFC Messages**

UFC Message	S_AXI_UFC_TX_MS Value
2 Bytes	0
4 Bytes	1
6 Bytes	2
8 Bytes	3
10 Bytes	4
12 Bytes	5
14 Bytes	6
16 Bytes	7

AXI4 Interface Width	Number of Data Beats
2 Bytes	1
	2
	3
	4
	5
	6
	7
	8

AXI4 Interface Width	Number of Data Beats
10 Bytes	1
	2

Table 2-13: **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	4 Bytes	1	12 Bytes	1
4 Bytes	1				
6 Bytes	2		2		
8 Bytes	3				
10 Bytes	4		3		
12 Bytes	5				
14 Bytes	6		4		2
16 Bytes	7				
2 Bytes	0	6 Bytes	1	14 Bytes	1
4 Bytes	1				
6 Bytes	2				
8 Bytes	3		2		
10 Bytes	4				
12 Bytes	5				
14 Bytes	6		3		
16 Bytes	7				2
2 Bytes	0	8 Bytes	1	16 Bytes or more	1
4 Bytes	1				
6 Bytes	2				
8 Bytes	3				
10 Bytes	4		2		
12 Bytes	5				
14 Bytes	6				
16 Bytes	7				

Example A: Transmitting a Single-Cycle UFC Message

The procedure for transmitting a single cycle UFC message is shown in Figure 2-18. 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.

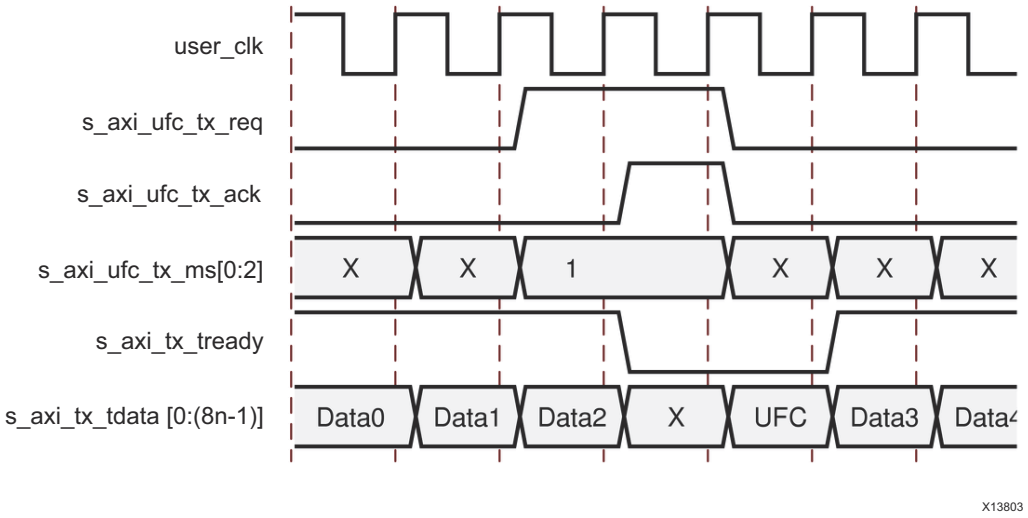


Figure 2-18: **Transmitting a Single-Cycle UFC Message**

Example B: Transmitting a Multicycle UFC Message

The procedure for transmitting a two-cycle UFC message is shown in Figure 2-19. 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.

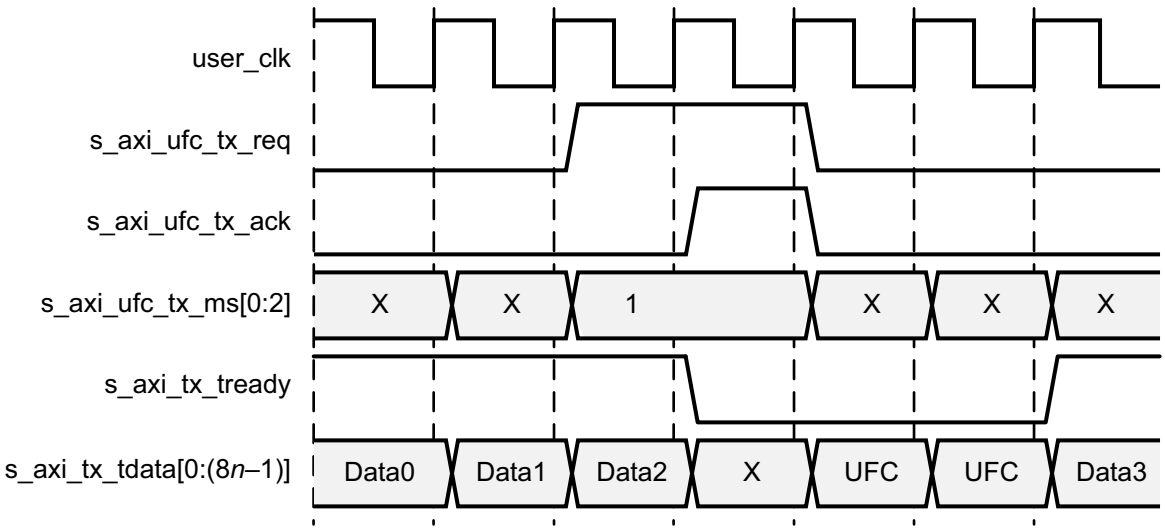


Figure 2-19: **Transmitting a Multicycle UFC Message**

Receiving User Flow Control Messages

When the Aurora 8B/10B core receives a UFC message, it passes the data 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.

Example C: Receiving a Single-Cycle UFC Message

Figure 2-20 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.

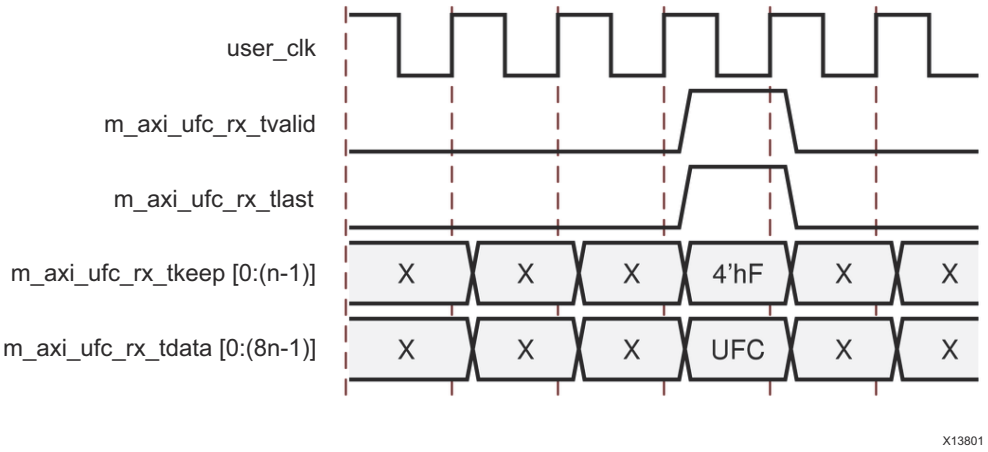


Figure 2-20: **Receiving a Single-Cycle UFC Message**

Example D: Receiving a Multicycle UFC Message

Figure 2-21 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.

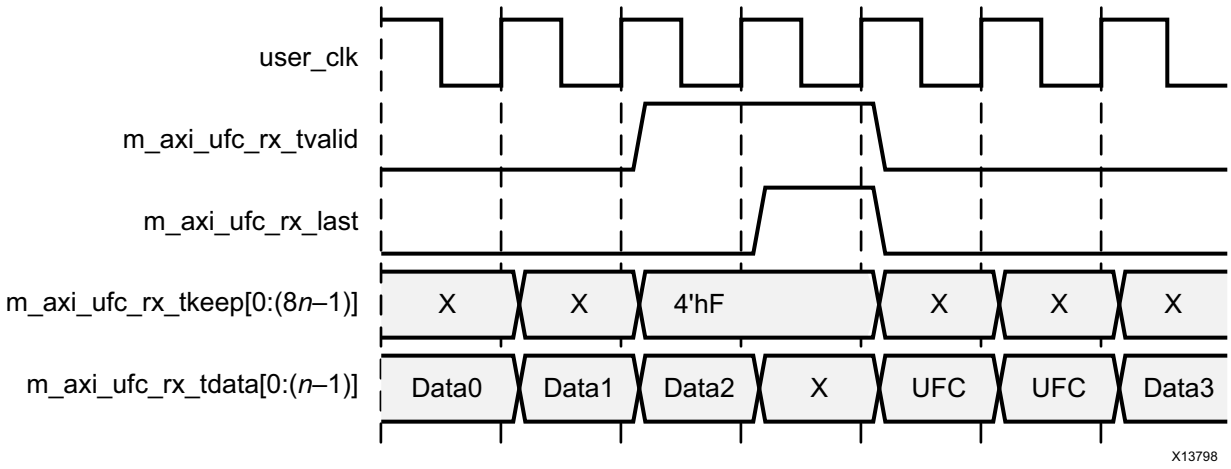


Figure 2-21: **Receiving a Multicycle UFC Message**

Native Flow Control

The Aurora 8B/10B protocol includes the native flow control (NFC) interface (Figure 2-22) which allows receivers to control the rate at which data is received by specifying the number of idle data beats that must be placed into the data stream. The data flow can even be turned off completely by requesting the transmitter to temporarily send only idles (XOFF). NFC is typically used to prevent FIFO overflow conditions. For a detailed explanation of NFC operation and codes, see the *Aurora 8B/10B Protocol Specification* (SP002) [Ref 4].

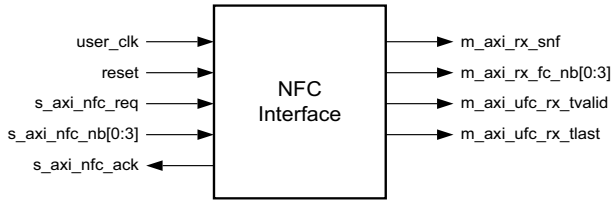


Figure 2-22: **Aurora 8B/10B Core NFC Interface**

Native Flow Control Interface

The NFC interface is created when the core is generated with the NFC option enabled. 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 2-14 lists the ports for the NFC interface available only in full-duplex Aurora 8B/10B cores.

Table 2-14: **NFC I/O Ports**

Name	Direction	Clock Domain	Description
s_axi_nfc_ack	Output	user_clk	Asserted when the core accepts an NFC request.
s_axi_nfc_nb[0:3] or s_axi_nfc_nb[3:0]	Input	user_clk	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	user_clk	Asserted to request an NFC message be sent to the channel partner. Must be held until `s_axi_nfc_ack` is asserted.
m_axi_rx_snf	Output	user_clk	Indicates an NFC message is received from the partner.
m_axi_rx_fc_nb[0:3] or m_axi_rx_fc_nb[3:0]	Output	user_clk	Indicates the PAUSE value of the received NFC message. This port should be sampled with `m_axi_rx_snf`.

Table 2-15 shows the codes for native flow control (NFC). These values are driven on bits [0:3] for Big Endian format and [3:0] for Little Endian format.

Table 2-15: **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 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. 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 2-23 shows an example of the transmit timing when the user application sends an NFC message to a channel partner.

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.

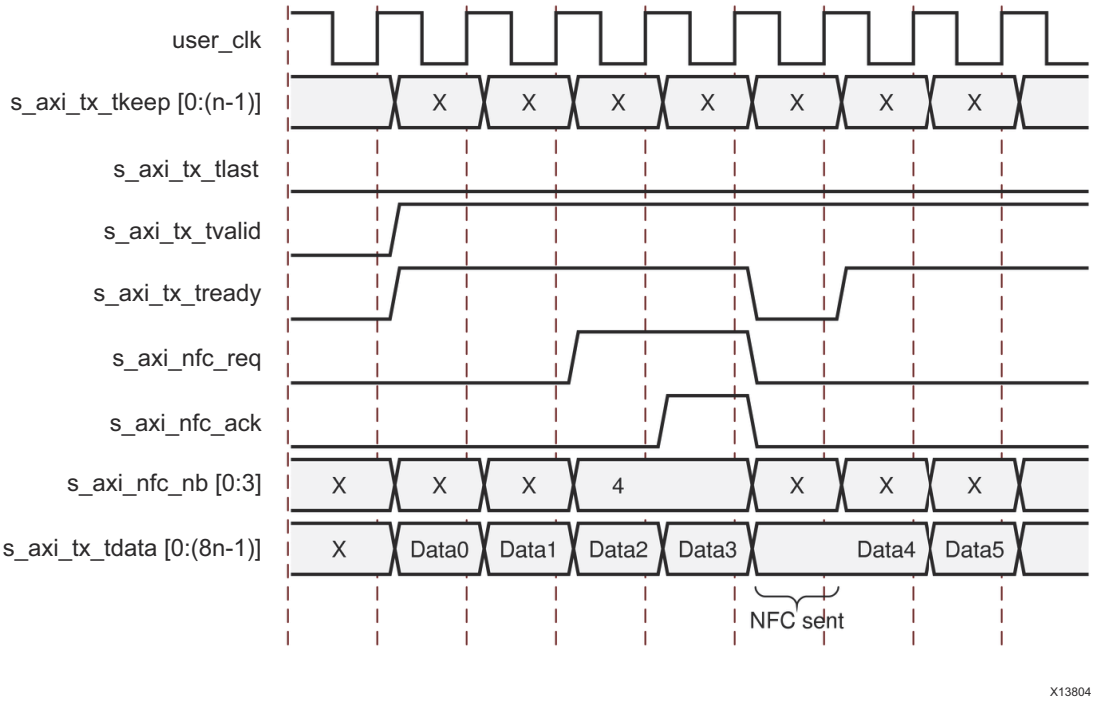


Figure 2-23: **Transmitting an NFC Message**

Example B: Receiving a Message with NFC Idles Inserted

Figure 2-24 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, where NFC idles are inserted immediately. 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.

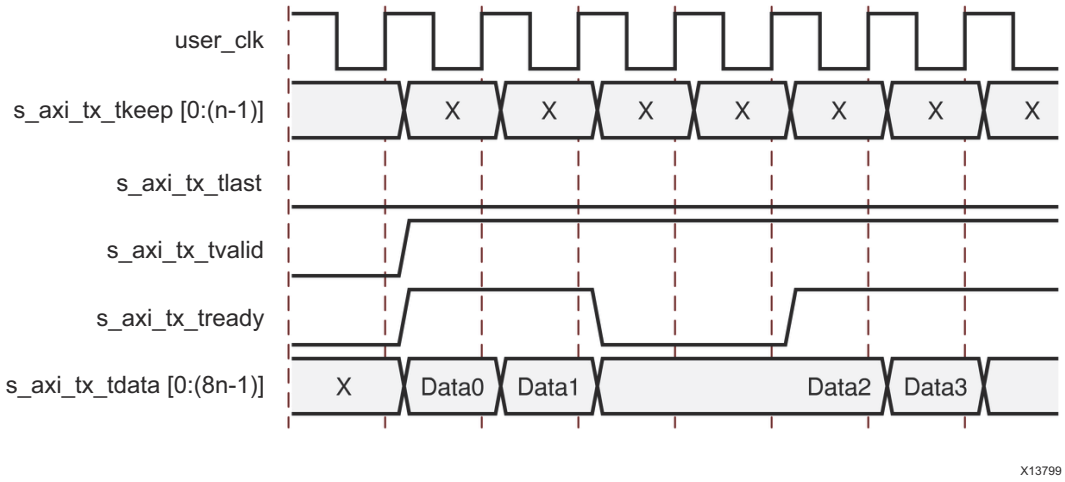


Figure 2-24: **Receiving a Message with NFC Idles Inserted**

Status, Control, and the Transceiver Interface

The status and control ports of the Aurora 8B/10B core allow applications to monitor the channel and use built-in features of the transceivers. This section provides diagrams and port descriptions for the status and control interface, the transceiver serial I/O interface, and the sideband initialization ports used exclusively for simplex modules.

Status and Control Ports

Table 2-16 describes the function of each of the status and control ports.

Table 2-16: **Status and Control Ports**

Name	Direction	Clock Domain	Description
channel_up/ tx_channel_up/ rx_channel_up	Output	user_clk	Asserted when Aurora 8B/10B channel initialization is complete and channel is ready for data transfer. `tx_channel_up` and `rx_channel_up` are only applicable to their respective simplex cores.
lane_up[0:*m*–1]/ tx_lane_up[0:*m*–1]/ rx_lane_up[0:*m*–1](1)	Output	user_clk	Asserted for each lane upon successful lane initialization, with each bit representing one lane. `tx_lane_up[0:(m–1)]` and `rx_lane_up[0:(m–1)]` are only applicable to their respective simplex cores.
frame_err	Output	user_clk	Channel frame/protocol error detected. This port is asserted for a single clock. Not available on simplex TX core.
hard_err/ tx_hard_err/ rx_hard_err	Output	user_clk	Hard error detected (asserted until Aurora 8B/10B core resets). `tx_hard_err` and `rx_hard_err` are only applicable to their respective simplex cores.
soft_err	Output	user_clk	Soft error detected in the incoming serial stream. Not available on simplex TX core.
reset/ tx_system_reset/ rx_system_reset	Input	user_clk	Resets the Aurora 8B/10B core (active-High). This signal must be asserted for at least six `user_clk` cycles. `tx_system_reset` and `rx_system_reset` are only applicable to their respective simplex cores.
gt_reset	Input	init_clk_ in	The reset signal for the transceivers is connected to the top level through a debouncer. The gt_reset port should be asserted 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.
link_reset_out	Output	init_clk	Driven High if hot-plug count expires.

Table 2-16: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
init_clk_in	Input	NA	The `init_clk_in` port is required because `user_clk` stops when gt_reset is asserted. It is recommended that the frequency chosen for `init_clk_in` be lower than the GT Reference Clock input frequency. For UltraScale architecture designs: The `init_clk_in` frequency should be equal to the `TXUSERCLK` frequency and the value should not exceed 200 MHz. The `TXUSERCLK` frequency depends on the line rate and the internal datapath width. Refer to the *UltraScale FPGAs GTH Transceivers User Guide* (UG576) [Ref 1] for more details. This `init_clk_in` is connected to `DRPCLK` of the DRP ports of GTHE3_CHANNEL as well.
tx_aligned(2)	Input	user_clk	Asserted when RX channel partner has completed lane initialization for all lanes. Typically connected to `rx_aligned`.
tx_bonded(2)	Input	user_clk	Asserted when RX channel partner has completed channel bonding. Not needed for single-lane channels. Typically connected to `rx_bonded`.
tx_verify(2)	Input	user_clk	Asserted when RX channel partner has completed verification. Typically connected to `rx_verify`.
tx_reset(2)	Input	user_clk	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`.
rx_aligned(3)	Output	user_clk	Asserted when RX module has completed lane initialization. Typically connected to `tx_aligned`.
rx_bonded(3)	Output	user_clk	Asserted when RX module has completed channel bonding. Not used for single-lane channels. Typically connected to `tx_bonded`.
rx_verify(3)	Output	user_clk	Asserted when RX module has completed verification. Typically connected to `tx_verify`.
rx_reset(3)	Output	user_clk	Asserted when the RX module needs the TX module to restart initialization. Typically connected to `tx_reset`.

Notes:

1. *m* is the number of transceivers. See Error Status Signals for more details.
2. Only available in TX-only simplex dataflow mode and sideband as back channel core configuration.
3. Only available in RX-only simplex dataflow mode and sideband as back channel core configuration.

Full-Duplex Cores

Full-Duplex Status and Control Ports

Full-duplex cores provide a TX and an RX Aurora 8B/10B channel connection. Figure 2-25 shows the status and control interface for a full-duplex Aurora 8B/10B core.

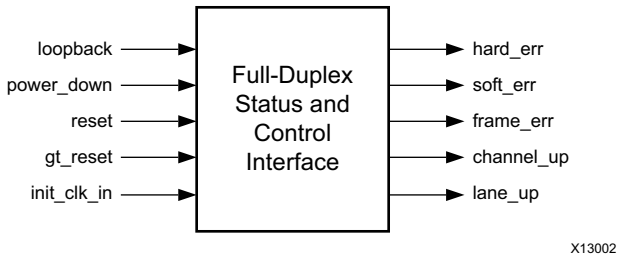


Figure 2-25: **Status and Control Interface for Full-Duplex Cores**

Error Status Signals

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 and to assert `soft_err` on every cycle. The TX simplex cores do not include a `soft_err` port. All transmit data is assumed to be correct at transmission unless there is an equipment issue.

The core also monitors each transceiver for hardware errors such as buffer overflow/underflow and loss of lock and asserts the `hard_err` signal. RX-side hard errors in simplex cores are reported using the `rx_hard_err` signal. Catastrophic hardware errors can also manifest themselves as a burst of soft errors. The core uses the leaky bucket algorithm described in the *Aurora 8B/10B Protocol Specification* (SP002) [Ref 4] to detect large numbers of soft errors occurring in a short period of time, and asserts the `hard_err` or `rx_hard_err` signal.

Whenever a hard error is detected, the core automatically resets itself and attempts to re-initialize. This allows the channel to re-initialize and 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.

Aurora 8B/10B cores with the AXI4-Stream data interface can also detect errors in Aurora 8B/10B frames and assert the `frame_err` signal. Frame errors can be frames with no data, consecutive Start of Frame symbols, and consecutive End of Frame symbols. This signal is not available with simplex TX cores. When available, this signal is usually asserted close to a `soft_err` assertion, with soft errors being the main cause of frame errors.

Table 2-17 summarizes the error conditions Aurora 8B/10B cores can detect and the error signals used to alert the user application.

Table 2-17: **Error Signals in 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
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

Full-Duplex Initialization

Full-duplex cores initialize automatically after power-up, reset, or hard error and 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 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 2-26 shows the status and control interface for a simplex TX core.

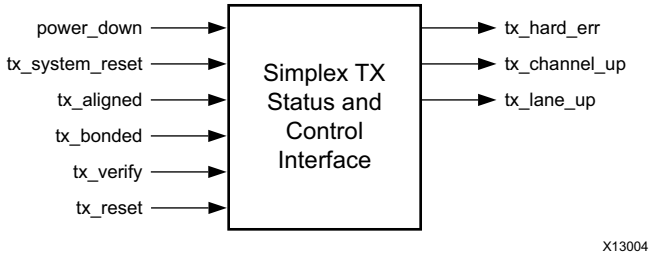


Figure 2-26: **Status and Control Interface for Simplex TX Core**

Simplex RX Status and Control Ports

Simplex RX cores allow user applications to receive data from a simplex TX core. Figure 2-27 shows the status and control interface for a simplex RX core.

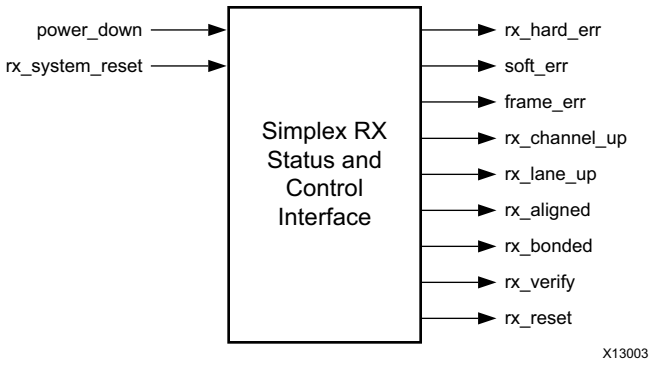


Figure 2-27: **Status and Control Interface for Simplex RX Core**

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: `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

A back channel is the safest way to initialize and maintain a simplex channel in the absence of a channel between RX and TX. The back channel need only deliver messages to the TX side to indicate which of the sideband initialization signals is asserted when the signals change.

The 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

If a back channel is not possible, 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.

Note: These signals are not updated on the actual status of the channel, but after the timer expires.

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 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, typically, the RX side detects the failure while the condition must be handled by the TX side. The solution 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.

Transceiver Interface

This interface includes the serial I/O ports of the transceivers, and the control and status ports of the Aurora 8B/10B core. Table 2-18 describes the transceiver ports.

Table 2-18: **Transceiver Ports**

Name	Direction	Clock Domain	Description
rxp[0:*m*–1](1)	Input	RX Serial Clock	Positive differential serial data input pin.
rxn[0:*m*–1](1)	Input	RX Serial Clock	Negative differential serial data input pin.
txp[0:*m*–1](1)	Output	TX Serial Clock	Positive differential serial data output pin.
txn[0:*m*–1](1)	Output	TX Serial Clock	Negative differential serial data output pin.
power_down	Input	user_clk	Drives the power-down input of the transceiver. For more information, see the applicable transceiver user guide.
loopback[2:0]	Input	user_clk	The loopback[2:0] port selects between the normal operation and differential loopback modes of the transceiver. See the *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476) [Ref 3].
tx_resetdone_out	Output	user_clk	The `TXRESETDONE` signal of the transceiver.
rx_resetdone_out	Output	user_clk	The `RXRESETDONE` signal of the transceiver.
tx_lock	Output	user_clk	Indicates incoming serial transceiver refclk is locked by the transceiver PLL. See the applicable transceiver guide for more information.
Transceiver DRP Ports			
drpaddr_in/ gt<=: lane :>_drpaddr	Input	drpclk_in	DRP address bus.
drpclk_in/ gt<=: lane :>_drpclk	Input	drpclk_in	DRP interface clock.

Table 2-18: **Transceiver Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
drpdi_in/ gt<=: lane :>_drpdi	Input	drpclk_in	Data bus for writing configuration data from the FPGA logic resources to the transceiver.
drpdo_out/ gt<=: lane :>_drpdo	Output	drpclk_in	Data bus for reading configuration data from the transceiver to the FPGA logic resources.
drpen_in/ gt<=: lane :>_drpen	Input	drpclk_in	DRP enable signal.
drprdy_out/ gt<=: lane :>_drprdy	Output	drpclk_in	Indicates operation is complete for write operations and data is valid for read operations.
drpwe_in/ gt<=: lane :>_drpwe	Input	drpclk_in	DRP write enable.
Transceiver Debug Ports			
gt<lane>_txpostcursor_in/ gt_txpostcursor(4)(6)	Input	async	Transmitter post-cursor TX pre-emphasis control.
gt<lane>_txprecursor_in/ gt_txprecursor(4)(6)	Input	async	Transmitter pre-cursor TX pre-emphasis control.
gt<lane>_txchardispmode_in(4)(6)(10)	Input	user_clk	Set High to work with `txchardispval` to force running disparity negative or positive when encoding TXDATA. Set Low to use normal running disparity.
gt<lane>_txchardispval_in(4)(6)(10)	Input	user_clk	Work with `txchardispmode` to provide running disparity control.
gt<lane>_txdiffctrl_in/ gt_txdiffctrl(4)(6)	Input	async	Driver Swing Control.
gt<lane>_txmaincursor_in(4)(6)(10)	Input	async	Allows the main cursor coefficients to be directly set if the TX_MAINCURSOR_SEL attribute is set to 1'b1.
gt<lane>_txpolarity_in/ gt_txpolarity(4)(6)	Input	user_clk	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.
gt<lane>_tx_buf_err_out(4)(6)(10)	Output	user_clk	TX buffer status. `txbufstatus[1]` is connected to this port.
gt<lane>_rxlpmhfhold_in(4)(7)(10)	Input	user_clk	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.
gt<lane>_rxlpmlfhold_in(4)(7)(10)	Input	user_clk	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.
gt<lane>_rxlpmen_in/ gt_rxlpmen(4)(8)	Input	user_clk	RX datapath • 0: DFE • 1: LPM

Table 2-18: **Transceiver Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<lane>_rxcdrovrden_in/ gt_rxcdrovrden[4][8]	Input	async	Reserved.
gt<lane>_rxcdrhold_in/ gt_rxcdrhold[4][9]	Input	async	Hold the CDR control loop frozen.
gt<lane>_rxdfelpmreset_in/ gt_rxdfelpmreset[4][8]	Input	async	This port is driven High and then deasserted to start the DFE reset process.
gt<lane>_rxmonitorout_out[4][8][10]	Output	async	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]
gt<lane>_rxmonitorsel_in[4][8][10]	Input	async	Select signal for `rxmonitorout[6:0]` • `2'b00`: Reserved • `2'b01`: Select AGC loop • `2'b10`: Select UT loop • `2'b11`: Select VP loop
gt<lane>_eyescanreset_in/ gt_eyescanreset[4][9]	Input	async	This port is driven High and then deasserted to start the EYESCAN reset process.
gt<lane>_eyescandataerror_out/ gt_eyescandataerror[4][9]	Output	async	Asserts High for one `rec_clk` cycle when an (unmasked) error occurs while in the COUNT or ARMED state.
gt<lane>_eyescantrigger_in/ gt_eyescantrigger[4][9]	Input	user_clk	Causes a trigger event.
gt<lane>_rxbyteisaligned_out[4][9][10]	Output	user_clk	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
gt<lane>_rxcommadet_out/ gt_rxcommadet[4][9]	Output	user_clk	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
gt<lane>_rx_disp_err_out[4][9][10]	Output	user_clk	Indicates the corresponding byte shown on `rxdata` has a disparity error. The `rxdisperr` pin of the transceiver is connected to this port.

Table 2-18: **Transceiver Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<lane>_rx_not_in_table_out[4][9][10]	Output	user_clk	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.
gt<lane>_rx_realign_out[4][9][10]	Output	user_clk	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.
gt<lane>_rx_buf_err_out[4][9][10]	Output	user_clk	RX buffer status. `rxbufstatus[2]` is connected to this port.
gt0_pll0lock_out, gt0_pll1lock_out[4]	Output	async	PLL0LOCK and PLL1LOCK of the 7 series FPGA GTP transceiver COMMON block. Available shared logic is in core.
gt1_pll0lock_out, gt1_pll1lock_out[4]	Output	async	PLL0LOCK and PLL1LOCK of the 7 series FPGA GTP transceiver COMMON block. Available shared logic is in the core and two quads are selected during core configuration.
gt<lane>_cplllock_out/ gt_cplllock[4]	Output	async	This active-High PLL frequency lock signal indicates that the PLL frequency is within predetermined tolerance. The transceiver and its clock outputs are not reliable until this condition is met. Not available with 7 series FPGAs GTP transceivers.
gt<lane>_txprbsforceerr_in/ gt_prbsforceerr[4][6]	Input	async	When this port is driven High, errors are forced in the PRBS transmitter. While this port is asserted, the output data pattern contains errors. When `txprbssel` is set to 000, this port does not affect `txdata`.
gt<lane>_txprbssel_in/ gt_prbssel[4][6]	Input	async	Transmitter PRBS generator test pattern control.
gt<lane>_txpcsreset_in/ gt_txpcsreset[4][6]	Input	async	This port is used to reset the TX PCS. It is driven High and then deasserted to start the PCS reset process. In sequential mode, activating this port only resets the TX PCS.
gt<lane>_txpmareset_in/ gt_txpmareset[4][6]	Input	async	This port is used to reset the TX PMA. It is driven High and then deasserted to start the TX PMA reset process. In sequential mode, activating this port resets both the TX PMA and the TX PCS.

Table 2-18: **Transceiver Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<lane>_txresetdone_out/ gt_txresetdone[4][6]	Output	async	This active-High signal indicates the GTX or GTH transceiver TX has finished reset and is ready for use. This port is driven Low when `gttxreset` goes High and is not driven High until the GTX/GTH transceiver TX detects `txuserrdy` High.
gt<lane>_txbufstatus_out/ gt_txbufstatus[4][6]	Output	user_clk	TX buffer status.
gt<lane>_rxresetdone_out/ gt_rxresetdone[4][9]	Output	user_clk	When asserted, this active-High signal indicates the GTX or GTH transceiver RX has finished reset and is ready for use. In sequential mode, this port is driven Low when `gtrxreset` is driven High. This signal is not driven High until `rxuserrdy` goes High. In single mode, this port is driven Low when any of the RX resets are asserted. This signal is not asserted until all RX resets are deasserted and `rxuserrdy` is asserted.
gt<lane>_rxbufstatus_out/ gt_rxbufstatus[4][9]	Output	user_clk	RX buffer status.
gt<lane>_rxlpmhfovrden_in[4][7]	Input	user_clk	When set to `1'b1`, the high-frequency boost is controlled by the RXLPM_HF_CFG attribute. When set to `1'b0`, the high-frequency boost is controlled by the `rxlpmhfhold` signal.
gt<lane>_rxlpmreset_in[4][7]	Input	async	Resets the LPM circuitry.
gt<lane>_rxprbserr_out/ gt_rxprbserr[4][9]	Output	user_clk	This non-sticky status output indicates that PRBS errors have occurred.
gt<lane>_rxprbssel_in/ gt_rxprbssel[4][9]	Input	user_clk	Receiver PRBS checker test pattern control.
gt<lane>_rxpcsreset_in/ gt_rxpcsreset[4][9]	Input	user_clk	This port is driven High and then deasserted to start the PCS reset process.
gt<lane>_rxpmareset_in/ gt_rxpmareset[4][9]	Input	async	This port is driven High and then deasserted to start RX PMA reset process.
gt<lane>_rxpmaresetdone_out/ gt_rxpmaresetdone[4]	Output	async	This active-High signal indicates GTH/GTP RX PMA reset is complete. This port is driven Low when `gtrxreset` or `rxpmareset` is asserted. Available for Duplex and RX-Only Simplex configuration and applicable for 7 series GTP and GTH transceivers only.
gt<lane>_dmonitorout_out/ gt_dmonitorout[4][9]	Output	async	Digital Monitor Output Bus

Table 2-18: **Transceiver Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<lane>_rxbufreset_in/ gt_rxbufreset(4)(9)	Input	async	This port is driven High and then deasserted to start the RX elastic buffer reset process. In either single mode or sequential mode, activating `rxbufreset` resets the RX elastic buffer only.

Notes:

1. *m* is the number of transceivers.
2. The transceiver debug ports are enabled if the **Additional transceiver control and status ports** check-box option is selected in the Vivado IDE.
3. <lane> takes values from 0 to AURORA_LANES.
4. For designs using UltraScale devices, the prefixes of the optional transceiver debug ports for single-lane cores are changed from gt<lane> to gt, and the postfixes _in and _out are removed. For multi-lane cores, the prefixes of the optional transceiver debug ports gt(n) are aggregated into a single port.
5. See the relevant transceiver user guide for more information on transceiver debug ports.
6. Available with duplex and TX-only simplex configurations.
7. Available with duplex and RX-only simplex configurations and applicable to 7 series FPGAs GTP transceivers only.
8. Available with duplex and RX-only simplex configurations and applicable to 7 series FPGAs GTX and GTH transceivers only.
9. Available with duplex and RX-only simplex configurations.
10. Not available with UltraScale devices.

Clock Interface

The clock interface has ports for the transceiver reference clock, and parallel clocks that the Aurora 8B/10B core shares with application logic.

Table 2-19 describes the Aurora 8B/10B core clock ports.

Table 2-19: **Clock Ports for 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. 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. `user_clk` and `sync_clk` are the outputs of a PLL or BUFG driven by `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.
sync_clk	Input	Parallel clock used by the internal synchronization logic of the transceivers. `sync_clk` goes as the `txusrclk` input to the transceiver.
gt_refclk	Input	The `gt_refclk (clkp/clkn)` port is a dedicated external transceiver reference clock fed through IBUFDS.

Table 2-19: **Clock Ports for Aurora 8B/10B Core *(Cont'd)***

Clock Ports	Direction	Description
gt0_pll0outclk_in/ gt1_pll0outclk_in	Input	This port should be connected to the PLL0OUTCLK/PLL1OUTCLK clock output generated by GTPE2_COMMON. This port is internally connected to the PLL0CLK/PLL1CLK port on the GTPE2_CHANNEL primitive.
gt0_pll0outrefclk_in/ gt0_pll1outrefclk_in	Input	This port should be connected to the PLL0OUTREFCLK/PLL1OUTREFCLK clock output generated by GTPE2_COMMON. This port is internally connected to the PLL0REFCLK/PLL1REFCLK port on the GTPE2_CHANNEL primitive.
quad1_common_lock_in	Input	GTPE2_COMMON PLL lock input port.

Table 2-20 provides details about the port changes due to selection of the Shared Logic option.

Table 2-20: **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 in core** is selected
gt_refclk2_p gt_refclk2_n	Input	Transceiver reference clock 2	Enabled when **Shared Logic in core** is selected
gt_refclk1_out	Output	Output of IBUFDS_GTE2 for transceiver reference clock 1	Enabled when **Shared Logic in example design** is selected
gt_refclk2_out	Output	Output of IBUFDS_GTE2 for transceiver reference clock 2	Enabled when **Shared Logic in example design** is selected
user_clk_out	Output	Parallel clock shared by Aurora 8B/10B 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 example design** is selected
gt0_pll0refclklost_out gt1_pll0refclklost_out(1)	Output	Indicates refclklost port of the GTPE2_COMMON	Enabled when **Shared Logic in core** is selected.

Table 2-20: **Port Changes Due to Shared Logic Option *(Cont'd)***

Name	Direction	Description	Remarks
quad1_common_lock_out quad2_common_lock_out(1)	Output	Indicates PLL of the GTPE2_COMMON is achieved lock	Enabled when **Shared Logic in core** is selected.
gt0_pll0outclk_out gt0_pll1outclk_out gt0_pll0outrefclk_out gt0_pll1outrefclk_out gt1_pll0outclk_out gt1_pll1outclk_out gt1_pll0outrefclk_out gt1_pll1outrefclk_out(1)	Output	Clock outputs generated by GTPE2_COMMON	Enabled when **Shared Logic in core** is selected.
gt<quad>_qpllock_out(2)(3)	Output	Indicates PLL of the GTXE2_COMMON/GTHE2_COMMON has achieved lock	Enabled when **Shared Logic in core** is selected.
gt<quad>_qpllrefclklost_out(2)(3)	Output	Indicates `refclklost` port of the GTXE2_COMMON/GTHE2_COMMON	Enabled when **Shared Logic in core** is selected.
gt_qpllclk_quad<quad>_out gt_qpllclk_quad<quad>_out(2)(3)	Output	Clock outputs generated by GTXE2_COMMON/GTHE2_COMMON	Enabled when **Shared Logic in core** is selected.

Notes:

1. Ports from GTPE2_COMMON are applicable only to Artix-7 FPGA GTP transceiver designs.
2. Ports from GTXE2_COMMON/GTHE2_COMMON are applicable only to 7 series FPGA GTX/GTH transceiver designs.
3. These ports are enabled for each selected quad. <*quad*> refers to the transceiver quad numbered from 1 to 12.

Clock Compensation

Clock compensation is a feature allowing up to ±100 ppm difference in the reference clock frequencies used on each side of an Aurora 8B/10B channel. A standard clock compensation module is generated with the core in accordance with the *Aurora 8B/10B Protocol Specification* (SP002) [Ref 4]. 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, `do_cc` and `warn_cc` of the interface can be tied to ground to disable clock compensation.

Clock Compensation Interface

Table 2-21 describes the function of the clock compensation interface ports.

Table 2-21: **Clock Compensation I/O Ports**

Name	Direction	Clock Domain	Description
do_cc	Input	user_clk	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	user_clk	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.

Figure 2-28 and Figure 2-29 are waveform diagrams showing how the clock compensation signal works.

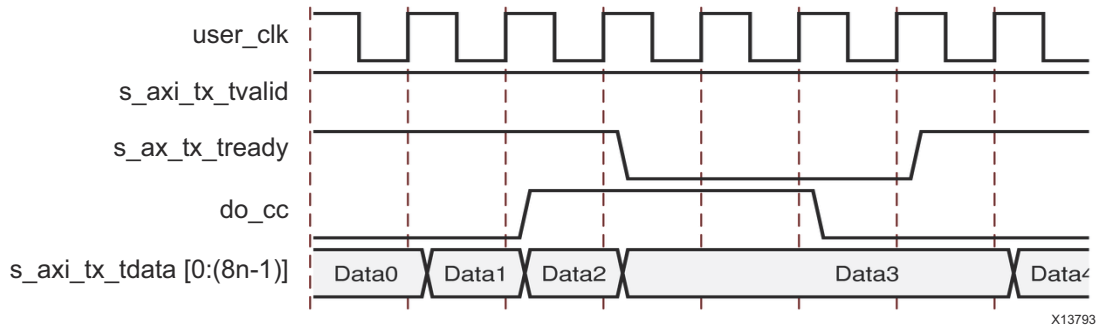


Figure 2-28: **Streaming Data with Clock Compensation Inserted**

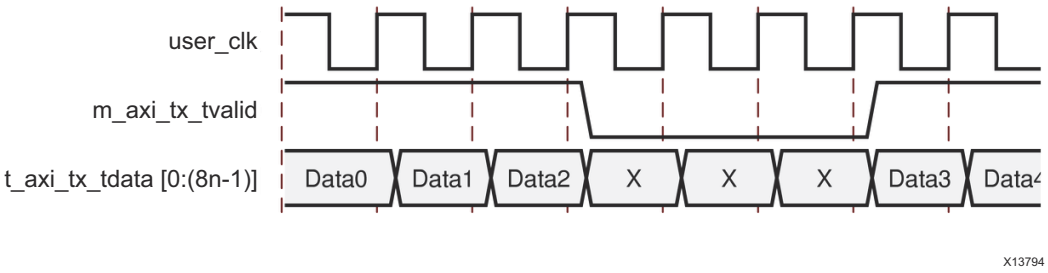


Figure 2-29: **Data Reception Interrupted by Clock Compensation**

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 transceiver data interface. Table 2-22 shows the required durations and periods for 2-byte and 4-byte wide lanes.

Table 2-22: **Clock Compensation Cycles**

Lane Width	USER_CLK Cycles Between DO_CC	DO_CC Duration (USER_CLK cycles)
2	5,000	6
4	2,500	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 2-23 shows the number of lookahead cycles required.

Table 2-23: **Lookahead Cycles**

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

Table 2-23: **Lookahead Cycles *(Cont'd)***

Data Interface Width	UFC Message Size	WARN_CC Lookahead
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. This module must always be connected to the clock compensation port on the Aurora 8B/10B module, except in special cases. See Table 2-21, page 51 for descriptions of the clock compensation interface ports.

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.

Note: For Vivado IP Integrator (IPI), The Aurora 8B/10B example design can be generated from the IP catalog and, using the `standard_cc_module.v` file, update the example design as a standalone package for use in IPI. Refer to *Packaging Custom AXI IP for Vivado IP Integrator Application Note* (XAPP1168) [Ref 16], or Packaging Custom IP for use with IP Integrator for the steps to package the standard CC module.

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 can reduce EMI.
- The `do_cc` signal has no effect until after `lane_up`; `do_cc` should be asserted immediately after `lane_up` because no clock compensation can occur during initialization

CRC

The CRC module provides a 16-bit or 32-bit CRC, implemented for user data. Table 2-24 describes the CRC module ports.

Table 2-24: **CRC Module Ports**

Port Name	Direction	Clock Domain	Description
crc_valid	Output	user_clk	CRC valid port. When asserted High, enables sampling the `crc_pass_fail_n` signal.
crc_pass_fail_n	Output	user_clk	The `crc_pass_fail_n` signal is asserted High on transmit and receive when the CRC values for both the transmitter and receiver match. The `crc_pass_fail_n` signal should only be sampled with the `crc_valid` signal.

See Using CRC, page 66 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 with flip-flops between the user application and the core. 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.

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).

Serial Transceiver Reference Clock Interface

Functional Description

The core requires a high-quality, low-jitter reference clock to drive the high-speed TX clock and clock recovery circuits in the transceivers. It also requires at least one frequency-locked parallel clock for synchronous operation with the user application. The Aurora 8B/10B core configures CPLL in UltraScale, Virtex®-7, Kintex®-7, and Zynq®-7000 family designs.

Clock Interface Ports for the Aurora 8B/10B Core

See Table 2-19, page 48 for descriptions of the transceiver ports on the clock interface.

Clocking from Neighboring Transceiver Quads

The Xilinx implementation tools make the necessary adjustments to the north-south routing and pin swapping to the 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 total number of GTX or GTH transceiver quads sourced by an external clock pin pair (`mgtrefclkn/mgtrefclkp`) in 7 series FPGAs must not exceed three quads (one quad above and one quad below), or 12 GTXE2_CHANNEL/GTHE2_CHANNEL transceivers. Designs with more than 12 transceivers or more than three quads in 7 Series FPGAs should use multiple external clock pins.
- The total number of transceiver quads sourced by an external clock pin pair (`mgtrefclkn/mgtrefclkp`) in UltraScale architecture FPGAs must not exceed five quads (two quads above and two quads below), or 20 GTHE3_CHANNEL transceivers.

IMPORTANT: *Manual edits are not recommended, but are possible using the recommendations in the UltraScale FPGAs GTH Transceivers User Guide (UG576) [Ref 1] and 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 3].*

Reset and Power Down

Reset

The reset signals are used to set the Aurora 8B/10B core to a known starting state. On reset, the core stops any current operation and re-initializes a new channel.

On full-duplex modules, the RESET signal resets both the TX and RX sides of the channel. On simplex modules, `tx_system_reset` resets TX channels and `rx_system_reset` resets RX channels. The `gt_reset` signal resets the transceivers which eventually resets the core.

Note: The `tx_system_reset` is separate from the `tx_reset` and `rx_reset` signals used on the simplex sideband interface.

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. As a result, `channel_up` is deasserted after three `user_clk` cycles as shown in Figure 3-1.

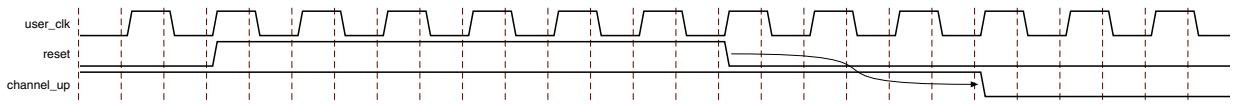


Figure 3-1: **RESET assertion in duplex core**

Use Case 2: gt_reset assertion in duplex core

Figure 3-2 shows the `gt_reset` assertion in the duplex core and should be a minimum of six `init_clk_in` cycles. As a result, `user_clk` is stopped after a few clock cycles because there is no `txoutclk` from the transceiver and `channel_up` is subsequently deasserted.

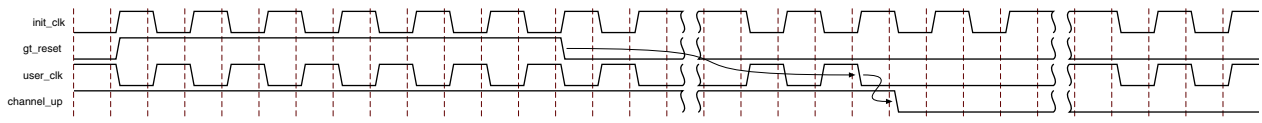


Figure 3-2: **gt_reset assertion in duplex core**

Use Case 3: tx_system_reset and rx_system_reset assertion in simplex core

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

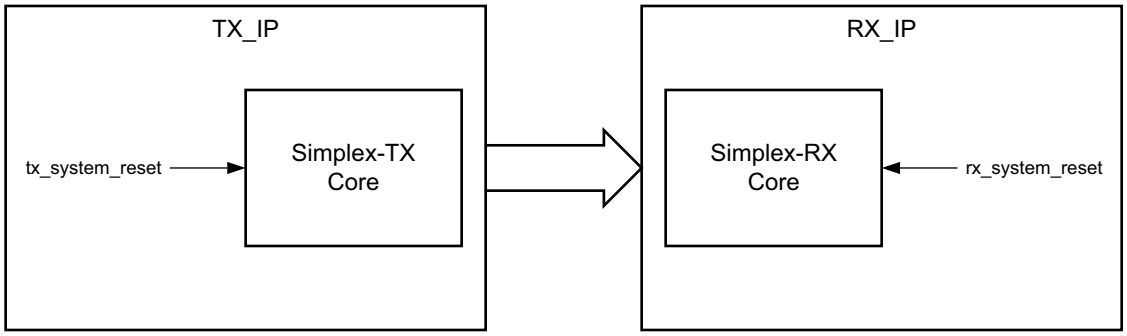


Figure 3-3: **System with Simplex Cores**

Figure 3-4 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 `user_clk` 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 8B/10B protocol channel initialization sequence transmission for the configured time. Assertion of `tx_channel_up` last ensures that the Simplex-TX core transmits the Aurora initialization sequence when the Simplex-RX core is ready.

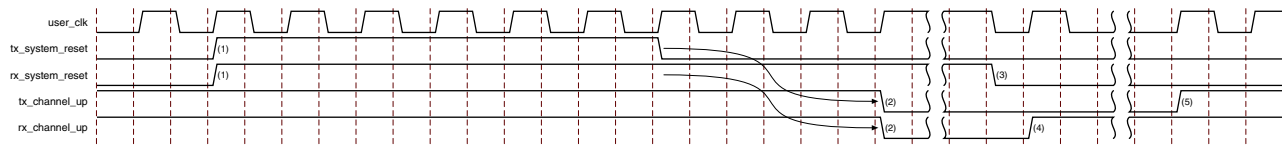


Figure 3-4: **tx_system_reset and rx_system_reset assertion in simplex core**

Aurora 8B/10B Duplex Power On Sequence

During the board power on sequence, both `gt_reset` and `reset` signals must be High. The transceiver reference clock and the core free running clocks are expected to be stable during power on for the proper functioning of the Aurora 8B/10B core. After the deassertion of `gt_reset`, the reset signal is expected to be deasserted synchronously with `user_clk` (Figure 3-5).

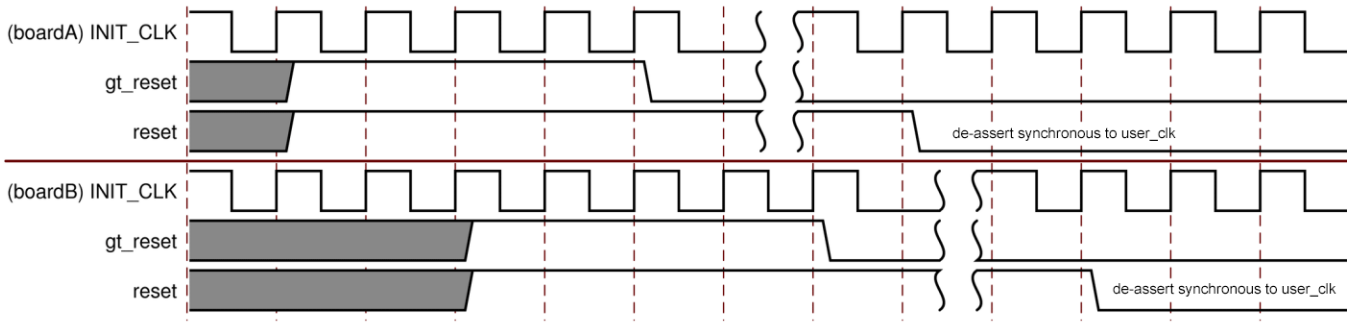


Figure 3-5: **Aurora 8B/10B Duplex Power On Sequence**

Aurora 8B/10B Duplex Normal Operation Reset Sequence

During normal operation, the `reset` signal is expected to be asserted for at least 128 `user_clk` cycles before assertion of the `gt_reset` signal to ensure that the portion of the core in programmable logic reaches a known reset state before the `user_clk` signal is suppressed due to the assertion of `gt_reset` (Figure 3-6).

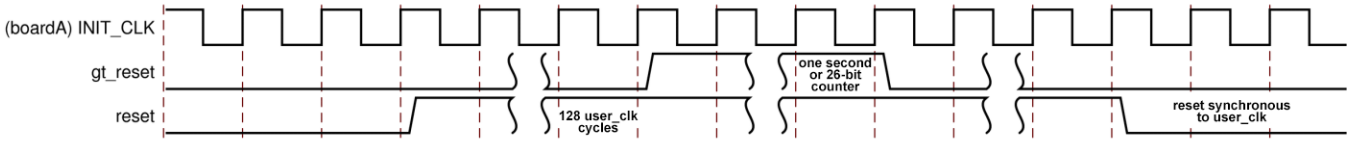


Figure 3-6: **Aurora 8B/10B Duplex Normal Operation Reset Sequence**

Aurora 8B/10B Simplex Power On Sequence

During power on, the `gt_reset` and `reset` signals of both TX simplex and RX simplex cores are expected to be High. It is expected that `INIT_CLK` and `GT_REFCLK` are stable during power-on. The `gt_reset` signal on the TX board must be deasserted first, followed by the deassertion of `gt_reset` on the RX side; this ensures proper CDR lock on the RX side (Figure 3-7).

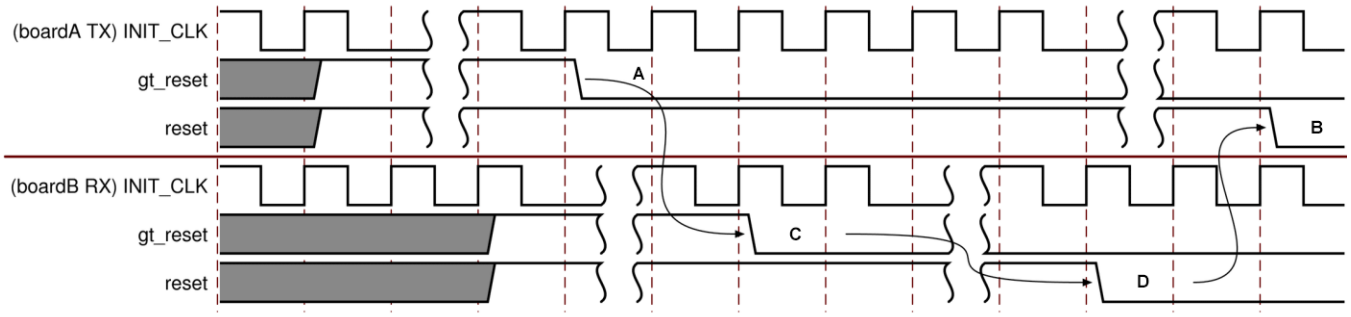


Figure 3-7: **Aurora 8B/10B Simplex Power On Reset Sequence**

Simplex power-on sequence:

1. Deassert TX-side `gt_reset` (A)
2. Deassert RX-side `gt_reset` (C)
3. Deassert RX-side reset synchronous to `user_clk` (D)
4. Deassert TX-side reset synchronous to `user_clk` (B)

Note: Care must be taken to ensure that the (D) to (B) time difference is as minimal as possible.

Aurora 8B/10B Simplex Normal Operation Reset Sequence

For the simplex configuration, it is recommended that the TX side reset sequence is tightly coupled with the RX side reset sequence because the TX and RX links do not have a communication feedback path. Note that if the RX side is reset, there is no direct mechanism to notify the TX side of the reset. Hence, for Aurora 8B/10B simplex cores, reset coupling needs to be handled at the system level. Every TX-side reset must be followed by the RX-side and, as shown in Figure 3-8, the time between RX-side reset de-assertion and TX-side reset de-assertion must be kept as minimal as possible. Before asserting `gt_reset`, a minimum of 128 clock cycles is required for ensuring that the portion of the core in programmable logic reaches a known reset state before the `user_clk` is suppressed by the assertion of `gt_reset`. The assertion time of `gt_reset` must be a minimum of six `init_clk` cycles, to satisfy the de-bouncing circuit included in the core.

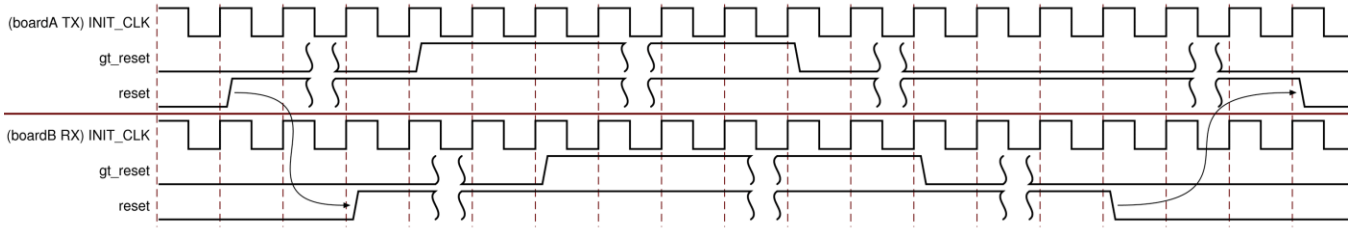


Figure 3-8: **Aurora 8B/10B Simplex Normal Operation Reset Sequence**

Power Down

This is an active-High signal. When `powerdown` is asserted, the transceivers in the Aurora 8B/10B core are turned off, putting them into a non-operating, low-power mode. When `powerdown` is deasserted, the core automatically resets. `gt_reset` must be asserted after `powerdown` deassertion as indicated in the guidelines of the transceiver user guide.

CAUTION! *Use care when asserting the `powerdown` signal on cores that use `tx_out_clk` (see Serial Transceiver Reference Clock Interface, page 56). tx_out_clk stops when the GTP, GTX, and GTH transceivers are powered down. See the 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 3], 7 Series FPGAs GTP Transceivers User Guide (ug482) [Ref 2] and the UltraScale Architecture GTH Transceivers User Guide (UG576) [Ref 1].*

Core Features

Shared Logic

The shared logic option in the Vivado® IDE configures the core to include sharable resources such as the transceiver quad PLL (QPLL), the transceiver differential `refclk` buffer (IBUFDS_GTE2), and clocking and reset logic in core or in the example design. When the **include shared logic in core** option is selected, all sharable resources are available to multiple instances of the core minimizing the amount of HDL modifications required while retaining the flexibility to address more use cases.

The shared logic hierarchy is called `<component_name>_support`. Figure 4-1 and Figure 4-2 show two hierarchies where the shared logic block is contained either in the core or in the example design. 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 5-3, page 73).

Note: Grayed blocks in Figure 4-1 and Figure 4-2 refer to the IP core.

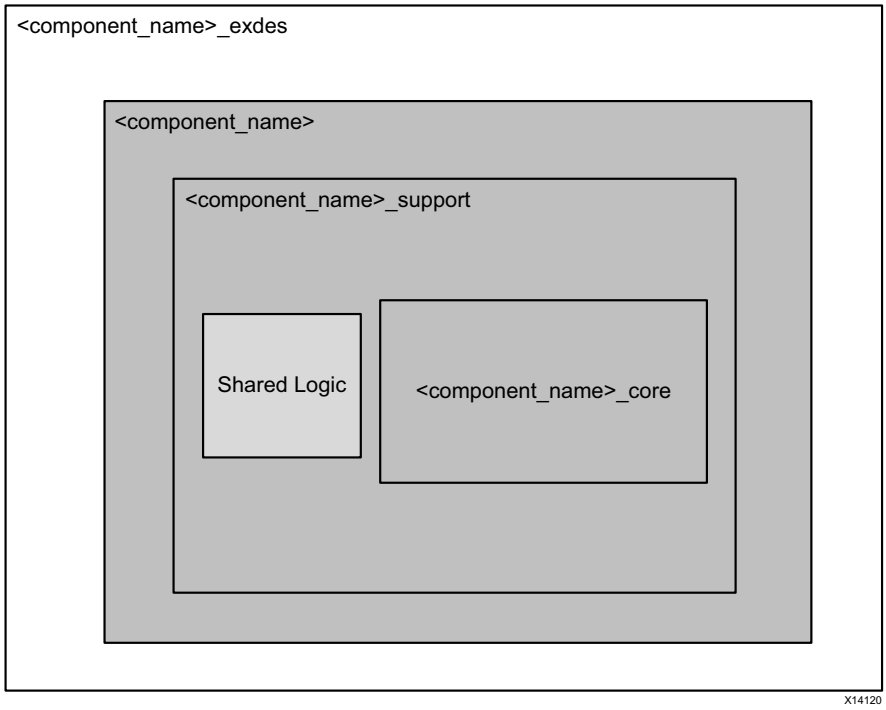


Figure 4-1: **Shared Logic Included in Core**

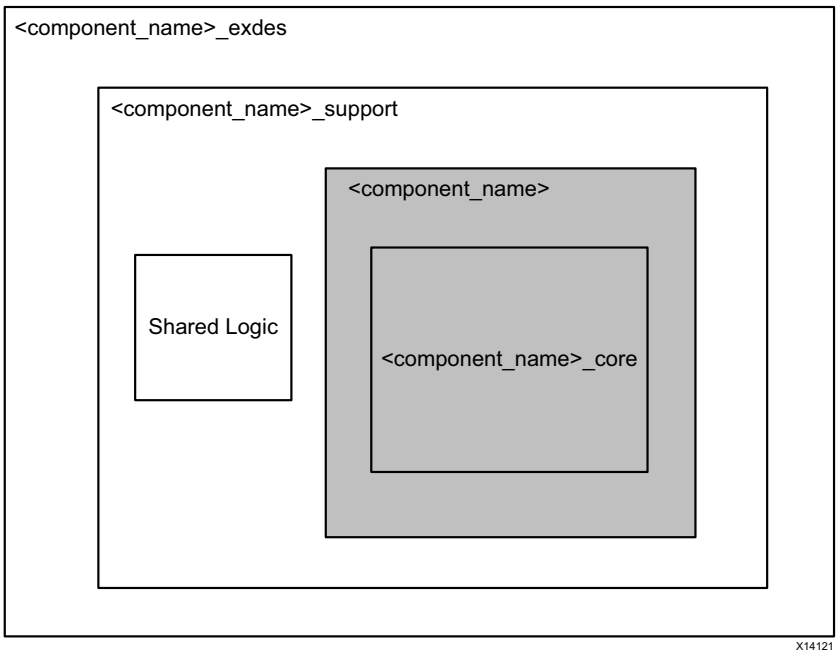


Figure 4-2: **Shared Logic Included in Example Design**

The contents of the shared logic depend upon the physical interface and the target device. Shared logic contains instance(s) of the transceiver differential buffer (IBUFDS_GTE2), support reset logic and an instantiation of <USER_COMPONENT_NAME:>_ CLOCK_MODULE. Shared logic also contains an instance of the transceiver common, GTPE2_COMMON, GTXE2_COMMON or GTHE2_COMMON, based on the selected transceiver type. 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 for Zynq®-7000 and 7 series devices and instantiated in shared logic for uniformity with other Xilinx® serial connectivity cores.

Table 4-1 lists shared resources for each family.

Table 4-1: **Sharable Resources**

Transceiver Type used in the Aurora 8B/10B Core	Resources
Zynq-7000 and 7 series device GTP transceivers in 2-byte mode	IBUFDS_GTE2: transceiver reference clock GTPE2_COMMON: transceiver clocking; BUFG: clocking IBUFDS: init_clk
Zynq-7000 and 7 series device GTP transceivers in 4-byte mode	IBUFDS_GTE2: transceiver reference clock GTPE2_COMMON: transceiver clocking MMCM: clocking 2xBUFG: clocking; IBUFDS:init_clk
Zynq-7000 and 7 series device GTX/GTH Transceivers	IBUFDS_GTE2: transceiver reference clock GTXE2_COMMON: GTX transceiver clocking GTHE2_COMMON: GTH transceiver clocking BUFG: clocking IBUFDS: init_clk
UltraScale™ GTH Transceivers	IBUFDS_GTE3: transceiver reference clock BUFG_GT: clocking

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.

Figure 4-3 shows the sharable resource connections from the core having shared logic included (aurora_8b10b_0) to the instance of another core without shared logic (aurora_8b10b_1). Some ports might change based on the core configuration and the type of transceiver selected. Table 2-20, page 49 provides details about the port changes due to selection of the Shared Logic option.

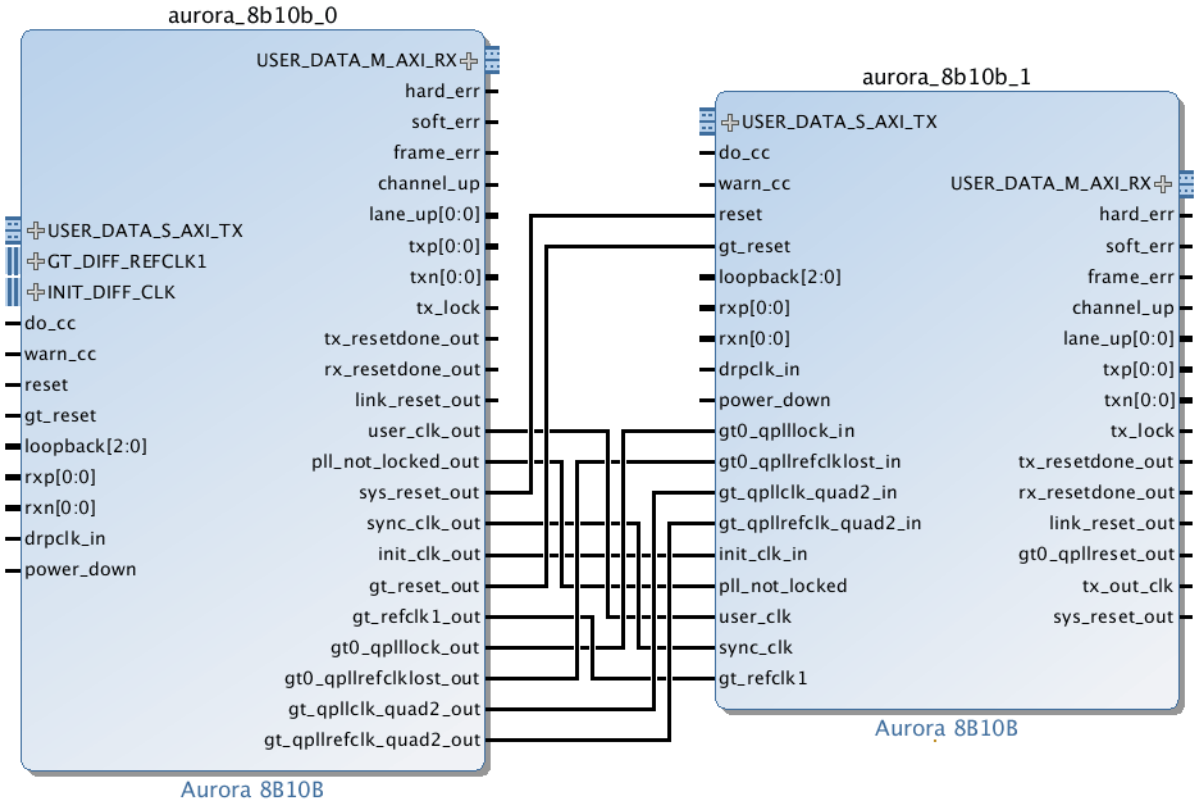


Figure 4-3: **Sharable Resource Connection Example Using IP Integrator**

Using the Scrambler/Descrambler

A 16-bit additive scrambler/descrambler, implemented for data with the polynomial: G(x) = X16 + X5 + X4 + X3 + 1, is available in the `<component name>_scrambler.v[hd]` module.

It ensures non-occurrence of repetitive data over long periods of time. The scrambler and descrambler are synchronized based on transmission and reception of the clock compensation characters respectively. `do_cc` must be asserted to load the seed value of the scrambler and descrambler simultaneously. Thus, the `standard_cc_module` that comes with the Aurora 8B/10B example design should always be used if the **Use Scrambler/Descrambler** option is selected in the Vivado Integrated Design Environment.

Note: The scrambler affects data symbols only.

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 2-24, page 54).

Hot-Plug Logic

Hot-plug logic in Aurora 8B/10B (using the free-running `init_clk` signal) is based on the received clock compensation characters. Reception of clock compensation characters by the Aurora RX interface implies that the 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.

IMPORTANT: *To ensure predictable link operation, It is highly recommended that hot-plug logic is not disabled.*

Using Little Endian Support

The Aurora 8B/10B core supports user interfaces in big endian format by default. It also supports the little endian format to connect seamlessly to AXI4-Stream compliant IP cores.

Design Flow Steps

This chapter describes customizing, generating, constraining and the simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about the standard design flows in the Vivado® IP Integrator can be found in these Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 6]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 8]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 9]

Customizing and Generating the Core

This section includes information about using Xilinx tools to customize and generate the Aurora 8B/10B core in the Vivado design suite.

When customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 6] for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

IP can be customized for use in a design by specifying values for the various parameters associated with the IP core using these steps:

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

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 8].

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

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 Customize IP interface.

Customize IP

Figure 5-1 shows the Core Options tab of the Customize IP interface with the default options for Zynq®-7000 and 7 series devices. 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. Figure 5-2 shows the Core Options tab for UltraScale™ devices.

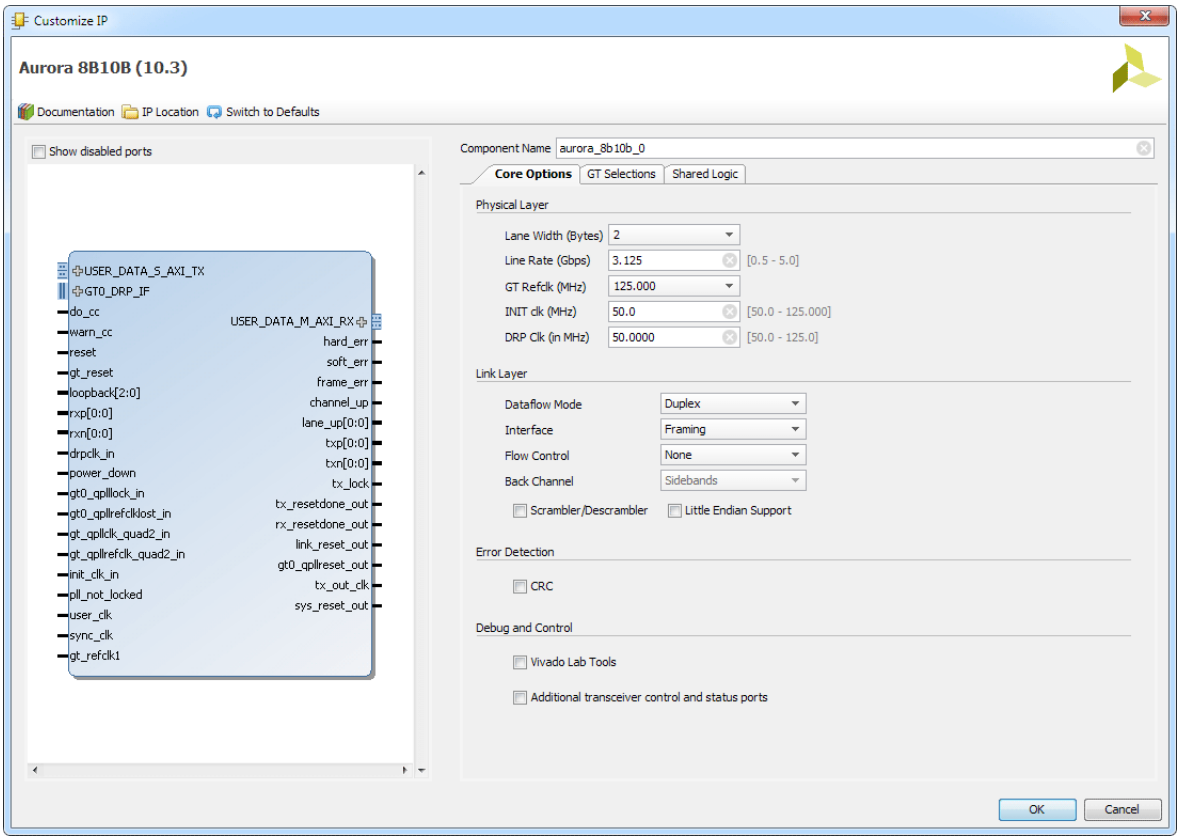


Figure 5-1: **Aurora 8B/10B Core Options Tab for Zynq-7000 and 7 Series Devices**

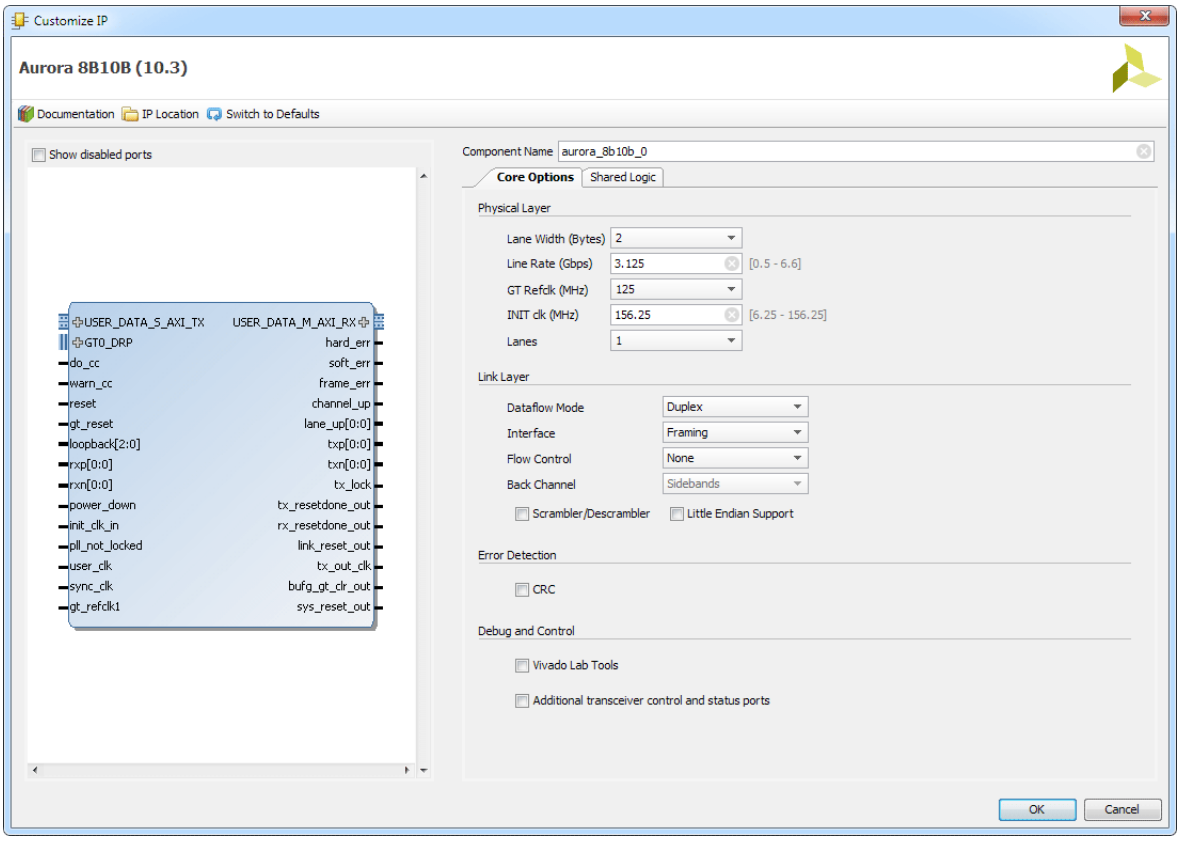


Figure 5-2: **Aurora 8B/10B Core Options Tab for UltraScale Devices**

The second tab, GT Selections, is shown in Figure 5-4, page 74 for Virtex®-7 and Kintex®-7 FPGA GTX and GTH transceivers.

Component Name

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

Default: aurora_8b10b_0

Lane Width

Select the byte width of the 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 line rate value in gigabits per second within the valid range from 0.5 (Gb/s) to 6.6 (Gb/s).

This value is 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 device family data sheet for details on the line rate limits).

Default: 3.125 Gb/s

GT REFCLK (MHz)

Select a reference clock frequency for the transceiver from the drop-down list. Reference clock frequencies 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

INIT clk (MHz)

Enter a valid INIT clock frequency into the text box.

Default: 50 MHz for Zynq-7000 and 7 series devices, (`line_rate/lane_width`) for UltraScale devices.

DRP clk (MHz)

Enter a valid DRP clock frequency into the text box. INIT clock and DRP clock frequencies are the same for UltraScale devices.

Default: 50 MHz

Dataflow Mode

Select the options for the direction of the channel which 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**.

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 AXI4-Stream interface to stream data through the Aurora 8B/10B channel.

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 sent to them. Immediate mode allows idle codes to be inserted within data frames while completion mode only inserts idle codes between complete data frames.

Available options follow:

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

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

Little Endian Support

Select to change all of the interface(s) to little endian format. See Using Little Endian Support in Chapter 4 for more information. By default, the core uses big endian format.

Default: Unchecked

Additional Transceiver Control and Status Ports

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

Default: Unchecked

Vivado Lab Tools

Select to add Vivado lab tool cores to the Aurora 8B/10B core. 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. Depending on the Lane Width of 2 or 4, the core implements CRC16 or CRC32 respectively. See Using CRC in Chapter 4 for more information.

Default: Unchecked

Shared Logic

Figure 5-3 shows the Shared Logic tab of the Customize IP interface.

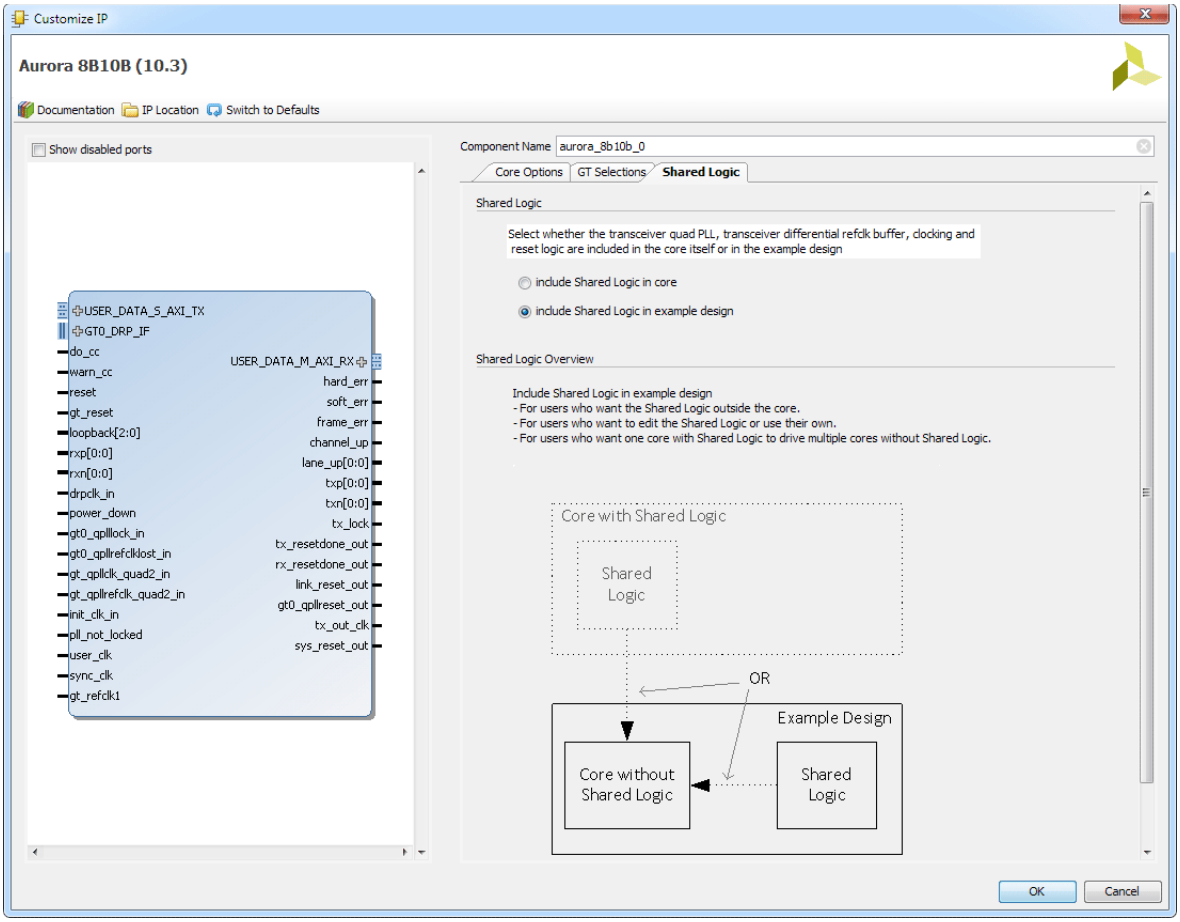


Figure 5-3: **Aurora 8B/10B Shared Logic Tab**

Select the option to include the transceiver common PLL and its logic either in the 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

Figure 5-4 shows the GT Selections tab of the Customize IP interface.

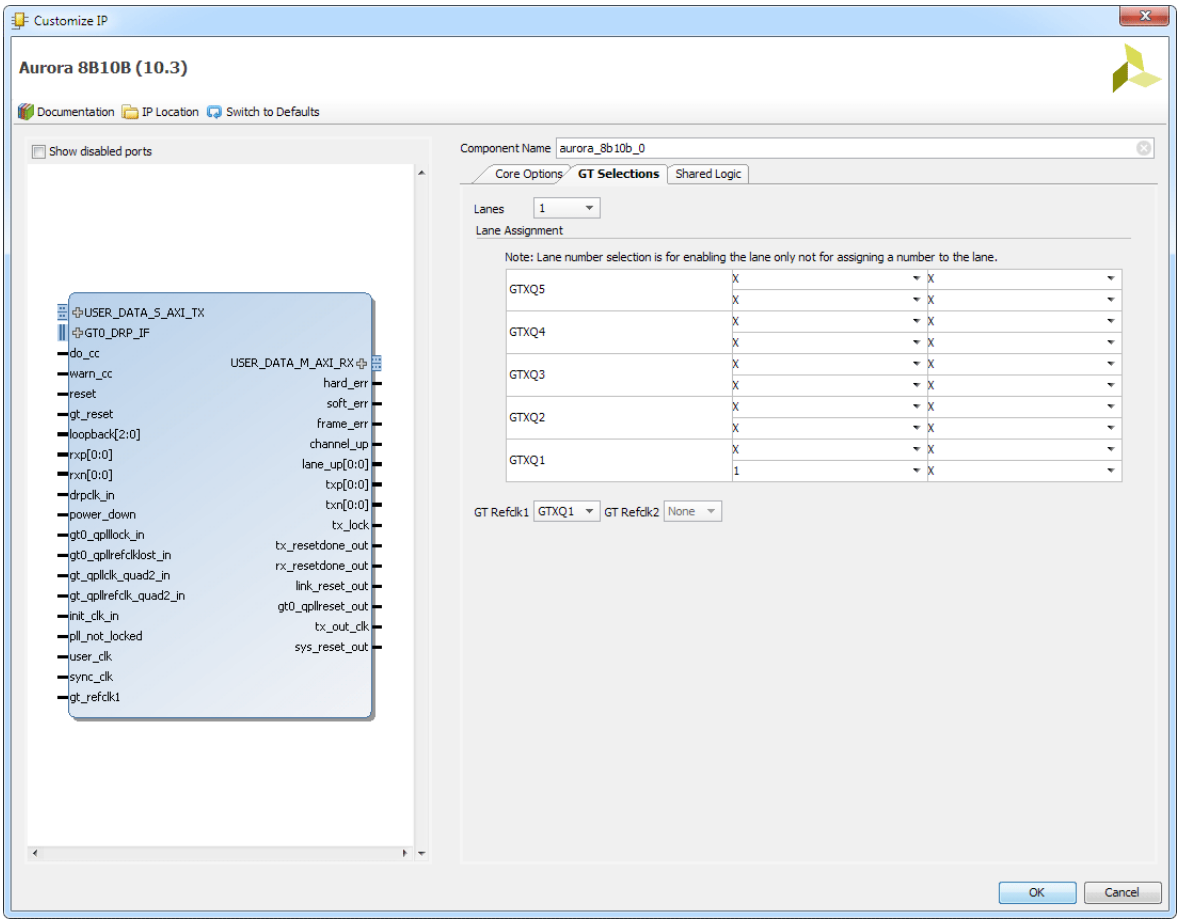


Figure 5-4: **Aurora 8B/10B GT Selections Tab**

Column/Row Used

Select the appropriate column/row of transceivers used from the drop-down list. The column used is enabled only for Virtex-7 and Kintex-7 devices and the row used is enabled only for Artix®-7 devices.

Default: left/top

Lanes

Select the number of lanes (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 5-4. Two rows or four boxes represent a quad. Each active box represents an available transceiver. A tooltip is provided to specify which transceiver (for example, GTXE2_CHANNEL_X0Y0) is being implemented in hardware.

The Aurora 8B/10B core generates transceiver placement (LOC) constraints in ascending fashion. Lane numbering serves only to enable the lanes and not to assign lane numbers.

RECOMMENDED: *Lane Assignment is not available for UltraScale architecture based designs. It is strongly recommended that lane selection should be continuous for timing closure. It is necessary to update the transceiver location for your design requirements in the file at the path shown .*

Path:

```
<project_name>.srcs/sources_1/ip/<component_name>/ip_0/synth/
<component_name>_gt.xdc
```

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 Source 1: GTPQ*n*/GTXQ*n*/GTHQ*n*; GT REFCLK Source 2: None

Note: The value of *n* depends upon the serial transceiver (GTX or GTH) position.

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 78 for details about the example_design directory and files.

User Parameters

Table 5-1 shows the relationship between the fields in the Vivado IDE and the User Parameters (which can be viewed in the Tcl console).

Table 5-1: **GUI Parameter to User Parameter Relationship**

GUI Parameter/Value[1]	User Parameter/Value[1]	Default Value
Core Options		
Physical Layer		
Lane Width (Bytes)	C_LANE_WIDTH	2
Line Rate (Gbps)	C_LINE_RATE	3.125
GT Refclk (MHz)	C_REFCLK_FREQUENCY	125.000
INIT clk (MHz)	C_INIT_CLK	50.000
DRP clk (MHz)[4]	DRP_FREQ	50.000
Link Layer		
Dataflow Mode	Dataflow_Config	Duplex
Interface	Interface_Mode	Framing
Flow Control	Flow_Mode	None
Back Channel	Backchannel_mode	Sidebands
Scrambler/Descrambler	C_USE_SCRAMBLER	false
Little Endian Support	C_USE_BYTESWAP	false
Error Detection		
CRC	C_USE_CRC	false
Debug and Control		
Vivado Lab Tools	C_USE_CHIPSCOPE	false
Additional transceiver control and status ports	TransceiverControl	false
GT Selections		
Lanes	C_AURORA_LANES	1
Lane Assignment[2][4]		
Select transceiver to include GTXE2_CHANNEL_X0Y0 in your design	C_GT_LOC_1	1
Select transceiver to include GTXE2_CHANNEL_X0Y1 in your design	C_GT_LOC_2	X
Select transceiver to include GTXE2_CHANNEL_X0Y2 in your design	C_GT_LOC_3	X
Select transceiver to include GTXE2_CHANNEL_X0Y3 in your design	C_GT_LOC_4	X
Select transceiver to include GTXE2_CHANNEL_X0Y4 in your design	C_GT_LOC_5	X

Table 5-1: **GUI Parameter to User Parameter Relationship *(Cont'd)***

GUI Parameter/Value[1]	User Parameter/Value[1]	Default Value
Select transceiver to include GTXE2_CHANNEL_X0Y5 in your design	C_GT_LOC_6	X
Select transceiver to include GTXE2_CHANNEL_X0Y5 in your design	C_GT_LOC_7	X
Select transceiver to include GTXE2_CHANNEL_X0Y7 in your design	C_GT_LOC_8	X
Select transceiver to include GTXE2_CHANNEL_X0Y8 in your design	C_GT_LOC_9	X
Select transceiver to include GTXE2_CHANNEL_X0Y9 in your design	C_GT_LOC_10	X
Select transceiver to include GTXE2_CHANNEL_X0Y10 in your design	C_GT_LOC_11	X
Select transceiver to include GTXE2_CHANNEL_X0Y11 in your design	C_GT_LOC_12	X
Select transceiver to include GTXE2_CHANNEL_X0Y12 in your design	C_GT_LOC_13	X
Select transceiver to include GTXE2_CHANNEL_X0Y13 in your design	C_GT_LOC_14	X
Select transceiver to include GTXE2_CHANNEL_X0Y14 in your design	C_GT_LOC_15	X
Select transceiver to include GTXE2_CHANNEL_X0Y15 in your design	C_GT_LOC_16	X
GT Refclk (MHz)		
GT Refclk1[4]	C_GT_CLOCK_1	GTXQ0
GT Refclk2[4]	C_GT_CLOCK_2	None
Shared Logic		
Include Shared Logic in core[3]	SupportLevel	1
Include Shared Logic in example design (Default Mode)		0

1. Parameter values are listed in the table where the GUI parameter value differs from the user parameter value. Such values are shown in this table as indented below the associated parameter.
2. X0Y0 GT selection is based on column.
3. If Shared Logic in core option is selected, SupportLevel is 1.
4. Not available with UltraScale devices.

Output Generation

The customized Aurora 8B/10B core is delivered as a set of HDL source modules in the language selected. 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 Figure 5-5.

Figure 5-5: **Aurora 8B/10B Directory Structure**

Note: If the VHDL language is selected for UltraScale devices, the IP top-level wrapper file is VHDL and the underlying design source files are Verilog.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7].

Constraining the Core

This section provides information for constraining the Aurora 8B/10B core in the Vivado design suite.

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

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 example design uses a minimum of one and a maximum of two reference clocks. The number of GT reference clocks is dependant upon transceiver selection. The GT REFCLK value selected on the first page of the Vivado IDE is used to constrain the GT reference clock using the create_clock XDC command.

  Note: For UltraScale devices, the GT reference clock location constraint should be added to `<user_component_name>_example.srcs/constrs_1/imports/<user_component_name>_example.xdc`.

- TXOUTCLK clock constraint

  TXOUTCLK is generated by the transceiver based on the input reference clock and the divider settings of the transceiver. The create_clock XDC command is used to constrain TXOUTCLK.

- INIT CLK constraint

  The Aurora 8B/10B core example design uses a debounce circuit to sample GT_RESET which is clocked asynchronously by the system clock. The create_clock XDC command is used to constrain the system clock.

RECOMMENDED: *It is recommended to use a system clock frequency lower than the GT reference clock frequency.*

Clock Management

Not Applicable

Clock Placement

Not Applicable

Banking

Not Applicable

Transceiver Placement

The set_property XDC command is used to constrain the transceiver location. This is provided as a tooltip 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 location of the GT reference clock pins.

False Paths

The set_false_path XDC command is used to constrain the false paths (signal crossing clock domain).

Example Design XDC

The generated verilog example design is configured with a two-byte lane width, 3.125 Gb/s line rate, and a 125.0 MHz reference clock. The XDC file generated for the XC7VX690T-FFG1761-2 device follows:

```
##############################################################################
## XDC generated for xc7vx690t-ffg1761-2 device
# 125.0MHz GT Reference clock constraint
create_clock -name GT_REFCLK1 -period 8.0 [get_ports GTHQ1_P]
######################## GT reference clock LOC ########################
set_property LOC AW9 [get_ports GTHQ1_N]
set_property LOC AW10 [get_ports GTHQ1_P]

# USER_CLK Constraint : Value is selected based on the line rate (3.125 Gbps) and lane width (2-Byte)
# create_clock -name user_clk_i -period 6.400 [get_pins aurora_module_i/clock_module_i/user_clk_buf_
i/I]

# 20.0 ns period Board Clock Constraint
create_clock -name init_clk_i -period 20.0 [get_ports INIT_CLK_P]
```

```
# 20.000 ns period DRP Clock Constraint
create_clock -name drp_clk_i -period 20.000 [get_ports DRP_CLK_IN]

###### CDC in RESET_LOGIC from INIT_CLK to USER_CLK ###############
set_false_path -through [get_pins -hier *cdc_to*]

###################### Locatoin constrain #########################
##Note: User should add LOC based upon the board
#       Below LOC's are place holders and need to be changed as per the device and board
#set_property LOC D17 [get_ports INIT_CLK_P]
#set_property LOC D18 [get_ports INIT_CLK_N]
#set_property LOC G19 [get_ports RESET]
#set_property LOC K18 [get_ports GT_RESET_IN]
#set_property LOC A20 [get_ports CHANNEL_UP]
#set_property LOC A17 [get_ports LANE_UP]
#set_property LOC Y15 [get_ports HARD_ERR]
#set_property LOC AH10 [get_ports SOFT_ERR]
#set_property LOC AD16 [get_ports ERR_COUNT[0]]
#set_property LOC Y19 [get_ports ERR_COUNT[1]]
#set_property LOC Y18 [get_ports ERR_COUNT[2]]
#set_property LOC AA18 [get_ports ERR_COUNT[3]]
#set_property LOC AB18 [get_ports ERR_COUNT[4]]
#set_property LOC AB19 [get_ports ERR_COUNT[5]]
#set_property LOC AC19 [get_ports ERR_COUNT[6]]
#set_property LOC AB17 [get_ports ERR_COUNT[7]]
#set_property LOC AC17 [get_ports FRAME_ERR]


#set_property LOC AG29 [get_ports DRP_CLK_IN]
#// DRP CLK needs a clock LOC

##Note: User should add IOSTANDARD based upon the board
#       Below IOSTANDARD's are place holders and need to be changed as per the device and board
#set_property IOSTANDARD DIFF_HSTL_II_18 [get_ports INIT_CLK_P]
#set_property IOSTANDARD DIFF_HSTL_II_18 [get_ports INIT_CLK_N]
#set_property IOSTANDARD LVCMOS18 [get_ports RESET]
#set_property IOSTANDARD LVCMOS18 [get_ports GT_RESET_IN]

#set_property IOSTANDARD LVCMOS18 [get_ports CHANNEL_UP]
#set_property IOSTANDARD LVCMOS18 [get_ports LANE_UP]
#set_property IOSTANDARD LVCMOS18 [get_ports HARD_ERR]
#set_property IOSTANDARD LVCMOS18 [get_ports SOFT_ERR]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[0]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[1]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[2]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[3]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[4]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[5]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[6]]
#set_property IOSTANDARD LVCMOS18 [get_ports ERR_COUNT[7]]
#set_property IOSTANDARD LVCMOS18 [get_ports FRAME_ERR]


#set_property IOSTANDARD LVCMOS18 [get_ports DRP_CLK_IN]
#// DRP CLK needs a clock IOSTDLOC

#####################################################################


################################ GT LOC ###################################
set_property LOC GTHE2_CHANNEL_X1Y4 [get_cells
aurora_module_i/aurora_8b10b_0_i/inst/gt_wrapper_i/aurora_8b10b_0_multi_gt_i/gt0_aurora_8b10b_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

This section contains information about simulating IP in the Vivado design suite. For comprehensive information about Vivado design suite simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 9]

The Aurora 8B/10B core delivers a demonstration test bench for the example design. The TEST COMPLETED SUCCESSFULLY message signifies the completion of the example design simulation.

Note: `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. Simplex core simulation requires partner generation. The partner core is generated automatically and the synthesized netlist is available under the simulation file set when clicking **Open IP Example Design**. Due to the synthesizing of the partner core, opening a simplex core example design takes more time than the duplex example design generation.

Note: Simulation requires that the **Labtools** option to be unchecked.

For more information, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7].

Synthesis and Implementation

This section contains information about synthesis and implementation in the Vivado design suite. For more details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7].

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 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 QuestaSim for simulation.

Implementing the Example Design

To generate the example design, follow these steps:

1. Right-click the generated IP and select **Open Example Design**.
2. Click **Run Implementation**.
3. When the implementation process completes, click **Generate Bitstream** to create a bitstream for the selected target device.

Note: The LOC and IO standards must be specified in the XDC file for all input and output ports of the design.

Generating the Core

To generate an Aurora 8B/10B core with default values using the Vivado design tools, see *Designing a System Using the Aurora 8B10B Core (Duplex) on the KC705 Evaluation Kit* (XAPP1193) [Ref 14].

Detailed Example Design

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

Directory and File Contents

See Output Generation, page 78 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.

The example design consists these components:

- Frame generator (FRAME_GEN) connected to the TX interface
- Frame checker (FRAME_CHECK) connected to the RX user interface
- VIO/ILA instance for debug and testing

Figure 6-1 illustrates the block diagram of the example design for a full-duplex core.

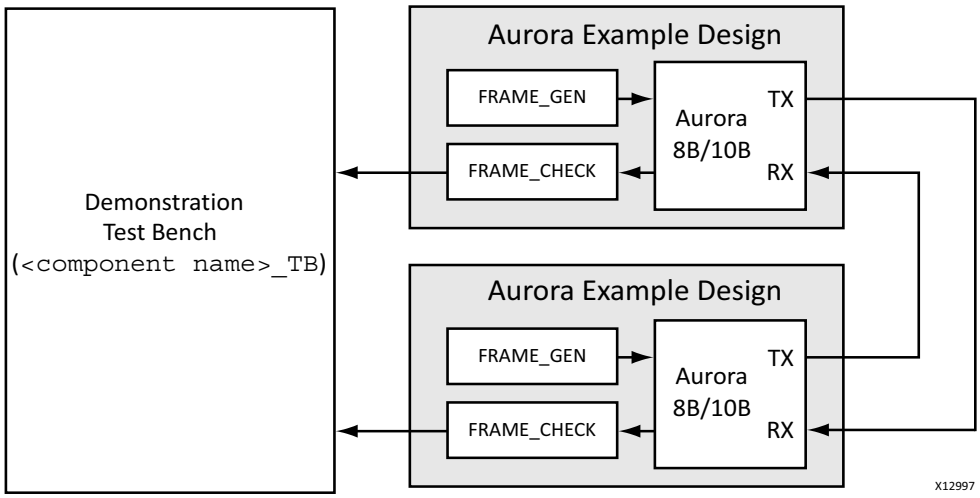


Figure 6-1: **Example Design**

The example design uses all of the core interfaces. Simplex cores without a TX or RX interface have no FRAME_GEN or FRAME_CHECK block, respectively.

The FRAME_GEN module generates user traffic to each of the PDU, UFC and NFC interfaces following AXI4-Stream protocol. This module contains a pseudo-random number generator using a linear feedback shift register (LFSR) with a specific initial value to generate a predictable sequence of data. The FRAME_CHECK module uses this data sequence to verify the integrity of the Aurora data channel. Module inputs are `user_clk`, `reset` and `channel_up`.

The FRAME_CHECK module verifies the integrity of the RX data. This module uses the same LFSR and initial value as the FRAME_GEN module to generate the expected RX frame data. The received user data is compared with the locally-generated stream and any errors are reported per the AXI4-Stream protocol. The FRAME_CHECK module is applicable to PDU, UFC and NFC interfaces.

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 more complicated interfaces of 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 to supply the correct pins and clock constraints.

Table 6-1 describes the ports of the example design.

Table 6-1: **Example Design I/O Ports**

Port	Direction	Clock Domain	Description
rxn[0:*m*–1]	Input	RX Serial Clock	Negative differential serial data input pin.
rxp[0:*m*–1]	Input	RX Serial Clock	Positive differential serial data input pin.
txn[0:*m*–1]	Output	TX Serial Clock	Negative differential serial data output pin.
txp[0:*m*–1]	Output	TX Serial Clock	Positive differential serial data output pin.
err_count[0:7]	Output	user_clk	Count of the number of data words received by the frame checker that did not match the expected value.
reset	Input	user_clk	Reset signal for the example design. The reset is debounced using a `user_clk` signal generated from the reference clock input.
gt_reset	Input	init_clk_in	GT Reset signal for the example design. `gt_reset` is debounced using the `init_clk_in` signal.
<*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 56 for details about the reference clocks.
<*core error signals*>(1)	Output	user_clk	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.
<*core channel up signals*>(1)	Output	user_clk	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.
<*core lane up signals*>(1)	Output	user_clk	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.

Table 6-1: **Example Design I/O Ports *(Cont'd)***

Port	Direction	Clock Domain	Description
<*simplex initialization signals*>(1)	Input/ Output	user_clk	If the core is a simplex core, its sideband initialization ports are registered and brought to the top level of the example design.

Notes:

1. See Status, Control, and the Transceiver Interface, page 36 for details.

Using Vivado Lab Tools

The Integrated Logic Analyzer (ILA) and Virtual Input Output (VIO) cores in the Vivado lab tools feature 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 on the Core Options tab of the Customize IP interface in the Vivado IDE to include the ILA and VIO cores in the example design. Alternatively, the USE_CHIPSCOPE parameter in the `<component name>_exdes` module can be set to 1 before running implementation.

See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 13].

Test Bench

The Aurora 8B/10B 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 8B/10B core configurations, but the basic functionality remains the same for all of the core configurations.

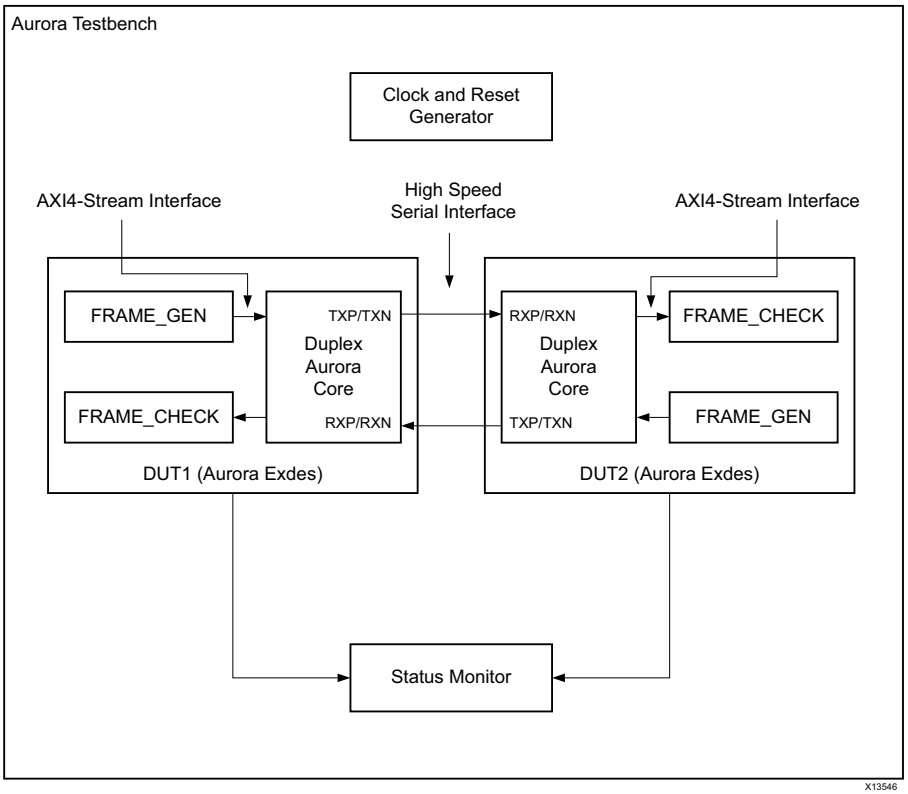


Figure 7-1: **Aurora Test Bench for Duplex Configuration**

The Aurora test bench environment connects the Aurora Duplex core in loopback using a high-speed serial interface. Figure 7-1 shows the Aurora test bench for the Duplex configuration.

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

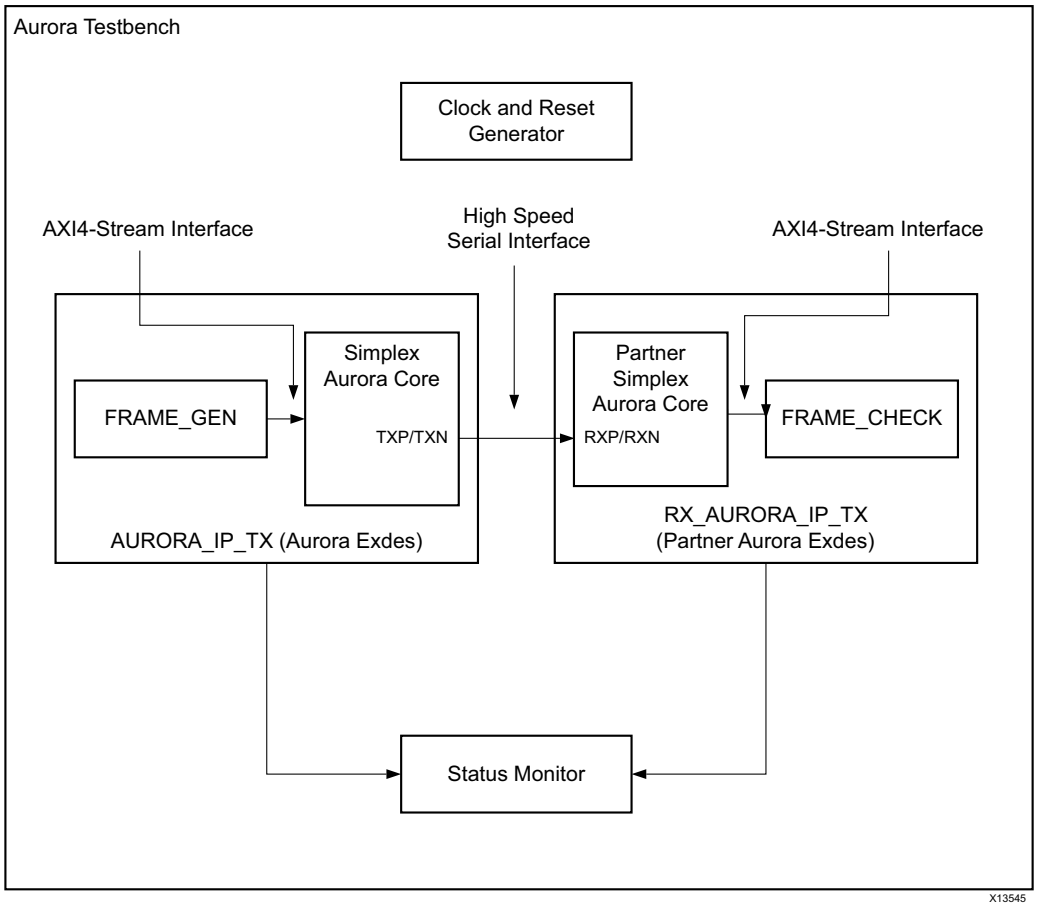


Figure 7-2: **Aurora Test Bench for Simplex Configuration**

The Aurora test bench environment connects the Aurora Simplex core to the partner Simplex Aurora core using the high-speed serial interface. Figure 7-2 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 first verifies the state of the transmitter and receiver channels, then monitors the integrity of the user data for a predetermined simulation time. The `tx_channel_up` and `rx_channel_up` assertion messages indicate successful link training and channel bonding (in case of multi-lane designs).

Verification, Compliance, and Interoperability

This appendix provides details about how this IP core was tested for compliance.

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.

Aurora 8B/10B cores are tested in hardware for functionality, performance, and reliability using Xilinx® evaluation platforms. 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.

A series of Aurora 8B/10B core test scenarios are validated using the various Xilinx development boards listed in Table A-1. These boards permit the prototyping of system designs where the Aurora 8B/10B core allows high-speed serial communication between two boards.

Table A-1: **Xilinx Development Boards**

Target Family	Evaluation Boards	Characterization Boards
7 series	ZC706, AC701, KC705, VC707, VC709	ZC720, ZC723, AC722, KC724, VC7203, VC7215
UltraScale architecture	KCU105, VCU107	UC1250, UC1283

Migrating and Upgrading

This appendix contains information about migrating a design from the Xilinx® ISE® Design Suite 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 10].

Device Migration

When migrating from a 7 series GTX or GTH transceiver device to an UltraScale architecture GTH transceiver device, the prefixes of the optional transceiver debug ports for single-lane cores are changed from "gt0", "gt1" to "gt", and the postfix "_in" and "_out" are dropped. For multi-lane cores, the prefixes of the optional transceiver debug ports, `gt(n)`, are aggregated into a single port. For example: `gt0_gtrxreset` and `gt1_gtrxreset` now become `gt_gtrxreset[1:0]`. This rule applies for all ports, with the exception of the DRP buses which follow the convention of `gt(n)_drpxyz`.

IMPORTANT: *Update your design to use the new transceiver debug port names. For more information about migration to UltraScale architecture devices, see the UltraScale Architecture Migration Methodology Guide (UG1026) [Ref 15].*

Upgrading in the Vivado Design Suite

In the latest revision of the core, there have been several changes to ensure pin-incompatibility with the previous core 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.

When updating from a previous version to a recent version 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 Aurora cores based on LocalLink (LL) to the AXI4-Stream Aurora core.

Prerequisites

- Vivado design tools build containing the Aurora 8B/10B 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 Data Sheet* (DS637) [Ref 11] and *LogiCORE IP Aurora 8B/10B User Guide* (UG353) [Ref 12] for reference.
- Migration guide (this appendix)

Limitations

This section outlines the limitations of the Aurora 8B/10B core for AXI4-Stream support. It is essential to observe 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 the packet, but the Aurora 8B/10B core expects at least one byte to be valid at 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 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 shim module is introduced in the AXI4-Stream Aurora 8B/10B 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 LL.
  - 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 shims 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.

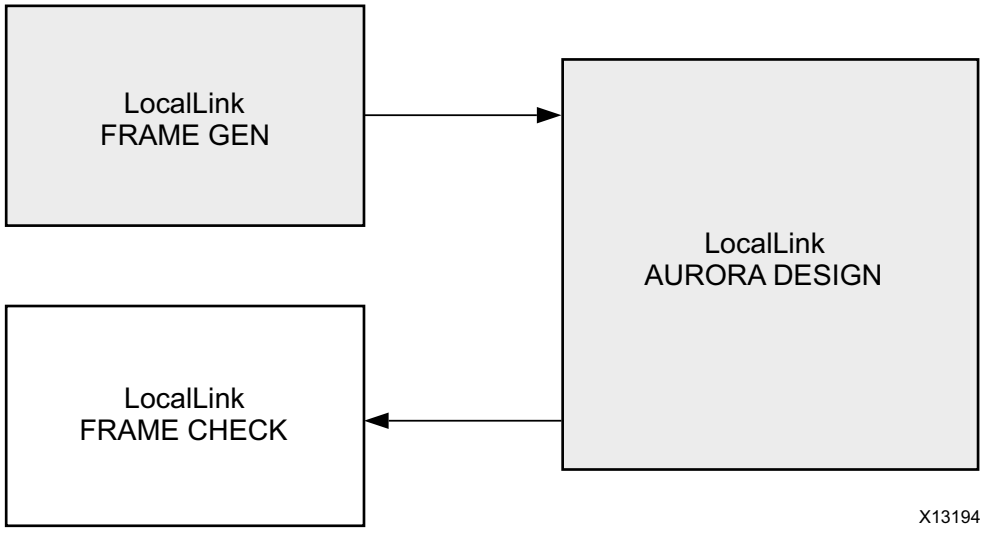


Figure B-1: **Legacy Aurora Example Design**

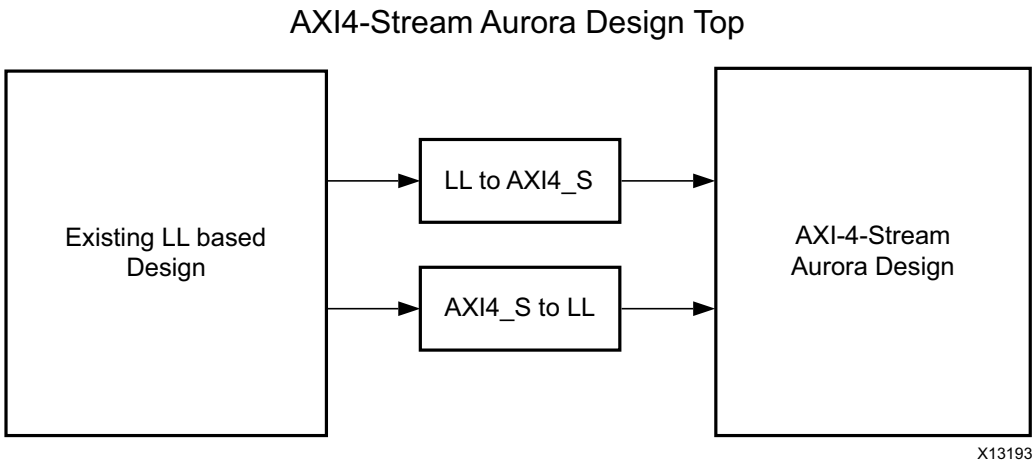


Figure B-2: **AXI4-Stream Aurora Example Design**

Migration Steps

Begin by generating an AXI4-Stream Aurora 8B/10B core from the Vivado Design Suite.

Simulate the Core

1. Click **Run Simulation** in the Vivado IDE and select the type of simulation.
2. QuestaSim 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. The frame checker continuously checks the received data and reports any data mismatch.
5. A TEST PASS or TEST FAIL status is printed on the QuestaSim console providing the status of the test.

Implement the Core

1. Click **Run Implementation** in the Vivado IDE to run synthesis and implementation.

Integrate to an Existing LocalLink-based Aurora 8B/10B Design

1. The Aurora 8B/10B core provides a light-weight shim to interface to any existing LL based interface. The shims are delivered along with the core.
2. See Figure B-2, page 94 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 94.
6. The latest AXI4-Stream Aurora core can now be used with any existing LL design.

Vivado IDE Changes

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

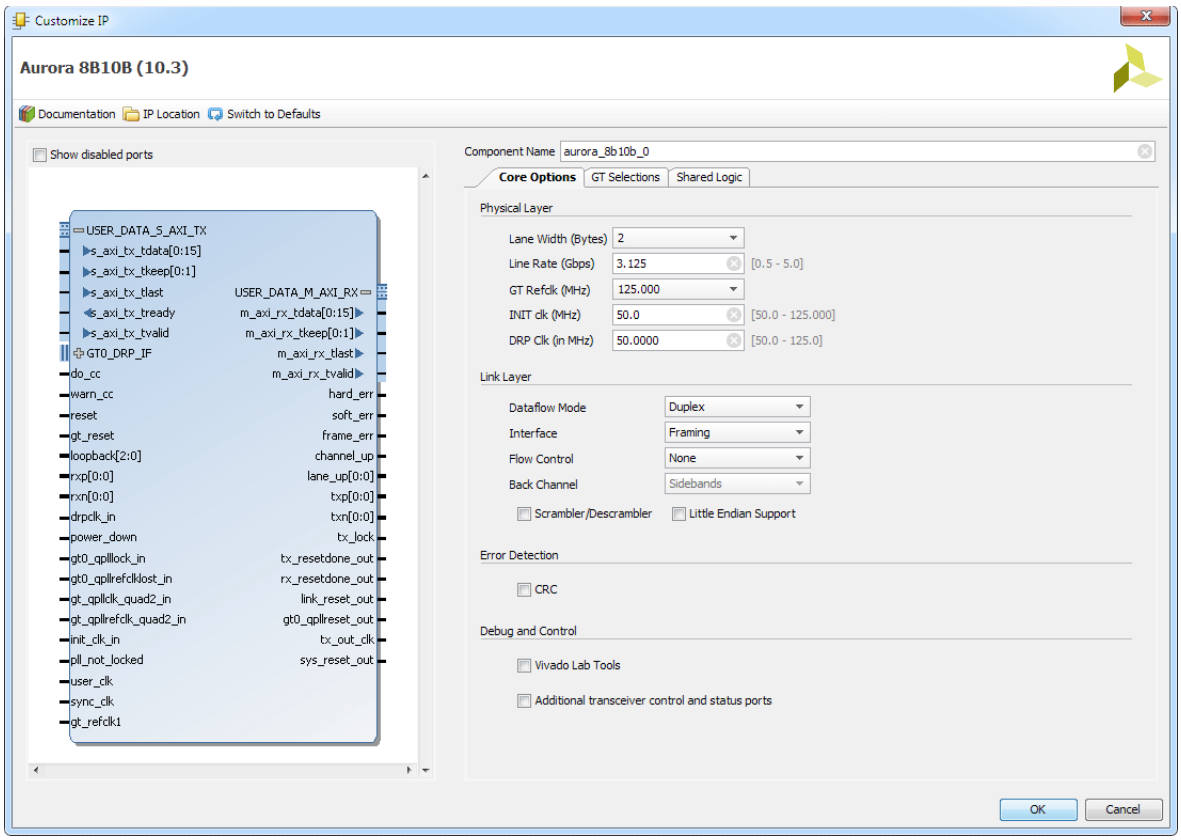


Figure B-3: **AXI4-Stream Signals**

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 as a guide to 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 family, visit www.xilinx.com/support.

To see the available documentation by solution:

1. Visit 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 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. Log in 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.

Simulation Debug

Lanes and Channel do not come up in simulation

- The most effective method of debugging this condition is to view the signals from one instance of the serial transceivers that is 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 not, it might take longer for the PMA to finish locking. Wait for lane up and channel up. It might take even longer for simplex designs.
- Make sure that `txn` and `txp` are toggling. If not, make sure to wait long enough and ensure that the TX signal is not being driven by another signal.
- Check the `pll_not_locked` signal in the design. If it is held active-High, the Aurora module is unable to initialize.
- Be sure the `power_down` signal is not asserted.
- If using Verilog simulation, instantiate the `glbl` module and use it to drive the `power_up` reset at the beginning of the simulation. This procedure simulates the reset that occurs after configuration. 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 using a multilane channel, make sure all of the serial transceivers on each side of the channel are connected in the correct order.

Channel comes up in simulation but s_axi_tx_tvalid is never asserted (never goes High)

- If the module includes flow control but is not being used, 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` stays Low because the channel is being 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 NFC is enabled, make sure the channel partner 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 using the AXI4-Stream interface, make sure the data is written correctly. The most common mistake is to assume words are written without monitoring `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 to read correctly from the RX interface. Data and framing signals are only valid when `tvalid` is asserted.

Problems while compiling the design

- Make sure to include all the files from the src directory when compiling
- If using VHDL, make sure to include the `aurora_pkg.vhd` file in synthesis
- Make sure the simulator and libraries are set up correctly
- Make sure the simulator language is set to *mixed*

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

Most Vivado IDE fields have tool tips which serve as guidelines to configure and generate the core properly. Refer to and follow all RECOMMENDED and IMPORTANT notes in the product guide.

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.

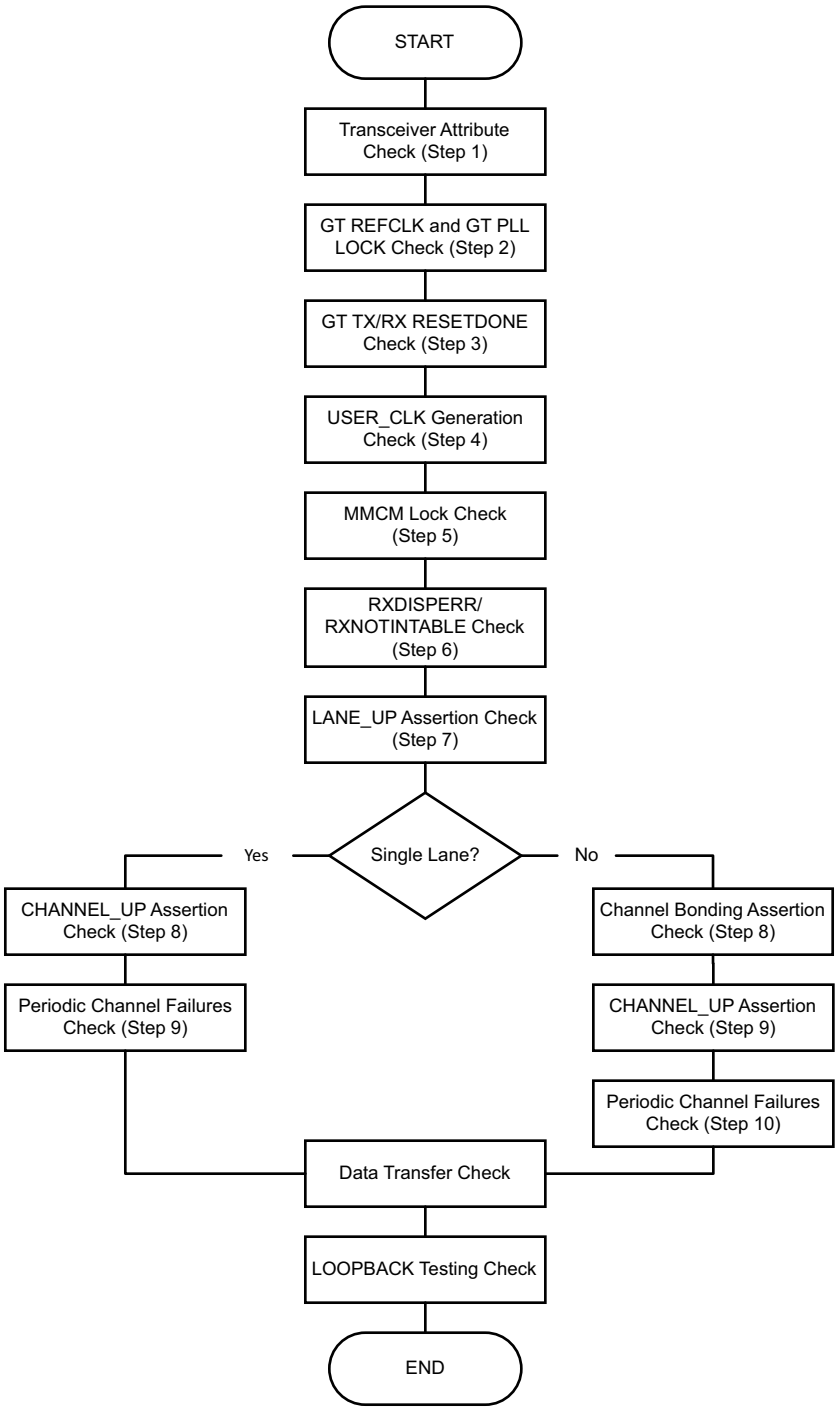


Figure C-1: **Flow Chart**

STEP 1: Transceiver Debug

With the transceiver being the critical building block in the Aurora 8B/10B core, debugging and ensuring proper transceiver operation is very important.

1. Transceiver attribute check:

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

2. GT REFCLK and GT PLL LOCK check

   A low-jitter differential clock must be provided to the transceiver reference clock. Check and make sure the REFCLK location constraints are correct with respect to the board schematics. REFCLK should be active and should meet the phase noise requirements of the transceiver.

   The transceiver locks on to the incoming GT REFCLK signal and asserts the `PLL0LOCK` signal. If `PLL0LOCK` is toggling periodically, check that the FSM reset done signals are toggling. Make sure that GT PLL attributes are set correctly and that the transceiver generates the `txoutclk` with the expected frequency for the given line rate and datapath width options. Note that the Aurora 8B/10B 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. Check the transceiver power supply `MGTAVCC` value.

3. Transceiver TX/RX FSM RESETDONE check

   The Aurora 8B/10B core uses sequential reset mode; all of the transceiver components are reset sequentially, one after another. The `txresetdone` and `rxresetdone` signals should be asserted at the end of the transceiver initialization. In general, `rxresetdone` assertion takes longer compared to the TXRESETDONE assertion. Check if `user_clk` and `sync_clk` are connected properly. Make sure the `gt_reset` signal pulse width duration complies with the respective transceiver guideline. Probe the signals and FSM states from the RX/TX STARTUP FSM module. If the RX/TX `fsm_resetdone` signals are asserted and the partner is reprogrammed, `GTRXRESET` should be asserted manually if hot-plug logic is disabled.

STEP 2: USER_CLK Generation Check

The transceiver generates `txoutclk` based on the line rate and lane-width parameters. The `user_clk` signal is generated from `txoutclk` and is used by the Aurora 8B/10B core to clock FPGA logic. Therefore, ensure that `user_clk` is generated properly with the expected frequency from `txoutclk`. If `user_clk` frequency is not in the expected range, check the frequency of the transceiver reference clock and verify the transceiver PLL attributes.

STEP 3: MMCM Lock Check

The Aurora 8B/10B core expects all clocks to be stable. If clocks are generated using the MMCM, ensure that the reset inputs are held High until the generated clock is stable. It is recommended to stop the output clock from the MMCM until it is locked. For cores generated with a 4-byte lane width in Artix devices, the MMCM is used to generate `user_clk` and `sync_clk`. Make sure that the `TX_LOCK` output from the Aurora 8B/10B core is inverted and connected to `MMCM_RESET`. If `MMCM_LOCK` is toggling periodically, check that the TX_STARTUP_FSM module is restarting and probe the signals and states of the FSM.

STEP 4: RXDISPERR/RXNOTINTABLE Check

The Aurora 8B/10B core defines `RXDISPERR` and `RXNOTINTABLE` using the `soft_error` signal. If the core asserts the `soft_error` signal, probe the `RXDISPERR` and `RXNOTINTABLE` ports of the transceiver. If the transceiver indicates a `RXDISPERR` or `RXNOTINTABLE` error, enable internal loopback and check again. If the loopback test passes, check the transmitted data and cable for channel integrity. Run IBERT to confirm the link connectivity and SI on the channel. If the IBERT runs fail, monitor the power supplies, check the termination circuit, run SI simulations, check LPM versus DFE based on attenuation, etc. Also, enabling the scrambler option in the core is useful to check EMI issues generated over the link.

STEP 5: LANE_UP Assertion Check

The `lane_up` assertion indicates the communication between the transceiver and its channel partner is established and link training is successful. Make sure the `do_cc` signal from the standard_CC block is connected properly. The LANE_INIT_SM module FSM state signals must be brought to debug if `lane_up` is not asserted. For a simplex-timer core, check and follow the reset sequence requirement. If the TX transceiver needs to be reset as per the system design, increase the C_ALIGNED_TIMER, C_BONDED_TIMER, and C_VERIFY_TIMER attributes based on the latency between the release of `TX_RESET` and `RX_RESET`. See the Lane Initialization Procedure in the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 4] for `lane_up` assertion.

STEP 6: CHANNEL_UP Assertion Check

The criteria for `channel_up` assertion verification are:

- The sequence defined in the Aurora 8B/10B protocol being transferred between channel partners
- Successful reception of four verification sequences

Enable loopback mode and check for `lane_up` assertions. The CHANNEL_INIT_SM module FSM state signals must be brought to debug if `channel_up` is not asserted. For a simplex link, the simplex TX transceiver might have already achieved `channel_up` status. If the TX transceiver needs to be reset as per the system design, increase the SIMPLEX_TIMER_VALUE attribute based on the latency between the release of `TX_RESET` and `RX_RESET`. See the Channel Verification Procedure in the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 4] for `channel_up` assertion.

STEP 6A: Channel Bonding Assertion Check

Channel bonding is necessary for a multi-lane Aurora design. Channel bonding is performed by the transceiver and the required logic is present in the transceiver_wrapper module. Make sure that channel bonding level, master and slave connections are correct. Check that the CLK_COR_MIN_LAT and CLK_COR_MAX_LAT attributes of the transceiver are set as recommended. See the Channel Bonding Procedure in the *Aurora 8B/10B Protocol Specification v2.2* (SP002) [Ref 4] for `channel_up` assertion.

STEP 6B: CHANNEL_UP Assertion Check

This step is the same as STEP 6 described previously.

STEP 7: Periodic Channel Failures Check

If the Aurora 8B/10B core asserts and deasserts the `channel_up` signal, enable internal loopback and check for a stable channel up condition. Probe RXBUFSTATUS of the transceiver. If there is overflow or underflow, CLK_COR_MIN_LAT and CLK_COR_MAX_LAT attribute values for the transceiver must be adjusted. Also make sure the hot-plug logic is disable when the standard_cc block is not used.

STEP 8: Data Transfer Check

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

STEP 9: LOOPBACK Configuration Testing

Loopback modes are specialized configurations of the transceiver datapath. The Aurora 8B/10B example design loopback port controls the loopback modes. Four loopback modes are available. Refer to the respective transceiver user guide for more information. Figure C-2 illustrates a loopback test configuration with four different loopback modes.

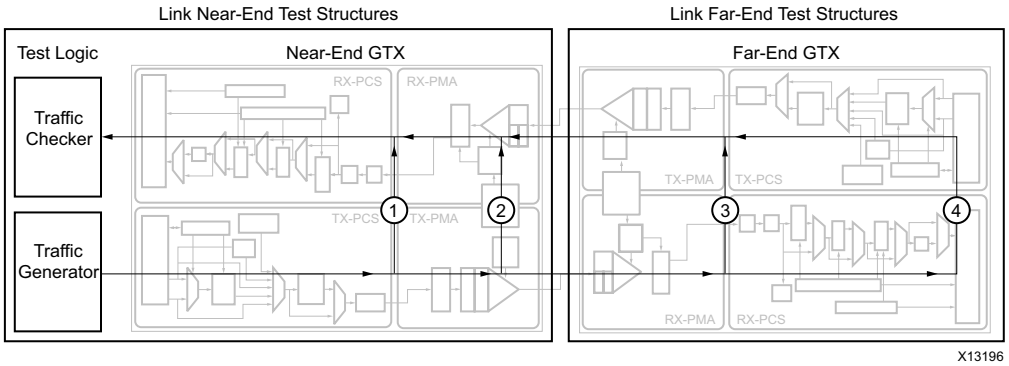


Figure C-2: **Loopback Testing Overview**

Channel comes up in simulation but not in hardware

- Both `reset` and `gt_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 8B/10B 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.
- A burst of soft errors results in a hard error and re-initializes the channel. Set ENABLE_SOFT_ERR_MONITOR to 0 in the `<component name>_err_detect` module to disable hard error assertion from soft errors.

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 2-8, page 19 for data transfer and Figure 2-12, page 22 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 the transceiver wrapper file is updated 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.

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

1. Using the Vivado IP catalog, run the latest version of the 7 Series FPGAs Transceivers Wizard. Make sure the Component Name of the transceiver wizard matches the Component Name of the Aurora 8B/10B core.
2. Select the protocol template from the following based on the number of lane(s) and lane width:
   - Aurora 8B/10B single lane 2 byte
   - Aurora 8B/10B single lane 4 byte
   - Aurora 8B/10B multi lane 2 byte
   - Aurora 8B/10B multi lane 4 byte
3. Change the Line Rate in both TX and RX based on the application requirement.
4. Select the Reference Clock from the drop-down 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 the default for all other settings.
7. Generate the core.
8. Replace the `<component name>_gt.v[hd]` and `<component name>_multi_gt.v[hd]` files in the `gt` directory available in the Aurora 8B/10B core with the generated `<component name>_gt.v[hd]` and `<component name>_multi_gt.v[hd]` files generated from the 7 series FPGAs transceivers wizard.

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

Note: The UltraScale™ architecture Aurora 8B/10B core uses the hierarchical core calling method to call the UltraScale device gtwizard IP core. In this way, all the transceiver attributes, parameters, and required workarounds are in place and correct. Manual editing of the UltraScale device transceiver files are not required in most of the cases. The attribute(s) in the Aurora 8B/10B core example design XDC file can be updated.

Handling Timing Errors

This appendix describes how to handle timing errors resulting from transceivers that are located far apart from each other. The Aurora 8B/10B core allows selecting 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 7] for instructions on how to use the Strategies options.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

For a glossary of technical terms used in Xilinx documentation, see the Xilinx Glossary.

References

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. *UltraScale Architecture GTH Transceivers User Guide* (UG576)
2. *7 Series FPGAs GTP Transceivers User Guide* (UG482)
3. *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476)
4. *Aurora 8B/10B Protocol Specification* (SP002)
5. *AMBA® AXI4-Stream Protocol Specification* (v1.0)
6. *Vivado® Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
7. *Vivado Design Suite User Guide: Designing with IP* (UG896)
8. *Vivado Design Suite User Guide: Getting Started* (UG910)
9. *Vivado Design Suite User Guide - Logic Simulation* (UG900)
10. *ISE® to Vivado Design Suite Migration Guide* (UG911)
11. *LogiCORE™ IP Aurora 8B/10B v5.3 Data Sheet* (DS637)
12. *LogiCORE IP Aurora 8B/10B v5.3 User Guide* (UG353)
13. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
14. *Designing a System Using the Aurora 8B10B Core (Duplex) on the KC705 Evaluation Kit* (XAPP1193)
15. *UltraScale Architecture Migration Methodology Guide* (UG1026)

16. *Packaging Custom AXI IP for Vivado IP Integrator Application Note* (XAPP1168)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/01/2014	10.3	• Added new v10.3 core features and attributes • Rearranged content to consolidate topics and better conform to template
06/06/2014	10.2	• Added information about migrating transceiver ports to UltraScale devices.
06/04/2014	10.2	• Added User Parameter information. • Fixed gt0_dmonitorout_out port width for GTX devices in transceiver debug ports
04/02/2014	10.2	• Added UltraScale architecture support. • Updated `init_clk` frequency requirements. • Added Little Endian support to User Data, NFC and UFC interfaces.
12/18/2013	10.1	• Added transceiver debug ports. • Updated all screen captures. • Updated all signals in figures to lowercase.
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 uppercase signal names to lowercase. • Updated Migrating and Upgrading appendix.
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.
03/20/2013	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
12/18/2012	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
10/16/2012	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.
07/25/2012	1.0	Initial Xilinx release. This release supports core version 8.2 with Vivado® Design Suite v2012.2. This document replaces UG766 and DS797.

Please Read: Important Legal Notices