Aurora 64B/66B v9.3

LogiCORE IP Product Guide

Vivado Design Suite

PG074 October 1, 2014

XILINX
ALL PROGRAMMABLE™

Table of Contents

Introduction

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

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

Features

- Aurora 64B/66B cores supported on the Vivado Design Suite
- General-purpose data channels with throughput range from 500 Mb/s to over 200 Gb/s
- Supports up to any consecutive 16 bonded GTX transceivers or 16 bonded Virtex®-7 FPGA GTH transceivers and 16 bonded UltraScale™ device GTH transceivers.
- Aurora 64B/66B protocol specification v1.3 compliant (64B/66B encoding)
- Low resource cost with very low (3%) transmission overhead
- Easy-to-use AXI4-Stream (framing) or streaming interface and optional flow control
- Automatically initializes and maintains the channel
- Full-duplex or simplex operation
- 32-bit Cyclic Redundancy Check (CRC) for user data
- Supports RX polarity inversion
- Big Endian/Little Endian AXI4-Stream user interface
- Fully compliant AXI4-Lite DRP interface
- Configurable DRP, INIT clock

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family[1]	UltraScale architecture, Zynq®-7000 All Programmable SoC, Virtex-7[2], Kintex®-7[2]
Supported User Interfaces	AXI4-Stream
Resources[3]	See Table 2-2 and Table 2-3.
Provided with Core	
Design Files	RTL
Example Design	Verilog[4]
Test Bench	Verilog
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	

Notes:

1. For a complete list of supported devices, see the Vivado IP catalog.
2. For more information, see *7 Series FPGAs Overview* (DS180) [Ref 1]. and *UltraScale Architecture and Product Overview* (DS890) [Ref 2]
3. For more complete performance data, see Performance, page 9.
4. The core is delivered as open-source code and supports Verilog design environments. Each core comes with an example design and supporting modules.
5. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

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

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

The LogiCORE IP Aurora 64B/66B core supports the AMBA® protocol AXI4-Stream user interface. It implements the Aurora 64B/66B protocol using the high-speed serial GTX or GTH transceivers in applicable UltraScale™, Zynq®-7000, Virtex®-7 and Kintex®-7 devices. The core can use up to 16 consecutive device GTX or GTH transceivers running at any supported line rate to provide a low-cost, general-purpose, data channel with throughput from 500 Mb/s to over 200 Gb/s.

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

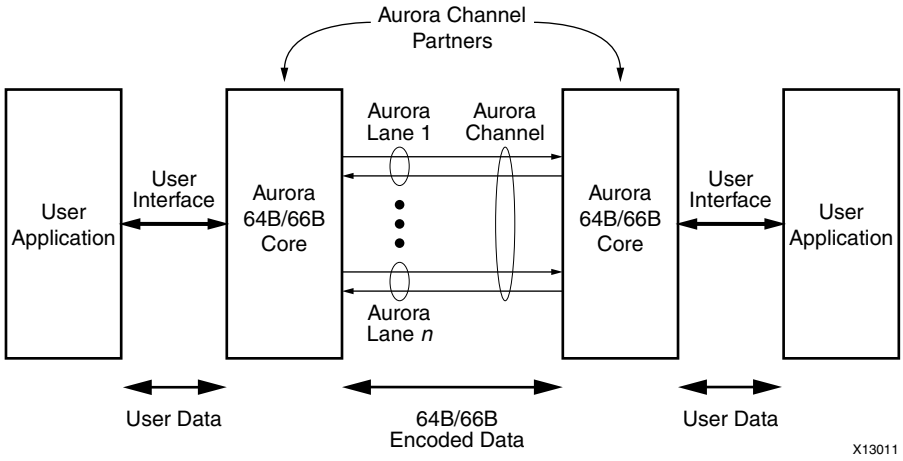


Figure 1-1: **Aurora 64B/66B Channel Overview**

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

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

RECOMMENDED:

1. *Although the Aurora 64B/66B core is a fully-verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. For best results, prior experience in building high-performance, pipelined FPGA designs using Xilinx implementation tools and Xilinx® Design Constraints (XDC) user constraints files is recommended.*
2. *Consult the PCB design requirements information in the UltraScale FPGAs GTH Transceivers User Guide (UG576) [Ref 3] and 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 4]. Contact your local Xilinx representative for a closer review and estimation for your specific requirements.*

Applications

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

- **Chip-to-chip links:** Replacing parallel connections between chips with high-speed serial connections can significantly reduce the number of traces and layers required on a PCB.
- **Board-to-board and backplane links:** Aurora 64B/66B uses standard 64B/66B encoding, which is the preferred encoding scheme for 10-Gigabit Ethernet making it compatible with many existing hardware standards for cables and backplanes. Aurora 64B/66B can be scaled, both in line rate and channel width, to allow inexpensive legacy hardware to be used in new, high-performance systems.

- **Simplex connections (unidirectional):** The Aurora 64B/66B simplex protocol provides unidirectional channel initialization, making it possible to use the GTX and GTH transceivers when a back channel is not available, and to reduce costs due to unused full-duplex resources.
- **ASIC applications:** Aurora 64B/66B is not limited to FPGAs, and can be used to create scalable, high-performance links between programmable logic and high-performance ASICs. The simplicity of the Aurora 64B/66B protocol leads to low resource costs in ASICs as well as in FPGAs, and design resources like the Aurora 64B/66B bus functional model (BFM) with automated compliance testing make it easy to get an Aurora 64B/66B connection up and running. Contact Xilinx Sales or auroramkt@xilinx.com for information on licensing Aurora for ASIC applications.

Unsupported Features

AXI4-Stream non-strict aligned mode is not supported in the Aurora 64B/66B core.

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

For more information, visit the Aurora 64B/66B product page.

Product Specification

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

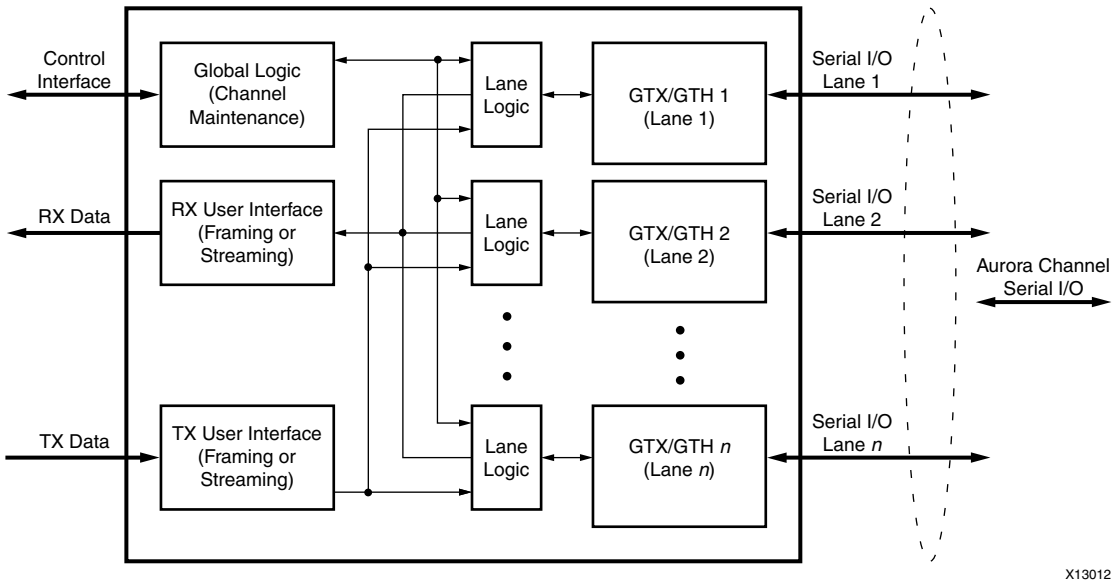


Figure 2-1: **Aurora 64B/66B Core Block Diagram**

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

- **Lane logic:** Each GTX and GTH transceiver is driven by an instance of the lane logic module which initializes each individual transceiver, handles the encoding and decoding of control characters, and performs error detection.
- **Global logic:** The global logic module in the core performs the channel bonding for channel initialization. During operation, the channel keeps track of the Not Ready idle characters defined by the Aurora 64B/66B protocol and monitors all the lane logic modules for errors.
- **RX user interface:** The AXI4-Stream receive (RX) user interface moves data from the channel to the application and also performs flow control functions.

- **TX user interface:** The AXI4-Stream transmit (TX) user interface moves data from the application to the channel and also performs flow control TX functions. The module has an interface for controlling clock compensation (the periodic transmission of special characters to prevent errors due to small clock frequency differences between channel partners. This interface can be driven by a standard clock compensation manager module provided with the core, or driven by custom logic.

Performance

This section details the performance information for various core configurations.

Maximum Frequencies

The maximum frequency of the core operation is dependent on the line rates supported and the speed grade of the devices.

Latency

For a default single lane configuration, latency through an Aurora 64B/66B core is caused by pipeline delays through the protocol engine (PE) and through the GTX and GTH transceivers. The PE pipeline delay increases as the AXI4-Stream interface width increases. The transceiver delays are determined by the transceiver features.

This section outlines a method of measuring the latency for the Aurora 64B/66B core AXI4-Stream user interface in terms of `user_clk` cycles for UltraScale™, Zynq®-7000, Virtex®-7, and Kintex®-7 device GTX and GTH transceiver-based designs. For the purposes of illustrating latency, the Aurora 64B/66B modules are partitioned between logic in the GTX and GTH transceivers and protocol engine (PE) logic implemented in the FPGA.

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

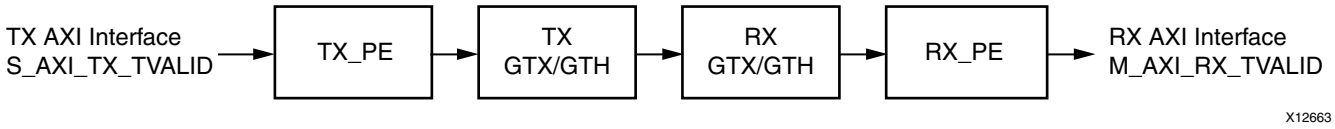


Figure 2-2: **Latency of the Frame Path**

Note: Figure 2-2 does not include the latency incurred due to the length of the serial connection between each side of the Aurora 64B/66B channel.

The latency must be measured from the rising edge of the transmitter `user_clk` at the first assertion of `s_axi_tx_tvalid` and `s_axi_tx_tready` to the rising edge of the receiver `user_clk` at the first assertion of `m_axi_rx_tvalid`. Figure 2-3 shows the transmitter and receiver path reference points between which the latency has been measured for the default core configuration.

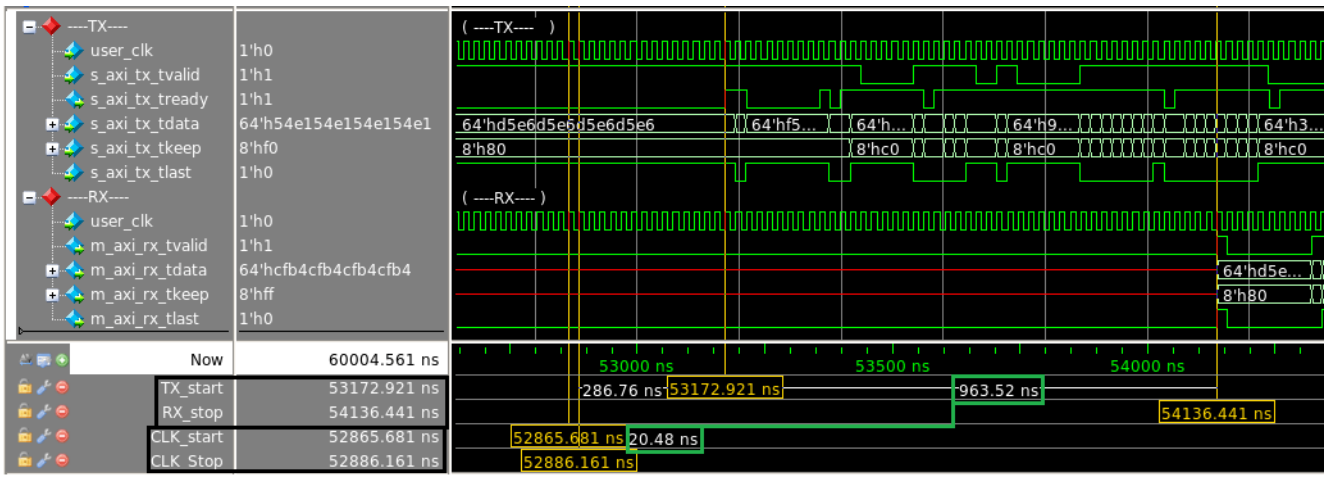


Figure 2-3: **Latency Waveform with Reference Points**

Table 2-1 shows the maximum latency and the individual latency values of the contributing pipeline components for the default core configuration. Latency can vary with the addition of flow controls.

Table 2-1: **Latency for the Default Aurora 64B/66B Core Configuration**

Latency Component	user_clk Cycles
Logic	46
Gearbox	1 or 2
Clock Compensation	7
Maximum (total)	54 or 55

The pipeline delays are designed to maintain the clock speed.

Throughput

Aurora 64B/66B core throughput depends on the number of the transceivers and the target line rate of the transceivers selected. Throughput varies from 0.48 Gb/s to 203.3 Gb/s for a single-lane design to a 16-lane design, respectively. The throughput was calculated using 3% overhead of Aurora 64B/66B protocol encoding and 0.5 Gb/s to 13.1 Gb/s line rate range.

Resource Utilization

Table 2-2 through Table 2-3 show the number of look-up tables (LUTs) and flip-flops (FFs) used in selected Aurora 64B/66B *framing* and *streaming* modules in the Vivado® Design Suite implemented on the xc7vx485tffg1157-1 device. The Aurora 64B/66B core is also available in configurations not shown in the tables. The tables do not include the additional resource usage for flow controls. Resource utilization in the following tables do not include the additional resource usage for the example design modules, such as FRAME_GEN and FRAME_CHECK. Values provided are exact values for a given configuration. Values in the following tables are for the default configuration (3.125G) with support logic included.

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

Virtex-7 Family (GTX Transceiver)		Streaming		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
1	LUTs	549	315	377
	FFs	1359	476	957
2	LUTs	1044	467	686
	FFs	2379	761	1721
4	LUTs	1971	711	1452
	FFs	4347	1380	3139
8	LUTs	3610	1256	2805
	FFs	8219	2539	5973
16	LUTs	6656	1949	5496
	FFs	15966	4825	11641

Table 2-3: **Virtex-7 Family GTX Transceiver Resource Usage for Framing**

Virtex-7 Family (GTX Transceiver)		Framing		
		Duplex	Simplex	
Lanes	Resource Type	Full-Duplex	TX-Only	RX-Only
1	LUTs	873	315	597
	FFs	1398	499	975
2	LUTs	1475	471	1106
	FFs	2442	799	1748
4	LUTs	2628	764	2012
	FFs	4444	1425	3182
8	LUTs	4997	1566	3896
	FFs	8391	2623	6046
16	LUTs	9418	2874	7560
	FFs	16273	5018	11771

Note: UltraScale device utilization results are expected to be similar to 7 series devices.

Port Descriptions

The parameters used to generate each Aurora 64B/66B core determine the interfaces available for that specific core. Aurora 64B/66B cores have four to eight interfaces; the ports of each interface are shown in a figure and listed in a table followed by a brief description of the interface:

- User Interface, page 14
- Clock Interface, page 24
- Native Flow Control Interface, page 28
- User Flow Control Interface, page 31
- User K-Block Interface, page 35
- Status, Control, and the Transceiver Interface, page 38
- CRC Interface, page 53

Note:
1. [*n*:0] bus format is used when the Little Endian support option is selected. [0:*n*] bus format is used when the Big Endian support option is selected. The core has an option to configure the AXI4-Stream User I/O as little endian from the Vivado IDE. The default is big endian.
2. Ports are active-High unless specified otherwise.

Figure 2-4 shows the Aurora 64B/66B IP symbol for the default core configuration with all flow control options and CRC enabled.

Figure 2-4: **Aurora 64B/66B IP Symbol**

User Interface

The Aurora 64B/66B core can be generated with either a framing or streaming user data interface. Data port width depends on the number of lanes selected. Table 2-4 lists simplex/duplex port descriptions for the AXI4-Stream TX data ports.

Table 2-4: **User Interface Ports**

Name	Direction	Clock Domain	Description
USER_DATA_S_AXIS_TX			
s_axi_tx_tdata[0:(64*n*–1)] or s_axi_tx_tdata[(64*n*–1):0](1)	Input	user_clk	Outgoing data (ascending bit order).
s_axi_tx_tready	Output	user_clk	Asserted when signals from the source are accepted. Deasserted when signals from the source are ignored.
s_axi_tx_tvalid	Input	user_clk	Asserted when AXI4-Stream signals from the source are valid. Deasserted when AXI4-Stream control signals and/or data from the source should be ignored.
s_axi_tx_tlast	Input	user_clk	Indicates the end of the frame. *This port is not available if the Streaming interface option is chosen*.
s_axi_tx_tkeep[0:(8*n*–1)] or s_axi_tx_tkeep[(8*n*–1):0](1)	Input	user_clk	Specifies the number of valid bytes in the last data beat (number of valid bytes = number of 1s in `tkeep`). `s_axi_tx_tkeep` is sampled only when `s_axi_tx_tlast` is asserted. The core supports continuous aligned and continuous unaligned data streams and expects data to be filled continuously from LSB to MSB. There cannot be invalid bytes interleaved with the valid `s_axi_tx_tdata` bus. *This port is not available if the Streaming interface option is chosen*.

Table 2-4: **User Interface Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
USER_DATA_M_AXIS_RX			
m_axi_rx_tdata[0:(64*n*–1)] or m_axi_rx_tdata[(64*n*–1):0](1)	Output	user_clk	Incoming data from channel partner (ascending bit order).
m_axi_rx_tvalid	Output	user_clk	Asserted when data from core is valid. Deasserted when data from core should be ignored.
m_axi_rx_tlast	Output	user_clk	Indicates the end of the incoming frame. *This port is not available if the Streaming interface option is chosen.*
m_axi_rx_tkeep[0:(8*n*–1)] or m_axi_rx_tkeep[(8*n*–1):0](1)	Output	user_clk	Specifies the number of valid bytes in the last data beat. *This port is not available if the Streaming interface option is chosen.*

Notes:
1. *n* is the number of lanes.

Top-Level Interface

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

Figure 2-5 shows the Aurora 64B/66B top level for a duplex configuration.

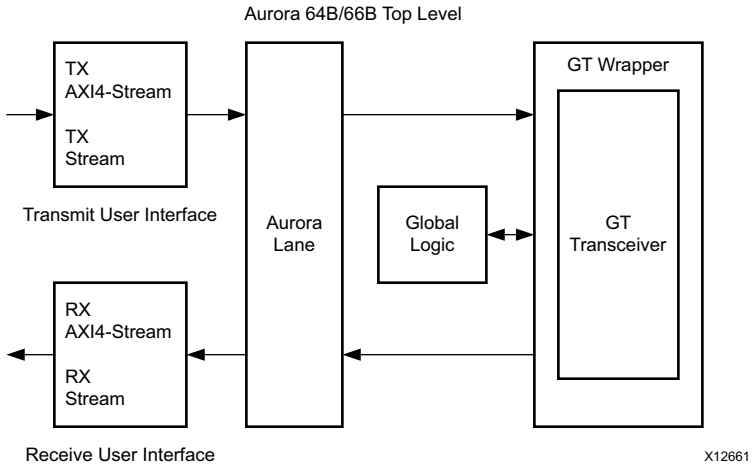


Figure 2-5: **Aurora 64B/66B Duplex Top-Level Architecture**

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

Figure 2-6 shows an *n*-byte example of the Aurora 64B/66B AXI4-Stream data interface bit ordering.

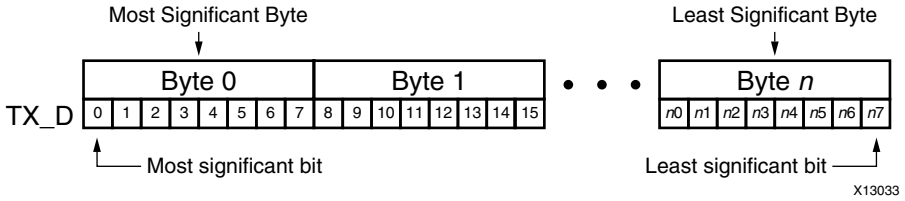


Figure 2-6: **AXI4-Stream Interface Bit Ordering**

Framing Interface

The framing user interface (Figure 2-7) complies with the *AXI4-Stream Protocol Specification* [Ref 22] and comprises the signals necessary for transmitting and receiving framed user data. A detailed description of the framing interface follows.

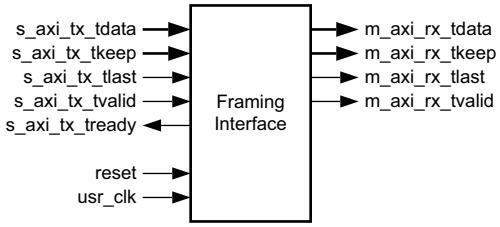


Figure 2-7: **Aurora 64B/66B Framing User Interface (AXI4-Stream)**

Transmitting Data

The Aurora 64B/66B core samples the data only if both `s_axi_tx_tready` and `s_axi_tx_tvalid` are asserted. The user application can deassert `s_axi_tx_tvalid` on any clock cycle (Figure 2-9, page 19) to ignore the AXI4-Stream input for that cycle. If this occurs in the middle of a frame, idle symbols are sent through the Aurora channel.

The AXI4-Stream data is only valid when it is framed. Data outside of a frame is ignored. To end a frame, assert `s_axi_tx_tlast` while the last word (or partial word) of data is on the `s_axi_tx_tdata` port.

Data Strobe

In framing, the AXI4-Stream interface allows the last word of a frame to be a partial word (8*n* bits where *n* > 1). If `TKEEP` is `0x0F` in the last beat of data, then byte 0 to byte 3 of 8 bytes are valid. All 1s in the `s_axi_tx_tkeep` value indicate that all bytes in the `s_axi_tx_tdata` port are valid. `s_axi_tx_tkeep` does not specify the position of the valid bytes, but is the number of valid bytes on the last beat of data with `s_axi_tx_tlast` asserted. The core expects `TKEEP` to be left aligned from the LSB. If the CRC option is selected, CRC is calculated and inserted into the data stream after the last data word. See Appendix B, Migrating and Upgrading for limitations on the types of data stream supported by the core.

Note that high priority is assigned to these requests for any type of transfer:

- TXDATAVALID deasserted from the transceiver TX interface (1 cycle)
- CC transmission (8 cycles)

Aurora 64B/66B Frames

All Aurora 64B/66B data is sent as part of a data block or a separator block. A separator block consists of a count field indicating how many bytes are valid in that particular block. In framing, each frame begins with data blocks and ends with a separator block containing the last bytes of the frame. Idle blocks are inserted whenever data is not available. Blocks are eight bytes of scrambled data or control information with a two-bit control header (a total of 66 bits).

Table 2-5 shows a typical Aurora 64B/66B frame with an even number of data bytes.

Table 2-5: **Typical Channel Frame**

Data Byte 0	Data Byte 1	Data Byte 2	Data Byte 3	. . .	Data Byte *n*–2	Data Byte *n*–1	Data Byte *n*
SEP (1E)	Count (4)	Data Byte 0	Data Byte 1	Data Byte 2	Data Byte 3	x	x

To transmit data, the user application manipulates the control signals causing the core to perform these steps:

1. Accept data from the user application on the `s_axi_tx_tdata` bus.
2. Indicate end of frame when `s_axi_tx_tlast` is asserted and stripe data across lanes in the Aurora Channel.
3. Insert idle or pause cycles on the serial line when the user application deasserts `s_axi_tx_tvalid`.

When the core receives data, it performs these steps:

1. Detects and discards control bytes (idles, clock compensation).

2. Recovers data from the lanes.
3. Assembles data for presentation to the user application on the `m_axi_rx_tdata` bus including providing the number of valid bytes on `m_axi_rx_tkeep` and asserts `m_axi_rx_tvalid` during the `m_axi_rx_tlast` cycle.

Example A: Simple Data Transfer

Figure 2-8 shows an example of a simple *n* byte wide data transfer. 3*n* bytes of data are sent requiring three data beats. `s_axi_tx_tready` is asserted indicating that the AXI4-Stream interface is ready to transmit data.

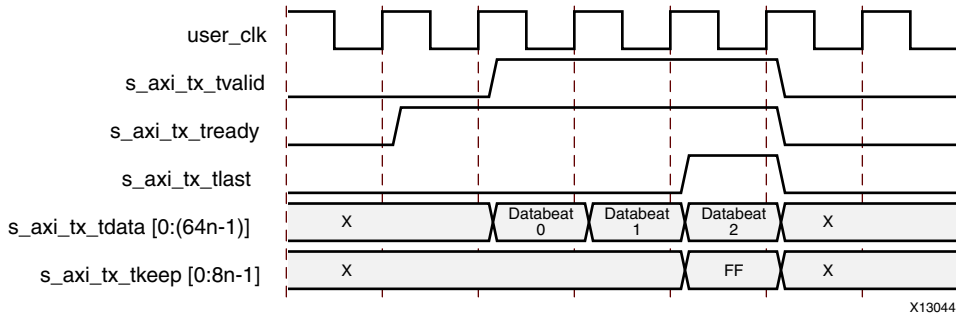


Figure 2-8: **Simple Data Transfer**

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

To end the data transfer, the user application asserts `s_axi_tx_tlast`, `s_axi_tx_tvalid`, the last data bytes, and the appropriate `TKEEP` value (`0xFF`) on the `s_axi_tx_tkeep` bus. The core sends the final data word in blocks, and must send an empty separator block on the next cycle to indicate the end of the frame. `s_axi_tx_tready` is reasserted on the next cycle so that more data transfers can continue. If there is no new data, the Aurora 64B/66B core sends idles.

Example B: Data Transfer with Pause

Figure 2-9 shows the user application pausing data transmission during a frame transfer. The application sends 3*n* bytes of data and pauses the data flow after the first *n* bytes. After the first data word, the application deasserts `s_axi_tx_tvalid` causing the TX Aurora 64B/66B core to ignore all data on the bus and transmit idle blocks. The pause continues until `s_axi_tx_tvalid` is asserted.

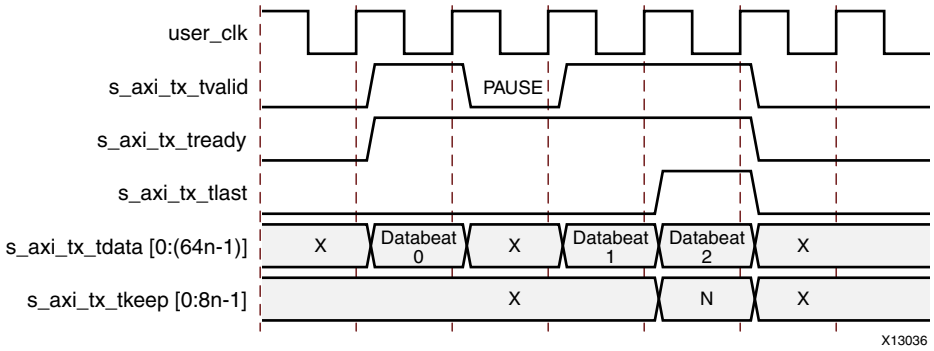
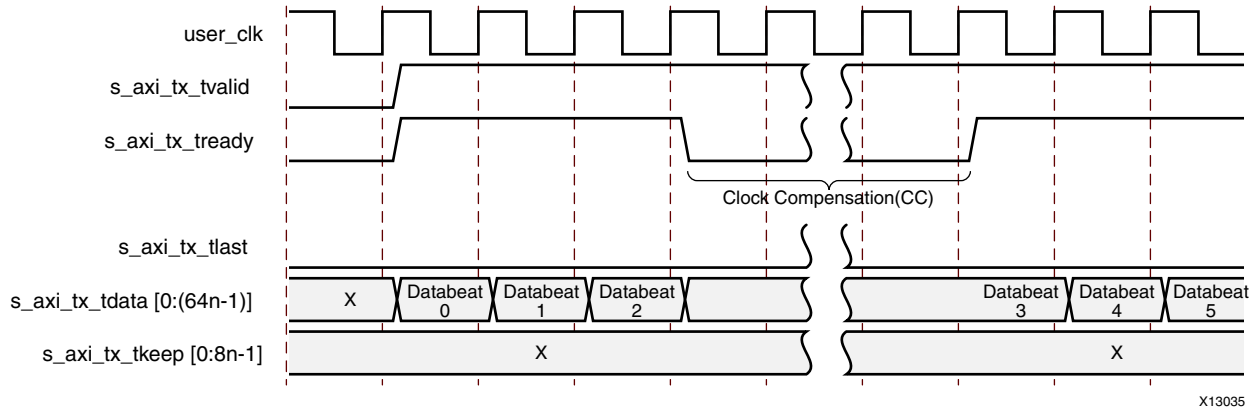


Figure 2-9: **Data Transfer with Pause**

Example C: Data Transfer with Clock Compensation

Figure 2-10 shows the Aurora 64B/66B core automatically interrupting data transmission when clock compensation sequences are sent.



Notes:

1. When clock compensation is used, uninterrupted data transmission is not possible. See Clock Compensation Interface, page 26 for more information about when clock compensation is required.

Figure 2-10: **Data Transfer Paused by Clock Compensation**

Receiving Data

Because the Aurora 64B/66B core has no built-in buffer for user data, there is no m_axi_rx_tready signal on the RX AXI4-Stream interface. User application control of the flow of data from an Aurora channel is limited to one of the optional core flow control features.

The m_axi_rx_tvalid signal is asserted concurrently with the first word of each frame from the Aurora 64B/66B core. The m_axi_rx_tlast signal 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 using the same byte indication procedure as s_axi_tx_tkeep. All bytes valid is indicated (all 1s) when m_axi_rx_tkeep is not asserted and the exact number of bytes valid is specified when m_axi_rx_tkeep is asserted.

If the CRC option is selected, the received data stream is computed for the expected CRC value. The CRC block re-calculates the m_axi_rx_tkeep value and correspondingly asserts m_axi_rx_tlast.

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

Example A: Data Reception with Pause

Figure 2-11 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, the m_axi_rx_tvalid output is asserted to indicate that data is ready for the user application.

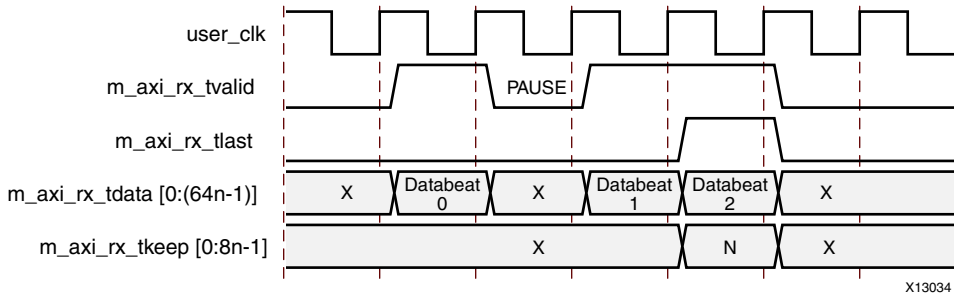


Figure 2-11: **Data Reception with Pause**

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.

Framing Efficiency

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

1. The size of the frame.
2. A data invalid request from the gear box that occurs after every 32 user_clk cycles.

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

The gear box in GTX and GTH transceivers requires a periodic pause to account for the clock divider ratio and 64B/66B encoding. This appears as a back pressure in the AXI4-Stream interface and user data needs to be stopped for one cycle after every 32 cycles (Figure 2-12). The `s_axi_tx_tready` signal in the user interface from the Aurora core is deasserted for one cycle, once every 32 cycles. The pause cycle is used to compensate the gear box for the 64B/66B encoding.

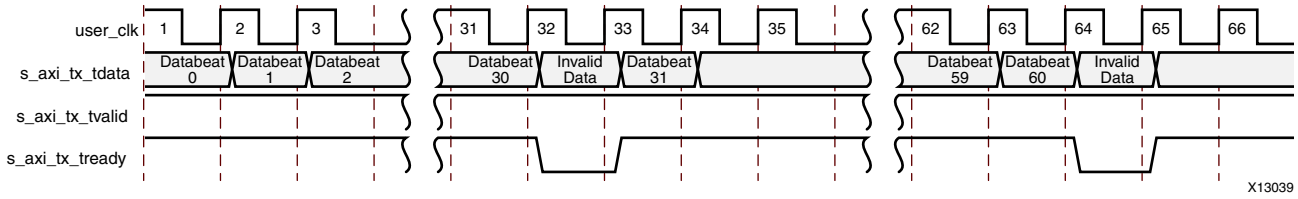


Figure 2-12: **Framing Efficiency**

For more information on gear box pause in GTX and GTH transceivers, see the *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476) [Ref 4] or *UltraScale FPGAs GTH Transceivers User Guide* (UG576) [Ref 3].

The Aurora 64B/66B core implements the Strict Aligned option of the Aurora 64B/66B protocol. No data blocks are placed after idle blocks or SEP blocks on a given cycle. Table 2-6 is an example calculated after including overhead for clock compensation and shows the efficiency for a single-lane channel while illustrating that the efficiency increases as frame length increases.

Table 2-6: **Framing Efficiency Example**

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

Table 2-7 shows the overhead in a single-lane channel when transmitting 256 bytes of frame data. The resulting data unit is 264 bytes long due to the end-of-frame SEP block. This results in a 3.03% transmitter overhead. Also, the clock compensation blocks must be transmitted for three cycles every 10,000 cycles resulting in an additional 0.03% overhead in the transmitter.

Table 2-7: **Typical Overhead for Transmitting 256 Data Bytes**

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

Streaming Interface

The streaming interface (Figure 2-13) allows sending data without frame delimiters while being simple to operate with fewer resources than framing. A detailed description of the streaming interface follows.

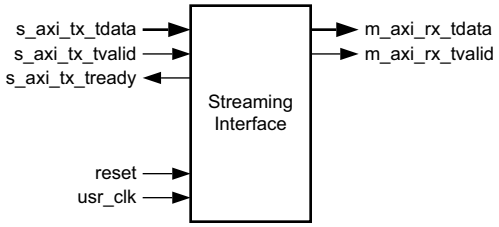


Figure 2-13: **Aurora 64B/66B Streaming User Interface (AXI4-Stream)**

Transmitting and Receiving Data

In streaming, the Aurora channel is used as a pipe. The streaming Aurora interface expects data to be filled for the entire `s_axi_tx_tdata` port width (integral multiple of eight bytes). When `s_axi_tx_tvalid` is deasserted, gaps are created between words which are preserved except when clock compensation sequences are being transmitted.

When data arrives at the RX side of the Aurora channel, it 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 streaming data transfer beginning with neither of the ready signals asserted to indicate that both the user logic and the Aurora 64B/66B core are not ready to transfer data. During the next clock cycle, the core indicates that it is ready to transfer data by asserting `s_axi_tx_tready`. One cycle later, the user logic asserts the `s_axi_tx_tvalid` signal and places data on the `s_axi_tx_tdata` bus indicating that it is ready to transfer data. Because both signals are now asserted, data D0 is transferred from the user logic to the core. Data D1 is transferred on the following clock cycle. In this example, the 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 application deasserts `s_axi_tx_tvalid` on the next clock cycle and no data is transferred until both signals are asserted.

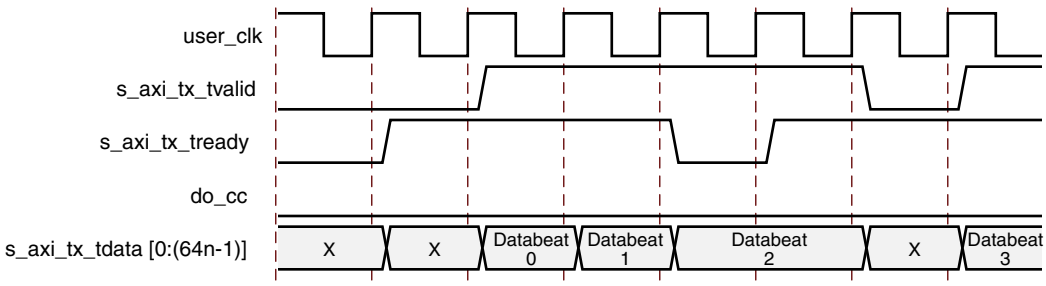


Figure 2-14: **Typical Streaming Data Transfer**

Example B: RX Streaming Data Transfer

Figure 2-15 shows a typical streaming data reception example.

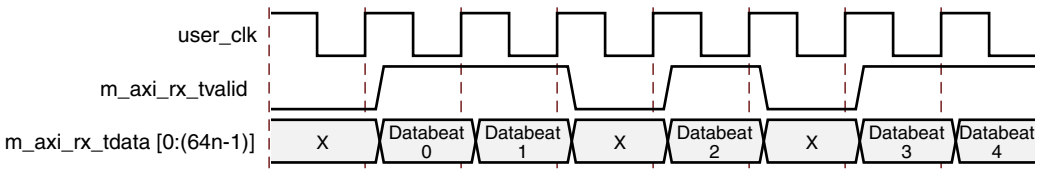


Figure 2-15: **Typical Streaming Data Reception**

Clock Interface

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

Table 2-8 describes the core clock ports. In GTX and GTH transceiver designs, the reference clock can be taken from the GTXQ/GTHQ signal, which is a differential input clock for each GTX or GTH transceiver. The reference clock for GTX/GTH transceivers is provided through the `clkin` port. For more details on the clock interface, see Clocking, page 54.

Table 2-8: **Aurora 64B/66B Core Clock Ports**

Name	Direction	Clock domain	Description
init_clk/init_clk_p/init_clk_n	Input	-	The `init_clk` signal is used to register and debounce the `pma_init` signal. The preferred `init_clk` range is 50 to 200 MHz. The default `init_clk` frequency set by the core is 50 MHz for 7 series designs and line_rate/64 for UltraScale architecture designs. `init_clk` frequency is a user-configurable parameter. With the **include Shared Logic in core** option, the `init_clk` signal is differential. For UltraScale architecture designs: The `init_clk` frequency should be ≤ TXUSERCLK frequency and the value should not exceed 200 MHz. Refer to the *UltraScale Architecture GTH Transceivers User Guide* (UG576) [Ref 3] for more details. `init_clk` is also connected to the DRPCLK port of the GTHE3_CHANNEL DRP interface.
init_clk_out(2)	Output	init_clk	Init clock output.
mmcm_not_locked	Input	init_clk	If MMCM is used to generate clocks for the Aurora 64B/66B core, the `mmcm_not_locked` signal should be connected to the inverse of the serial transceiver PLL locked signal. The clock modules provided with the core use the PLL for clock division. The `mmcm_not_locked` signal from the clock module should be connected to the core `mmcm_not_locked` signal. The `mmcm_not_locked` signal is available when shared logic is included in the example design. For UltraScale devices: `mmcm_not_locked` is connected to `gtwiz_userclk_tx_active_out` driven from the `<user_component_name>_ultrascale_tx_userclk` module. The signal is driven based on the clocking helper core status and signifies that the helper core is out of reset. Active High signal. The `mmcm_not_locked_out` signal is available when shared logic is included in the core.
mmcm_not_locked_out	Output	init_clk	

Table 2-8: **Aurora 64B/66B Core Clock Ports *(Cont'd)***

Name	Direction	Clock domain	Description
user_clk	Input	-	Parallel clock shared by the core and the user application. The `user_clk` signal is a BUFG output deriving its input from `tx_out_clk`. The clock generators are available in the <component name>_clock_module file. `user_clk` serves as the `txusrclk2` input to the transceiver. See the related transceiver user guide/data sheet for rate-related information. `user_clk` is available when shared logic is included in the example design. `user_clk_out` is the user clock output.
user_clk_out(2)	Output	user_clk	
tx_out_clk	Output	tx_out_clk	Generated from the GTX or GTH transceiver reference clock based on the transceiver PLL frequency setting. Should be buffered and used to generate the user clock for the logic connected to the core.
sync_clk	Input	-	Parallel clock used by the serial transceiver internal synchronization logic. Provided as the `txusrclk` signal to the transceiver interface. The `sync_clk` is twice the rate of `user_clk`. See the related transceiver user guide/data sheet for rate-related information. `sync_clk` is available when shared logic is included in the example design. `sync_clk_out` is the sync clock output.
sync_clk_out(2)	Output	sync_clk	
gt_refclk1_p/gt_refclk1_n gt_refclk2_p/gt_refclk2_n refclk1_in refclk2_in	Input	-	`gt_refclk` (`clkp`/`clkn`) is a dedicated external clock generated from an oscillator and fed through a dedicated IBUFDS. • `gt_refclk1_p`/`gt_refclk1_n` = Differential Transceiver Reference Clock 1(2). • `gt_refclk2_p`/`gt_refclk2_n` = Differential Transceiver Reference Clock 2(3). • `refclk1_in` = Single Ended Transceiver Reference Clock 1(4). • `refclk2_in` = Single Ended Transceiver Reference Clock 2(5). • gt_refclk1_out = Single Ended Transceiver Reference Clock 1(2). • `gt_refclk2_out` = Single Ended Transceiver Reference Clock 1(3).
gt_reflck1_out gt_refclk2_out	Output	-	
gt_qpllclk_quad<*quad_no*>_in, gt_qpllrefclk_quad<*quad_no*>_in(1)	Input	-	Clock inputs generated by GTXE2_COMMON/GTHE2_COMMON/GTHE3_COMMON.

Table 2-8: **Aurora 64B/66B Core Clock Ports *(Cont'd)***

Name	Direction	Clock domain	Description
gt_qpllclk_quad<*quad_no*>_out, gt_qpllrefclk_quad<*quad_no*>_out(1)	Output	-	Clock outputs generated by GTXE2_COMMON/GTHE2_COMMON/GTHE3_COMMON. If the line rate is < 6.6 Gb/s in the GTX transceivers and < 8.0 Gb/s in 7 series and UltraScale FPGAs GTH transceivers, the `gt_qpllclk_quad<quad_no>_out` signal is tied High.

Notes:

1. In 7 series, *quad_no* varies from 1 to the number of active transceiver quads –1. In UltraScale FPGAs, varies from 1 to the number of active transceiver quads.
2. Enabled when **include Shared Logic in Core** is selected.
3. Enabled when **include Shared Logic in Core** is selected and more than one reference clock is required.
4. Enabled when **include Shared Logic in Example Design** is selected.
5. Enabled when **include Shared Logic in Example Design** is selected and more than one reference clock is required.

Clock Compensation Interface

The Clock Compensation (CC) interface (Figure 2-16) is included in modules that transmit data. CC characters are assigned highest priority on the channel. Each Aurora 64B/66B core is accompanied by a clock compensation management module that is used to drive the clock compensation interface in accordance with the *Aurora 64B/66B Protocol Specification* (SP011) [Ref 5].

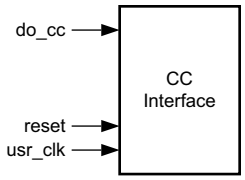


Figure 2-16: **CC Port Interface**

It is highly recommended to enable CC logic for reliable link operation. The clock compensation feature allows up to ±100 ppm difference in the reference clock frequencies used on each side of an Aurora channel. The CC action is triggered using the `do_cc` signal (Table 2-9). This feature must be used in systems where a separate reference clock source is used for each device connected by the channel.

Table 2-9: **Clock Compensation Interface Port**

Name	Direction	Clock Domain	Description
do_cc	Input	user_clk	When asserted, the core stops the flow of data and flow control messages. The core sends the CC sequences on all lanes for every clock cycle.

The clock compensation interface enables full control over the core clock compensation features. Figure 2-17 and Figure 2-18 are waveform diagrams illustrating clock compensation.

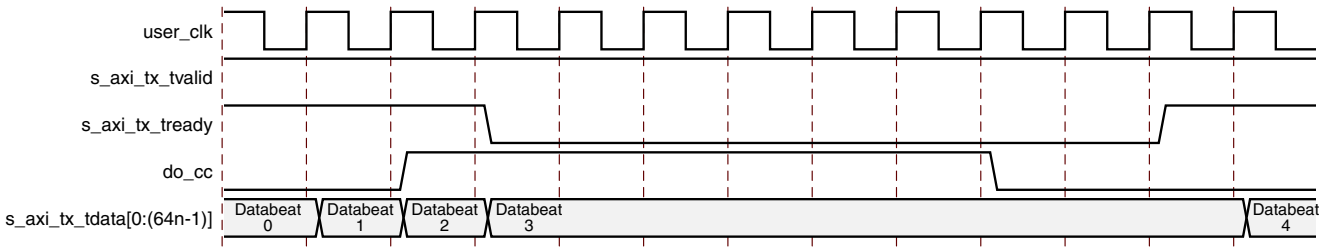


Figure 2-17: **Streaming Data with Clock Compensation Inserted**

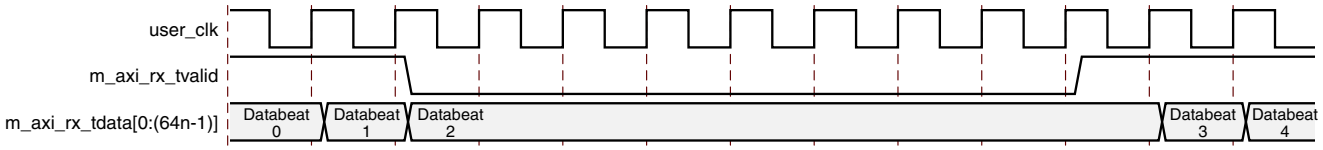


Figure 2-18: **Data Reception Interrupted by Clock Compensation**

To perform Aurora-compliant clock compensation, the `do_cc` signal must be asserted for eight user_clk cycles every 5,000 cycles. `s_axi_tx_tready` is deasserted on the TX user interface while the channel is being used to transmit clock compensation sequences.

Note: For Vivado IP Integrator (IPI), The Aurora 64B/66B 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 23], or Packaging Custom IP for use with IP Integrator for the steps to package the standard CC module.

The `do_cc` port can be used to send clock compensation sequences at any time and for any duration to meet the needs of specific channels. The most common use of this feature is scheduling clock compensation events to occur outside of frames, or at specific times during a stream to avoid interrupting data flow.

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

Clock Compensation Logic Customizing Guidelines

- Clock compensation sequences should last at least three `user_clk` cycles to ensure that they are recognized by all receivers
- Ensure the duration and period selected are sufficient to correct for the maximum difference between the frequencies of the clocks used
- Do not perform multiple clock compensation sequences within eight cycles of one another
- Clock Compensation should not be disabled when hot-plug logic is enabled

Flow Control Interface

The flow control interface consists of three configurations: the native flow control, the user flow control, and the USER_K flow control interfaces as described in the subsequent sections.

Native Flow Control Interface

The Aurora 64B/66B protocol includes native flow control (NFC) allowing receivers to control the rate at which data is sent by specifying the number of cycles during which the channel partner cannot send data. The data flow can even be turned off completely (XOFF) by requesting that the transmitter temporarily send only idles. NFC is typically used to prevent FIFO overflow conditions. Figure 2-19 and Table 2-10 detail the NFC port Interface.

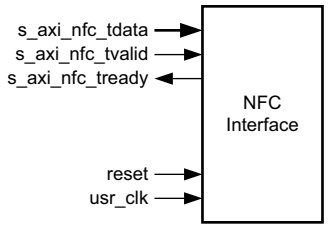


Figure 2-19: **NFC Port Interface**

Table 2-10: **Native Flow Control (NFC) Interface Ports**

Name	Direction	Clock Domain	Description
s_axi_nfc_tx_tvalid	Input	user_clk	Asserted (High) to request sending an NFC message to the channel partner. Must be held until `s_axi_nfc_tx_tready` is asserted.
s_axi_nfc_tx_tready	Output	user_clk	Asserted (High) when an Aurora 64B/66B core accepts an NFC request.
s_axi_nfc_tx_tdata[0:15] or s_axi_nfc_tx_tdata[15:0]	Input	user_clk	Incoming NFC message data from the channel partner.

For a detailed explanation of NFC operation, see the *Aurora 64B/66B Protocol Specification* (SP011) [Ref 5].

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

Figure 2-20 and Figure 2-21 show the NFC message format in big endian (default) and little endian modes.

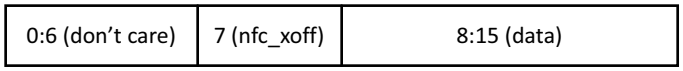


Figure 2-20: **NFC Message in Default Big Endian Mode**

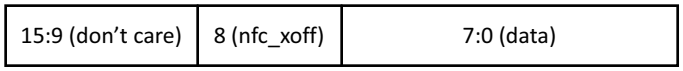


Figure 2-21: **NFC Message in Little Endian Mode**

NFC Message in Default Mode

To send an NFC message to a channel partner, the user application asserts `s_axi_nfc_tx_tvalid` and writes an 8-bit pause count to `s_axi_nfc_tx_tdata[8:15]`. The pause code indicates the minimum number of cycles the channel partner must wait after receiving an NFC message prior to resuming data send. The number of `user_clk` cycles without data is equal to `s_axi_nfc_tx_tdata` + 1.

When asserted, the `s_axi_nfc_tx_tdata[7]` signal indicates `nfc_xoff`, requesting that the channel partner stop sending data until it receives a non-XOFF NFC message or reset. When a request is transmitted with PAUSE and XOFF both set to 0, NFC is set to XON mode. To cancel XOFF mode, all 0s (XON) should be transmitted. After reception of this XON request, any new NFC request is honored by the core. The user application must hold `s_axi_nfc_tx_tvalid`, `s_axi_nfc_tx_tdata[8:15]`, and `s_axi_nfc_tx_tdata[7]` (`nfc_xoff`, if used) until `s_axi_nfc_tx_tready` is asserted on a positive `user_clk` edge indicating that the core can transmit the NFC message.

Aurora 64B/66B cores cannot transmit data while sending NFC messages. `s_axi_tx_tready` is always deasserted on the cycle following an `s_axi_nfc_tx_tready` assertion. NFC Completion mode is available only for the framing Aurora 64B/66B interface.

Example A: Transmitting an NFC Message

Figure 2-22 shows an example of the transmit timing when the user application sends an NFC message to a channel partner using an AXI4-Stream interface.

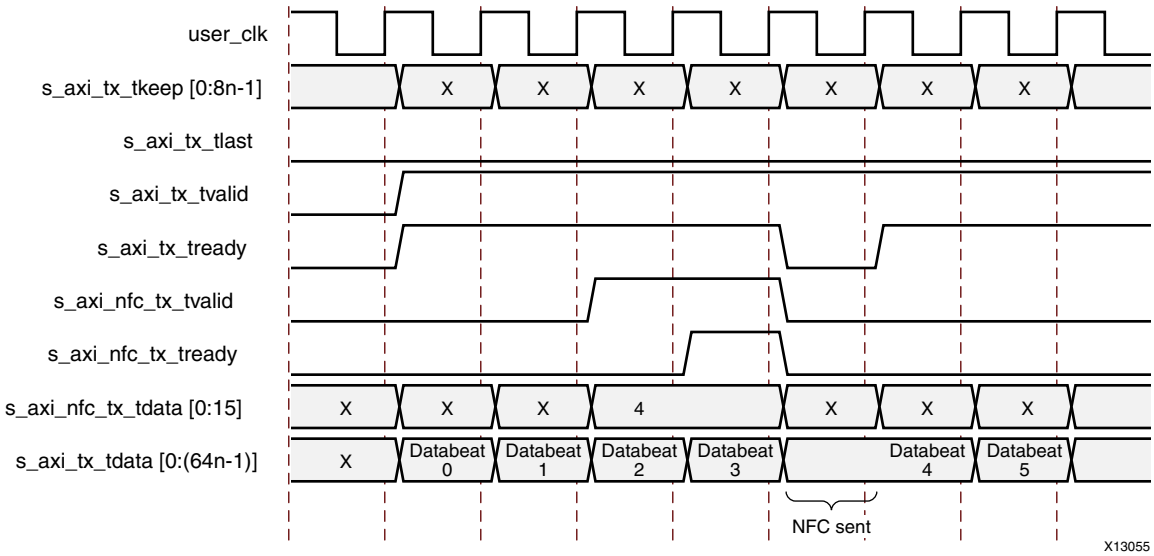


Figure 2-22: **Transmitting an NFC Message**

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

Example B: Receiving a Message with NFC Idles Inserted

Figure 2-23 shows an example of the TX user interface signals in immediate NFC mode when an NFC message is received. The NFC message sends `8'b01`, requesting two cycles without data transmission. The core deasserts `s_axi_tx_tready` to prevent data transmission for two cycles.

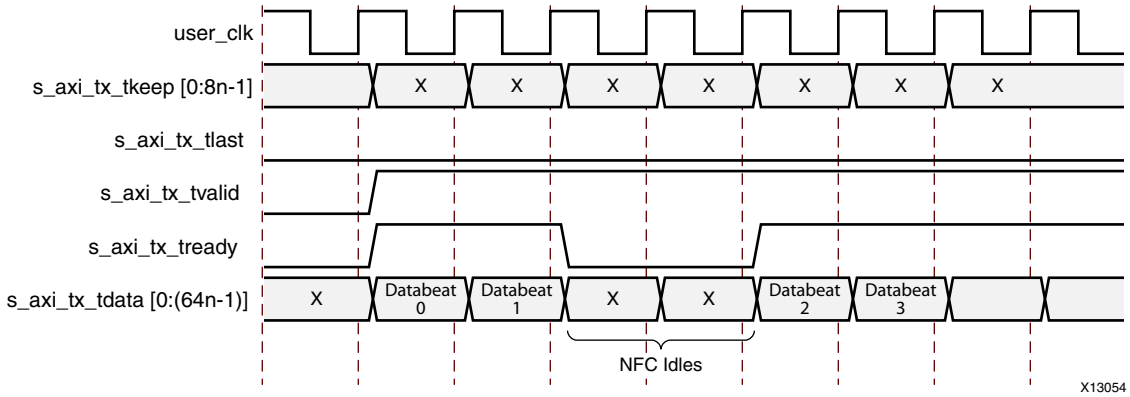


Figure 2-23: **Transmitting a Message with NFC Idles Inserted**

Aurora 64B/66B cores can also operate in completion mode where NFC Idles are only inserted before the first data bytes of a new frame. If a completion mode core receives an NFC message while it is transmitting a frame, the core finishes transmitting the frame before deasserting `s_axi_tx_tready` to insert idles.

User Flow Control Interface

The Aurora 64B/66B protocol includes user flow control (UFC) to allow channel partners to send control information using a separate in-band channel. Applications send short UFC messages to the channel partner without waiting for the frame in progress to end. The higher priority UFC message shares the channel with lower-priority regular frame data. UFC messages are interruptible by high-priority control blocks such as CC/NR/CB/NFC blocks. UFC message interruption is visible when the UFC option is selected.

Figure 2-24 shows the UFC port interface. Table 2-11 describes the UFC interface ports.

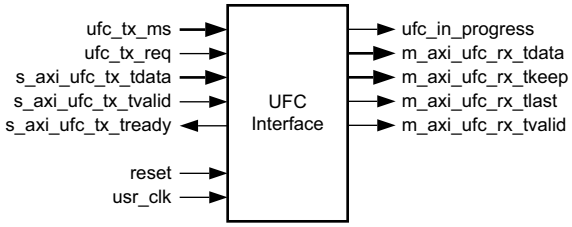


Figure 2-24: **UFC Port Interface**

Table 2-11: **User Flow Control (UFC) Interface Ports**

Name	Direction	Clock Domain	Description
ufc_tx_req	Input	user_clk	Indicates a UFC message request to send by channel partner. After a request, the `s_axi_ufc_tx_tdata` bus is ready to send data after two cycles unless interrupted by a higher priority event.
ufc_tx_ms[0:7] or ufc_tx_ms[7:0]	Input	user_clk	Specifies the number of bytes in the UFC message (message size). The maximum UFC message size is 256 bytes. The value specified on `ufc_tx_ms` is one less than the actual number of bytes transferred (a value of 3 transmits 4 bytes of data).

Table 2-11: **User Flow Control (UFC) Interface Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
s_axi_ufc_tx_tready	Output	user_clk	Indicates the Aurora 64B/66B core is ready to accept UFC data on `s_axi_ufc_tx_tdata`. This signal is asserted two clock cycles after `ufc_tx_req` when no high-priority requests are in progress. `s_axi_ufc_tx_tready` continues to be asserted while the core waits for data for the most recently requested UFC message. The signal is deasserted for CC, CB, and NFC requests which are higher priority. While `s_axi_ufc_tx_tready` is asserted, `s_axi_tx_tready` is deasserted.
s_axi_ufc_tx_tdata[0:(64*n*–1)] or s_axi_ufc_tx_tdata[(64*n*–1):0](1)	Input	user_clk	Input bus for Aurora channel UFC message data. Sampled only if `s_axi_ufc_tx_tvalid` and `s_axi_ufc_tx_tready` are asserted. If the number of message bytes is not an integer multiple of the bus width in bytes, the only bytes used are those needed on the last cycle to finish the message starting from the leftmost byte of the bus.
s_axi_ufc_tx_tvalid	Input	user_clk	Indicates valid UFC data on `s_axi_ufc_tx_tdata`. If deasserted while `s_axi_ufc_tx_tready` is asserted, Idle blocks are sent in the UFC message.
m_axi_ufc_rx_tdata[0:(64*n*–1)] or m_axi_ufc_rx_tdata[(64*n*–1):0](1)	Output	user_clk	Incoming UFC message data from the channel partner.
m_axi_ufc_rx_tvalid	Output	user_clk	Indicates valid UFC data on the `m_axi_ufc_rx_tdata` port. When not asserted, all values on the `m_axi_ufc_rx_tdata` port should be ignored.
m_axi_ufc_rx_tlast	Output	user_clk	Indicates the end of the incoming UFC message.
m_axi_ufc_rx_tkeep[0:(8*n*–1)] or m_axi_ufc_rx_tkeep[(8*n*–1):0](1)	Output	user_clk	Specifies the number of valid data bytes 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. Each bit indicates one valid byte. Maximum size of the UFC message is 256 bytes.
ufc_in_progress	Output	user_clk	Specifies the status of the current UFC transmission.

Notes:

1. *n* is the number of lanes.

Transmitting UFC Messages

To send a UFC message, the application asserts `ufc_tx_req` while driving `ufc_tx_ms` with the desired SIZE code for a single cycle. After a request, a new request cannot be made until `s_axi_ufc_tx_tready` is asserted for the final cycle of the previous request. The UFC message data must be placed on `s_axi_ufc_tx_tdata` and the `s_axi_ufc_tx_tvalid` signal must be asserted whenever the bus contains valid message data.

The core deasserts `s_axi_tx_tready` while sending UFC data and keeps `s_axi_ufc_tx_tready` asserted until it has enough data to complete the requested message. If `s_axi_ufc_tx_tvalid` is deasserted during a UFC message, idles are sent, `s_axi_tx_tready` remains deasserted, and `s_axi_ufc_tx_tready` remains asserted. If a CC, CB, or NFC request is made, `s_axi_ufc_tx_tready` is deasserted while the requested operation is performed because CC, CB, and NFC requests have higher priority.

Note: The `s_axi_tx_tready` and `s_axi_ufc_tx_tready` signals are deasserted for one cycle before the core accepts message data to allow The UFC header to be sent.

Example A: Transmitting a Single-Cycle UFC Message

Figure 2-25 shows the procedure for transmitting a single-cycle UFC message. This example shows a 4-byte message being sent on an 8-byte interface.

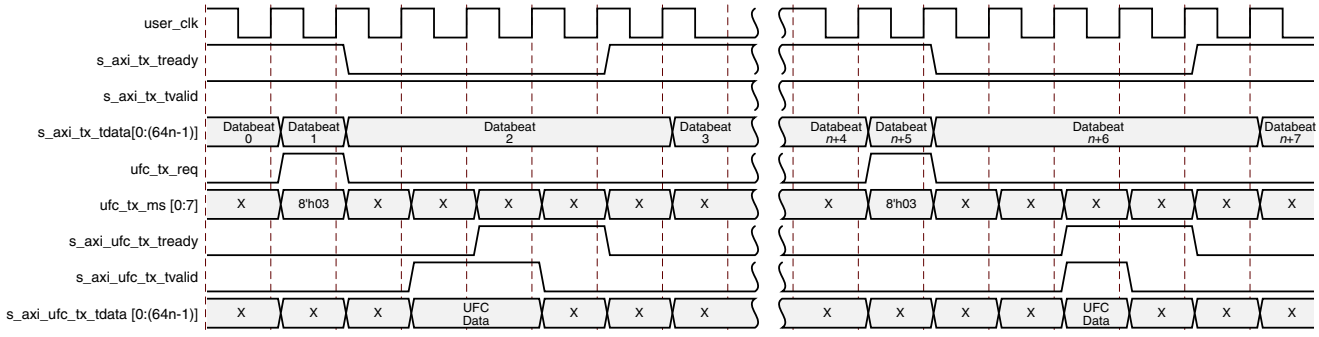


Figure 2-25: **Transmitting a Single-Cycle UFC Message**

Example B: Transmitting a Multicycle UFC Message

Figure 2-26 shows the procedure for transmitting a two-cycle UFC message. This example shows a 16-byte message being sent on an 8-byte interface.

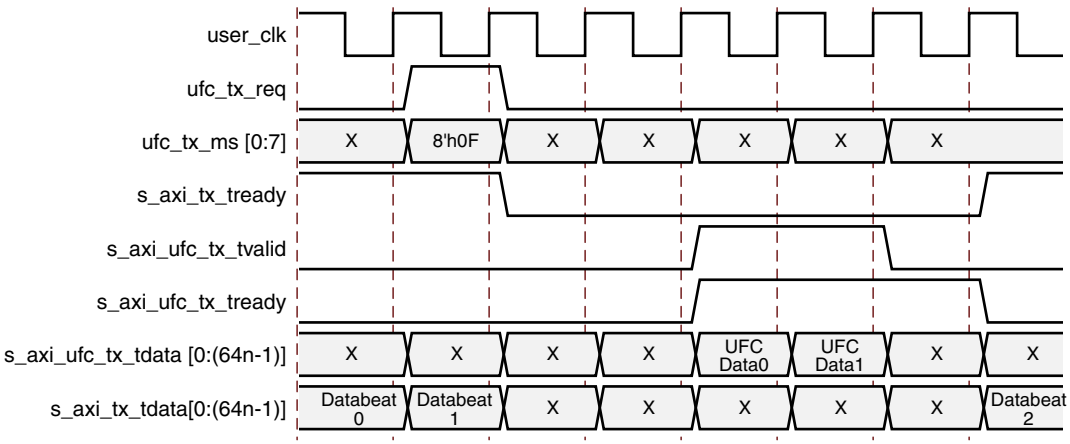


Figure 2-26: **Transmitting a Multi-Cycle UFC Message**

Example C: Receiving a Single-Cycle UFC Message

Figure 2-27 shows an Aurora 64B/66B core with an 8-byte data interface receiving a 4-byte UFC message. The core presents this data to the application by asserting `m_axi_ufc_rx_tvalid` and `m_axi_ufc_rx_tlast` indicating a single-cycle frame. The `m_axi_ufc_rx_tkeep` bus is set to `0xF`, indicating only the four most significant interface bytes are valid (each bit in `TKEEP` indicates a valid byte in the UFC data).

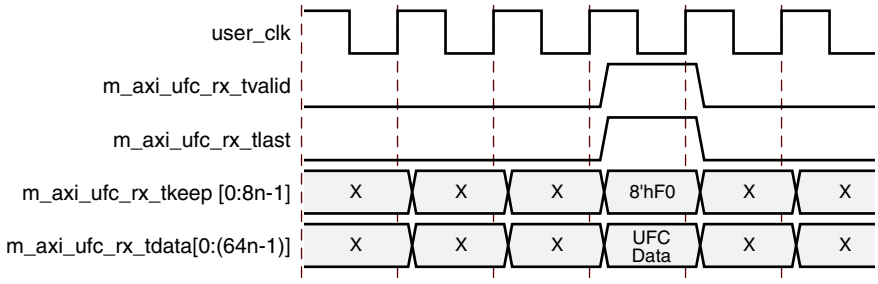


Figure 2-27: **Receiving a Single-Cycle UFC Message**

Example D: Receiving a Multicycle UFC Message

Figure 2-28 shows an Aurora 64B/66B core with an 8-byte interface receiving a 15-byte message. The resulting frame is two cycles long, with `m_axi_ufc_rx_tkeep` set to `8'hFF` for the first cycle indicating all bytes are valid and `8'hFE` for the second cycle indicating seven of the bytes are valid.

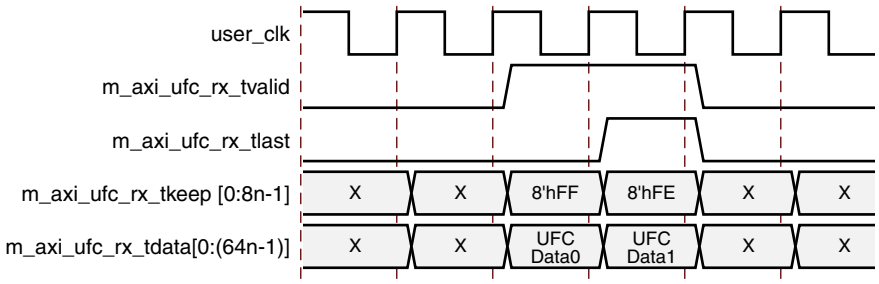


Figure 2-28: **Receiving a Multi-Cycle UFC Message**

User K-Block Interface

Figure 2-29 shows the USER-K interface ports for a single-lane design with the USER-K interface enabled.

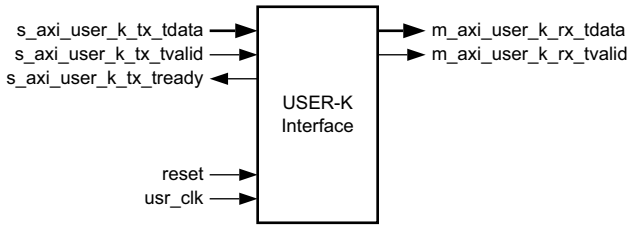


Figure 2-29: **USER-K Port Interface**

User K-blocks are special, single-block codes that include control blocks passed directly to the user application without being decoded by the Aurora interface. These blocks can be used to implement application-specific control functions and have a lower priority than UFC blocks but higher than user data blocks.

Table 2-12 lists the USER-K interface ports.

Table 2-12: **USER-K Interface Ports**

Name	Direction	Clock Domain	Description
s_axi_user_k_tx_tdata[0:(64*n*–1)] or s_axi_user_k_tx_tdata[(64*n*–1):0](1)	Input	user_clk	User K-block data is 64-bit aligned. Signal Mapping per lane: Default: s_axi_user_k_tx_tdata={4'h0, user_k_blk_no[0:4n–1], s_axi_user_k_tdata[0:56n–1]}. Little Endian format: s_axi_user_k_tx_tdata={s_axi_user_k_tdata[56n–1:0], 4'h0, user_k_blk_no[4n–1:0]}.
s_axi_user_k_tx_tvalid	Input	user_clk	Indicates valid User-K data on the `s_axi_user_k_tx_tdata` port.
s_axi_user_k_tx_tready	Output	user_clk	Indicates the Aurora 64B/66B core is ready to accept data on the `s_axi_user_k_tx_tdata` interface.
m_axi_rx_user_k_tvalid	Output	user_clk	Indicates valid User-K data on the `m_axi_user_k_tx_tdata` port.
m_axi_rx_user_k_tdata or m_axi_rx_user_k_tdata[(64*n*–1):0](1)	Output	user_clk	Received User K-blocks from the Aurora lane are 64-bit aligned. Signal Mapping per lane: Default: m_axi_rx_user_k_tdata={4'h0, rx_user_k_blk_no[0:4n–1], m_axi_rx_user_k_tdata[0:56n–1]}. Little Endian format: m_axi_rx_user_k_tdata= {m_axi_rx_user_k_tdata[56n–1:0], 4'h0, rx_user_k_blk_no[4n–1:0]}.

Notes:

1. *n* is the number of lanes.

The user K-block is not differentiated for streaming or framing designs. Each user K-block is eight bytes wide and is encoded with a USER-K BTF value as specified in Table 2-13. The BTF value is indicated by the user application in the `s_axi_user_k_tx_tdata` port as a user K-block number. The user K-block is a single block code and is always delineated by a user K-block number. Provide the user K-block number as specified in Figure 2-30 and Figure 2-31. The user K-block data is limited to the specified seven bytes of the `s_axi_user_k_tx_tdata` port.

Table 2-13: **User K-Block Valid Block Type Field (BTF) Values**

User K-Block Number	User K-Block BTF
User K-Block 0	0xD2
User K-Block 1	0x99

Table 2-13: **User K-Block Valid Block Type Field (BTF) Values *(Cont'd)***

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

Figure 2-30 shows the USER-K format in default (big-endian) mode.

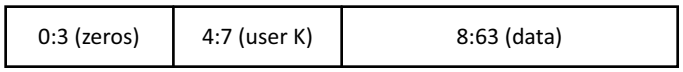


Figure 2-30: **USER-K Format in Default Mode**

Figure 2-31 shows the USER-K format in little-endian mode.

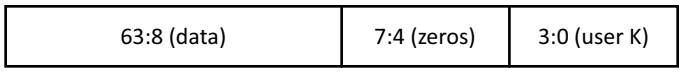


Figure 2-31: **USER-K Format in Little-Endian Mode**

Transmitting User K-Blocks

The s_axi_user_k_tx_tready signal is asserted by the core and is prioritized by CC, CB, NFC, and UFC messages. After asserting s_axi_user_k_tx_tdata, the User K block number and s_axi_user_k_tx_tvalid is asserted. If required, the user application can change s_axi_user_k_tx_tdata when s_axi_user_k_tx_tready is asserted (Figure 2-32). This action enables the Aurora core to select the appropriate USER-K BTF from the nine user K-blocks. The data available during assertion of s_axi_user_k_tx_tready is always serviced.

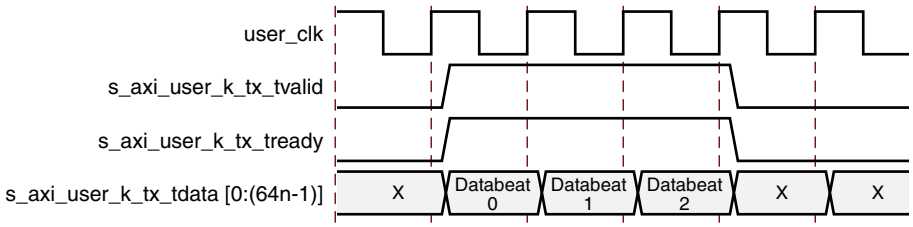


Figure 2-32: **Transmitting User K Data and User K-Block Number**

Receiving User K-Blocks

The receive BTF is decoded and the block number for the corresponding BTF is passed to the user application (Figure 2-33). The user application can validate the data available on `m_axi_rx_user_k_tdata` when `m_axi_rx_user_k_tvalid` is asserted.

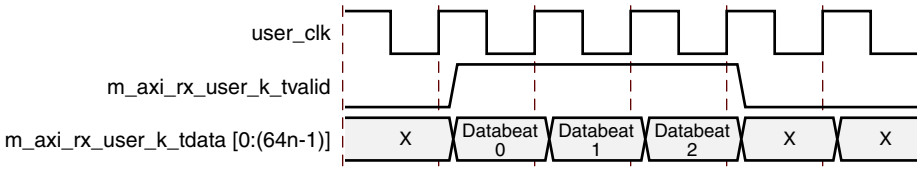


Figure 2-33: **Receiving User K Data and User K-Block Number**

Status, Control, and the Transceiver Interface

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

Status Control and Transceiver Ports

Table 2-14 describes the function of the Aurora 64B/66B core status and control ports allowing user applications to monitor the Aurora channel and access built-in features of the serial transceiver interface. The DRP interface allows reading and updating the serial transceiver parameters and settings through the AXI4-Lite protocol-compliant or Native DRP interfaces.

Table 2-14: **Status and Control Ports**

Name	Direction	Clock Domain	Description
channel_up/ tx_channel_up/ rx_channel_up	Output	user_clk	Asserted when the Aurora channel initialization is complete and channel is ready to send/receive data. *`channel_up` is available in duplex mode.* *`tx_channel_up` is available in TX-only_Simplex and TX/RX_Simplex mode.* *`rx_channel_up` is available in RX-only_Simplex and TX/RX_Simplex mode.*
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. The Aurora 64B/66B core can only receive data after all lane_up signals are asserted. *`lane_up` is available in duplex mode.* *`tx_lane_up` is available in TX-only_Simplex and TX/RX_Simplex mode.* *`rx_lane_up` is available in RX-only_Simplex and TX/RX_Simplex mode.*
soft_err/ tx_soft_err/ rx_soft_err	Output	user_clk	Indicates a soft error is detected in the incoming serial stream (asserted for a single user_clkperiod). *`soft_err` is available in duplex mode.* *`tx_soft_err` is available in TX-only_Simplex and TX/RX_Simplex mode.* *`rx_soft_err` is available in RX-only_Simplex and TX/RX_Simplex mode.*
hard_err/ tx_hard_err/ rx_hard_err	Output	user_clk	Hard error detected (asserted until the core resets). *`hard_err` is available in duplex mode.* *`tx_hard_err` is available in TX-only_Simplex and TX/RX_Simplex mode.* *`rx_hard_err` is available in RX-only_Simplex and TX/RX_Simplex mode.*
reset/ tx_reset/ rx_reset	Input	user_clk	Resets the Aurora 64B/66B core. Connected to top level through a debouncer. Systematically resets all of the Aurora core logic. Debounced with `user_clk` for at least six `user_clk` cycles. *`reset` is available in duplex, TX-only_Simplex, and RX-only simplex modes.* *`tx_reset` is available in TX/RX_Simplex mode.* *`rx_reset` is available in TX/RX_Simplex mode.*
reset_pb/ tx_reset_pb/ rx_reset_pb	Input	user_clk	Push Button Reset. The top-level reset input at the example design Level. Required to drive the Support Reset logic inside the core. *`reset_pb` is available in Duplex, TX-only_simplex and RX-only_Simplex modes.* *`tx_reset_pb` is available in TX/RX_Simplex mode.* *`rx_reset_pb` is available in TX/RX_Simplex mode.*

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt_reset_out	Output	init_clk	Output of de-bouncer for `gt_reset`. Enabled when **include Shared Logic in Core** is selected.
sys_reset_out	Output	user_clk	System reset output to be used by example design level logic.
link_reset_out	Output	init_clk	Driven High if hot-plug count expires.
gt_pll_lock	Output	init_clk	Asserted when `tx_out_clk` is stable. When deasserted (Low), circuits using `tx_out_clk` should be held in reset.
pma_init	Input	init_clk	The transceiver `pma_init` reset signal is connected to the top level through a debouncer. Systematically resets all Physical Coding Sublayer (PCS) and Physical Medium Attachment (PMA) subcomponents of the transceiver. The signal is debounced using `init_clk_in` for at least six `init_clk` cycles. See the Reset section in the related transceiver user guide for more details.
rxp[0:*m*–1](1)	Input	RX serial clk	Positive differential serial data input pin. *This input is not available in the TX-only_Simplex configuration.*
rxn[0:*m*–1](1)	Input	RX serial clk	Negative differential serial data input pin. *This input is not available in the TX-only_Simplex configuration.*
txp[0:*m*–1](1)	Output	TX serial clk	Positive differential serial data output pin. *This output is not available in the RX-only_Simplex configuration.*
txn[0:*m*–1](1)	Output	TX serial clk	Negative differential serial data output pin. *This output is not available in the RX-only_Simplex configuration.*
gt_rxcdrovrden_in	Input	async	RXCDR Override. Configures the transceiver in loopback mode.
loopback[2:0]	Input	async	See the *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476) [Ref 4] or *UltraScale Architecture GTH Transceivers User Guide* (UG576) [Ref 3] for details about loopback.
power_down	Input	init_clk	Drives the Aurora 64B/66B core to reset.
gt_qpllock_quad<*quad_no*>_in, gt_qpllrefclklost_quad<*quad_no*>_in(3)	Input	init_clk	QPLL lock and reference clock lost signal slave partner inputs. Should be connected to the master partner shared logic output ports `gt_qpllock_quad<quad_no>_out` and `gt_qpllrefclklost_quad<quad_no>_out` respectively.
gt_qpllock_quad<*quad_no*>_out, gt_qpllrefclklost_quad<*quad_no*>_out(3)	Output	init_clk	QPLL lock and reference clock lost signal master partner shared logic outputs.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt_to_common_qpllreset_out	Output	async	QPLL common reset output used by the slave partner shared logic.
CHANNEL_DRP_IF(6)			
drp_clk_in	Input	-	A user-configurable parameter only applicable to 7 series FPGA designs. The default value is 100 MHz. The `drp_clk` frequency can be set from 50 MHz to *x* MHz where *x* is device and speed grade dependent.
drpaddr_in/ gt<*lane*>_drpaddr(14)(8)	Input	drp_clk_ in	DRP address bus. In 7 series FPGAs, the drpaddress bus is multiplexed across all lanes. In UltraScale FPGAs, the drpaddress bus is available on a per lane basis.
drpdi_in/ gt<*lane*>_drpdi(14)(8)	Input	drp_clk_ in	Data bus for reading configuration data from the transceiver to the FPGA logic resources. In 7 series FPGAs, the DRP data input bus is multiplexed across all lanes. In UltraScale FPGAs, the DRP data input bus is available on a per lane basis.
drpen_in_lane_<*lane*>/ gt<*lane*>_drpen(14)(8)	Input	drp_clk_ in	DRP enable signal. **0**: No read or write operation performed. **1**: Enables a read or write operation. For write operations, `drpwe` and `drpen` should be driven High concurrently for one `drp_clk_in` cycle only. For read operations, `drpen` should be driven High for one `drp_clk_in` cycle. The DRP enable is available on a per lane basis.
drpwe_in_lane_<*lane*>/ gt<*lane*>_drpwe(8)(14)	Input	drp_clk_ in	DRP write enable. **0**: Read operation when drpen is 1. **1**: Write operation when drpen is 1. For write operations, `drpwe` and `drpen` should be driven High for one `drpclk` cycle only. The DRP write enable is available on a per lane basis.
drpdo_out_lane_<*lane*>/ gt<*lane*>_drpdo(8)(14)	Output	drp_clk_ in	Data bus for reading configuration data from the GTX or GTH transceiver to the FPGA logic resources. The DRP data out bus is available on a per lane basis.
drprdy_out_lane_<*lane*>/ gt<*lane*>_drprdy(8)(14)	Output	drp_clk_ in	Indicates write operation is complete and read data is valid. The `drprdy` signal is available on a per lane basis.
AXILITE_DRP_IF(6)			
s_axi_awaddr_lane_<*lane_no*>(2)(9)	Input	drp_clk_ in	AXI4-Lite Write address for DRP.
s_axi_awvalid_lane_<*lane_no*>(2)(9)	Input	drp_clk_ in	Write address valid.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
s_axi_awready_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Write address ready.
s_axi_araddr_lane_<*lane_no*>(2)(9)	Input	drp_clk_in	AXI4-Lite Read address for DRP.
s_axi_arvalid_lane_<*lane_no*>(2)(9)	Input	drp_clk_in	Read address valid.
s_axi_arready_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Read address ready.
s_axi_wdata_lane_<*lane_no*>(2)(9)	Input	drp_clk_in	Write data for DRP.
s_axi_wvalid_lane_<*lane_no*>(2)(9)	Input	drp_clk_in	Write data valid.
s_axi_wready_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Write data ready.
s_axi_wstrb_lane_<*lane_no*>(2)(9)	Input	drp_clk_in	Write data strobe.
s_axi_bvalid_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Write response valid.
s_axi_bresp_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Write response.
s_axi_rdata_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Read data.
s_axi_rvalid_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Read data valid.
s_axi_rresp_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Read response.
s_axi_rready_lane_<*lane_no*>(2)(9)	Output	drp_clk_in	Read data ready.
s_axi_bready_lane_<*lane_no*>(2)(9)	Input	drp_clk_in	Write data ready.
TRANSCEIVER_DEBUG(11)			
gt<*lane*>_cplllock_out/ gt_cplllock(4)(5)(10)(14)	Output	init_clk	Active-High PLL frequency lock signal indicating PLL frequency is within predetermined tolerance. The transceiver and its clock outputs are not reliable until this condition is met.
gt<*lane*>_dmonitorout_out[*j*:0]/ gt_dmonitorout(4)(5)(10)(12)(14)	Output	async	Digital Monitor Output Bus. *j* = 7 for GTX transceivers. *j* = 14 for GTH transceivers.
gt<*lane*>_eyescandataerror_out/ gt_eyescandataerror(4)(5)(10)(12)(14)	Output	async	Asserted High for one `rec_clk` cycle when an (unmasked) error occurs while in the COUNT or ARMED state.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<*lane*>_eyescanreset_in/ gt_eyescanreset[4][5][10][12][14]	Input	async	Driven High, then deasserted to start the EYESCAN reset process.
gt<*lane*>_eyescantrigger_in/ gt_eyescantrigger[4][5][10][12][14]	Input	user_clk	Causes a trigger event.
gt<*lane*>_rxbufreset_in/ gt_rxbufreset[4][5][10][12][14]	Input	async	Driven High, then deasserted to start the RX elastic buffer reset process. In either single or sequential mode, activating `rxbufreset` resets the RX elastic buffer only.
gt<*lane*>_rxbufstatus_out/ gt_rxbufstatus[4][5][10][12][14]	Output	rxoutclk	RX buffer status. **`000b`**: Nominal condition. **`001b`**: Number of bytes in the buffer are less than CLK_COR_MIN_LAT. **`010b`**: Number of bytes in the buffer are greater than CLK_COR_MAX_LAT. **`101b`**: RX elastic buffer underflow. **`110b`**: RX elastic buffer overflow.
gt<*lane*>_rxcdrhold_in/ gt_rxcdrhold[4][5][10][12][14]	Input	async	Holds the CDR control loop frozen.
gt<*lane*>_rxdfeagchold_ in[5][10][12][13][14]	Input	rxoutclk	HOLD RX DFE **`2'b00`**: Automatic gain control (AGC) loop adapt. **`2'b10`**: Freeze current AGC adapt value. **`2'bx1`**: Override AGC value according to attribute. RX_DFE_GAIN_CFG
gt<*lane*>_rxdfeagcovrden_ in[5][10][12][13][14]	Input	rxoutclk	OVRDEN RX DFE **`2'b00`**: Automatic gain control (AGC) loop adapt. **`2'b10`**: Freeze current AGC adapt value. **`2'bx1`**: Override AGC value according to attribute. RX_DFE_GAIN_CFG
gt<*lane*>_rxdfelfhold_ in[5][10][12][13][14]	Input	rxoutclk	When set to `1'b1`, the current low-frequency boost value is held. When set to `1'b0`, the low-frequency boost is adapted.
gt<*lane*>_rxdfelpmreset_in/ gt_rxdfelpmreset[4][5][10][12][14]	Input	rxoutclk	Driven High, then deasserted to start the DFE reset process.
gt<*lane*>_rxlpmen_in/ gt_rxplmen[4][5][10][12][14]	Input	rxoutclk	RX datapath **0**: DFE **1**: LPM
gt<*lane*>_rxlpmhfovrden_ in[5][10][12][13][14]	Input	rxoutclk	OVRDEN RX LPM **`2'b00`**: KH high frequency loop adapt value. **`2'b10`**: Freeze current adapt value. **`2'bx1`**: Override KH value according to RXLPM_HF_CFG attribute.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<*lane*>_rxlpmlfklovrden_in[5][10][12][13][14]	Input	rxoutclk	OVRDEN RX LPM `2'b00`: KL low frequency loop adapt value. `2'b10`: Freeze current adapt value. `2'bx1`: Override KL value according to RXLPM_LF_CFG attribute.
gt<*lane*>_rxmonitorout_out[5][10][12][13][14]	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[5][10][12][13][14]	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*>_rxpcsreset_in/ gt_rxpcsreset[4][5][10][12][14]	Input	async	Driven High, then deasserted to start RX PMA reset process. The `rxpcsreset` signal does not start the reset process until `rxuserrdy` is High.
gt<*lane*>_rxpmareset_in/ gt_rxpmareset[4][5][10][12][14]	Input	async	Driven High, then deasserted to start RX PMA reset process.
gt<*lane*>_rxpmaresetdone_out/ gt_rxpmaresetdone[4][5][10][12][14]	Output	async	Indicates RX PMA reset is complete. Driven Low when GTRXRESET or RXPMARESET is asserted. Available for duplex and RX-only simplex configurations. *Available only with GTH transceivers*.
gt<*lane*>_rxprbscntreset_in/ gt_rxprbscntreset[4][5][10][12][14]	Input	rxoutclk	Resets the PRBS error counter.
gt<*lane*>_rxprbserr_out/ gt_rxprbserr[4][5][10][12][14]	Output	rxoutclk	Non-sticky status output indicates PRBS errors have occurred.
gt<*lane*>_rxprbssel_in/ gt_rxprbssel[4][5][10][12][14]	Input	rxoutclk	Receiver PRBS checker test pattern control. Valid settings: `000`: Standard operation (PRBS check off). `001`: PRBS-7. `010`: PRBS-15. `011`: PRBS-23. `100`: PRBS-31. No checking is done for non-PRBS patterns. Single-bit errors cause bursts of PRBS errors because the PRBS checker uses data from the current cycle to generate expected data for the next cycle.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt_rxrate[4][10][12]	Input	rxoutclk	Dynamic pins to automatically change effective PLL dividers in the GTH transceiver RX. These ports are used for PCI Express® and other standards. Available only with UltraScale FPGAs.
gt<*lane*>_rxresetdone_out/ gt_rxresetdone[4][5][10][12][14]	Output	rxoutclk	When asserted, indicates the GTX/GTH transceiver RX has finished reset and is ready for use. Driven Low when `gtrxreset` is driven High. Not driven High until `rxuserrdy` goes High.
gt<*lane*>_txbufstatus_out/ gt_txbufstatus[4][5][10][11][14]	Output	user_clk	**`txbufstatus[1]`**: TX buffer overflow or underflow status. When `txbufstatus[1]` is set High, signal remains High until the TX buffer is reset. **1**: TX FIFO has overflow or underflow. **0**: No TX FIFO overflow or underflow error. **`txbufstatus[0]`**: TX buffer fullness. **1**: TX FIFO is at least half full. **0**: TX FIFO is less than half full.
gt<*lane*>_txdiffctrl_in/ gt_txdiffctrl[4][5][10][11][14]	Input	user_clk	Driver Swing Control. Available for duplex and TX-only simplex configurations.
gt<*lane*>_txmaincursor_ in[5][10][11][13][14]	Input	async	Allows the main cursor coefficients to be set directly if the TX_MAINCURSOR_SEL attribute is set to `1'b1`.
gt<*lane*>_txpcsreset_in/ gt_txpcsreset[4][5][10][11][14]	Input	async	Resets the TX PCS. Driven High, then deasserted to start the PCS reset process. Activating this port only resets the TX PCS.
gt<*lane*>_txpmareset_in/ gt_txpmareset[4][5][10][11][14]	Input	async	Resets the TX PMA. Driven High, then deasserted to start the TX PMA reset process. Activating this port resets both the TX PMA and the TX PCS.
gt<*lane*>_txpolarity_in/ gt_txpolarity[4][5][10][11][14]	Input	user_clk	Inverts 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*>_txpostcursor_in/ gt_txpostcursor[4][5][10][11][14]	Input	async	Transmitter post-cursor TX pre-emphasis control.
gt<*lane*>_txprbsforceerr_in/ gt_txprbsforceerr[4][5][10][11][14]	Input	user_clk	When driven High, errors are forced into the PRBS transmitter. While asserted, the output data pattern contains errors. When `txprbssel` is set to `000`, this port does not affect TXDATA.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt<*lane*>_txprbssel_in/ gt_txprbssel(4)(5)(10)(11)(14)	Input	user_clk	Transmitter PRBS generator test pattern control. For 7 series devices: **000**: Standard mode (pattern generation off). **001**: PRBS-7. **010**: PRBS-15. **011**: PRBS-23. **100**: PRBS-31. **101**: PCI Express compliance pattern. Only works with 20-bit and 40-bit modes. **110**: Square wave with 2 UI (alternating 0s/1s). **111**: Square wave with 16 UI, 20 UI, 32 UI, or 40 UI period (based on data width). For UltraScale devices: **4'b0000**: Standard mode (pattern generation off). **4'b0001**: PRBS-7. **4'b0010**: PRBS-9. **4'b0011**: PRBS-15. **4'b0100**: PRBS-23. **4'b0101**: PRBS-31. **4'b1000**: PCI Express compliance pattern. Only works with internal data width 20 bit and 40 bit modes. **4'b1001**: Square wave with 2 UI (alternating 0s/1s). **4'b1010**: Square wave with 16 UI, 20 UI, 32 UI, or 40 UI period (based on internal data width).
gt<*lane*>_txprecursor_in/ gt_txprecusor(4)(5)(10)(11)(14)	Input	async	Transmitter pre-cursor TX pre-emphasis control.
gt<*lane*>_txresetdone_out/ gt_txresetdone(4)(5)(10)(11)(14)	Output	user_clk	Indicates the GTX/GTH transceiver TX has finished reset and is ready for use. Driven Low when `gttxreset` goes High and not driven High until the GTX/GTH transceiver TX detects `txuserrdy` High.

Table 2-14: **Status and Control Ports *(Cont'd)***

Name	Direction	Clock Domain	Description
gt_qplllock_quad<*quad_no*>/ gt_qplllock[3][4][5][11][10]	Output	init_clk	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.

Notes:

1. *m* is the number of GTX or GTH transceivers.
2. *lane_no* varies from 1 to (number of lanes –1).
3. In 7 series FPGAs, *quad_no* varies from 1 to (number of active transceiver quads –1). In UltraScale FPGAs, *quad_no* varies from 1 to the number of active transceiver quads.
4. Refer to the *UltraScale FPGAs GTH Transceivers User Guide* (UG576) [Ref 3] for more information about debug ports.
5. Refer to the *7 series FPGAs Transceivers User Guide* (UG476) [Ref 4] for more information about debug ports.
6. In Ultrascale devices, all DRP and AXI4-LITE ports are sampled on `init_clk`.
7. The Transceiver_Debug ports are enabled if the **Additional transceiver control and status ports** option is selected in the Debug and Control section of the Vivado IDE Core Options page. 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.
8. This port is available if the **Native** option is selected in the DRP Mode section of the Vivado IDE Core Options page.
9. This port is available if the **AXI4LITE** option is selected in the DRP Mode section of the Vivado IDE Core Options page.
10. This port is available if the **Additional transceiver control and status ports** option is selected in the DRP Mode section of the Vivado IDE Core Options page.
11. Available for Duplex, TX-Only Simplex and TX/RX_Simplex configurations.
12. Available for Duplex, RX-Only Simplex and TX/RX_Simplex configurations.
13. Not available with UltraScale devices.
14. *lane* varies from 0 to (number of lanes –1).
15. *quad* varies from 0 to (number of active transceiver quads –1).

Figure 2-34 shows the status and control interface for an Aurora 64B/66B duplex core.

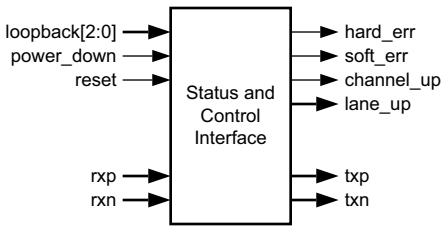


Figure 2-34: **Aurora 64B/66B Duplex Status and Control Interface**

Figure 2-35 shows the status and control interface for an Aurora 64B/66B TX-only_Simplex core.

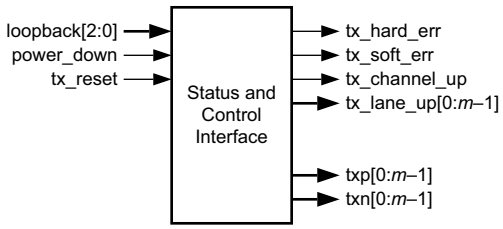


Figure 2-35: **Aurora 64B/66B TX-Only Simplex Status and Control Interface**

Figure 2-36 shows the status and control interface for an Aurora 64B/66B RX-only_Simplex core.

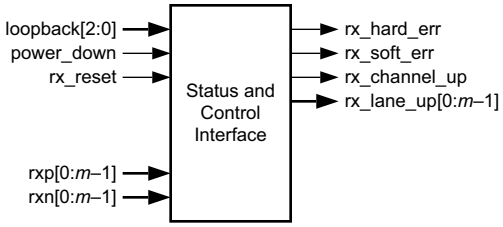


Figure 2-36: **Aurora 64B/66B RX-Only Simplex Status and Control Interface**

Figure 2-37 shows the status and control interface for an Aurora 64B/66B TX/RX_Simplex core.

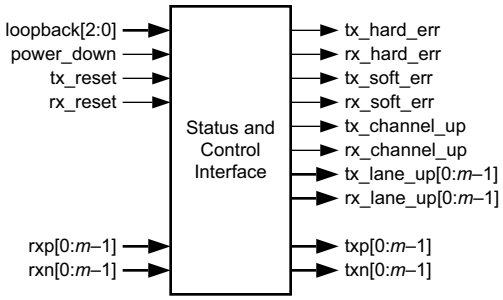


Figure 2-37: **Aurora 64B/66B TX/RX Simplex Status and Control Interface**

Error Signals in Aurora 64B/66B Cores

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

The core also monitors each high-speed serial GTX and GTH transceiver for hardware errors such as buffer overflow and loss of lock. Hardware errors are reported by asserting the `hard_err` signal. Catastrophic hardware errors can also manifest themselves as a burst of soft errors. The Block Sync algorithm described in the *Aurora 64B/66B Protocol Specification v1.3* (SP011) [Ref 5] determines whether to treat a burst of soft errors as a hard error.

Whenever a hard error is detected, the core automatically resets itself and attempts to re-initialize. In most cases, this permits reestablishing the Aurora channel when the hardware issue causing the hard error is resolved. Soft errors do not lead to a reset unless enough occur in a short period of time to trigger the block sync state machine.

Table 2-15 describes the core error signals.

Table 2-15: **Aurora 64B/66B Core Error Signals**

Signal	Description	TX	RX
hard_err	**TX Overflow/Underflow**: Overflow or underflow condition exists in the TX data elastic buffer. This condition can occur when the user clock and the reference clock sources are not operating at the same frequency.	X	
	RX Overflow/Underflow: Overflow or underflow condition exists in the RX data clock correction and channel bonding FIFO. This condition can occur when the clock source frequencies for the two channel partners are not within ±100 ppm.		X
soft_err	**Soft Errors**: Too many soft errors occurred within a short period of time. The alignment block sync state machine automatically attempts to realign if too many invalid sync headers are detected. Soft errors are not transformed into hard errors.		X
	Invalid SYNC Header: The 2-bit header on the 64-bit block was not a valid control or data header.		X
	Invalid BTF: The block type field (BTF) of a received control block contained an unrecognized value. This condition usually results from a bit error.		X

Initialization

Aurora 64B/66B cores initialize automatically after power-up, reset, or hard error (Figure 2-38). Core modules on each side of the channel perform the Aurora initialization procedure until the channel is ready for use. The `lane_up` bus indicates which lanes in the channel have finished the lane initialization portion of the procedure. The `lane_up` 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.

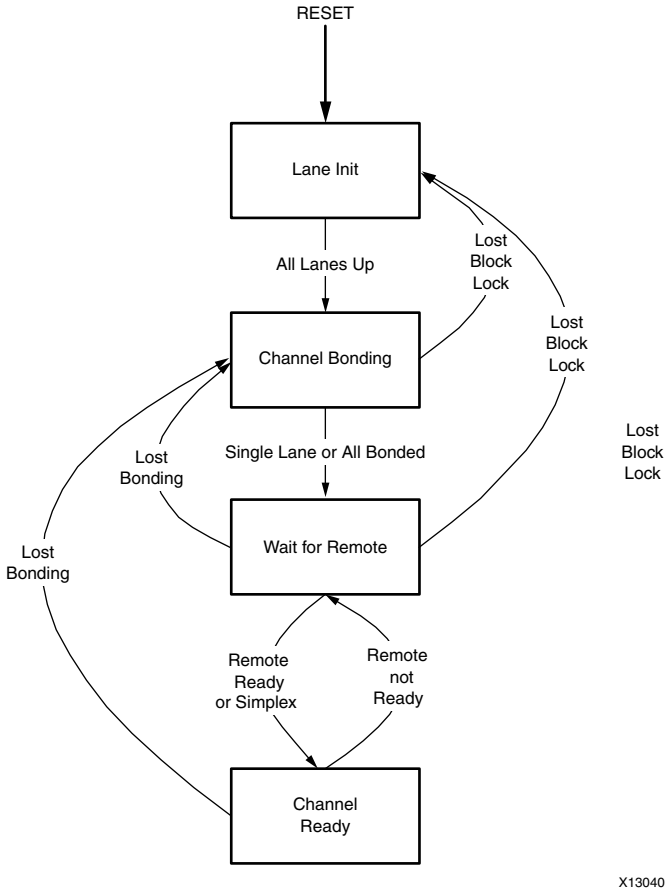


Figure 2-38: **Initialization Overview**

Aurora 64B/66B cores can receive data before `channel_up` is asserted. Only the user interface `m_axi_rx_tvalid` signal should be used to qualify incoming data. Because no transmission can occur until after `channel_up` is asserted, `channel_up` can be inverted and used to reset modules that drive the TX side of a full-duplex channel. If user application modules need to be reset before data reception, an inverted `lane_up` signal can be used for this purpose. Data cannot be received until all of the `lane_up` signals are asserted.

Aurora Simplex Operation

Simplex Aurora 64B/66B cores have no sideband connection and use timers to declare that the partner is out of initialization and, thus, ready for data transfer. Simplex TX/RX cores have both transmit and receive portions of the transceiver configured to operate independently. However, the Simplex TX/RX cores have reset and `pma_init` signals in common between the transmit and receive path of the core.

The BACKWARD_COMP_MODE3 TX/RX_Simplex core parameter can be used to prevent unintentional hot plug events from inhibiting channel up assertion. This parameter is available in the `<user_component_name>_core.v` file and is available for all core configurations.

- BACKWARD_COMP_MODE3 = 0 clears the hot plug counter only on reception of CC characters
- BACKWARD_COMP_MODE3 = 1 clears the hot plug counter on reception of any valid BTF characters

RECOMMENDED: *Follow the reset sequence given in Chapter 3.*

DRP Interface

The DRP interface allows user applications to monitor and modify the transceiver status. The Native interface provides the native transceiver DRP interface ports. The AXI4-LITE interface (fully compliant with the AXI4-LITE protocol) is the default interface.

For native DRP sequences, the read and write operations are as specified in the respective FPGA Transceivers User Guides. For AXI4-LITE DRP sequences, the read and write operations from the user interface are specified in the AXI4-LITE protocol. The Aurora core does not use the `wstrb` signal (refer to the Vivado AXI Reference Guide, (UG1037) [Ref 18]). The axi_to_drp module is used to translate between the transceiver DRP and AXI4-LITE protocols.

Transceiver Debug Interface

Figure 2-39 and Figure 2-40 show the additional available transceiver debugging control and status ports when the TRANSCEIVER DEBUG interface is selected for 7 series and UltraScale devices, respectively.

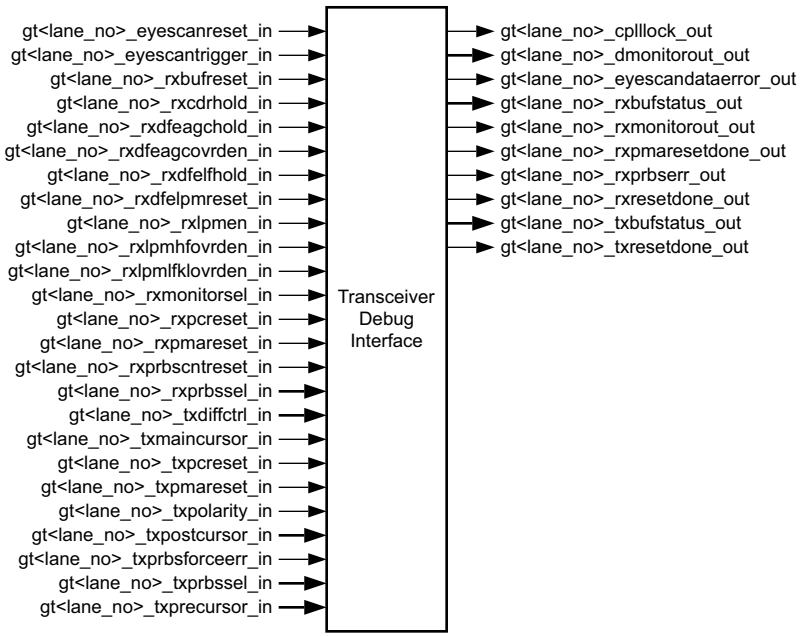


Figure 2-39: **7 Series Devices Transceiver Debug Interface Ports**

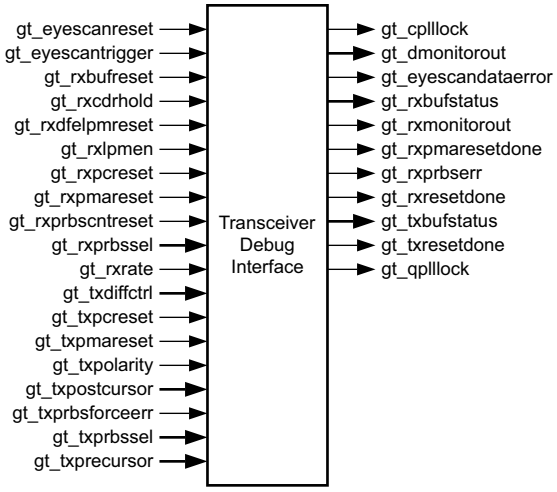


Figure 2-40: **UltraScale Devices Transceiver Debug Interface Ports**

CRC Interface

CRC is an optional interface. The `crc_valid` and `crc_pass_fail_n` signals (Table 2-16) indicate the result of a received frame transmitted with CRC. See Using CRC, page 67 for more information.

Table 2-16: **CRC Interface Ports**

Name	Direction	Clock Domain	Description
crc_valid	Output	user_clk	Samples the `crc_pass_fail_n` signal.
crc_pass_fail_n	Output	user_clk	The `crc_pass_fail_n` signal is asserted High when the received CRC matches the transmitted CRC. This signal is not asserted if the received CRC is not equal to the transmitted CRC. The `crc_pass_fail_n` signal should always be sampled with the `crc_valid` signal.

Designing with the Core

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

General Design Guidelines

All Aurora 64B/66B 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 a flip-flop between the user application and the core. While registering signals might not be possible for all paths, it simplifies timing analysis and makes it easier for the Xilinx® tools to place-and-route the design.

Recognize Timing Critical Signals

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

Clocking

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

Each Aurora 64B/66B core is generated in the `example_project` directory which includes a design called `aurora_example`. This design instantiates the generated Aurora 64B/66B core and demonstrates a working clock configuration for the core. First-time users should examine the aurora example design for use as a template when connecting the clock interface.

Clock Interface and Clocking

Aurora 64B/66B Clocking Architecture

Figure 3-1 shows the clocking architecture in the Aurora 64B/66B core for Zynq®-7000, Virtex®-7, and Kintex®-7 device GTX or GTH transceivers.

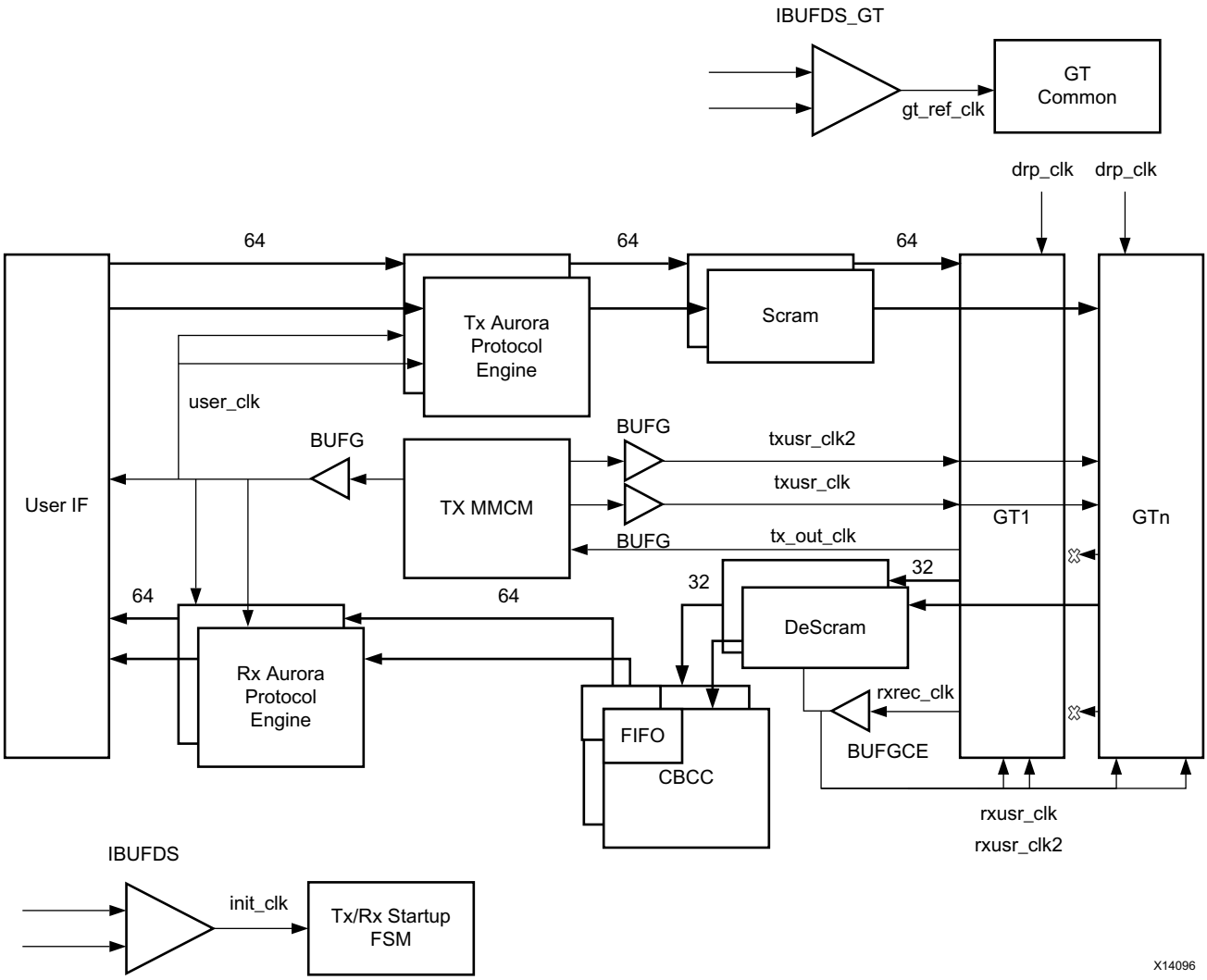


Figure 3-1: **Aurora 64B/66B Clocking for Zynq-7000, Virtex-7, and Kintex-7 Device GTX or GTH Transceivers**

Connecting user_clk, sync_clk, and tx_out_clk

The Aurora 64B/66B cores use three phase-locked parallel clocks. The first is `user_clk`, which synchronizes all signals between the core and the user application. All logic touching the core must be driven by `user_clk`, which in turn must be the output of a global clock buffer (BUFG).

The `user_clk` signal is used to drive the `txusrclk2` port of the serial transceiver. The `tx_out_clk` is selected such that the data rate of the parallel side of the module matches the data rate of the serial side of the module, taking into account 64B/66B encoding and decoding.

The third phase-locked parallel clock is `sync_clk`. This clock must also come from a BUFG and is used to drive `txusrclk` port of the serial transceiver. It is also connected to the Aurora 64B/66B core to drive the internal synchronization logic of the serial transceiver.

To make it easier to use the two parallel clocks, a clock module is provided in a subdirectory called `clock_module` under `example_design/support` or under `src` based on shared logic settings. The ports for this module are described in Table 2-14, page 39. If the clock module is used, the `mmcm_not_locked` signal should be connected to the `mmcm_not_locked` output of the clock module; `tx_out_clk` should connect to the clock module `clk` port, and `pll_lock` should connect to the clock module `pll_not_locked` port. If the clock module is not used, connect the `mmcm_not_locked` signal to the inverse of the `locked` signal from any PLL used to generate either of the parallel clocks, and use the `pll_lock` signal to hold the PLLs in reset during stabilization if `tx_out_clk` is used as the PLL source clock. The `txusrclk` could be unreliable during assertion of `pma_init`; hence, the core uses a stable clock (`init_clk`) for MMCM synchronization. Using a stable clock to sample adds more robustness to the link.

If MMCM is used to generate a stable clock (`init_clk`), `pma_init` needs to be applied to the Aurora core until MMCM lock is established. This ensures that the core remains in a known state before a stable clock is available for the core.

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

The Aurora 64B/66B core uses four BUFGs for a given core configuration using GTX or GTH transceivers. Aurora 64B/66B is an eight-byte-aligned protocol, and the datapath from the user interface is 8-bytes aligned. For GTX or GTH transceivers, the core configures the transmit path as eight bytes and the receive path as four bytes.

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

Reference Clocks for FPGA Designs

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

For UltraScale™ devices, the Xilinx implementation tools make necessary adjustments to the north-south routing and the pin swapping necessary to the GTHE3 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:*

1. *In 7 Series FPGAs, the total number of GTX or GTH transceiver quads sourced by an external clock pin pair must not exceed three quads (one quad above and one quad below), or 12 GTXE2_CHANNEL/GTHE2_CHANNEL transceivers. Designs in 7 Series FPGAs with more than 12 transceivers or more than three quads should use multiple external clock pins.*
2. *In Ultrascale FPGAs, the total number of transceiver quads sourced by an external clock pin pair must not exceed five quads (two quads above and two quads below), or twenty GTHE3_CHANNEL transceivers.*

See 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 4] and UltraScale FPGAs GTH Transceivers User Guide (UG576) [Ref 3] for more details on transceiver reference clocks. Manual editing of the transceiver attributes is not recommended but can be performed following the recommendations in the aforementioned guides.

Reset and Power Down

Reset

The reset signal is used to restore the Aurora 64B/66B core to a known starting state. Upon reset, the core stops the current operation and re-initializes the channel. It is expected that `user_clock` is stable when the reset signal is applied. When the reset signal to the Aurora channel partner1 is asserted, channel partner2 also loses lock. Channel partner2 regains lock when channel partner1 is out of reset and begins transmitting valid patterns. On full-duplex Aurora cores, when asserted on the positive edge of `user_clk`, the reset signal resets both the TX and RX sides of the channel. Simplex Aurora cores have similar reset ports for both partners and require a different reset sequence. Asserting `pma_init` resets the entire serial transceiver which eventually resets the Aurora core as well. Also, it is assumed `init_clk` is always stable and the `ref_clk` is stable at the time of de-assertion of the `pma_init` signal.

Reset Sequence

The following are recommended reset sequences for the Aurora 64B/66B core at the example design level for the available dataflow configurations.

Aurora 64B/66B Duplex

During the board power on sequence, both `pma_init` and `reset_pb` signals are expected to be High. `INIT_CLK` and `GT_REFCLK` are expected to be stable during power on for the proper functioning of the Aurora core. When both clocks are stable, `pma_init` is de-asserted followed by the de-assertion of `reset_pb` synchronous with `user_clk`.

Aurora 64B/66B Duplex Power On Sequence

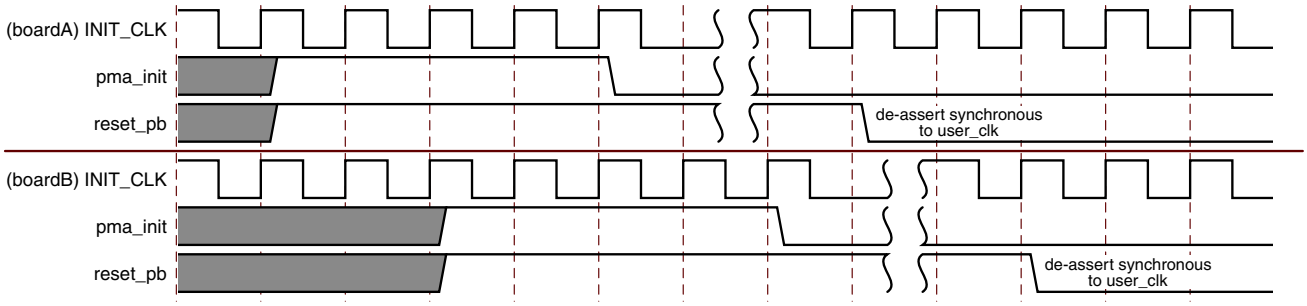


Figure 3-2: **Aurora 64B/66B Duplex Power On Reset Sequence**

Aurora 64B/66B Duplex Normal Operation Reset Sequence

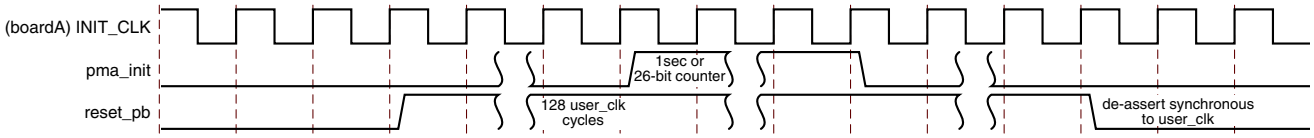


Figure 3-3: **Aurora 64B/66B Duplex Normal Operation Reset Sequence**

Reset Sequencing

1. Assert reset. Wait for a minimum of 128 `user_clk` cycle times
2. Assert `pma_init`. Keep `pma_init` and reset asserted for at least one second to prevent the transmission of CC characters and ensure that the remote agent detects a hot plug event.
3. Deassert `pma_init`.
4. Deassert `reset_pb` synchronous to `user_clk`.

Aurora 64B/66B Simplex

During power on for both TX simplex and RX simplex cores, the pma_init and reset_pb signals are expected to be High. INIT_CLK and GT_REFCLK are expected to be stable during the power-on sequence. The pma_init on the TX board must be deasserted first, followed by the deassertion of pma_init on the RX side to ensure proper RX side CDR lock.

Aurora 64B/66B Simplex Power On Sequence

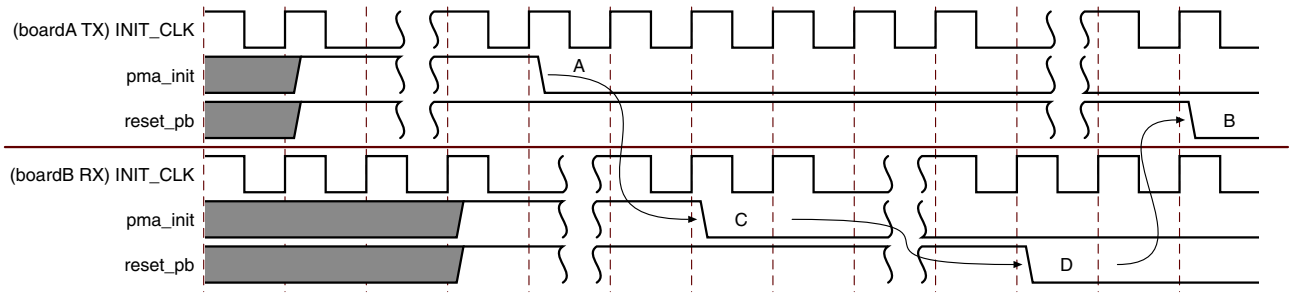


Figure 3-4: **Aurora 64B/66B Simplex Power On Sequence**

1. Deassert TX side pma_init (A).
2. Deassert RX side pma_init (C).
3. Deassert RX side reset_pb synchronous to user_clk (D).
4. Deassert TX side reset_pb synchronous to user_clk (B).

Note: Care must be taken to minimize the D -> B time difference.

Aurora 64B/66B Simplex Normal Operation Reset Sequence

For simplex configurations, because the TX and RX links mostly do not have a feedback path, TX side and RX side reset_pb sequences are always recommended to be tightly coupled. Note that if only the RX side reset_pb signal is asserted, there is no direct mechanism to notify the TX side of the reset_pb event. Every TX side reset_pb assertion must be followed by an RX side reset_pb assertion as shown in Figure 3-5. The time difference between the RX side reset_pb de-assertion and the TX side reset_pb de-assertion must be kept as minimal as possible. Before asserting pma_init, a minimum of 128 clock cycles is required to ensure that the portion of the core in programmable logic goes to a known reset state before the user_clk is held Low during pma_init assertion. The assertion time of pma_init must be a minimum of six INIT_CLK cycles to satisfy the requirements of the core de-bouncing circuit.

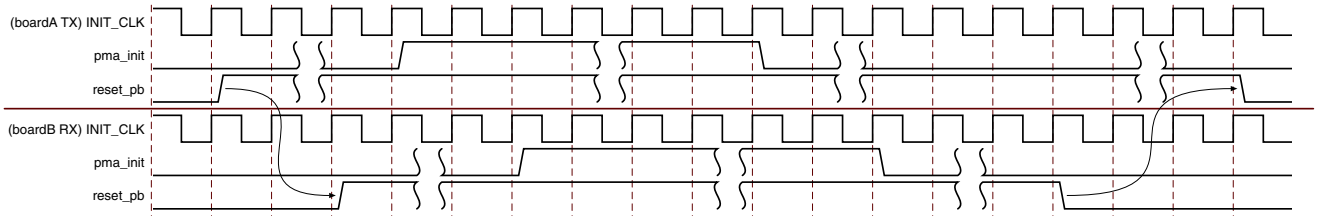


Figure 3-5: **Aurora 64B/66B Simplex Normal Operation Reset Sequence**

Aurora 64B/66B Simplex TX and RX

After the `pma_init` signal is deasserted, both `tx_reset_pb` and `rx_reset_pb` can be deasserted at the same time. For proper functioning of the link, it is recommended that the deassertion of `pma_init` for both boards should be as close to simultaneous as possible. After `rx_reset_pb` is deasserted, the link partner `tx_reset_pb` signal must be pulsed to bring the simplex link up (Figure 3-6).

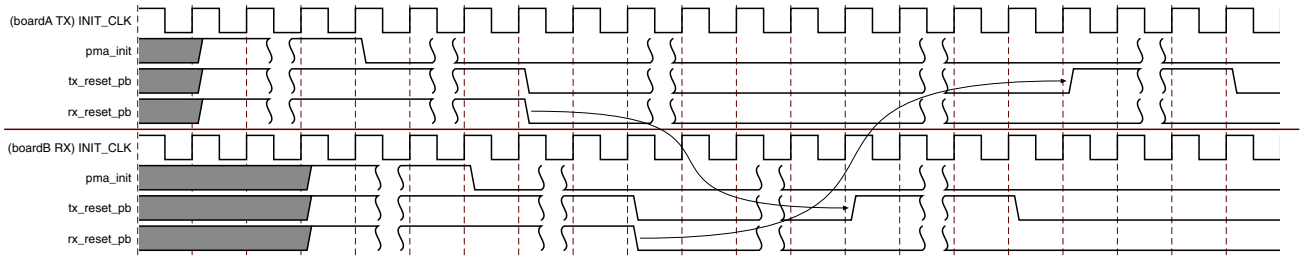


Figure 3-6: **Aurora 64B/66B Simplex TX and RX Power On Sequence**

For normal operation, follow the Aurora 64B/66B simplex normal operation reset sequence.

pma_init Staging in Example Design

The top level `pma_init` input at the example design level is delayed for 128 cycles (pma_init_stage). This signal is pulse-stretched for the duration of a 24-bit counter (pma_init_assertion). An aggregate signal from the instantiating logic is provided to the core as the `pma_init` input. This ensures that the assertion of the `pma_init` signal to the core results in reset assertion to the entire core.

Inside the `<user_component_name>_support_reset_logic.v` source file, the debouncer logic (reset_debounce_r) remains in reset state until the `gt_reset_in` signal (pma_init_assertion) is High ensuring an internally generated reset whenever the top level `pma_init` is asserted. Figure 3-7 illustrates this behavior.

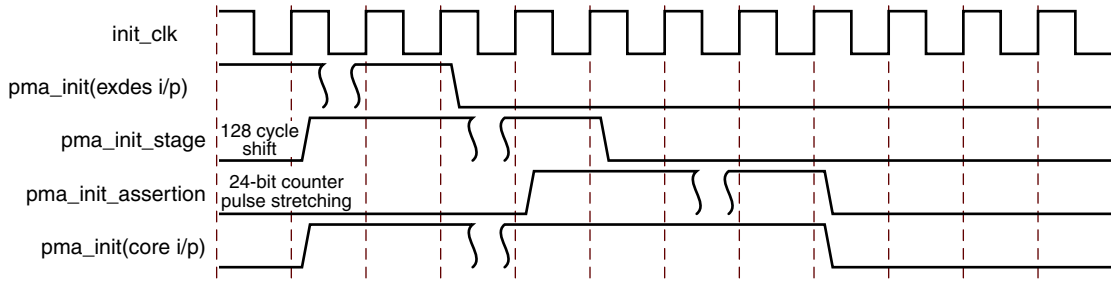


Figure 3-7: **pma_init Signal Staging**

Assertion of the `pma_init` signal to the core results in hot-plug reset assertion in the channel partner core. The reset sequence after hot-plug reset assertion is shown in Figure 3-8.

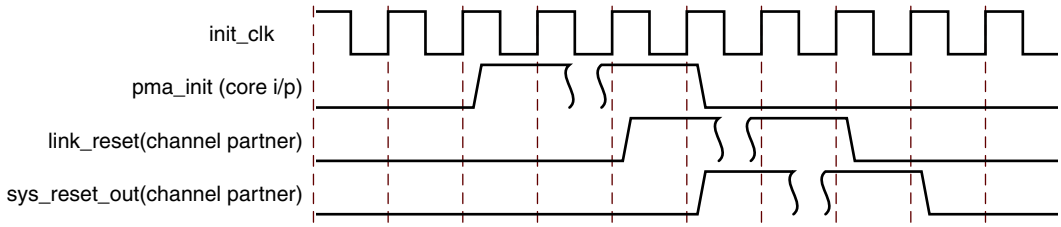


Figure 3-8: **pma_init Signal Used to Reset Remote System**

Reset Flow

The top-level RESET input (example design level) is debounced and connected to the core (reset_pb). This signal is aggregated with the serial transceiver reset status and the hot-plug reset from within the core reset logic (sys_reset_out) to generate a reset to the core. This signal is expected to connect to the core reset input. Figure 3-9 illustrates this behavior.

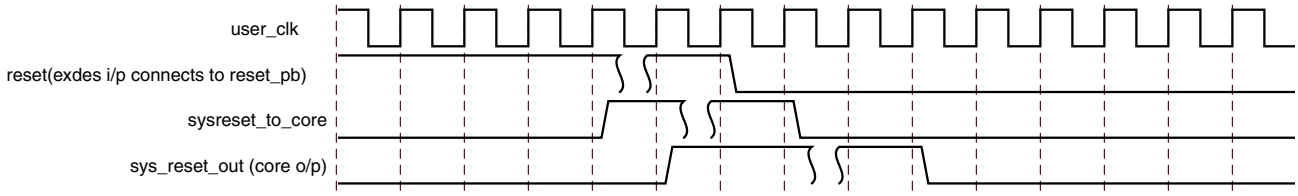


Figure 3-9: **Reset Flow**

`reset_pb` and reset at the core input should not be tied together to accommodate the preceding requirement. `sys_reset_out` should drive the reset input to the core along with additional system specific resets, if any.

Single Reset Use Cases

Use Case 1: `reset` assertion in duplex core

The `reset` assertion in the duplex core should be a minimum of 128 `user_clk` cycles. As a result, `channel_up` is deasserted as shown in the Figure 3-10.

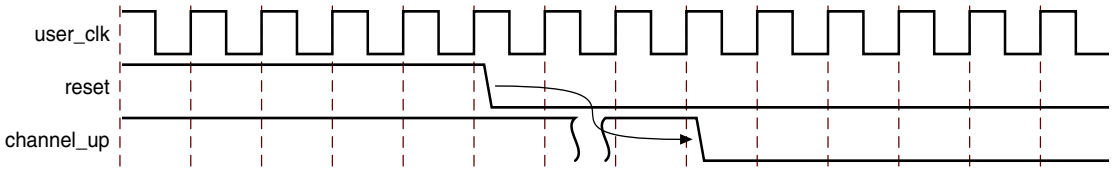


Figure 3-10: **Assertion of reset in the Duplex Core**

Use Case 2: `PMA_INIT` assertion in duplex core

Figure 3-11 shows the `pma_init` assertion in the duplex core and should be a minimum of 128 `init_clk` cycles. As a result, `user_clk` is stopped after a few clock cycles because there is no `txoutclk` from the transceiver and `channel_up` is deasserted.

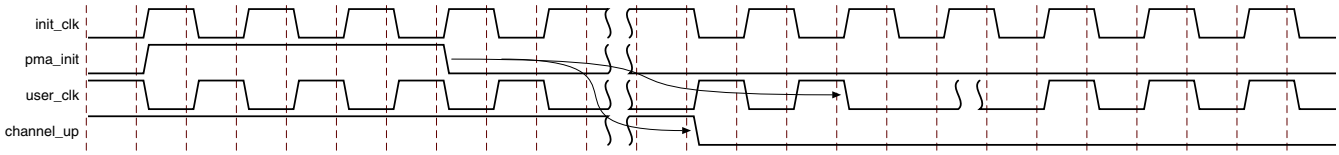


Figure 3-11: **pma_init Assertion in the Duplex Core**

Use Case 3: Assertion of reset in the Simplex Core

Figure 3-12 shows the Simplex-TX core and Simplex-RX core connected in a system. CONFIG1 and CONFIG2 can be in same or multiple device(s).

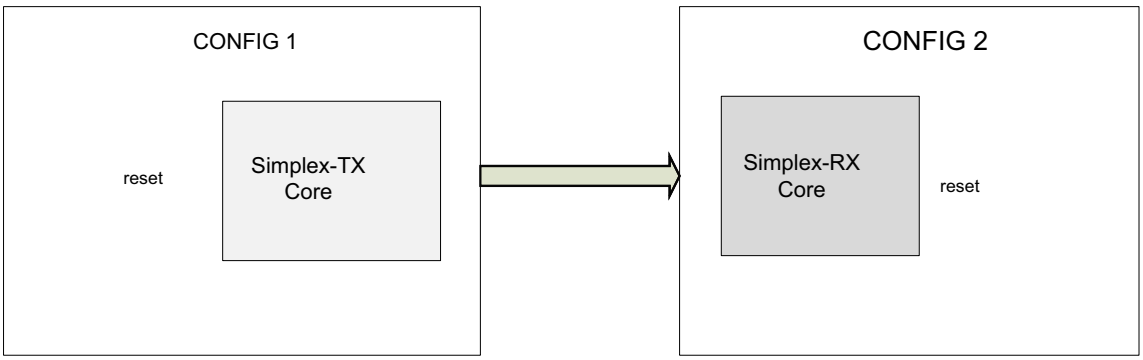


Figure 3-12: **System with Simplex Cores**

Following is the recommended procedure of TX cores reset and RX cores reset assertion in the Simplex core (see Figure 3-13).

1. The signal RX cores `reset` is asserted for 128 `user_clk` cycles followed by `reset` on the RX Simplex core asserted for 128 `user_clk` cycles.
2. `tx_channel_up` and `rx_channel_up` are deasserted after a minimum of five `user_clk` clock cycles.
3. The signal `reset` in the RX Simplex core is deasserted (or) released before `reset` in the TX Simplex core is deasserted. This ensures that the transceiver in the Simplex-RX core has sufficient transitions for CDR lock before the Simplex-TX core achieves TX_CHANNELUP.
4. `rx_channel_up` is asserted before `tx_channel_up` assertion. This condition must be satisfied by Simplex-RX core and simplex timer parameters (SIMPLEX_TIMER_VALUE) in Simplex-TX core needs to be adjusted to meet this criteria. The SIMPLEX_TIMER_VALUE parameter can be updated in `<user_component_name>_core.v`.
5. `tx_channel_up` is asserted after Simplex-TX core completes the Aurora protocol channel initialization sequence transmission for configured time. Assertion of `tx_channel_up` last ensures that the Simplex-TX core transmits an Aurora initialization sequence when Simplex-RX core is ready.

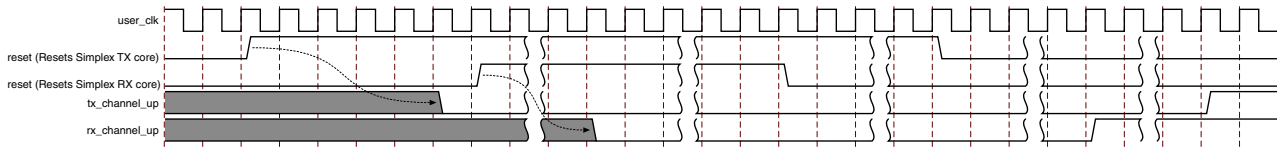


Figure 3-13: **Reset Assertion in Simplex Cores**

6. In the case of TX/RX Simplex cores, the reset sequence in duplex cores for `reset` and `pma_init` assertions can be followed. However, the SIMPLEX_TIMER_VALUE needs to be tuned based on the use model of the core.

Power Down

When `power_down` is asserted, only the Aurora 64B/66B core logic is reset. This does not turn off the GTX or GTH transceivers used in the design.

CAUTION! *Be careful when asserting this signal on cores that use `tx_out_clk` (see the Reference Clocks for FPGA Designs, page). `tx_out_clk` stops when the GTX and GTH transceivers are powered down. See the 7 Series FPGAs GTX/GTH Transceivers User Guide (UG476) [Ref 4] and the UltraScale Architecture GTH Transceivers User Guide (UG576) [Ref 3] for details.*

Timing

Figure 3-14 shows the timing for the reset signal. In a quiet environment, t_{CU} is generally less than 500 clock cycles. In a noisy environment, t_{CU} can be much longer.

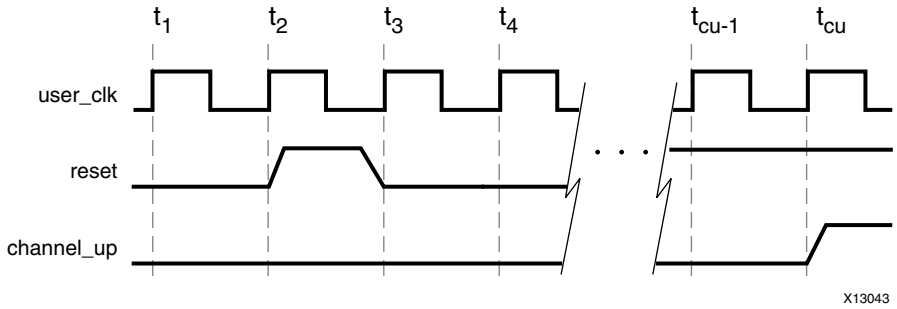


Figure 3-14: **Reset and Power Down Timing**

Core Features

This chapter contains information about individual features provided with the core.

Shared Logic

The **include Shared Logic in core** option on the Vivado® IDE Shared Logic page can be used to configure the core to include sharable resources such as the transceiver quad PLL (QPLL), the transceiver differential `refclk` buffer (IBUFDS_GTE2), and including clocking and reset logic either in the 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 which might not include them. This minimizes the amount of HDL modifications required while retaining core flexibility.

The shared logic hierarchy is called `<user_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. In these figures, `<user_component_name>` is the name of the generated core. The difference between the two hierarchies is the boundary of the core. The hierarchy is controlled using the Shared Logic options in the Vivado IDE.

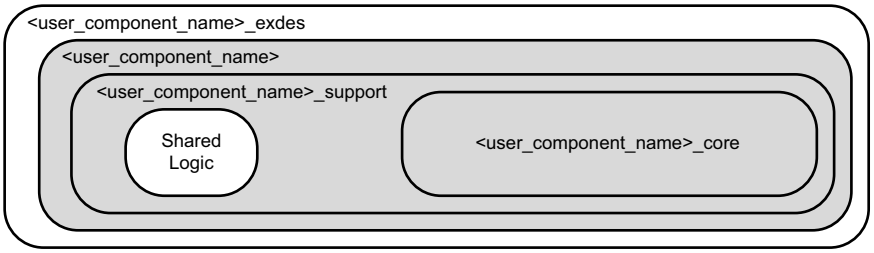


Figure 4-1: **Shared Logic Included in Core (highlighted in gray is the xci top)**

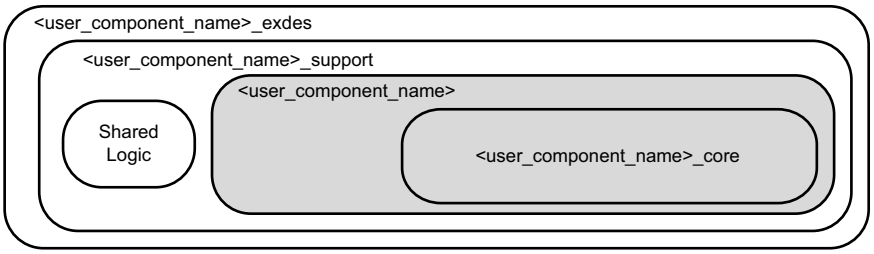


Figure 4-2: **Shared Logic Included in Example Design (highlighted in gray is the xci top)**

The contents of the shared logic depend upon the physical interface and the target device. Shared logic contains one or more instances of the transceiver differential buffer, support reset logic and instantiations of `<user_component_name>_clock_module`. Shared logic also contains either the GTXE2_COMMON or the GTHE2_COMMON block which is instantiated based on the selected transceiver type (GTX or GTH). Support reset logic contains the de-bouncer logic for the reset and `gt_reset` ports.

In Vivado IP Integrator, the connections between the two modes are shown for CPLL-based designs. As can be seen in Figure 4-3, the master `gt_reset` signal affects slave operation. For QPLL-based designs, `gt_qpllrefclklost` and `gt_qplllock` need to be connected. In Figure 4-3, the master core includes shared logic in the core and the slave core includes shared logic in the example design.

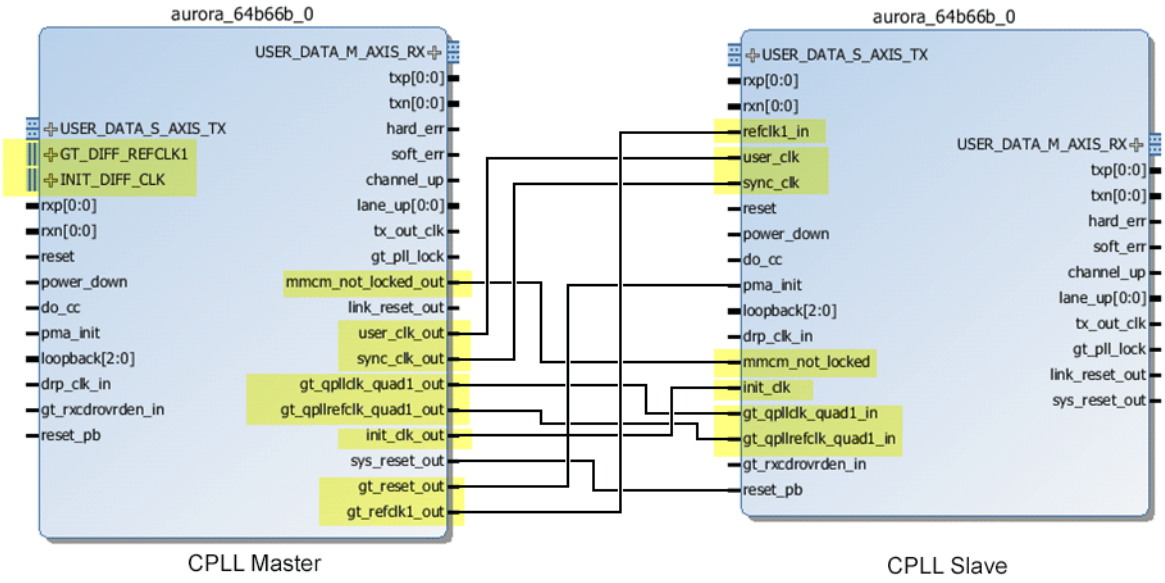


Figure 4-3: **Shareable Resource Connection Example Using IP Integrator**

See Table 2-8, page 24 and Table 2-14, page 39 for details about the port changes resulting from the setting of the shared logic option.

Table 4-1 shows the available shared resources based on transceiver type.

Table 4-1: **Aurora 64B/66B Core Available Shared Resources by Transceiver Type**

Transceiver Type	Available Shared Resources
Zynq®-7000 and 7 series devices GTX/GTH transceivers	IBUFDS_GTE2: transceiver reference clock GTXE2_COMMON: GTX transceiver clocking GTHE2_COMMON: GTH transceiver clocking BUFG: clocking IBUFDS: `init_clk`
UltraScale™ architecture GTH Transceivers	IBUFDS_GTE3: transceiver reference clock BUFG_GT: clocking

Using CRC

CRC is included in the core if the CRC mode option is selected. A framing user data interface with 32-bit CRC is available in the `<user_component_name>_crc_top.v` file. The `crc_valid` and `crc_pass_fail_n` signals (see Table 2-16, page 53) indicate the result of a received frame transmitted with CRC.

Hot Plug Logic

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

IMPORTANT: *It is highly recommended to keep hot plug logic enabled for predictable operation of the link.*

Following is the description of the hot-plug sequence.

1. Requirements: Before replacing the card, powering down a specific system, or reprogramming the bit file, it is required to assert `reset` before performing a hot plug sequence so that the remote agent channel goes down gracefully and gets ready when the link is removed and reconnected.
2. How it works: When `reset` is asserted at for least 128 cycles before performing a hot plug sequence, enough NA_IDLES are generated for the remote link to deassert Channel Up without errors.

3. Limitations: If the preceding sequence is not followed, SOFT/DATA errors are possible so that the link does not have a graceful shutdown.

Using Little Endian Support

The Aurora 64B/66B core supports the user interfaces in big endian format by default. The core also supports little endian format allowing easy connection to AXI4-Stream compliant IP designs.

Design Flow Steps

This chapter describes customizing and generating, constraining, simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about using standard design flows in the Vivado® IP Integrator can be found in the following 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 on using Vivado Design Suite to customize and generate the LogiCORE™ IP Aurora 64B/66B core.

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. The 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.

Use the following steps to customize the IP for use in your design by specifying values for the various parameters associated with the IP core:

1. Select the IP from the Vivado IP catalog (**IP Catalog -> Aurora 64B66B**).
2. Double-click the selected IP or select **Customize IP** from the toolbar or context 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].

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

IP Catalog

Figure 5-1 and Figure 5-2 show the features described in the corresponding sections. The left side displays a representative block diagram of the Aurora 64B/66B core as currently configured. The right side consists of user-configurable parameters. Details on the customizing options are provided in the following subsections, starting with Component Name, page 70.

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

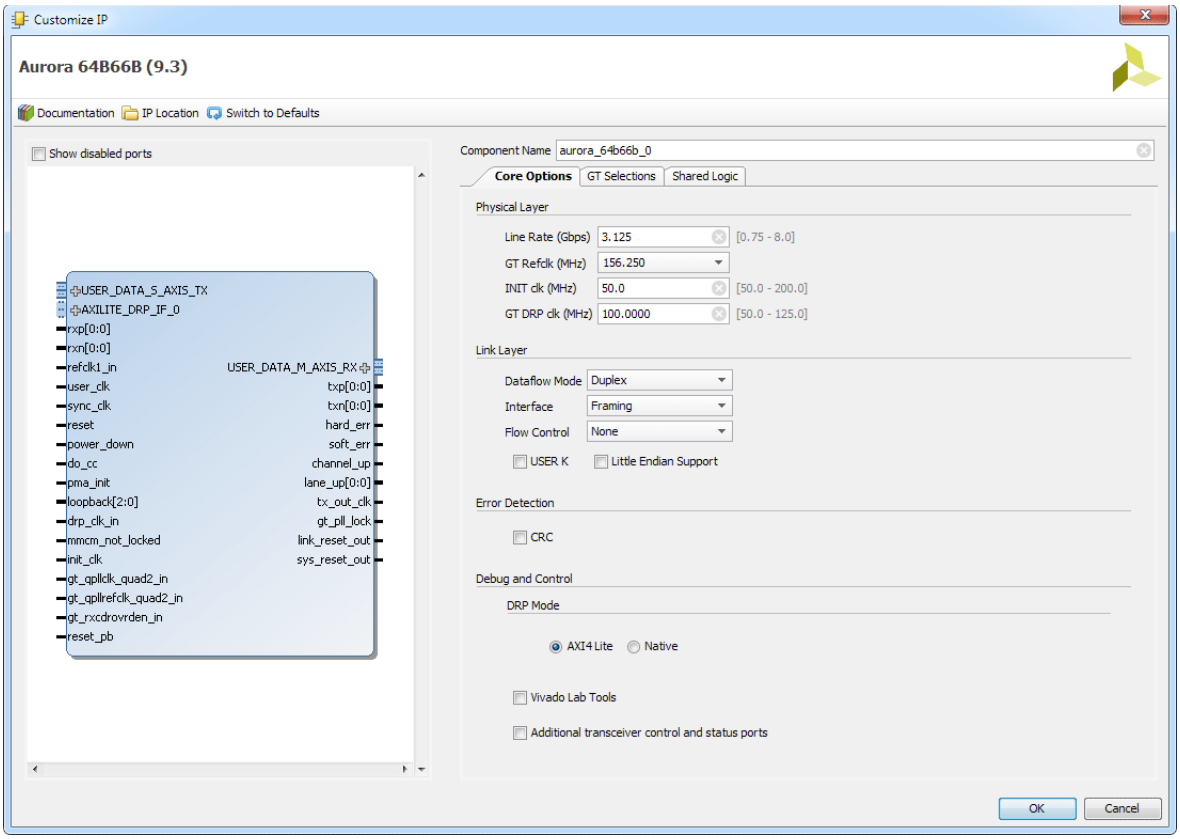


Figure 5-1: **Aurora 64B/66B IP Catalog Core Options Tab for 7 Series FPGAs**

Component Name

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

Default: aurora_64b66b_0

Line Rate

Enter the line rate value in gigabits per second. The value entered must be within the valid range shown. This value determines the unencoded bit rate at which data is transferred over the serial link. Calculations based on line rate use enhanced precision.

Default: 3.125 Gb/s for 7 series FPGA transceivers, 10.3125 Gb/s for UltraScale™ FPGA GTH transceivers

GT Reference Clock Frequency

Select a reference clock frequency in MHz from the list box which provides options based on the selected line rate. For best results, select the highest rate that can be practically applied to the reference clock input of the target device.

Default: 156.25 MHz

INIT clk (MHz)

Enter a valid INIT clock frequency in the text box.

Default: 50 MHz for 7 series FPGAs and Zynq® AP SoCs, line_rate/datapath_width for UltraScale™ devices.

DRP clk (MHz)

Enter a valid DRP clock frequency in the text box. Available only with 7 series FPGA transceivers.

Default: 100 MHz

Data Flow Mode

Select the options for the direction of the channel that the Aurora 64B/66B core supports. Simplex cores have a single, unidirectional serial port that connects to a complementary simplex core. Three simplex options are provided to select the channel direction supported by the core: **RX-only_Simplex**, **TX-only_Simplex** or **TX/RX_Simplex**.

Duplex or **TX/RX_Simplex** specify that the core has both TX and the corresponding RX communication channels.

Default: Duplex

Interface

Select the type of datapath interface used for the core. Select **Framing** to use a complete AXI4-Stream interface that allows encapsulation of data frames of any length. Select

Streaming to use a simple AXI4-Stream interface to stream data through the Aurora channel.

Default: Framing

Flow Control

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

Available options are:

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

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

Default: None

User K

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

Default: Unchecked

CRC

Select the option to insert CRC32 in the data stream.

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

DRP

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

Available options are:

- Native
- AXI4_Lite

Default: AXI4_Lite

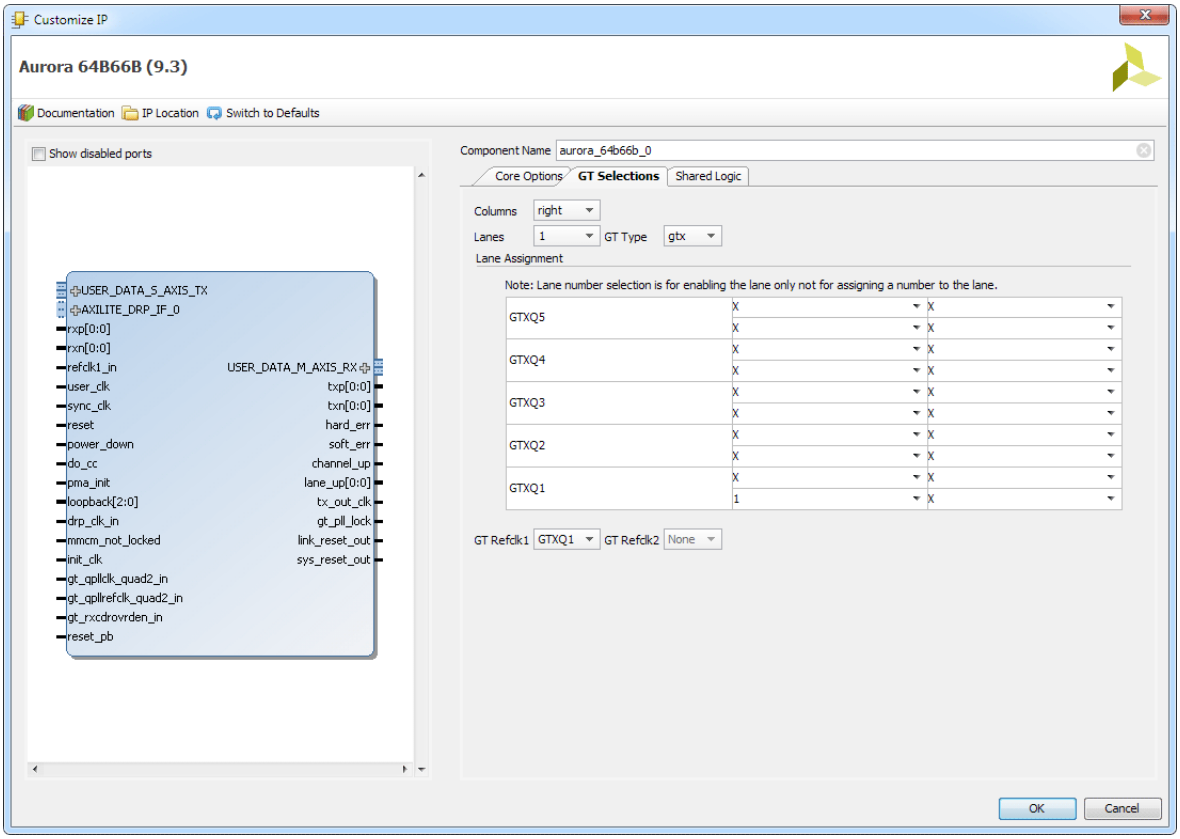


Figure 5-2: **Aurora 64B/66B IP Catalog GT Selections Tab for 7 Series FPGAs**

Columns

Select appropriate GT column from the drop-down list.

Default: left

Lanes

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

Default: 1

GT_TYPE

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

Available options are:

- gtx
- v7gth

Default: gtx

Lane Assignment

See the diagram in the information area in Figure 5-2. Each numbered row represents a serial transceiver tile and each active box represents an available GTX or GTH transceiver. For each Aurora lane in the core, starting with Lane 1, select a GTX or GTH transceiver and place the lane by selecting its number in the GTX or GTH transceiver placement box.

- X in the drop-down menu means that lane is not selected.
- <1—16> selected from the drop-down menu means that particular lane is selected. It does not assign that number to the physical lane.

RECOMMENDED: *Always select consecutive/physically adjacent lanes for a multi-GT design.*

Note: The core generates transceiver placement (LOC) constraints in ascending fashion. Move the cursor in the Vivado IDE to see the transceiver being selected in the 7 series and Zynq®-7000 family-based design. Lane numbering serves only to enable the lanes and not to assign numbers to the lanes. The Lane Assignment is not available for UltraScale™ architecture-based designs. It is strongly recommended that lane selection should be continuous for timing closure.

GT REFCLK1 and GT REFCLK2

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

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

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

Vivado Lab Tools

Select to add Vivado lab tools to the Aurora 64B/66B core. (See Using Vivado Lab Tools, page 91.) This option provides a debugging interface that shows the core status signals.

Default: Unchecked

Shared Logic

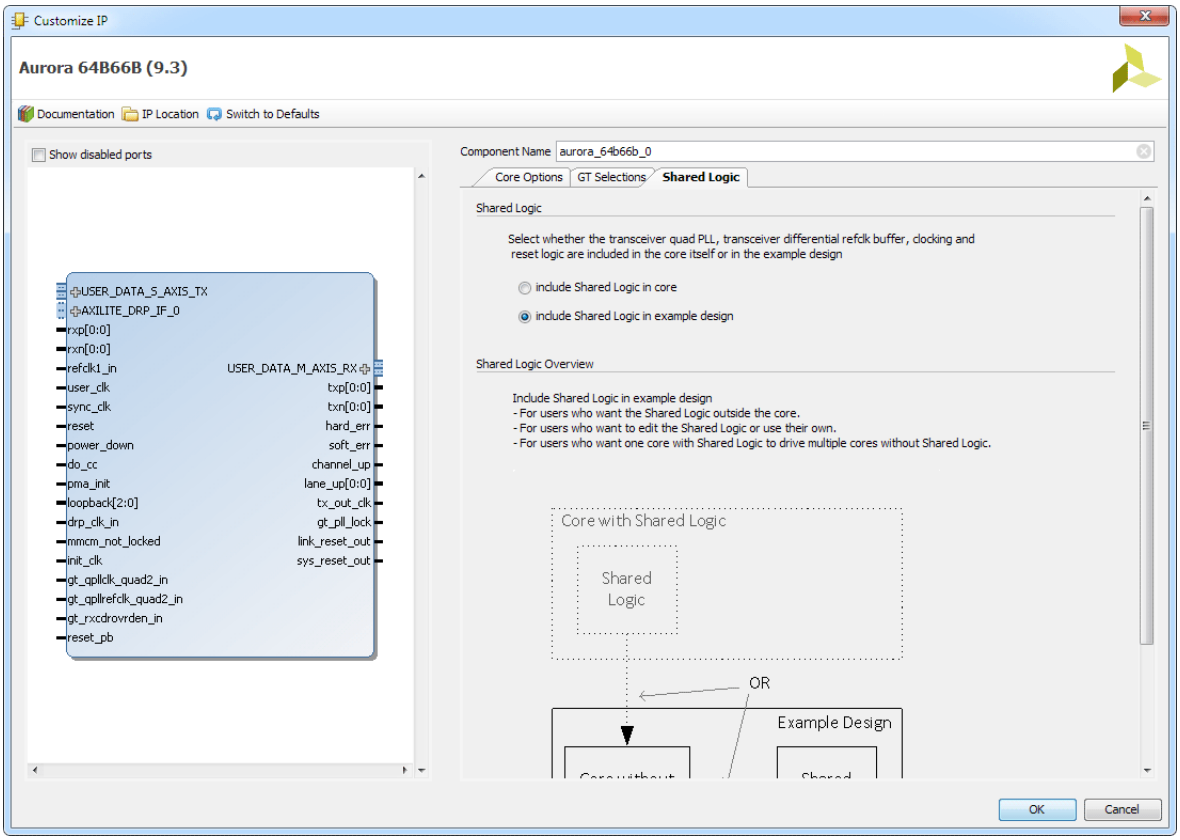


Figure 5-3: **Aurora 64B/66B IP Catalog Shared Logic Tab for 7 Series FPGAs**

Select to include transceiver common PLL and its logic 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

Additional Transceiver Control and Status Ports

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

Default: Unchecked

OK

Click **OK** to generate the core. (See Generating the Core, page 86.) The modules for the Aurora 64B/66B core are written to the IP catalog tool project directory using the same name as the top level of the core.

User Parameters

Table 5-1 shows the relationship between the fields in the Vivado IDE and the User Parameters in XCI files (which can be viewed in the Tcl console). Use the information in the tables for batch-driven Tcl flows to set GUI parameters and generate the Aurora 64B/66B core.

Table 5-1: **Vivado IDE Parameter to User Parameter Mapping**(1)

Vivado IDE Parameter/Value	User Parameter/Value	Default Value(1)
Core Options		
Physical Layer		
Line Rate (Gbps)	C_LINE_RATE	3.125/ 10.3125
GT Refclk (MHz)	C_REFCLK_FREQUENCY	156.250
INIT clk (MHz)	C_INIT_CLK	50.0/ 161.1328125
GT DRP clk (MHz)(9)	DRP_FREQ	100.0000

Table 5-1: **Vivado IDE Parameter to User Parameter Mapping[1] *(Cont'd)***

Vivado IDE Parameter/Value	User Parameter/Value	Default Value[1]
Link Layer		
Dataflow Mode	Dataflow_Config	Duplex
Interface	Interface_Mode	Framing
Flow Control	Flow_Mode	None
User K	C_USER_K	false
Little Endian Support	C_USE_BYTESWAP	false
Error Reduction		
CRC	CRC_MODE	NONE
DRP Mode		
AXI4 Lite (default mode)	drp_mode	AXI4_LITE
Native		
Vivado Lab Tools	C_USE_CHIPSCOPE	false
Additional transceiver control and status ports	TransceiverControl	false
GT Selections[9]		
Columns	C_COLUMN_USED	right[3]
Lanes[10]	C_AURORA_LANES	1
GT Type[10]	C_GT_TYPE	gtx[4]
Lane Assignment		
Select transceiver to include GTXE2_CHANNEL_X1Y4 in your design[5]	C_GT_LOC_5[6]	1
Select transceiver to include GTXE2_CHANNEL_X1Y5 in your design	C_GT_LOC_6	X
Select transceiver to include GTXE2_CHANNEL_X1Y5 in your design	C_GT_LOC_7	X
Select transceiver to include GTXE2_CHANNEL_X1Y7 in your design	C_GT_LOC_8	X
Select transceiver to include GTXE2_CHANNEL_X1Y8 in your design	C_GT_LOC_9	X
Select transceiver to include GTXE2_CHANNEL_X1Y9 in your design	C_GT_LOC_10	X
Select transceiver to include GTXE2_CHANNEL_X1Y10 in your design	C_GT_LOC_11	X
Select transceiver to include GTXE2_CHANNEL_X1Y11 in your design	C_GT_LOC_12	X
Select transceiver to include GTXE2_CHANNEL_X1Y12 in your design	C_GT_LOC_13	X
Select transceiver to include GTXE2_CHANNEL_X1Y13 in your design	C_GT_LOC_14	X
Select transceiver to include GTXE2_CHANNEL_X1Y14 in your design	C_GT_LOC_15	X
Select transceiver to include GTXE2_CHANNEL_X1Y15 in your design	C_GT_LOC_16	X
Select transceiver to include GTXE2_CHANNEL_X1Y16 in your design	C_GT_LOC_17	X
Select transceiver to include GTXE2_CHANNEL_X1Y17 in your design	C_GT_LOC_18	X
Select transceiver to include GTXE2_CHANNEL_X1Y18 in your design	C_GT_LOC_19	X
Select transceiver to include GTXE2_CHANNEL_X1Y19 in your design	C_GT_LOC_20	X

Table 5-1: **Vivado IDE Parameter to User Parameter Mapping(1) *(Cont'd)***

Vivado IDE Parameter/Value	User Parameter/Value	Default Value(1)
GT Refclk (MHz)		
GT Refclk1	C_GT_CLOCK_1	GTXQ1
GT Refclk2	C_GT_CLOCK_2	None
Shared Logic		
Include Shared Logic in core	SupportLevel(8)	0
Include Shared Logic in example design (default mode)		

Notes:

1. The values in this table reflect the default device (xc7vx485tffg1157-1). Default values for UltraScale architecture devices are denoted with a slash (/) where appropriate.
2. X0Y0 GT selection is based upon columns.
3. If a device has transceivers on both sides, *left* is the default value.
4. If 7 series device has GTX transceivers, *gtx* is the default value. If GTH transceivers, *v7gth* is the default value.
5. Numbering for default device starts from GTXE2_CHANNEL_X1Y4. Otherwise, numbering starts from GTXE2_CHANNEL_X0Y0.
6. C_GT_LOC_i where *i* varies from 1 to 48.
7. By default, the lowest *i* C_GT_LOC_i is assigned.
8. If Shared Logic in Core option is selected, SupportLevel is 1.
9. Not available in UltraScale devices.
10. The Lanes and GT Type options for UltraScale devices are available on the Core Options page in the Vivado IDE.

Transceiver Channel Locations

In UltraScale architecture transceiver implementations for the Aurora 64B/66B core, it is expected that the transceiver locations are consecutive. Based upon the number of lanes selected, and the targeted UltraScale device, the core provides consecutive transceiver channel locations which are, by default, set by the GT Wizard. In designs using the QPLL1 (line rate above 8.0 Gb/s), the GT common becomes part of the shared or non-shared logic choice for the core. The connection between the GT common and the transceiver channel is based upon the number of lanes (see Figure 5-4).

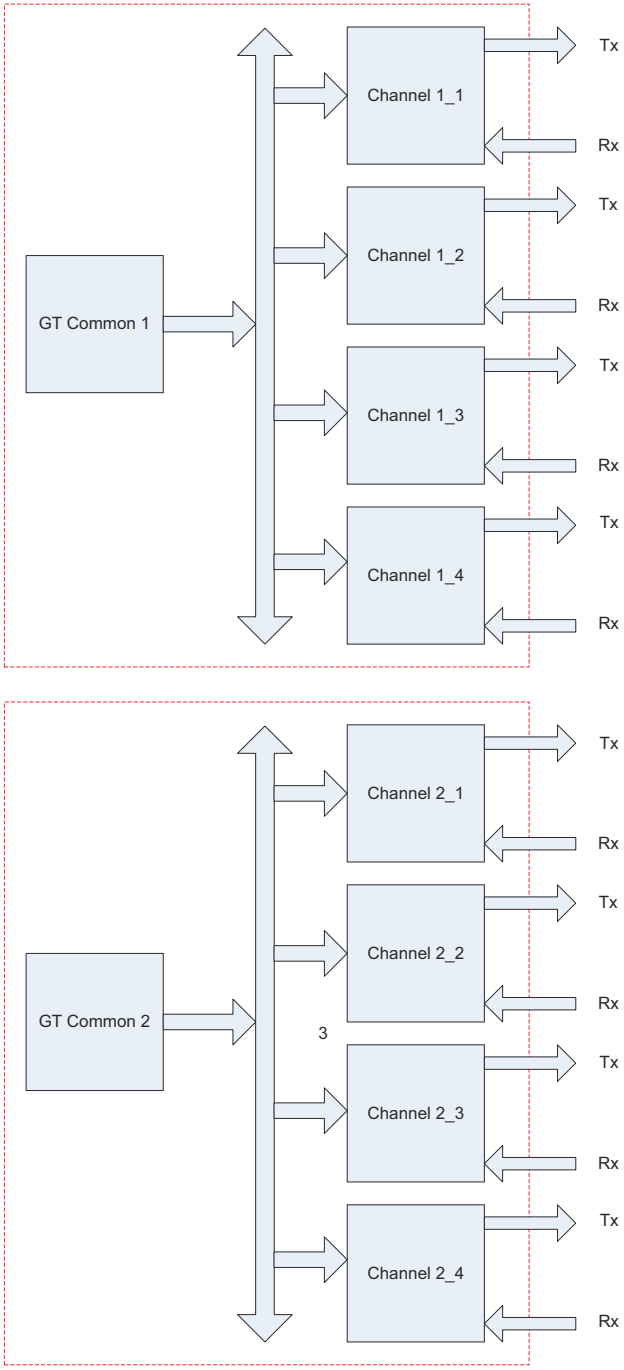


Figure 5-4: **Pictorial Representation of the UltraScale Architecture GT Common Interface with GT Channels (Aurora 64B/66B Configuration: 9 Gb/s and 8 lanes)**

The relative path to the file in which to change the transceiver locations for designs using UltraScale devices is:

```
<user_component_name>_example.srcs/sources_1/ip/<user_component_name>/ip_0/synth/
<user_component_name>_gt.xdc
```

RECOMMENDED: *Do not alter any default locations, unless otherwise absolutely needed after the design is generated, else the design functionality cannot not be guaranteed. In the case when the line rate is less than 8.0 Gb/s is chosen, the CPLL becomes part of GT Wizard hierarchical core.*

Output Generation

The customized Aurora 64B/66B core is delivered as a set of HDL source modules in Verilog. 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 64B/66B Project Directory Structure**

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

Constraining the Core

This section contains information about constraining the core in the Vivado Design Suite.

Device, Package, and Speed Grade Selections

Not Applicable

Clock Frequencies

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

- GT reference clock constraint

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

  Note: The GT reference clock location constraint should be added to the `<user_component_name>_example.srcs/constrs_1/imports/<user_component_name>_example.xdc` file.

- CORECLK clock constraint

  CORECLKs are the clock signals on which the core functions. CORECLKS such as USER_CLK and SYNC_CLK are derived from the TXOUTCLK signal which is generated by the transceiver based on the applied reference clock and the transceiver divider settings. The Aurora 64B/66B core calculates the USER_CLK/SYNC_CLK frequency based on the line rate and transceiver interface width. The `create_clock` XDC command is used to constrain all CORECLKs.

- INIT CLK constraint

  The Aurora 64B/66B example design uses a debounce circuit to sample PMA_INIT asynchronously clocked by the `init_clk` clock. The `create_clock` XDC command is used to constrain the `init_clk` clock.

RECOMMENDED: *It is recommended to have the system clock frequency lower than the transceiver reference clock frequency and in the range of 50 to 200 MHz for 7 series and Zynq devices. For UltraScale devices, the recommended range is 6.25 MHz to line_rate/64 or 200 MHz whichever is less.*

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 GT Selections tab 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 transceiver reference clock. The `set_property` XDC command is used to constrain the transceiver reference clock pins.

False Paths

The False Path constraint is defined on the first stage of the flip-flop of the CDC module.

Example Design

The generated example design including support logic has a 3.125 Gb/s line rate and a 156.25 MHz reference clock. The XDC file generated for the xc7vx485tffg1157-1, the default device, follows:

```
################################# CLOCK CONSTRAINTS ########################
# User Clock Contsraint: the value is selected based on the line rate of the module
#create_clock -name user_clk -period 20.480 [get_pins
<user_component_name>_block_i/clock_module_i/user_clk_net_i/I]

# SYNC Clock Constraint
#create_clock -name sync_clk -period 10.240 [get_pins
<user_component_name>_block_i/clock_module_i/sync_clock_net_i/I]

################################# IP LEVEL CONSTRAINTS START ################
# Following constraints present in <user_component_name>.xdc
# Create clock constraint for TXOUTCLK from GT
#create_clock -period 10.240 [get_pins -hier -filter
{name=~*<user_component_name>_wrapper_i*<user_component_name>_multi_gt_i*<user_component_name
>_gtx_inst/gtxe2_i/TXOUTCLK}]
# Create clock constraint for RXOUTCLK from GT
#create_clock -period 10.240 [get_pins -hier -filter
{name=~*<user_component_name>_wrapper_i*<user_component_name>_multi_gt_i*<user_component_name
>_gtx_inst/gtxe2_i/RXOUTCLK}]
################################# IP LEVEL CONSTRAINTS END ##################
# Reference clock contraint for GTX
create_clock -name GTXQ1_left_i -period 6.400 [get_ports GTXQ1_P]
  ### DRP Clock Constraint
create_clock -name drp_clk -period 10.000 [get_ports DRP_CLK_IN]

# 50MHz board Clock Constraint
create_clock -name init_clk -period 20.000 [get_ports INIT_CLK_P]
```

```
###### No cross clock domain analysis. Domains are not related ###############
## set_false_path -from [get_clocks init_clk] -to [get_clocks user_clk]
## set_false_path -from [get_clocks user_clk] -to [get_clocks init_clk]
## set_false_path -from [get_clocks init_clk] -to [get_clocks sync_clk]
## set_false_path -from [get_clocks sync_clk] -to [get_clocks init_clk]

## ## set_false_path -from init_clk -to [get_clocks -of_objects [get_pins
<user_component_name>_block_i/clock_module_i/mmcm_adv_inst/CLKOUT0]]
##
## ## set_false_path -from [get_clocks -of_objects [get_pins
<user_component_name>_block_i/clock_module_i/mmcm_adv_inst/CLKOUT0]] -to init_clk
##
## ## set_false_path -from init_clk -to [get_clocks -of_objects [get_pins
<user_component_name>_block_i/clock_module_i/mmcm_adv_inst/CLKOUT1]]
##
## ## set_false_path -from [get_clocks -of_objects [get_pins
<user_component_name>_block_i/clock_module_i/mmcm_adv_inst/CLKOUT1]] -to init_clk
##
## ## set_false_path -from [get_clocks -of_objects [get_pins
<user_component_name>_block_i/clock_module_i/initclk_bufg_i/O]] -to [get_clocks -of_objects
[get_pins -hier -filter
{name=~*<user_component_name>_i*inst*<user_component_name>_wrapper_i*<user_component_name>_mu
lti_gt_i*<user_component_name>_gtx_inst/gtxe2_i/RXOUTCLK}]]
##
## ## set_false_path -from [get_clocks -of_objects [get_pins -hier -filter
{name=~*<user_component_name>_i*inst*<user_component_name>_wrapper_i*<user_component_name>_mu
lti_gt_i*<user_component_name>_gtx_inst/gtxe2_i/RXOUTCLK}]] -to [get_clocks -of_objects
[get_pins <user_component_name>_block_i/clock_module_i/initclk_bufg_i/O]]
##
## ## set_false_path -from [get_clocks -of_objects [get_pins -hier -filter
{name=~*<user_component_name>_i*inst*<user_component_name>_wrapper_i*<user_component_name>_mu
lti_gt_i*<user_component_name>_gtx_inst/gtxe2_i/RXOUTCLK}]] -to [get_clocks -of_objects
[get_pins <user_component_name>_block_i/clock_module_i/mmcm_adv_inst/CLKOUT0]]
##
## ## set_false_path -from [get_clocks -of_objects [get_pins
<user_component_name>_block_i/clock_module_i/mmcm_adv_inst/CLKOUT0]] -to [get_clocks
-of_objects [get_pins -hier -filter
{name=~*<user_component_name>_i*inst*<user_component_name>_wrapper_i*<user_component_name>_mu
lti_gt_i*<user_component_name>_gtx_inst/gtxe2_i/RXOUTCLK}]]

## GT CLOCK Locations   ##
# Differential SMA Clock Connection
set_property LOC AH6 [get_ports GTXQ1_P]
set_property LOC AH5 [get_ports GTXQ1_N]

  set_property LOC GTXE2_CHANNEL_X1Y4 [get_cells
<user_component_name>_block_i/<user_component_name>_i/inst/<user_component_name>_wrapper_i/<u
ser_component_name>_multi_gt_i/<user_component_name>_gtx_inst/gtxe2_i]

# false path constraints to the example design logic
set_false_path -to [get_pins -hier *<user_component_name>_cdc_to*/D]

##############################################################################################
########
##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 PMA_INIT]

  #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 DATA_ERR_COUNT[0]]
  #set_property LOC Y19 [get_ports DATA_ERR_COUNT[1]]
  #set_property LOC Y18 [get_ports DATA_ERR_COUNT[2]]
  #set_property LOC AA18 [get_ports DATA_ERR_COUNT[3]]
  #set_property LOC AB18 [get_ports DATA_ERR_COUNT[4]]
  #set_property LOC AB19 [get_ports DATA_ERR_COUNT[5]]
  #set_property LOC AC19 [get_ports DATA_ERR_COUNT[6]]
  #set_property LOC AB17 [get_ports DATA_ERR_COUNT[7]]

  #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 PMA_INIT]

  #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 DATA_ERR_COUNT[0]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[1]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[2]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[3]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[4]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[5]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[6]]
  #set_property IOSTANDARD LVCMOS18 [get_ports DATA_ERR_COUNT[7]]

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

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

Simulation

This section contains information about simulating in the Vivado Design Suite. For details, see the *Vivado Design Suite User Guide - Logic Simulation* (UG900) [Ref 9].

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

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

Simulating the Duplex core is a single-step process after generating the 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 an

example design of a Simplex core takes more time than the Duplex example design generation.

Simulation speed up:

The C_EXAMPLE_SIMULATION parameter has been introduced to speed up post synthesis/implementation netlist functional simulations.

1. If core generation is through batch mode, include this command as part of the core generation:

   ```
   set c_example_simulation true
   ```

2. Run the Tcl command to speed up simulation. The generated core with the preceding command is *only* for simulation.
3. If core generation is through the Vivado IDE, change the EXAMPLE_SIMULATION parameter in the generated RTL code to 1 in these files to speed up simulation:
   - `<USER_COMPONENT_NAME>_exdes.v`
   - `<USER_COMPONENT_NAME>_core.v`

Synthesis and Implementation

This section contains information about synthesis and implementation in the Vivado Design Suite.

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

Implementation

The quick start example consists of the following components:

Overview

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

The Aurora 64B/66B example design has been tested with the Vivado Design Suite for synthesis and Mentor Graphics QuestaSim for simulation.

Generating the Core

To generate an Aurora 64B/66B core with default values using the Vivado design tools:

1. Launch the Vivado design tools. For help starting and using the Vivado design tools, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7].
2. Under Quick Start, click **Create New Project** and click **Next.**
3. Enter the new project name and the project location, then Click **Next**.
4. Select **RTL Project** and click **Next**.
5. Click **Next** to accept the default part number (xc7vx485tffg1157–1).
6. Click **Finish**.
7. After creating the project, click **IP catalog** in the Project Manager panel.
8. Locate the Aurora 64B/66B core in the IP catalog.
9. Double-click the core name.
10. Click **OK**.
11. Click **Generate**.

For more detailed information on generating an Aurora 64B/66B core, see *Designing a System Using the Aurora 64B66B Core (Duplex) on the KC705 Evaluation Kit* (XAPP1192) [Ref 24]

Implementing the Example Design

The example design must be generated from the IP core.

1. Under Sources, Right-click the generated Design Sources file and click **Open IP Example Design...**
2. Enter the path to the directory in which to create the example design and click **OK**.
3. To run synthesis followed by implementation, in the Flow Navigator panel under Implementation, click **Run Implementation**.
4. To generate a bitstream, under Program and Debug, click **Generate Bitstream**.

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

For details about synthesis and implementation, see "Synthesizing IP" and "Implementing IP" in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 7].

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 80 for the directory structure and file contents of the example design.

Quick Start Example Design

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

The quick start example design consists of these components:

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

Detailed Example Design

Each Aurora 64B/66B core includes an example design (`<user_component_name>_exdes`) that uses the core in a simple data transfer system. For more details about the `example_design` directory, see Output Generation in Chapter 5.

The example design based on the selected configurations consists of the following:

- Frame generator (FRAME_GEN) connected to the TX interface
- Frame checked (FRAME_CHECK) connected to the RX user interface
- VIO/ILA instance for debug and testing
- Hardware-based reset fsm to perform repeat reset and channel integrity testing (only for duplex mode)

Figure 6-1 shows a block diagram of the example design for a full-duplex core. Table 6-1, page 89 describes the ports of the example design.

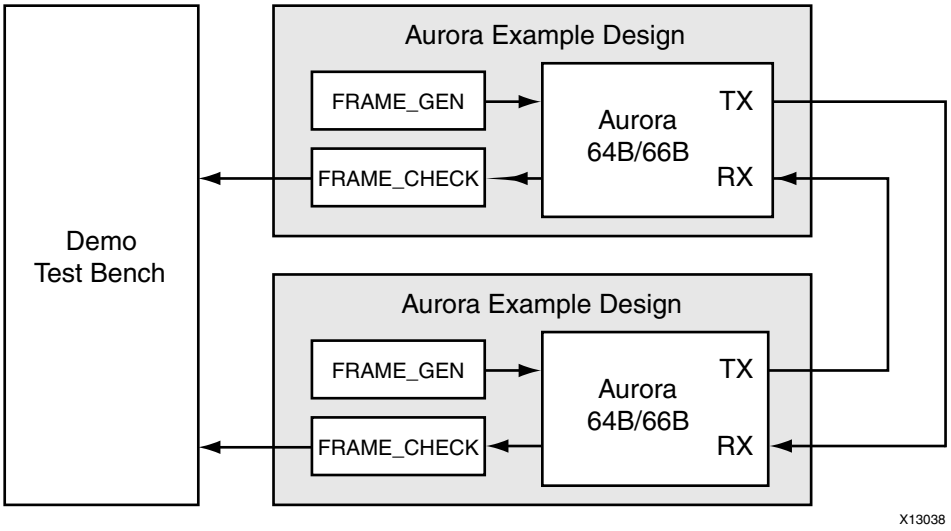


Figure 6-1: **Example Design**

The example design uses all the interfaces of the core. There are separate AXI4-Stream interfaces for optional flow control. Simplex cores without a TX or RX interface have no FRAME_GEN or FRAME_CHECK block, respectively.

FRAME_GEN

This module is used to generate user traffic to each of the PDU, UFC, NFC and USER_K interfaces. The module contains a pseudo-random data generated using a linear feedback shift register (LFSR) with a known seed value. The FRAME_CHECK module can use the same configuration to check the data integrity of the aurora channel. The inputs for this module are user_clk, reset and channel_up. The FRAME_GEN module follows the AXI4-Stream protocol and transmits the user-traffic.

FRAME_CHECK

This module is used to check the RX data integrity. As the FRAME_GEN module uses an LFSR with a known seed value, the same LFSR and seed value are used in the FRAME_CHECK module to compute the expected frame RX data. The received user data is validated against the expected data and any errors are reported as per the AXI4-Stream protocol. The FRAME_CHECK module is applicable to the PDU, UFC, NFC and USER_K interfaces.

The design can also be used as a reference for connecting the trickier interfaces on the Aurora 64B/66B core, such as the clocking interface.

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

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

Port	Direction	Clock Domain	Description
rxn[0:*m*–1]	Input	Serial Clock	Negative differential serial data input pin.
rxp[0:*m*–1]	Input	Serial Clock	Positive differential serial data input pin.
txn[0:*m*–1]	Output	Serial Clock	Negative differential serial data output pin.
txp[0:*m*–1]	Output	Serial Clock	Positive differential serial data output pin.
reset	Input	user_clk	Reset signal for the example design.
<reference clock(s)>	Input	user_clk	The reference clocks for the Aurora 64B/66B core are brought to the top level of the example design. See Clock Interface and Clocking in Chapter 3 for details about the reference clocks.
<core error signals>(1)	Output	user_clk	The error signals from the Aurora 64B/66B core Status and Control interface are brought to the top level of the example design and registered.
<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.

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

Port	Direction	Clock Domain	Description
<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 GTX and GTH transceiver they use.
pma_init	Input	init_clk	The reset signal for the PCS and PMA modules in the GTX and GTH transceivers is connected to the top level through a debouncer. The signal is debounced using the `init_clk`. See the Reset section in the *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476) [Ref 4] for further details on GT RESET.
init_clk_p/ init_clk_n	Input	-	The `init_clk` signal is used to register and debounce the PMA_INIT signal. The `init_clk` signal must not come from a GTX or GTH transceiver, and should be set to a low rate, preferably lower than the reference clock. The `init_clk` signal is single-ended for UltraScale™ devices.
data_err_count[0:7]	Output	user_clk	Count of the number of frame data words received by the FRAME_CHECK that did not match the expected value.
ufc_err	Output	user_clk	Asserted (active-High) when UFC data words received by the FRAME_CHECK that did not match the expected value.
user_k_err	Output	user_clk	Asserted (active-High) when User K data words received by the FRAME_CHECK that did not match the expected value.

Notes:
1. See Status, Control, and the Transceiver Interface in Chapter 2 for details.

Using Vivado Lab Tools

The ILA and VIO cores aid in debugging and validating the design in the board and are provided with the Aurora 64B/66B core. The Aurora 64B/66B core connects the relevant signals to the VIO to facilitate easier bring-up or debug of the design. Select the Vivado lab tools option from the Core Options tab in the Vivado Integrated Design Environment (IDE) (see Figure 5-1, page 70) to include it as a part of the example design.

Cores generated with the Vivado lab tools option enabled have three VIO interfaces and one ILA interface.

- vio1_inst – contains core Lane Up, Channel Up, Data Error count, Soft Error count, Channel Up transition count along with System Reset, GT Reset and Loopback ports
- vio2_inst – contains status of reset quality counters
- vio3_inst – contains test pass/fail status for repeat reset test

Implementing the Example Design

The example design must be generated from the IP core. See Implementing the Example Design, page 86.

Hardware Reset FSM in the Example Design

The Aurora 64B/66B core example design for duplex mode incorporates a hardware reset FSM to perform repeated resets and monitoring robustness of the link. This FSM also contains an option to set different time periods between reset assertions. Also continuous `channel_up` and `link_reset` transition counters are monitored and the test status is reported through VIO.

The following signals are added in to the default ILA and VIOs for probing the link:

i_ila:

- `tx_d_i[0:15]`: TX Data from the LocalLink Frame Gen module
- `rx_d_i[0:15]`: RX Data to the LocalLink Frame check module
- `data_err_count_o`: 8-bit Data error count value, it is expected to be 'd0 in normal operations
- `lane_up_vio_usrclk`: `lane_up` signal

- `channel_up_i`: `channel_up` signal
- `soft_err_i`: Soft error monitor
- `hard_err_i`: Hard error monitor

vio1_inst:

- `sysreset_from_vio_i`: `reset` input to example design
- `gtreset_from_vio_i`: `pma_init` to example design
- `vio_probe_in2`: Quality counters for Link status
- `rx_cdrovrden_i`: Used while enabling loopback mode
- `loopback_i`: Used while enabling loopback mode

vio2_inst:

- `reset_quality_cntrs`: Used to reset all the quality counters in the example design
- `reset_test_fsm_from_vio`: Used to reset the hardware reset test FSM
- `reset_test_enable_from_vio`: Used to enable/start the repeat reset test from the vio ports on the hardware.
- `iteraion_cnt_sel_from_vio`: Number of repeat reset iterations to be initiated. This is a 4-bit encoded value for a fixed number of iterations that can be seen in the example design when Vivado lab tools are enabled.
- `lnk_reset_in_initclk`: Input probe to monitor the assertion of `link_reset`
- `soft_err_in_initclk`: Input probe to monitor the `soft_err` status
- `chan_up_transcnt_20bit_i [15:8]`: Number of `channel_up` transaction counts; this can be used to monitor the number of reset iterations that have been completed.

  Note:

  a. `chan_up_transcnt_20bit_i` is probed only [15:8] bits; hence, this probe takes some time to update the status.

  b. To change the number of reset iterations, modify the respective value for `iteration_cnt_sel_from_vio` and correspondingly select `chan_up_transcnt_20bit_i` to probe the status.

vio3_inst:

- `test_passed_r`: Test pass status is asserted after the respective iteration count if resets are done successfully.
- `test_failed_r`: Test fail status is asserted if there is either a lack of `channel_up` or some data errors have occurred.
- `lnkrst_cnt_20bit_vio_i`: Probe to monitor the number of times the `link_reset` is asserted.
- `reset_test_fsm_chk_time_sel`: 3-bit encoded value probe to select the hardware `reset_fsm` check time for `channel_up` assertions after reset is deasserted.

Hardware FSM Operation:

In the example design (`<user_component_name>_exdes.v`), a hardware initiated repeat reset FSM has been added to test the robustness of the link when subject to repeat reset. The FSM consists of IDLE, ASSERT_RST, DASSERT_RST, WAIT, WAIT1, CHECK, FAIL and DONE states.

1. In IDLE state, `test_passed_r` indicates reset test passed, `test_failed_r` indicates reset test fail, and `timer_r` provides an iteration count of resets. Defaults to 0.
2. When the `reset_test_enable_from_vio` signal is asserted, the hardware FSM traverses to the ASSERT_RST state where `pma_init` is asserted for a pre-determined time (28-bit count time).
3. This `pma_init` assertion ensures that a hot plug sequence is detected by the link partner. The hardware FSM then traverses to the DEASSERT_RST state where the `pma_init` is deasserted and the timer is loaded with a default value that can be configured using the `reset_test_fsm_chk_time_sel` vio signal.
4. The FSM then moves to the WAIT state until the selected time has expired. In this state, all checks such as for data errors and soft error occurrences are performed and the `channel_up` signal is verified to be asserted High and not toggled more than once for this iteration of `pma_init`.
5. If this condition is not met, the FSM moves to FAIL state and the repeat reset run is stopped. Otherwise, the FSM moves to WAIT1 state where a few data packets are transmitted and received.
6. The FSM then moves to the CHECK state, in which the `channel_up` transitions are checked again. If there is not more than one transition, the FSM returns to the IDLE state until the requested iterations are completed. This ensures that the link is robust and recovers reliably across multiple repeat resets of the link.

Test Bench

The Aurora 64B/66B core delivers a demonstration test bench for the example design. This chapter describes the Aurora test bench and its functionality. The test bench consist of the following modules:

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

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

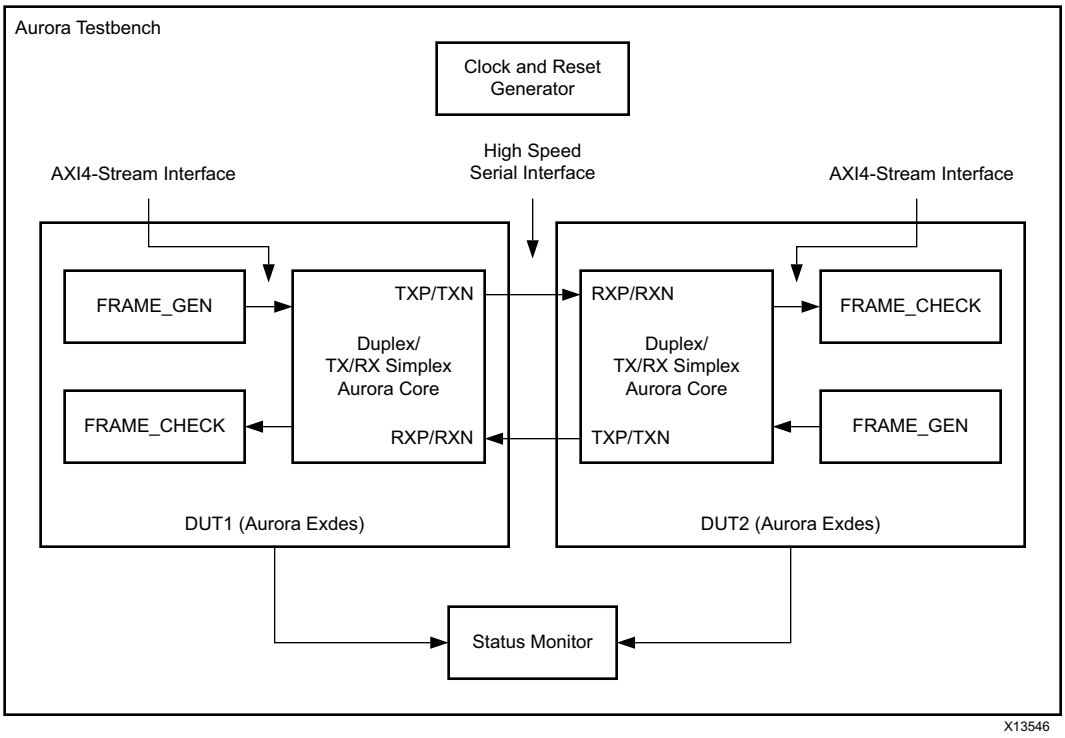


Figure 7-1: **Aurora Test Bench for Duplex Configuration**

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

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

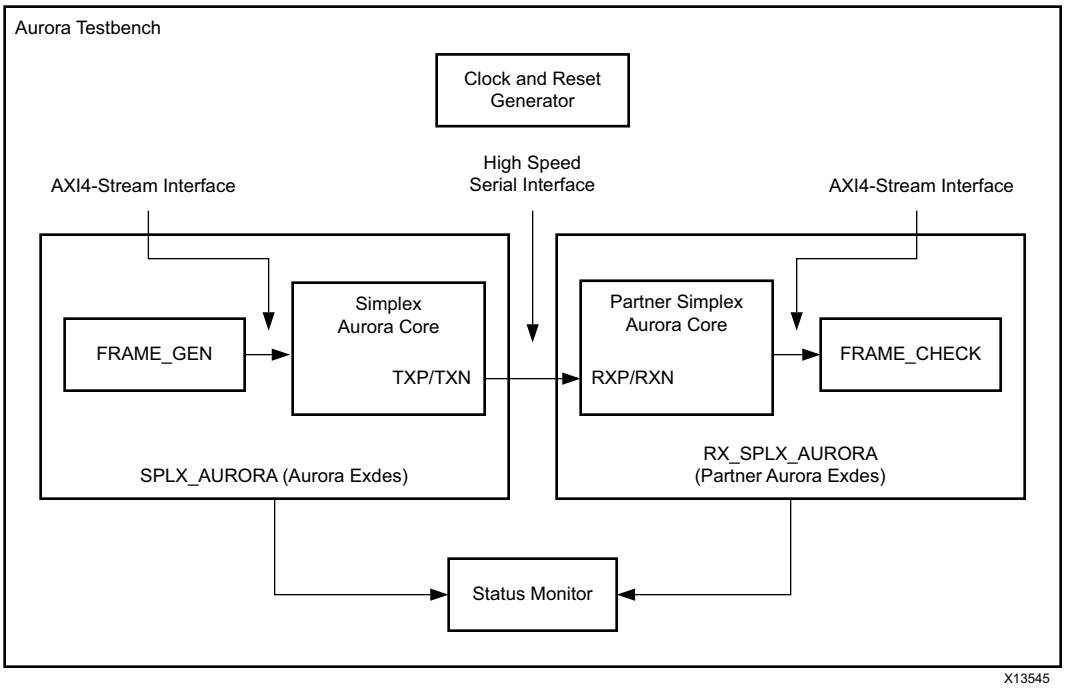


Figure 7-2: **Aurora Test Bench for Simplex Configuration**

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

Verification, Compliance, and Interoperability

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

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

Aurora 64B/66B cores are tested in hardware for functionality, performance, and reliability using Xilinx® evaluation boards. Aurora verification test suites for all possible modules are continuously being updated to increase test coverage across the range of possible parameters for each individual module.

A series of test scenarios are validated using various Xilinx development boards which are listed in Table A-1. These boards can be used to prototype system designs and the core can be used to communicate with other systems.

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

Target Device	Evaluation Boards	Characterization boards
7 series FPGAs	KC705, VC707, VC709, ZC706	KC724, VC7203, ZC723, VC7215
UltraScale™ architecture	VCU107, KCU105	UC1250, UC1283

To achieve interoperability among different versions of Aurora 64B/66B cores for 7 series FPGA transceivers, a user-level parameter is provided which must be set to achieve proper interoperability between cores as shown in Table A-2.

Table A-2: **Aurora 64B/66B Interoperability**

2014.1 (7 series FPGAs) Interoperability with 2013.2 (7 series FPGAs) of Aurora 64B/66B		
2014.1\2013.2	**2013.2 (7 series) GTX Transceivers**	**2013.2 (7 series) GTH Transceivers**
2014.1 (7 series) GTX transceivers	√	√
2014.1 (7 series) GTH transceivers	√	√
2014.1 (7 series) Interoperability with ISE 14.7 (6 series) of Aurora 64B/66B		
2014.1\ISE 14.7	**ISE 14.7 (6 series) GTX Transceivers**	**ISE 14.7 (6 series) GTH Transceivers**
2014.1 (7 series) GTX transceivers	√	x
2014.1 (7 series) GTH transceivers	√	x

To handle backward compatibility with earlier core versions, three parameters, BACKWARD_COMP_MODE1, BACKWARD_COMP_MODE2 and BACKWARD_COMP_MODE3 are included in the `<user_component_name>_core.v` module. These parameters allow 2014.1 (7 series FPGAs) core versions to provide the characteristics and functionality of previous versions of the core. These parameters were created to conveniently handle the condition where updates to the previous core versions are not practical. Hence, the overall stability of the linked system (new <-> old) is equivalent to the stability of links achievable between previous core versions (old <-> old) as shown in Table A-2.

BACKWARD_COMP_MODE1 /BACKWARD_COMP_MODE2

- Default value is 0. This ensures compatibility between v9.2 core and v9.1 core and between v9.2 core and v9.0 core.
- Set both these parameters to 1 to make the v9.2 core compatible with the v8.1 core or with the v7.3 core.

BACKWARD_COMP_MODE3

- Default value is 0. Set this parameter to 1 (from v9.3 core) if the core needs to clear the hot plug counter on reception of any valid BTF. When this parameter is 0, the hot-plug counter is only cleared by reception of CC blocks.

Migrating and Upgrading

This appendix contains information about migrating a design from ISE® to the Vivado® Design Suite, upgrading to a more recent version of the IP core, and migrating legacy (LocalLink based) Aurora 64B/66B Cores to the AXI4-Stream Aurora 64B/66B 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.

Device Migration

If migrating from a 7 series device with GTX or GTH transceivers to an UltraScale™ device with GTH transceivers, the prefixes of the optional transceiver debug ports for single-lane cores are changed from "`gt0`", "`gt1`" to "`gt`", and the suffix "`_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 is true for all ports, with the exception of the DRP buses which follow the convention of `gt(n)_drpxyz`.

IMPORTANT: *It is important that designs are updated to use the new transceiver debug port names. For more information about migration to UltraScale devices, see the UltraScale Architecture Migration Methodology Guide (UG1026) [Ref 12]*

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 13].

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the Vivado Design Suite.

In the latest revision of the core, there have been several changes which make the core pin-incompatible with the previous version (s). 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.

RECOMMENDED: *If upgrading to a later core 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.*

Updates from v9.2 Core

Table B-1 explains the ports added in v9.3 of the Aurora 64B/66B core and provides guidance on the impact of these port additions on designs using pre v9.3 cores.

Table B-1: **New Ports Added to Aurora 64B/66B in 2014.3**

Port	Direction	Clock domain	Description
s_axi_bresp(1)(2)	Output	drp_clk	Added for AXI4-Lite protocol compliance.
s_axi_rresp(1)(2)	Output	drp_clk	
s_axi_wstrb(1)(2)	Input	drp_clk	
tx_reset/ tx_reset_pb	Input	user_clk	TX and RX logic separately reset in TX/RX-Simplex.
rx_reset/ rx_reset_pb	Input	user_clk	
tx_sys_reset_out	Output	user_clk	
rx_sys_reset_out	Output	user_clk	

Notes:

1. Refer to *Vivado AXI Reference Guide* (UG1037) [Ref 18] for details.
2. In UltraScale devices, all DRP and AXI4-LITE ports are sampled on `init_clk`.
3. All AXI4-LITE DRP ports are on per-lane basis.

When IP is upgraded, critical warnings occur due to these port additions. The DRP port additions are for protocol compliance and do not interfere with basic functionality. The resets are split in TX/RX-Simplex mode to promote ease of use.

Updates from v9.0 Core

- In the TX Startup FSM, the prior counting mechanism for `mmcm_lock_count` was based on `txuserclk`. Limitations resulted because this was a recovered clock. `stable_clock` is now used for the MMCM Lock synchronization.
- The RX datapath is now 32 bits up to the CBCC module, thus avoiding width conversion logic and `clk_en` generation. These functions are handled in the CBCC module before writing data to the FIFO.
- Lane skew tolerance is enhanced. The core is now able to tolerate more lane-to-lane skew.
- Logic added to detect Polarity inversion and to invert polarity while lane init is enabled.
- The core internally generates `tx_channel_up` for Aurora TX logic and `rx_channel_up` for Aurora RX logic. This action ensures that the RX logic is active and ready to receive before the TX logic begins sending. `rx_channel_up` is presented as `channel_up`.
- Reset and controls are common across all lanes.
- The RX CDR lock time was increased from 50 KUI to 37 MUI as suggested by the transceiver user guide.
- The Block Sync header max count was increased from 64 to 60,000 to improve the robustness of the link.
- Allowed transmission of more idle characters during channel initialization to improve robustness of the link.
- Removed the scrambler reset making it free running to achieve faster CDR lock. The default pattern sent by the scrambler is the scrambled value of NA idle character.
- Updated the GTH transceiver QPLL attributes - See AR 56332.
- Added shared logic and optional transceiver control and status debug ports.
- Updated clock domain crossing synchronizers to increase Mean Time Between Failures (MTBF) from meta-stability. Currently using a common synchronizer module and applying false path constraints only for the first stage of the flops.
- Added support for Cadence IES and Synopsys VCS simulators.
- Added Vivado lab tools support for debug.
- Added quality counters in the example design to increase the test quality.
- Added a hardware reset state machine in the example design to perform repeat reset testing.

Migrating Legacy (LocalLink based) Aurora Cores to the AXI4-Stream Aurora Core

Prerequisites

- Vivado design tools build containing the Aurora 64B/66B v9.x core supporting the AXI4-Stream protocol
- Familiarity with the Aurora directory structure
- Familiarity with running the Aurora example design
- Basic knowledge of the AXI4-Stream and LocalLink protocols
- Latest product guide (PG074) of the core with the AXI4-Stream updates
- Legacy documents: *LogiCORE IP Aurora 64B/66B v4.2 Data Sheet* (DS528) [Ref 14], *LogiCORE IP Aurora 64B/66B v4.1 Getting Started Guide* (UG238) [Ref 15], and *LogiCORE IP Aurora 64B/66B v4.2 User Guide* (UG237) [Ref 16] for reference.
- Migration guide (this Appendix)

Overview of Major Changes

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

- The user interface is modified from the legacy LocalLink (LL) to AXI4-Stream.
- All AXI4-Stream signals are active-High, whereas LocalLink signals are active-Low.
- The user interface in the example design and design top file is AXI4-Stream.

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

Block Diagrams

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

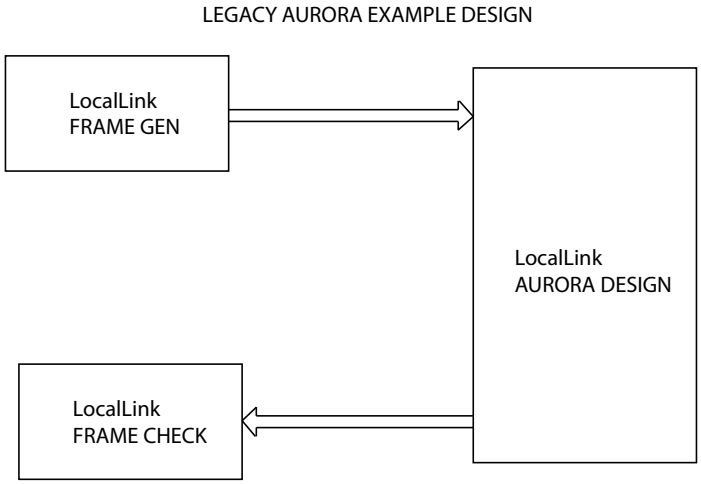


Figure B-1: **Legacy Aurora Example Design**

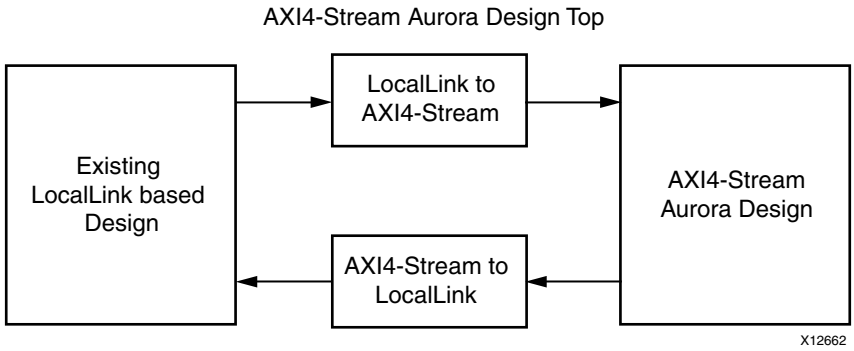


Figure B-2: **AXI4-Stream Aurora Example Design**

Signal Changes

Table B-2: **Interface Changes**

LocalLink Name	AXI4-S Name	Difference
TX_D	s_axi_tx_tdata	Name change only
TX_REM	s_axi_tx_tkeep	Name change. For functional differences, see Table 2-4, page 14
TX_SOF_N		Generated Internally
TX_EOF_N	s_axi_tx_tlast	Name change; Polarity
TX_SRC_RDY_N	s_axi_tx_tvalid	Name change; Polarity
TX_DST_RDY_N	s_axi_tx_tready	Name change; Polarity
UFC_TX_REQ_N	ufc_tx_req	Name change; Polarity
UFC_TX_MS	ufc_tx_ms	No Change
UFC_TX_D	s_axi_ufc_tx_tdata	Name change only
UFC_TX_SRC_RDY_N	s_axi_ufc_tx_tvalid	Name change; Polarity
UFC_TX_DST_RDY_N	s_axi_ufc_tx_tready	Name change; Polarity
NFC_TX_REQ_N	s_axi_nfc_tx_tvalid	Name change; Polarity
NFC_TX_ACK_N	s_axi_nfc_tx_tready	Name change; Polarity
NFC_PAUSE	s_axi_nfc_tx_tdata	Name change. For signal mapping, see Table 2-10, page 28
NFC_XOFF		
USER_K_DATA	s_axi_user_k_tdata	Name change. For signal mapping, see Table 2-12, page 36
USER_K_BLK_NO		
USER_K_TX_SRC_RDY_N	s_axi_user_k_tx_tvalid	Name change; Polarity
USER_K_TX_DST_RDY_N	s_axi_user_k_tx_tready	Name change; Polarity
RX_D	m_axi_rx_tdata	Name change only
RX_REM	m_axi_rx_tkeep	Name change. For functional difference, see Table 2-4, page 14

Table B-2: **Interface Changes *(Cont'd)***

LocalLink Name	AXI4-S Name	Difference
RX_SOF_N		Removed
RX_EOF_N	m_axi_rx_tlast	Name change; Polarity
RX_SRC_RDY_N	m_axi_rx_tvalid	Name change; Polarity
UFC_RX_DATA	m_axi_ufc_rx_tdata	Name change only
UFC_RX_REM	m_axi_ufc_rx_tkeep	Name change For functional difference, see Table 2-11, page 31
UFC_RX_SOF_N		Removed
UFC_RX_EOF_N	m_axi_ufc_rx_tlast	Name change; Polarity
UFC_RX_SRC_RDY_N	m_axi_ufc_rx_tvalid	Name change; Polarity
RX_USER_K_DATA	m_axi_rx_user_k_tdata	Name change For functional difference, see Table 2-12, page 36
RX_USER_K_BLK_NO		
RX_USER_K_SRC_RDY_N	m_axi_rx_user_k_tvalid	Name change; Polarity

Migration Steps

Generate an AXI4-Stream Aurora core from the Vivado design tools.

Simulate the Core

1. Run simulation from the Vivado IDE. Select simulation type to launch.
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. Frame checker continuously checks the received data and reports for any data mismatch.
5. A 'TEST PASS' or 'TEST FAIL' status is printed on the QuestaSim console providing the status of the test.

Implement the Core

1. Click **Run Implementation** to run synthesis and implementation consecutively.

Integrate to an Existing LocalLink-based Aurora Design

1. The Aurora core provides a lightweight 'shim' to interface to any existing LL based interface. The shims are delivered along with the core from the aurora_64b66b_v8_0 version of the core.
2. See Figure B-2, page 103 for the emulation of a LL Aurora core from a AXI4-Stream Aurora core.
3. Two shims `<user_component_name>_ll_to_axi.v` and `<user_component_name>_axi_to_ll.v` are provided in the `src` directory of the AXI4-Stream Aurora core.
4. Instantiate both the shims along with `<user_component_name>.v` in the existing LL based design top.
5. Connect the shim and AXI4-Stream Aurora design as shown in Figure B-2, page 103.
6. The latest AXI4-Stream Aurora core is now usable in any existing LL design environment.

Vivado IDE Changes

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

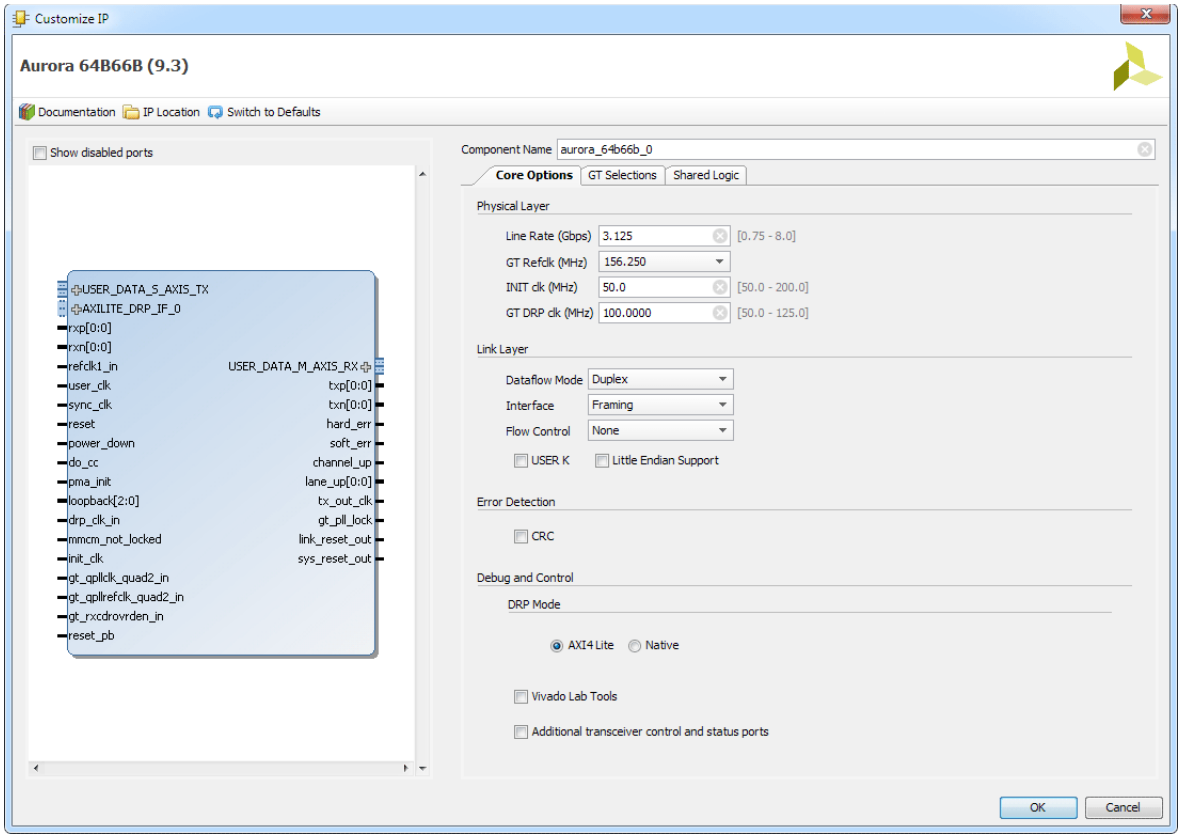


Figure B-3: **AXI4-Stream Signals**

Limitations

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

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

Limitation 1:

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

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

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

Limitation 2:

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

Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

Finding Help on Xilinx.com

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

Documentation

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

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

Answer Records

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

Answer Records for this core 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 64B/66B core, search for the phrase "Aurora 64B66B"

Master Answer Record for the Aurora 64B/66B Core

AR: 54368

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

Contacting 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 Xilinx 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 Xilinx Design Tools and simulator software versions.
- The XCI file created during Aurora 64B/66B core generation
- Additional files based on the specific issue might also be required. See the relevant sections in this debug guide for guidelines about which file(s) to include with the WebCase.

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

Simulation Debug

Lanes and Channel Do Not Come Up in Simulation

- The quickest way to debug this issue is to view the signals from one of the GTX or GTH transceiver instances that is not working
- Make sure that the reference clock and user clocks are all toggling

  Note: Only one of the reference clocks should be toggling, The rest are tied Low.
- Check to see that `recclk` and `txoutclk` are toggling. If they are not toggling, it might be necessary to wait longer for the PMA to finish locking. Wait for lane up and channel up. It might be necessary to wait longer for simplex/7 series FPGA designs.
- Make sure that `txn` and `txp` are toggling. If they are not, make sure to wait long enough (see the previous bulleted item) and make sure that another signal is not driving the `txn/txp` signal.
- Check in the <user_component_name>_support module whether the `pll/mmcm_not_locked` signal and the `reset` signals are present in the design. If these are being held active, the Aurora module cannot initialize.
- Be sure the `power_down` signal is not being asserted
- Make sure the `txn` and `txp` signals from each GTX or GTH transceiver are connected to the appropriate `rxn` and `rxp` signals from the corresponding GTX or GTH transceiver on the other side of the channel
- Instantiate the "glbl" module and use it to drive the `power_up` reset at the beginning of the simulation to simulate the reset that occurs after configuration. Hold this reset for a few cycles. The following code 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 transceivers on each side of the channel are connected in the correct order

Channel Comes Up in Simulation But s_axi_tx_tready is Never Asserted (Never Goes High)

- If the module includes flow control but it is not being used, make sure the request signals are not currently driven High. `s_axi_nfc_tx_tvalid` and `ufc_tx_req` are active-High: if they are High, `s_axi_tx_tready` stays Low because the channel is allocated for flow control.
- Make sure `do_cc` is not being driven High continuously. Whenever `do_cc` is High on a positive clock edge, the channel is used to send clock correction characters, so `s_axi_tx_tready` is deasserted.
- If the module includes USER K blocks but they are not being used, make sure the `s_axi_user_k_tx_tvalid` is not driven High. If it is High, `s_axi_tx_tready` stays Low as channel is allocated for USER K Blocks.
- If NFC is enabled, make sure the design on the other side of the channel did not send an NFC XOFF message. This cuts off the channel for normal data until the other side sends an NFC XON message to turn the flow on again.

Bytes and Words Are Being Lost As They Travel Through the Aurora Channel

- If using the AXI4-Stream interface, make sure to write data correctly. The most common mistake is to assume words are written without looking at `s_axi_tx_tready`. Also remember that the `s_axi_tx_tkeep` signal must be used to indicate which bytes are valid when `s_axi_tx_tlast` is asserted.
- Make sure to read correctly from the RX interface. Data and framing signals are only valid when `m_axi_rx_tvalid` is asserted.

Problems While Compiling the Design

Make sure to include all the files from the `src` directory when compiling. Check if simulator and libraries are set up properly. Check if Simulator language is set to **mixed**.

Next Step

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
- Results of the steps listed previously
- Attach a VCD or WLF dump of the observation

Hardware Debug

Most fields in the Vivado® Integrated Design Environment have tool tips serving as guidelines to properly configure and generate the core.

Observe and follow all RECOMMENDED and IMPORTANT notes in product guides.

As the transceiver is the critical building block in the Aurora core, debugging and ensuring proper operation of the transceiver is extremely important. Figure C-1 shows the steps involved for debugging transceiver-related issues.

IMPORTANT: *Ensure that the serial transceiver attributes are updated. See Appendix D, Generating a GT Wrapper File from the Transceiver Wizard for information regarding updating the serial transceiver attribute settings.*

This section provides a debug flow diagram for resolving some of the most common issues.

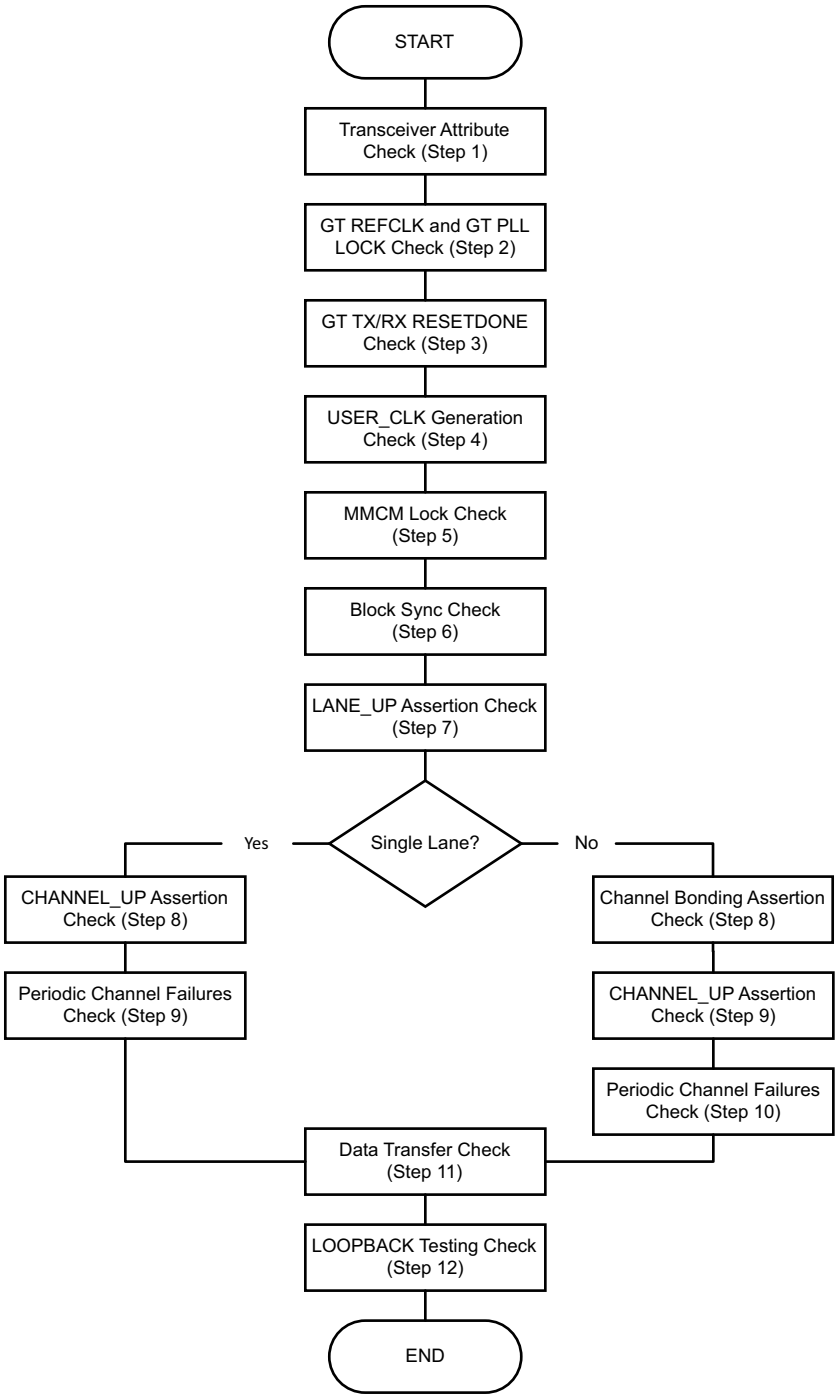


Figure C-1: **Transceiver Debug Flow Chart**

1. Transceiver Attribute Check

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

2. GT REFCLK and GT PLL LOCK Check

   A low-jitter differential clock must be provided as the transceiver reference clock. Ensure 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. Ensure that the transceiver locks into the incoming GT REFCLK and asserts the `gt_pll_lock` signal. This signal is available in the Aurora example design. If the `gt_pll_lock` signal is toggling periodically, check if `FSM_RESETDONE` is also toggling. Make sure that the GT PLL attributes are set correctly and that the transceiver generates `txoutclk` with expected frequency for the given line rate and datapath width options. Note that the Aurora core uses channel PLL/Quad PLL (CPLL/QPLL) in the generated core for GTX or GTH transceivers.

3. GT TX/RX RESETDONE Check

   The Aurora 64B/66B core uses the sequential reset mode; all of the transceiver components are reset sequentially, one after another. `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. Make sure the `gt_reset` signal pulse width duration matches with the respective transceiver guideline. Probe the signals and FSM states from the RX/TX STARTUP FSM module.

4. USER_CLK Generation Check

   The transceiver generates `txoutclk` based on the line rate parameter. The `user_clk` signal is generated from `txoutclk` and the Aurora 64B/66B core uses it as an FPGA logic clock. Check that `user_clk` is generated properly with the expected frequency from `txoutclk`. If the `user_clk` frequency is not in the expected range, check the frequency of the transceiver reference clock and PLL attributes.

5. MMCM Lock Check

   Aurora 64B/66B cores expect all clocks to be stable. If clocks are generated using an MMCM, ensure 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. This can be accomplished by using a BUFGCE with output clock where CE is driven by the MMCM lock output. If the `MMCM_LOCK` signal is toggling periodically, check if the TX_STARTUP_FSM is restarting and probe the signals and states of the FSM.

6. BLOCK SYNC Check

   See the block sync algorithm described in the *Aurora 64B/66B Protocol Specification* (SP011) [Ref 5] for block sync done.

7. LANE_UP Assertion Check

   Assertion of the `lane_up` signal indicates the communication between the transceiver and its channel partner is established and link training is successful. Enable loopback mode and check for `lane_up` assertions. Make sure the `do_cc` signal from the standard_CC block is connected properly. Bring the LANE_INIT_SM module FSM state signals to debug if `lane_up` is not asserted. See the Lane Initialization Procedure in the *Aurora 64B/66B Protocol Specification* (SP011) [Ref 5] for `lane_up` assertion details.

Single Lane:

8. CHANNEL_UP Assertion Check

   The criteria for channel_up signal assertion are the verification sequence (defined in the Aurora 64B/66B protocol) being transferred between channel partners, and the successful reception of four verification sequences. Enable loopback mode and check for `lane_up` assertions. Bring the CHANNEL_INIT_SM module FSM state signals to debug if `channel_up` is not asserted. For a simplex link, the simplex TX partner might have already achieved `channel_up` status. See the Channel Verification Procedure in the *Aurora 64B/66B Protocol Specification* (SP011) [Ref 5] for `channel_up` assertion details.

9. Periodic Channel Failures Check

   If the Aurora 64B/66B 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, the CLK_COR_MIN_LAT and CLK_COR_MAX_LAT transceiver values need to be adjusted.

Multiple Lane:

8. 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 transceiver CLK_COR_MIN_LAT and CLK_COR_MAX_LAT attributes are set as per the recommendation. See the channel bonding procedure in the *Aurora 64B/66B Protocol Specification* (SP011) [Ref 5] for channel_up assertion details.

9. CHANNEL_UP Assertion Check

   (See Single Lane, step 8)

10. Periodic Channel Failures Check

    (See Single Lane, step 9)

11. Data Transfer Check

    After `channel_up` is asserted, the Aurora 64B/66B core is ready to transfer data (wait for `tready` assertion). Data errors can be monitored with VIO. The `tx_d` and `rx_d` signals are connected to monitor the data transfer. Also, `soft_err` and `hard_err` are connected to VIO. If `data_err_count` is incrementing, perform a loopback test as described in step 12. If the loopback test passes, check the transmitted data and cable for channel integrity. Run IBERT to confirm link connectivity and SI on the channel. If IBERT runs are unsuccessful, monitor the power supplies and termination circuit, then run SI simulations on the transceiver.

12. LOOPBACK Testing Check

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

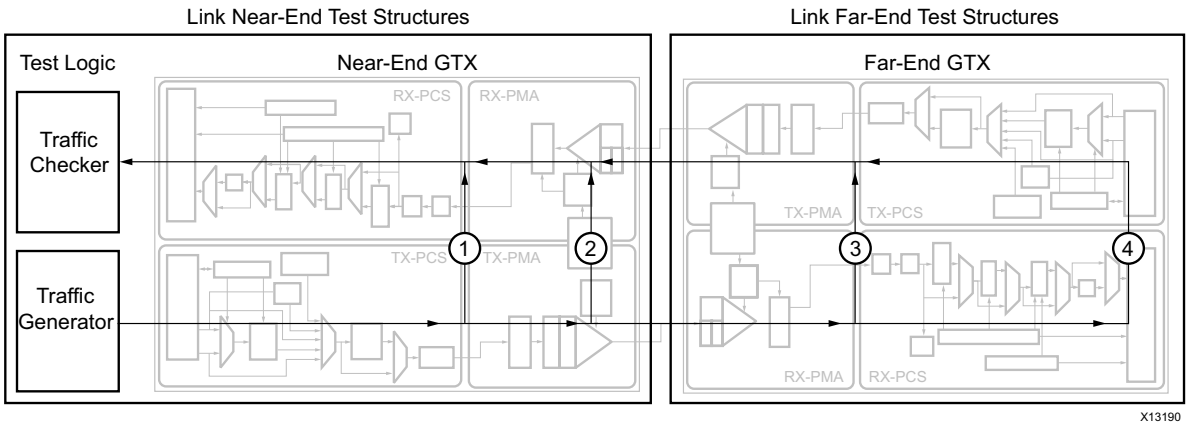


Figure C-2: **Loopback Testing Overview**

Design Bring-Up on Evaluation Board

Aurora Validation on the KC705 Board

For detailed procedures to set up and operate the Aurora 64B/66B core on the KC705 evaluation board, see *Designing a System Using the Aurora 64B66B Core (Duplex) on the KC705 Evaluation Kit Application Note* (XAPP1192) [Ref 24].

Interface Debug

If data is not being transmitted or received for the AXI4-Stream Interfaces, check the following conditions:

- If transmit `s_axi_tx_tready` is stuck Low following the `s_axi_tx_tvalid` input being asserted, the core cannot send data.
- If the receive `s_axi_tx_tvalid` is stuck Low, the core is not receiving data.
- Check that the `user_clk` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed. See Figure 2-8, page 18.
- Check core configuration.
- Add appropriate core specific checks.

Generating a GT Wrapper File from the Transceiver Wizard

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

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

This appendix provides instructions to generate the 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 64B/66B core.
2. Select the protocol template: Aurora 64B/66B.
3. Set the Line Rate for both the TX and RX transceivers based on the application requirement.
4. Select the Reference Clock from the drop-down menu for both the TX and RX transceivers based on the application requirement.
5. Select transceiver(s) and the clock source(s) based on the application requirement.
6. On Page 3, select External Data Width of the RX transceiver to be 32 Bits and Internal Data Width to be 32 bits. Ensure the TX transceiver is configured with 64-bit external data width and 32-bit internal data width.
7. Keep all other settings as default.
8. Generate the core.
9. Replace the `<user_component_name>_gtx.v` file in the `example_design/gt/` `directory` available in the Aurora 64B/66B core with the generated `<user_component_name>_gt.v` file generated from the 7 series FPGAs Transceivers Wizard.

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

Note: The UltraScale™ architecture Aurora 64B/66B 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 up to date. Manual editing of the UltraScale device transceiver files are not required in most cases.

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:

1. *7 Series FPGAs Overview* (DS180)
2. *UltraScale Architecture and Product Overview* (DS890)
3. *UltraScale Architecture GTH Transceivers User Guide* (UG576)
4. *7 Series FPGAs GTX/GTH Transceivers User Guide* (UG476)
5. *Aurora 64B/66B Protocol Specification* (SP011)
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. *7 Series GTZ Transceiver User Guide* (UG478)
11. *UltraScale FPGAs Transceivers Wizard Product Guide* (PG182)
12. *UltraScale Architecture Migration Methodology Guide* (UG1026)
13. *ISE to Vivado Design Suite Migration Guide* (UG911)
14. *LogiCORE IP Aurora 64B/66B v4.2 Data Sheet* (DS528)
15. *LogiCORE IP Aurora 64B/66B v4.1 Getting Started Guide* (UG238)
16. *LogiCORE IP Aurora 64B/66B v4.2 User Guide* (UG237)

17. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
18. *Vivado AXI Reference Guide* (UG1037)
19. *Virtex-7 FPGAs Data Sheet: DC and Switching Characteristics* (DS183)
20. *Kintex-7 FPGAs Data Sheet: DC and Switching Characteristics* (DS182)
21. *Synthesis and Simulation Design Guide* (UG626)
22. *ARM® AMBA® 4 AXI4-Stream Protocol v1.0 Specification* (ARM IHI 0051A)
23. *Packaging Custom AXI IP for Vivado IP Integrator Application Note* (XAPP1168)
24. *Designing a System Using the Aurora 64B66B Core (Duplex) on the KC705 Evaluation Kit Application Note* (XAPP1192)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/01/2014	9.3	• Added new v9.3 core features and attributes • Rearranged content to consolidate topics and better conform to template
06/04/2014	9.2	• Added User Parameter information.
04/02/2014	9.2	• Added C_EXAMPLE_SIMULATION parameter for post synthesis/implementation simulation speedup. • Added support for UltraScale™ devices. • Enhanced support for IP Integrator. • Added Little endian support for data and flow control interfaces as non-default Vivado® IDE selectable option. • Provided interoperability guidance. • Resolved functional issue seen with specific frame lengths in certain scenarios.
12/18/2013	9.1	• Added default information to `init_clk_p`, `initclk_n`, and INIT_CLK description. • Updated reset sequencing steps and waveform. • Added information about `pma_init` staging. • Updated screen captures. • Added sequence of steps describing hardware FSM reset

Date	Version	Revision
10/02/2013	9.0	• Added new chapters: Simulation, Test Bench and Synthesis and Implementation. • Added shared logic and transceiver debug features. • Updated directory and file structure. • Changed signal and port names to lowercase. • Added Zynq®-7000 device support. • Updated RX datapath architecture. • Updated Aurora Simplex Operation description. • Updated Figure 3-2 and screen captures in Chapter 4. • Updated Hot-Plug Logic description. • Added IP Integrator support. • Updated XDC file for the example design. • Added design bring-up on evaluation board information.
06/19/2013	8.1	• Revision number advanced to 8.1 to align with core version number. • Updated for 2013.2 release and core version 8.1. • Fixed a NFC transmit failure scenario when Clock Correction is transmitted in conjunction with the second NFC request. NFC state machine is updated to handle such scenarios.
03/20/2013	2.0	• Updated for 2013.1 release and core version 8.0. • Removed all ISE® design tools and Virtex®-6 related device information. • Added Reset waveforms • Updated debug guide with core and transceiver debug details • Created lowercase ports for Verilog • Added Simplex TX/RX support • Enhanced protocol to increase Channel Init time • Included TXSTARTUPFSM and RXSTARTUPFSM modules to control GT reset sequence
12/18/2012	1.0.1	• Updated for 14.4 and 2012.4 release. • Added TKEEP description • Updated Debugging appendix.
10/16/2012	1.0	Initial Xilinx release as a product guide. This document replaces UG775, *LogiCORE IP Aurora 64B/66B User Guide* and DS815, *LogiCORE IP Aurora 64B/66B Data Sheet*. • Added section explaining constraining of the core. • Added section explaining core debugging.

Please Read: Important Legal Notices