LogiCORE™ IP Aurora 64B/66B v5.1

User Guide

UG775 (v1.0) December 14, 2010

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/14/10	1.0	First release of the core with AXI interface support. The previous release of this document was the *LogiCORE IP Aurora 64B/66B v4.1 Getting Started Guide* (UG238) and *User Guide* (UG237). The two documents have been integrated to be this new Aurora 64B/66B user guide that supports the AXI interface.

Table of Contents

Chapter 3: User Interface

Chapter 4: Flow Control

Chapter 5: User K-Block Interface

Chapter 6: Status, Control, and the GT Interface

Chapter 7: Clock Interface and Clocking

Chapter 8: Clock Compensation Interface

Chapter 9: Quick Start Example Design

Chapter 10: Example Design Overview

Chapter 11: Project Directory Structure

Appendix A: Two Aurora 64B/66B Cores in Virtex-5 Family Sharing a RocketIO Transceiver Tile

Appendix B: Generating a GTX Wrapper File from the GTX Transceiver Wizard

Appendix C: Aurora AXI4-Stream Migration Guide

Schedule of Figures

Chapter 1: Introduction

Chapter 2: Customizing the Aurora 64B/66B Core

Chapter 3: User Interface

Chapter 4: Flow Control

Chapter 5: User K-Block Interface

Chapter 6: Status, Control, and the GT Interface

Chapter 7: Clock Interface and Clocking

Chapter 8: Clock Compensation Interface

Chapter 9: Quick Start Example Design

Chapter 10: Example Design Overview

Chapter 11: Project Directory Structure

Appendix A: Two Aurora 64B/66B Cores in Virtex-5 Family Sharing a RocketIO Transceiver Tile

Appendix B: Generating a GTX Wrapper File from the GTX Transceiver Wizard

Appendix C: Aurora AXI4-Stream Migration Guide

Schedule of Tables

Chapter 1: Introduction

Chapter 2: Customizing the Aurora 64B/66B Core

Chapter 3: User Interface

Chapter 4: Flow Control

Chapter 5: User K-Block Interface

Chapter 6: Status, Control, and the GT Interface

Chapter 7: Clock Interface and Clocking

Chapter 8: Clock Compensation Interface

Chapter 9: Quick Start Example Design

Chapter 10: Example Design Overview

Chapter 11: Project Directory Structure

Appendix A: Two Aurora 64B/66B Cores in Virtex-5 Family Sharing a RocketIO Transceiver Tile

Appendix B: Generating a GTX Wrapper File from the GTX Transceiver Wizard

Appendix C: Aurora AXI4-Stream Migration Guide

About This Guide

The LogiCORE™ IP Aurora 64B/66B core supports the AMBA® protocol AXI4-Stream user interface. The *LogiCORE IP Aurora 64B/66B v5.1 User Guide* provides information for generating a LogiCORE™ IP Aurora 64B/66B core using Virtex®-5 FPGA GTX transceivers and Virtex-6 FPGA GTX/GTH transceivers.

The core implements the Aurora 64B/66B protocol using the high-speed serial transceivers on the Virtex-5 FXT, and TXT family, the Virtex-6 LXT, SXT, HXT, and lower-power family.

This user guide describes the function and operation of the LogiCORE™ IP Aurora 64B/66B v5.1 core and provides information about designing, customizing, and implementing the core.

Contents

This guide contains the following chapters:

- Preface, "About this Guide" introduces the organization and purpose of the design guide, a list of additional resources, and the conventions used in this document.
- Chapter 1, Introduction describes the core and related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.
- Chapter 2, Customizing the Aurora 64B/66B Core describes how to customize an Aurora 64B/66B core with the available parameters.
- Chapter 3, User Interface provides port descriptions for the user interface.
- Chapter 4, Flow Control describes the user flow control and native flow control options for sending and receiving data.
- Chapter 5, User K-Block Interface describes short single block data transmission and reception.
- Chapter 6, Status, Control, and the GT Interface provides diagrams and port descriptions for the Aurora 64B/66B core's status and control interface, along with the GTX serial I/O interface.
- Chapter 7, Clock Interface and Clocking describes how to connect FPGA clocking resources.
- Chapter 8, Clock Compensation Interface covers Aurora 64B/66B clock compensation, and explains how to customize it for a given system.
- Chapter 9, Quick Start Example Design provides an overview of the Aurora 64B/66B protocol and core, and gives a step-by-step tutorial on how to generate Aurora 64B/66B designs with the CORE Generator™ software.

- Chapter 10, Example Design Overview defines the main components of the example design.
- Chapter 11, Project Directory Structure provides detailed information about the example design, including a description of files and the directory structure generated by the Xilinx CORE Generator tool, the purpose and contents of the provided scripts, the contents of the example HDL wrappers, and the operation of the demonstration test bench.
- Appendix A, Two Aurora 64B/66B Cores in Virtex-5 Family Sharing a RocketIO Transceiver Tile
- Appendix B, Generating a GTX Wrapper File from the GTX Transceiver Wizard
- Appendix C, Aurora AXI4-Stream Migration Guide explains about migration of legacy (LocalLink based) Aurora cores to the AXI4-Stream based Aurora core.

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/support/documentation/index.htm.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support/mysupport.htm.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands you enter in a syntactical statement	**`ngdbuild`** *design_name*
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *User Guide* for details.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Dark Shading	Items that are not supported or reserved	This feature is not supported

Convention	Meaning or Use	Example
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required.	**ngdbuild** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**lowpwr ={on|off}**
Vertical bar |	Separates items in a list of choices	**lowpwr ={on|off}**
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Omitted repetitive material	**allow block** *block_name loc1 loc2 ... locn;*
Notations	The prefix '0x' or the suffix 'h' indicate hexadecimal notation	A read of address 0x00112975 returned 45524943h.
	An '_n' means the signal is active-Low	usr_teof_n is active-Low.

Online Document

The following linking conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-5 FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Introduction

This chapter introduces the LogiCORE™ IP Aurora 64B/66B core and provides related information, including recommended design experience, additional resources, technical support, and how to submit feedback to Xilinx. The Aurora 64B/66B core is based on the *Aurora 64B/66B Protocol Specification* and uses the high-speed serial GTX or GTH transceivers in applicable Virtex®-5 and Virtex-6 FPGAs. The core is delivered as open-source code and supports Verilog and VHDL design environments. Each core comes with an example design and supporting modules.

About the Core

The Aurora 64B/66B core is a Xilinx® CORE Generator™ IP core, included in the latest IP Update on the Xilinx IP Center. For detailed information about the core, see www.xilinx.com/aurora.

System Requirements

Windows

- Windows XP Professional 32-bit/64-bit
- Windows Vista Business 32-bit/64-bit

Linux

- Red Hat Enterprise Linux WS v4.0 32-bit/64-bit
- Red Hat Enterprise Desktop v5.0 32-bit/64-bit (with Workstation Option)
- SUSE Linux Enterprise (SLE) desktop and server v10.1 32-bit/64-bit

Software

- ISE® v12.4 software
- Mentor Graphics ModelSim v6.5c
- Cadence Incisive Enterprise Simulator (IES) v9.2
- Synopsys Synplify Pro D-2009.12
- Synopsys VCS and VCS MX 2009.12

Before You Begin

Before installing the core, you must have a MySupport account and the v12.4 software installed on your system. If you already have an account and have the software installed, go to Installing the Core, otherwise do the following:

1. Click Login at the top of the Xilinx home page; then follow the onscreen instructions to create a MySupport account.
2. Install the v12.4 software. For the software installation instructions, see the ISE Design Suite Release Notes and Installation Guide available in the ISE software documentation located on the ISE Design Suite product page.

Installing the Core

The Aurora 64B/66B core is included with the v12.4 software.

See *ISE CORE Generator IP Updates - Installation Instructions* for details about installing ISE 12.4.

Recommended Design Experience

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

See Chapter 6, Status, Control, and the GT Interface carefully, and consult the PCB design requirements information in the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, *Virtex-6 FPGA GTX Transceivers User Guide*, and *Virtex-6 FPGA GTH Transceivers User Guide*. Contact your local Xilinx representative for a closer review and estimation for your specific requirements.

Related Documents

Before generating an Aurora 64B/66B core, users should be familiar with the following:

- SP011 *Aurora 64B/66B Protocol Specification* is located on the Aurora product page
- *AMBA AXI4-Stream Protocol Specification*
- UG761, Xilinx *AXI Reference Guide*
- UG198, *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*
- UG366, *Virtex-6 FPGA GTX Transceivers User Guide*
- UG371, *Virtex-6 FPGA GTH Transceivers User Guide*
- ISE software documentation on the ISE Design Suite product page

Additional Core Resources

For detailed information and updates about the Aurora 64B/66B core, see the following documents, located on the Aurora product page.

- DS815, *Aurora 64B/66B v5.1 Data Sheet*
- Virtex-5/Virtex-6 FPGA Aurora 64B/66B Release Notes

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team of engineers with expertise using the Aurora 64B/66B core.

Xilinx will provide technical support for use of this product as described in this guide. Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines, or for modifications to the source code.

Feedback

Xilinx welcomes comments and suggestions about the Aurora 64B/66B core and the accompanying documentation.

Aurora 64B/66B Core

For comments or suggestions about the Aurora 64B/66B core, please submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

- Product name
- Core version number
- List of parameter settings
- Explanation of your comments

Document

For comments or suggestions about this document, please submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

XILINX®

Chapter 2

Customizing the Aurora 64B/66B Core

Introduction

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

Using the IP Customizer

The Aurora 64B/66B IP customizer is presented when the user selects the Aurora 64B/66B core in the CORE Generator software. For help starting and using the CORE Generator software, see the *CORE Generator Guide* in the ISE™ software documentation. Figure 2-1, page 22, Figure 2-2, page 22, Figure 2-3, page 23, and Figure 2-4, page 23 show features that are described in corresponding sections.

Note: The options shown in Figure 2-3 and Figure 2-4, page 23 are only available for the Virtex®-6 HXT devices.

IP Customizer

Figure 2-1, page 22 shows the customizer. The left side displays a representative block diagram of the Aurora 64B/66B core as currently configured. The right side consists of user-configurable parameters. Details on the customizing options are provided below, starting with Component Name, page 24.

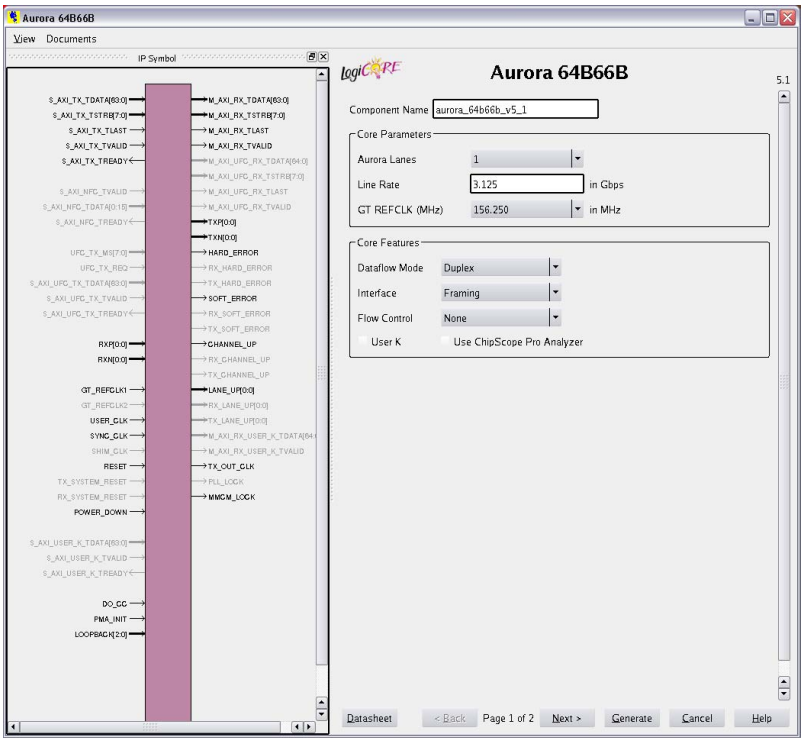


Figure 2-1: **Aurora 64B/66B IP Customizer Page 1**

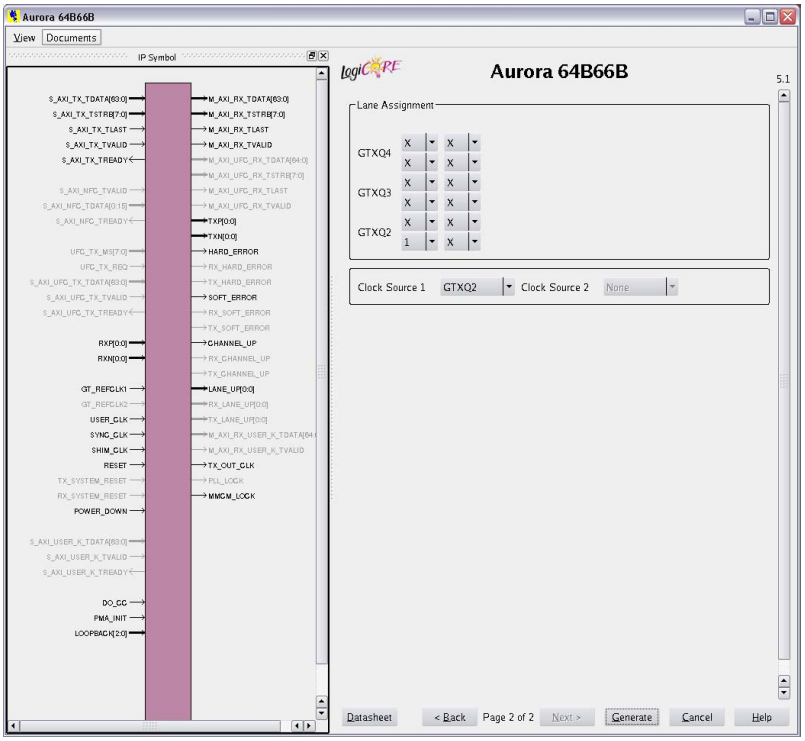


Figure 2-2: **Aurora 64B/66B IP Customizer Page 2**

The options shown in Figure 2-3 and Figure 2-4 are available only for the Virtex-6 HXT devices.

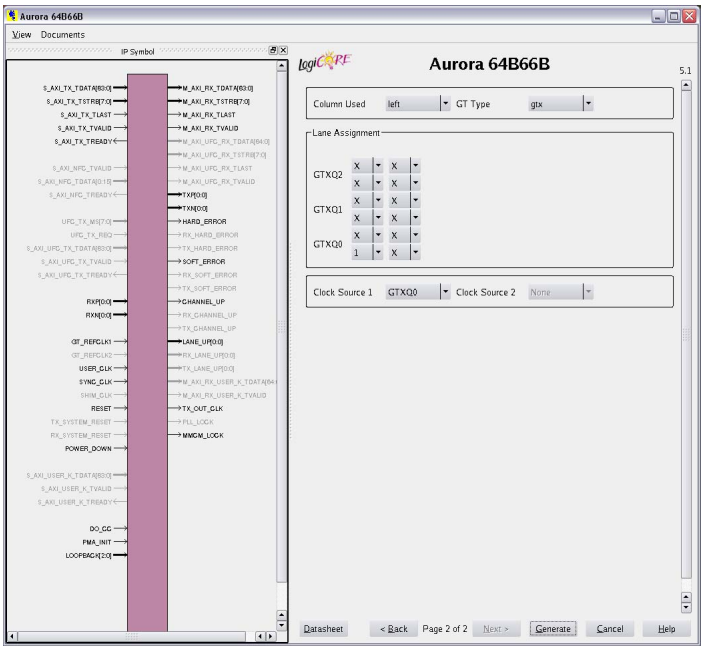


Figure 2-3: **Aurora 64B/66B IP Customizer Page 2 (Virtex-6 HXT Devices for GTX Transceivers Only)**

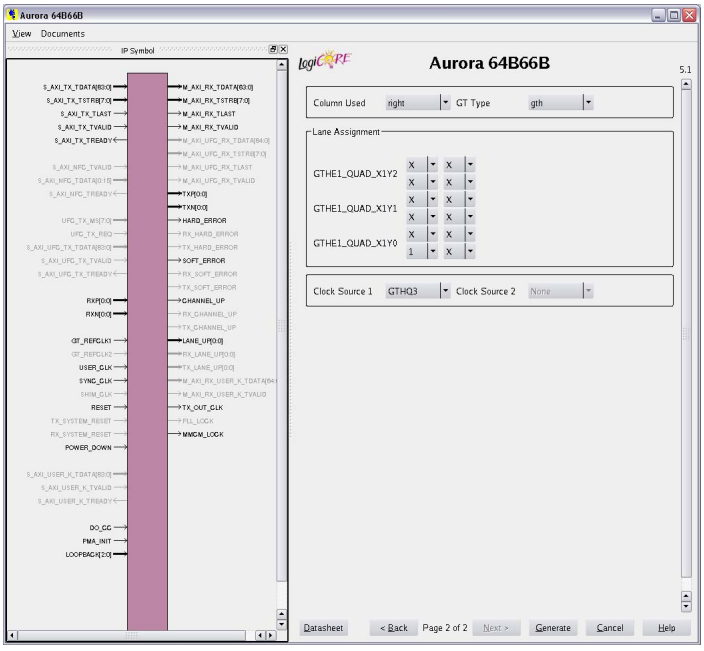


Figure 2-4: **Aurora 64B/66B IP Customizer Page 2 (Virtex-6 HXT Devices for GTH Transceivers Only)**

Component Name

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

Default: aurora_64b66b_v5_1

Lane Assignment

Refer to the diagram in the information area in Figure 2-1, page 22. Each numbered row represents a GT tile and each active box represents an available GTX/GTH transceiver. For each Aurora lane in the core, starting with Lane 1, select a GTX/GTH transceiver and place the lane by selecting its number in the GTX/GTH placement box.

Aurora Lanes

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

Default: 1

Interface

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

Default: Framing

Data Flow Mode

Select the options for direction of the channel the Aurora 64B/66B core will support. Simplex Aurora 64B/66B cores have a single, unidirectional serial port that connects to a complementary simplex Aurora 64B/66B core. Three options are provided as RX-only simplex, TX-only simplex, or RX/TX simplex. These options select the direction of the channel the Aurora 64B/66B core will support. RX/TX simplex creates two cores, one RX and one TX, that share GTX/GTH transceivers.

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

Default: Duplex

Flow Control

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

Available options are:

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

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

Default: None

Line Rate

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

Default: 3.125 Gbps for GTX transceivers and 10.3125 Gbps for GTH transceivers

GT Reference Clock Frequency

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

Default: 156.25 MHz

Clock Source 1 and Clock Source 2

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

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

Note: n depends on the GT position.

Column Used

Select appropriate column from the drop down list. This option is applicable only for Virtex-5 TXT and Virtex-6 HXT devices. For other devices, the drop down box is not visible.

Default: left

GT_TYPE

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

Default: gtx

Use ChipScope Pro Analyzer

Select to add ChipScope™ Pro cores to the Aurora 64B/66B core. (See Using ChipScope Pro Cores with the Aurora 64B/66B Core, page 73.) This option provides users a debugging interface that shows the core status signals in the ChipScope Pro analyzer tool.

Default: Unchecked

User K

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

Default: Unchecked

Generate

Click Generate to generate the core. (See Generating the Core, page 70.) The modules for the Aurora 64B/66B core are written to the CORE Generator software project directory using the same name as the top level of the core

Using the Build Script

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

Designing with the Core

This section provides a general description of how to use the Aurora 64B/66B core in your designs, and should be used in conjunction with Chapter 3, User Interface that describes core specific interfaces.

General Design Guidelines

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

Keep It Registered

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

Recognize Timing Critical Signals

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

Use Supported Design Flows

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

Make Only Allowed Modifications

The Aurora 64B/66B core is not user modifiable. Any modifications may 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 CORE Generator tool.

XILINX®

Chapter 3

User Interface

Introduction

An Aurora 64B/66B core can be generated with either a *framing* or *streaming* user data interface. In addition, flow control options are available for designs with framing or streaming interfaces. See Chapter 4, Flow Control.

The framing user interface complies with the *AXI4-Stream Protocol Specification* (see Related Documents, page 18). It comprises the signals necessary for transmitting and receiving framed user data. The streaming interface allows users to send data without special frame delimiters. It is simple to operate and uses fewer resources than framing.

Top Level Architecture

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

Figure 3-1, page 30 shows Aurora 64B/66B top level for a duplex configuration. The top level file is the starting point for a user design.

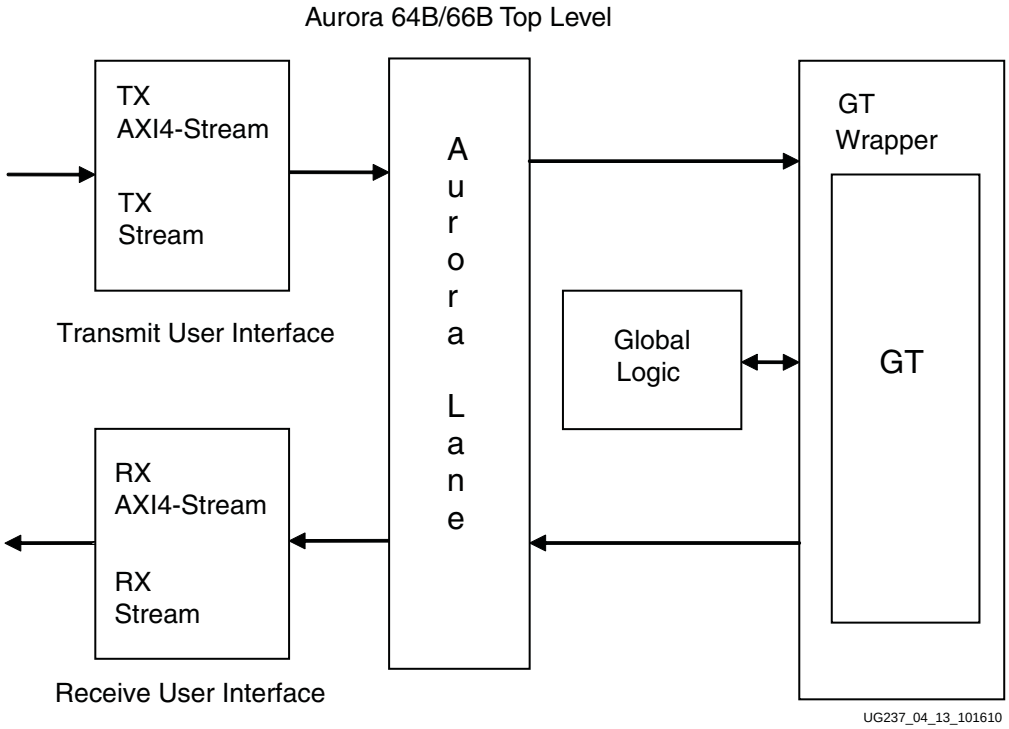


Figure 3-1: **Top-Level Architecture**

Following sections describe streaming and framing interface in details. User interface logic should be designed such that it complies with timing requirement of the respective interface as explained in the subsequent sections.

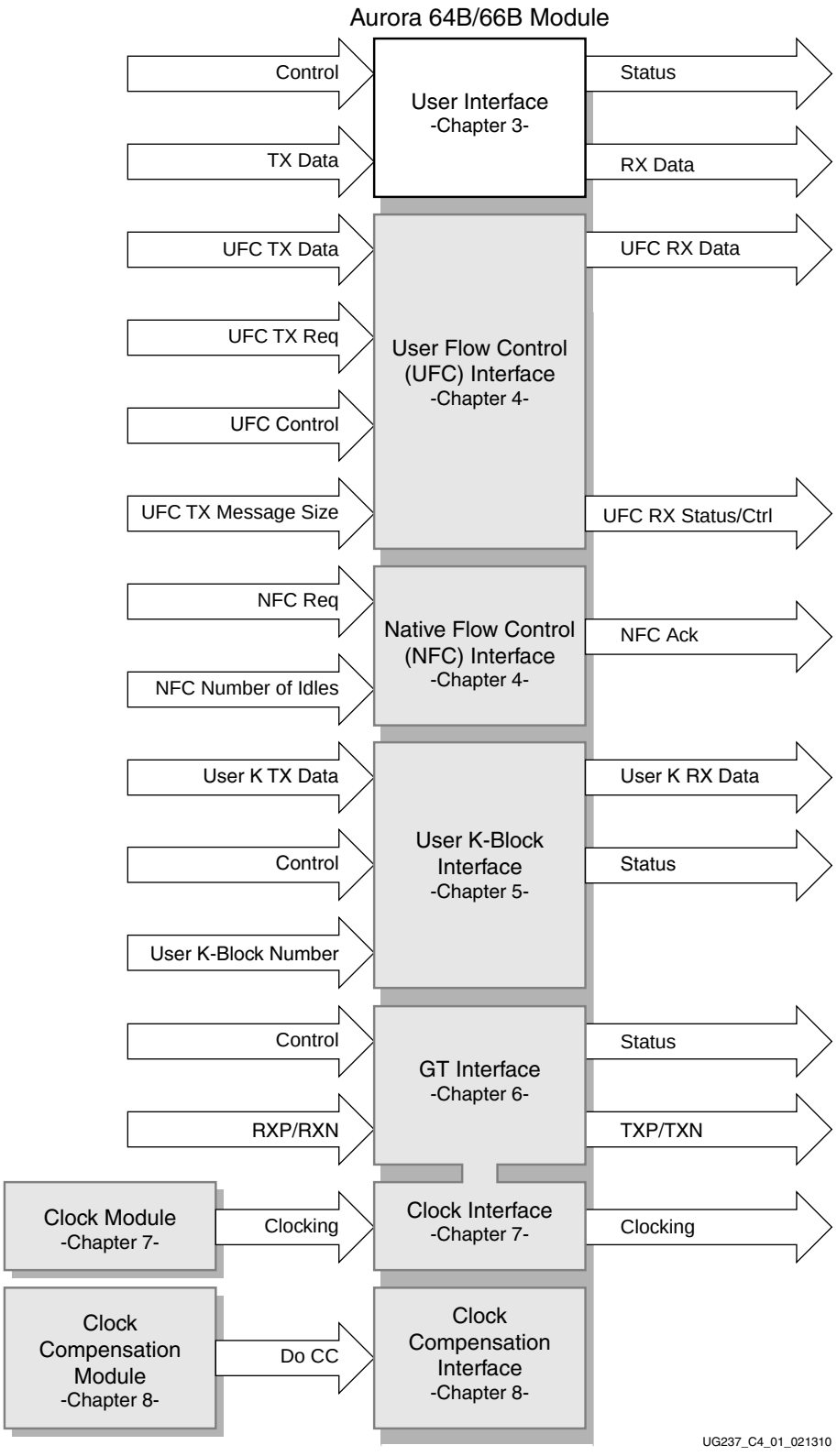


Figure 3-2: **Top-Level User Interface**

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

Framing Interface

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

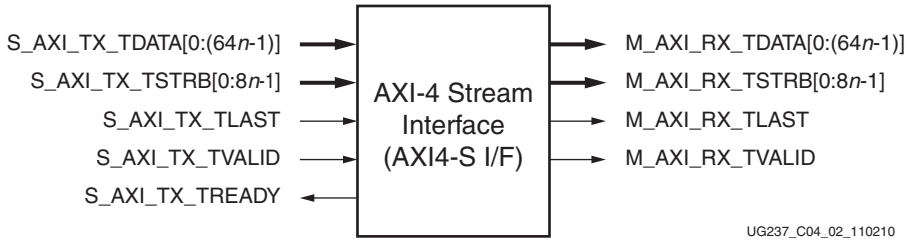


Figure 3-3: **Aurora 64B/66B Core Framing Interface (AXI4-Stream)**

AXI4-Stream TX Ports

Table 3-1 lists the AXI4-Stream TX data ports and their descriptions.

Table 3-1: **AXI4-Stream User I/O Ports (TX)**

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

AXI4-Stream RX Ports

Table 3-2 lists the AXI4-Stream RX data ports and their descriptions.

Table 3-2: **AXI4-Stream User I/O Ports (RX)**

Name	Direction	Description
M_AXI_RX_TDATA[0:(64*n*-1)]	Output	Incoming data from channel partner (Ascending bit order).
M_AXI_RX_TLAST	Output	Signals the end of the incoming frame (active-High, asserted for a single user clock cycle).
M_AXI_RX_TSTRB[0:8*n*-1]	Output	Specifies the number of valid bytes in the last data beat; valid only when M_AXI_RX_TLAST is asserted.
M_AXI_RX_TVALID	Output	Asserted (active-High) when data and control signals from an Aurora 64B/66B core are valid. Deasserted (active-Low) when data and/or control signals from an Aurora 64B/66B core should be ignored.

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

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

When the core receives data, it does the following:

- Detects and discards control bytes (idles, clock compensation)
- Asserts framing signals (M_AXI_RX_TLAST)
- Recovers data from the lanes
- Assembles data for presentation to the user on the M_AXI_RX_TDATA bus along with valid no of bytes (M_AXI_RX_TSTRB) during the M_AXI_RX_TLAST cycle

AXI4-Stream Bit Ordering

The Aurora 64B/66B cores AXI4 Stream User Interface use ascending ordering. They transmit and receive the most significant bit of the least significant byte first. Figure 3-4 shows the organization of an *n*-byte example of the AXI4-Stream data interfaces of an Aurora 64B/66B core.

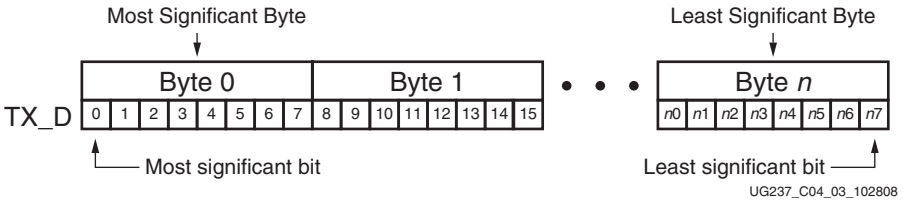


Figure 3-4: **AXI4-Stream Interface Bit Ordering**

Transmitting Data

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

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

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

Data Strobe

AXI4-Stream allows the last word of a frame to be a partial word. This lets a frame contain any number of bytes, regardless of the word size. The S_AXI_TX_TSTRB bus is used to indicate the number of valid bytes in the final word of the frame. The bus is only used when S_AXI_TX_TLAST is asserted. STRB is the number of valid bytes in the S_AXI_TX_TDATA bus. An 'n' STRB value indicates all bytes in the S_AXI_TX_TDATA port are valid.

Aurora 64B/66B Frames

The TX submodule translates each user frame that it receives through the TX interface to an Aurora 64B/66B frame. The core starts an Aurora 64B/66B frame by sending a data block with the first word of data, and ends the frame by sending a separator block containing the last bytes of the frame. Idle blocks are inserted whenever data is not available. Blocks are eight bytes of scrambled data or control information with a 2-bit control header (a total of 66 bits). All data in Aurora 64B/66B is sent as part of a data block or a separator block (a separator block consists of a count field, indicating how many bytes are valid in that particular block). Table 3-3 shows a typical Aurora 64B/66B frame with an even number of data bytes.

Length

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

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

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

Example A: Simple Data Transfer

Figure 3-5 shows an example of a simple data transfer on a AXI4-Stream interface that is *n* bytes wide. In this case, the amount of data being sent is 3*n* bytes and so requires three data

beats. S_AXI_TX_TREADY is asserted, indicating that the AXI4-Stream interface is already ready to transmit data. When the Aurora 64B/66B is not sending data, it sends idle blocks.

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

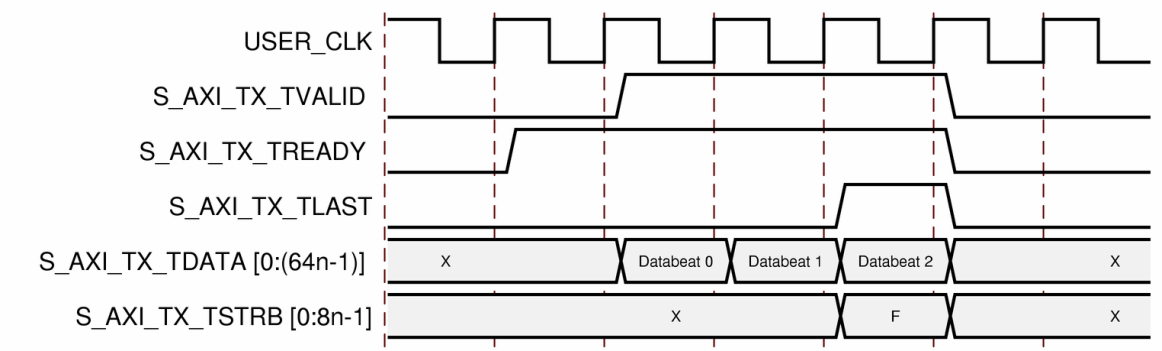


Figure 3-5: **Simple Data Transfer**

Example B: Data Transfer with Pause

Figure 3-6 shows how a user can pause data transmission during a frame transfer. In this example, the user is sending 3*n* bytes of data, and pauses the data flow after the first *n* bytes. After the first data word, the user deasserts S_AXI_TX_TVALID, causing the TX Aurora 64B/66B core to ignore all data on the bus and transmit idle blocks instead. The pause continues until S_AXI_TX_TVALID is deasserted.

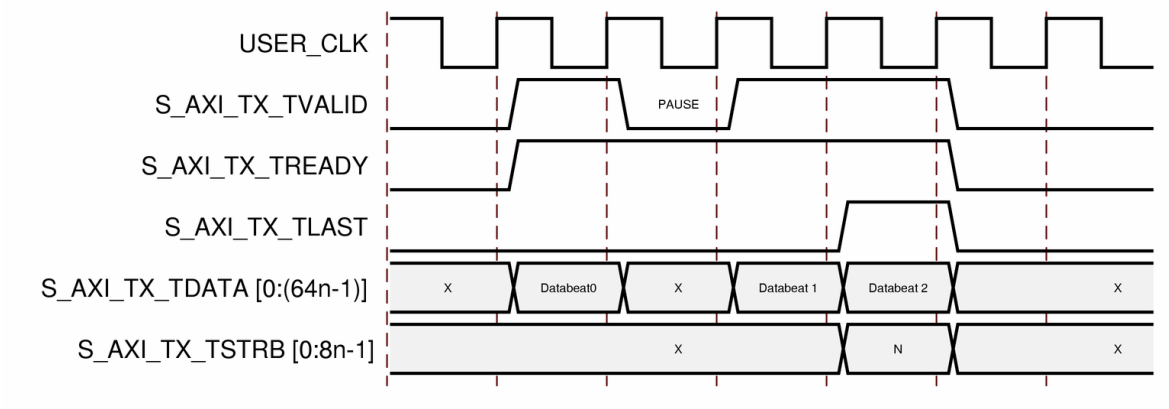
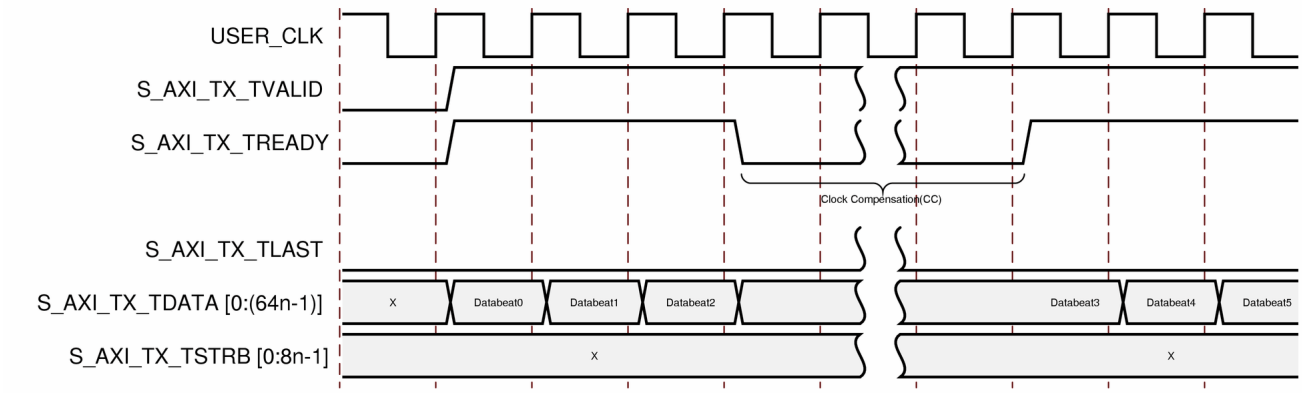


Figure 3-6: **Data Transfer with Pause**

Example C: Data Transfer with Clock Compensation

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

Figure 3-7 shows how the Aurora 64B/66B core pauses data transmission during the clock compensation sequence.



Notes:

1. When clock compensation is used, uninterrupted data transmission is not possible. See Chapter 8, Clock Compensation Interface for more information about when clock compensation is required.

Figure 3-7: **Data Transfer Paused by Clock Compensation**

TX Interface Example

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

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

Figure 3-8 is a diagram of a typical connection between an Aurora 64B/66B core and the data source (in this example, a FIFO), including the simple logic needed to generate, S_AXI_TX_TVALID and S_AXI_TX_TLAST from typical FIFO buffer status signals. While RESET is false, the example application waits for a FIFO to fill, then generates the S_AXI_TX_TVALID signal. These signals cause the Aurora 64B/66B core to start reading the FIFO by asserting the S_AXI_TX_TREADY signal.

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

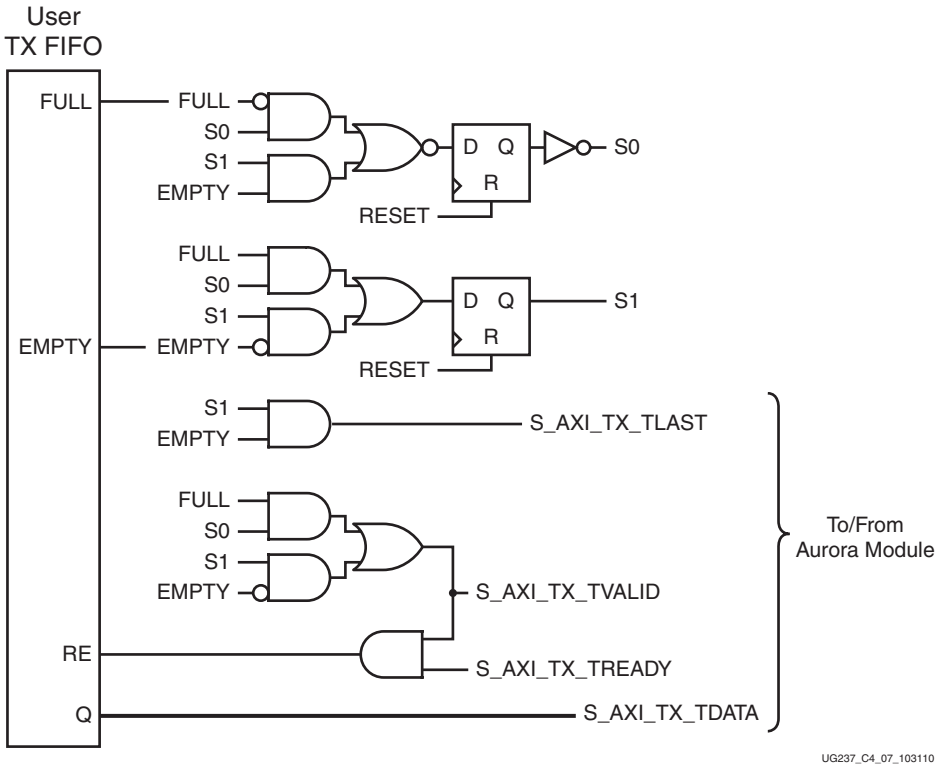


Figure 3-8: **Transmitting Data**

Receiving Data

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

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

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

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

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

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

Example A: Data Reception with Pause

Figure 3-9 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. On the clock cycle following the first data beat, the core deasserts M_AXI_RX_TVALID, indicating to the user that there is a pause in the data flow.

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

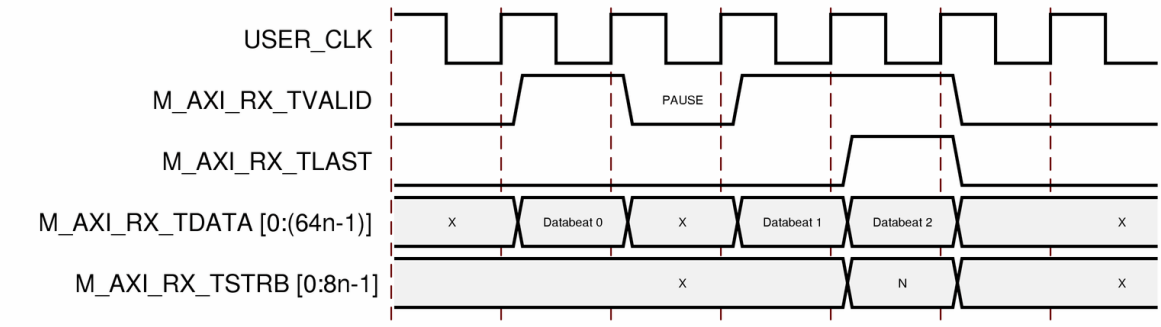


Figure 3-9: **Data Reception with Pause**

RX Interface Example

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

Framing Efficiency

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

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

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

The gear box in GTX/GTH transceivers requires periodic pause to account for the clock divider ratio and 64B/66B encoding. This appears as a back pressure in the AXI4-Stream interface and user data needs to be stopped for 2 cycles after every 32 cycles (Figure 3-10). The User Interface will have the S_AXI_TX_TREADY signal from the Aurora core being deasserted (active-Low) for 2 cycles once every 32 cycles. The 2-cycle pause is used to compensate the Gearbox for the 64B/66B encoding and to provide timing margin for the core.

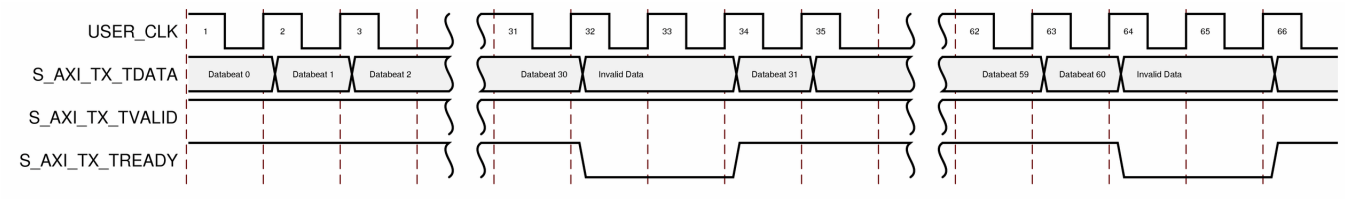


Figure 3-10: **Framing Efficiency**

For more information on gear box pause in GTX/GTH, see the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, the *Virtex-6 FPGA GTX Transceivers User Guide*, and the *Virtex-6 FPGA GTH Transceivers User Guide*.

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

Example

Table 3-4 is an example calculated after including overhead for clock compensation. It shows the efficiency for a single-lane channel and illustrates that the efficiency increases as frame length increases.

Table 3-4: **Efficiency Example**

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

Table 3-5 shows the overhead in single-lane channel when transmitting 256 bytes of frame data. The resulting data unit is 264 bytes long due to the SEP block used to end the frame. This results in 3.03% overhead in the transmitter. In addition, clock compensation blocks must be transmitted for three cycles every 10,000 cycles, resulting in an additional 0.03% overhead in the transmitter.

Table 3-5: **Typical Overhead for Transmitting 256 Data Bytes**

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

Streaming Interface

Figure 3-11 shows an example of an Aurora 64B/66B core configured with a streaming user interface.

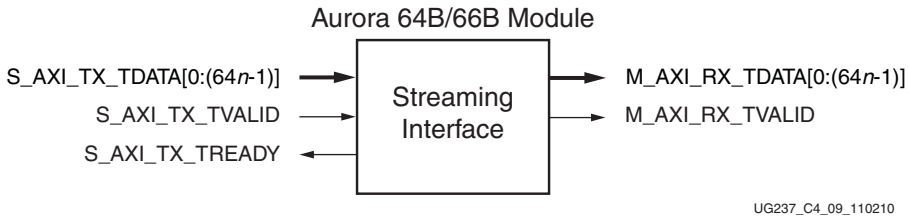


Figure 3-11: **Aurora 64B/66B Core Streaming User Interface**

Streaming TX Ports

Table 3-6 lists the streaming TX data ports.

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

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

Streaming RX Ports

Table 3-7 lists the streaming RX data ports. These ports are included on full-duplex, simplex RX, and RX/TX simplex framing cores.

Table 3-7: **Streaming User I/O Ports (RX)**

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

Transmitting and Receiving Data

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

When S_AXI_TX_TVALID is deasserted, gaps are created between words. These gaps are preserved, except when clock compensation sequences are being transmitted. Clock compensation sequences are replicated or deleted by the CC logic to make up for frequency differences between the two sides of the Aurora channel. As a result, gaps created when DO_CC is asserted can shrink and grow. For details on the DO_CC signal, see Chapter 8, Clock Compensation Interface.

When data arrives at the RX side of the Aurora channel it is presented on the M_AXI_RX_TDATA bus and M_AXI_RX_TVALID is asserted. The data must be read

immediately or it will be lost. If this is unacceptable, a buffer must be connected to the RX interface to hold the data until it can be used.

Figure 3-12, page 42 shows a typical example of streaming data transfer. The example begins with neither of the ready signals asserted, indicating that both the user logic and the Aurora 64B/66B core are not ready to transfer data. During the next clock cycle, the Aurora 64B/66B core indicates that it is ready to transfer data by asserting S_AXI_TX_TREADY. One cycle later, the user logic indicates that it is ready to transfer data by asserting the S_AXI_TX_TVALID signal and placing data on the S_AXI_TX_TDATA bus. Because both ready signals are now asserted, data D0 is transferred from the user logic to the Aurora 64B/66B core. Data D1 is transferred on the following clock cycle. In this example, the Aurora 64B/66B core deasserts its ready signal, S_AXI_TX_TREADY, and no data is transferred until the next clock cycle when, once again, the S_AXI_TX_TREADY signal is asserted. Then the user deasserts S_AXI_TX_TVALID on the next clock cycle, and no data is transferred until both ready signals are asserted.

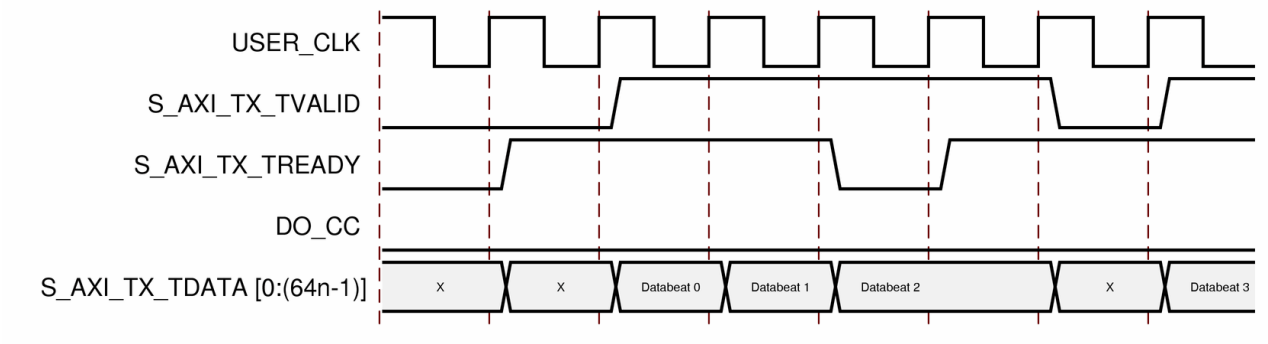


Figure 3-12: **Typical Streaming Data Transfer**

Figure 3-13, page 42 shows a typical example of streaming data reception.

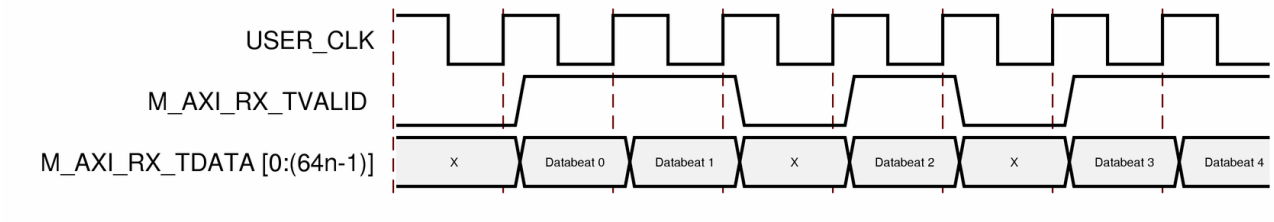


Figure 3-13: **Typical Streaming Data Reception**

Chapter 4

Flow Control

Introduction

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

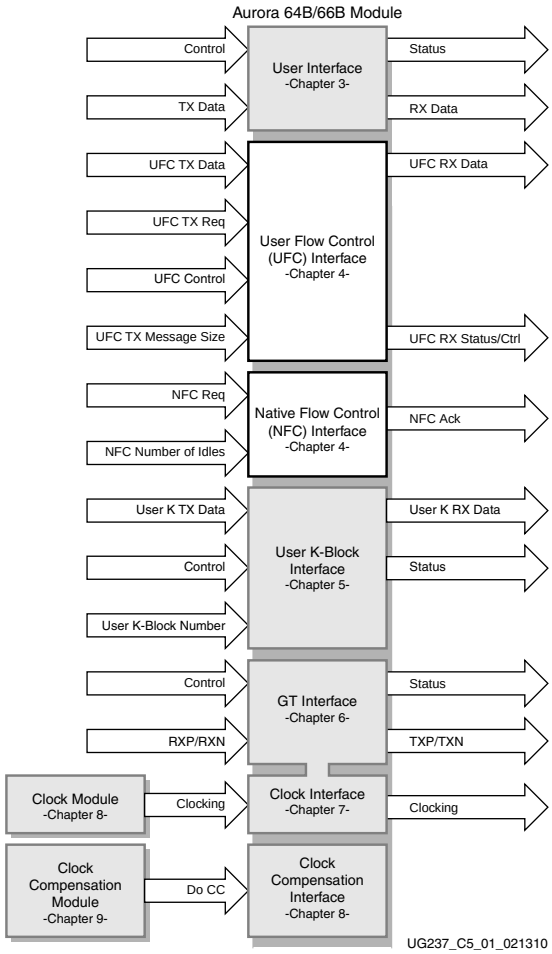


Figure 4-1: **Top-Level Flow Control**

Native Flow Control

Table 4-1 lists the ports for the native flow control (NFC) interface.

Table 4-1: **NFC I/O Ports**

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

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

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

Example A: Transmitting an NFC Message

Figure 4-2 shows an example of the transmit timing when the user sends an NFC message to a channel partner using a AXI4-Stream interface.

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

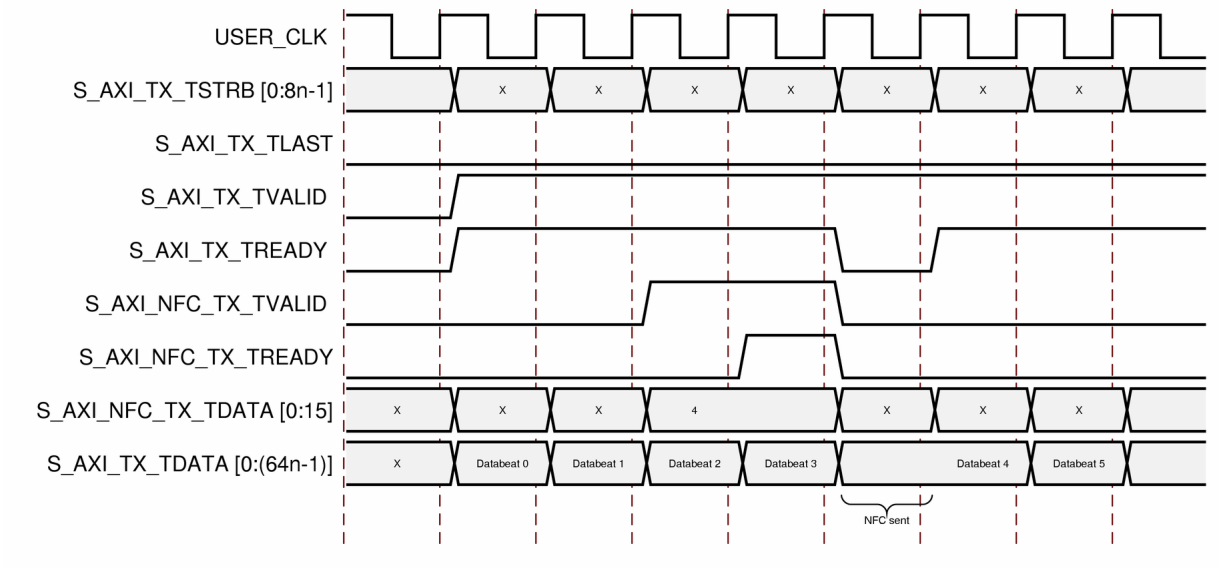


Figure 4-2: **Transmitting an NFC Message**

Example B: Receiving a Message with NFC Idles Inserted

Figure 4-3 shows an example of the signals on the TX user interface when an NFC message is received. In this case, the NFC message sends the number 8'b01, requesting two cycles without data transmission. The core deasserts S_AXI_TX_TREADY on the user interface to prevent data transmission for two cycles. In this example, the core is operating in Immediate NFC mode. Aurora 64B/66B cores can also operate in completion mode, where NFC Idles are only inserted before the first data bytes of a new frame. If a completion mode core receives an NFC message while it is transmitting a frame, it finishes transmitting the frame before deasserting S_AXI_TX_TREADY to insert idles.

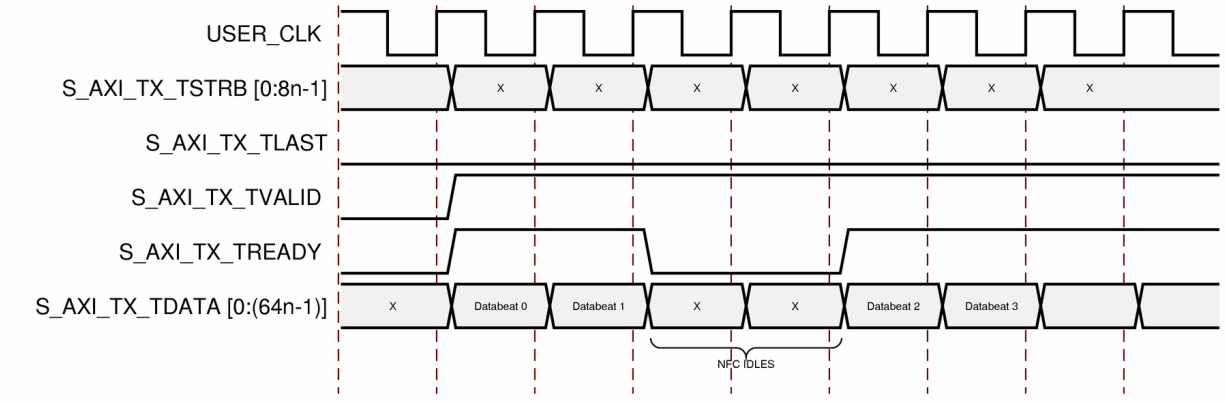


Figure 4-3: **Transmitting a Message with NFC Idles Inserted**

User Flow Control

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

Table 4-2 describes the ports for the UFC interface.

Table 4-2: **UFC I/O Ports**

Name	Direction	Description
UFC_TX_REQ	Input	Asserted (active-High) to request a UFC message be sent to the channel partner. Requests are processed after a single cycle, unless another UFC message is in progress and not on its last cycle. After a request, the S_AXI_UFC_TX_TDATA bus will be ready to send data within two cycles unless interrupted by a higher priority event.
UFC_TX_MS[0:7]	Input	Specifies the number of bytes in the UFC message (the message size). The max UFC message size is 256.
S_AXI_UFC_TX_TREADY	Output	Asserted (active-High) when an Aurora 64B/66B core is ready to read data from the S_AXI_UFC_TX_TDATA interface. This signal is asserted one clock cycle after UFC_TX_REQ is asserted and no high priority requests in progress. S_AXI_UFC_TX_TREADY continues to be asserted while the core waits for data for the most recently requested UFC message. The signal is deasserted for CC, CB, and NFC requests, which are higher priority. While S_AXI_UFC_TX_TREADY is asserted, S_AXI_TX_TREADY is deasserted.
S_AXI_UFC_TX_TDATA[0:(64*n*-1)]	Input	Input bus for UFC message data to the Aurora channel. Data is read from the bus into the channel only when both S_AXI_UFC_TX_TVALID and S_AXI_UFC_TX_TREADY are asserted on a positive USER_CLK edge. If the number of bytes in the message is not an integer multiple of the bytes in the bus, on the last cycle, only the bytes needed to finish the message starting from the left of the bus are used.
S_AXI_UFC_TX_TVALID	Input	Asserted (active-High) when data on S_AXI_UFC_TX_TDATA is valid. If deasserted while S_AXI_UFC_TX_TREADY is asserted, Idle blocks are inserted in the UFC message.
M_AXI_UFC_RX_TDATA[0:(64*n*-1)]	Output	Incoming UFC message data from the channel partner.
M_AXI_UFC_RX_TVALID	Output	Asserted (active-High) when the values on the M_AXI_UFC_RX_TDATA port is valid. When this signal is not asserted, all values on the M_AXI_UFC_RX_TDATA port should be ignored.
M_AXI_UFC_RX_TLAST	Output	Signals (active-High) the end of the incoming UFC message.
M_AXI_UFC_RX_TSTRB[0:(8*n*-1)]	Output	Specifies the number of valid bytes of data presented on the M_AXI_UFC_RX_TDATA port on the last word of a UFC message. Valid only when M_AXI_UFC_RX_TLAST is asserted. Max size of UFC is 256 bytes.

Transmitting UFC Messages

UFC messages can carry from 1 to 256 data bytes. The user application specifies the length of the message by driving the number of bytes required minus one on the UFC_TX_MS port.

To send a UFC message, the user application asserts UFC_TX_REQ while driving the UFC_TX_MS port with the desired SIZE code for a single cycle. After a request, a new request cannot be made until S_AXI_UFC_TX_TREADY is asserted for the final cycle of the previous request. The data for the UFC message must be placed on the S_AXI_UFC_TX_TDATA port and the S_AXI_UFC_TX_TVALID signal must be asserted whenever the bus contains valid message data. The core deasserts S_AXI_TX_TREADY while sending UFC data, and keeps S_AXI_UFC_TX_TREADY asserted until it has enough data to complete the message that was requested. If S_AXI_UFC_TX_TVALID is deasserted during a UFC message, Idles are sent in the channel, S_AXI_TX_TREADY remains deasserted, and S_AXI_UFC_TX_TREADY remains asserted. If a CC request, CB request, or NFC request is made to the core, S_AXI_UFC_TX_TREADY is deasserted while the requested operation is performed, since CC, CB, and NFC have higher priority.

Example A: Transmitting a Single-Cycle UFC Message

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

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

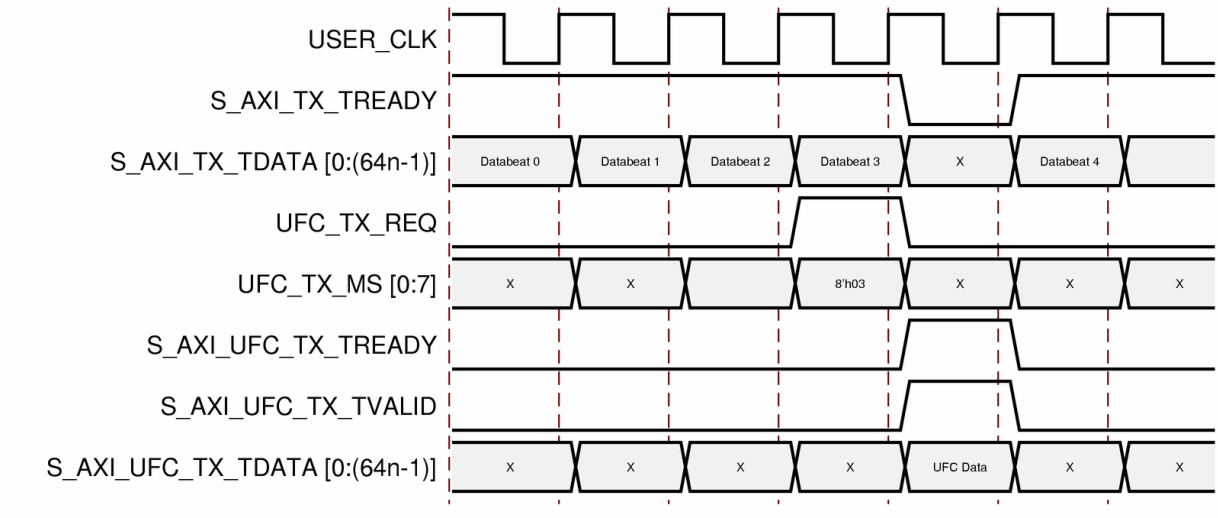


Figure 4-4: **Transmitting a Single-Cycle UFC Message**

Example B: Transmitting a Multi-Cycle UFC Message

The procedure for transmitting a two-cycle UCF message is shown in Figure 4-5, page 48. In this case the user application is sending a 16-byte message using an 8-byte interface.

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

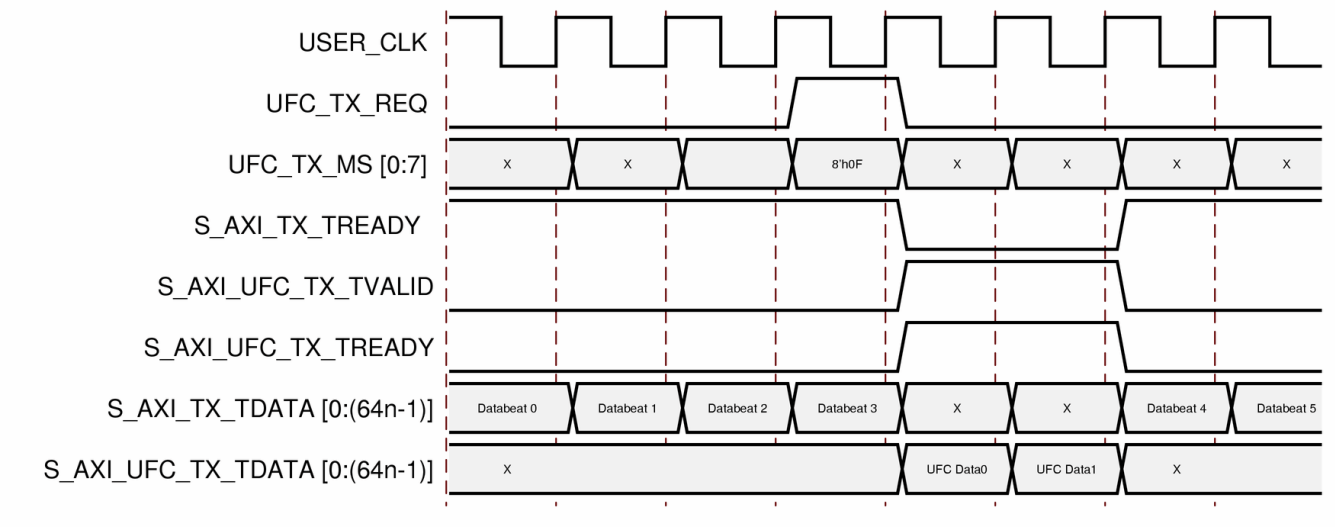


Figure 4-5: **Transmitting a Multi-Cycle UFC Message**

Receiving User Flow Control Messages

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

Example C: Receiving a Single-Cycle UFC Message

Figure 4-6 shows an Aurora 64B/66B core with an 8-byte data interface receiving a 4-byte UFC message. The core presents this data to the user application by asserting M_AXI_UFC_RX_TVALID and M_AXI_UFC_RX_TLAST to indicate a single cycle frame. The M_AXI_UFC_RX_TSTRB bus is set to 4, indicating only the four most significant bytes of the interface are valid.

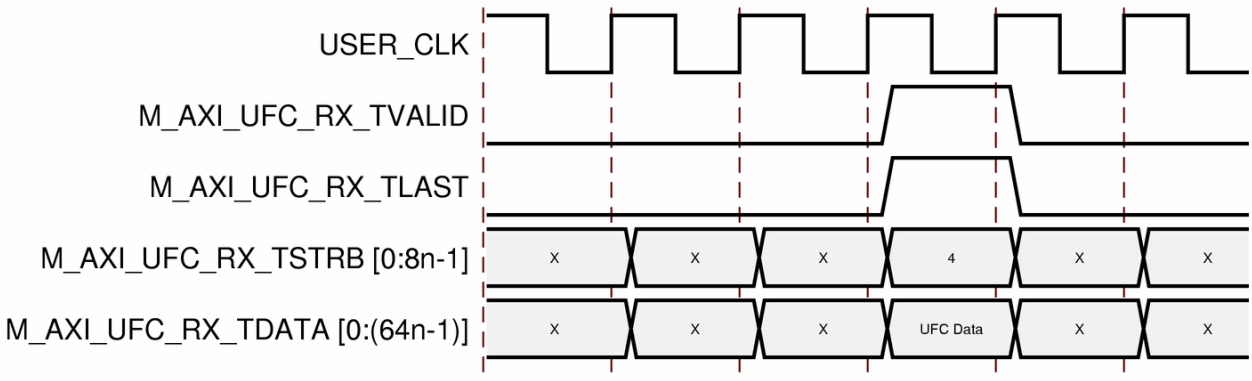


Figure 4-6: **Receiving a Single-Cycle UFC Message**

Example D: Receiving a Multi-Cycle UFC Message

Figure 4-7 shows an Aurora 64B/66B core with an 8-byte interface receiving a 15-byte message.

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

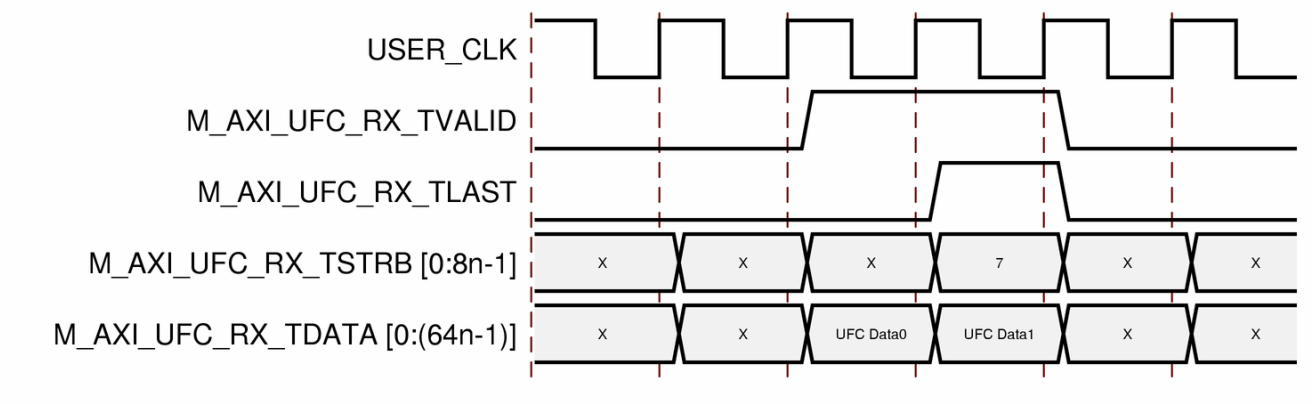


Figure 4-7: **Receiving a Multi-Cycle UFC Message**

Chapter 5

User K-Block Interface

Introduction

This chapter describes short single block data transmission and reception.

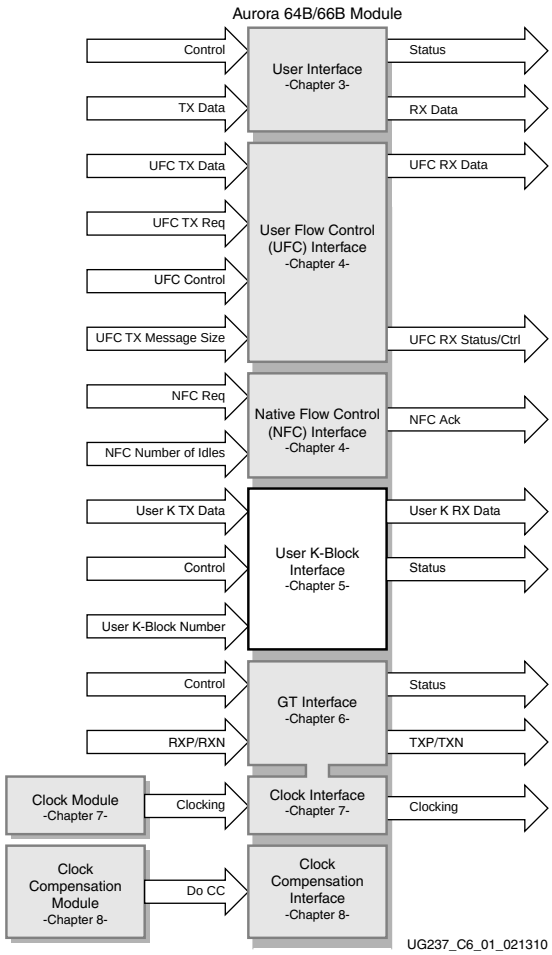


Figure 5-1: **Top-Level User K-Block Interface**

User K-blocks are special single block codes which includes control blocks that are not decoded by the Aurora interface, but are instead passed directly to the user. These blocks can be used to implement application specific control functions. There are nine available User K-blocks (Table 5-1). They have priority next to UFC but higher than user data.

Table 5-1: **Valid Block Type Field Values for User K-Block**

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

The User K-block is not differentiated for streaming or framing designs. Each block code of User K is eight bytes wide and is encoded with a User K BTF, which is indicated by the user in S_AXI_USER_K_TX_TDATA[4:7] as User K Block No. The User K-block is a single block code and is always delineated by User K Block No. User should provide the User K Block No as specified in the Table 5-2. It can have only seven bytes of S_AXI_USER_K_TDATA.

Table 5-2 lists the ports for the User K-block interface.

Table 5-2: **User K-Block I/O Ports**

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

Transmitting User K-Block

The S_AXI_USER_K_TX_TREADY is asserted by Aurora and is prioritized by CC, CB, NFC and UFC. After placing the S_AXI_USER_K_TX_TDATA and along with User K Block No and the S_AXI_USER_K_TX_TVALID is asserted, the user can change the S_AXI_USER_K_TX_TDATA if required when S_AXI_USER_K_TX_TREADY is asserted (Figure 5-2). This enables Aurora to select appropriate User K BTF among the nine User K-blocks. The data available during assertion of the S_AXI_USER_K_TX_TREADY is always serviced.

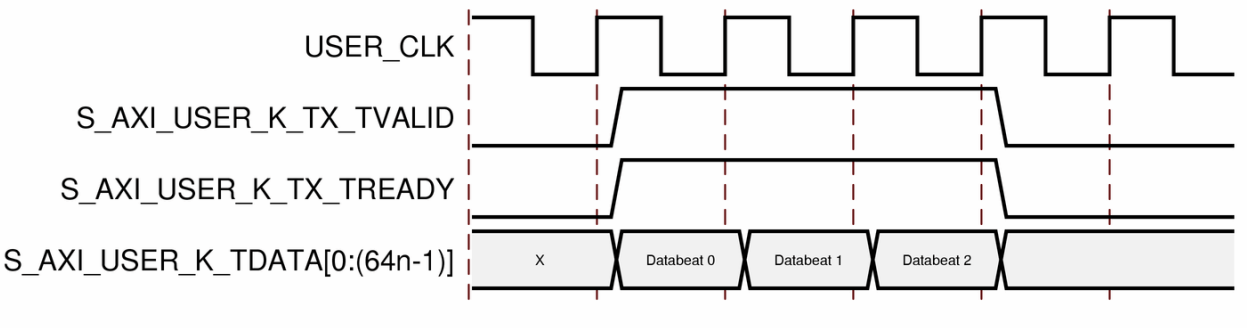


Figure 5-2: **Transmitting User K Data and User K-Block Number**

Receiving User K-Block

The receive BTF is decoded and the block number for the corresponding BTF is passed on to the user as such (Figure 5-3). The user can validate the M_AXI_USER_K_RX_TDATA available on the bus when M_AXI_USER_K_RX_TVALID is asserted.

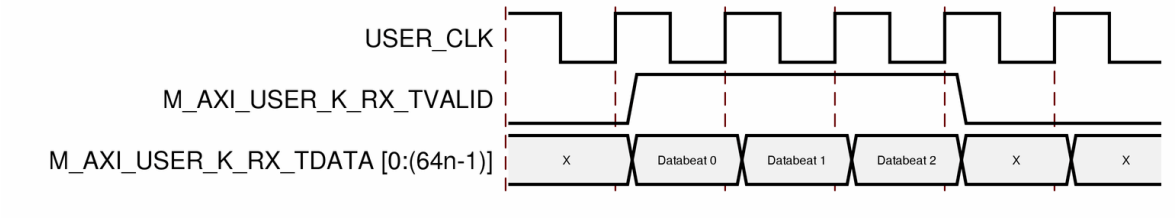


Figure 5-3: **Receiving User K Data and User K-Block Number**

Chapter 6

Status, Control, and the GT Interface

Introduction

The status and control ports of the Aurora 64B/66B core allow user applications to monitor the Aurora channel and use built-in features of the GT interface.

This chapter provides diagrams and port descriptions for the Aurora 64B/66B core's status and control interface, along with the GTX/GTH serial I/O interface.

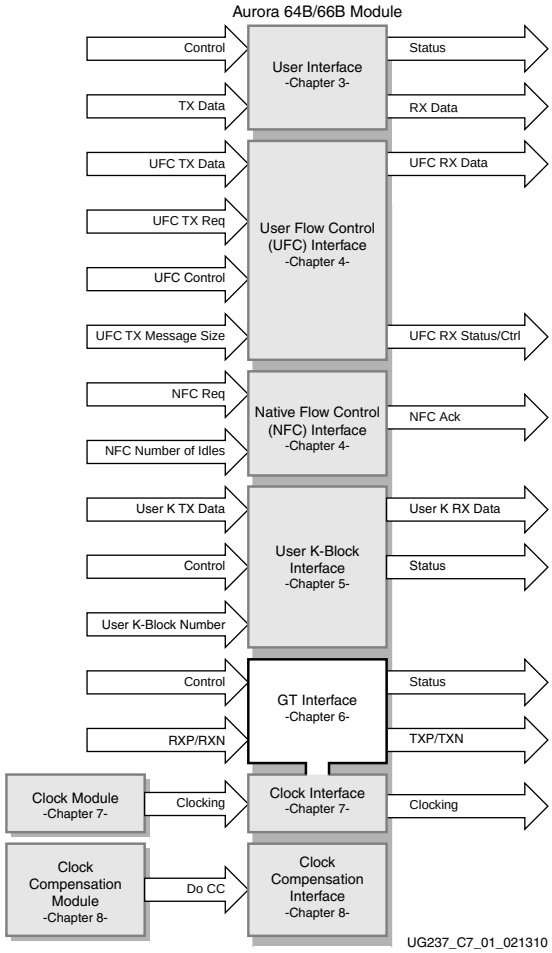


Figure 6-1: **Top-Level GTX Interface**

Status and Control Ports

Aurora 64B/66B cores are full-duplex/simplex, and provide a TX and an RX Aurora channel connection. The Aurora 64B/66B core does not require any side band signals for simplex mode of operation. Figure 6-2 shows the status and control interface for an Aurora 64B/66B core. Table 6-1 describes the function of each of the ports in the interface.

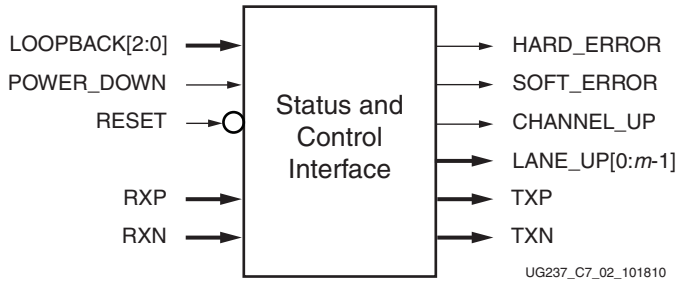


Figure 6-2: **Status and Control Interface for the Aurora 64B/66B Core**

Table 6-1: **Status and Control Ports for Full-Duplex Cores**

Name	Direction	Description
CHANNEL_UP	Output	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
LANE_UP[0:*m*-1](1)	Output	Asserted (active-High) for each lane upon successful lane initialization, with each bit representing one lane. The Aurora 64B/66B core can only receive data after all LANE_UP signals are asserted.
HARD_ERROR	Output	Hard error detected (active-High, asserted until Aurora 64B/66B core resets). See Error Signals in Aurora 64B/66B Cores, page 58 for more details.
LOOPBACK[2:0]	Input	See the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, the *Virtex-6 FPGA GTX Transceivers User Guide*, and the *Virtex-6 FPGA GTH Transceivers User Guide* for details about loopback. See Related Documents in Chapter 1.
POWER_DOWN	Input	Drives the power-down input to the GTX/GTH transceiver (active-High).
RESET	Input	Resets the Aurora 64B/66B core (active-High).
SOFT_ERROR	Output	Soft error detected in the incoming serial stream. See Error Signals in Aurora 64B/66B Cores, page 58 for more details. (active-High, asserted for a single clock).
RXP[0:*m*-1]	Input	Positive differential serial data input pin.
RXN[0:*m*-1]	Input	Negative differential serial data input pin.
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
TXN[0:*m*-1]	Output	Negative differential serial data output pin.

Notes:

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

Table 6-2: **Status and Control Ports for Simplex-TX Cores**

Name	Direction	Description
TX_CHANNEL_UP	Output	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
TX_LANE_UP[0:*m*-1](1)	Output	Asserted (active-High) for each lane upon successful lane initialization, with each bit representing one lane. The Aurora 64B/66B core can only transmit data after all TX_LANE_UP signals are asserted.
TX_HARD_ERROR	Output	Hard error detected (active-High, asserted until Aurora 64B/66B core resets). See Error Signals in Aurora 64B/66B Cores, page 58 for more details.
POWER_DOWN	Input	Drives the power-down input to the GTX/GTH transceiver (active-High).
TX_SYSTEM_RESET	Input	Resets the Aurora 64B/66B core (active-High).
TX_SOFT_ERROR	Output	Soft error detected in the transmit logic. See Error Signals in Aurora 64B/66B Cores, page 58 for more details. (active-High, asserted for a single clock).
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
TXN[0:*m*-1]	Output	Negative differential serial data output pin.

Notes:

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

Table 6-3: **Status and Control Ports for Simplex-RX Cores**

Name	Direction	Direction
RX_CHANNEL_UP	Output	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to receive data.
RX_LANE_UP[0:*m*-1](1)	Output	Asserted (active-High) for each lane upon successful lane initialization, with each bit representing one. The Aurora 64B/66B core can only receive data after all RX_LANE_UP signals are asserted.
RX_HARD_ERROR	Output	Hard error detected (active-High, asserted until Aurora 64B/66B core resets). SeeError Signals in Aurora 64B/66B Cores, page 58 for more details.
POWER_DOWN	Input	Drives the power-down input to the GTX/GTH transceiver (active-High).
RX_SYSTEM_RESET	Input	Resets the Aurora 64B/66B core (active-High).
RX_SOFT_ERROR	Output	Soft error detected in the receive logic. See Error Signals in Aurora 64B/66B Cores, page 58 for more details. (active-High, asserted for a single clock).
RXP[0:*m*-1]	Input	Positive differential serial data input pin.
RXN[0:*m*-1]	Input	Negative differential serial data input pin.

Notes:

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

Error Signals in Aurora 64B/66B Cores

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

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

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

Table 6-4 summarizes the error conditions that the Aurora 64B/66B core can detect and the error signals used to alert the user application.

Table 6-4: **Error Signals in Full-Duplex Cores**

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

Initialization

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

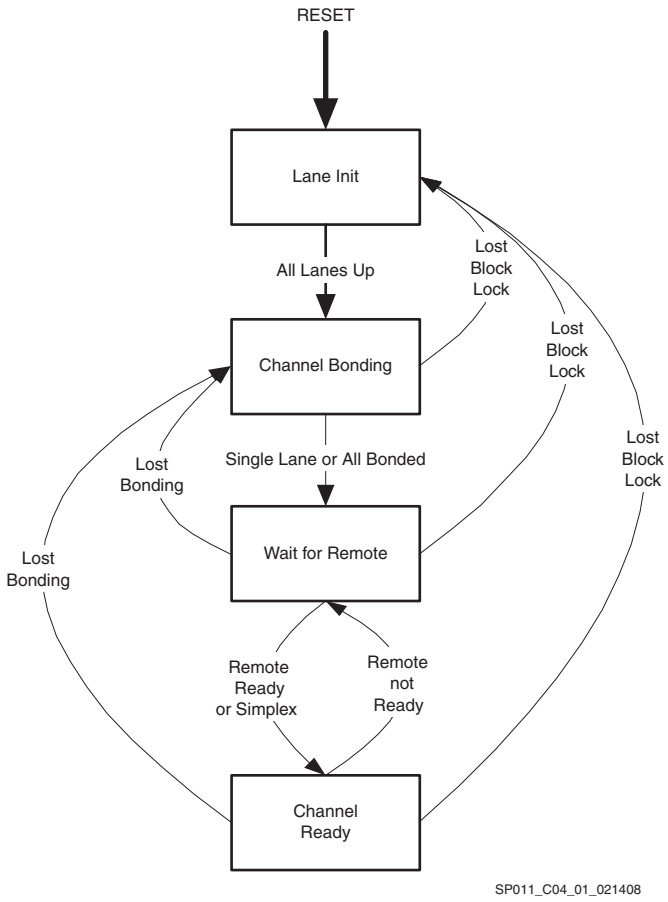


Figure 6-3: **Initialization Overview**

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

Simplex Aurora 64B/66B cores do not have any sideband connection and use timers to declare that the partner is out of initialization and is ready for data transfer. These simplex cores transmit periodic channel bonding characters to ensure the channel partner is bonded. If at any time during the data transfer, the link is broken or reinitialized, the

channel will auto recover after the periodic channel bond character is sent to the partner and does not require any reset.

After the link is broken, the Aurora 64B66B initialization state machine (Figure 6-3, page 59) will move from Channel Ready state to Wait for Remote state. It waits in this state until the periodic Channel Bond character is received by the partner and then moves to the Channel Ready state. The data transferred during the reinitialization process will be lost.

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

Reset and Power Down

Reset

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

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

Power Down

When POWER_DOWN is asserted, the GTX/GTH transceivers in the Aurora 64B/66B core are turned off, putting them into a non-operating low-power mode. When POWER_DOWN is deasserted, the core automatically resets. Be careful when asserting this signal on cores that use TX_OUT_CLK (see Chapter 7, Clock Interface and Clocking). TX_OUT_CLK stops when the GTX/GTH transceivers are powered down. See the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, *Virtex-6 FPGA GTX Transceivers User Guide*, and the *Virtex-6 FPGA GTH Transceivers User Guide* for details about powering down GTX/GTH transceivers.

Timing

Figure 6-4 shows the timing for the RESET signal. In a quiet environment, t_{CU} is generally less than 500 clocks; In a noisy environment, t_{CU} can be much longer.

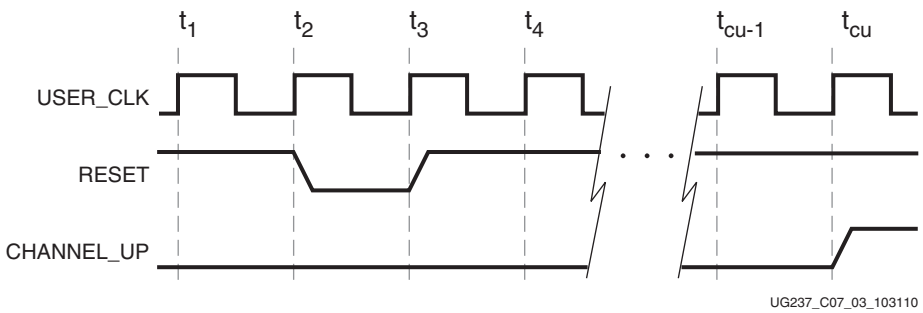


Figure 6-4: **Reset and Power Down Timing**

Chapter 7

Clock Interface and Clocking

Introduction

Good clocking is critical for the correct operation of the Virtex®-5 and Virtex-6 FPGA Aurora 64B/66B core. The core requires a low-jitter reference clock to drive the high-speed TX clock and clock recovery circuits in the GTX/GTH transceiver. It also requires at least one frequency locked parallel clock for synchronous operation with the user application.

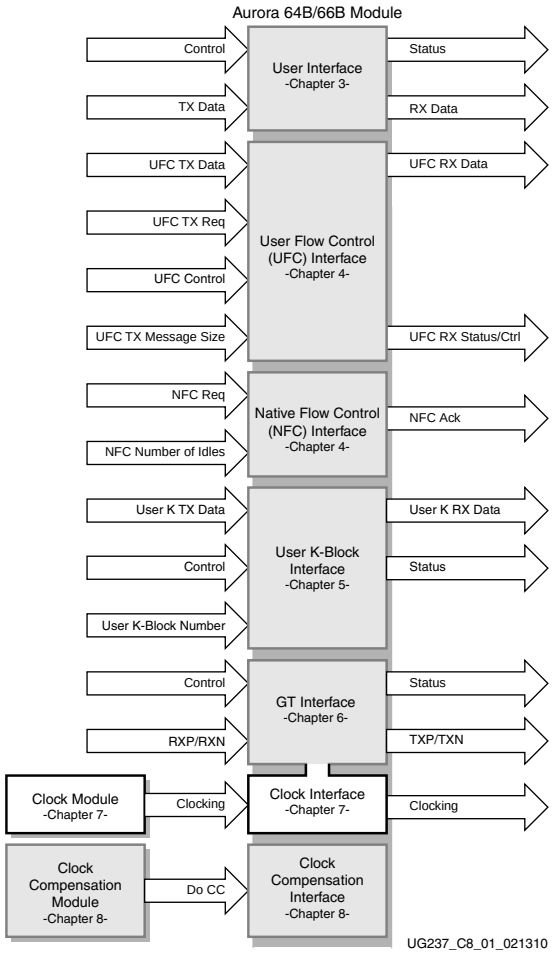


Figure 7-1: **Top-Level Clocking**

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

Clock Interface Ports for Virtex-5 and Virtex-6 FPGA Cores

Table 7-1 describes the Virtex-5 and Virtex-6 FPGA Aurora 64B/66B core clock ports. In GTX/GTH transceiver designs, the reference clock can be from GTXD/GTXQ/GTHQ, which is a differential input clock for each GT tile. The reference clock for GT tile is provided through the CLKIN port.

Table 7-1: **Clock Ports for a Virtex-5 and Virtex-6 FPGA Aurora 64B/66B Core**

Name	Direction	Description
DCM_NOT_LOCKED/ MMCM_NOT_LOCKED	Input	If a PLL is used to generate clocks for the Aurora 64B/66B core, the DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal should be connected to the inverse of the PLL LOCKED signal. The clock modules provided with the Aurora 64B/66B core use the PLL for clock division. The DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal from the clock module should be connected to the DCM_NOT_LOCKED/ MMCM_NOT_LOCKED signal on the Aurora 64B/66B core.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application. The USER_CLK is the output of a BUFG whose input is derived from TX_OUT_CLK. The rate is one fourth the rate of TX_OUT_CLK, which is determined by the clock rate settings of the RocketIO transceivers in the core.
TX_OUT_CLK	Output	Clock signal from Virtex-5/Virtex-6 FPGA GTX/GTH transceivers. The GTX/GTH transceiver generates TX_OUT_CLK from its reference clock based on its PLL speed setting. This clock, when buffered, should be used as the user clock for logic connected to the Aurora 64B/66B core.
SHIM_CLK	Input	Parallel clock used by the internal synchronization logic of the RocketIO transceivers in the Aurora 64B/66B core. SHIM_CLK is double the rate of USER_CLK. ***Note:*** This clock is not available for Virtex-6 FPGA GTH transceivers.
SYNC_CLK	Input	Parallel clock used by internal synchronization logic of the RocketIO transceivers in the Aurora 64B/66B core. SYNC_CLK is double the rate of SHIM_CLK.
PLL_LOCK	Output	Active-High, asserted when TX_OUT_CLK is stable. When this signal is deasserted (Low), circuits using TX_OUT_CLK should be held in reset.

Parallel Clocks for Virtex-5/Virtex-6 FPGA Designs

Connecting USER_CLK, SYNC_CLK, SHIM_CLK, and TX_OUT_CLK

The Virtex-5/Virtex-6 FPGA Aurora 64B/66B cores use three phase-locked parallel clocks. The first is USER_CLK, which synchronizes all signals between the core and the user application. All logic touching the core must be driven by USER_CLK, which in turn must be the output of a global clock buffer (BUFG). USER_CLK runs at one-fourth the rate of TX_OUT_CLK, which is determined by the Clock Settings for the design. The

TX_OUT_CLK is selected so that the data rate of the parallel side of the module matches the data rate of the serial side of the module, taking into account 64B/66B encoding and decoding.

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

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

To make it easier to use the two parallel clocks, a clock module is provided in a subdirectory called clock_module under example_design. The ports for this module are described in Table 7-1, page 62; If the clock module is used, the DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal should be connected to the DCM_NOT_LOCKED/MMCM_NOT_LOCKED output of the clock module, TX_OUT_CLK should connect to the clock module's CLK port, and PLL_LOCK should connect to the clock module's PLL_NOT_LOCKED port. If the clock module is not used, connect the DCM_NOT_LOCKED/MMCM_NOT_LOCKED signal to the inverse of the LOCKED signal from any PLL used to generate either of the parallel clocks, and use the PLL_LOCK signal to hold the PLLs in reset during stabilization if TX_OUT_CLK is used as the PLL's source clock.

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

The Aurora 64B/66B core uses 6 BUFG for a given core configuration using GTX transceivers. The Aurora 64B/66B is a 8Byte aligned protocol, and the datapath from the user interface is 8bytes aligned. For GTX based devices wherein the datapath is 4bytes max, the core generates respective clocks (USER_CLK, SHIM_CLK and SYNC_CLK). Also the CB/CC logic is internal to the core, which is primarily based on the received recovered clock from the GT. Hence the Aurora 64B/66B core uses 3 BUFGs each in TX and RX for clock routing.

Reference Clocks for Virtex-5/Virtex-6 FPGA Designs

Aurora 64B/66B cores require low-jitter reference clocks for generating and recovering high-speed serial clocks in the GTX/GTH transceivers. Each reference clock can be set to Virtex-5/Virtex-6 FPGA reference clock input ports: GTXD/GTXQ/GTHQ. Reference clocks should be driven with high-quality clock sources whenever possible to decrease jitter and prevent bit errors. DCMs should never be used to drive reference clocks, because they introduce too much jitter.

For multi-lane designs, the Aurora 64B/66B wizard allows selecting clocks three tiles above and three tiles below the selected tile per north-south clocking criteria. A second reference clock source can be selected if the tile selection exceeds the 7-tile boundary. For details on north-south clocking, see the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, the *Virtex-6 FPGA GTX Transceivers User Guide*, and the *Virtex-6 FPGA GTH Transceivers User Guide*.

Chapter 8

Clock Compensation Interface

Introduction

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

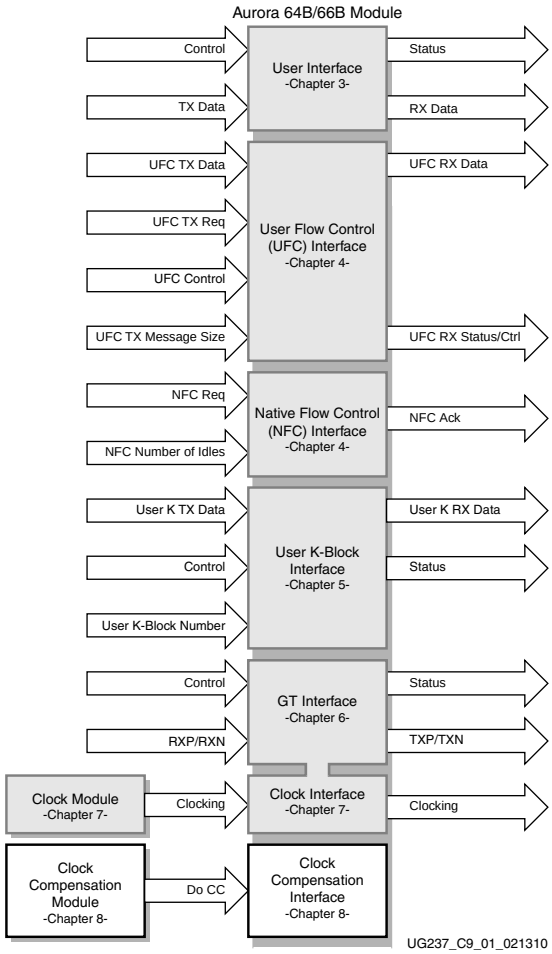


Figure 8-1: **Top-Level Clock Compensation Interface**

The Aurora 64B/66B core's clock compensation interface enables full control over the core's clock compensation features. A standard clock compensation module is generated

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

Clock Compensation Interface

All Aurora 64B/66B cores include a clock compensation interface for controlling the transmission of clock compensation sequences. Table 8-1 describes the function of the clock compensation interface ports.

Table 8-1: **Clock Compensation I/O Ports**

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

Figure 8-2 and Figure 8-3 are waveform diagrams showing how the DO_CC signal works.

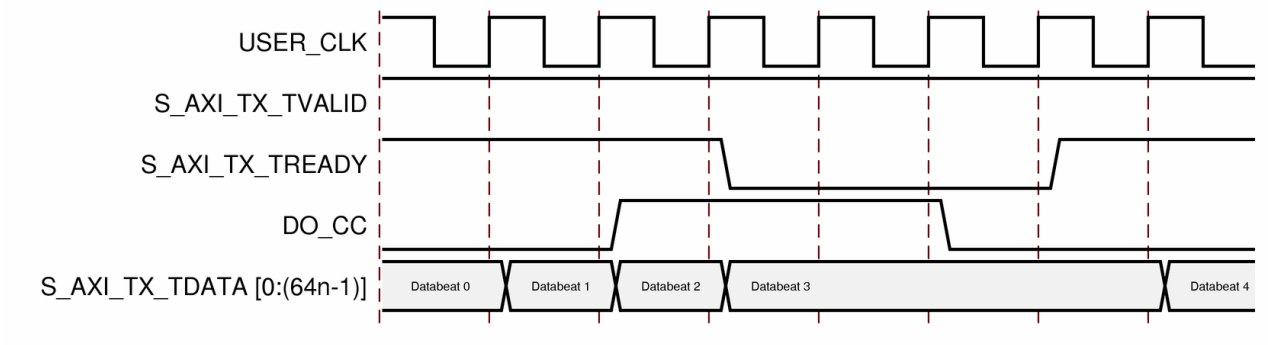


Figure 8-2: **Streaming Data with Clock Compensation Inserted**

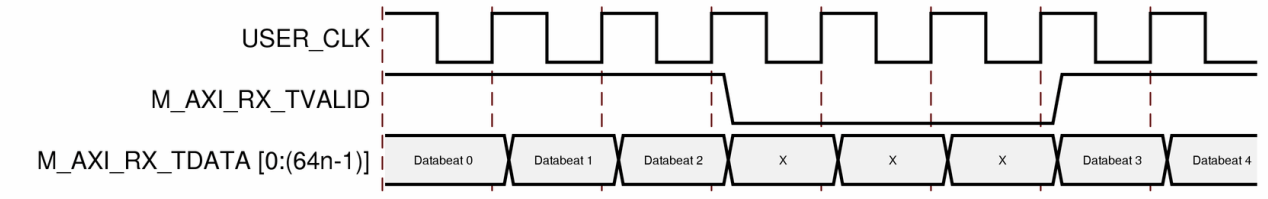


Figure 8-3: **Data Reception Interrupted by Clock Compensation**

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

A standard clock compensation module is generated along with each Aurora 64B/66B core from the CORE Generator™ software, in the cc_manager subdirectory under example_design. It automatically generates pulses to create Aurora compliant clock compensation sequences on the DO_CC port. This module always be connected to the clock compensation port on the Aurora module, except in special cases. Table 8-2 shows the port description for the standard CC module.

Table 8-2: **Standard CC I/O Port**

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

Clock compensation is not needed when both sides of the Aurora channel are being driven by the same clock (see Figure 8-3, page 66) because the reference clock frequencies on both sides of the module are locked. In this case, DO_CC should both be tied to ground.

Other special cases when the standard clock compensation module is not appropriate are possible. The DO_CC port can be used to send clock compensation sequences at any time, for any duration to meet the needs of specific channels. The most common use of this feature is scheduling clock compensation events to occur outside of frames, or at specific times during a stream to avoid interrupting data flow. In general, customizing the clock compensation logic is not recommended, and when it is attempted, it should be performed with careful analysis, testing, and consideration of the following guidelines:

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

XILINX®

Chapter 9

Quick Start Example Design

This chapter introduces the example design that is included with the LogiCORE™ IP Aurora 64B/66B core.

The quick start instructions provides 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 Chapter 11, Project Directory Structure.

Overview

The quick start example design consists of the following 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
  - Virtex®-5 FPGA/Virtex-6 FPGA target device
- A top-level example design (aurora_64b66b_v5_1_example_design) with a user constraints file (UCF) for the ML523 or ML623 board
- A demonstration test bench to simulate two instances of the example design

Generating the Core

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

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

   For help starting and using the CORE Generator software, see CORE Generator Help in the ISE® software documentation.

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

   On the Part tab, for Family select Virtex5. For Device, select an appropriate device that supports GTX transceivers, such as xc5vfx70t.

   Note: If an unsupported silicon family is selected, the Aurora 64B/66B appears light grey in the taxonomy tree and cannot be customized. Only devices containing GTX/GTH transceivers are supported by the core. For a list of supported architectures, see DS815, *Aurora 64B/66B v5.1 Data Sheet*.

   No further project options need to be set.

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

5. After creating the project, locate the Aurora 64B/66B core v5.1 in the taxonomy tree under:

   `/Communication_&_Networking/Serial_Interfaces`

6. Double-click the core for generation.

The customization screens are shown in Figure 9-1 and Figure 9-2, page 71.

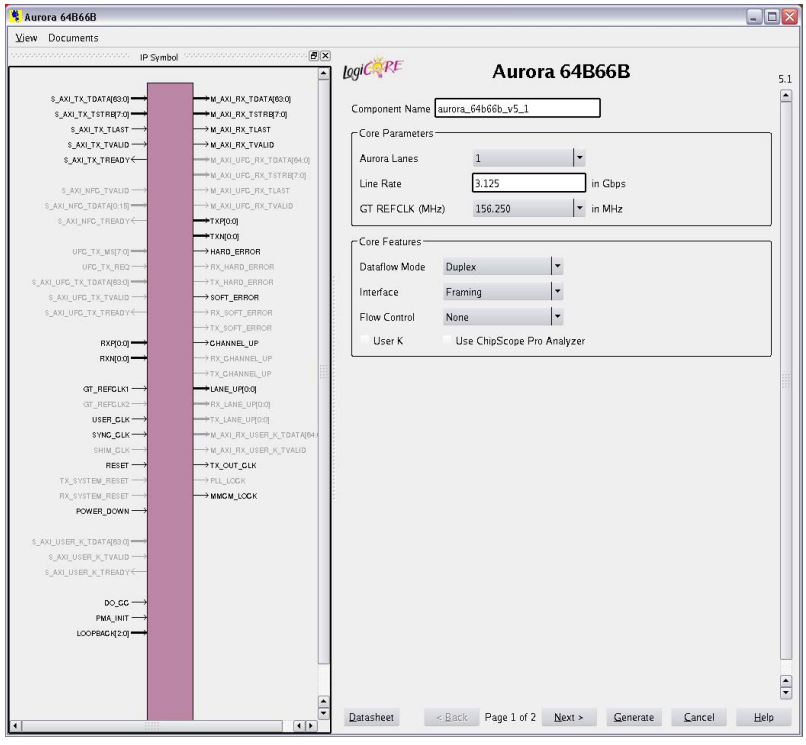


Figure 9-1: **CORE Generator Tool Aurora 64B/66B Customization Screen - Page 1**

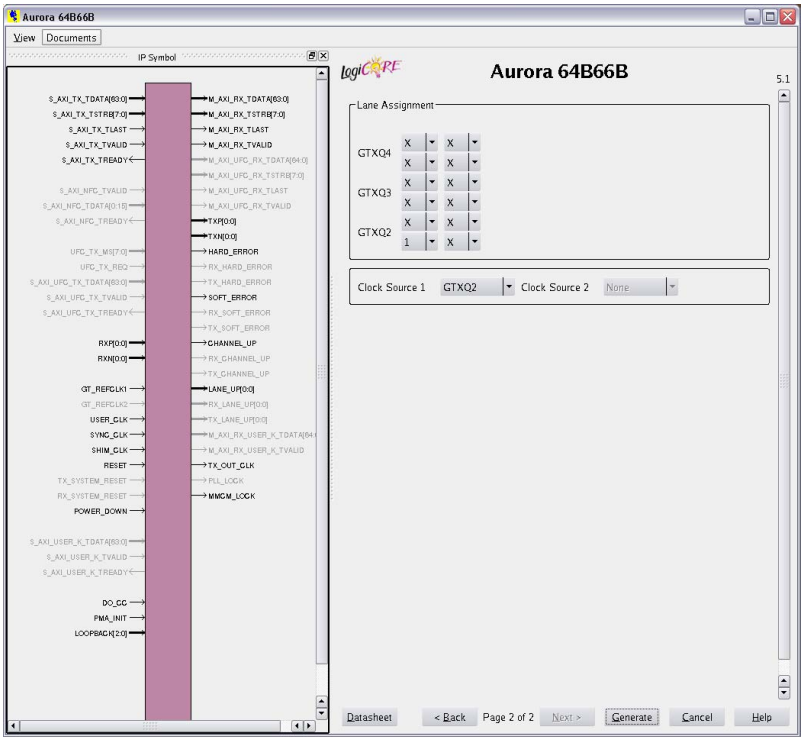


Figure 9-2: **CORE Generator Tool Aurora 64B/66B Customization Screen - Page 2**

7. In the Component Name field, enter a name for the core instance. This example uses the name **aurora_64b66b_v5_1**.
8. Click **Generate**.

The core and its supporting files, including the example design, are generated in the project directory. For detailed information about the example design files and directories, see Chapter 11, Project Directory Structure.

Simulating the Example Design

The Aurora 64B/66B core provides a quick way to simulate and observe the behavior of the core using the provided example design. Prior to simulating the core, the functional (gate-level) simulation models must be generated. You must compile all source files in the following directories to a single library as shown in Table 9-1. Refer to the *Synthesis and Verification Design Guide* for ISE 12.4 software for instructions on how to compile ISE software simulation libraries.

Table 9-1: **Required Simulation Libraries**

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

Notes:

1. SIMULATOR can be Modelsim.

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

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

   <*project directory*>/aurora_64b66b_v5_1/simulation/functional

2. Set the MTI_LIBS variable:

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

3. Launch the simulation script:

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

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

Implementing the Example Design

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

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

For Windows

```
ms-dos> cd aurora_64b66b_v5_1\implement
ms-dos> .\implement.bat
```

For Linux

```
% cd aurora_64b66b_v5_1/implement
%./implement.sh
```

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

Using ChipScope Pro Cores with the Aurora 64B/66B Core

Description

The ChipScope™ Pro ICON, ILA, and VIO cores aid in debugging and validating the design in board and are provided with the Aurora 64B/66B core. Select the CHIPSCOPE option from the core GUI (see Figure 9-1, page 70) to include it as a part of the example design.

Chapter 10

Example Design Overview

Introduction

Each Aurora 64B/66B core includes an example design (aurora_example) that uses the core in a simple data transfer system. For more details about the example_design directory, see Chapter 11, Project Directory Structure.

The example design consists of two main components:

- Frame generator (FRAME_GEN, page 77) connected to the TX interface
- Frame checked (FRAME_CHECK, page 84) connected to the RX user interface

Figure 10-1 shows a block diagram of the example design for a full-duplex core. Table 10-1, page 76 describes the ports of the example design.

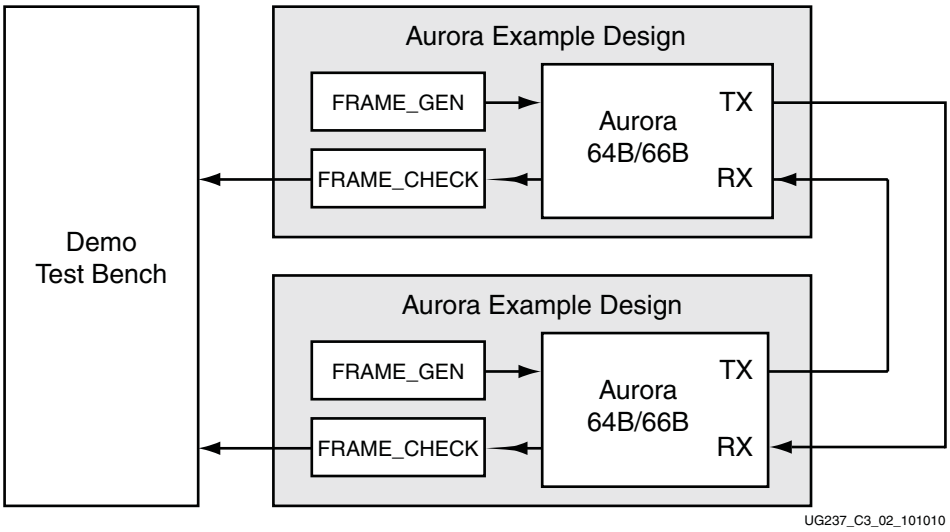


Figure 10-1: **Example Design**

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

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

When using the example design on a board, be sure to edit the <component name>_example_design.ucf file in the ucf subdirectory to supply the correct pins and clock constraints. Table 10-1 describes the ports available in the example design.

Table 10-1: **Example Design I/O Ports**

Port	Direction	Description
RXN[0:*m*-1]	Input	Negative differential serial data input pin.
RXP[0:*m*-1]	Input	Positive differential serial data input pin.
TXN[0:*m*-1]	Output	Negative differential serial data output pin.
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
RESET	Input	Reset signal for the example design. The reset is debounced using a USER_CLK signal generated from the reference clock input.
<reference clock(s)>	Input	The reference clocks for the Aurora 64B/66B core are brought to the top level of the example design. See Chapter 7, Clock Interface and Clocking for details about the reference clocks.
<core error signals>	Output	The error signals from the Aurora 64B/66B core's Status and Control interface are brought to the top level of the example design and registered. See Chapter 6, Status, Control, and the GT Interface for details.
<core channel up signals>	Output	The channel up status signals for the core are brought to the top level of the example design and registered. See Chapter 6, Status, Control, and the GT Interface for details.
<core lane up signals>	Output	The lane up status signals for the core are brought to the top level of the example design and registered. Cores have a lane up signal for each GTX/GTH transceiver they use. See Chapter 6, Status, Control, and the GT Interface for details.
PMA_INIT	Input	The reset signal for the PCS and PMA modules in the GTX/GTH transceivers is connected to the top level through a debouncer. The signal is debounced using the INIT_CLK. See the Reset section in *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*, *Virtex-6 FPGA GTX Transceivers User Guide*, or *Virtex-6 FPGA GTH Transceivers User Guide*for further details on GT RESET.
INIT_CLK	Input	INIT_CLK is used to register and debounce the PMA_INIT signal. INIT_CLK must not come from a GTX/GTH transceiver, and should be set to a slow rate, preferably slower than the reference clock.
DATA_ERROR_COUNT[0:7]	Output	Count of the number of frame data words received by the FRAME_CHECK that did not match the expected value.
UFC_ERROR	Output	Asserted (active-High) when UFC data words received by the FRAME_CHECK that did not match the expected value.
USER_K_ERROR	Output	Asserted (active-High) when User K data words received by the FRAME_CHECK that did not match the expected value.

FRAME_GEN

Framing TX Data Interface

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

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

Figure 10-2 shows the FRAME_GEN framing user interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for TX data.

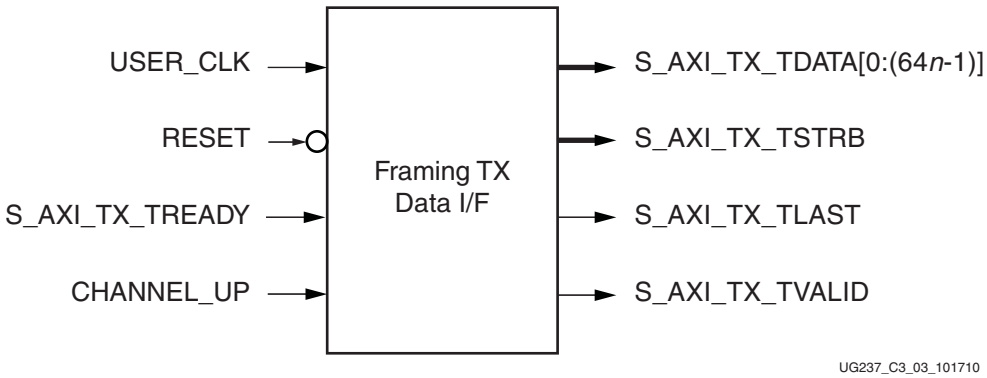


Figure 10-2: **Aurora 64B/66B Core Framing TX Data Interface (FRAME_GEN)**

Table 10-2 lists the FRAME_GEN framing TX data ports and their descriptions.

Table 10-2: **FRAME_GEN Framing User I/O Ports (TX)**

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

Streaming TX Data Interface

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

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

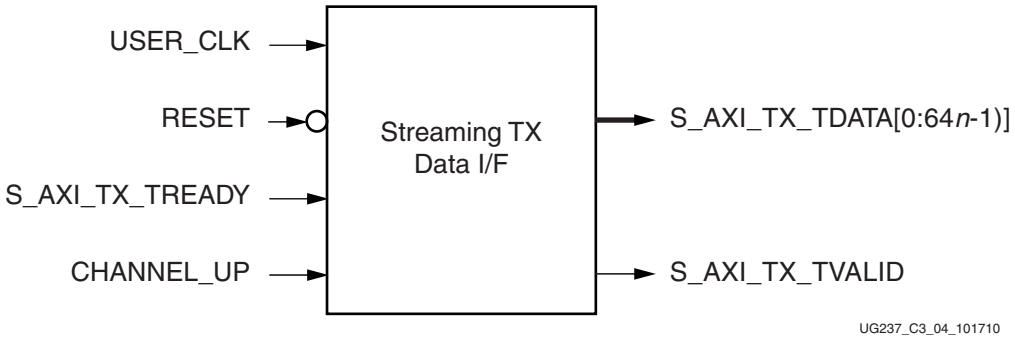


Figure 10-3: **Aurora 64B/66B Core Streaming TX Data Interface (FRAME_GEN)**

Table 10-3 lists the FRAME_GEN streaming TX data ports and their descriptions.

Table 10-3: **FRAME_GEN Streaming User I/O Ports (TX)**

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

UFC TX Interface

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

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

Figure 10-4 shows the FRAME_GEN UFC TX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for UFC TX data.

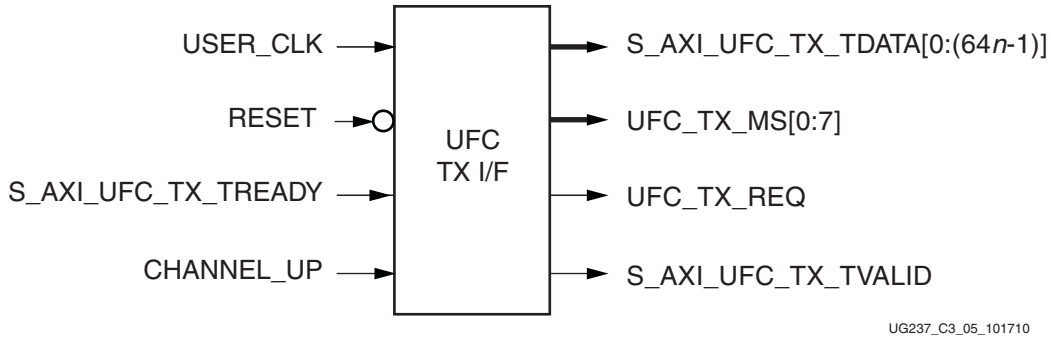


Figure 10-4: **Aurora 64B/66B Core UFC TX Interface (FRAME_GEN)**

Table 10-4 lists the FRAME_GEN UFC TX data ports and their descriptions.

Table 10-4: **FRAME_GEN UFC User I/O Ports (TX)**

Name	Direction	Description
UFC_TX_REQ	Output	Asserted to request a UFC message to be sent to the channel partner (active- High). Requests are processed after a single cycle, unless another UFC message is in progress and not on its last cycle. After a request, the S_AXI_UFC_TX_TDATA bus will be ready to send data within 2 cycles unless interrupted by a higher priority event.
UFC_TX_MS[0:7]	Output	Specifies the number of bytes in the UFC message (the Message Size). The max UFC Message Size is 256.
S_AXI_UFC_TX_TDATA [0:(64*n*-1)]	Output	Output bus for UFC message data to the Aurora channel. Data is read from the bus into the channel only when both S_AXI_UFC_TX_TVALID and S_AXI_UFC_TX_TREADY are asserted on a positive USER_CLK edge. If the number of bytes in the message is not an integer multiple of the bytes in the bus, on the last cycle, only the bytes needed to finish the message starting from the left of the bus are used.
S_AXI_UFC_TX_TVALID	Output	Assert (active-High) when data on S_AXI_UFC_TX_TDATA is valid. If deasserted while S_AXI_UFC_TX_TREADY is asserted, Idle blocks will be inserted in the UFC message.
S_AXI_UFC_TX_TREADY	Input	Asserted (active-High) when an Aurora 64B/66B core is ready to read data from the S_AXI_UFC_TX_TDATA interface. This signal will be asserted one clock cycle after UFC_TX_REQ is asserted and no high priority requests in progress. S_AXI_UFC_TX_TREADY will continue to be asserted while the core waits for data for the most recently requested UFC message. The signal is deasserted for CC and NFC requests, which are higher priority. While S_AXI_UFC_TX_TREADY is asserted, S_AXI_TX_TREADY is deasserted.
CHANNEL_UP	Input	Asserted when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-Low).

NFC TX Interface

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

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

Figure 10-5 shows the FRAME_GEN NFC TX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for NFC TX data.

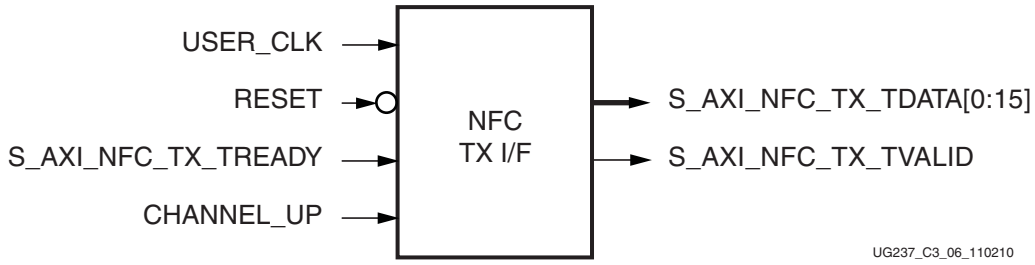


Figure 10-5: **Aurora 64B/66B Core NFC TX Interface (FRAME_GEN)**

Table 10-5 lists the FRAME_GEN NFC TX data ports and their descriptions.

Table 10-5: **FRAME_GEN NFC User I/O Ports (TX)**

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

User K TX Interface

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

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

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

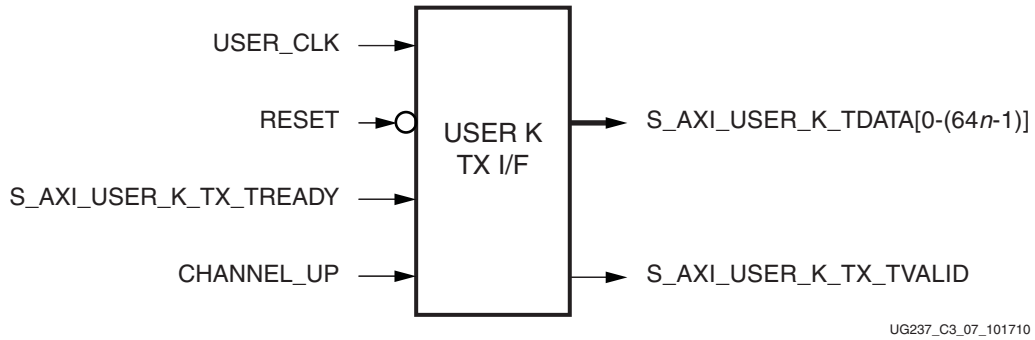


Figure 10-6: **Aurora 64B/66B Core User K TX Interface (FRAME_GEN)**

Table 10-6 lists the FRAME_GEN User K TX data ports and their descriptions.

Table 10-6: **FRAME_GEN User K User I/O Ports (TX)**

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

FRAME_CHECK

Framing RX Data Interface

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

Start the frame when M_AXI_RX_TVALID is asserted

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

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

Note: The counter can be cleared by applying reset.

Figure 10-7 shows the FRAME_CHECK framing user interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for RX data.

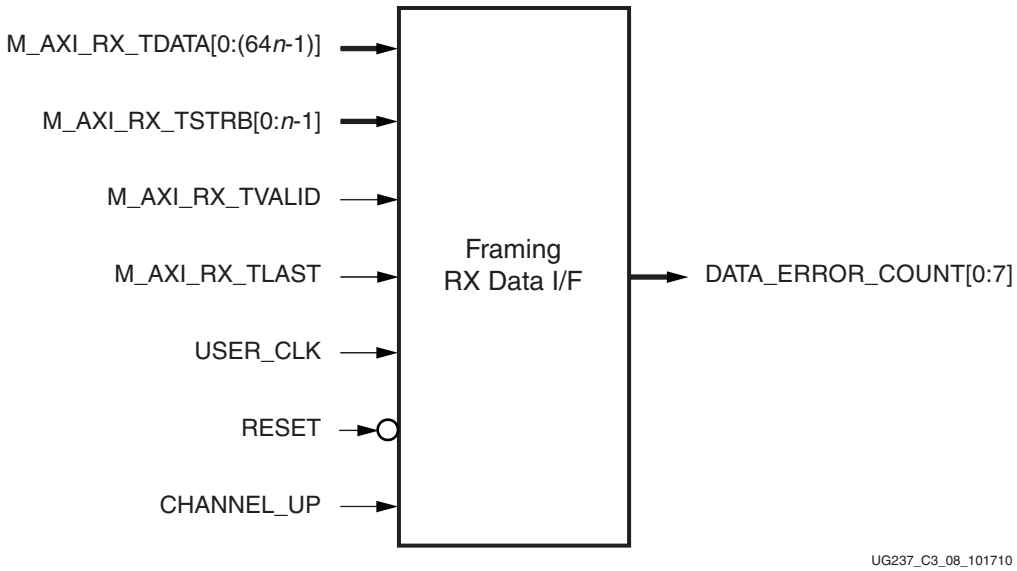


Figure 10-7: **Aurora 64B/66B Core Framing RX Data Interface (FRAME_CHECK)**

Table 10-7 lists the FRAME_CHECK framing RX data ports and their descriptions.

Table 10-7: **FRAME_CHECK Framing User I/O Ports (RX)**

Name	Direction	Description
M_AXI_RX_TDATA[0:(64*n*-1)]	Input	Incoming frame data from channel partner (Ascending bit order).
M_AXI_RX_TSTRB[0:*n*-1]	Input	Specifies the number of valid bytes in the last data beat. Valid only when M_AXI_RX_TLAST is asserted.
M_AXI_RX_TVALID	Input	Asserted (active-High) when data and control signals from an Aurora core are valid. Deasserted (Low) when data and/or control signals from an Aurora core should be ignored.
M_AXI_RX_TLAST	Input	Signals the end of the incoming frame (active-High, asserted for a single USER_CLK cycle).
DATA_ERROR_COUNT[0:7]	Output	Count of the number of RX frame data words received by the frame checker that did not match the expected value.
CHANNEL_UP	Input	Asserted (active-High) when Aurora channel initialization is complete and channel is ready to send data.
USER_CLK	Input	Parallel clock shared by the Aurora 64B/66B core and the user application.
RESET	Input	Resets the Aurora core (active-Low).

Streaming RX Data Interface

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

Figure 10-8 shows the FRAME_CHECK streaming user interface of the Aurora 64B/66B core ports for RX data.

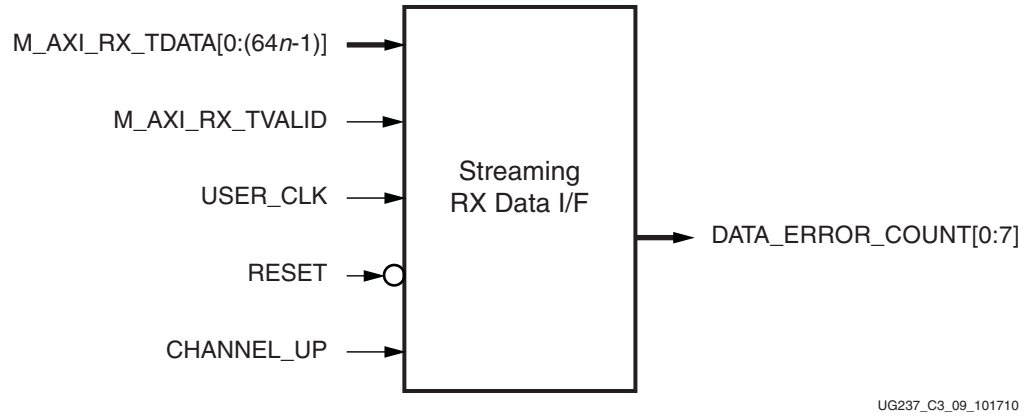


Figure 10-8: **Aurora 64B/66B Core Streaming RX Data Interface (FRAME_CHECK)**

Table 10-8 lists the FRAME_CHECK streaming RX data ports and their descriptions.

Table 10-8: **FRAME_CHECK Streaming User I/O Ports (RX)**

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

UFC RX Interface

- Expected UFC RX data is computed by LFSR.
- Error checking and counter logic is similar to that of Framing RX Data Interface.
- If the incoming M_AXI_UFC_RX_TDATA does not match with expected RX UFC data, an 8-bit error counter will be incremented.
- The error counter is indicated to the user through the UFC_ERROR_COUNT port.

Figure 10-9 shows the FRAME_CHECK UFC RX interface of the Aurora 64B/66B core, with AXI4-Stream compliant ports for UFC RX data.

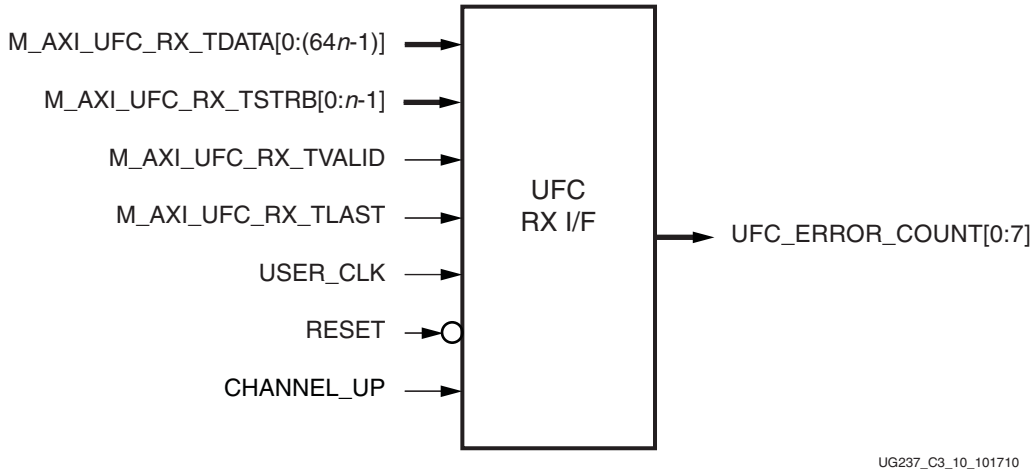


Figure 10-9: **Aurora 64B/66B Core UFC RX Interface (FRAME_CHECK)**

Table 10-9 lists the FRAME_CHECK UFC RX data ports and their descriptions.

Table 10-9: **FRAME_CHECK UFC User I/O Ports (RX)**

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

User K RX Interface

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

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

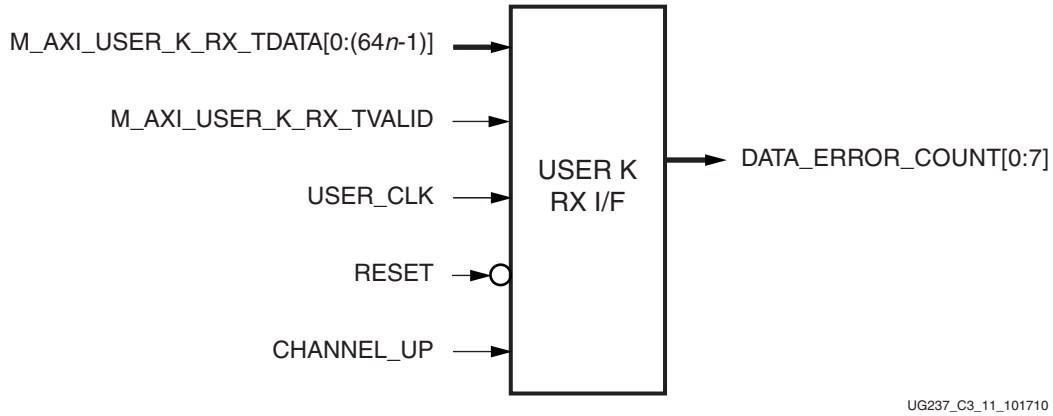


Figure 10-10: **Aurora 64B/66B Core User K RX Interface (FRAME_CHECK)**

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

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

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

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

XILINX®

Chapter 11

Project Directory Structure

Aurora 64B/66B Project Directory Structure

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

Directory and File Structure

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

Simulation test bench and simulation script files

Functional simulation files

Verilog/VHDL files for the core

Directory and File Contents

The Aurora 64B/66B core directories and their associated files are defined below.

<project directory>

The project directory contains the CORE Generator software project files.

Table 11-1: **project Directory**

Name	Description
<project directory>	
<coregen project filename>.cgp	CORE Generator software project file

Back to Top

<project directory>/<component name>

The component name directory contains the core file.

Table 11-2: **component name Directory**

Name	Description
<project directory>/<component name>	
aurora_64b66b_readme.txt	Readme file

Back to Top

<component name>/doc

The doc directory contains the product documentation.

Table 11-3: **doc Directory**

Name	Description
<component name>/doc	
ds815_aurora_64b66b.pdf	*Aurora 64B/66B v5.1 Data Sheet*
ug775_aurora_64b66b.pdf	*LogiCORE IP Aurora 64B/66B v5.1 User Guide*

Back to Top

<component name>/example_design

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

Table 11-4: **example_design Directory**

Name	Description
<component name>/example_design	
`<component name>_example_design.v[hd]` `<component name>_block.v[hd]`	Example design source file Aurora 64B/66B core top level file

Back to Top

/example_design/cc_manager

The cc_manager directory contains the clock compensation source file.

Table 11-5: **cc_manager Directory**

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

Back to Top

/example_design/clock_module

The clock_module directory contains the clock module source file.

Table 11-6: **clock_module Directory**

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

Back to Top

/example_design/gt

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

Table 11-7: **gt Directory**

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

1. For Virtex-6 FPGA GTX transceivers.
2. For Virtex-6 FPGA GTH transceivers.

Back to Top

/example_design/traffic_gen_and_check

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

Table 11-8: **traffic_gen_and_check Directory**

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

Back to Top

/example_design/ucf

The ucf directory contains the user constraints files provided with the core.

Table 11-9: **ucf Directory**

Name	Description
<component name>/example_design/ucf	
<component name>.ucf	Aurora 64B/66B core design constraints
<component name>_example_design.ucf	Aurora 64B/66B example design constraints

Back to Top

<component name>/implement

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

Table 11-10: **implement Directory**

Name	Description
<component name>/implement	
`implement.bat`	Windows batch file that processes the example design through the Xilinx tool flow
`implement.sh`	Linux shell script that processes the example design through the Xilinx tool flow
`xst.scr`	XST script file for the example design
`xst.prj`	XST project file for the example design
`Chipscope_prj.cpj`	ChipScope™ Pro tool project file

`ila.ncf`
`icon.ncf`
`vio.ncf`
	NCF files for the debug cores compatible with the ChipScope Pro Analyzer tool

`ila.ngc`
`icon.ngc`
`vio.ngc`
	NGC files for the debug cores compatible with the ChipScope Pro Analyzer tool

Back to Top

/implement/results

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

Table 11-11: **results Directory**

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

Back to Top

<component name>/simulation

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

Table 11-12: **simulation Directory**

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

Back to Top

/simulation/functional

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

Table 11-13: **functional Directory**

Name	Description
<component name>/simulation/functional	
simulate_mti.do	ModelSim macro file that compiles the example design sources, the structural simulation model, and the demonstration test bench then runs the functional simulation to completion
wave_mti.do	ModelSim macro file that opens a Wave window

Back to Top

/simulation/timing

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

Table 11-14: **timing Directory**

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

Back to Top

<component name>/src

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

Table 11-15: **src Directory**

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

Back to Top

XILINX®

Appendix A

Two Aurora 64B/66B Cores in Virtex-5 Family Sharing a RocketIO Transceiver Tile

The RocketIO™ transceivers in the Virtex®-5 family have dual architecture wherein two transceivers in each tile share a common PLL (see the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide*). Aurora 64B/66B core requires one transceiver per lane and the transceiver can be mapped to GTX0 or GTX1 in a tile. The unused transceiver is powered down during core generation. In order to use the powered down transceiver for another Aurora 64B/66B core sharing a common reference clock source, the user has to bring out the unused transceiver ports to the Aurora 64B/66B GTX wrapper file.

Aurora 64B/66B core instantiates GTX_DUAL primitive in <component name>_tile.v[hd] file which is located in <component name>/example_design/gtx/. This file contains GTX0 and GTX1 specific ports together with shared tile ports. This file is hierarchically called in <component name>_gtx_wrapper.v[hd]. User has to bring out unused transceiver ports to <component name>_gtx_wrapper.v[hd] file which is instantiated in the Aurora 64B/66B block level file. See Figure A-1, page 99 for file level hierarchy of Aurora 64B/66B core.

Shared Ports and Attributes

Aurora 64B/66B automatically sets shared PLL ports and attributes based on the CORE Generator™ software options. Reference clock input and GTX reset to a tile is driven from example design. Aurora 64B/66B core derives clock from TXOUTCLK0/1 and based on GTX0 or GTX1 selection, TXOUTCLK0 or TXOUTCLK1 is fed to the PLL input for generation of core clock. The user can use the same TXOUTCLK source used in core 1 (see Figure A-1, page 99) or optionally can use other TXOUCLK port for core 2. Since reference clock and GTX reset ports are common for GTX0 and GTX1, application of GTX reset will reset both the cores.

Steps to Modify Transceiver Specific Ports and Attributes

When user shares a tile across two Aurora 64B/66B core, shared PLL settings are not user modifiable. Follow the following steps in order to modify GTX0 or GTX1 specific ports and attributes

1. Generate an Aurora 64B/66B v3.1 core (core 1) using CORE Generator™ software. Select transceiver locations such that GTX0 or GTX1 in a tile is used for Aurora 64B/66B core. Generate another Aurora 64B/66B core (core 2) such that unused transceiver (GTX1 or GTX0) in core 1 is used for core 2.
2. Go to directory <component name>/example_design/gtx/ of core 1.
3. Open <component name>_tile.v[hd] file.
4. If line rate selection for Aurora 64B/66B core 2 is different from that of core 1, modify PLL_TXDIVSEL_OUT_1 and PLL_RXDIVSEL_OUT_1 with appropriate value in case GTX0 is selected for core 1. Modify PLL_TXDIVSEL_OUT_0 and PLL_RXDIVSEL_OUT_0 in case GTX1 is selected for core 1. See the *Virtex-5 FPGA RocketIO GTX Transceiver User Guide* for valid values of attributes PLL_TXDIVSEL_OUT_0/1 and PLL_RXDIVSEL_OUT_0/1. User does not have to modify any other attribute.

Steps to Bring Out Unused Transceiver Ports to Aurora 64B/66B GTX Wrapper File

Follow the following steps to bring out unused transceiver ports to Aurora 64B/66B <component name>_gtx_wrapper.v[hd] file:

1. Go to directory <component name>/example_design/gtx/ of core 1.
2. Open <component name>_gtx_wrapper.v[hd] file.
3. Search for GTX_TILE_INST in <component name>_gtx_wrapper.v[hd].

   For a multi-lane Aurora 64B/66B core, search for <component name>_TILE_INST_LANEn (*n*=1,2,3 ….etc). Select the instance in which one transceiver is used for Aurora 64B/66B core and the other is powered down.
4. If GTX0 is used for Aurora 64B/66B core, attributes and ports which have "1" appended are hard coded to default values. If GTX1 is used for Aurora 64B/66B core, attributes and ports that have "0" appended are hard coded to default values. Port map all such hard coded ports and attributes to new ports and attributes and declare them in module list (Verilog) or entity list (VHDL).

Steps to Bring Out Aurora 64B/66B Core Specific Logic to GTX Wrapper File

Follow the following steps to copy Aurora 64B/66B core specific logic:

1. Go to directory <component name>/example_design/gtx/ of core 2.
2. Open <component name>_gtx_wrapper.v[hd] file.
3. Copy instances GTX_WIDTH_AND_CLK_CONV_TX, SCRAMBLER_64B66B, DESCRAMBLER_64B66B, BLOCK_SYNC_SM, GTX_WIDTH_AND_CLK_CONV_RX and GTX_WIDTH_AND_CLK_CONV_RX along with associated logic to <component name>_gtx_wrapper.v[hd] of core 1. Map ports appropriately as described in step 4.

Steps to Instantiate Two Aurora 64B/66B Cores in the Top Level File

Follow the following steps to instantiate two Aurora 64B/66B cores in the top level file:

1. Go to directory <component name>/example_design of core 1.
2. Open < component name>_block.v(hd).
3. Search gtx_wrapper_i instance. Add all new ports and attributes in step 5 in the port map list. Finally map these new ports to another Aurora 64B/66B core (core 2 in Figure A-1).

Figure A-1 shows two single lane Aurora 64B/66B cores sharing a tile in Virtex-5 family.

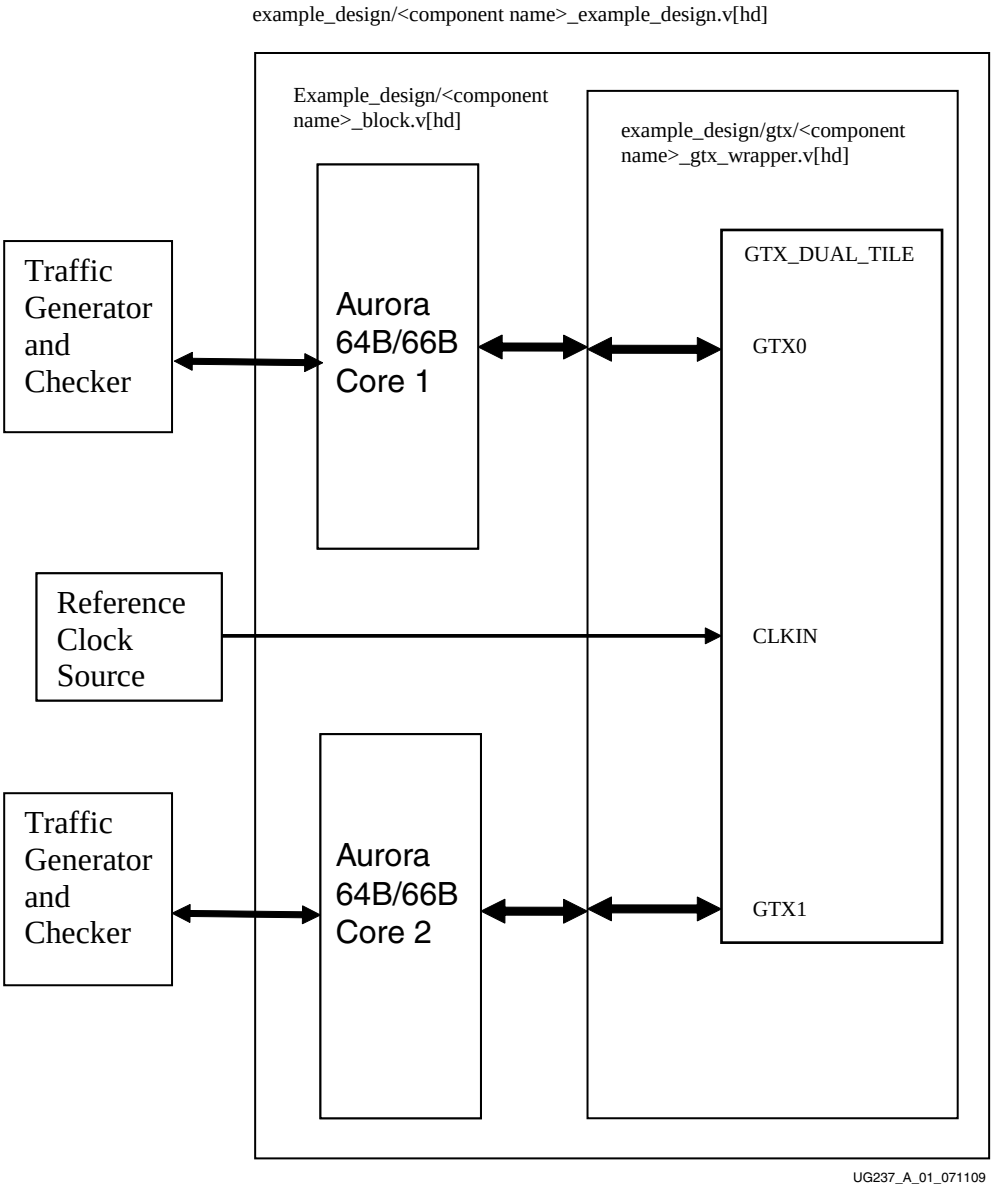


Figure A-1: **Example Showing Two Single Lane Aurora 64B/66B Cores Sharing a Transceiver Tile in Virtex-5 Family**

XILINX®

Appendix B

Generating a GTX Wrapper File from the GTX Transceiver Wizard

Follow the following steps to generate transceiver wrapper file using RocketIO™ transceiver wizard:

1. Using CORE Generator™ tool, run the Virtex-5 FPGA RocketIO and Virtex-6 FPGA GTX transceiver wizard.

   Make the component name match the Aurora 64B/66B core.
2. Select transceivers and clock source based on application requirement.
3. Select internal data width as 16 bits for Aurora 64B/66B protocol.
4. Select TX encoding as 64B/66B with Ext Seq Ctr and data path width as 32. Same settings should be selected for RX.
5. Select RXRESET, RXRECCLK, RXBUFSTATUS, RXBUFRESET, TXOUTCLK, TXRESET TXPOLARITY, TXENPRBSTST, TXBUFSTATUS and TXINHIBIT optional ports.
6. Select TXPREEMPHASIS, TXDIFFCTRL, RXEQMIX, RXCDRRESET and RXPOLARITY ports.
7. Keep all other settings as default. Generate the core.
8. Replace the <component name>_tile.v[hd] file in the example_design/gtx directory with the newly created <component name>_tile.v[hd] file.
9. Add TX and RX power down ports in replaced <component name>_tile.v[hd] file in the example_design/gtx/ directory.
10. Resynthesize and re-implement.

XILINX®

Appendix C

Aurora AXI4-Stream Migration Guide

Introduction

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

Pre-Requisites

- ISE® 12.4 build containing the Aurora 64B/66B v5.1 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 data sheet (DS815) and user guide (UG775) of the core with the AXI4-Stream updates
- Legacy data sheet (DS528), getting started guide (UG238), and user guide (UG237) for reference
- Migration guide (this Appendix)

Overview of Major Changes

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

- The user interface is modified from the legacy LocalLink (LL) to AXI4-Stream
- All AXI4-Stream signals are active-High, whereas LocalLink signals are active-Low
- The user interface in the example design and design top file will be AXI4-Stream
- A new shim module is introduced in the AXI4-Stream Aurora core to convert AXI4-Stream signals to LL and LL back to AXI4-Stream
  - The AXI4-Stream to LL shim on the transmit will convert all AXI4-Stream signals to LL
  - The shim will deal 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 will convert all LL signals to AXI4-Stream
- Each interface (PDU, UFC, and NFC) will have a separate AXI4-Stream to LL and LL to AXI4-Stream shim instantiated from the design top file

- Frame generator and checker will 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 Diagram

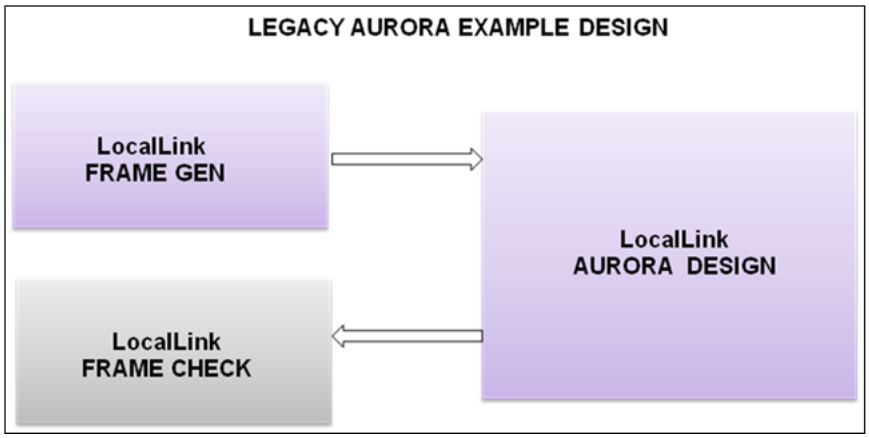


Figure C-1: **Legacy Aurora Example Design**

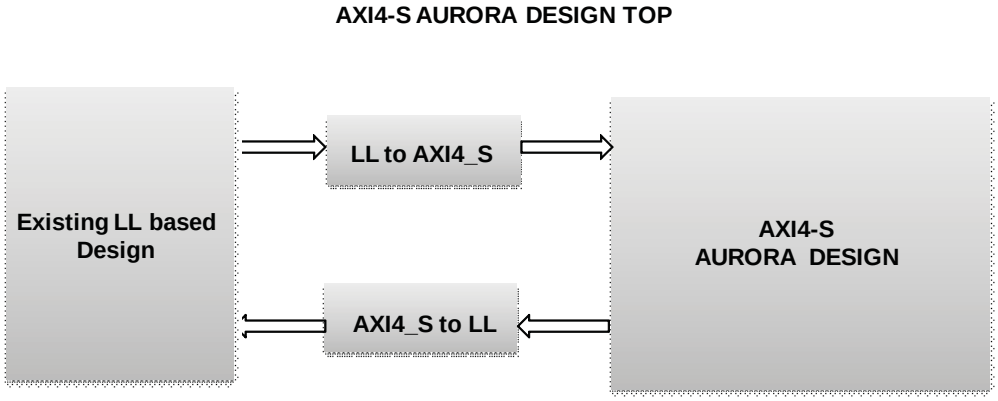


Figure C-2: **AXI4-Stream Aurora Example Design**

Signal Changes

Table C-1: **Interface Changes**

LocalLink Name	AXI4-S Name	Difference
TX_D	S_AXI_TX_TDATA	Name change only
TX_REM	S_AXI_TX_TSTRB	Name change. For functional difference, refer to Table 3-1, page 32
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, refer to Table 4-1, page 44
NFC_XOFF		
USER_K_DATA	S_AXI_USER_K_TDATA	Name change. For signal mapping, refer Table 5-2, page 52
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_TSTRB	Name change. For functional difference refer to Table 3-2, page 33
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_TSTRB	Name change For functional difference refer to Table 4-2, page 46
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_USER_K_RX_TDATA	Name change For functional difference refer to Table 5-2, page 52
RX_USER_K_BLK_NO		
RX_USER_K_SRC_RDY_N	M_AXI_USER_K_RX_TVALID	Name change; Polarity

Migration Steps

Generate an AXI4-Stream Aurora core from the CORE Generator™ tool using the ISE® tool v12.4 release.

Simulate the Core

1. Run '**vsim -do simulate_mti.do**' file from '/simulation/functional' directory.
2. Modelsim GUI will launch and compile the modules.
3. The **wave_mti.do** file will load automatically and populate AXI4-Stream signals.
4. Allow the simulation to run. This might take some time.
   a. Initially lane up is asserted.
   b. Channel up will then be asserted and the data transfer will begin.
   c. Data transfer from all flow control interfaces will now begin.
   d. Frame checker will continuously check the received data and report for any data mismatch.
5. A 'TEST PASS' or 'TEST FAIL' status is printed on the ModelSim console providing the status of the test.

Implement the Core

1. Run './implement.sh' (for Linux) from '/implement' directory.
2. The implement script will compile the core and run through ISE tool and generate a bit file and netlist for the core.

Integrate to an Existing LocalLink-based Aurora Design

1. The Aurora core provides a light-weight 'shim' to interface to any existing LL based interface. The shims are delivered along with the core from aurora_64b66b_v5_1 version of the core.
2. See Figure C-2, page 104 for the emulation of a LL Aurora core from a AXI4-Stream Aurora core.
3. Two shims <user_component_name>_ll_to_axi.v[hd] and <user_component_name>_axi_to_ll.v[hd] are provided in the 'src' directory of the AXI4-Stream Aurora core.
4. Instantiate both the shims along with <user_component_name>.v[hd] in the existing LL based design top.
5. Connect the shim and AXI4-Stream Aurora design as shown in Figure C-2, page 104.
6. The latest AXI4-Stream Aurora core can be plugged into any existing LL design environment.

GUI Changes

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

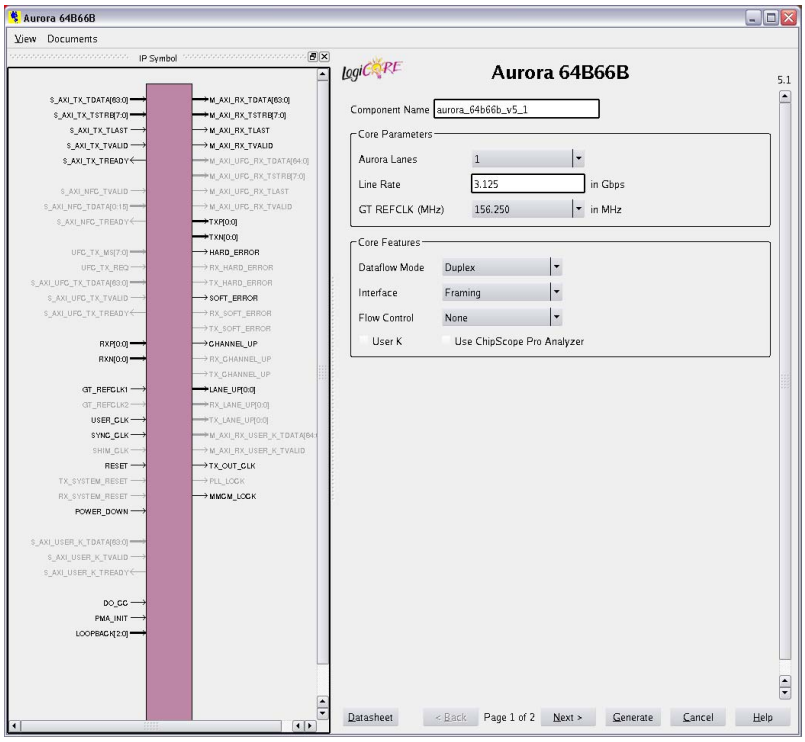


Figure C-3: **AXI4-Stream Signals**

Limitations

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

The user has to take care of these 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:

1. Byte stream
2. Continuous aligned stream
3. Continuous unaligned stream
4. 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.