LogiCORE Aurora v2.7

User Guide

UG061 (v2.7) May 17, 2007

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/27/04	2.0	Initial Xilinx release.
06/17/04	2.1	Added support for Virtex-II Pro X devices.
10/12/04	2.2 beta	Added simplex material for v2.2 beta release.
10/26/04	2.2	Added streaming interface.
04/28/04	2.3	Added support for Virtex-4 devices. First LogiCORE release. Reorganized material and moved some material into the LogiCORE Aurora v2.3 Getting Started Guide.
04/28/04	2.3.1	Corrected Table 8-4, page 92.
01/10/06	2.4	LogiCORE Aurora v2.4 release. Corrected Figure 4-1, page 33, Figure 4-6, page 39, Figure 5-1, page 49, Figure 6-1, page 57, Figure 7-1, page 71, and Figure 8-1, page 89. Added PMA-SPEED Table 7-3, page 81 and Table 7-4, page 81.
09/12/06	2.5	LogiCORE Aurora v2.5 release. Updated Chapter 3, "Customizing the Aurora Core" text and screenshots.
11/30/06	2.5.1	LogiCORE Aurora v2.5.1 release. Updated Chapter 2, "Installing and Licensing the Core."
03/01/07	2.6	LogiCORE Aurora v2.6 release.
05/17/07	2.7	LogiCORE Aurora v2.7 release. Added "Maximum Allowable Channel Skew," page 34. Added RX_SIGNAL_DETECT and RESET_CALBLOCKS to Table 6-1, page 58.

Chapter 4: User Interface

Chapter 5: Flow Control

Chapter 6: Status, Control, and the MGT Interface

Chapter 7: Clock Interface and Clocking

Chapter 8: Clock Compensation

Appendix A: Framing Interface Latency

Schedule of Figures

Chapter 6: Status, Control, and the MGT Interface

Chapter 7: Clock Interface and Clocking

Chapter 8: Clock Compensation

Appendix A: Framing Interface Latency

Schedule of Tables

Chapter 7: Clock Interface and Clocking

Chapter 8: Clock Compensation

Appendix A: Framing Interface Latency

About This Guide

This user guide describes the function and operation of the LogiCORE™ Aurora core for Vitrex™-II Pro or Virtex-4 FPGAs, as well as 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, "Installing and Licensing the Core" provides information about installing and licensing the core.
- Chapter 3, "Customizing the Aurora Core" describes how to customize an Aurora core with the available parameters.
- Chapter 4, "User Interface" provides port descriptions for the user interface.
-

Additional Resources

For additional information, go to http://www.xilinx.com/support. The following table lists some of the resources you can access from this website or by using the provided URLs.

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://www.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx solution records http://www.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes
Data Sheets	Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://www.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx design environment http://www.xilinx.com/xlnx/xil_tt_home.jsp

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-II Handbook*.
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Chapter 1

Introduction

The LogiCORE Aurora core is a high-speed serial solution based on the Aurora protocol and the Virtex-II Pro and Virtex-4 RocketIO™ multi-gigabit transceivers (MGT). The core is delivered as open-source code and supports both Verilog and VHDL design environments. Each core comes with an example design and supporting modules.

This chapter introduces the Aurora core and provides related information, including recommended design experience, additional resources, technical support, and how to submit feedback to Xilinx.

About the Core

The Aurora core is a CORE Generator™ IP core, included in the latest IP Update on the Xilinx IP Center. For detailed information about the core, see http://www.xilinx.com/aurora. For information about system requirements, installation, and licensing options, see Chapter 2, "Installing and Licensing the Core."

Recommended Design Experience

Although the Aurora core is a fully verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. For best results, previous experience building high performance, pipelined FPGA designs using Xilinx implementation software and user constraints files (UCF) is recommended. Read Chapter 6, "Status, Control, and the MGT Interface" carefully, and consult the PCB design requirements information in the *RocketIO Transceiver User Guide* for the device family that will be used. Contact your local Xilinx representative for a closer review and estimation for your specific requirements.

Related Xilinx Documents

Prior to generating an Aurora core, users should be familiar with the following:

- SP002: *Aurora Protocol Specification*
- SP006: *LocalLink Interface Specification*
- UG058: *Aurora Bus Functional Model User Guide*
- Xilinx RocketIO Transceiver User Guides:
  - UG024: *RocketIO Transceiver User Guide* (for Virtex™-II Pro transceivers)
  - UG076: *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide*
- ISE™ documentation
  http://www.xilinx.com/support/sw_manuals/xilinx9/index.htm

Additional Core Resources

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

- DS128: (*LogiCORE) Aurora v2.7 Data Sheet*
- UG061: *LogiCORE Aurora v2.7 User Guide*
- UG173: *LogiCORE Aurora v2.7 Getting Started Guide*
- Aurora v2.7 Release Notes: Answer Record25067

Technical Support

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

Xilinx will provide technical support for use of this product as described in the *LogiCORE Aurora v2.7 User Guide* and the *LogiCORE Aurora v2.7 Getting Started 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 core and the accompanying documentation.

Core

For comments or suggestions about the Aurora 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

Chapter 2

Installing and Licensing the Core

This chapter provides instructions for installing the Aurora core in the CORE Generator tool and how to obtain a free license to use the core.

System Requirements

Windows

- Windows 2000 Professional with Service Pack 2-4
- Windows XP Professional with Service Pack 2

Solaris

- Sun Solaris 2.8/5.8, 2.9/5.9

Linux

- Red Hat Enterprise WS 3.0/4.0 (32-bit or 64-bit)

Before You Begin

Before installing the Wizard, you must have a MySupport account and the ISE 9.1i SP 3 software installed on your system. If you have already completed these steps, go to "Installing the Core," page 20, 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 ISE 9.1i software and the applicable Service Pack. ISE Service Packs can be downloaded from www.xilinx.com/support/download.htm

Installing the Core

You can install the Wizard in two ways: using the CORE Generator WebUpdate facility, which lets you select from a list of updates, or by performing a manual installation after downloading the core from the web.

Automated Installation Using WebUpdate

Note: To use this installation method behind a firewall, you must know your proxy settings. If necessary, contact your administrator to determine the proxy host address and port number before you begin.

1. From the main CORE Generator window, choose **Tools** → **Software Update...**
2. Click **OK** to close the CORE Generator tool and start WebUpdate.
3. If necessary, click **Advanced...** to specify a proxy host.
4. Click **Check for Updates**.
5. Ensure **ISE 9.1i IP Update 3** is selected in the list of software updates.
6. Click **OK**.
7. WebUpdate downloads and installs the selected software updates. Restart your computer when the install is finished.
8. To confirm the installation, from the main CORE Generator window, choose **Help** → **About Xilinx CORE Generator**.
9. Look for the following lines in the About dialog box:

```
Updates installed:

ISE 9.1i IP Update 3
```

Manual Installation

1. Close the CORE Generator application if it is running.
2. Download the ZIP file from the following location and save it to a temporary directory: http://www.xilinx.com/xlnx/xil_sw_updates_home.jsp?update=ip&software=9.1i

   Note: Before you can access this page and the files listed on it, you must be registered for CORE Generator IP Updates access.
3. Extract the ZIP archive file `ise_91i_ip_update3.zip` to a temporary location.

   For Windows

   - Extract the ZIP archive file using WinZip 7.0 SR-1 or later.

     Note: When extracting the files using WinZip, you must check the **Use Folder Names** option.

   For UNIX

   - Extract the ZIP archive file using unzip.

     Note: You might need system administrator privileges to install the update.
4. In the root level of the extracted directory structure, run the setup (.exe) executable to install the update.
5. Restart the CORE Generator tool; it automatically detects and displays the newly installed IP cores.
6. Determine whether the installation was successful by verifying that the new cores are visible in the main CORE Generator window.

Obtaining Your License

To obtain your license for the Aurora core, perform the following steps:

- Navigate to the Aurora product page: http://www.xilinx.com/aurora
- Click the Aurora LogiCORE link at the bottom of the page
- Click **Order and Register**

Follow the onscreen instructions to review and electronically sign the Aurora License Agreement and download your license file for the Aurora core.

Installing Your License File

After selecting a license option, an email will be sent to you that includes instructions for installing your license file. In addition, information about advanced licensing options and technical support is provided.

Chapter 3

Customizing the Aurora Core

Introduction

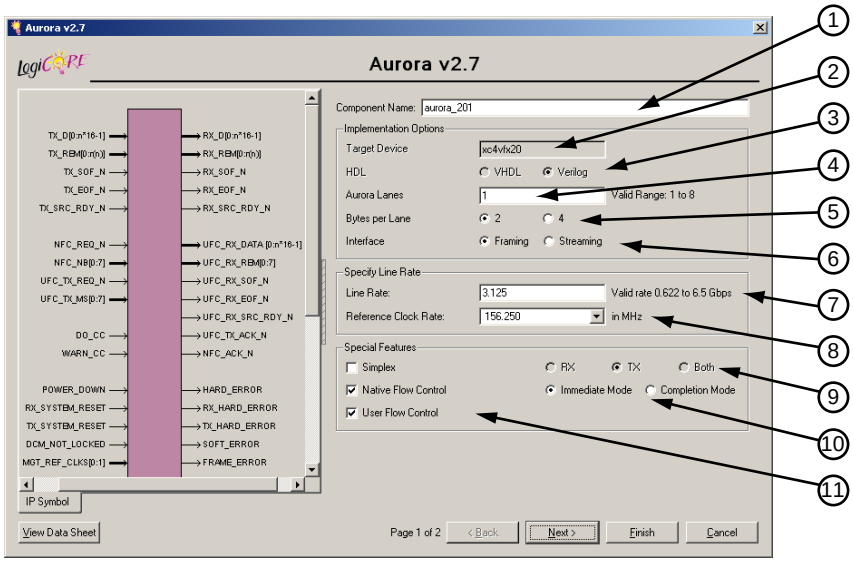
The LogiCORE Aurora core can be customized to suit a wide variety of requirements using the CORE Generator tool. This chapter details the customization parameters available to the user and how these parameters are specified within the IP Customizer interface.

Using the IP Customizer

The Aurora IP Customizer is presented when the user selects the Aurora core in the CORE Generator tool. For help starting and using the CORE Generator tool, see the *CORE Generator Guide* in the ISE documentation. Each numbered item in Figure 3-1 and Figure 3-2 corresponds to a section that describes the purpose of the feature.

Page 1 of the IP Customizer

Figure 3-1 shows Page 1 of the customizer. The left side displays a representative block diagram of the Aurora core as currently configured. The right side consists of user-configurable parameters.



Note:
The Target Device and HDL parameters are determined by the Project Settings. They are shown here for information only.

UG173_03_02_041007

Figure 3-1: **Aurora IP Customizer, Page 1**

1: Module Name

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

Default: aurora_201

2: Target Device

This area displays the device selected in Project Settings during project creation.

3: HDL

This area displays the HDL selected in Project Settings during project creation.

4: Aurora Lanes

Enter the number of lanes (MGTs) to be used in the core. The valid range depends on the target device selected.

Default: 1

5: Lane Width

Use these buttons to select the byte-width of the MGTs used in this instance of the core.

Default: 2

6: Interface

Use these buttons to select the type of data path interface used for the core. Select Framing to use a LocalLink 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. See Chapter 4, "User Interface" for more information.

Default: Framing

7: Line Rate

Enter a floating-point value in gigabits per second. The value entered must be within the valid range shown. This determines the un-encoded bit rate at which data will be transferred over the serial link. The aggregate data rate of the core is (0.8*line rate)*Aurora lanes.

Note: The Line Rate setting is disabled unless a Virtex-4 device is selected.

8: 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.

Note: Reference Clock Frequency setting is disabled unless a Virtex-4 device is selected.

9: Simplex

Check this box to create a simplex core. Simplex Aurora cores have a single, unidirectional serial port that connects to a complementary simplex Aurora core. Three buttons representing the mode of the simplex core become active when the simplex checkbox is activated: RX only, TX only, or Both. Use these buttons to select the direction of the channel the Aurora core will support. The Both selection creates two cores, one RX and one TX, which share MGTs. See Chapter 6, "Status, Control, and the MGT Interface" for more information.

Default: Not checked (full-duplex mode)

10: Native Flow Control

Check this box to add native flow control to the core. Native flow control allows full-duplex receivers to regulate the rate of the data send to them. Two buttons represent the native flow control mode, *Immediate Mode* or *Completion Mode*, for the core. These buttons are selectable only when Native Flow Control is checked. Immediate mode allows idle codes to be inserted within data frames while completion mode only inserts idle codes between complete data frames. See Chapter 5, "Flow Control" for more information.

Default: Immediate Mode

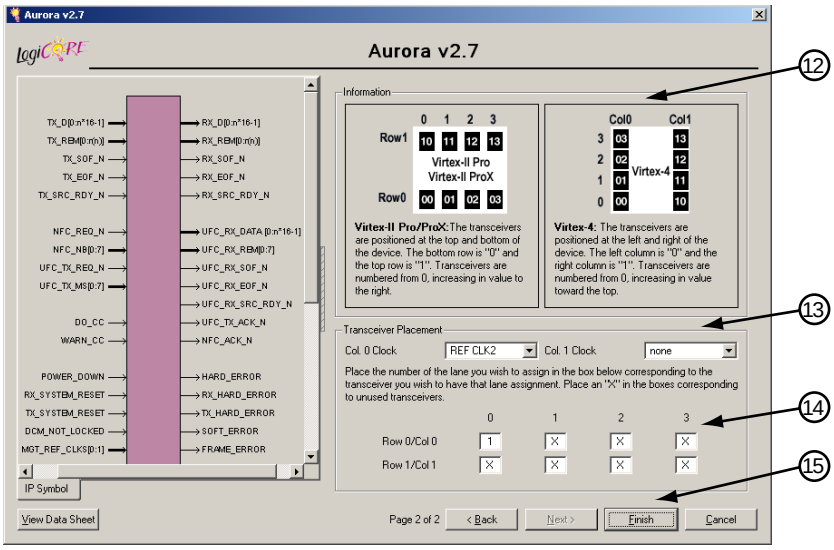
11: User Flow Control

Check this box to add user flow control to the core. User flow control allows applications to send each other brief, high-priority messages through the Aurora channel. See Chapter 5, "Flow Control" for more information.

Default: Not checked

Page 2 of the IP Customizer

Figure 3-2 shows Page 2 of the customizer.



UG061_03_02_041007

Figure 3-2: **Aurora IP Customizer, Page 2**

12: Information

This section details the layout of the MGTs for the target device. Refer to the diagram displayed in the information area when selecting the lane placement.

13: Clock Source

Select a reference clock source for each edge with assigned MGTs from the drop-down lists in this section. Virtex-II Pro devices have upper and lower MGT rows with separate reference clock sources. Virtex-4 devices have left and right MGT columns.

14: Transceiver Placement

Refer to the diagram in the information area. Each active box represents an available MGT. For each Aurora Lane in the core, starting with Lane 1, select an MGT and place the lane by typing its number in the MGT placement box. If there are fewer lanes in this instance of the core than there are available MGTs, put an X in the boxes for all the unused MGTs.

15: Finish Button

Click Finish to generate the core. The modules for the Aurora core are written to the CORE Generator project directory using the same name as the top level of the core. See "Aurora Project Directory Structure," page 27.

Aurora Project Directory Structure

The customized Aurora core is delivered as a set of HDL source modules in the selected language 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 system when the project is created, as shown below.

```
<top-level name>
        |___readme.txt
        |
        |___ug061.pdf   (user guide)
        |
        |___aurora_gs_ug173.pdf (getting started guide)
        |
        |___src
        |   |
        |   |___(top-level source file)
        |   |
        |   |___(remaining Aurora module source files)
        |
        |___ucf
        |   |
        |   |___<top-level name>.ucf  (Aurora module design constraints)
        |   |
        |   |___aurora_example.ucf     (Aurora example design constraints)
        |
        |___examples
        |   |
        |   |___(aurora_example source file)
        |   |
        |   |___(remaining source files for submodules used in the example
        |        design, not including the cc_manager or the clock module)
        |
        |___scripts
        |   |
        |   |___(make_aurora build script)
        |   |
        |   |___(config setup shell script)
        |   |
        |   |___(simulation script for modelsim)
        |
        |___clock_module (optional)
        |   |
        |   |___(clock_module source file)
        |
        |___cc_manager (optional)
        |   |
        |   |___(standard_cc_module source file)
        |
        |___testbench
            |
            |___(testbench files for simulating aurora_example design)
```

Using the Build Script

A PERL script called make_aurora.pl is delivered with the Aurora core in the scripts subdirectory. The script can be used to ease implementation of the Aurora core. Run the script to synthesize the Aurora core using XST. The script can also be run with the options shown in Table 3-1 to synthesize using Synplify, generate project files, or implement the core. Make sure the XILINX environment variable is set properly then run the script by entering the following command in the scripts directory:

```
xilperl make_aurora.pl <options>
```

Table 3-1: **Build Script Options**

Option	Description
-b	For XST synthesis, use this option to generate a bitstream file after place and route.
-files	Use this option to generate synthesis project files for the core without running the synthesis or implementation tools.
-h	Use this option to display a help message for the make script without running any tools.
-m	Use this option to run ngdbuild and map after synthesis is complete.
-npl	Use this option in place of the **-files** option to create a project file (.npl) for use with Project Navigator. Use the *update* option when opening the file in Project Navigator.
-p	Use this option to run place and route (par). The **-m** option is automatically used when **-p** is selected.
-example	Causes the `aurora_example` module to be used as the top level. Specify this option when building the example design.
-synplify	Use this option to invoke the Synplicity synthesis tool instead of XST. Specify with the **-files** option above to create a Synplicity project file without running the tool.
-win	Use this option to run the synthesis on windows platform.

Example Design Overview

Each core includes a example design that uses the core in a simple data transfer system. In the example designs, a frame generator is connected to the TX user interface, and a frame checker is connected to the RX user interface. Figure 3-3 is a block diagram of the example design for a full-duplex core. Table 3-2 describes the ports of the example design.

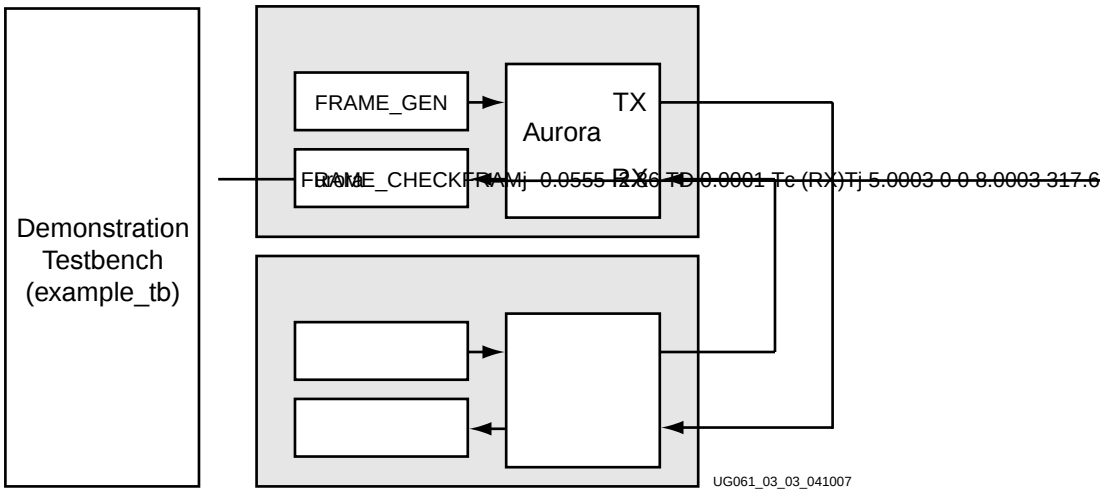


Figure 3-3: **Example Design**

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

Using the scripts provided in the scripts subdirectory, the example design can be used to quickly get an aurora design up and running on a board, or perform a quick simulation of the module. The design can also be used as a reference for the connecting the trickier interfaces on the Aurora core, such as the clocking interface.

When using the example design on a board, be sure to edit the `aurora_example.ucf` file in the ucf subdirectory to supply the correct pins and clock constraints.

Table 3-2: **Example Design I/O Ports**

Port	Direction	Description
RXN[0:*m*-1]	Input	Positive differential serial data input pin.
RNP[0:*m*-1]	Input	Negative differential serial data input pin.
TXN[0:*m*-1]	Output	Positive differential serial data output pin.
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
ERROR_COUNT[0:7]	Output	Count of the number of data words received by the frame checker that did not match the expected value.
RESET	Input	Reset signal for the example design. The reset is debounced using a USER_CLK signal generated from the reference clock input.

Table 3-2: **Example Design I/O Ports *(Continued)***

Port	Direction	Description
<reference clock(s)>	Input	The reference clocks for the Aurora 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 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 MGT 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. Full-duplex cores will have a single channel up signal; simplex cores will have one for each channel direction implemented in the core. See Chapter 6, "Status, Control, and the MGT 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 MGT they use. Simplex cores have a separate lane up signal per MGT for each channel direction supported. See Chapter 6, "Status, Control, and the MGT Interface" for details.
<simplex initialization signals>	Input/ Output	If the core is a simplex core, its sideband initialization ports will be registered and brought to the top level of the example design. See Chapter 6, "Status, Control, and the MGT Interface" for details.
PMA_INIT	Input	If the core is targeted to a Virtex-4 device, the reset signal for the PMA modules in the MGTs is connected to the top level through a debouncer. The PMA_INIT should be asserted (active-High) when the module is first powered up in hardware. The signal is debounced using the INIT_CLK
INIT_CLK	Input	INIT_CLK is used to register and debounce the PMA_INIT signal in cores targeted for the Virtex- 4 device. INIT_CLK is required since USER_CLK stops when PMA_INIT is asserted. INIT_CLK should be set to a slow rate, preferably slower than the reference clock. See Chapter 7, "Clock Interface and Clocking" for more details.

Using the Testbench and Simulation Scripts

A testbench for simulating a pair of example design modules communicating with each other is included in the testbench subdirectory. A script is included in the testbench subdirectory to compile the complete test and prepare a waveform view in ModelSim. For instructions explaining how to run the simulation, see the *LogiCORE Aurora v2.7 Getting Started Guide*.

Because cores are generated one at a time, simulating simplex cores requires additional steps (except for simplex Both cores, which can be connected to themselves). To simulate a simplex TX or simplex RX core, perform the following steps:

1. Generate the core for simulation.
2. Generate a complimentary simplex core. If using a Virtex-4 device, be sure to set the line rate to match the core for simulation.
3. Go to the scripts directory of the first core generated.
4. Set the environment variable SIMPLEX_PARTNER to point to the directory for the complementary core.
5. Run the script according to the instructions in the *LogiCORE Aurora v2.7 Getting Started Guide.*

Recommendations

The following are core and naming recommendations:

- If you are simulating a core for a Virtex-4 target, and the complementary core uses a different reference clock or different lane placements, rename the mgt_wrapper module of the complementary core, and rename its instance in the top level of the complementary core. This ensures that each module uses its own version of the MGT wrapper.
- Do not use a simplex Both core as a complementary core. This will cause module name collisions in simulation.
- The top level module name of the simplex design and simplex partner design should be similar.
  - For example, if the top level module name of the simplex design is simplex_201_tx, then the top level module name of the simplex partner should be rx_simplex_201_tx.

Chapter 4

User Interface

Introduction

An Aurora core can be generated with either a *framing* or *streaming* user data interface. In addition, flow control options are available for designs with framing interfaces. See Chapter 5, "Flow Control."

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

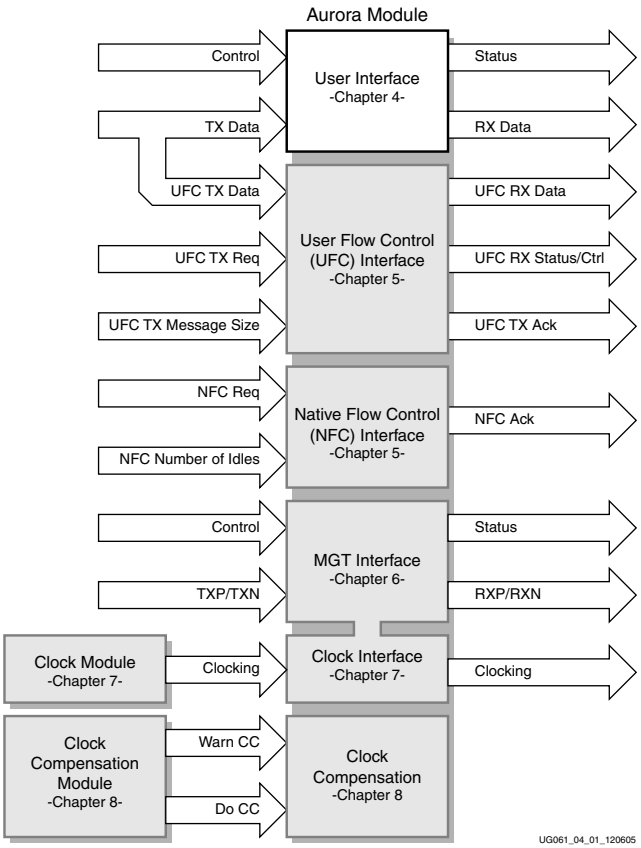


Figure 4-1: **Top-Level User Interface**

Note: The user interface signals vary depending upon the selections made when generating an Aurora core in the CORE Generator tool.

Maximum Allowable Channel Skew

The Aurora 8B/10B protocol specifies the distance between channel bonding characters to be between 16 to 32 codes. The minimum distance is 16, which is 160 bits after 8B/10B

LocalLink RX Ports

The tables in this section list port descriptions for LocalLink RX data ports. These ports are included on full-duplex, simplex RX, and simplex Both framing cores.

Table 4-3: **LocalLink User I/O Ports (RX)**

Name	Direction	Description
RX_D[0:(8*n*-1)]	Output	Incoming data from channel partner (Ascending bit order).
RX_EOF_N	Output	Signals the end of the incoming frame (active-Low, asserted for a single user clock cycle).
RX_REM[0:r(*n*)]	Output	Specifies the number of valid bytes in the last data beat; valid only when RX_EOF_N is asserted. REM bus widths are given by [0:r(*n*)], where r(*n*) = ceiling {$\log_2(n)$}-1.
RX_SOF_N	Output	Signals the start of the incoming frame (active-Low, asserted for a single user clock cycle).
RX_SRC_RDY_N	Output	Asserted (low) when data and control signals from an Aurora core are valid. Deasserted (high) when data and/or control signals from an Aurora core should be ignored (active-Low).

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

- Take data from the user on the TX_D bus
- Encapsulate and stripe the data across lanes in the Aurora channel (TX_SOF_N, TX_EOF_N)
- Pause data (that is, insert idles) (TX_SRC_RDY_N)

When the core receives data, it does the following:

- Detects and discards control bytes (idles, clock compensation, SCP, ECP)
- Asserts framing signals (RX_SOF_N, RX_EOF_N)
- Recovers data from the lanes
- Assembles data for presentation to the user on the RX_D bus

LocalLink Bit Ordering

The *LocalLink Interface Specification* allows both ascending and descending bit ordering. Aurora cores use ascending ordering. They transmit and receive the most significant bit of the most significant byte first. Figure 4-3 shows the organization of an *n*-byte example of the LocalLink data interfaces of an Aurora core.

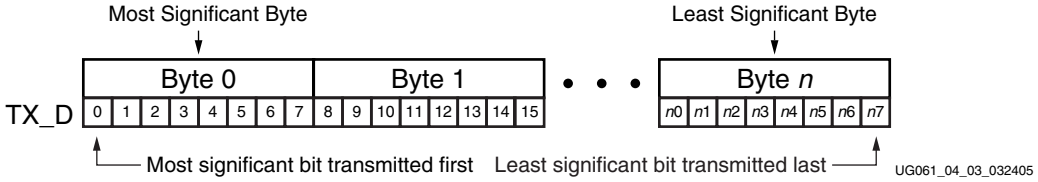


Figure 4-3: **LocalLink Interface Bit Ordering**

Transmitting Data

LocalLink is a synchronous interface. The Aurora core samples the data on the interface

Length

The user controls the channel frame length by manipulation of the **TX_SOF_N** and **TX_EOF_N** signals. The Aurora core responds with start-of-frame and end-of-frame ordered sets, /SCP/ and /ECP/ respectively, as shown in Table 4-5.

Table 4-5: **Typical Channel Frame**

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

Example A: Simple Data Transfer

Figure 4-4 shows an example of a simple data transfer on a LocalLink 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. TX_DST_RDY_N is asserted, indicating that the LocalLink interface is already ready to transmit data. When the Aurora core is not sending data, it sends idle sequences.

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

To end the data transfer, the user asserts TX_EOF_N, TX_SRC_RDY_N, the last data bytes, and the appropriate value on the TX_REM bus. In this example, TX_REM is set to *n*-1 to indicate that all bytes are valid in the last data beat. One clock cycle after TX_EOF_N is asserted, the LocalLink interface deasserts TX_DST_RDY_N and uses the gap in the data flow to send the final offset data bytes and the /ECP/ ordered set, indicating the end of the frame. TX_DST_RDY_N is reasserted on the next cycle so that more data transfers can continue. As long as there is no new data, the Aurora core sends idles.

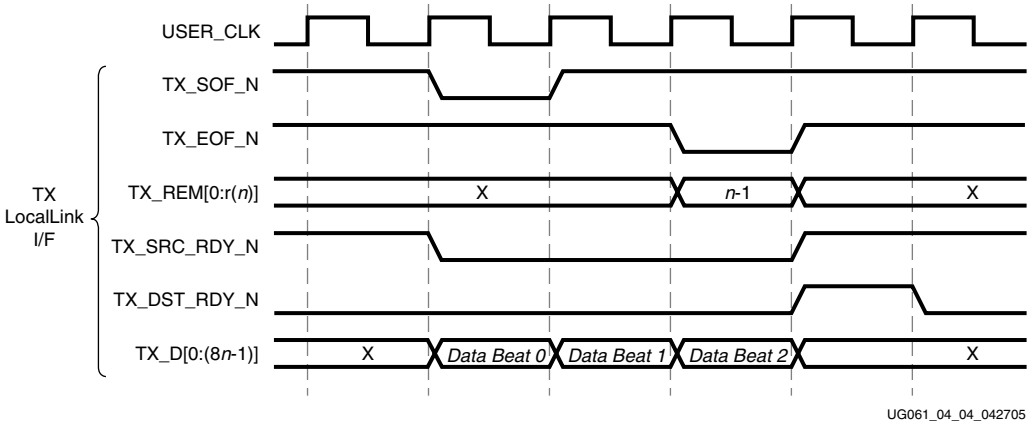


Figure 4-4: **Simple Data Transfer**

Example B: Data Transfer with Pad

Figure 4-5 shows an example of a (3*n*-1)-byte data transfer that requires the use of a pad. Since there is an odd number of data bytes, the Aurora core appends a pad character at the end of the Aurora frame, as required by the protocol. A transfer of 3*n*-1 data bytes requires two full *n*-byte data words and one partial data word. In this example, TX_REM is set to *n*-2 to indicate *n*-1 valid bytes in the last data word.

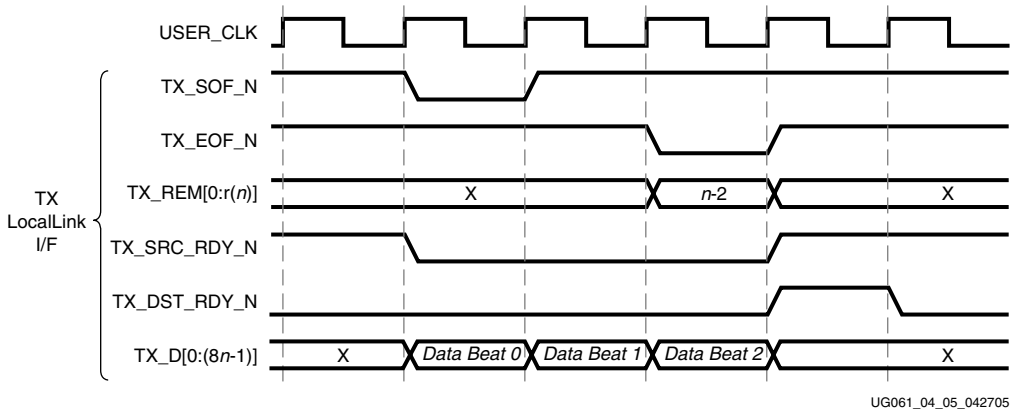


Figure 4-5: **Data Transfer with Pad**

Example C: Data Transfer with Pause

Figure 4-6 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 TX_SRC_RDY_N, causing the TX Aurora core to ignore all data on the bus and transmit idles instead. The offset data from the first data word in the previous cycle still is transmitted on lane 0, but the next data word is replaced by idle characters. The pause continues until TX_SRC_RDY_N is deasserted.

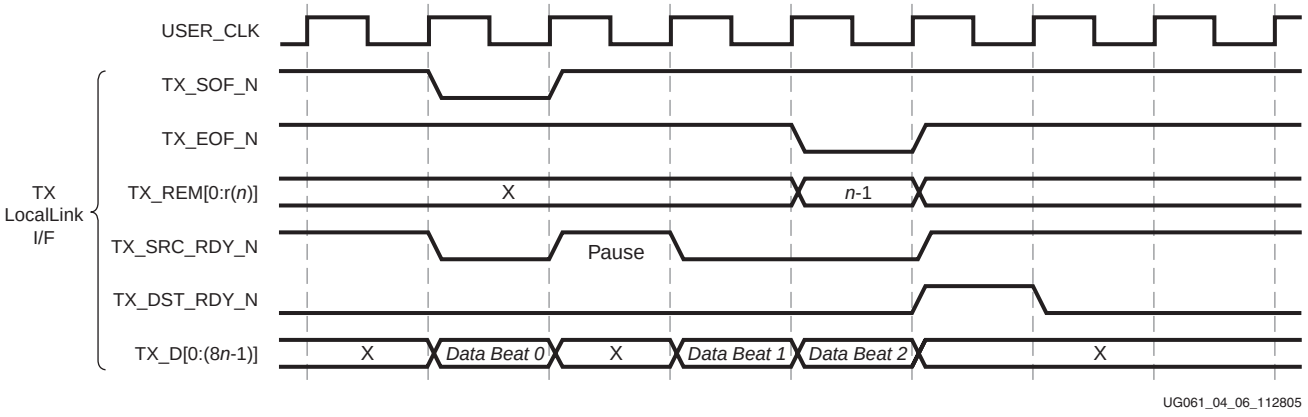


Figure 4-6: **Data Transfer with Pause**

Example D: Data Transfer with Clock Compensation

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

Figure 4-7 shows how the Aurora core pauses data transmission during the clock compensation(1) sequence.

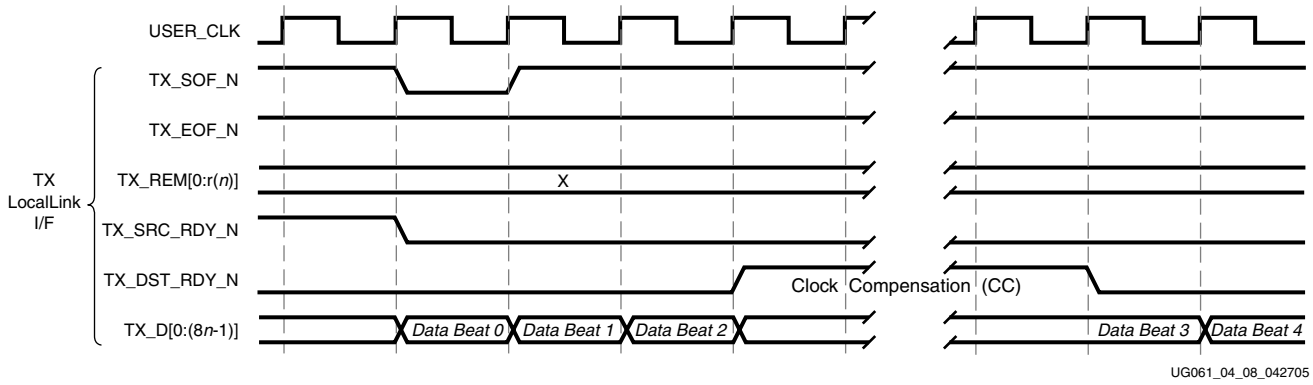


Figure 4-7: **Data Transfer Paused by Clock Compensation**

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

TX Interface Example

This section illustrates a simple example of how a user might design an interface between the user's transmit FIFO and the LocalLink interface of an Aurora core.

To review, in order to transmit data, the user asserts TX_SOF_N and TX_SRC_RDY_N. TX_DST_RDY_N indicates that the data on the TX_D bus will be transmitted on the next rising edge of the clock, assuming TX_SRC_RDY_N remains asserted.

Figure 4-8 is a diagram of a typical connection between an Aurora core and the user's data source (in this example, a FIFO), including the simple logic needed to generate TX_SOF_N, TX_SRC_RDY_N, and TX_EOF_N from typical FIFO buffer status signals. While RESET is false, the example application waits for a FIFO to fill and then it generates the TX_SOF_N and TX_SRC_RDY_N signals. These two signals cause the Aurora core to start reading the FIFO by asserting the TX_DST_RDY_N signal.

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

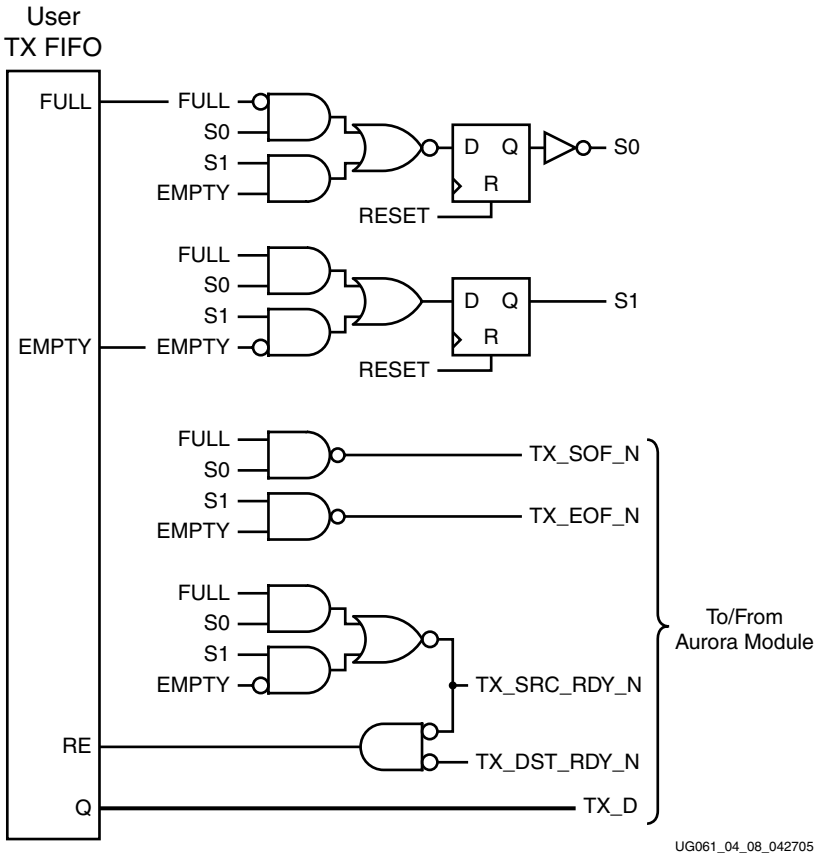


Figure 4-8: **Transmitting Data**

Receiving Data

When the Aurora core receives an Aurora frame, it presents it to the user through the RX LocalLink interface after discarding the framing characters, idles, and clock compensation sequences.

The RX_LL submodule has no built in elastic buffer for user data. As a result, there is no RX_DST_RDY_N signal on the RX LocalLink 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 core asserts the RX_SRC_RDY_N signal when the signals on its RX LocalLink interface are valid. Applications should ignore any values on the RX LocalLink ports sampled while RX_SRC_RDY_N is deasserted (high).

RX_SOF_N is asserted concurrently with the first word of each frame from the Aurora core. RX_EOF_N is asserted concurrently with the last word or partial word of each frame. The RX_REM port indicates the number of valid bytes in the final word of each frame. It uses the same encoding as TX_REM and is only valid when RX_EOF_N is asserted.

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

"Example A: Data Reception with Pause," page 43 shows the reception of a typical Aurora frame.

Example A: Data Reception with Pause

Figure 4-9 shows an example of 3*n* bytes of received data interrupted by a pause. Data is presented on the RX_D bus. When the first n bytes are placed on the bus, the RX_SOF_N and RX_SRC_RDY_N outputs are asserted to indicate that data is ready for the user. On the clock cycle following the first data beat, the core deasserts RX_SRC_RDY_N, indicating to the user that there is a pause in the data flow.

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

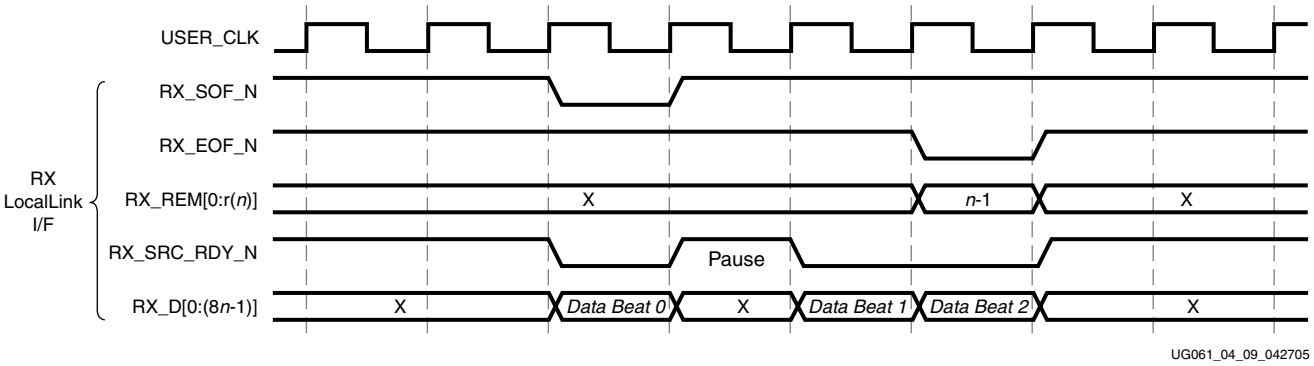


Figure 4-9: **Data Reception with Pause**

RX Interface Example

Figure 4-10 is a simple example of how a user might design an interface between the LocalLink interface of an Aurora core and a FIFO. To receive data, the user monitors the RX_SRC_RDY_N signal. When valid data is present on the RX_D port, RX_SRC_RDY_N is asserted. Because the inverse of RX_SRC_RDY_N is connected to the FIFO's WE port, the data and framing signals and REM value are written to the FIFO.

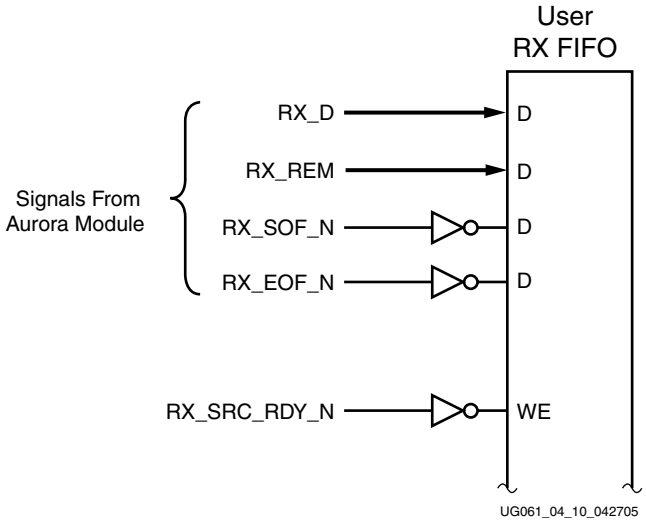


Figure 4-10: **Receiving Data**

Framing Efficiency

There are two factors that affect framing efficiency in the Aurora core:

- Size of the frame
- Width of the data path

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

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

Like many parallel interfaces, LocalLink processes data from only one frame at a time. The core must drop data when it arrives on the MGT interface at the same time as data from a previous cycle. The *LocalLink Interface Specification* includes advanced options for handling multiple frames on a single cycle, but these options are not implemented in this core.

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

Use the formula shown in Figure 4-11 to calculate the efficiency for a design of any number of lanes, any width of interface, and frames of any number of bytes. Note that this formula includes the overhead for clock compensation.

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

Where:

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

UG061_04_11_042705

Figure 4-11: **Formula for Calculating Overhead**

Example

Table 4-6 is an example calculated from the formula given in Figure 4-11. It shows the efficiency for an 8-byte, 4-lane channel and illustrates that the efficiency increases as the length of channel frames increases.

Table 4-6: **Efficiency Example**

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

Table 4-7 shows the overhead in an 8-byte, 4-lane channel when transmitting 256 bytes of frame data across the four lanes. The resulting data unit is 264 bytes long due to start and end characters, and due to the idles necessary to fill out the lanes. This amounts to 3.03 percent of overhead in the transmitter. In addition, a 12-byte clock compensation sequence occurs on each lane every 10,000 bytes, which adds a small amount more to the overhead. The receiver can handle a slightly more efficient data stream because it does not require any idle pattern.

Table 4-7: **Typical Overhead for Transmitting 256 Data Bytes**

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

Table 4-8 shows the overhead that occurs with each value of **TX_REM**.

Table 4-8: **TX_REM Value and Corresponding Bytes of Overhead**

TX_REM Bus Value	SCP	Pad	ECP	Idles	Total
0	2	1	2	6	11
1		0			10
2		1		4	9
3		0			8
4		1		2	7
5		0			6
6		1		0	5
7		0			4

Streaming Interface

Figure 4-12 shows an example of an Aurora core configured with a streaming user interface.

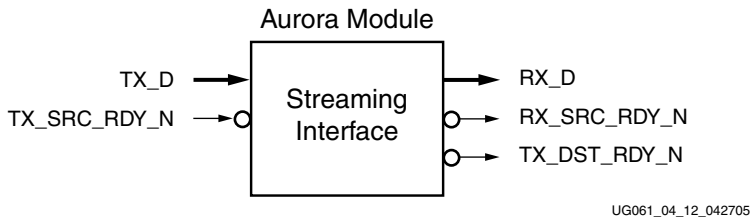


Figure 4-12: **Aurora Core Streaming User Interface**

Streaming TX Ports

Table 4-9 lists the streaming TX data ports. These ports are included on full-duplex, simplex TX, and simplex Both framing cores.

Table 4-9: **Streaming User I/O Ports (TX)**

Name	Direction	Description
TX_D[0:(8*n*-1)]	Input	Outgoing data (Ascending bit order).
TX_DST_RDY_N	Output	Asserted (low) during clock edges when signals from the source will be accepted (if TX_SRC_RDY_N is also asserted). Deasserted (high) on clock edges when signals from the source will be ignored.
TX_SRC_RDY_N	Input	Asserted (low) when LocalLink signals from the source are valid. Deasserted (high) when LocalLink control signals and/or data from the source should be ignored (active-Low).

Streaming RX Ports

Table 4-10 lists the streaming RX data ports. These ports are included on full-duplex, simplex RX, and simplex Both framing cores.

Table 4-10: **Streaming User I/O Ports (RX)**

Name	Direction	Description
RX_D[0:(8*n*-1)]	Output	Incoming data from channel partner (Ascending bit order).
RX_SRC_RDY_N	Output	Asserted (low) when data and control signals from an Aurora core are valid. Deasserted (high) when data and/or control signals from an Aurora core should be ignored (active-Low).

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 TX_D port, and use the TX_SRC_RDY_N port to indicate when the data is valid (asserted low). The Aurora core will deassert TX_DST_RDY_N (high) when the channel is not ready to receive data. Otherwise, TX_DST_RDY_N will remain asserted.

When TX_SRC_RDY_N 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 MGT 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."

When data arrives at the RX side of the Aurora channel it is presented on the RX_D bus and RX_SRC_RDY 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 4-13, page 48 shows a typical example of streaming data. The example begins with neither of the ready signals asserted, indicating that both the user logic and the Aurora core are not ready to transfer data. During the next clock cycle, the Aurora core indicates that it is ready to transfer data by asserting TX_DST_RDY_N. One cycle later, the user logic indicates that it is ready to transfer data by asserting the TX_D bus and the TX_SRC_RDY_N signal. Because both ready signals are now asserted, data D0 is transferred from the user logic to the Aurora core. Data D1 is transferred on the following clock cycle. In this example, the Aurora core deasserts its ready signal, TX_DST_RDY_N, and no data is transferred until the next clock cycle when, once again, the TX_DST_RDY_N

signal is asserted. Then the user deasserts TX_SRC_RDY_N on the next clock cycle, and no data is transferred until both ready signals are asserted.

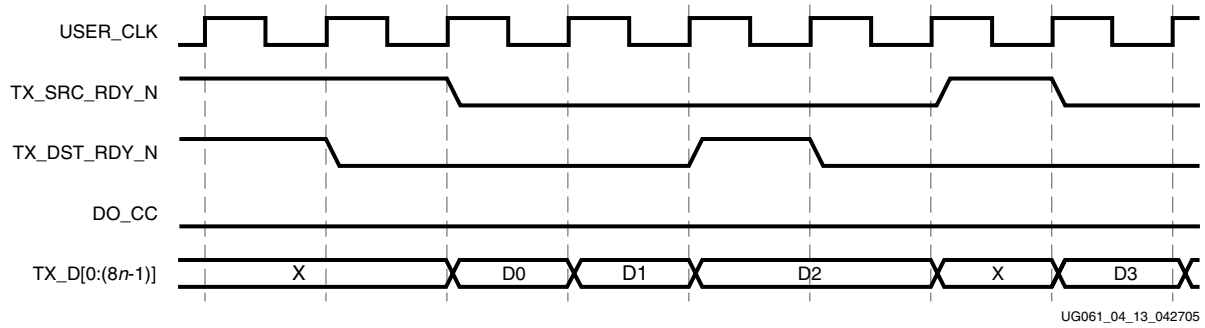


Figure 4-13: **Typical Streaming Data Transfer**

Figure 4-14 shows the receiving end of the data transfer that is shown in Figure 4-13.

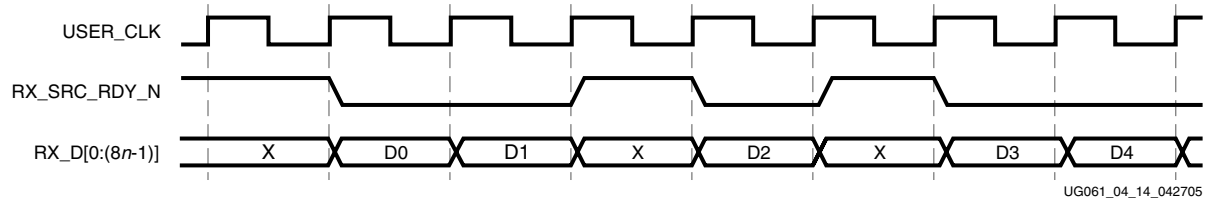


Figure 4-14: **Typical Data Reception**

Chapter 5

Flow Control

Introduction

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

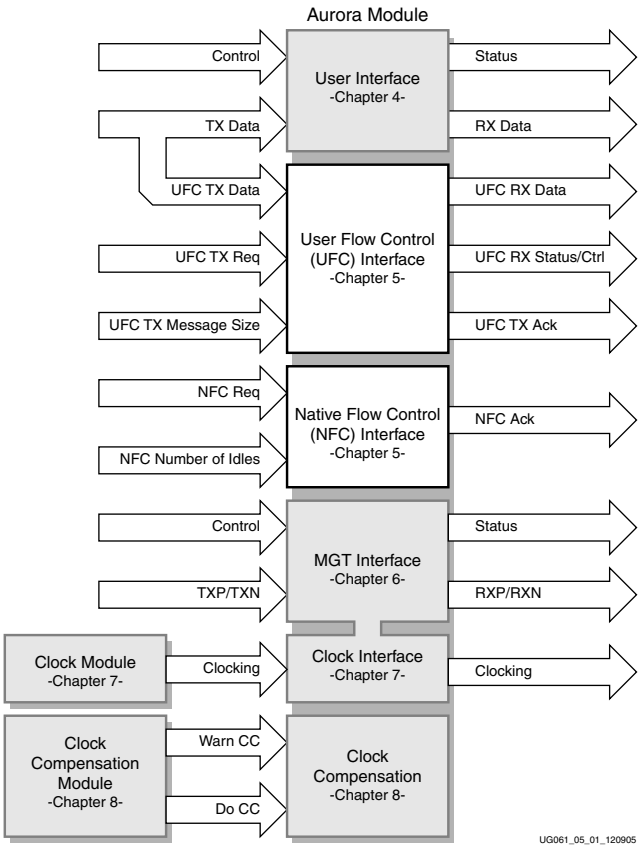


Figure 5-1: **Top-Level Flow Control**

Native Flow Control

Table 5-1 shows the codes for native flow control (NFC).

Table 5-1: **NFC Codes**

NFC_NB	Idle cycles requested
0000	0 (XON)
0001	2
0010	4
0011	8
0100	16
0101	32
0110	64
0111	128
1000	256
1001 to 1110	Reserved
1111	Infinite (XOFF)

Table 5-2 lists the ports for the NFC interface available only in full-duplex Aurora cores.

Table 5-2: **NFC I/O Ports**

Name	Direction	Description
NFC_ACK_N	Output	Asserted when an Aurora core accepts an NFC request (active-Low).
NFC_NB[0:3]	Input	Indicates the number of PAUSE idles the channel partner must send when it receives the NFC message. Must be held until NFC_ACK_N is asserted.
NFC_REQ_N	Input	Asserted to request an NFC message be sent to the channel partner (active-Low). Must be held until NFC_ACK_N is asserted.

The Aurora protocol includes native flow control (NFC) to allow receivers to control the rate at which data is sent to them by specifying a number of idle data beats that must be placed into the data stream. The data flow can even be turned off completely by requesting that the transmitter temporarily send only idles (XOFF). NFC is typically used to prevent FIFO overflow conditions. For detailed explanation of NFC operation and NFC codes, see the *Aurora Protocol Specification*.

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

Example A: Transmitting an NFC Message

Figure 5-2 shows an example of the transmit timing when the user sends an NFC message to a channel partner. Note that TX_DST_RDY_N is deasserted for one cycle(1) to create the gap in the data flow in which the NFC message is placed.

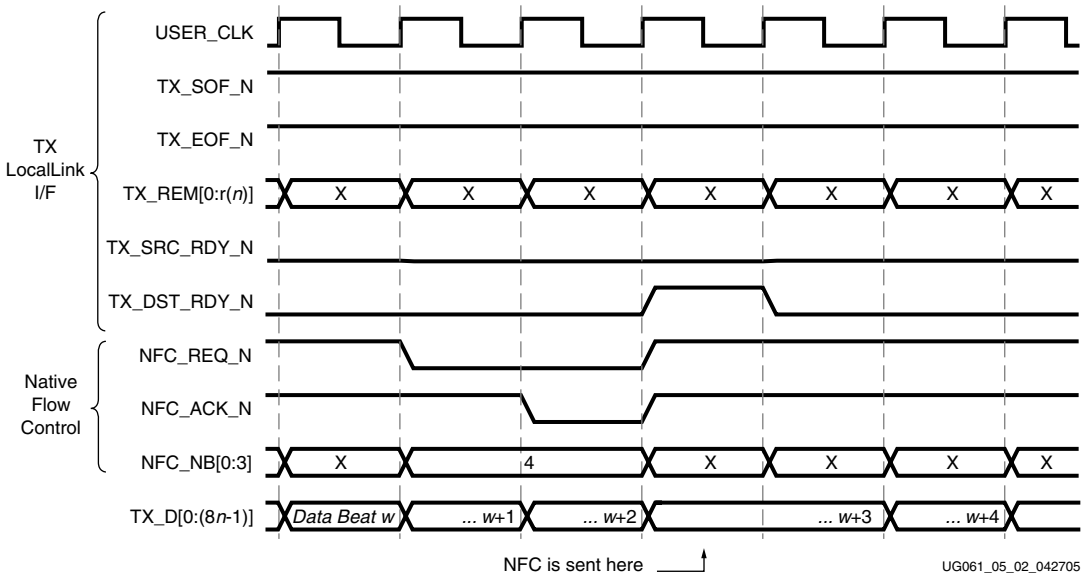


Figure 5-2: **Transmitting an NFC Message**

Example B: Receiving a Message with NFC Idles Inserted

Figure 5-3 shows an example of the signals on the TX user interface when an NFC message is received. In this case, the NFC message has a code of 0001, requesting two data beats of Idles. The core deasserts TX_DST_RDY_N on the user interface until enough Idles have been sent to satisfy the request. In this example, the core is operating in Immediate NFC mode. Aurora cores can also operate in Completion mode, where NFC Idles are only inserted between frames. If a completion mode core receives an NFC message while it is transmitting a frame, it finishes transmitting the frame before deasserting TX_DST_RDY_N to insert idles.

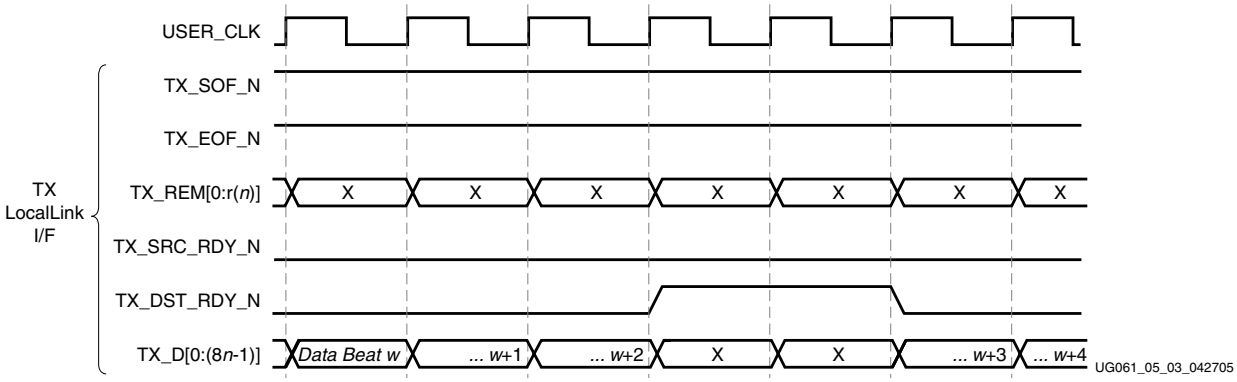


Figure 5-3: **Transmitting a Message with NFC Idles Inserted**

1. Assumes that *n* is at least 2.

User Flow Control

The Aurora 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.

Table 5-3 describes the ports for the UFC interface.

Table 5-3: **UFC I/O Ports**

Name	Direction	Description
UFC_TX_REQ_N	Input	Asserted to request a UFC message be sent to the channel partner (active-Low). Must be held until UFC_TX_ACK_N is asserted. Do not assert this signal unless the entire UFC message is ready to be sent; a UFC message cannot be interrupted once it has started.
UFC_TX_MS[0:2]	Input	Specifies the size of the UFC message that will be sent. The SIZE encoding is a value between 0 and 7. See Table 5-4, page 53.
UFC_TX_ACK_N	Output	Asserted when an Aurora core is ready to read the contents of the UFC message (active-Low). On the cycle after the ACK signal is asserted, data on the TX_D port will be treated as UFC data. TX_D data continues to be used to fill the UFC message until enough cycles have passed to send the complete message. Unused bytes from a UFC cycle are discarded.
UFC_RX_DATA[0:(8*n*-1)]	Output	Incoming UFC message data from the channel partner (n = 16 bytes max).
UFC_RX_SRC_RDY_N	Output	Asserted when the values on the UFC_RX ports are valid. When this signal is not asserted, all values on the UFC_RX ports should be ignored (active-Low).
UFC_RX_SOF_N	Output	Signals the start of the incoming UFC message (active-Low).
UFC_RX_EOF_N	Output	Signals the end of the incoming UFC message (active-Low).
UFC_RX_REM[0:r(n)]	Output	Specifies the number of valid bytes of data presented on the UFC_RX_DATA port on the last word of a UFC message. Valid only when UFC_RX_EOF_N is asserted. n = 16 bytes max. REM bus widths are given by [0:r(n)], where r(n) = ceiling $\{\log_2(n)\}$-1.

Transmitting UFC Messages

UFC messages can carry an even number of data bytes from 2 to 16. The user application specifies the length of the message by driving a SIZE code on the UFC_TX_MS port. Table 5-4 shows the legal SIZE code values for UFC.

Table 5-4: **SIZE Encoding**

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

To send a UFC message, the user application asserts UFC_TX_REQ_N while driving the UFC_TX_MS port with the desired SIZE code. UFC_TX_REQ_N must be held until the Aurora core asserts the UFC_TX_ACK_N signal, indicating that the core is ready to send the UFC message. The data for the UFC message must be placed on the TX_D port of the data interface, starting on the first cycle after UFC_TX_ACK_N is asserted. The core deasserts TX_DST_RDY_N while the TX_D port is being used for UFC data.

Figure 5-4 shows a useful circuit for switching TX_D from sending regular data to UFC data.

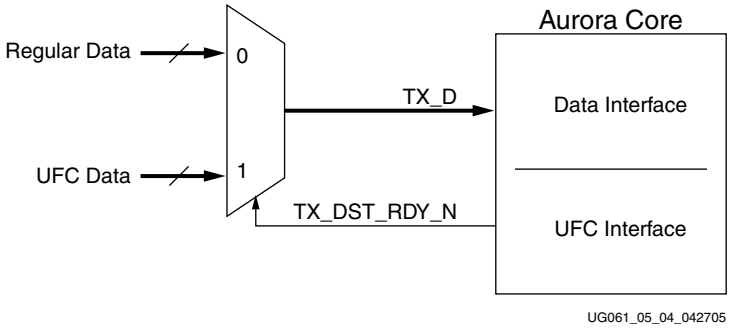


Figure 5-4: **Data Switching Circuit**

Table 5-5, page 54 shows the number of cycles required to transmit UFC messages of different sizes based on the width of the LocalLink data interface. UFC messages should never be started until all message data is available. Unlike regular data, UFC messages cannot be interrupted after UFC_TX_ACK_N has been asserted.

Table 5-5: **Number of Data Beats Required to Transmit UFC Messages**

UFC Message	UFC_TX_MS Value	LL I/F Width	Number of Data Beats	LL I/F Width	Number of Data Beats
2 Bytes	0	2 Bytes	1	10 Bytes	1
4 Bytes	1		2		
6 Bytes	2		3		
8 Bytes	3		4		
10 Bytes	4		5		
12 Bytes	5		6		2
14 Bytes	6		7		
16 Bytes	7		8		
2 Bytes	0	4 Bytes	1	12 Bytes	1
4 Bytes	1				
6 Bytes	2		2		
8 Bytes	3				
10 Bytes	4		3		
12 Bytes	5				
14 Bytes	6		4		2
16 Bytes	7				
2 Bytes	0	6 Bytes	1	14 Bytes	1
4 Bytes	1				
6 Bytes	2				
8 Bytes	3		2		
10 Bytes	4				
12 Bytes	5				
14 Bytes	6		3		
16 Bytes	7				2
2 Bytes	0	8 Bytes	1	16 Bytes or more	1
4 Bytes	1				
6 Bytes	2				
8 Bytes	3				
10 Bytes	4		2		
12 Bytes	5				
14 Bytes	6				
16 Bytes	7				

Example A: Transmitting a Single-Cycle UFC Message

The procedure for transmitting a single cycle UFC message is shown in Figure 5-5. In this case a 4-byte message is being sent on an 8-byte interface. Note that TX_DST_RDY_N is deasserted for two cycles. Aurora cores use this gap in the data flow to transmit the UFC header and message data.

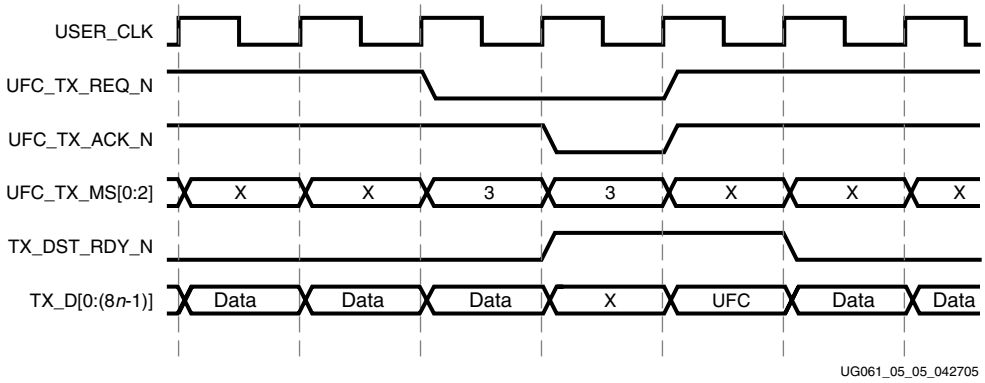


Figure 5-5: **Transmitting a Single-Cycle UFC Message**

Example B: Transmitting a Multi-Cycle UFC Message

The procedure for transmitting a two-cycle UCF message is shown in Figure 5-6. In this case the user application is sending a 16-byte message using an 8-byte interface. TX_DST_RDY_N is asserted for three cycles; one cycle for the UFC header which is sent during the UFC_TX_ACK_N cycle, and two cycles for data.

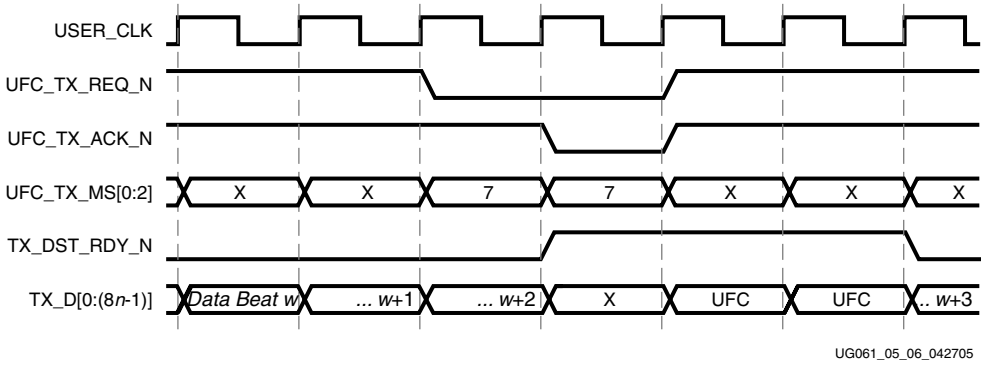


Figure 5-6: **Transmitting a Multi-Cycle UFC Message**

Receiving User Flow Control Messages

When the Aurora core receives a UFC message, it passes the data from the message to the user application through a dedicated UFC LocalLink interface. The data is presented on the UFC_RX_DATA port; UFC_RX_SOF_N indicates the start of the message data and UFC_EX_EOF_N indicates the end. UFC_RX_REM is used to show the number of valid bytes on UFC_RX_DATA during the last cycle of the message (for example, while UFC_RX_EOF_N is asserted). Signals on the UFC_RX LocalLink interface are only valid when UFC_RX_SRC_RDY_N is asserted.

Example C: Receiving a Single-Cycle UFC Message

Figure 5-7 shows an Aurora core with an 8-byte data interface receiving a 4-byte UFC message. The core presents this data to the user application by asserting UFC_RX_SRC_RDY_N, UFC_RX_SOF_N and UFC_RX_EOF_N to indicate a single cycle frame. The UFC_RX_REM bus is set to 3, indicating only the four most significant bytes of the interface are valid.

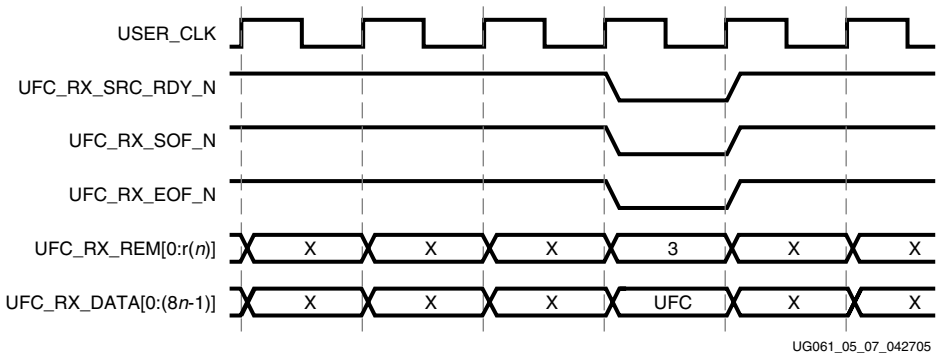


Figure 5-7: **Receiving a Single-Cycle UFC Message**

Example D: Receiving a Multi-Cycle UFC Message

Figure 5-8 shows an Aurora core with an 8-byte interface receiving a 16-byte message. Note that the resulting frame is two cycles long, with UFC_RX_REM set to 7 on the second cycle indicating that all eight bytes of the data are valid.

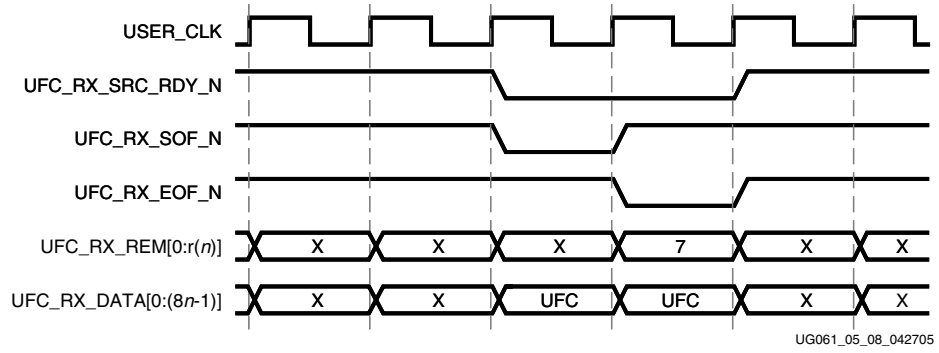


Figure 5-8: **Receiving a Multi-Cycle UFC Message**

Chapter 6

Status, Control, and the MGT Interface

Introduction

The status and control ports of the Aurora core allow user applications to monitor the Aurora channel and use built-in features of the MGT.

Aurora cores can be configured as full-duplex or simplex modules. Full-duplex modules provide high-speed TX and RX links. Simplex modules provide a link in only one direction and are initialized using sideband ports.

This chapter provides diagrams and port descriptions for the Aurora core's status and control interface, along with the MGT serial I/O interface and the sideband initialization ports that are used exclusively for simplex modules.

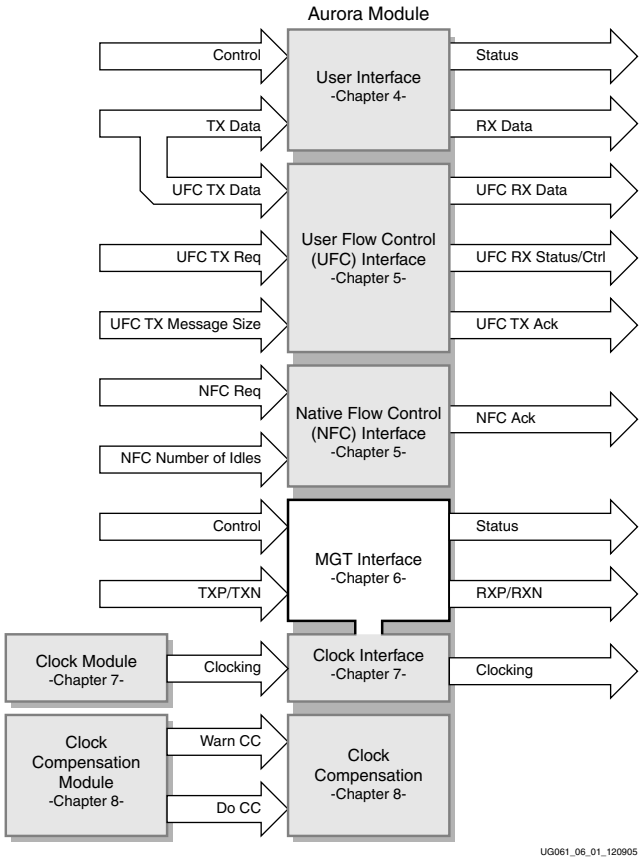


Figure 6-1: **Top-Level MGT Interface**

Full-Duplex Cores

Full-Duplex Status and Control Ports

Full-duplex cores provide a TX and an RX Aurora channel connection. Figure 6-2 shows the status and control interface for a full-duplex Aurora core. Table 6-1 describes the function of each of the ports in the interface.

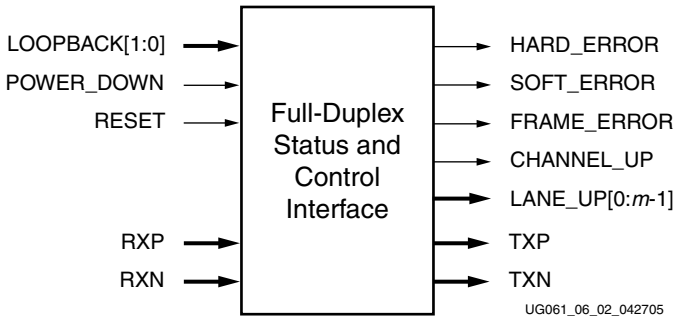


Figure 6-2: **Status and Control Interface for Full-Duplex Cores**

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

Name	Direction	Description
CHANNEL_UP	Output	Asserted when Aurora channel initialization is complete and channel is ready to send data. The Aurora core can receive data before CHANNEL_UP.
LANE_UP[0:*m*-1]	Output	Asserted for each lane upon successful lane initialization, with each bit representing one lane (active-High). The Aurora core can only receive data after all LANE_UP signals are high.
FRAME_ERROR	Output	Channel frame/protocol error detected. This port is active-High and is asserted for a single clock.
HARD_ERROR	Output	Hard error detected. (active-High, asserted until Aurora core resets). See "Error Signals in Full-Duplex Cores," page 59 for more details.
LOOPBACK[0] LOOPBACK[1]	Input	Refer to the *RocketIO User Guide* for details about loopback. See "Related Xilinx Documents" in Chapter 1.
POWER_DOWN	Input	Drives the powerdown input to the MGT (active-High).
RESET	Input	Resets the Aurora core (active-High).
SOFT_ERROR	Output	Soft error detected in the incoming serial stream. See "Error Signals in Full-Duplex Cores," page 59 for more details. (active-High, asserted for a single clock).
RXP[0:*m*-1]	Input	Positive differential serial data input pin.
RXN[0:*m*-1]	Input	Negative differential serial data input pin.
TXP[0:*m*-1]	Output	Positive differential serial data output pin.

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

Name	Direction	Description
TXN[0:*m*-1]	Output	Negative differential serial data output pin.
RX_SIGNAL_DETECT	Input	Receive signal detection. Assert this signal: • Low: MGT receiver is inactive • High: MGT receiver is active
RESET_CALBLOCKS	Input	Reset the calibration blocks (active-High).This reset should be independent of MGT operation.

Notes:

1. *n* is the number of bytes in the interface; maximum value of *n* is 16 for the UFC interface
2. Data width is given by [0:(8*n*-1)]
3. REM bus widths are given by [0:r(*n*)], where r(*n*) = ceiling {$\log_2(n)$}-1
4. *m* is the number of MGTs
5. LANE_UP width is given by [0:*m*-1]
6. The RX_SIGNAL_DETECT and RESET_CALBLOCK ports are used only in Aurora designs that use Virtex-4 devices.

Error Signals in Full-Duplex Cores

Equipment problems and channel noise can cause errors during Aurora channel operation. 8B/10B encoding allows the Aurora core to detect all single bit errors and most multi-bit errors that occur in the channel. The core reports these errors by asserting the SOFT_ERROR signal on every cycle they are detected.

The core also monitors each RocketIO 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 a burst of soft errors. The core uses the leaky bucket algorithm described in the *Aurora Protocol Specification* to detect large numbers of soft errors occurring in a short period of time, and will assert the HARD_ERROR signal when it detects them.

Whenever a hard error is detected, the Aurora core automatically resets itself and attempts to reinitialize. In most cases, this will allow 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 Aurora leaky bucket algorithm.

Aurora cores with a LocalLink data interface can also detect errors in Aurora frames. Errors of this type include frames with no data, consecutive Start of Frame symbols, and consecutive End of Frame symbols. When the core detects a frame problem, it asserts the FRAME_ERROR signal. This signal is usually asserted close to a SOFT_ERROR assertion, with soft errors being the main cause of frame errors.

Table 6-2 summarizes the error conditions the Aurora core can detect and the error signals used to alert the user application.

Table 6-2: **Error Signals in Full-Duplex Cores**

Signal	Description
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 elastic buffer for RX data overflows or underflows. This can occur when the clock source frequencies for the two channel partners are not within 200 ppm. **Bad Control Character:** The protocol engine attempts to send a bad control character. This is an indication of design corruption or catastrophic failure. **Soft Errors:** There are too many soft errors within a short period of time. The Aurora protocol defines a leaky bucket algorithm for determining the acceptable number of soft errors within a given time period. When this number is exceeded, the physical connection may be too poor for communication using the current voltage swing and pre-emphasis settings.
SOFT_ERROR	**Invalid Code:** The 10-bit code received from the channel partner was not a valid code in the 8B/10B table. This usually means a bit was corrupted in transit, causing a good code to become unrecognizable. Typically, this will also result in a frame error or corruption of the current channel frame. **Disparity Error:** The 10-bit code received from the channel partner did not have the correct disparity. This error is also usually caused by corruption of a good code in transit, and can result in a frame error or bad data if it occurs while a frame is being sent. **No Data in Frame:** A channel frame is received with no data.
FRAME_ERROR	**Truncated Frame:** A channel frame is started without ending the previous channel frame, or a channel frame is ended without being started. **Invalid Control Character:** The protocol engine receives a control character that it does not recognize. **Invalid UFC Message Length:** A UFC message is received with an invalid length.

Full-Duplex Initialization

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

Aurora cores can receive data before CHANNEL_UP is asserted. Only the RX_SRC_RDY_N 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 Cores

Simplex TX Status and Control Ports

Simplex TX cores allow user applications to transmit data to a simplex RX core. They have no RX connection. Figure 6-3 shows the status and control interface for a simplex TX core. Table 6-3 describes the function of each of the ports in the interface.

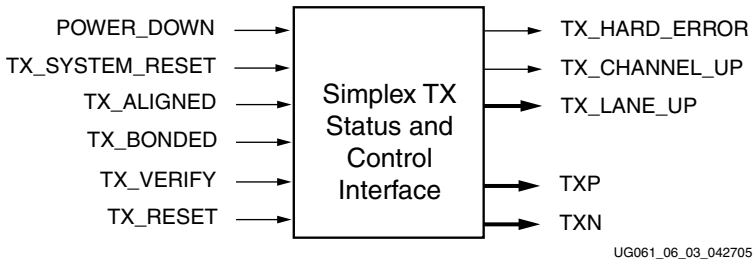


Figure 6-3: **Status and Control Interface for Simplex TX Core**

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

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

Table 6-3: **Status and Control Ports for Simplex TX Cores *(Continued)***

Name	Direction	Description
TX_CHANNEL_UP	Output	Asserted when Aurora channel initialization is complete and channel is ready to send data. The Aurora core can receive data before TX_CHANNEL_UP.
TX_LANE_UP[0:*m*-1]	Output	Asserted for each lane upon successful lane initialization, with each bit representing one lane (active-High).
TX_HARD_ERROR	Output	Hard error detected. (active-High, asserted until Aurora core resets). See "Error Signals in Simplex Cores," page 66 for more details.
POWER_DOWN	Input	Drives the powerdown input to the MGT (active-High).
TX_SYSTEM_RESET	Input	Resets the Aurora core (active-High).
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
TXN[0:*m*-1]	Output	Negative differential serial data output pin.

Notes:

1. *n* is the number of bytes in the interface; maximum value of *n* is 16 for the UFC interface
2. Data width is given by [0:(8*n*-1)]
3. REM bus widths are given by [0:r(*n*)], where r(*n*) = ceiling {$\log_2(n)$}-1
4. *m* is the number of MGTs
5. TX_LANE_UP width is given by [0:*m*-1]

Simplex RX Status and Control Ports

Simplex RX cores allow user applications to receive data from a simplex TX core. Figure 6-4 shows the status and control interface for a simplex RX core. Table 6-4, page 63 describes the function of each of the ports in the interface.

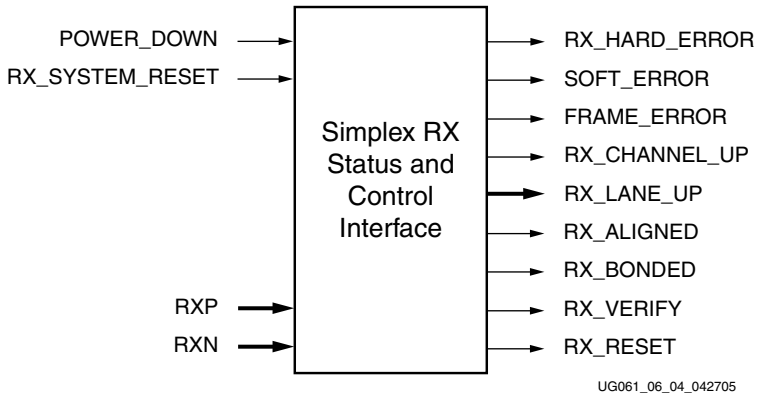


Figure 6-4: **Status and Control Interface for Simplex RX Core**

Table 6-4: **Status and Control Ports for Simplex RX Cores**

Name	Direction	Description
RX_ALIGNED	Output	Asserted when RX module has completed lane initialization. Typically connected to TX_ALIGNED.
RX_BONDED	Output	Asserted when RX module has completed channel bonding. Not used for single-lane channels. Typically connected to TX_BONDED.
RX_VERIFY	Output	Asserted when RX module has completed verification. Typically connected to TX_VERIFY.
RX_RESET	Output	Asserted when the RX module needs the TX module to restart initialization. Typically connected to TX_RESET.
RX_CHANNEL_UP	Output	Asserted when Aurora channel initialization is complete and channel is ready to send data. The Aurora core can receive data before RX_CHANNEL_UP.
RX_LANE_UP[0:*m*-1]	Output	Asserted for each lane upon successful lane initialization, with each bit representing one lane (active-High). The Aurora core can only receive data after all RX_LANE_UP signals are high.
FRAME_ERROR	Output	Channel frame/protocol error detected. This port is active-High and is asserted for a single clock.
RX_HARD_ERROR	Output	Hard error detected. (active-High, asserted until Aurora core resets). See "Error Signals in Simplex Cores," page 66 for more details.
POWER_DOWN	Input	Drives the powerdown input to the MGT (active-High).
RX_SYSTEM_RESET	Input	Resets the Aurora core (active-High).
SOFT_ERROR	Output	Soft error detected in the incoming serial stream. See "Error Signals in Simplex Cores," page 66 for more details. (Active-High, asserted for a single clock).
RXP[0:*m*-1]	Input	Positive differential serial data input pin.
RXN[0:*m*-1]	Input	Negative differential serial data input pin.

Notes:

1. *n* is the number of bytes in the interface; maximum value of *n* is 16 for the UFC interface
2. Data width is given by [0:(8*n*-1)]
3. REM bus widths are given by [0:r(*n*)], where r(*n*) = ceiling $\{\log_2(n)\}$-1
4. *m* is the number of MGTs
5. RX_LANE_UP width is given by [0:*m*-1]

Simplex Both Status and Control Ports

Simplex Both cores consist of a simplex TX core and a simplex RX core sharing the same set of RocketIO transceivers. Like a full-duplex core, the simplex Both core transmits and receives data. Two key differences are as follows:

- The TX and RX sides of the simplex Both core initialize and run independently of each other, unlike the duplex core, where both TX and RX must be operational for either direction to work.
- Simplex Both cores only connect to other simplex cores, while full-duplex cores only connect to other full-duplex cores. The TX side of a simplex Both core connects to a simplex RX core, or to the RX side of a simple Both. Likewise, the RX side of a simplex Both core connects to a simplex TX core, or to the TX side of a simplex Both.

Figure 6-5 shows the status and control interface for a simplex Both core. Table 6-5, page 65 describes the function of each of the ports in the interface.

Figure 6-5: **Status and Control Interface for Simplex Both Cores**

Table 6-5: **Status and Control Ports for Simplex Both Cores *(Continued)***

Name	Direction	Description
RX_SYSTEM_RESET TX_SYSTEM_RESET	Input	Resets the Aurora core (active-High).
SOFT_ERROR	Output	Soft error detected in the incoming serial stream. See "Error Signals in Simplex Cores," page 66 for more details. (Active-High, asserted for a single clock).
RXP[0:*m*-1]	Input	Positive differential serial data input pin.
RXN[0:*m*-1]	Input	Negative differential serial data input pin.
TXP[0:*m*-1]	Output	Positive differential serial data output pin.
TXN[0:*m*-1]	Output	Negative differential serial data output pin.

Notes:

1. *n* is the number of bytes in the interface; maximum value of *n* is 16 for the UFC interface
2. Data width is given by [0:(8*n*-1)]
3. REM bus widths are given by [0:r(*n*)], where r(*n*) = ceiling {$\log_2(n)$}-1
4. *m* is the number of MGTs
5. LANE_UP width is given by [0:*m*-1]

Error Signals in Simplex Cores

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

All simplex cores monitor their MGTs for hardware errors such as buffer overflow and loss of lock. Hardware errors on the TX side of the channel are reported by asserting the TX_HARD_ERROR signal; RX side hard errors are reported using the RX_HARD_ERROR signal. Simplex RX and simplex Both cores use the Aurora protocol's leaky bucket algorithm to evaluate bursts of soft errors. If too many soft errors occur in a short span of time, RX_HARD_ERROR is asserted.

Whenever a hard error is detected, the Aurora core automatically resets itself and attempts to reinitialize. In simplex Both cores, TX hard errors reset only the TX side, and RX errors reset only the RX side. Resetting allows the Aurora channel to be re-established as soon as the hardware issue that caused the hard error is resolved in most cases. Soft errors do not lead to a reset unless enough of them occur in a short period of time to trigger the Aurora leaky bucket algorithm.

Simplex RX and simplex Both cores with a LocalLink data interface can also detect errors in Aurora frames when they are received. Errors of this type include frames with no data, consecutive Start of Frame symbols, and consecutive End of Frame symbols. When the core detects a frame problem, it asserts the FRAME_ERROR signal. This signal will usually be asserted close to a SOFT_ERROR assertion, as soft errors are the main cause of frame errors. Simplex TX modules do not use the FRAME_ERROR port.

Table 6-6, page 67 summarizes the error conditions simplex Aurora cores can detect and the error signals uses to alert the user application.

Table 6-6: **Error Signals in Simplex Cores**

Signal	Description	TX	RX	Both
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.	x		x
	RX Overflow/Underflow: The elastic buffer for RX data overflows or underflows. This can occur when the clock source frequencies for the two channel partners are not within 200 ppm.		x	x
	Bad Control Character: The protocol engine attempts to send a bad control character. This is an indication of design corruption or catastrophic failure.	x		x
	Soft Errors: There are too many soft errors within a short period of time. The Aurora protocol defines a leaky bucket algorithm for determining the acceptable number of soft errors within a given time period. When this number is exceeded, the physical connection may be too poor for communication using the current voltage swing and pre-emphasis settings.		x	x
SOFT_ERROR	**Invalid Code:** The 10-bit code received from the channel partner was not a valid code in the 8B/10B table. This usually means a bit was corrupted in transit, causing a good code to become unrecognizable. Typically, this will also result in a frame error or corruption of the current channel frame.		x	x
	Disparity Error: The 10-bit code received from the channel partner did not have the correct disparity. This error is also usually caused by corruption of a good code in transit, and can result in a frame error or bad data if it occurs while a frame is being sent.		x	x
	No Data in Frame: A channel frame is received with no data.		x	x
FRAME_ERROR	**Truncated Frame:** A channel frame is started without ending the previous channel frame, or a channel frame is ended without being started.	x		x
	Invalid Control Character: The protocol engine receives a control character that it does not recognize.		x	x
	Invalid UFC Message Length: A UFC message is received with an invalid length.		x	x

Simplex Initialization

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

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

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

Both initialization methods are described in the "Using a Back Channel" and "Using Timers" sections below.

Using a Back Channel

If there is a communication channel available from the RX side of the connection to the TX side, using a back channel is the safest way to initialize and maintain a simplex channel. There are very few requirements on the back channel; it need only deliver messages to the TX side to indicate which of the sideband initialization signals is asserted when the signals change. Examples of suitable back channels include:

The aurora_example design included in the examples directory with simplex Aurora cores shows a simple side channel that uses 3-4 I/O pins on the device.

Using Timers

For some systems a back channel is not possible. In these cases, serial channels can be initialized by driving the TX simplex initialization with a set of timers. The timers must be designed carefully to meet the needs of the system since the average time for initialization depends on many channel specific conditions such as clock rate, channel latency, skew between lanes, and noise.

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

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

Reset and Power Down

Reset

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

On full-duplex modules, the RESET signal resets both the TX and RX sides of the channel when asserted on the positive edge of USER_CLK. On simplex modules, the resets for the TX and RX channels are separate. TX_SYSTEM_RESET resets TX channels; RX_SYSTEM_RESET resets RX channels. The TX_SYSTEM_RESET is separate from the TX_RESET and RX_RESET signals used on the simplex sideband interface.

Power Down

When POWER_DOWN is asserted, the RocketIO transceivers in the Aurora core are turned off, putting them into a non-operating low-power mode. When POWER_DOWN is deasserted, the core automatically resets. Be careful when asserting this signal on cores that use TX_OUT_CLK (see the Chapter 6, "Status, Control, and the MGT Interface"). TX_OUT_CLK will stop when the RocketIO transceivers are powered down. See the *RocketIO User Guide* for the device you are using for details about powering down RocketIO transceivers.

Some transceivers have the ability to power down their TX and RX circuits separately. This feature is not currently supported in Aurora. The core can be modified to add this feature, but support for these modifications is not covered by the standard *Aurora LogiCORE License*.

Timing

Figure 6-6 shows the timing for the RESET and POWER_DOWN signals. In a quiet environment, t_{cu} is generally less than 800 clocks; In a noisy environment, t_{cu} can be much longer.

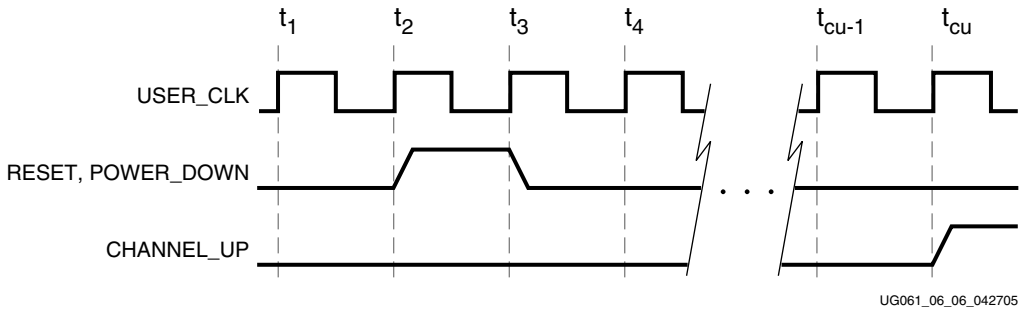


Figure 6-6: **Reset and Power Down Timing**

/

Virtex-II Pro Devices

Clock Interface Ports for Virtex-II Pro Cores

Table 7-1 describes the Virtex-II Pro Aurora core clock ports.

Table 7-1: **Clock Ports for a Virtex-II Pro Aurora Core**

Name	Direction	Description
DCM_NOT_LOCKED	Input	If a DCM is used to generate clocks for the Aurora core, the DCM_NOT_LOCKED signal should be connected to the inverse of the DCM's LOCKED signal. The clock modules provided with the Aurora core use the DCM for clock division. The DCM_NOT_LOCKED signal from the clock module should be connected to the DCM_NOT_LOCKED signal on the Aurora core. If no DCM is used to generate clock signals for the Aurora core, tie DCM_NOT_LOCKED to ground.
USER_CLK_N_2X	Input	Parallel clock required for Virtex-II Pro cores with 4-byte lanes. This clock is used to drive the internal synchronization logic of the RocketIO transceiver. The clock must be aligned to the negative edge of USER_CLK and twice the frequency. This port does not appear for Virtex-II Pro cores with 2-byte lanes.
USER_CLK	Input	Parallel clock shared by the Aurora core and the user application. On Virtex-II Pro cores with 2-byte lanes, the rate is the same as the reference clock. On Virtex-II Pro cores with 4-byte lanes, the rate is half the reference clock rate.
TOP_<*ref_clk(1 of 4)*>[1]	Input	This port is used when RocketIO transceivers in the Aurora core are placed on the top edge of a Virtex-II Pro device. The port can be TOP_BREFCLK, TOP_BREFCLK2, TOP_REFCLK, or TOP_REFCLK2. The rate of the clock depends on the desired data rate for the module. See the "Setting the Clock Rate in Virtex-II Pro Designs," page 75. [2]
BOTTOM_<*ref_clk(1 of 4)*>[1]	Input	This port is used when RocketIO transceivers in the Aurora core are placed on the bottom edge of a Virtex-II Pro device. The port can be BOTTOM_BREFCLK, BOTTOM_BREFCLK2, BOTTOM_REFCLK, or BOTTOM_REFCLK2. The rate of the clock depends on the desired data rate for the module. See the "Setting the Clock Rate in Virtex-II Pro Designs," page 75.[2]

Notes:

1. The variable, <*ref_clk(1 of 4)*>, refers to any one of four clocks: REFCLK, REFCLK2, BREFCLK, or BREFCLK2.
2. Refer to the *RocketIO Transceiver User Guide* for more on MGT clocking.

Parallel Clocks in Virtex-II Pro Designs

Connecting USER_CLK for Virtex-II Pro 2-Byte Designs

Virtex-II Pro 2-byte lane Aurora cores use a single clock to synchronize all signals between the core and the user application called USER_CLK. All logic that connects to the core must be driven by USER_CLK, which in turn must be the output of a global clock buffer (BUFG). USER_CLK must be frequency locked to the reference clock; typically the reference clock is used to drive the input of the USER_CLK BUFG. Figure 7-2, Figure 7-3, page 76, and Figure 7-6, page 79 show some typical 2-byte lane core configurations. In all cases, the reference clock drives a BUFG, which provides both the user application and the Aurora core with a USER_CLK signal. The DCM_NOT_LOCKED signal should typically be tied low for 2-byte Virtex-II Pro Aurora cores.

Connecting USER_CLK/USER_CLK_N_2X for Virtex-II Pro 4-Byte Designs

Virtex-II Pro 4-byte lane Aurora cores use two phase-locked parallel clocks. The first is USER_CLK, which synchronizes all signals between the core and the user application. All logic that connects to 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 half the rate of the reference clock.

The second phase-locked parallel clock is USER_CLK_N_2X. The rising edge of this clock must coincide with the falling edge of USER_CLK. This clock must also come from a BUFG and is frequency locked to the reference clock. It is connected directly to the Aurora core to drive the internal synchronization logic of the RocketIO MGT.

To make it easier to use the two parallel clocks, a clock module is provided with each 4-byte lane Virtex-II Pro Aurora core, in a subdirectory called clock_module. Figure 7-4, page 77, Figure 7-5, page 78, and Figure 7-7, page 80 show some typical 4-byte lane core configurations. In all cases, the clock module is used and its reference clock input is driven by the reference clock for the Aurora core. If the DCM is used, the DCM_NOT_LOCKED signal should be connected to the DCM_NOT_LOCKED output of the clock module. If the clock module is not used, connect the DCM_NOT_LOCKED signal to the inverse of the DCM_LOCKED signal from any DCM used to generate either of the parallel clocks.

Reference Clocks in Virtex-II Pro Designs

Virtex-II Pro Aurora cores require low jitter reference clocks for generating and recovering high-speed serial clocks in the RocketIO transceivers. The Aurora core clock interface includes separate reference clock inputs for each edge of the device (TOP and BOTTOM). Each reference clock can be set to one of the four possible Virtex-II Pro reference clock input ports, called REFCLK, REFCLK2, BREFCLK, and BREFCLK2. 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, since they introduce too much jitter.

Using REFCLK or REFCLK2

REFCLK and REFCLK2 are flexible reference clock inputs for low rate serial connections. They are flexible because they can be driven from any global clock resource in the FPGA except DCMs. REFCLK and REFCLK2 can also be routed anywhere on the chip. This means that unlike BREFCLK and BREFCLK2, the reference clocks for Aurora cores with RocketIO transceivers placed on both edges can be driven from the same FPGA resource (for example, the same BUFG or IBUFG).

REFCLK and REFCLK2 are only suitable for low rates. Regular FPGA clock resources introduce too much jitter for fast serial clock rates. REFCLK and REFCLK2 should not be used for line rates higher than 2.5 Gb/s (125 MHz reference clock).

If REFCLK or REFCLK2 is selected as the reference clock input for RocketIO transceivers on one or both edges of the device, driving the clock is easy. Simply connect the output of any of the normal global clock resources, such as a BUFG, and IBUFG or a BUFG multiplexer to the REFCLK or REFCLK2 input on the Aurora core. Make sure the clock is frequency locked to the USER_CLK (the USER_CLK could be used to drive the reference clock).

Using BREFCLK or BREFCLK2

BREFCLK and BREFCLK2 are dedicated, low-jitter differential clock networks for the RocketIO transceivers on each Virtex-II Pro device. They can support line rates up to 3.125 Gb/s (156.25 MHz reference clock rate) but have some placement limitations that make them less flexible than REFCLKs.

The most important placement limitation of the BREFCLKs is that the BREFCLK and BREFCLK2 networks are separate on each edge of the device. There is a BREFCLK differential input pair for the top edge of the device, and a separate BREFCLK differential input pair for the bottom. The same goes for BREFCLK2.

Connecting BREFCLK or BREFCLK2 for a single edge involves the following steps:

- Select a BREFCLKs for the edge based on the location of the pins on the package
- Instantiate an IBUFGDS_LVDS_25 module (a differential IBUFG set for 2.5V LVDS)
- Create an N and a P input signal for the buffer (for example, top_brefclk_n and top_brefclk_p)
- Place the input signals at the input pins for the selected reference clock using LOC constraints in the UCF
- Create an output signal for the buffer (such as TOP_BREFCLK) and connect it to the reference clock input for the desired edge
- In general, the output signal from the buffer will also be used to drive the BUFG that produces USER_CLK

Figure 7-2, page 75 and Figure 7-3, page 76 show examples of single edge reference clock placement for a 2-byte Virtex-II Pro Aurora core. Figure 7-4, page 77 and Figure 7-5, page 78 show examples of single edge reference clock placement for a 4-byte Virtex-II Pro Aurora core.

The same steps for connecting a BREFCLK for a single edge must be performed separately for each edge of a multi-lane design. Since both edges of a multi-lane design must be frequency locked, all the reference clock inputs for the core must be driven by the same physical clock. Figure 7-6, page 79 shows this configuration for a 2-byte Virtex-II Pro core, and Figure 7-7, page 80 shows it for a 4-byte Virtex-II Pro core.

Setting the Clock Rate in Virtex-II Pro Designs

Virtex-II Pro RocketIO transceivers have two multipliers for generating serial clocks from the reference clock, 10x and 20x. Aurora cores are built to use 20x rate: the line rate will always be 20x the reference clock rate. The minimum reference clock rate is 50 MHz and the maximum is 156.25 MHz. As a result, the minimum line rate for an Aurora core is 1 Gb/s. If a lower rate is required, the 10x mode of the RocketIO transceiver can be used with a reference clock rate as low as 62.2 MHz. To switch the multiplier, the following procedure must be followed:

- In the UCF, change the SERDES_10B attribute to true
- If the lane width is 2-bytes per lane:
  - Add a top-level port called USER_CLK_N_2X to the top level
  - Replace the USER_CLK connection for TXUSRCLK and RXUSRCLK with USER_CLK_N_2X
  - Generate a 4-byte core and copy the clock_module directory and its contents to the directory for the 2-byte core
  - Connect the clocks for the modified core as if it were a 4-byte core

Clock Distribution Examples for Virtex-II Pro Designs

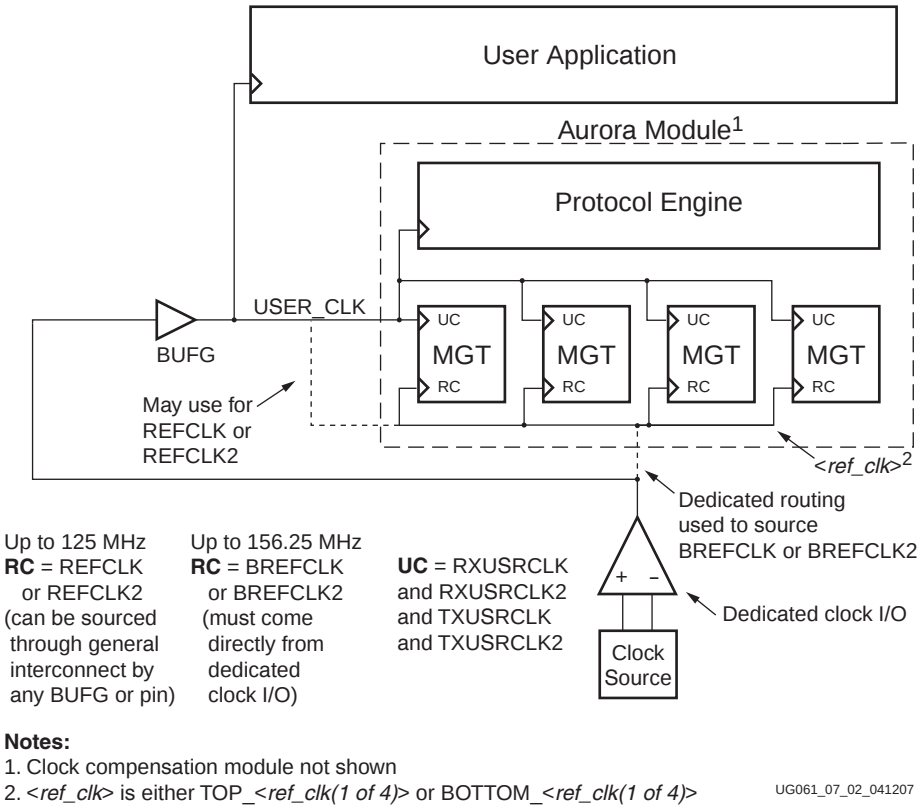


Figure 7-2: **Typical Clocking for Small 2-Byte Designs**

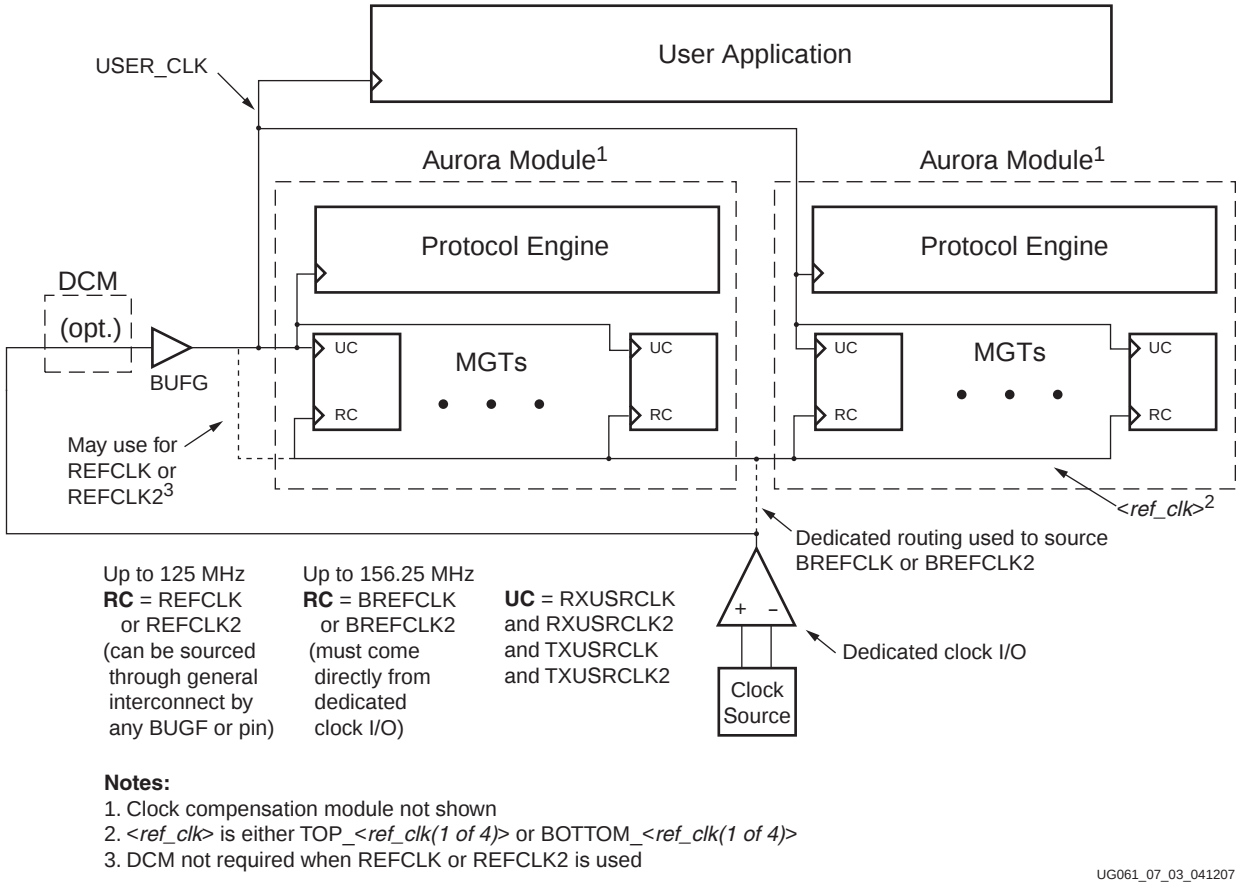


Figure 7-3: **Reference Clock Distribution for 2-Byte Designs on One Edge**

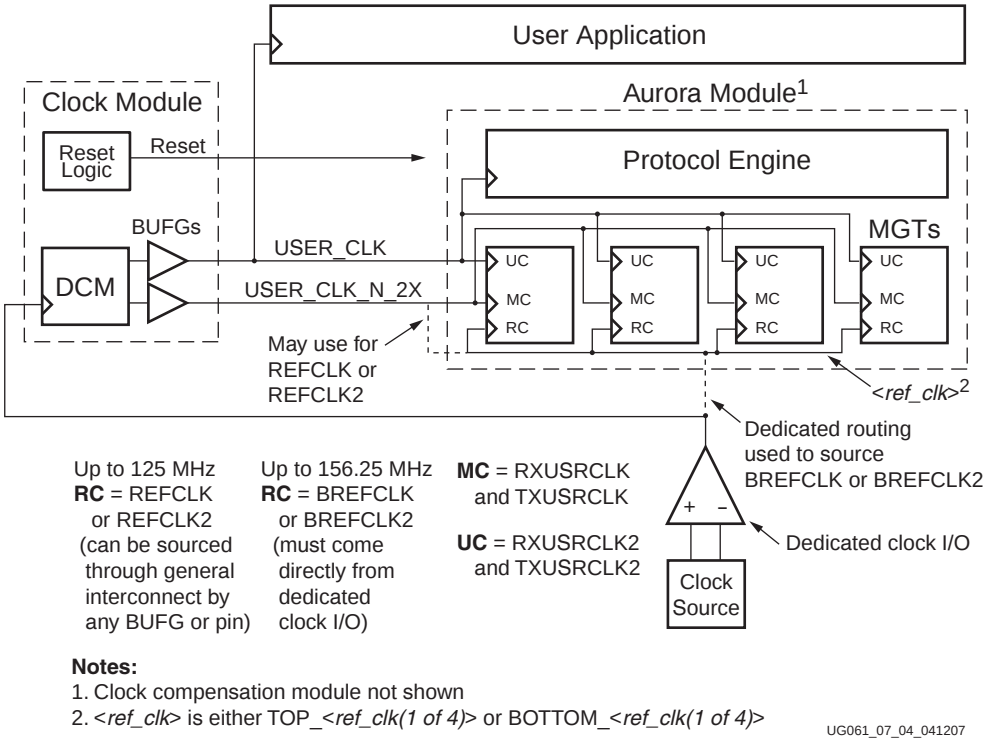


Figure 7-4: **Typical Clocking for Small 4-Byte Designs**

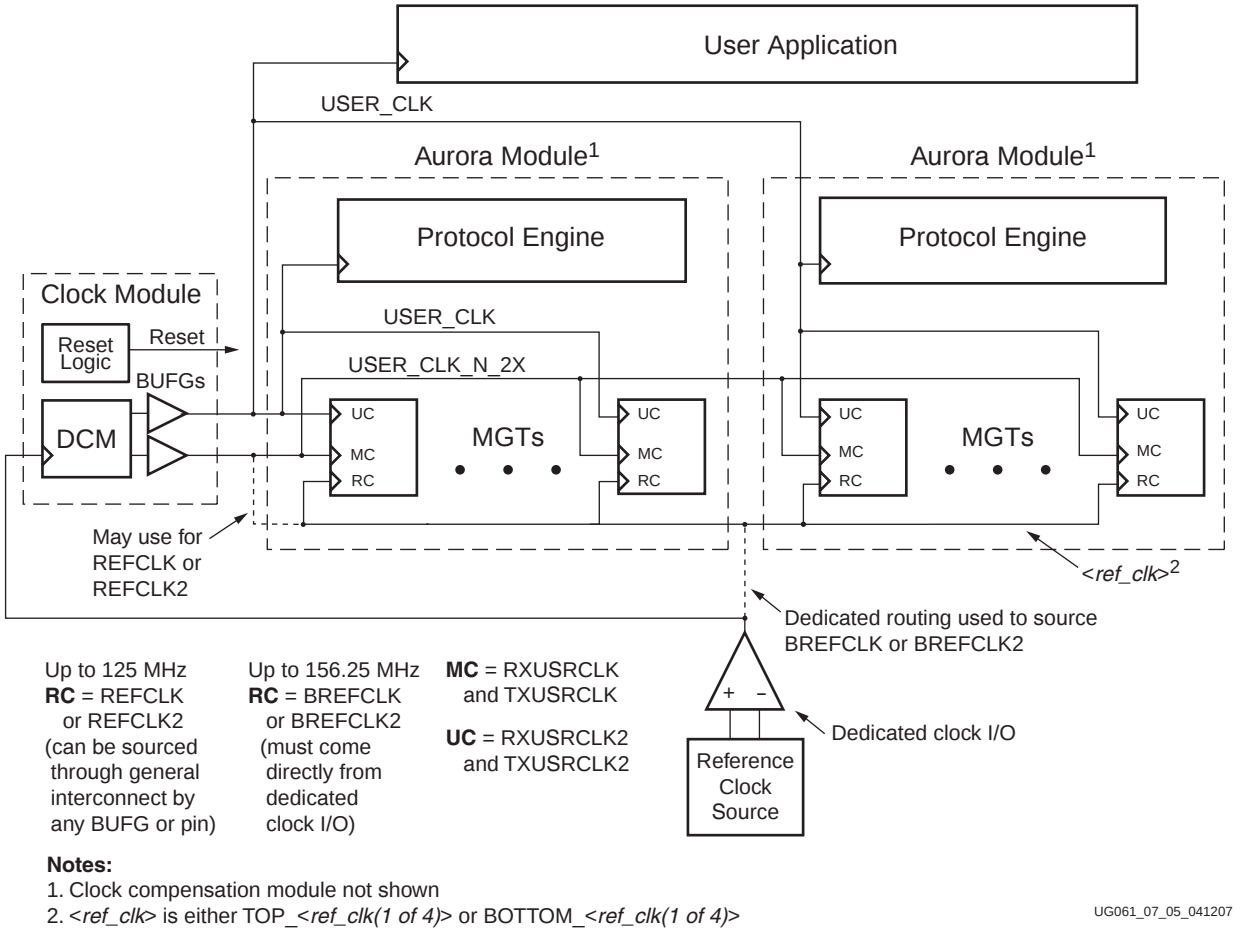


Figure 7-5: **Reference Clock Distribution for 4-Byte Designs on One Edge**

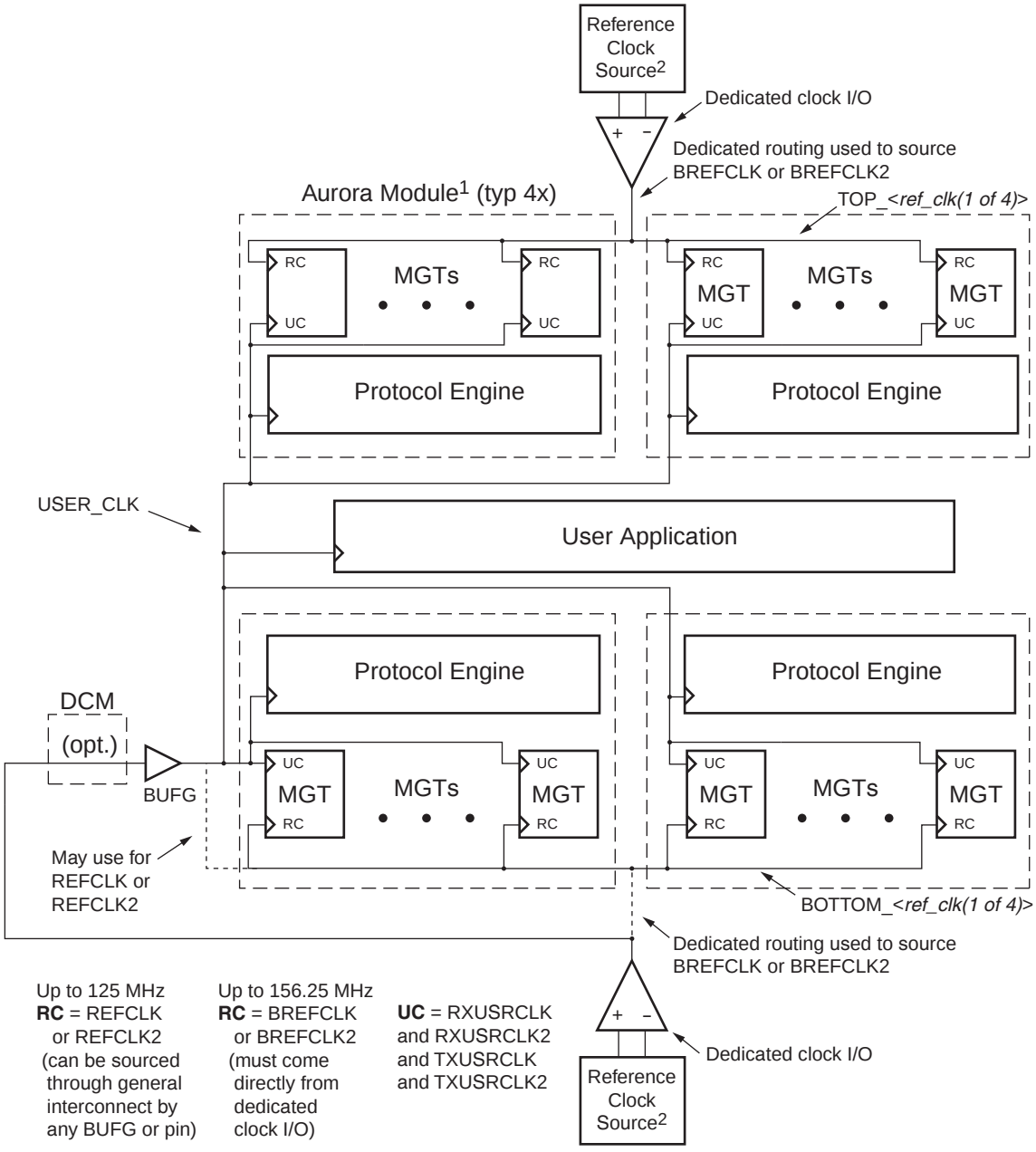


Figure 7-6: **Reference Clock Distribution for 2-Byte Designs on Opposite Edges**

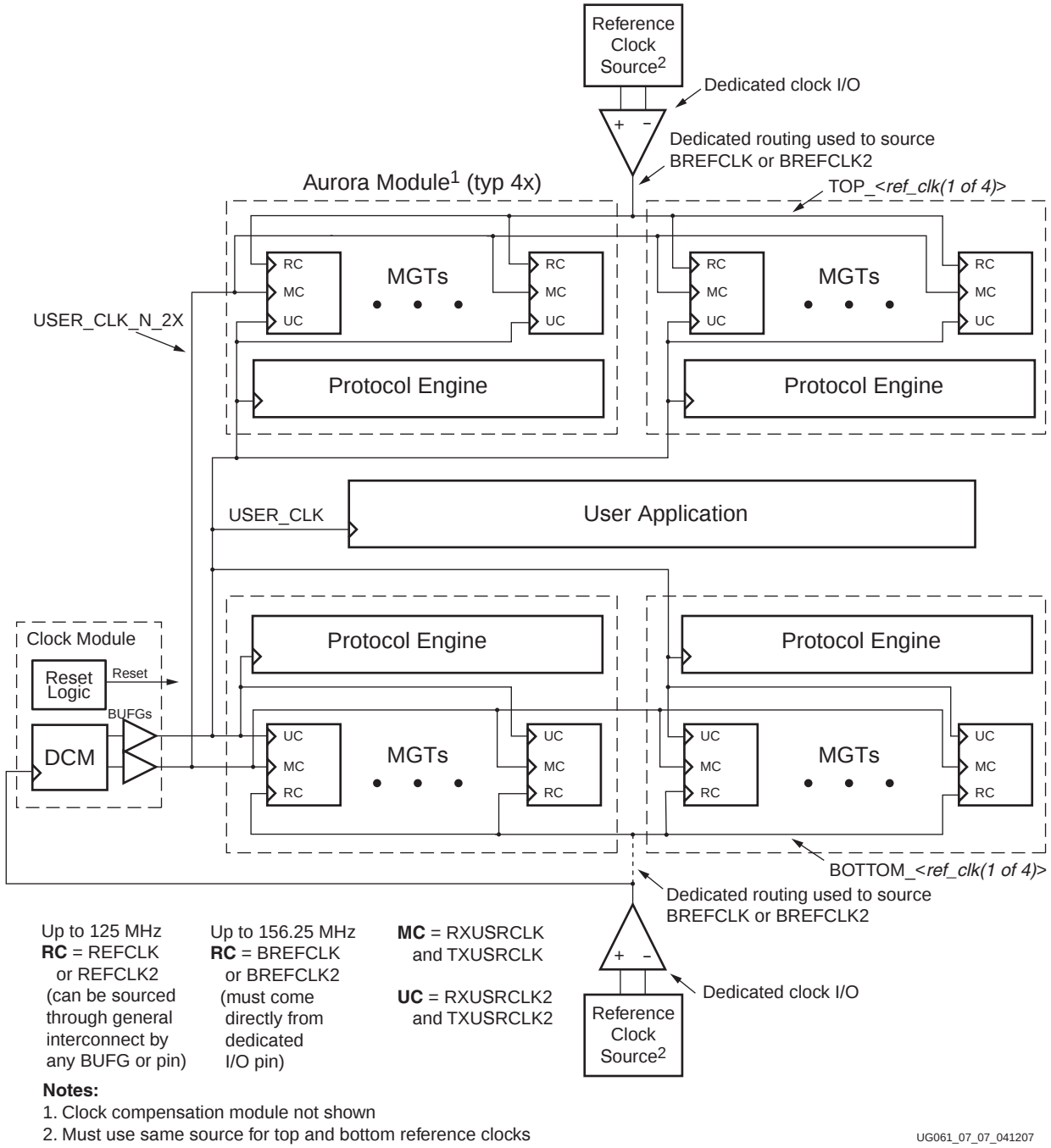


Figure 7-7: **Reference Clock Distribution for 4-Byte Designs on Opposite Edges**

Virtex-II Pro X Devices

Note: Virtex-II Pro X devices are not recommended for new designs.

Virtex-4 Devices

Clock Interface Ports for Virtex-4 Cores

Table 7-2 describes the Virtex-4 Aurora core clock ports. The *<ref_clk(1 of 3)>*_LEFT or the *<ref_clk(1 of 3)>*_RIGHT inputs are based on whether the MGTs are placed on the left or right side of the device, and which of the three (REFCLK1, REFCLK2, or GREFCLK) reference clocks are chosen.

Table 7-2: **Clock Ports for a Virtex-4 Aurora Core**

Name	Direction	Description
DCM_NOT_LOCKED	Input	If a DCM is used to generate clocks for the Aurora core, the DCM_NOT_LOCKED signal should be connected to the inverse of the DCM's LOCKED signal. The clock modules provided with the Aurora core use the DCM for clock division. The DCM_NOT_LOCKED signal from the clock module should be connected to the DCM_NOT_LOCKED signal on the Aurora core. If no DCM is used to generate clock signals for the Aurora core, tie DCM_NOT_LOCKED to ground.
USER_CLK	Input	Parallel clock shared by the Aurora core and the user application. On Virtex-4 cores the USER_CLK is the output of a BUFG whose input is derived from TX_OUT_CLK. The rate is the same as 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-4 MGTs. The GT11 MGT generates TX_OUT_CLK from its reference clock based on its PMA speed setting. This clock, when buffered, should be used as the user clock for logic connected to the Aurora core.
SYNC_CLK[1]	Input	Parallel clock used by the internal synchronization logic of the RocketIO transceivers in the Aurora core. SYNC_CLK is half the rate of USER_CLK.
TX_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.
*<ref_clk(1 of 3)>*_LEFT[2]	Input	This port is used when RocketIO transceivers in the Aurora core are placed on the left edge of a Virtex-4 device. The port can be REF_CLK1_LEFT, REF_CLK2_LEFT, or GREF_CLK_LEFT. The rate of the clock depends on the desired data rate for the module. See "Setting the Clock Rate for Virtex-4 Designs," page 86.[3]
*<ref_clk(1 of 3)>*_RIGHT[2]	Input	This port is used when RocketIO transceivers in the Aurora core are placed on the right edge of a Virtex-4 device. The port can be REF_CLK1_RIGHT, REF_CLK2_RIGHT, or GREF_CLK_RIGHT. The rate of the clock depends on the desired data rate for the module. See "Setting the Clock Rate for Virtex-4 Designs," page 86.[3]

Notes:

1. SYNC_CLK is used for 2-byte per lane designs only.
2. The variable, *<ref_clk(1 of 3)>*, refers to any one of three clocks: REFCLK1, REFCLK2, or GREFCLK.
3. Refer to the *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide* for more on MGT clocking.

Parallel Clocks for Virtex-4 Designs

Connecting USER_CLK and TX_OUT_CLK 4-Byte Lane Designs

Virtex-4 4-byte lane Aurora cores use a single clock to synchronize all signals between the core and the user application called USER_CLK. 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 must be frequency locked to the reference clock. Typically TX_OUT_CLK is used to drive the input of the USER_CLK BUFG. Figure 7-8, page 84 shows the typical configuration for a 4-byte lane Virtex-4 core. DCM_NOT_LOCKED is typically tied low in this configuration.

Connecting USER_CLK, SYNC_CLK, and TX_OUT_CLK for 2-Byte Lane Designs

Virtex-4 2-byte lane Aurora cores use two 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 the TX_OUT_CLK rate, which is determined by the Clock Settings for the design. The TX_OUT_CLK rate 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 8B/10B encoding and decoding.

The second phase-locked parallel clock is SYNC_CLK. The rising edge of this clock must coincide with the falling edge of USER_CLK. This clock must also come from a BUFG and is half the rate of TX_OUT_CLK. It is connected directly to the Aurora core to drive the internal synchronization logic of the RocketIO MGT.

To make it easier to use the two parallel clocks, a clock module is provided with each 2-byte lane Virtex -4 Aurora core, in a subdirectory called clock_module. The ports for this module are described in Table 7-2, page 81; Figure 7-9, page 85 shows the typical 2-byte lane configuration, including the clock module. If the clock module is used, the DCM_NOT_LOCKED signal should be connected to the DCM_NOT_LOCKED output of the clock module, TX_OUT_CLK should connect to the clock module's MGT_CLK port, and TX_LOCK should connect to the clock module's MGT_CLK_LOCKED port. If the clock module is not used, connect the DCM_NOT_LOCKED signal to the inverse of the DCM_LOCKED signal from any DCM used to generate either of the parallel clocks, and use the TX_LOCK signal to hold the DCMs in reset during stabilization if TX_OUT_CLK is used as the DCM's source clock.

Reference Clocks for Virtex-4 Designs

Virtex-4 Aurora cores require low jitter reference clocks for generating and recovering high speed serial clocks in the RocketIO transceivers. The Aurora core Clock interface includes separate reference clock inputs for each edge of the device (LEFT and RIGHT). Each reference clock can be set to one of the three possible Virtex-4 reference clock input ports, called REFCLK1, REFCLK2, and GREFCLK. 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, since they introduce too much jitter.

Using GREFCLK

GREFCLK is a flexible reference clock input for low rate serial connections. It is flexible because it can be driven from any global clock resource in the FPGA except DCMs. GREFCLK can also be routed anywhere on the chip. This means that unlike the REFCLKs, the reference clocks for Aurora cores with RocketIO transceivers placed on both edges can be driven from the same FPGA resources.

GREFCLK is only suitable for low rates. Regular FPGA clock resources introduce too much jitter for fast serial clock rates. GREFCLK should not be used for line rates higher than 1 Gb/s (125 MHz reference clock rate).

RocketIO transceivers on the Virtex-4 FPGA are arranged in pairs called tiles. If GREFCLK is selected as the reference clock input for RocketIO transceivers on one or both edges of the device, each tile in the core can be driven with the same clock input, providing each tile is sourced by its own BUFG. The Aurora core does not currently support separate BUFGs on a per tile basis: up to two RocketIO transceivers placed on the same tile can be driven with GREFCLK without changing the source code of the design. If more BUFGs must be added, they should be placed in the MGT_WRAPPER component, between the input ports of the module and the instances of the RocketIO transceivers. This operation is not recommended for normal applications.

Using REFCLK1 or REFCLK2

The REFCLK1 and REFCLK2 inputs are used for high data rate applications (1 Gb/s or higher). They are routed to the MGTs on the same edge through low-jitter reference clock trees. In order to route REFCLK1 or REFCLK2, an MGT clock module, GT11CLK_MGT, must be instantiated.

REFCLK1 Setup

Figure 7-8 shows REFCLK1 connected to the MGTs on the left (or right) edge of the device using the GT11CLK_MGT primitive.

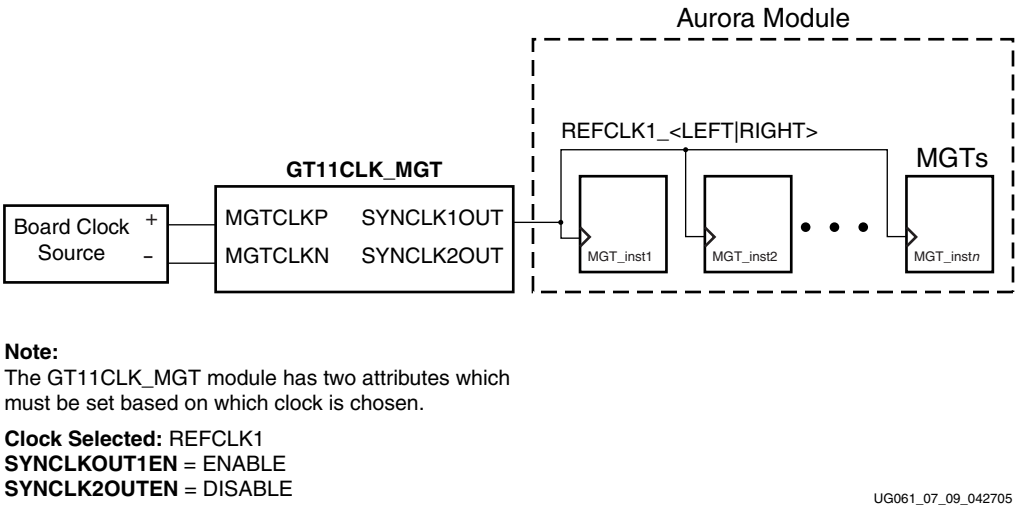
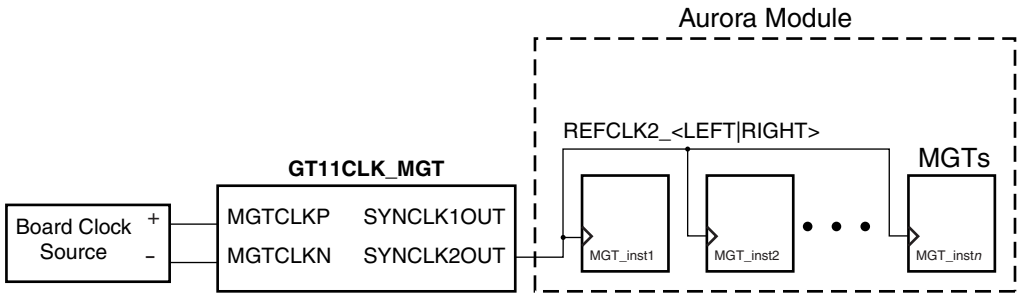


Figure 7-8: **GT11CLK_MGT Connections to the REFCLK1_* Port**

To assure proper configuration of the REFCLK1 connection, the user must perform the following steps:

1. Set GT11CLK_MGT attributes using the **`defparam`** statements (or generic map in case of VHDL) in the component instantiation section of the module.
   a. Set SYNCLK1OUTEN = ENABLE
   b. Set SYNCLK2OUTEN= DISABLE
2. Lock the clock pins to the clock source pin locations at the board level.
   a. The constraints must be specified in the UCF file.
   b. For MGTs on the left side, either the top or the bottom dedicated clock pins on the left side are used.
   c. For MGTs on the right side, either the top or the bottom dedicated clock pins on the right side are used.
3. Specify the LOC constraints for the intended GT11CLK_MGT module, ensuring the location matches the dedicated clock pins from Step 2 (above).



Note:
The GT11CLK_MGT module has two attributes which must be set based on which clock is chosen.

Clock Selected: REFCLK2
SYNCLKOUT1EN = DISABLE
SYNCLK2OUTEN = ENABLE

UG061_07_10_042705

Figure 7-9: **GT11CLK_MGT Connections to the REFCLK2_* Port**

As shown in Figure 7-9, the main difference between the GT11CLK_MGT connections for the REFCLK1 and REFCLK2 is that REFCLK1 has to be connected to the SYNCLK1OUT port while REFCLK2 has to come from the SYNCLK2OUT port.

- The SYNCLK1OUTEN and SYNCLK2OUTEN attributes of the GT11CLK_MGT have to be set. The correct values for these attributes are shown in Figure 7-9.
- The clock pins have to be locked to the clock source pins on the board. The constraints have to be specified in the UCF file. Either the top or the bottom dedicated clock pins on the left side of the board can be connected to the MGTs on the left edge.
- The LOC constraints for the GT11CLK_MGT module have to be specified. There are two GT11CLK_MGT modules on each edge of the MGT. The location of one of these has to be specified in the UCF file and connected up as in Figure 7-9.

Connecting REFCLK1 to the MGTs on the Right Edge

- If the MGTs on the right edge are chosen along with REFCLK1, the connections to the GT11CLK_MGT as well as the attribute settings are the same as in Figure 7-8, page 84.
- The clock pin locations have to be specified in the UCF file. Either the top or bottom dedicated clock pins on the right side of the board can be connected.
- The corresponding GT11CLK_MGT module on the right edge as the device has to be specified in the UCF file.

Connecting REFCLK2 to the MGTs on the Right Edge

- If the MGTs on the right edge are chosen along with REFCLK2, the connections to the GT11CLK_MGT as well as the attribute settings are the same as in Figure 7-9, page 85.
- The clock pin locations have to be specified in the UCF file. Either the top or bottom dedicated clock pins on the right side of the board can be connected.
- The corresponding GT11CLK_MGT module on the right edge of the device has to be specified in the UCF file.

Connecting REFCLK1/REFCLK2 to Multiple MGTs on Both Edges

If multiple MGTs on both edges as chosen, the same reference clock is routed to both edges.

- One GT11CLK_MGT module should be instantiated for each edge. Two GT11CLK_MGT modules are required. The connections and attribute settings to each should be done as in Figure 7-8, page 84 for REFCLK1 and Figure 7-9, page 85 for REFCLK2.
- The locations of the corresponding GT11CLK_MGT module on the left and right edges have to be specified in the UCF file.
- The clock pins have to be locked to dedicated clock pins on the left and the right edge. Either top or bottom clocks can be chosen for both edges.

Note:

1. To find the correct locations for the dedicated clock pins and the corresponding GT11CLK_MGT, refer to the *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide*. When a particular dedicated clock pin is chosen, the corresponding GT11CLK_MGT location has to also be specified. For example, in an XC4VFX60FF1152, if K1, L1 are chosen as the clock pins, then the corresponding GT11CLK_MGT location is X1_Y3.
2. For examples on how to connect the clocks and specify the constraints in the UCF, refer to the aurora_example.ucf file in the ucf directory as well as the aurora_example.v or aurora_example.vhd file in the examples directory.

Setting the Clock Rate for Virtex-4 Designs

Virtex-4 RocketIO transceivers have support a wide range of serial rates. The attributes used to configure the RocketIO transceivers in the Aurora core for a specific rate are kept in the UCF for each core; the same attribute settings also appear in the MGT_WRAPPER module for simulation. These attributes are set automatically by the CORE Generator tool in response to the line rate and reference clock selections made in the Configuration GUI window for the core. Manual edits of the attributes are not recommended, but are possible using the recommendations in the *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide*.

Clock Distribution Examples for Virtex-4 Designs

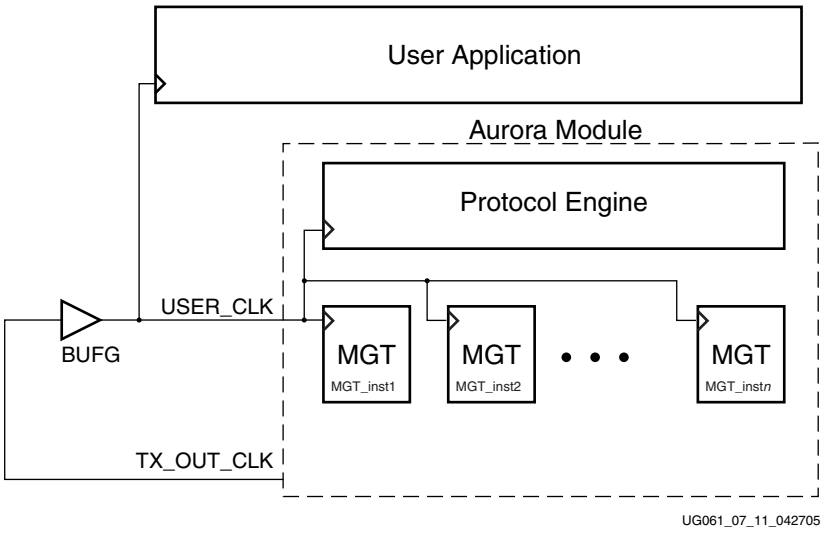


Figure 7-10: **TX_OUT_CLK and USER_CLK Connections**

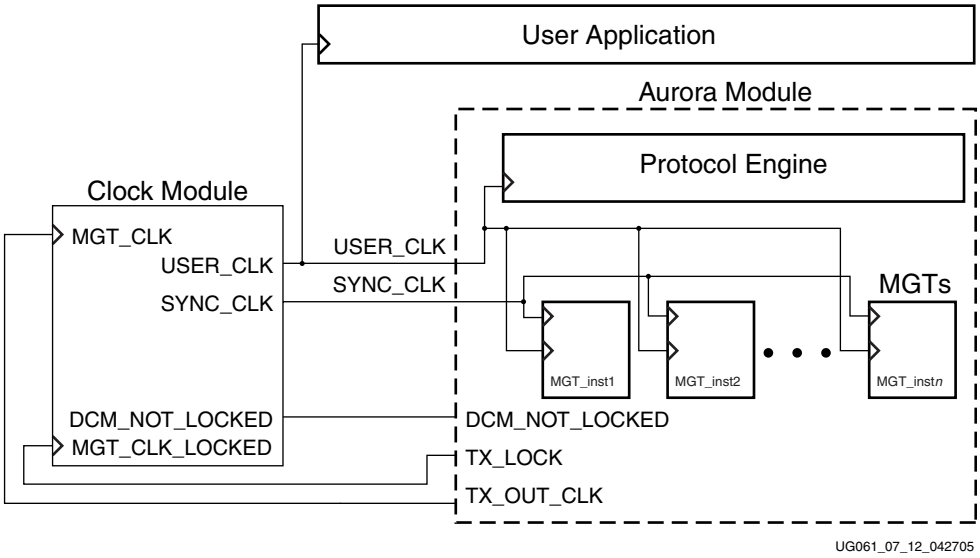


Figure 7-11: **TX_OUT_CLK, USER_CLK, SYNC_CLK**

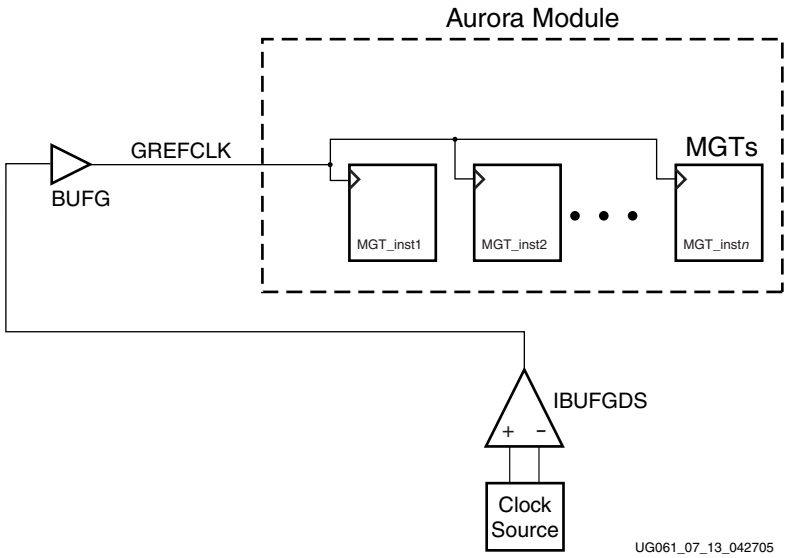


Figure 7-12: **GREFCLK Connections**

Chapter 8

Clock Compensation

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.

The Aurora 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 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.

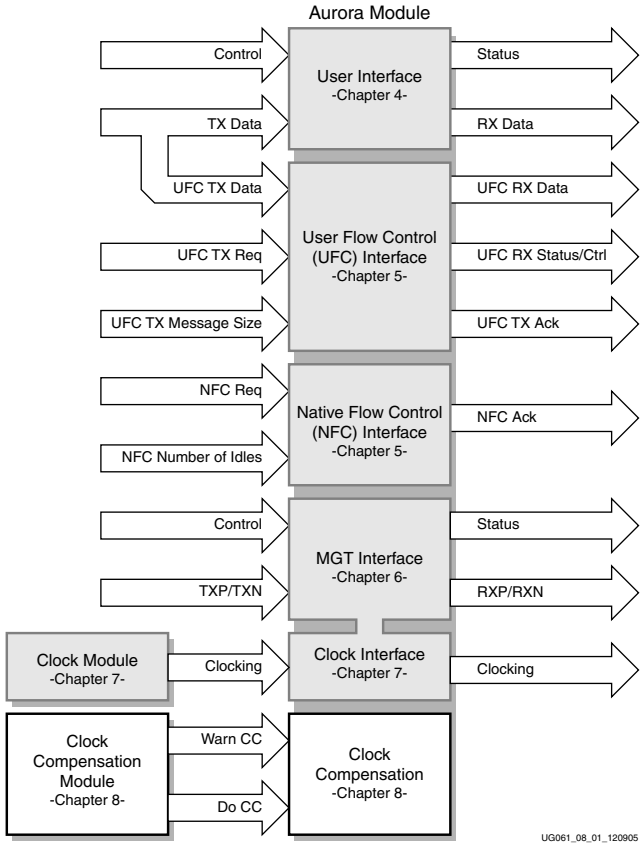


Figure 8-1: **Top-Level Clock Compensation**

Clock Compensation Interface

All Aurora 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 core sends CC sequences on all lanes on every clock cycle when this signal is asserted. Connects to the DO_CC output on the CC module.
WARN_CC	Input	The Aurora core will not acknowledge UFC requests while this signal is asserted. It is used to prevent UFC messages from starting too close to CC events. Connects to the WARN_CC output on the CC module.

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

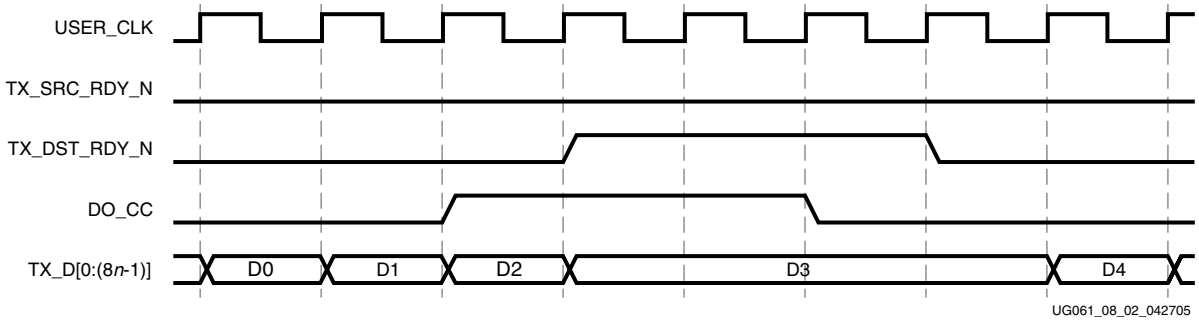


Figure 8-2: **Streaming Data with Clock Compensation Inserted**

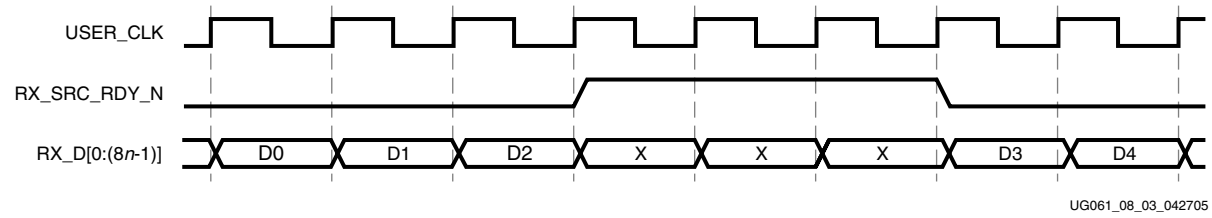


Figure 8-3: **Data Reception Interrupted by Clock Compensation**

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 several cycles every clock compensation period. The duration of the DO_CC assertion and the length of time between assertions is determined based on the width of the RocketIO data interface. While DO_CC is asserted, TX_DST_RDY_N on the user interface for modules

with TX while the channel is being used to transmit clock compensation sequences. Table 8-2 shows the required durations and periods for 2-byte and 4-byte wide lanes.

Table 8-2: **Clock Compensation Cycles**

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

WARN_CC is for cores with user flow control (UFC). Driving this signal before DO_CC is asserted prevents the UFC interface from acknowledging and sending UFC messages too close to a clock correction sequence. This precaution is necessary because data corruption occurs when CC sequences and UFC messages overlap. The number of lookahead cycles required to prevent a 16-byte UFC message from colliding with a clock compensation sequence depends on the number of lanes in the channel and the width of each lane. Table 8-3 shows the number of lookahead cycles required for each combination of lane width, channel width, and maximum UFC message size.

Table 8-3: **Lookahead Cycles**

Data Interface Width	Max UFC size	WARN_CC Lookahead
2	2	3
2	4	4
2	6	5
2	8	6
2	10	7
2	12	8
2	14	9
2	16	10
4	2-4	3
4	6-8	4
4	10-12	5
4	14-16	6
6	2-6	3
6	8-12	4
6	14-16	5
8	2-8	3
8	10-16	4
10	2-10	3
10	12-16	4
12	2-12	3
12	14-16	4

Table 8-3: **Lookahead Cycles *(Continued)***

Data Interface Width	Max UFC size	WARN_CC Lookahead
14	2-14	3
14	16	4
≥16	2-16	3

To make Aurora compliance easy, a standard clock compensation module is generated along with each Aurora core from the CORE Generator tool, in the cc_manager subdirectory. It automatically generates pulses to create Aurora compliant clock compensation sequences on the DO_CC port and sufficiently early pulses on the WARN_CC port to prevent UFC collisions with maximum-sized UFC messages. This module always be connected to the clock compensation port on the Aurora module, except in special cases. Table 8-4 shows the port description for the Standard CC module.

Table 8-4: **Standard CC I/O Port**

Name	Direction	Description
WARN_CC	Output	Connect this port to the WARN_CC input of the Aurora core when using UFC.
DO_CC	Output	Connect this port to the DO_CC input of the Aurora core.
CHANNEL_UP	Input	Connect this port to the CHANNEL_UP output of a full-duplex core, or to the TX_CHANNEL_UP output of a simplex TX or a simplex Both 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 90) because the reference clock frequencies on both sides of the module are locked. In this case, WARN_CC and DO_CC should both be tied to ground. Additionally, the CLK_CORRECT_USE attribute can be set to false in the user constraints file for the core. This can result in lower latencies for single lane modules.

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

- Clock compensation sequences should last at least 2 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 correction sequences within 8 cycles of one another
- Replacing long sequences of idles (>12 cycles) with CC sequences will result in increased EMI
- DO_CC will have no effect until after CHANNEL_UP; DO_CC should be asserted immediately after CHANNEL_UP since no clock compensation can occur during initialization

Appendix A

Framing Interface Latency

Introduction

Latency through an Aurora core is caused by pipeline delays through the protocol engine (PE) and through the MGTs. The PE pipeline delay increases as the LocalLink interface width increases. The MGT delays are fixed per the features of the MGT.

This section outlines expected latency for the Aurora core's LocalLink user interface in terms of USER_CLK cycles for 2-byte per lane and 4-byte per lane designs. For the purposes of illustrating latency, the following pages show the Aurora modules partitioned into MGT logic and protocol engine (PE) logic implemented in the FPGA fabric. Note that these figures do not include the latency incurred due to the length of the serial connection between each side of the Aurora channel.

For 2-Byte Lane Designs

For 2-byte lane designs, the pipeline delays are designed to maintain the clock speed.

Frame Path in 2-Byte Lane Designs

Figure A-1 illustrates the latency of the frame path. See Table A-1, page 94 for latency values of each of the four components that contribute to latency.

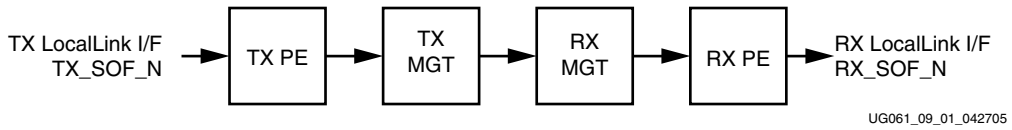


Figure A-1: **Frame Latency (2-Byte Lanes)**

Table A-1 lists the latency of the components of the frame path in terms of MGT clock cycles.

Table A-1: **Frame Latency for 2-Byte Per Lane Designs**

Design	TX PE[1]	TX MGT[2]	RX MGT[3]	RX PE[4]	Total Min[5]	Total Max[6]
201	5	8.5 ± 0.5	24.5 ± 1	5	41.5	44.5
402 . . . 1608				10	46.5	49.5
1809 . . . 3216				12	48.5	51.5
3417 . . . 4020				18	54.5	57.5

Notes Description of Latency

1. From TX_SOF_N to TX MGT
2. From TX MGT to RX MGT (SERDES_10B = FALSE)
3. From RX MGT to RX PE
4. From RX PE to RX_SOF_N
5. From TX_SOF_N to RX_SOF_N (minimum)
6. From TX_SOF_N to RX_SOF_N (maximum)

UFC Path in 2-Byte Lane Designs

Figure A-2 illustrates the latency of the UFC path. See Table A-2 for latency values of each of the four components that contribute to latency.

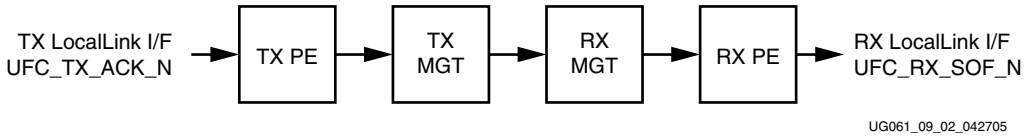


Figure A-2: **UFC Latency (2-Byte Lane)**

Table A-2 lists the latency of the components of the UFC path in terms of MGT clock cycles.

Table A-2: **UFC Latency for 2-Byte Per Lane Designs**

Design	TX PE[1]	TX MGT[2]	RX MGT[3]	RX PE[4]	Total Min[5]	Total Max[6]
201	5	8.5 ± 0.5	24.5 ± 1	5	41.5	41.5
402 . . . 1608				9	45.5	48.5
1809 . . . 4020				10	46.5	49.5

Notes Description of Latency

1. From UFC_TX_ACK_N to TX MGT
2. From TX MGT to RX MGT (SERDES_10B = FALSE)
3. From RX MGT to RX PE
4. From RX PE to UFC_RX_SOF_N
5. From UFC_TX_ACK_N to UFC_RX_SOF_N (minimum)
6. From UFC_TX_ACK_N to UFC_RX_SOF_N (maximum)

NFC Path in 2-Byte Lane Designs

Figure A-3 illustrates the latency of the NFC path. See Table A-3 for latency values of each of the four components that contribute to latency.

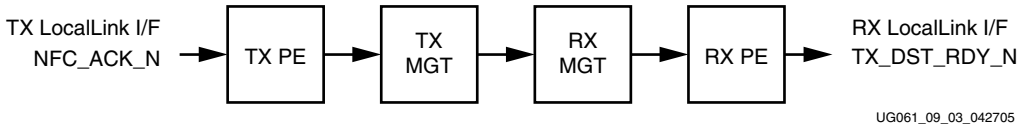


Figure A-3: **NFC Latency (2-Byte Lanes)**

Table A-3 lists the latency of the components of the NFC path in terms of MGT clock cycles.

Table A-3: **NFC Latency for 2-Byte Per Lane Designs**

Design	TX PE[1]	TX MGT[2]	RX MGT[3]	RX PE[4]	Total Min[5]	Total Max[6]
201	4	8.5 ± 0.5	24.5 ± 1	5	40.5	43.5
402 . . . 1608				6	41.5	44.5
1809 . . . 4020				7	42.5	45.5

Notes Description of Latency

1. From NFC_ACK_N to TX MGT
2. From TX MGT to RX MGT (SERDES_10B = FALSE)
3. From RX MGT to RX PE
4. From RX PE to TX_DST_RDY_N
5. From NFC_ACK_N to TX_DST_RDY_N (minimum)
6. From NFC_ACK_N to TX_DST_RDY_N (maximum)

For 4-Byte Per Lane Designs

For 4-byte per lane designs, the pipeline delays are designed to maintain the clock speed.

Frame Latency in 4-Byte Lane Designs

Figure A-4 illustrates the latency of the frame path. See Table A-4 for latency values of each of the four components that contribute to latency.

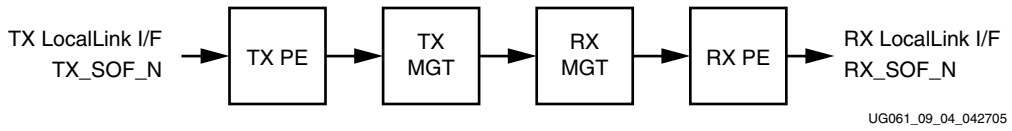


Figure A-4: **Frame Latency (4-Byte Lane)**

Table A-4 lists the latency of the components of the frame path in terms of MGT clock cycles.

Table A-4: **Frame Latency for 4-Byte Per Lane Designs**

Design	TX PE[1]	TX MGT[2]	RX MGT[3]	RX PE[4]	Total Min[5]	Total Max[6]
401 . . . 1604	5	10 ± 0.5	26 ± 1	10	49.5	52.5
2005 . . . 3208				12	51.5	54.5
3609 . . 5614				18	57.5	60.5
6015 . . 8020				TBD	TBD	TBD

Notes Description of Latency

1. From TX_SOF_N to TX MGT
2. From TX MGT to RX MGT (SERDES_10B = FALSE)
3. From RX MGT to RX PE
4. From RX PE to RX_SOF_N
5. From TX_SOF_N to RX_SOF_N (minimum)
6. From TX_SOF_N to RX_SOF_N (maximum)

UFC Latency in 4-Byte Lane Designs

Figure A-5 illustrates the latency of the UFC path. See Table A-5 for latency values of each of the four components that contribute to latency.

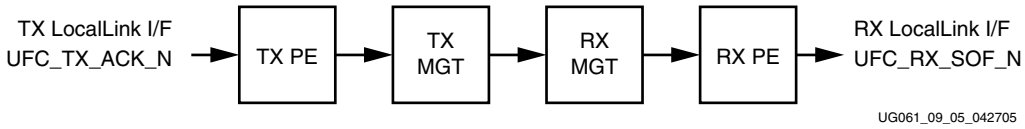


Figure A-5: **UFC Latency (4-Byte)**

Table A-5 lists the latency of the components of the UFC path in terms of MGT clock cycles.

Table A-5: **UFC Latency for 4-Byte Per Lane Designs**

Design	TX PE[1]	TX MGT[2]	RX MGT[3]	RX PE[4]	Total Min[5]	Total Max[6]
401 . . . 1604	5	10 ± 0.5	26 ± 1	9	48.5	51.5
2005 . . . 5614				10	49.5	52.5
6015 . . . 8020				TBD	TBD	TBD

Notes Description of Latency

1. From UFC_TX_ACK_N to TX MGT
2. From TX MGT to RX MGT (SERDES_10B = FALSE)
3. From RX MGT to RX PE
4. From RX PE to UFC_RX_SOF_N
5. From UFC_TX_ACK_N to UFC_RX_SOF_N (minimum)
6. From UFC_TX_ACK_N to UFC_RX_SOF_N (maximum)

NFC Latency in 4-Byte Lane Designs

Figure A-6 illustrates the latency of the NFC path. See Table A-6 for latency values of each of the four components that contribute to latency.

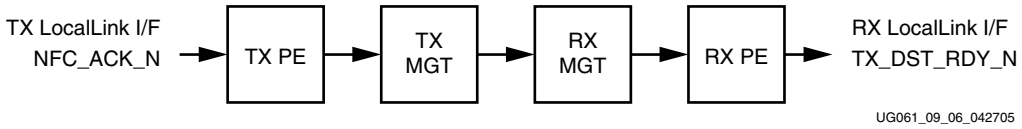


Figure A-6: **NFC Latency (4-Byte)**

Table A-6 lists the latency of the components of the UFC path in terms of MGT clock cycles.

Table A-6: **NFC Latency for 4-Byte Per Lane Designs**

Design	TX PE[1]	TX MGT[2]	RX MGT[3]	RX PE[4]	Total Min[5]	Total Max[6]
401 . . . 1604	4	10 ± 0.5	26 ± 1	6	44.5	47.5
2005 . . . 8020				7	45.5	48.5

Notes Description of Latency

1. From NFC_ACK_N to TX MGT
2. From TX MGT to RX MGT (SERDES_10B = FALSE)
3. From RX MGT to RX PE
4. From RX PE to TX_DST_RDY_N
5. From NFC_ACK_N to TX_DST_RDY_N (minimum)
6. From NFC_ACK_N to TX_DST_RDY_N (maximum)