RIFFA 2.2.2 Documentation

Dustin Richmond, Matt Jacobsen

Thursday 11th August, 2016

Contents

1 Introduction: RIFFA

1.1 What is RIFFA

RIFFA (Reusable Integration Framework for FPGA Accelerators) is a simple framework for communicating data from a host CPU to a FPGA via a PCI Express bus. The framework requires a PCIe enabled workstation and a FPGA on a board with a PCIe connector. RIFFA supports Windows and Linux, Altera and Xilinx, with bindings in C/C++, Python, MATLAB and Java.

On the software side there are two main functions: data send and data receive. These functions are exposed via user libraries in C/C++, Python, MATLAB, and Java. The driver supports multiple FPGAs (up to 5) per system. The software bindings work on Linux and Windows operating systems. Users can communicate with FPGA IP cores by writing only a few lines of code.

On the hardware side, users access an interface with independent transmit and receive signals. The signals provide transaction handshaking and a first word fall through FIFO interface for reading/writing data to the host. No knowledge of bus addresses, buffer sizes, or PCIe packet formats is required. Simply send data on a FIFO interface and receive data on a FIFO interface. RIFFA does not rely on a PCIe Bridge and therefore is not subject to the limitations of a bridge implementation. Instead, RIFFA works directly with the PCIe Endpoint and can run fast enough to saturate the PCIe link.

RIFFA communicates data using direct memory access (DMA) transfers and interrupt signaling. This achieves high bandwidth over the PCIe link. In our tests we are able to saturate (or near saturate) the link in all our tests. The RIFFA distribution contains examples and guides for setting up designs on several standard development boards.

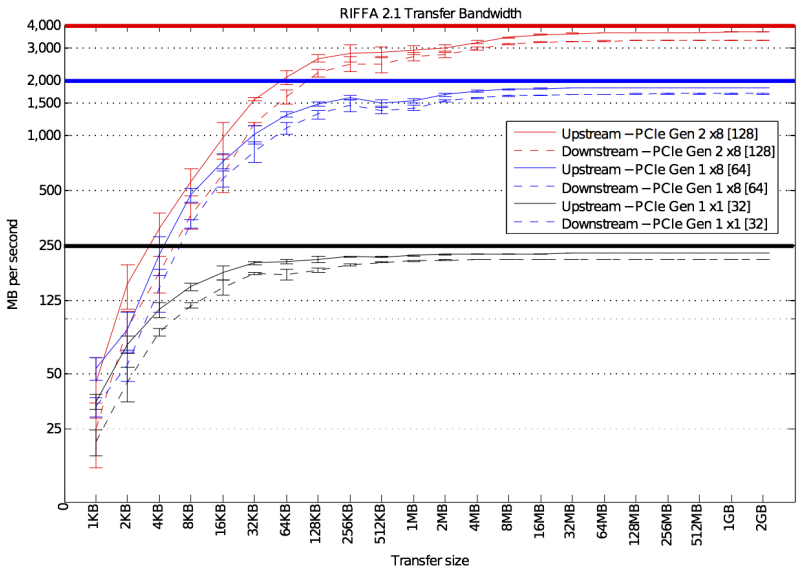

Figure 1.1: Graph of Bandwidth vs Transfer Size

RIFFA 2.2.2 is significantly more efficient than its predecesor RIFFA 1.0. RIFFA 2.2.2 is able to saturate the PCIe link for nearly all link configurations supported. Figure 1.1 shows the performance of designs using the 32 bit, 64 bit, and 128 bit interfaces. The colored bands show the bandwidth region between the theoretical maximum and the maximum achievable. PCIe Gen 1 and 2 use 8 bit / 10 bit encoding which limits the maximum achievable bandwidth to 80% of the theoretical. Our experiments show that RIFFA can achieve 80% of the theoretical bandwidth in nearly all cases. The 128 bit interface achieves 76% of the theoretical maximum.

If you are using RIFFA on a new platform not listed above let us know and well help you out!

1.2 Licensing

2 Getting Started

2.1 Development Board Support in RIFFA 2.2.2

RIFFA 2.2.2 supports:

- The VC707, ZC706 and similar boards with the Xilinx IP Core 7-Series Integrated Block for PCI Express. Example designs for the VC707 and ZC706 boards are provided, and contain this core. The current distribution supports all 64-bit interfaces for these devices, with 128-bit support coming soon after the initial release. (Support for the 128-bit interface is in RIFFA 2.1, but is temporarily mising due to changes)
- The VC709 board and similar boards with the Xilinx IP Gen3 Integrated Block for PCI Express. Example designs for the VC709 are provided, and contain this core. The current distribution supports all 64-bit and 128-bit AXI interfaces. 256-bit (PCIe Gen3 x8) support is planned for a later date.
- The DE5-Net board and similar boards with the Stratix V, Cyclone V, and Arria V, Hard IP for PCI express (Avalon Streaming Interface). Example designs for the DE5-net board are provided, and contain the Stratix V version of this core. The current distribution supports all 64-bit and 128-bit Avalon Streaming interfaces.
- The DE4 and similar boards with the IP Compiler for PCI Express Core, supporting Stratix IV, Cyclone IV and Arria II devices. Example designs for the DE4 board are provided. The current distribution supports all 64-bit and 128-bit Avalon Streaming interfaces.

2.2 Understanding this User Guide

In this user guide, we use the following conventions:

Object	Example
Directories and Paths	*RIFFA 2.2.2/source/fpga/riffa*
Xilinx Specific Content	vc709
Altera Specific Content	de5
Configuration Setting	**Number of Lanes**
Terminal Command, Code Snippet	$ `echo ''Hello World''`
RIFFA Parameter	***C_NUM_CHNL***

2.3 Decoding What's Provided

Fig 2.1 shows the directory hierarchy of RIFFA. This instruction manual uses this directory tree when specifying all directory paths.

The *RIFFA 2.2.2/source/fpga/* contains a directory for each board we have tested for the current distribution:de5, de4, VC709, VC707, ZC706. Each board directory has several example project directories (e.g. DE5Gen1x8If64 and VC709_Gen1x8If64). Each example project directory has 5 sub-directories:

- *prj/* contains all of the project files (.qsf,.qpf, .xpr).

- *ip/* contains all of the ip files (.qsys, .xci) generated for the project, when permitted by licensing agreements.
- *bit/* contains the example programming file for the corresponding FPGA example design. Quartus and Vivado do not modify this programming (.sof, .bit).
- *constr/* contains the user constraint files (.sdc, .xdc).
- *hdl/* contains any example-project specific Verilog files, such as the project top level file.

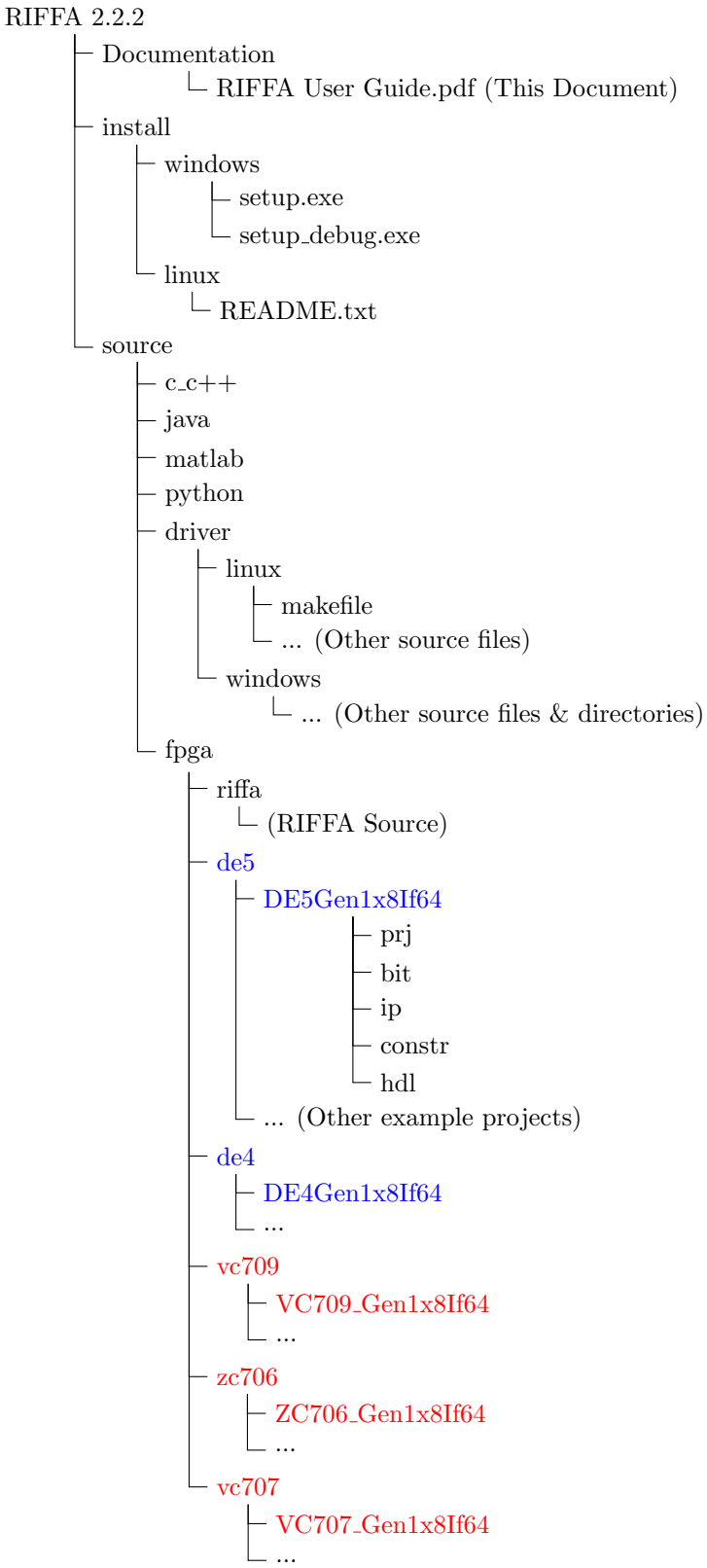

Figure 2.1: Directory hierarchy of the RIFFA 2.2.2 distribution

2.4 Release Notes

2.4.1 Version 2.2.2

- Fixed: Unsigned Windows Driver (Commit: 4e989fc)
- Fixed: A bug in the clock-crossing interface where a low-frequency user clock (¡40 MHz) could cause incorrect channel behavior. (Commit: 778c42e)
- Fixed: Support for 64-bit pointers in Python 3. Shout out to @jrobrien (Commit: af7c592)
- Fixed: Includes in linux driver. linux/slab.h was not included in riffa_driver.c (Commit: cd494e1)
- Fixed: Capability backwards/forwards compatibility issues to support Linux for Tegra. (Commit: 1d228c1)
- New: Support for new get_user_pages API in the linux kernel. Shout out to @marzoul (Commit: TBD)
- Removed: Non-Qsys DE5 Board Example Designs (QSys ¿ IP Generator) @marzoul (Commit: TBD)

2.4.2 Version 2.2.1

- New: Reset logic for the Engine layer to handle RIFFA induced resets
- New: Stability/multi-thread-concurrency warnings in the Linux driver (Shoutout to @marzoul)
- Fixed: A bug in the Linux Driver that prevented compilation on older kernels
- Fixed: Windows driver issue for back-to-back small transfers (See 2.2.0)
- Fixed: WORD_ENABLE bug fix for the Classic Xilinx (VC707, ZC706, AC701, KC705) 128-bit interface
- Fixed: TX Engine Buffer sizing (high-bandwidth transfers occasionally had corruption)
- Fixed: RX Engine rx_st_valid bug fix for Altera IP Compiler for PCI express (Cyclone IV, Stratix IV)

2.4.3 Version 2.2.0

- Added: Support for the new Gen3 Integrated Block for PCIe Express, and the VC709 Development board.
- Added: ZC706 Example Designs
- Changed: Xilinx example project packaging. All Xilinx Virtex 7 projects are now click-to-compile, and come with instantiated IP.
- Re-wrote and refactored: Various parts of the TX and RX engines to maximize code reuse between different vendors and PCIe endpoint implementations
- Fixed: A bug in the Linux Driver that prevented compilation on older kernels
- Fixed: A bug in the Windows Driver that prevented repeated small transfers.

2.4.4 Version 2.1.0

- Added reorder_queue and updated many rx/tx engine and channel modules that use it.
- Added parameters for number of tags to use and max payload length for sizing RAM for reorder_queue.
- Fixed: Bug in the riffa_driver.c, too few circular buffer elements.
- Fixed: Bug in the riffa_driver.c, bad order in which interrupt vector bits were processed. Can cause deadlock in heavy use situations.
- Fixed: Bug in the tx_port_writer.v, maxlen did not start with a value of 1. Can cause deadlock behavior on second transfer.
- Fixed: Bug in the rx_port_reader.v, added delay to allow FIFO flush to propagate.
- Fixed: Bug in rx_port_xxx.v, changed to use FWFT FIFO instead of existing logic that could cause CHNL_RX_DATA_VALID to drop for a cycle after CHNL_RX dropped even when there is still data in the FIFO. Can cause premature transmission termination.
- Changed rx_port_channel_gate.v to use FWFT FIFO.
- Removed unused signal from rx_port_requester_mux.v.
- Fixed: Typo/bug that would attempt to change state within tx_port_monitor_xxx.v.
- Added flow control for receive credits to avoid over driving upstream transactions (applies to Altera devices).

2.4.5 Version 2.0.2

- Fixed: Bug in Windows and Linux drivers that could report data sent/received before such data was confirmed.
- Fixed: Updated common functions to avoid assigning input values.
- Fixed: FIFO overflow error causing data corruption in tx_engine_upper and breaking the Xilinx Endpoint.
- Fixed: Missing default cases in rx_port_reader, sg_list_requester, tx_engine_upper, and tx_port_writer.
- Fixed: Bug in tx_engine_lower_128 corrupting s_axis_tx_tkeep, causing Xilinx PCIe endpoint core to shut down.
- Fixed: Bug in tx_engine_upper_128 causing incomplete TX data timeouts.
- Changed rx_engine to not block on nonposted TLPs. They're added to a FIFO and serviced in order.
- Reset rx_port FIFOs before a receive transaction to avoid data corruption from replayed TLPs.

2.4.6 Version 2.0.1

- RIFFA 2.0.1 is a general release. This means we've tested it in a number of ways. Please let us know if you encounter a bug.
- Neither the HDL nor the drivers from RIFFA 2.0.1 are backwards compatible with the components of any previous release of RIFFA.

- RIFFA 2.0.1 consumes more resources than 2.0 beta. This is because 2.0.1 was rewritten to support scatter gather DMA, higher bandwidth, and appreciably more signal registering. The additional registering was included to help meet timing constraints.
- The Windows driver is supported on Windows 7 32/64. Other Windows versions can be supported. The driver simply needs to be built for that target.
- Debugging on Windows is difficult because there exists no system log file. Driver log messages are visible only to an attached kernel debugger. So to see any messages you'll need the Windows Development Kit debugger (WinDbg) or a small utility called DbgView. DbgView is a standalone kernel debug viewer that
- http://technet.microsoft.com/ens/sysinternals/bb896647.aspx Run DbgView with administrator privileges and be sure to enable the following capture options: Capture Kernel, Capture Events, and Capture Verbose Kernel Output.
- The Linux driver is supported on kernel version 2.6.27+.
- The Java bindings make use of a native library (in order to connect Java JNI to the native library). Libraries for Linux and Windows for both 32/64 bit platforms have been compiled and included in the riffa.jar.
- Removed the CHNL_RX_ERR signal from the channel interface. Error handling now ends the transaction gracefully. Errors can be easily detected by comparing the number of words received to the CHNL_RX_LEN amount. An error will cause CHNL_RX will go low prematurely and not provide the advertised amount of data.
- Fixed: Bug in sg_list_requester which could cause an unbounded TLP request.
- Fixed: Bug in tx_port_buffer_128 which could stall the TX transaction.

2.5 Errata

While we have extensively tested the current distribution, we are human and cannot eliminate all bugs in our distribution. As a general rule of thumb, if you find yourself delving into the RIFFA code, you have gone too far. Contact us if you need additional assistance!

See the following notes for issues we are currently tracking:

2.5.1 Windows

2.5.2 Linux

No open issues

2.5.3 Altera

Issue 3: No support for the 256-bit, Gen3x8 Interface Coming soon...

2.5.4 Xilinx (Classic)

Issue 1: Missing example designs for ML605 There is no disadvantage to using RIFFA 2.1.0 until we return support in a future distribution.

Issue 2: Missing example design for Spartan 6 LXT Development board The 32-bit interface support has been removed from RIFFA 2.2 and may be added back in the future. Please use RIFFA 2.1 in the meantime

2.5.5 Xilinx (Ultrascale)

Issue 1: No support for the 256-bit, Gen3x8 Interface Coming soon...

3 Installing the RIFFA driver

3.1 Linux

To install the RIFFA driver in linux, you must build it against your installed version of the Linux kernel. RIFFA 2.2.2 comes with a makefile that will install the necessar linux kernel headers and the driver. This makefile will also build and install the C/C++ native library. To install RIFFA 2.2.2 in linux, follow these instructions:

1. Open a terminal in linux and navigate to the *RIFFA 2.2.2/source/driver/linux* directory.
2. Ensure you have the kernel headers installed, run:

   $ `sudo make setup`

   This will attempt to install the kernel headers using your system's package manager. You can skip this step if you've already installed the kernel headers.
3. Compile the driver and C/C++ library:

   $ `make`

   or

   $ `make debug`

   Using make debug will compile in code to output debug messages to the system log at runtime. These messages are useful when developing your design. However they pollute your system log and incur some overhead. So you may want to install the non-debug version after you've completed development.
4. Install the driver and library:

   $ `sudo make install`

   The system will be configured to load the driver at boot time. The C/C++ library will be installed in the default library path. The header files will be placed in the default include path. You will need to reboot after you've installed for the driver to be (re)loaded.
5. If the driver is installed and there is a RIFFA 2.2.2 configured FPGA when the computer boots, the driver will detect it. Output in the system log will provide additional information.
6. The C/C++ code must include the riffa.h header. An example inclusion is shown in Listing 3.1
7. When compiling (using GCC/G++, etc.) you must link with the RIFFA libraries using the -lriffa flag. For example, when compiling test.c from Listing 3.1:

   $ `gcc -g -c -lriffa -o test.o test.c`
8. Bindings for other languages can be installed by following the README files in their respective directories (See Figure 2.1

3.2 Windows

Currently only Windows 7 (32/64) is supported by RIFFA 2.2.2. In the *RIFFA 2.2.2/install/windows/* subdirectory use the provided setup.exe program to install the RIFFA driver and native C/C++ library. You can verify that RIFFA 2.2.2 installed correctly by checking the installation directory in Program Files. After installation, you'll be able to install the bindings for other languages.

The setup_dbg.exe installer installs a driver with additional debugging output. You can install the setup_dbg.exe version and then later use setup.exe to install the non-debug output version.

Listing 3.1: Inclusion of the RIFFA header files in a user application

```
#include <stdio.h>
#include <stdlib.h>
#include <riffa.h>
#define BUF_SIZE (1*1024*1024)

unsigned int buf[BUF_SIZE];

int main(int argc, char* argv[]) {
  fpga_t * fpga;
  int fid = 0; // FPGA id
  int channel = 0; // FPGA channel

  fpga = fpga_open(fid);
  fpga_send(fpga, channel, (void *)buf, BUF_SIZE, 0, 1, 0);
  fpga_recv(fpga, channel, (void *)buf, BUF_SIZE, 0);
  fpga_close(fpga);
  return 0;
}
```

4 Compiling and using the Xilinx Example Designs

Vivado 2014.4 was used in all example designs and documentation included in this distribution. We highly recommend using 2014.4 and all newer versions of the software, since we have encountered bugs in previous versions of the Vivado (e.g. 2014.2) software. This guide assumes that the end-user has already configured their board for PCI Express operation. See the VC709 User Guide [1], VC707 User Guide [2] or ZC706 User Guide [3].

While we have not tested all of the current-generation Xilinx development boards, we are confident that they can be supported with minimal modifications. For more information about supporting new boards, see the sections 4.1.2 and 4.2.2. These sections cover the settings used in the RIFFA example design IP.

The easiset way to use RIFFA is to start with one of the example designs included in the distribution. Sections 4.1.1 and 4.2.1 describe how to use and compile these designs for the VC707 and VC709 boards respectively. These example designs are ready to compile out of the box, and require no user IP configuration and generation. The designs also include pre-compiled bit-files in the *bit* directory of the example project. For advanced users, we also describe how we generated the PCIe IP in sections 4.1.2 and 4.2.2.

4.1 Classic - 7 Series Integrated Block for PCI Express - (VC707, ZC706 and older)

This is a step by step guide for using RIFFA 2.2.2 on a Xilinx FPGA with the 7 Series Integrated Block for PCI Express Core. This core is supported on the ZC706, and VC707 development boards, using the 64-bit and 128-bit AXI interfaces.

4.1.1 VC707 and ZC706 Example Designs

There is one VC707 example design and two ZC706 example designs in the RIFFA 2.2.2 distribution. The VC707 example design folders are in *RIFFA 2.2.2/source/fpga/vc707* and the ZC706 example design folders are in *RIFFA 2.2.2/source/fpga/zc706*.

1. Open Vivado to get the introductory screen shown in Figure 4.1.
2. Click 'Open an Existing Project' and navigate to your RIFFA 2.2.2 directory.
3. In the RIFFA 2.2.2 distribution, open *RIFFA 2.2.2/source/fpga/xilinx/vc707/* or *RIFFA 2.2.2/source/fpga/zc706* and choose from one of the existing example design directories for your board. In the example design directory, locate the *prj* folder and open it. Select the .xpr file and click open. This will open the example project, as shown in Figure 4.2.
4. This project was compiled in Vivado 2014.4. The bit file generated can be used to test the FPGA system. If you are using a newer version of Vivado, recompile the example design or use the programming file provided.

[1] http://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf
[2] http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug848-VC707-getting-started-guide.pdf
[3] http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf

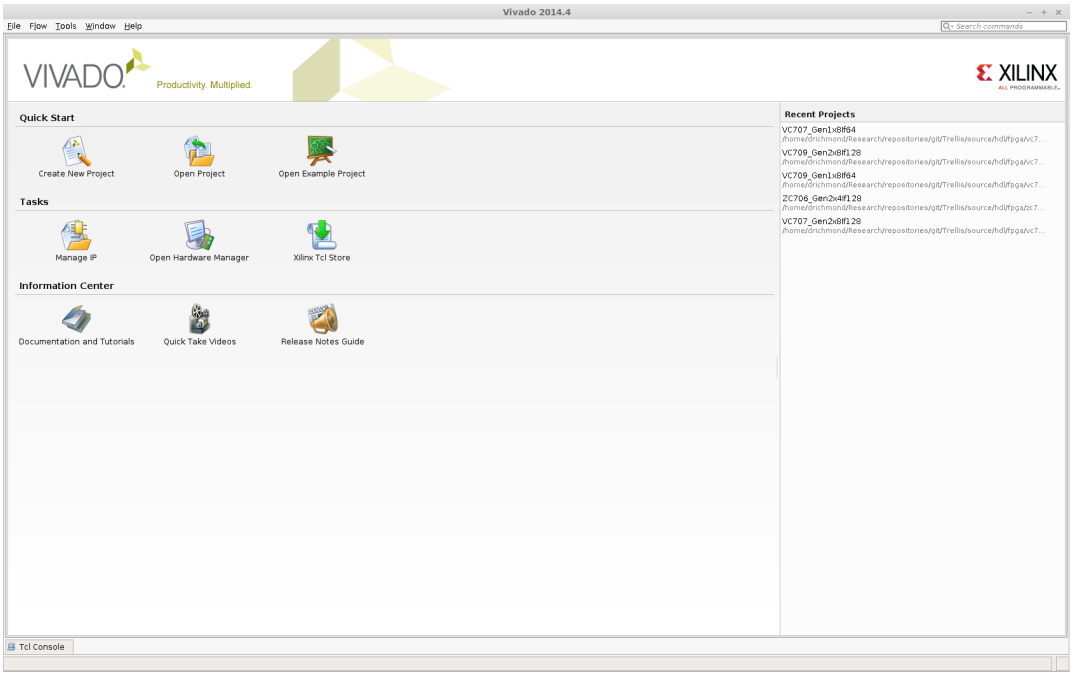

Figure 4.1: Welcome Screen for Vivado 2014.4

- IP Settings are now packaged as part of the example designs! Users no longer need to generate IP.
- To recompile the example design, click the generate bitstream button in the top left corner as shown in Figure 4.2.
- Recompiling your design will generate a new bitfile in the Xilinx project. The bit file in the *bit* will not be changed.

5. To program the FPGA, click 'Open Hardware Manager'. New bit files (generated by Vivado) will appear in Vivado's internal directory. An example bit file is provided in the example design's *bit*. Load the bitstream to your VC707 or ZC706 board and restart your computer.
   - Before programming your FPGA, you should install the RIFFA driver. See Section 3
6. The example design uses the chnl_tester (shown in Figure 4.3, which works with the example software in the *source/{C_C++,Java,python,matlab}* directories. Replace the chnl_tester instantiation with any user logic, matching the RIFFA interface.
7. Recompile the design and program the FPGA Device. Changing the ***C_NUM_CHNL*** will change the number of independent channel interfaces

4.1.2 Generating the 7 Series Integrated Block for PCI Express

The following steps are not required for general users. See the instructions above for how to compile RIFFA.

Alternatively, it is possible to generate the PCIe Endpoint with different settings than those provided in the example design. Modifying the RIFFA parameters ***C_PCI_DATA_WIDTH***, ***C_MAX_PAYLOAD_BYTES*** and ***C_LOG_NUM_TAGS***, change certain settings in the IP

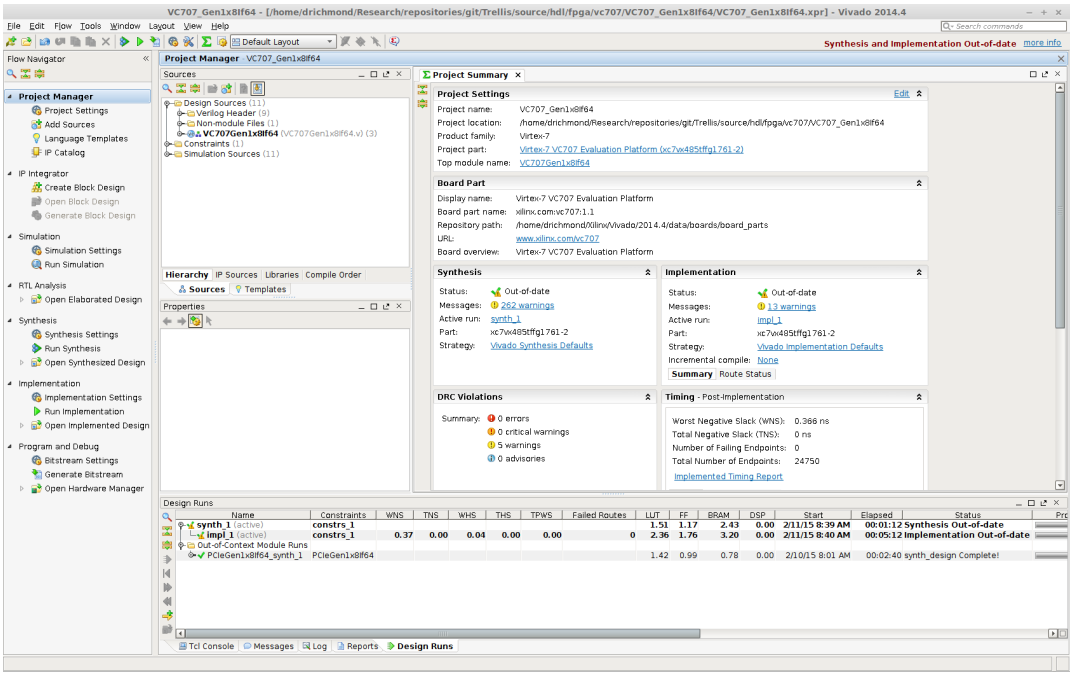

Figure 4.2: Project Splash Screen for 7Series Integrated Block for PCI Express Projects

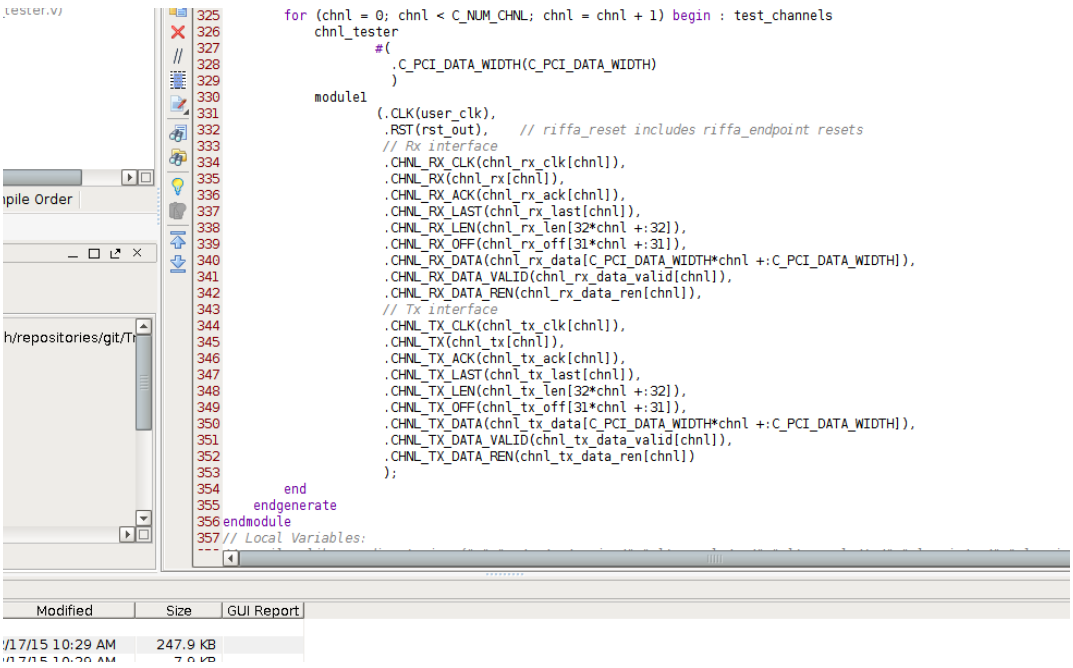

Figure 4.3: Project Splash Screen for 7Series Integrated Block for PCI Express Projects

Core. The ***C_NUM_LANES*** is a parameter in the top level file of each example project. How these parameters relate to IP core settings is highlighted in the following figures.

If the goal is to generate a RIFFA design completely from scratch, each board directory comes with a RIFFA wrapper verilog file and instantiates a vendor-specific translation layer. It is highly recommended to re-use these files RIFFA wrapper when creating designs from scratch.

To generate the PCIe IP select the 7 Series Integrated Block for PCI Express after selecting the IP Catalog shown in Figure 4.2. This will open the IP Customization window as shown in Figure 4.4

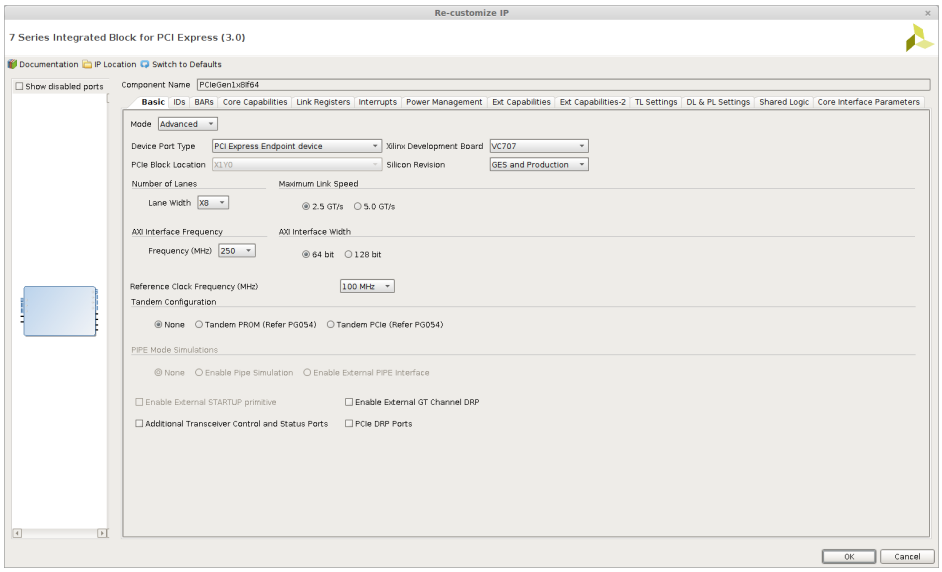

Figure 4.4: Basic settings tab.

First, select **Mode** to **ADVANCED** from the drop down menu. This will cause more tabs to appear in the bar. The following tabs are not used during customization: Link Registers, Power Mangement, Ext. Capabilities, Ext. Capabilities 2, TL Settings and DL/PL Settings.

In Figure 4.4 , we have set the **Xilinx Development Board** to **VC707**, selected the PCIe Gen1 rate **2.5 GT/s**, and a **Lane Width** of **8** (***C_NUM_LANES*** = 8). We have chosen to set the **AXI Interface Width** to **64-bits** (***C_PCI_DATA_WIDTH*** = 64). The choice of Link Rate, Lanes, and Interface Width will allow different AXI Interface Frequencies to be selected. The RIFFA core will run at this clock frequency, but the user logic can run at whatever frequency it desires.

Optional: Set the Component Name of the PCI Express block, and the IP Location. In our example projects, we use the name template PCIeGen**W**x**Y**If**Z** where **W** is the PCI Express Version (**Link Speed** in Figure 4.4), **Y** is the lane width, and **Z** is the AXI interface width. The IP location is the *ip* directory in the example project.

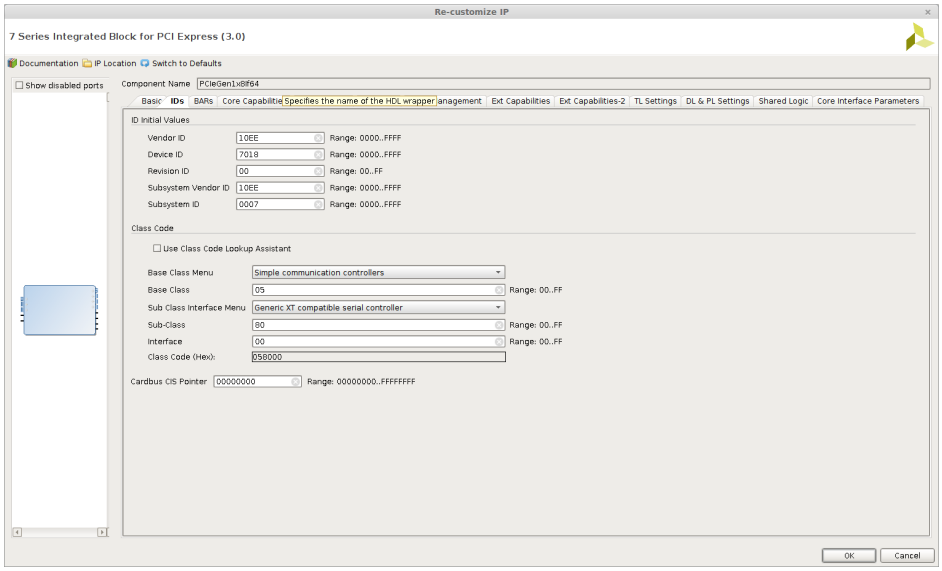

Figure 4.5: PCI Express ID Tab.

The tab in Figure 4.5 is optional. Setting the Device ID may assist in identifying different FPGAs in a multi-FPGA system. The other options, specifically the Vendor ID, must remain the same.

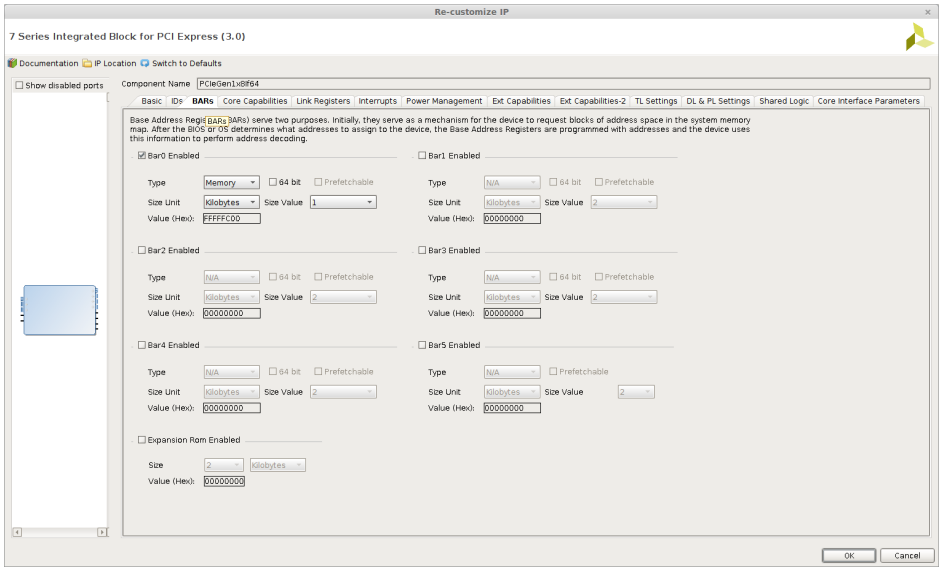

Figure 4.6: PCI Express Base Address Register (BAR) ID Tab.

The tab in Figure 4.6 must be configured so that BAR0 is **enabled** (checked). Set the **Type** to **Memory**, and **Unit Kilobyte**, and **Size Value** to **1** from the dropdown menus. If these values are not set correctly the RIFFA driver will not recognize the FPGA device.

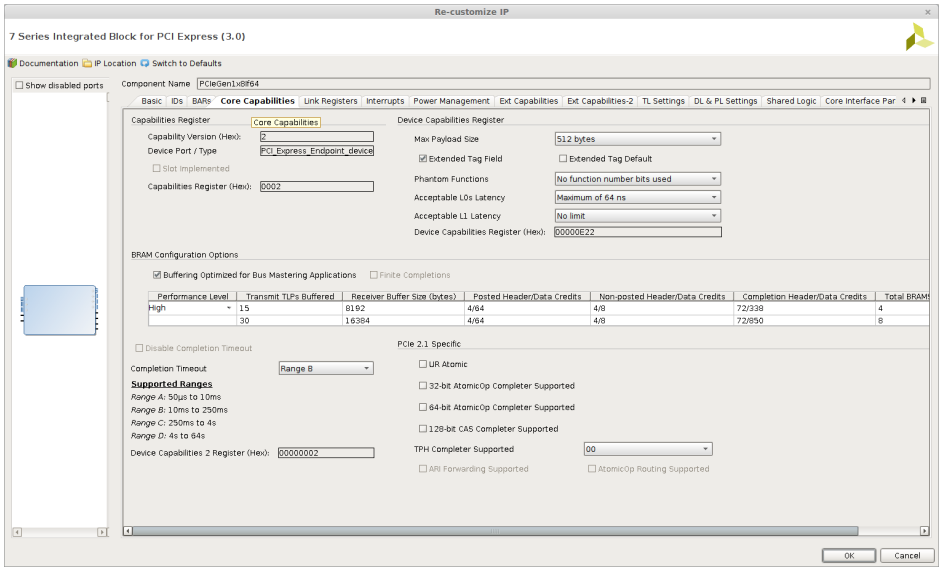

Figure 4.7: PCI Express Capabilities Tab.

In this tab select the boxes **Buffering Optimized for Bus Mastering Applications** and **Extended Tag Field**. If the **Extended Tag Field** is selected ***C_LOG_NUM_TAGS*** = 8, otherwise ***C_LOG_NUM_TAGS*** = 5. Select the **Maximum Payload Size** from the dropdown menu. Use this to set the RIFFA ***C_MAX_PAYLOAD_BYTES*** parameter.

Note: Maximum Payload sizes are typically set by the BIOS, and 256 bytes seems to be standard. RIFFA will default to the minimum of ***C_MAX_PAYLOAD_SIZE*** and the setting in your BIOS. Unless your BIOS is modified, or can support substantially larger packets, there will be no performance benefit to increasing the payload size. Increasing the maximum payload size will increase the resources consumed.

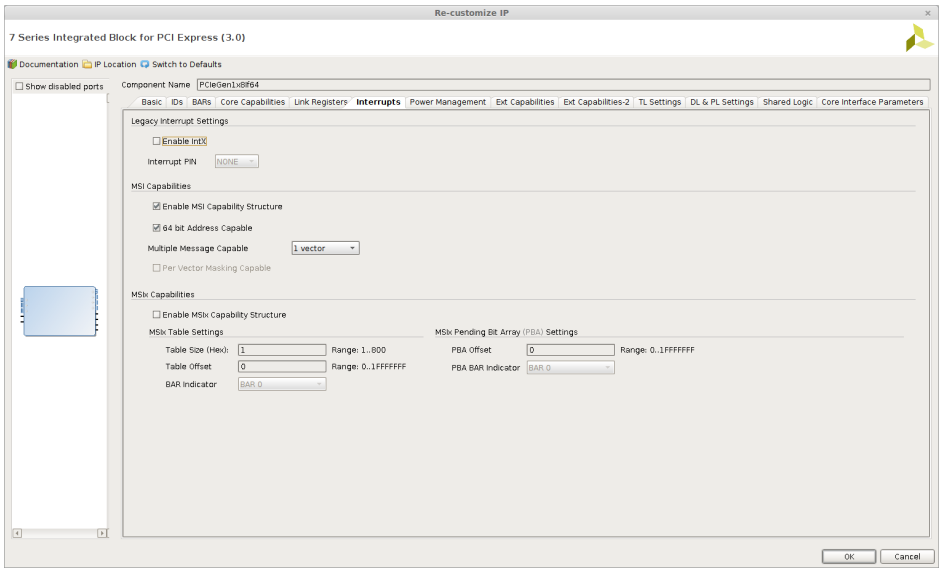

Figure 4.8: PCI Express Interrupts Tab.

In the Interrupts Tab shown in Figure 4.8 **clear** the checkbox for **Enable INTx** (To disable INTx). The remaining options should match those shown in Figure 4.8

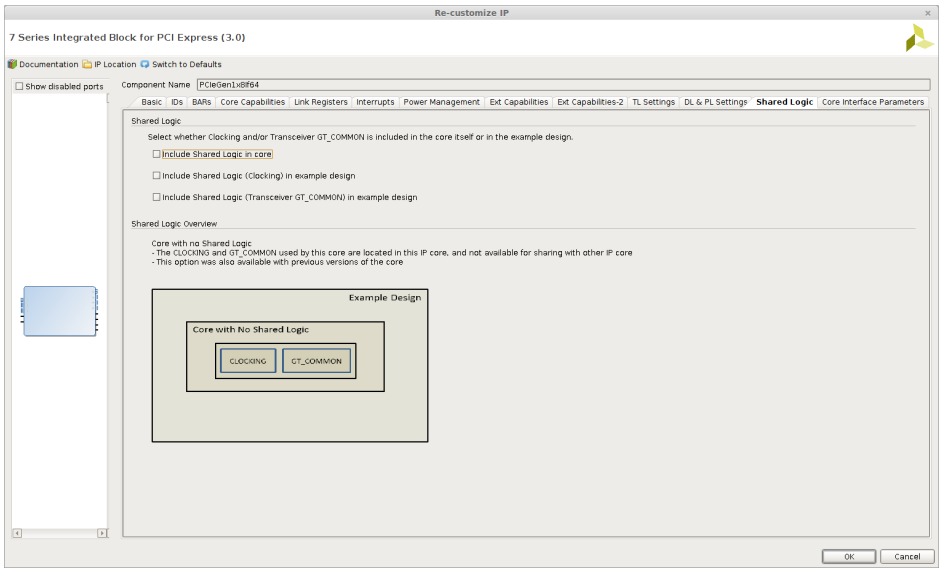

Figure 4.9: Shared Logic Tab

In the Shared Logic Tab shown in Figure 4.9 **clear** all of the checkboxes shown. These settings will not affect the core generated, but will affect the example designs generated by Vivado. As a result, the Example Design will mirror the RIFFA example design provided.

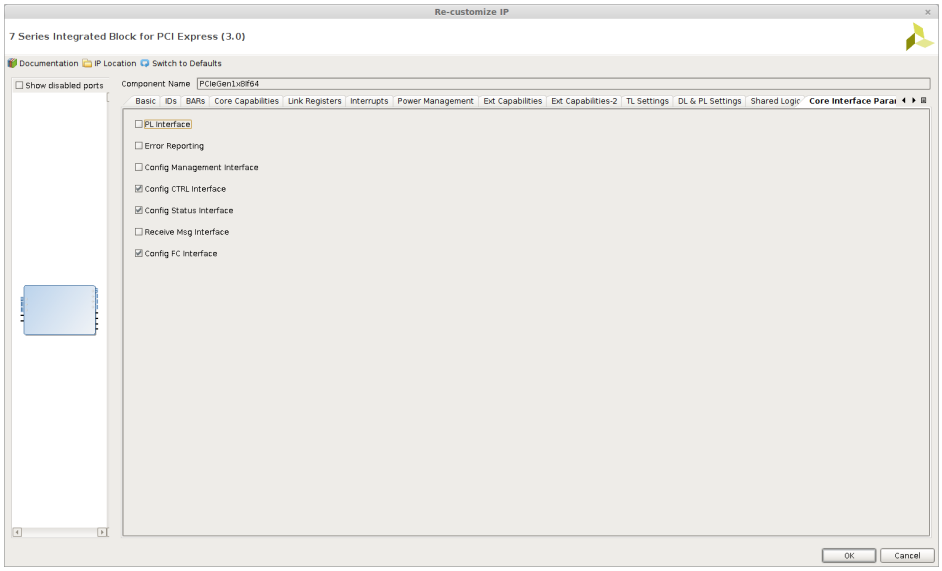

Figure 4.10: Core Interface Parameters Tab

Finally, in the Interface Parameters tab, match the checkboxes shown in Figure 4.10. These options simplify the interface to the generated core

4.1.3 Creating Constraints files for the VC707 Development Board

When generating a design for the VC707 board, the following constraints will correctly constrain the clocks. When using a different board, read the user guide for appropriate pin placment, or copy the constraints from the PCIe Endpoint Example Design. The remaining constraints are contained the generated PCIe IP.

Listing 4.1: .xdc constraints for the VC707 board

```
set_property PACKAGE_PIN AV35 [get_ports PCIE_RESET_N]
set_property IOSTANDARD LVCMOS18 [get_ports PCIE_RESET_N]
set_property PULLUP true [get_ports PCIE_RESET_N]
# The following constraints are BOARD SPECIFIC. This is for the VC707
set_property LOC IBUFDS_GTE2_X1Y5 [get_cells refclk_ibuf]
create_clock -period 10.000 -name pcie_refclk [get_pins refclk_ibuf/O]
set_false_path -from [get_ports PCIE_RESET_N]
```

4.1.4 Creating Constraints files for the ZC706 Development Board

When generating a design for the ZC706 board, the following constraints will correctly constrain the clocks. When using a different board, read the user guide for appropriate pin placment, or copy the constraints from the PCIe Endpoint Example Design. The remaining constraints are contained the generated PCIe IP.

Listing 4.2: .xdc constraints for the ZC706 board

```
set_property IOSTANDARD LVCMOS15 [get_ports PCIE_RESET_N]
set_property PACKAGE_PIN AK23 [get_ports PCIE_RESET_N]
set_property PULLUP true [get_ports PCIE_RESET_N]
# The following constraints are BOARD SPECIFIC. This is for the ZC706
set_property LOC IBUFDS_GTE2_X0Y6 [get_cells refclk_ibuf]
create_clock -period 10.000 -name pcie_refclk [get_pins refclk_ibuf/O]
set_false_path -from [get_ports PCIE_RESET_N]
```

4.2 Ultrascale - Gen3 Integrated Block for PCI Express - (VC709 and newer)

This is a step by step guide for building a RIFFA 2.2.2 reference design for Xilinx FPGA's compatible with the Gen3 Integrated Block for PCI Express. In RIFFA 2.2.2 there are three example designs for the VC709 board in the *RIFFA 2.2.2/source/fpga/vc709* directory: VC709_Gen1x8If64 (PCIe Gen1, 8 lanes, 64-bit CHNL interface), VC709_Gen2x8If128 (PCIe Gen2, 8 lanes, 128-bit CHNL interface), VC709_Gen3x4If128 (PCIe Gen3, 8 lanes, 128-bit CHNL interface). To use one of these example designs, follow the instructions below.

4.2.1 VC709 Example Designs

1. Open Vivado to get the introductory screen shown in Figure 4.1.
2. Click 'Open an Existing Project' and navigate to your RIFFA 2.2.2 directory.
3. In the RIFFA 2.2.2 distribution, open *RIFFA 2.2.2/source/fpga/xilinx/vc709/* and choose from one of the existing example design directories for your board. In the example design directory, locate the *prj* folder and open it. Select the .xpr file and click open. This will open the example project, as shown in Figure 4.11.
4. This project was compiled in Vivado 2014.4. The bit file generated can be used to test the FPGA system. If you are using a newer version of Vivado, recompile the example design or use the programming file provided.
   - IP Settings are now packaged as part of the example designs! Users no longer need to generate IP.
   - To recompile the example design, click the generate bitstream button in the top left corner as shown in Figure 4.11.
   - Recompiling your design will generate a new bitfile in the Xilinx project. The bit file in the *bit* will not be changed.
5. To program the FPGA, click 'Open Hardware Manager'. New bit files (generated by Vivado) will appear in the Vivado generated directories. An example bit file is provided in the example design's *bit*. Load the bitstream to your VC709 board and restart your computer.
   - Before programming your FPGA, you should install the RIFFA driver. See Section 3
6. The example design uses the chnl_tester (shown in Figure 4.3, which works with the example software in the *source/{C_C++,Java,python,matlab}* directories. Replace the chnl_tester instantiation with any user logic, matching the RIFFA interface.
7. Recompile the design and program the FPGA Device. Changing the ***C_NUM_CHNL*** will change the number of independent channel interfaces

4.2.2 Generating the Gen3 Integrated Block for PCI Express

The following steps are not required for general users. See the instructions above for how to compile RIFFA.

Alternatively, it is possible to generate the PCIe Endpoint with different settings than those provided in the example design. Changing the endpoint settings is required when changing the parameters ***C_PCI_DATA_WIDTH***, ***C_MAX_PAYLOAD_BYTES*** and ***C_LOG_NUM_TAGS***. The ***C_NUM_LANES*** is a parameter in the top level file of each example project. How these parameters relate to IP core settings is highlighted in the following figures.

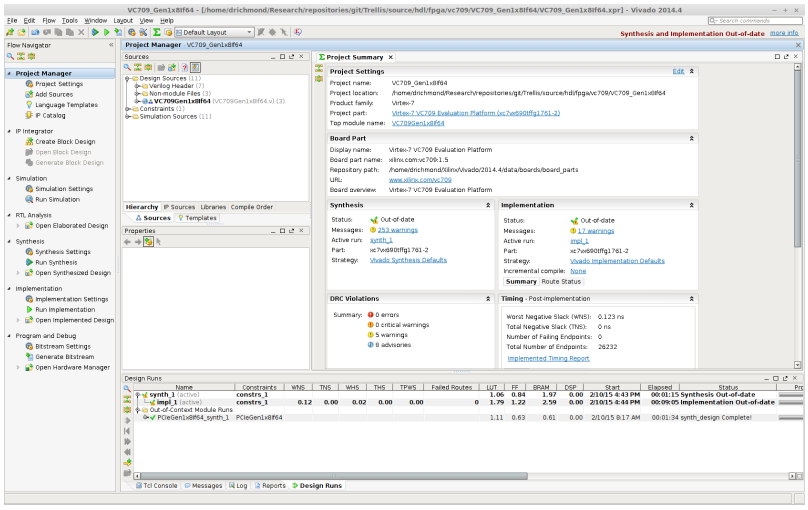

Figure 4.11: Project Splash Screen for Gen3 Integrated Block for PCI Express Projects

If the goal is to generate a RIFFA design completely from scratch, each board directory comes with a RIFFA wrapper verilog file and instantiates a vendor-specific translation layer. It is highly recommended to re-use these files RIFFA wrapper when creating designs from scratch.

To generate the PCIe IP select the 7 Series Integrated Block for PCI Express after selecting the IP Catalog shown in Figure 4.11. This will open the IP Customization window as shown in Figure 4.12

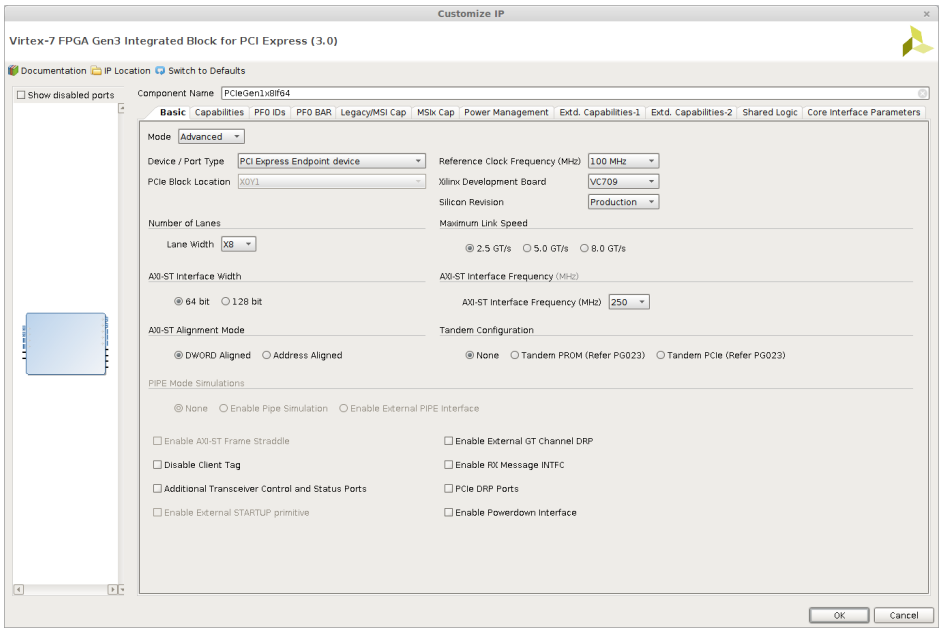

Figure 4.12: Basic settings tab.

First, select “ADVANCED” from the drop down menu. This will cause more tabs to appear in the bar. The following tabs are not used during customization: MSIx Cap (Capabilities), Extd.

Capabilities 1, and Extd Capabilites 2.

In this example, we have set the **Xilinx Development Board** to **VC709**, and selected the PCIe Gen1 rate of **2.5 GT/s**, and a Lane Width of 8 (***C_NUM_LANES*** = 8). We have chosen to set the **AXI Interface Width** to **64-bits** (***C_PCI_DATA_WIDTH*** = 64). Finally Clear the **Disable Client Tag** and **PCIe DRP Ports** boxes. The choice of **Link Rate**, **Lanes**, and **Interface Width** will allow different AXI Interface Frequencies to be selected. The RIFFA core will run at this clock frequency, but the user logic can run at whatever frequency it desires.

Optional: Set the Component Name of the PCI Express block, and the IP Location. In our example projects, we use the name template PCIeGen**W**x**Y**If**Z** where **W** is the PCI Express Version (**Link Speed** in Figure 4.12), **Y** is the lane width, and **Z** is the AXI interface width. The IP location is the *ip* directory in the example project.

Note: For RIFFA 2.2.2 the 256-bit interface is not supported, however the 128-bit interface is. This means PCIe Gen2 with 8 lanes, and PCIe Gen3 with 4 lanes are both supported.

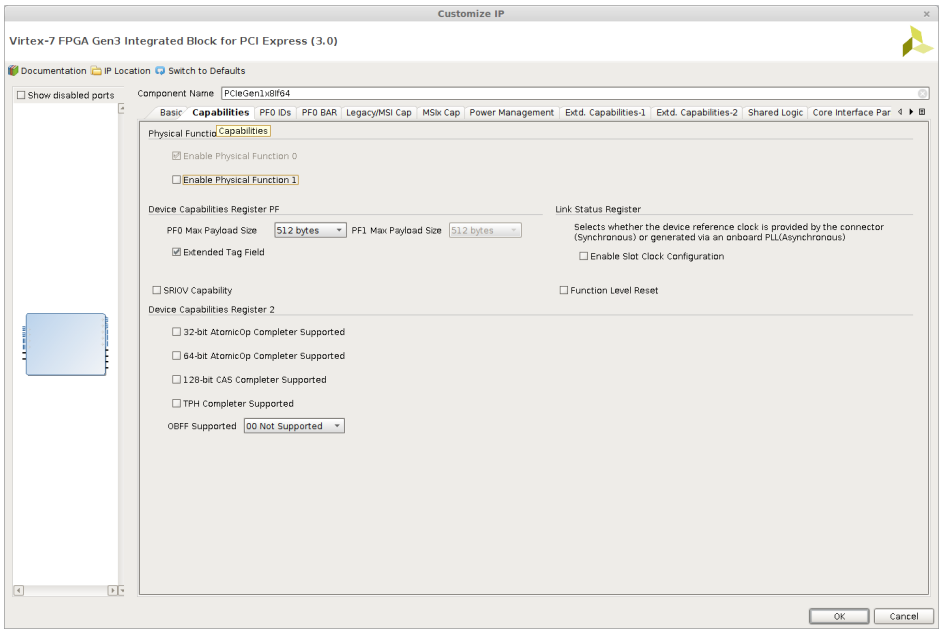

Figure 4.13: PCI Express Capabilities Tab.

In the Capabilities tab shown in Figure 4.13 check the **Extended Tag Field** box. If the **Extended Tag Field** is selected ***C_LOG_NUM_TAGS*** = 8, otherwise ***C_LOG_NUM_TAGS*** = 5. Set the **PFO Max Payload Size** from the dropdown menu; Use this to set the RIFFA ***C_MAX_PAYLOAD_BYTES*** parameter.

Note: Maximum Payload sizes are typically set by the BIOS, and 256 bytes seems to be standard. RIFFA will default to the minimum of ***C_MAX_PAYLOAD_SIZE*** and the setting in your BIOS. Unless your BIOS is modified, or can support substantially larger packets, there will be no performance benefit to increasing the payload size. Increasing the maximum payload size will increase the resources consumed.

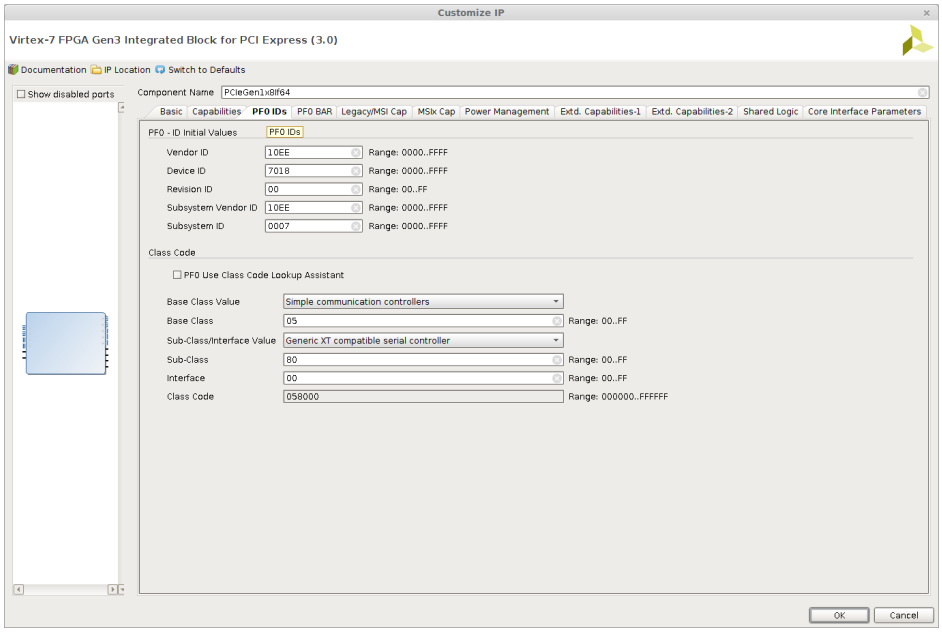

Figure 4.14: PCI Express IDs Tab.

The tab in Figure 4.14 is optional. Setting the Device ID may assist in identifying different FPGAs in a multi-FPGA system. The other options, specifically the Vendor ID, must remain the same.

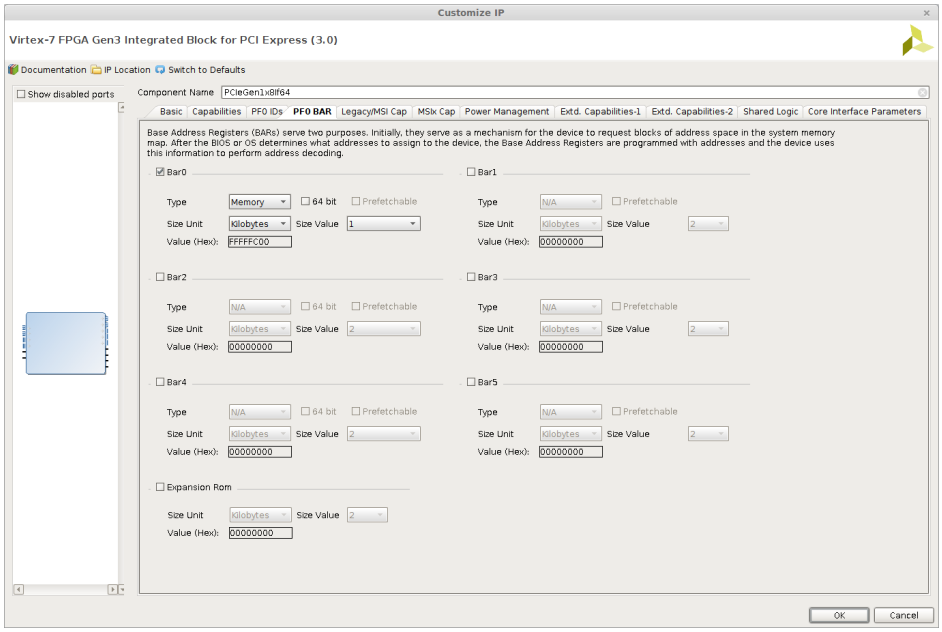

Figure 4.15: PCI Express Base Address Registers (BAR) Tab.

The tab in Figure 4.15 must be configured so that BAR0 is **enabled**. Select type **Memory**, and Unit **Kilobyte**, and **Size Value 1** from the dropdown menus. If these values are not set correctly

the RIFFA driver will not recognize the FPGA device.

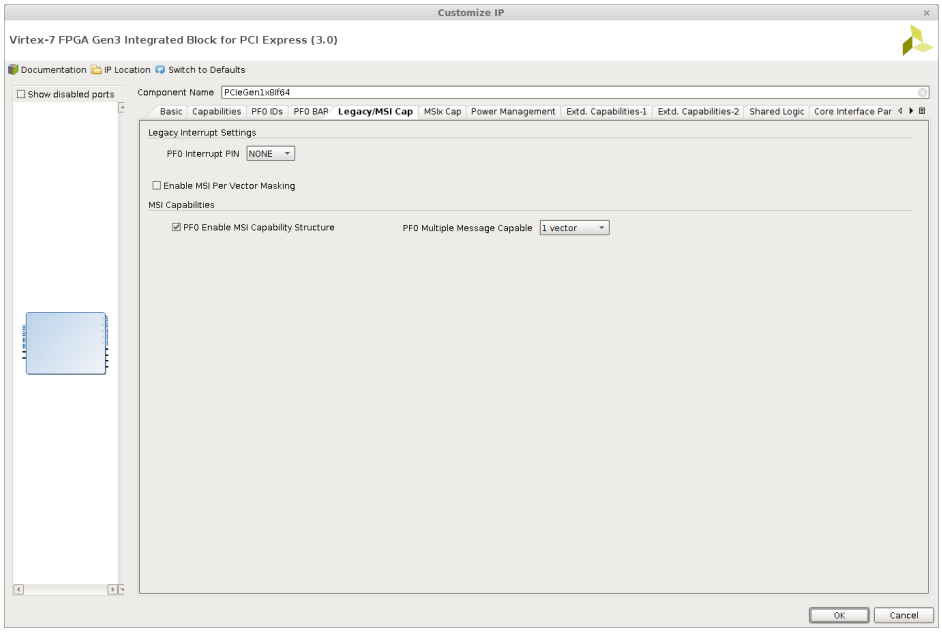

Figure 4.16: PCI Express Legacy and MSI Interrupts Tab.

In the Legacy/MSI Capabilites tab shown in Figure 4.16, select **None** in the **PFO Interrupt Pin Dropdown** menu and set the **PFO Multiple Message Capable** dropdown menu to **1 Vector**

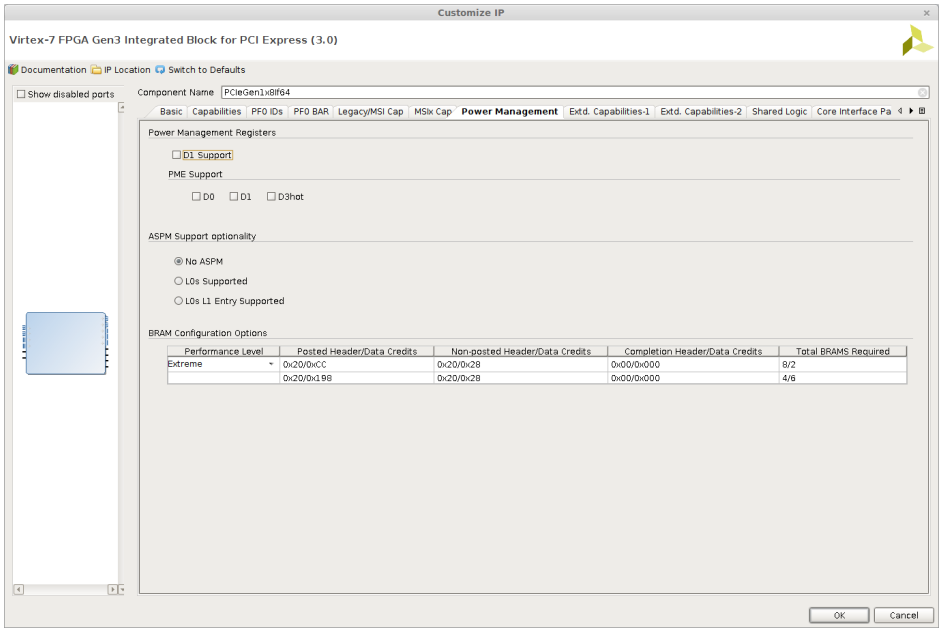

Figure 4.17: PCI Express Power Management Tab.

In the Power Management tab, shown in Figure 4.17, ensure that the **Performance Level** is set to **Extreme**.

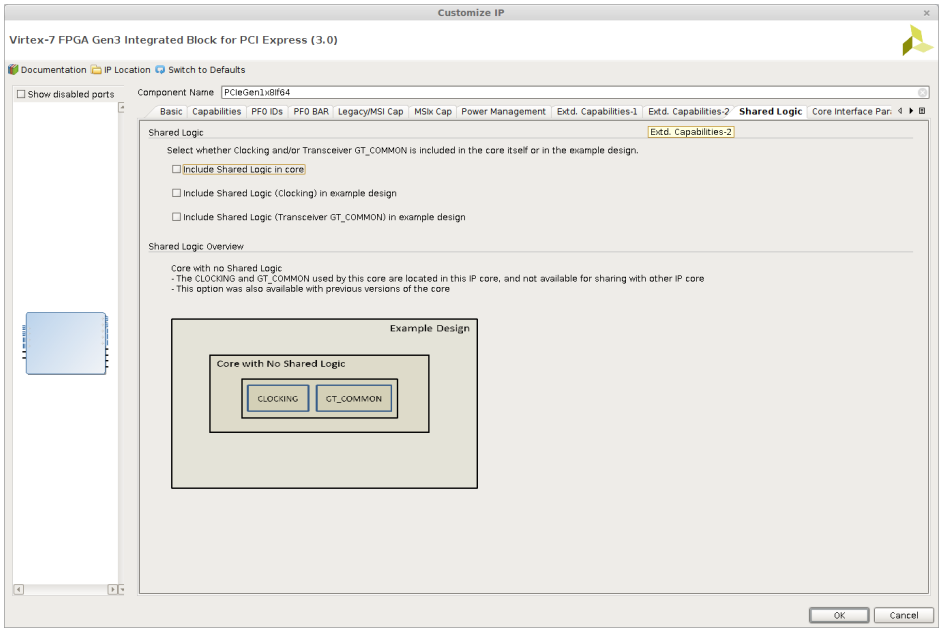

Figure 4.18: PCI Express Shared Logic Tab.

In the Shared Logic Tab shown in Figure 4.18 **clear** all of the checkboxes shown. These settings will not affect the core generated, but will affect the example designs generated by the Vivado, and make the Vivado example design mirror the RIFFA Example design.

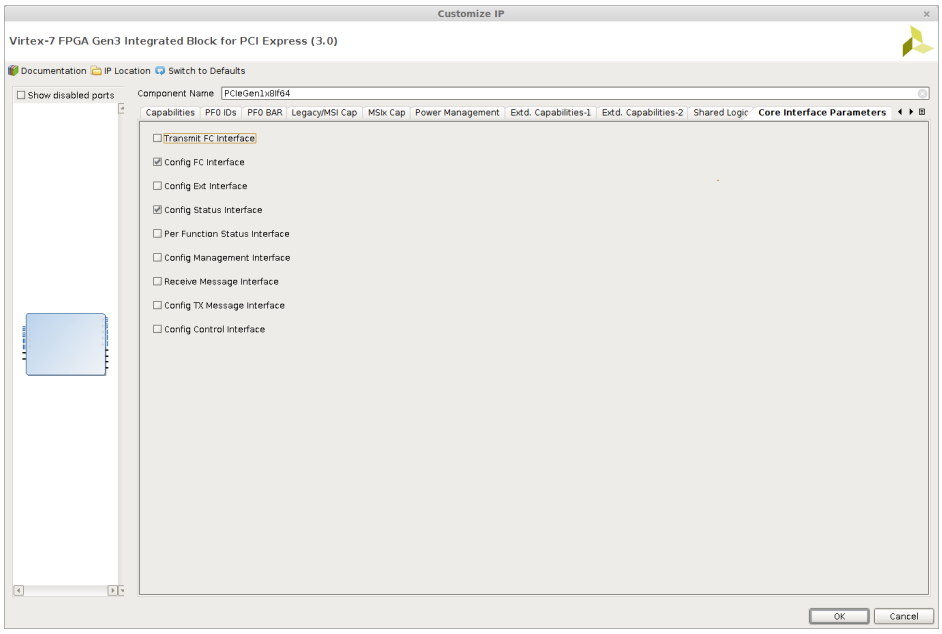

Figure 4.19: PCI Express Core Interface Parameters Tab.

Finally, in the Interface Parameters tab, match the checkboxes shown in Figure 4.19. These options simplify the interface to the generated core

4.2.3 Creating Constraints files for the VC709 Development Board

When generating a design for the VC709 board, the following constraints will correctly constrain the clocks. When using a different board, read the user guide for appropriate pin placment, or copy the constraints from the PCIe Endpoint Example Design.

Listing 4.3: .xdc constraints for the VC709 board

```
create_clock -period 10.000 -name pcie_refclk [get_pins refclk_ibuf/O]
set_false_path -from [get_ports PCIE_RESET_N]

# The following constraints are BOARD SPECIFIC. This is for the VC709
set_property LOC IBUFDS_GTE2_X1Y11 [get_cells refclk_ibuf]
set_property PACKAGE_PIN AV35 [get_ports PCIE_RESET_N]
set_property IOSTANDARD LVCMOS18 [get_ports PCIE_RESET_N]
set_property PULLUP true [get_ports PCIE_RESET_N]
```

5 Compiling and using the Altera Example Designs

This section describes how to use RIFFA 2.2.2 with Quartus 14.1. The example projects included in this distribution target Terasic DE5Net and DE4 boards. We are confident that RIFFA will work on all currently supported Altera devices using the Hard IP for PCI Express (Cyclone V, Arria V and Stratix V) devices, as well as all devices using IP Compiler for PCI Express (Stratix IV and prior). For device support in Quartus 14.1see [1]

The FPGA families that we have successfully tested RIFFA 2.2.2 are:

- Stratix V (DE5-Net)
- Stratix IV (DE4)

There are three options for starting a new RIFFA project:

- For first-time users with a DE5 board, we recommend the archived projects provided in the *RIFFA 2.2.2/source/fpga/de5_qsys* directory. Follow the instructions in Section 5.1.1
- Intermediate and advanced users, or users with a DE4 board, we have provided projects without instantiated IP. For DE5 boards, follow the instructions in Section 5.1.2. For DE4 boards, follow the instructions in Section 5.2
- For advanced users, or users wishing to support a new board, we provide full instructions for creating a top level and generating IP. Follow the instructions in Section 5.1

5.1 Example Designs with Qsys and MegaWizard (Stratix V, Cyclone V and newer)

5.1.1 Qsys (Stratix V and newer)

For first-time users with the DE5-Net board, copy one of the archived projects (.qar files) available in the *de5_qsys* directory.

1. Open Quartus to get the introductory screen shown in Figure 5.1.
2. Click 'Open an Existing Project' and navigate to your RIFFA 2.2.2 directory.
3. In the RIFFA 2.2.2 distribution, open *RIFFA 2.2.2/source/fpga/de4/* and choose from one of the existing example design directories for your board. In the example design directory, locate the *prj* folder and open it. Select the .qpf file and click open. This will open the example project, as shown in Figure 5.2.
4. This project was compiled in Quartus 14.1. The bit file generated can be used to test the FPGA system. If you are using a newer version of Quartus, recompile the example design or use the programming file provided.
    - To recompile the example design, click the compile button in the top left corner as shown in Figure 5.2.

[1] http://dl.altera.com/devices/

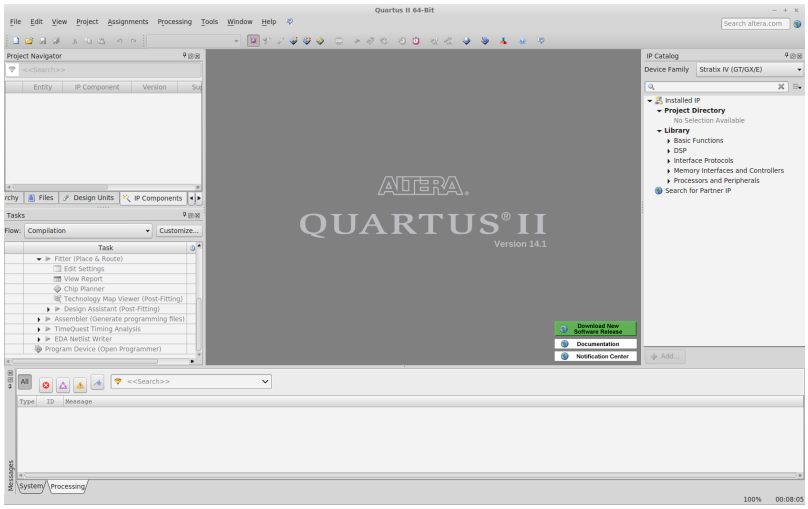

Figure 5.1: Welcome Screen for Quartus 14.1

- Recompiling your design will generate a new bitfile in the *prj* directory. The bit file in the *bit* will not be changed.

5. To program the FPGA, click 'Open Programmer'. New bit files (generated by Quartus) will appear in the *prj/output_files/* directory. An example bit file is provided in the example design's *bit* directory.

   - Before programming your FPGA, you should install the RIFFA driver. See Section 3

6. The example design uses the chnl_tester (shown in Figure 5.3, which works with the example software in the *source/{C_C++,Java,python,matlab}* directories. Replace the chnl_tester instantiation with any user logic, matching the RIFFA interface.

7. Recompile the design and program the FPGA Device. Changing the ***C_NUM_CHNL*** will change the number of independent channel interfaces

5.1.2 Generating IP using MegaWizard (Stratix V, Cyclone V and newer)

In some cases, it may be necessary to generate the PCIe Endpoint IP. For intermediate users, there are project example projects inside of the *de5* directory without instantiated IP (This is done to avoid licensing problems). For the DE5, the project directories are: DE5Gen1x8If64, DE5Gen2x8If128, DE5Gen3x4If128.

Modifying the RIFFA parameters ***C_PCI_DATA_WIDTH***, ***C_MAX_PAYLOAD_BYTES*** and ***C_LOG_NUM_TAGS*** require changing certain settings in the IP core file. The paramter ***C_NUM_LANES*** is located in the top level file of each example project. How these parameters relate to IP core settings is highlighted in the following figures.

For advanced users whose goal is to generate a RIFFA design completely from scratch, we provide instructions for generating the timing constraints and other low level details. Each board directory contains with a RIFFA wrapper verilog file and instantiates a vendor-specific translation layer. It is highly recommended to re-use these files RIFFA wrapper when creating designs from scratch. Users should also use the constraints file (.sdc) in the board directory, and in the *constr/*, or read the User Guide provided with each board and the instructions for generating constratints in Section 5.1.3.

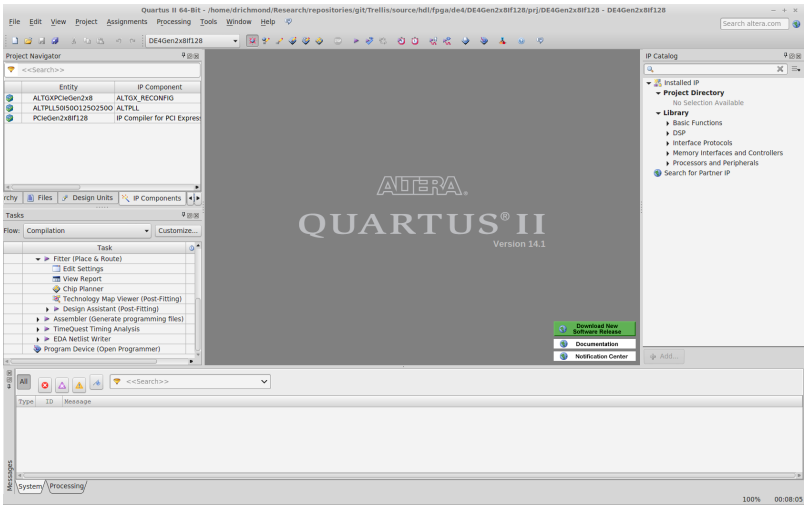

Figure 5.2: Project Splash Screen for Quartus Projects

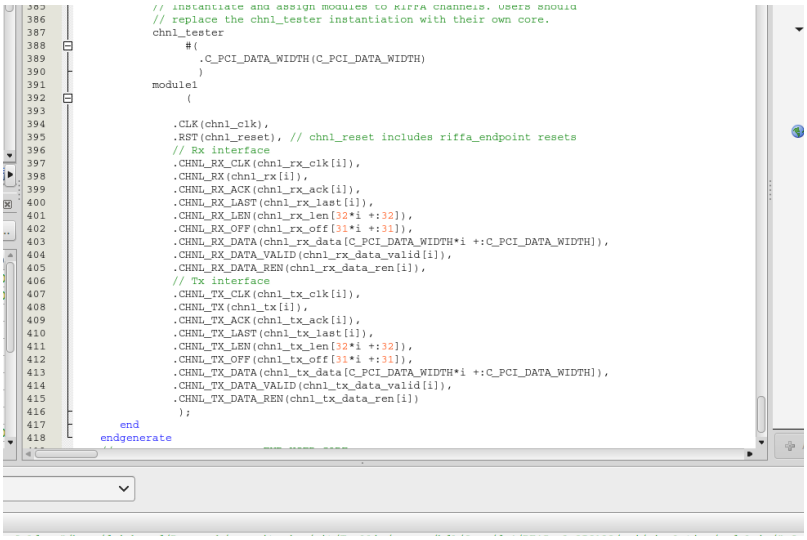

Figure 5.3: chnl_tester instantiation in the top level file

As stated in Section 2.3, each project directory contains five folders.

- The *prj/* directory contains the project .qpf and .qsf file.
- The *hdl/* contains the top level file, e.g. DE5Gen2x8If128.v, which instantiates the skeleton IP and the RIFFA Core.
- The *ip/* directory is empty but will contain Altera IP generated by Quartus in the following guide.
- The *constr/* directory contains project-specific timing constraint files.
- Finally the *bit/* directory contains the project .sof, or bit file that we have tested. This bitfile will not be overwritten by subsequent Quartus compilations.

Note: The bitfile in the bit directory is not modified by recompilation in Quartus. Quartus will generate a new bitfile (.sof) in the *prj/* directory for the DE5Net board.

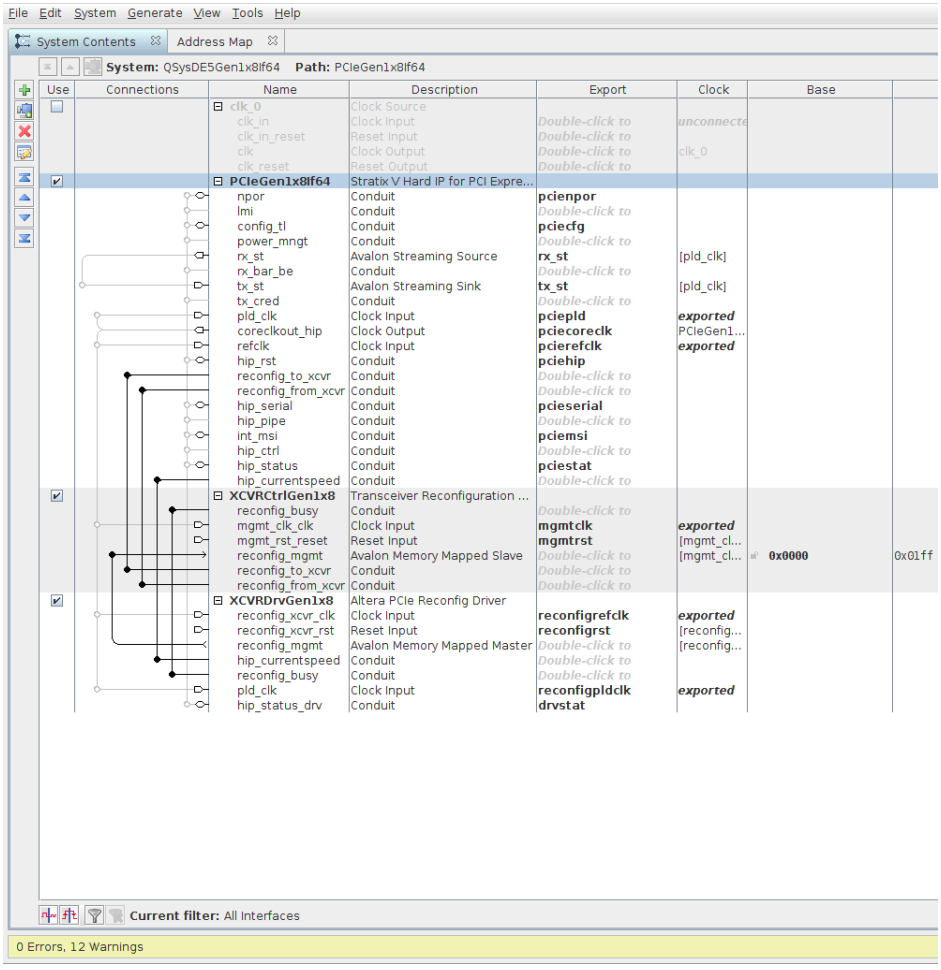

Figure 5.4: Qsys Diagram depicting the connections between the three Altera IP blocks.

Altera designs require additional IP to drive the PCIe Core Transcievers. For the DE5, these blocks are the Transciever Reconfiguration Controller and the Reconfiguration Driver. When

creating a new top level design, these blocks must be connected together with the PCIe Endpoint as shown in Figure 5.4.

First, we will generate the PCIe Endpoint. Click on the Avalon Streaming Interface for PCI Express in the Quartus IP Catalog. Figure 5.9.

Optional: Set the Component Name of the PCI Express block, and the IP Location. In our example projects, we typically use the name PCIeGen**W**x**Y**If**Z** where **W** is the PCI Express Version (**Link Speed** in Figure 4.12), **Y** is the lane width, and **Z** is the Avalon interface width. The IP location is the *ip/* directory in the example project.

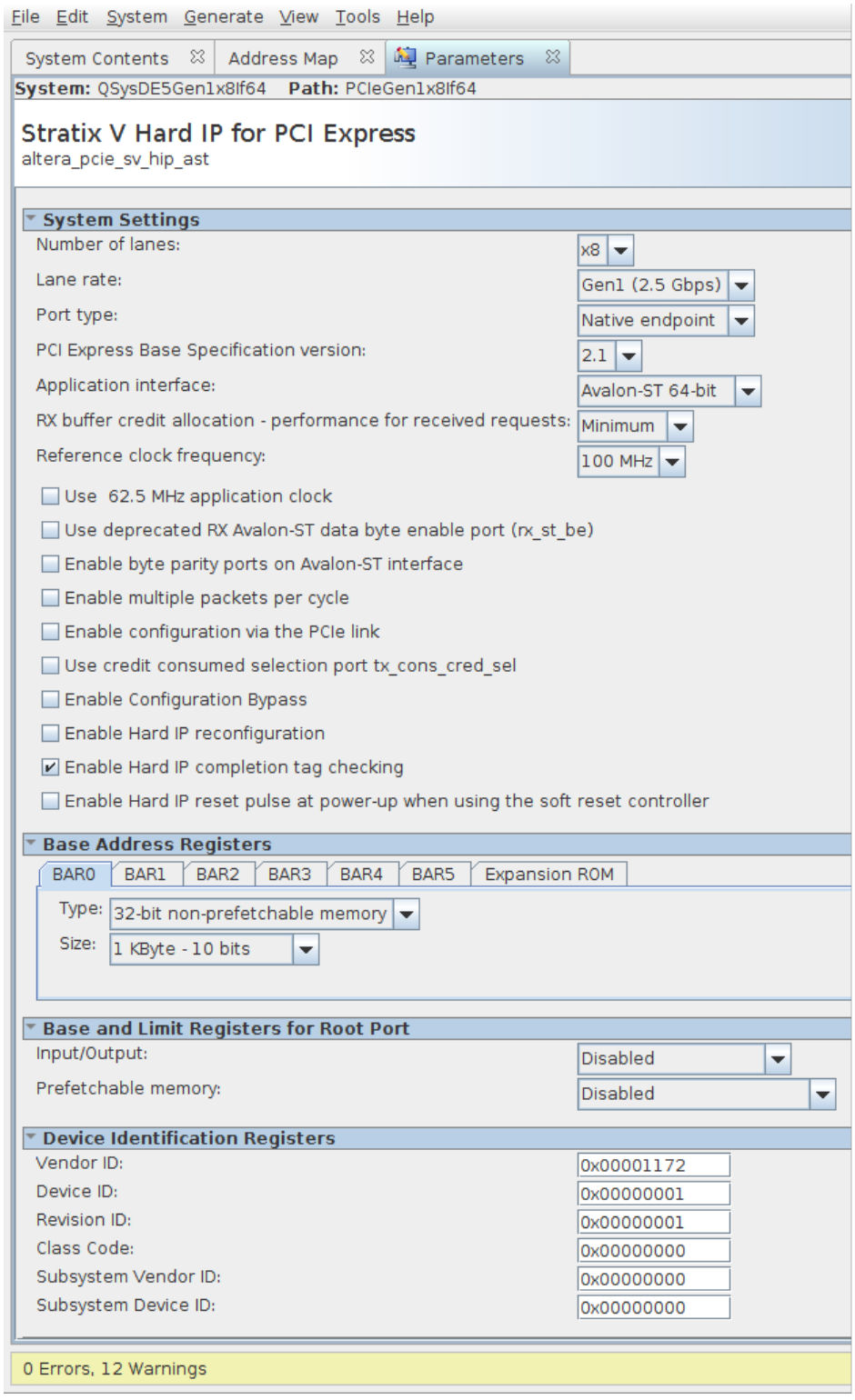

Figure 5.5: PCI Express Endpoint Configuration Menu

In Figure 5.5, select the **Number of Lanes**, which corresponds to the top level parameter ***C_NUM_LANES***, **Lane Rate**, and **PCI Express Base Specification** version from the drop-down menus (Choose the highest possible base specification version). Select an **Application Interface Width**; This corresponds to the ***C_PCI_DATA_WIDTH*** parameter in RIFFA. Currently the **64-bit** and **128-bit** interfaces are supported for all Altera designs. Some widths may not be possible depending on the **Lane Rate** and **Number of Lanes** selected.

The choice of **Link Rate**, **Number of Lanes**, and **Interface Width** will set the frequency for the PCI interface, which is clocked by the pld_clk signal. For the chosen settings, the frequency should be displayed in the messages bar at the bottom of the configuration menu (Messages bar not shown). The RIFFA core will run at this clock frequency, but the user logic can run at whatever frequency it desires.

In the Base Address Registers Section set BAR0's type to **32-bit non-prefetchable memory** and set the size to **1 KByte - 10 Bits**.

There are no required changes in the Device Identification Registers Section. However, in a multiple FPGA system, it may be useful to change the **Device ID** to allow identification of different FPGA platforms. The other options, specifically the **Vendor ID**, must remain the same.

Scroll down to view the final two sections shown in Figure 5.6.

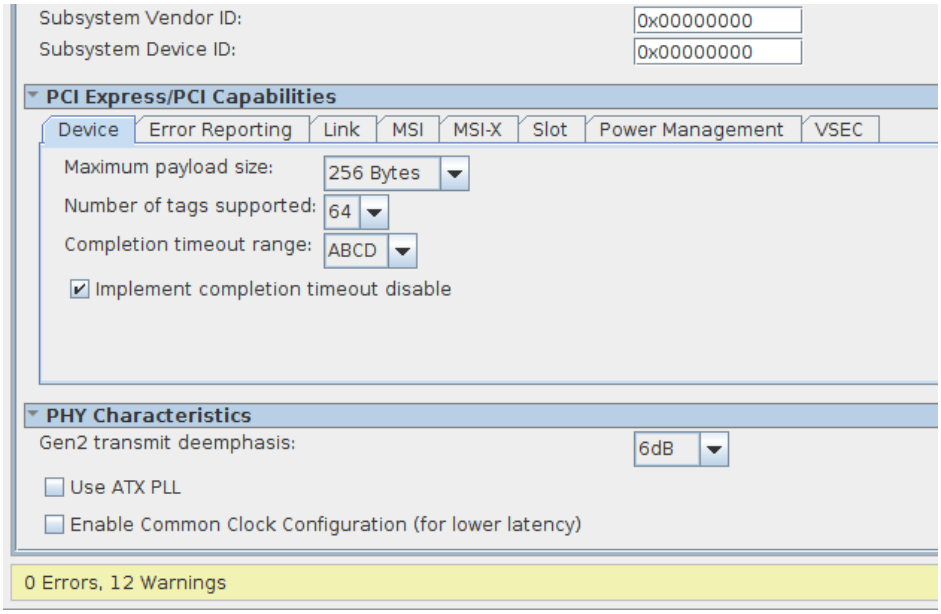

Figure 5.6: PCI Express Endpoint Configuration Menu

In the PCI Express/PCI Capabilities menu, set your desired **Maximum Payload Size**, which corresponds to the RIFFA parameter, ***C_MAX_PAYLOAD_BYTES*** and the **Number of Tags Supported**. The log of the **Number of Tags Supported** is the ***C_LOG_NUM_TAGS*** parameter in RIFFA.

In the MSI Tab, make sure that the number of MSI messages requested is equal to 1.

Note: Maximum Payload sizes are typically set by the BIOS, and 256 bytes seems to be standard. RIFFA will default to the minimum setting ***C_MAX_PAYLOAD_SIZE*** and the setting in your

BIOS. Unless your BIOS is modified, or can support substantially larger packets, there will be no performance benefit to increasing the payload size. Increasing the **Maximum Payload Size** will increase the resources consumed.

Finally, record the number of **Transciever Reconfiguration Interfaces** in the messages bar at the bottom of the screen, then close the PCIe IP Generation Menu. A window may ask if you wish to generate the example design. This is optional.

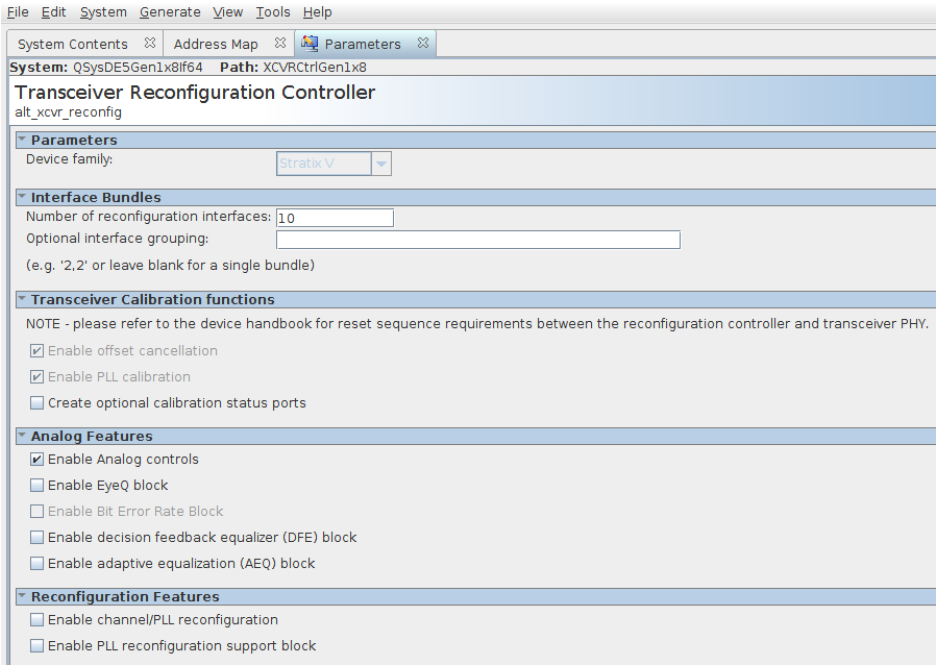

Figure 5.7: Transciever Reconfiguration IP Generation Menu

Next, generate the Transceiver Reconfiguration Controller by opening MegaWizard and selecting the Transceiver Reconfiguration Controller megafunction.

Set the appropriate number of **Transciever Reconfiguration Interfaces** in the Interface Bundles Menu. In the analog features section, **Enable Analog Controls** and **Enable Adaptive Equalization** block by clicking the appropriate boxes.

Optional: Set the Component Name of the Transciever Reconfiguration Controller, and the IP Location. In our example projects, we typically use the name XCVRCtrlGen**W**x**Y** where **W** is the **PCI Express Version** (**Link Speed** in Figure 5.5), **Y** is the lane width. The IP location is the *ip/* directory in the example project.

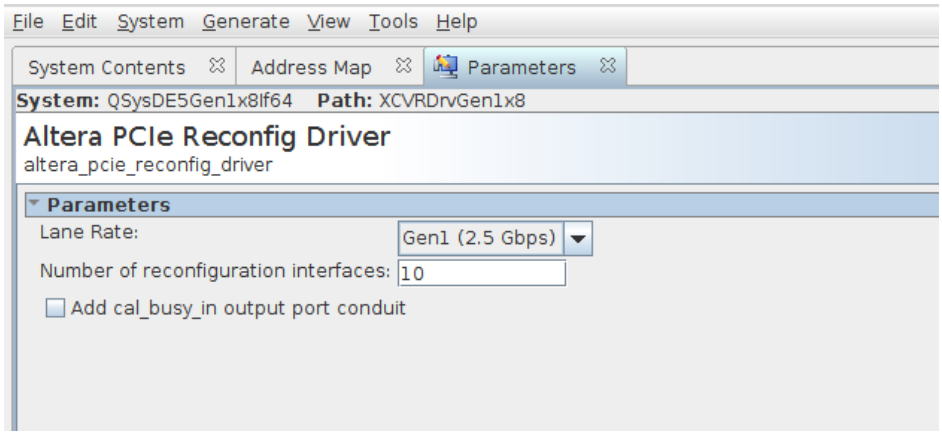

Figure 5.8: Transciever Reconfiguration Driver Menu

In Qsys, generate the Transciver Reconfiguration Controller. Select the appropriate lane rate for your design, and the number of Reconfiguration Intefaces. These should match the number of reconfiguration interfaces dictated when generating the PCIe IP in Figure 5.5 and the number selected in Figure 5.7.

Optional (in Qsys): Set the Component Name of the Transciever Reconfiguration Driver, and the IP Location. In our example projects, we typically use the name XCVRDrvGen**W**x**Y** where **W** is the **PCI Express Version** (**Link Speed** in Figure 4.12), and **Y** is the lane width. The IP location is the *ip* directory in the example project.

If you are using Megawizard to instantiate IP, you must manually instantiate the Transciver Reconfiguration Driver in the Top Level design. The instantiation template is shown in Listing 5.1 into your top-level file. Match the PCIe generation, and chip generation for your project.

Listing 5.1: Manual (non-qsys) instantiation of Reconfiguration Driver

```
altpcie_reconfig_driver
#(
/* These values should match the values used for the PCIe Endpoint */
.number_of_reconfig_interfaces(10), /*Set This*/
.gen123_lane_rate_mode_hwtcl(''Gen1 (2.5 Gbps)''), /*Set This*/
.INTENDED_DEVICE_FAMILY(''Stratix V'')) /*Set This*/
XCVRDriverGen2x8_inst
(
/*Ports Here -- Copy from Example Designs*/
);
```

5.1.3 Creating Constraints files for MegaWizard and QSys Designs

Advanced users may also want to edit and modify the constraint files. This not required or recommended for novice users. The example designs in the RIFFA 2.2.2 distribution contain appropriate constraint files for the example designs. However if the need arises, these constraints are documented below.

To appropriately constrain your PCIe reference clocks, place the constraints shown in Listing 5.2 in your .sdc file. Modify the names PCIE_REFCLK, PCIE_TX_OUT and PCIE_RX_IN to match your design.

Listing 5.2: .sdc constraints for Qsys and Megawizard designs

```
create_clock -name PCIE_REFCLK -period 10.000 [get_ports {PCIE_REFCLK}]
derive_pll_clocks -create_base_clocks
derive_clock_uncertainty
```

Likewise, copy the constraints in Listing 5.3 into your .qsf file. Copy the location assignment commands for each PCIe Pin in your reference design.

Listing 5.3: .qsf settings for Qsys and Megawizard designs

```
#============================================================
# PCIE Connections
#============================================================
set_location_assignment <PCIE_REFCLK_PIN> -to PCIE_REFCLK
set_instance_assignment -name IO_STANDARD HCSL -to PCIE_REFCLK
set_location_assignment <PCIE_REFCLK_PIN(n)> -to ''PCIE_REFCLK(n)''
set_instance_assignment -name IO_STANDARD HCSL -to ''PCIE_REFCLK(n)''
set_location_assignment <PCIE_RESET_N> -to PCIE_RESET_N
set_instance_assignment -name IO_STANDARD ''2.5 V'' -to PCIE_RESET_N

# For each PCIE Lane (L) set the pin locations from the board user guide!
#============================================================
#PCIE TX_OUT L
#============================================================
set_location_assignment <TX_LANE[L]_PIN>   -to PCIE_TX_OUT[0]
set_location_assignment <TX_LANE[L]_PIN(n)> -to ''PCIE_TX_OUT[0](n)''

#============================================================
#PCIE RX_IN L
#============================================================
set_location_assignment <RX_LANE[L]_PIN>  -to PCIE_RX_IN[L]
set_location_assignment <RX_LANE[L]_PIN(n)> -to ''PCIE_RX_IN[L](n)''
```

5.2 IP Compiler for PCI Express (Stratix IV, and older)

To avoid licensing problems, we do not package Altera IP for the DE4 board. Manual IP Instantiation is required when using the DE4 Board, and similar devices using the IP Compiler for PCI Express. Changing the endpoint settings described here may change the RIFFA parameters ***C_PCI_DATA_WIDTH***, ***C_MAX_PAYLOAD_BYTES*** and ***C_LOG_NUM_TAGS***. How these parameters relate to IP core settings is highlighted in the following figures.

There are sever example projects inside of the *de4* directory folder without instantiated IP. For the DE4 board, these projects are DE4Gen1x8If64, DE4Gen2x8If128.board.

As stated in Section 2.3, each project directory contains five folders.

- The *prj/* directory contains the project .qpf and .qsf file.
- The *hdl/* contains the top level file, e.g. DE5Gen2x8If128.v, which instantiates the skeleton IP and the RIFFA Core.
- The *ip/* directory is empty but will contain Altera IP generated by Quartus in the following guide.
- The *constr/* directory contains project-specific timing constraint files.
- Finally the *bit/* directory contains the project .sof, or bit file that we have tested. This bitfile will not be overwritten by subsequent Quartus compilations.

5.2.1 Generating IP with IP Compiler for PCI Express (Stratix IV, and older)

Note: The bitfile in the bit directory is not modified by recompilation in Quartus. Quartus will generate a new bitfile (.sof) in the */output_files* directory for the DE4 board.

First, we will generate the PCIe Endpoint. Open the Altera IP Catalog and select the IP Compiler for PCI Express. This will open the window shown in Figure 5.9.

Optional: Set the Component Name of the PCI Express block, and the IP Location. In our example projects, we typically use the name PCIeGen**W**x**Y**If**Z** where **W** is the PCI Express Version (Link Speed in Figure 4.12), **Y** is the lane width, and **Z** is the Avalon interface width. The IP location is the *ip* directory in the example project.

In this guide, we will skip the Power Mangement tab shown in Figure 5.9.

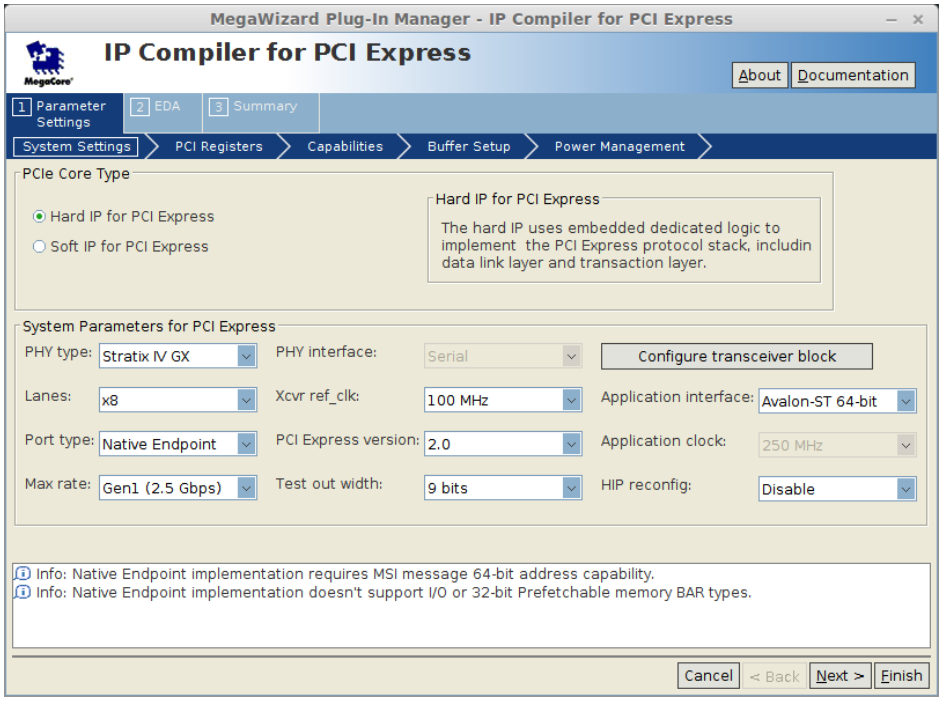

Figure 5.9: IP Compiler for PCI Express System Settings Tab

In the first column of the System Settings Tab, select your **Chip Generation/PHY Type** (**Stratix IV GX** for the DE4 board), **Lanes**, and **Max Rate**. The **Number of Lanes** is the parameter ***C_NUM_LANES*** in the project top level file. In the second column, select the **PCI Express Version** (2.0, or the highest possible) and set the **Test Out Width** to **0**. In the third column, select the **Application Interface Width**. The **Application Interface Width** corresponds to the RIFFA parameter ***C_PCI_DATA_WIDTH***.

The choice of **Link Rate**, **Lanes**, and **Interface Width** will set the frequency for the PCI interface, which is clocked by the signal pld_clk. For the chosen settings, the frequency is determined in the Chip User Guide (though, typically it is one of 62.5, 125, or 256 MHz)

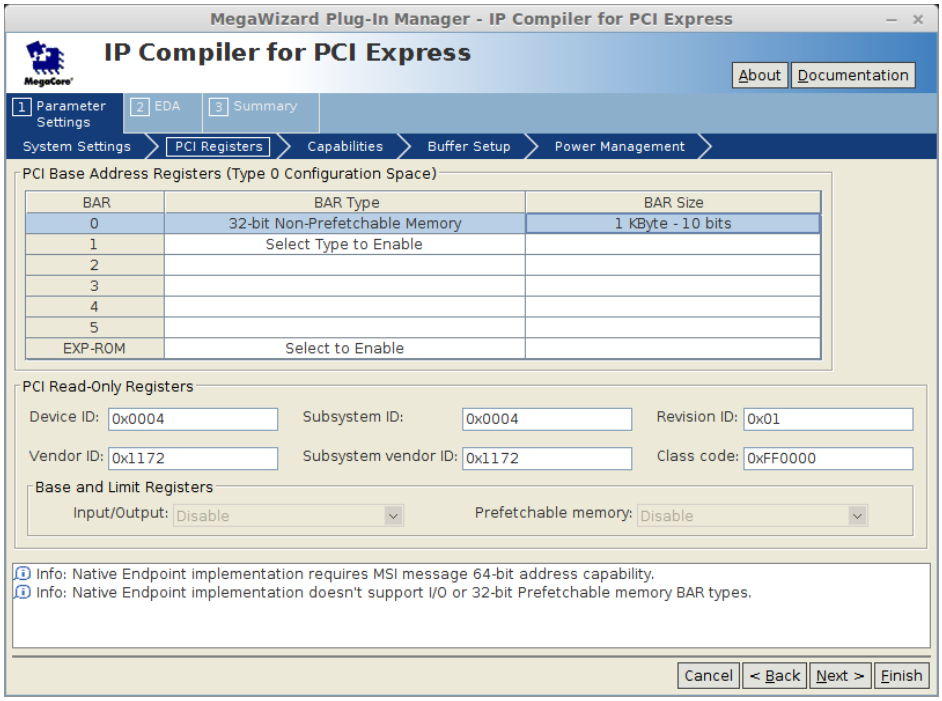

Figure 5.10: IP Compiler for PCI Express Registers Tab

PCI Registers Tab, shown in Figure 5.10, set BAR0's type to "32-bit non-prefetchable memory". Set the size to "1 KByte - 10 Bits".

There are no required changes in the PCI Registers Tab. However, in a multiple FPGA system, it may be useful to change the **Device ID** to identify different FPGA platforms. The other options, specifically the **Vendor ID**, must remain the same.

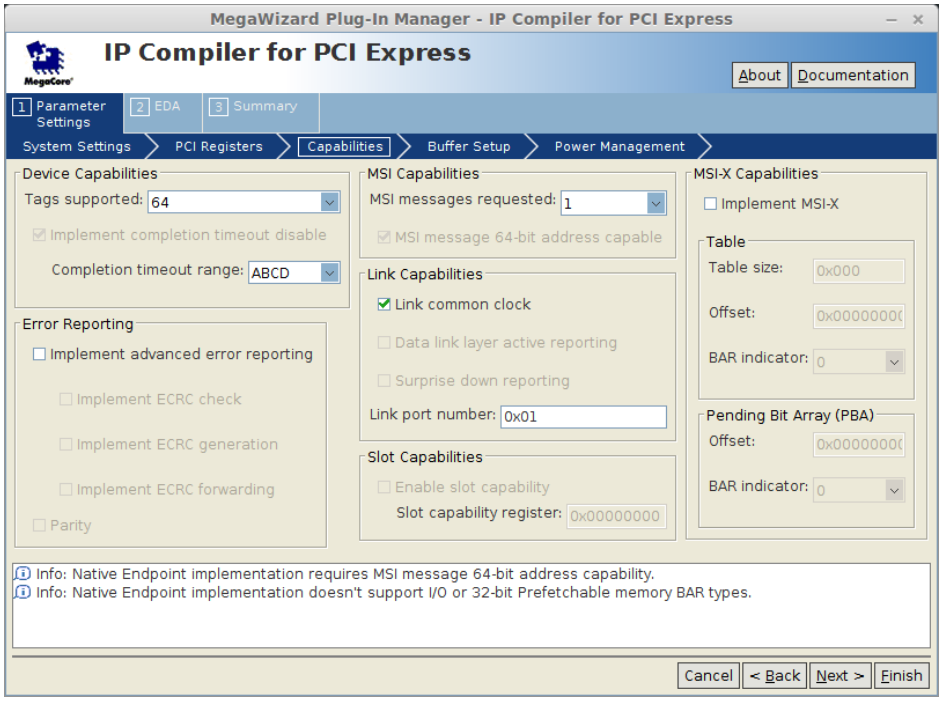

Figure 5.11: IP Compiler for PCI Express Capabilities Tab

Open the Capabilities Tab shown in Figure 5.11. In the Device Capabilities box, set the **Tags Supported** to **64**. The log of the maximum number of tags supported is the RIFFA parameter ***C_LOG_NUM_TAGS*** parameter in RIFFA. In the MSI Capabilities box, set the number of **MSI Messages Requested** to **1**. All the remaining settings must stay the same.

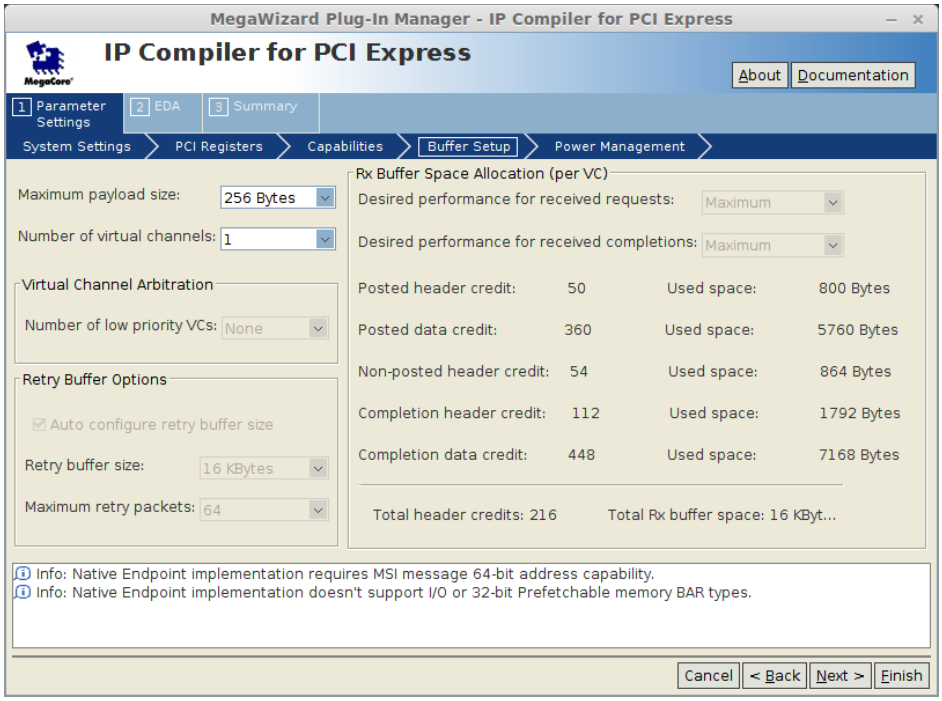

Figure 5.12: IP Compiler for PCI Express Buffer Setup Tab

In Figure 5.12 select the **Maximum Payload Size** from the dropdown menu. Use this to set the ***C_MAX_PAYLOAD*** parameter. Set the numer of **Virtual Channels** to **1**.

Note: **Maximum Payload** sizes are typically set by the BIOS, and 256 bytes seems to be standard. RIFFA will default to the minimum of ***C_MAX_PAYLOAD_SIZE*** and the setting in your BIOS. Unless your BIOS is modified, or can support substantially larger packets, there will be no performance benefit to increasing the payload size. Increasing the **Maximum Payload** size will increase the resources consumed.

Next, we need to generate the PLL for the example design. Select the ALTPLL megafunction from the Quartus IP Catalog, to open the window shown in Figure 5.13.

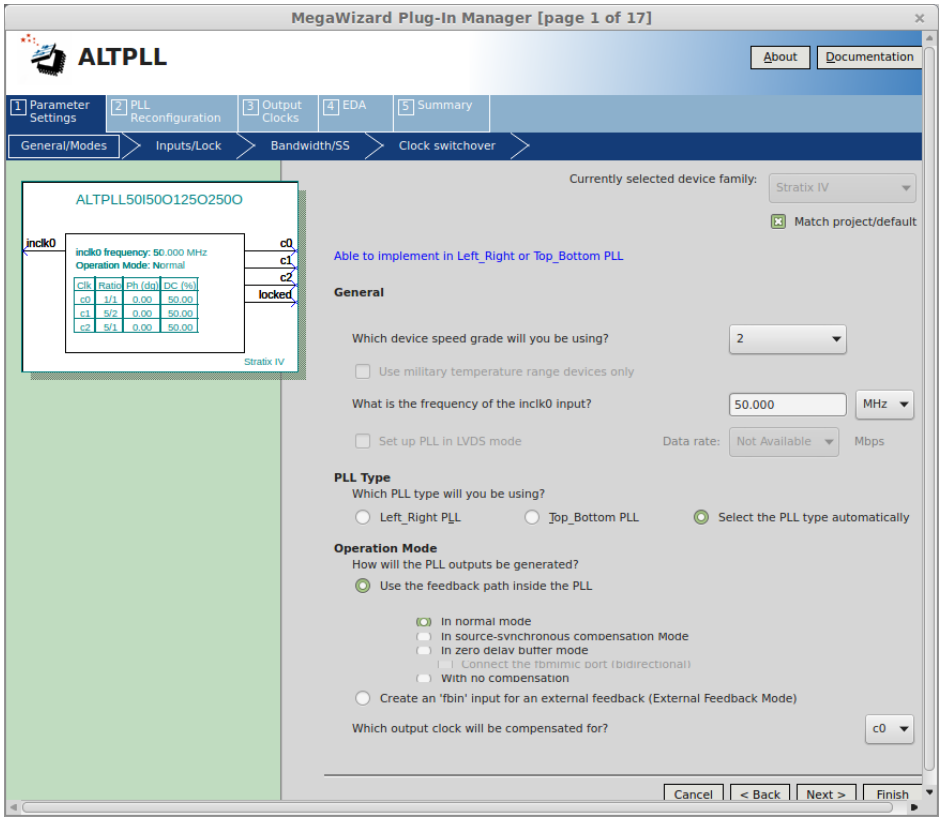

Figure 5.13: ALTPLL General Settings Tab

In Figure 5.13, select the **Speed Grade** that matches your board (Found in the User Guide and online). Next set the input clock frequency. The DE4 board provides 50 MHz clock inputs and we use these for convenience. The remaining settings are unchaged. Click on the Inputs/Lock tab to move on to Figure 5.14.

Optional: Set the name of the ALTPLL block. In the example designs we use the name ALTPLL50I50O125O250O, for 50 MHz Input clock, 50, 125, and 250 MHz output clocks. The 125 and 50 MHz Clocks are required for the PCIe Endpoint.

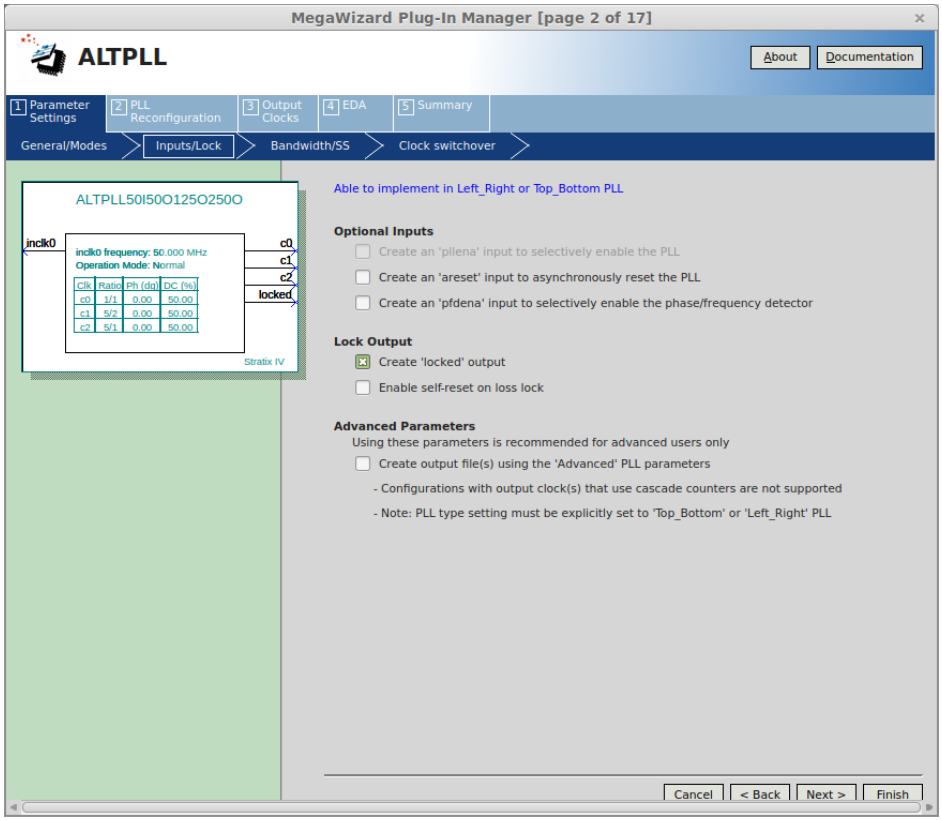

Figure 5.14: ALTPLL Input Settings Tab

Match the settings shown in Figure 5.14. In the Output Clocks Section, create a 50 MHz output clock, 125 MHz clock, and 250 MHz Clock. Click Finish when done.

Finally, we generate the ALTGX_RECONFIG Megafunction. Select the ALTGX_RECONFIG megafunction from the Quartus IP Catalog to produce the widown shown in Figure 5.15.

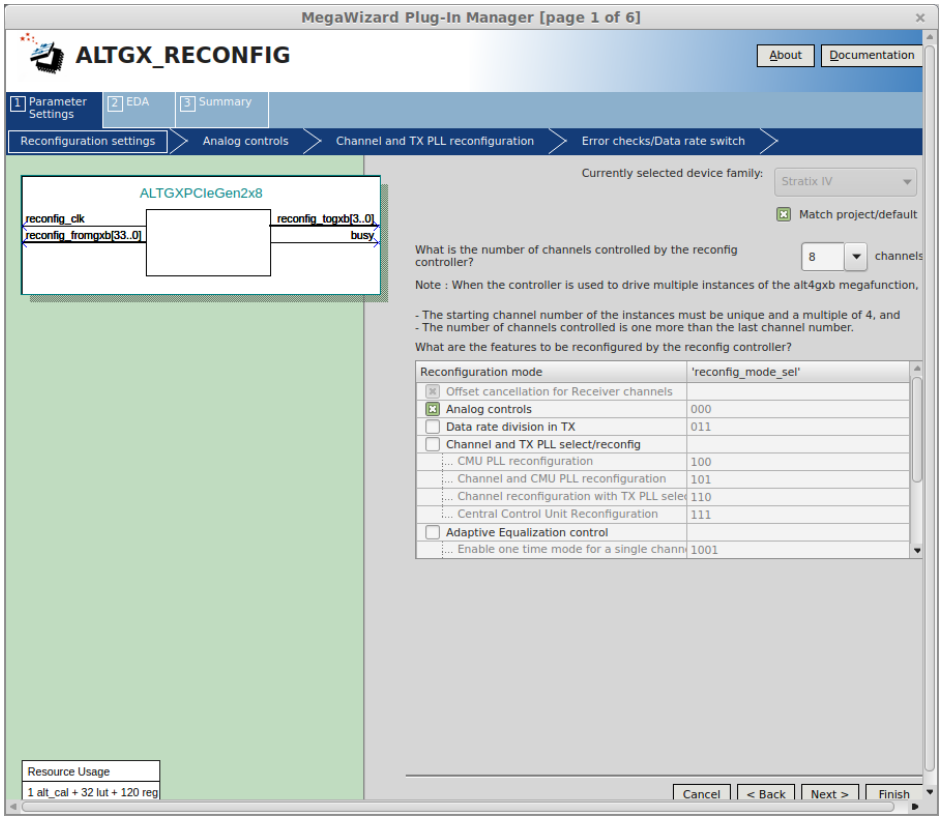

Figure 5.15: ALTGX Reconfiguration Settings Tab

In the Reconfiguration Tab shown in Figure 5.15, set the **Number of Channels**. This should be equal to the number of PCIe Lanes at the Top Level. In the Features Section, **Enable Analog Controls**. Match the settings in the remaining windows, shown in Figure 5.16,Figure 5.17, and Figure 5.18.

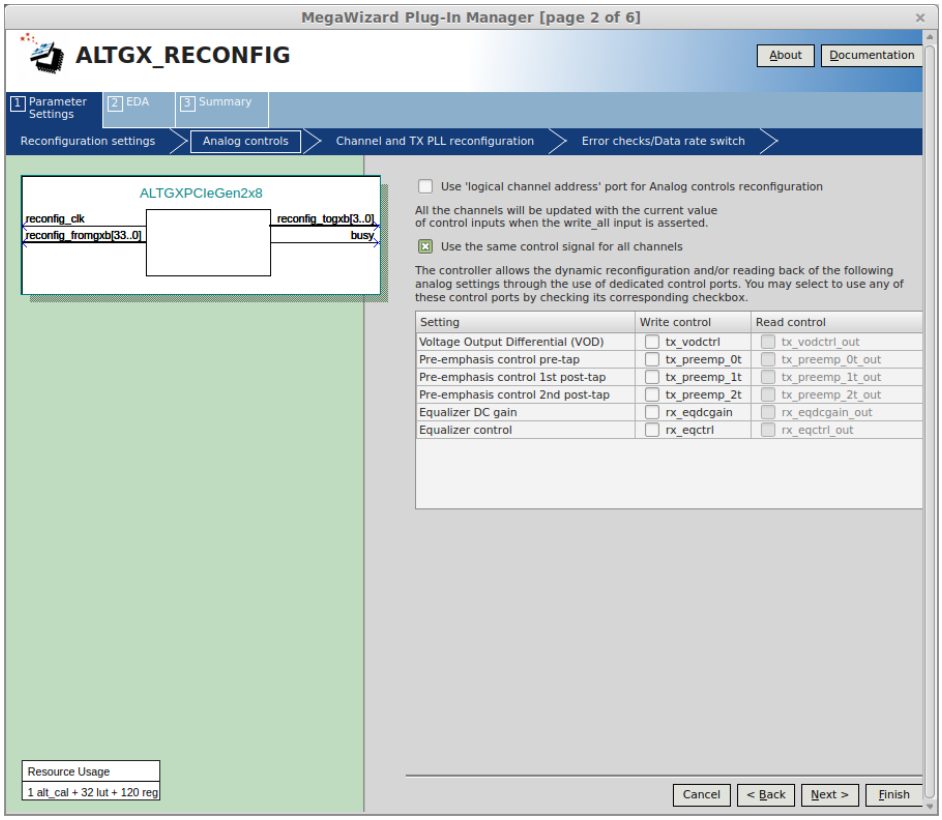

Figure 5.16: ALTGX Reconfiguration Analog Settings Tab

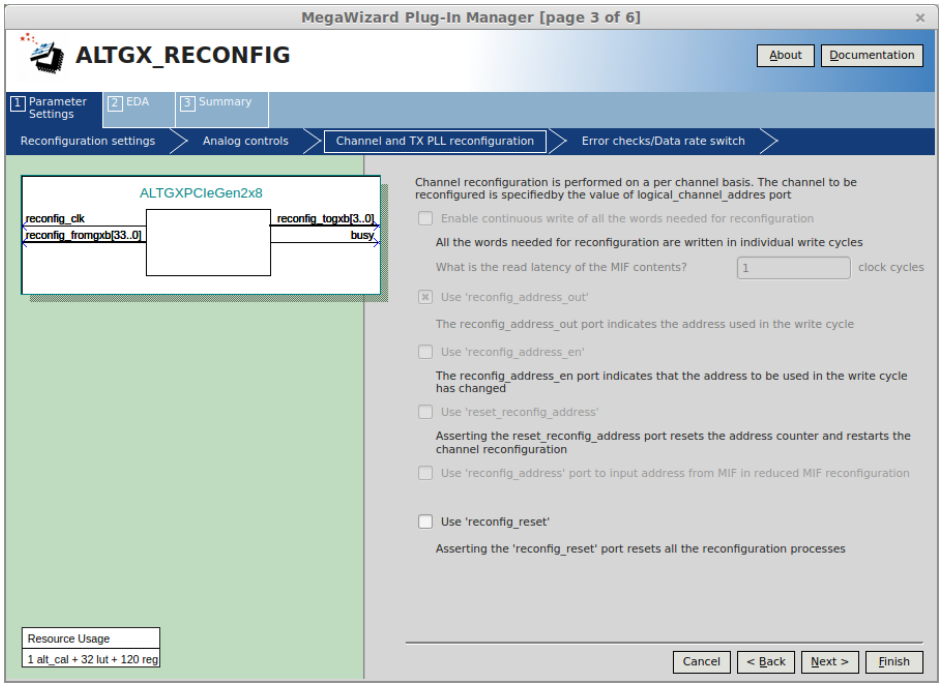

Figure 5.17: ALTGX Reconfiguration Channel Tab

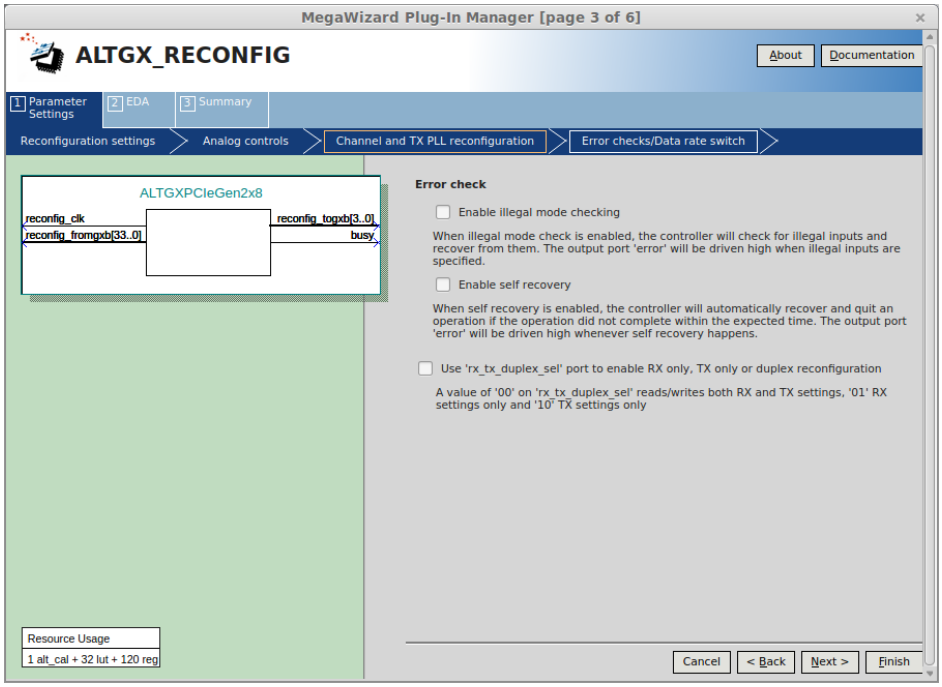

Figure 5.18: ALTGX Reconfiguration Error Tab

5.2.2 Creating Constraints files for IP Compiler Designs

Advanced users may also want to edit and modify the constraint files. This not required or recommended for novice users. The example designs in the RIFFA 2.2.2 distribution contain appropriate constraint files for the example designs. However if the need arises, we demonstrate the constraints we used below.

If the goal is to generate a RIFFA design completely from scratch, each board directory comes with a RIFFA wrapper verilog file and instantiates a vendor-specific translation layer. It is highly recommended to re-use these files RIFFA wrapper when creating designs from scratch. Users should also use the constraints file (.sdc) in the board directory, and in the *constr/*, or read the User Guide provided with each board.

To appropriately constrain your PCIe reference clocks, place the constraints shown in Listing 5.4 in your .sdc file. Modify the name of the osc_50MHz and PCIE_REFCLK ports to match your design

Listing 5.4: .sdc constraints for Qsys and Megawizard designs

```
create_clock -name PCIE_REFCLK -period 10.000 [get_ports {PCIE_REFCLK}]
create_clock -name osc_50MHz -period 20.000 [get_ports {OSC_BANK3D_50MHZ}]
derive_pll_clocks -create_base_clocks
derive_clock_uncertainty
# 50 MHZ PLL Clock
create_generated_clock -name clk50 -source [get_ports {OSC_50_BANK2}] \
[get_nets {*|altpll_component|auto_generated|wire_pll1_clk[0]}]
# 125 MHZ PLL Clock
create_generated_clock -name clk125 -multiply_by 5 -divide_by 2 -source \
[get_ports {OSC_50_BANK2}] \
[get_nets {*|altpll_component|auto_generated|wire_pll1_clk[1]}]
# 250 MHZ PLL Clock
create_generated_clock -name clk250 -multiply_by 5 \
-source [get_ports {OSC_50_BANK2}] [get_nets \
  {*|altpll_component|auto_generated|wire_pll1_clk[2]}]
```

Likewise, copy the constraints in Listing 5.5 into your .qsf file. Copy the location assignment commands for each PCIe Pin in your reference design.

Listing 5.5: .qsf settings for IP Compiler Designs

```
#============================================================
# PCIE Connections
#============================================================
set_location_assignment <PCIE_REFCLK_PIN> -to PCIE_REFCLK
set_instance_assignment -name IO_STANDARD HCSL -to PCIE_REFCLK
set_location_assignment <PCIE_REFCLK_PIN(n)> -to ''PCIE_REFCLK(n)''
set_instance_assignment -name IO_STANDARD HCSL -to ''PCIE_REFCLK(n)''
set_location_assignment <PCIE_RESET_N> -to PCIE_RESET_N
set_instance_assignment -name IO_STANDARD ''2.5 V'' -to PCIE_RESET_N

# For each PCIE Lane (L) set the pin locations from the board user guide!
#============================================================================
#PCIE TX_OUT L
#============================================================================
set_location_assignment <TX_LANE[L]_PIN>   -to PCIE_TX_OUT[L]
set_location_assignment <TX_LANE[L]_PIN(n)> -to ''PCIE_TX_OUT[L](n)''
#============================================================================
#PCIE RX_IN L
#============================================================================
set_location_assignment <RX_LANE[L]_PIN>  -to PCIE_RX_IN[L]
set_location_assignment <RX_LANE[L]_PIN(n)> -to ''PCIE_RX_IN[L](n)''
```

6 Developer Documentation

This chapter describes RIFFA 2.2.2 at a level of detail that is useful for RIFFA developers. Users of RIFFA should not read this section until they are comfortable developing for RIFFA or have experience with PCIe and DMA concepts.

6.1 Architecture Description

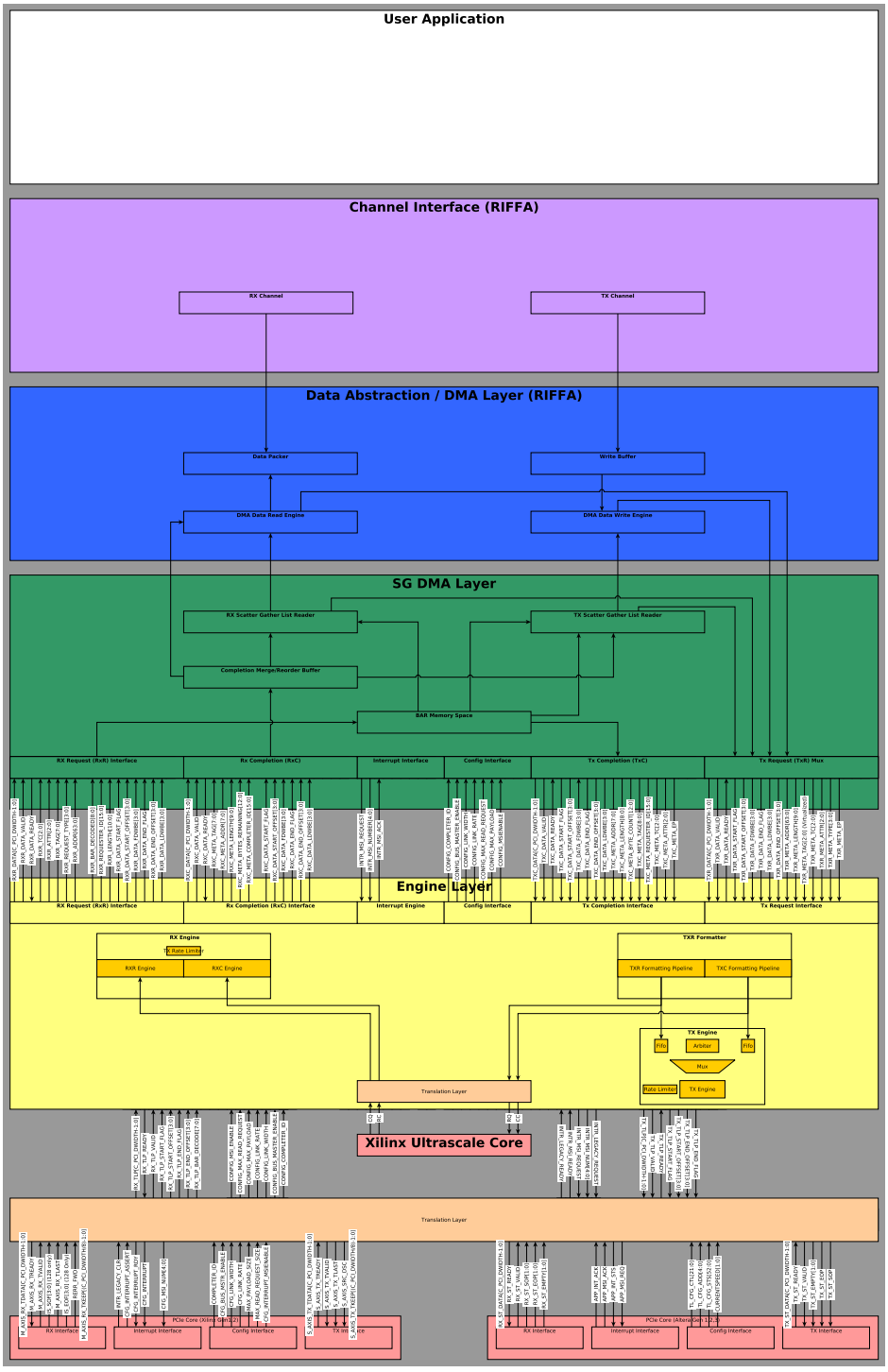

Figure 6.1: High level RIFFA Diagram

- **IP Interfaces** The Vendor IP interfaces provied low-level access to the PCIe bus. Each vendor provides a set of signals for communicating over PCIe. Xilinx FPGAs without PCIe Gen3 support provide an interface very similar to Altera FPGAs. We call this the "Classic Interface". Newer Xilinx devices with PCIe Gen3 support have completely different non-compatible interfaces (CC, CQ, RC, RQ instead of RX and TX). We call this the "Xilinx Ultrascale Interface".

  *Files: *.xci, *.qsys (And others generated by vendor tools)*

- **Translation Layer** The Translation Layer provides a set of vendor-independent interfaces and signal names.

  There is one translation layer for each interface. The "Classic Translation Layer" provides a set of interfaces (RX, TX, Interrupt, and Configuration) and vendor independent signal names to higher layers. There is very little logic in these layers, and there should be no timing-critical logic here.

  The "Ultrascale Translation Layer" operates on the ultrascale interface. Similar to the classic translation layer, it contains very little logic. It provides the interfaces: RX Completion, RX Request, TX Completion, TX Request, Interrupt, and Configuration.

  Files: translation_altera.v, translation_xilinx.v, txc_engine_ultrascale.v, txr_engine_ultrascale.v

- **Formatting Engine Layer** The Formatting Engine Layer is responsible for formatting requests and completions into packets. This layer provides four interfaces: RX Completion (RXC) for receiving completions (responses to memory read requests), RX Request (RXR) for receiving memory read and write requests, TX Completion (TXC) for transmitting completions (reponses to memory read requests), and TX Request (TXR) for transmitting read and write requests.

  The engine layer abstracts vendor specific features, such as Xilinx's Classic-Interface Big-Endian requirement and Altera's Quad-word Alignment. The C_VENDOR parameter for the engine layer switches between Xilinx, Altera, and Ultrascale logic to produce TLPs (Classic Interface) and AXI Descriptors (Ultrascale Interface).

  The RX path of the engine layer has packet parsers for TLPs and AXI Descriptors. These are parameterized by width, as of RIFFA 2.2. The TX Path of the engine layer has packet formatters for TLPs and AXI Descriptors.

  As alluded to in the Translation Layer, the Classic IP Cores provide only two transmit interfaces (RX, and TX), while the Xilinx Ultrascale IP Core handles RX Demultiplexing and multiplexing internally and provides four interfaces (RXC, RXR, TXC, and TXR). For this reason, the multiplexing/FIFO logic used in the Classic interfaces are not necessary for the Xilinx interface.

  After the Engine-Layer, higher layers should be vendor agnostic, if not bus agnostic. The exception will be sideband signals signals. (How much of this ideal can be achieved remains to be seen)

  Note: The engine layer currently uses word-aligned addresses, and byte-enable signals to specify sub-word addresses. In the future, all addresses will be byte-aligned and word enables will be handled in the formatting logic.

  Files: engine_layer.v, schedules.vh, rx_engine_classic.v, rxc_engine_classic.v, rxr_engine_classic.v, tx_engine_classic.v, txc_engine_classic.v, txr_engine_classic.v, rx_engine_ultrascale.v, rxc_engine_ultrascale.v,

rxr_engine_ultrascale.v, tx_engine_ultrascale.v, txc_engine_ultrascale.v, txr_engine_ultrascale.v

- **Scatter Gather (SG) DMA Layer** The Scatter Gather DMA Layer handles reading from and writing to scatter gather lists and providing the addresses found in these lists to the data-request logic in the Data Abstraction layer. In RIFFA, each channel has its own SG DMA list logic.

  The Completion Merge/Reorder buffer handles out-of-order completions. In the PCIe specification, a memory request can be serviced by multiple smaller completions (the responses must remain in order). Completions from different memory requests can be returned in any order. The reorder buffer releases data when all of the responses to a memory request have been received.

  Memory read and write requests to the host are multiplexed by the TX Request Mux. These are serviced fairly in round robin order.

  The Scatter Gather List Readers issue read requests to read data from the Scatter Gather List (SGL) created by the driver. This list contains the address and length of pages containing data to transmit. When an SGL has been exhausted, an interrupt is raised and the SGL is refilled or the transaction is comlete.

  Each element in the SGL 128-bit triple: 32'b0, 32'b Length of Data in 32-bit words, 64'b Address of Page. The addresses in this list are provided to the DMA Data Read Engine in the Data Abstraction layer. Since the SGL must be a single continuous stream of 128-bit elements regardless of the size of the interface, gaps and mis-alignments due to packet formatting are removed using the Data Packer, which receives its data from the reorder buffer.

  The location of the SGL in host memory is written to the BAR Memory space. The BAR Memory space is partitioned among the channels. Only the host can issue read and write requests to this memory space. Since the memory space is partitioned, the RX Request interface and TX Completion interface do not have demultiplexing or multiplexing logic.

  A more through treatment of the SG DMA Layer can be found in Sec. 6.1.1.

  Files: reorder_queue.v, sg_list_reader_*.v, sg_list_requester.v fifo_packer_*.v, registers.v, tx_multiplexer_*.v*

- **Data Abstraction / DMA Layer** The Data Abstraction / DMA Layer is responsible for making requests to read data from, or write data to host memory.

  The read and write addresses are provided by the Scatter Gather list readers. Since RIFFA provides a single continuous stream of 32-bit words regardless of the size of the interface, gaps and mis-alignments due to packet formatting are removed using the Data Packer, which receives its data from the reorder buffer. On the TX side, this is not necessary. However a write buffer, and other transaction tracking logic is necessary for buffering, and removing non-integral data.

  A more through treatment of the Data Abstraction Layer can be found in Sec. 6.1.2.

  Files: reorder_queue.v, rx_port_*.v, rx_port_reader.v, fifo_packer_*.v, tx_port_writer.v tx_port_buffer_*.v tx_port_monitor_*.v*

- **Channel Interface** *Files: rx_channel_gate_*.v, tx_channel_gate_*.v*

- User Logic

6.1.1 Scatter Gather DMA Layer

READS from the SG lists are prioritized

6.1.2 Data Abstraction DMA Layer

6.2 Software Description

6.3 FPGA RX Transfer / Host Send

Parameter	Value
Data Transfer Length	128 (32-bit words)
Data Transfer Offsfet	0
Data Transfer Last	1
Data Transfer Channel	0
Data Page Address (DMA)	0x00000000_FEED0000
SGL Head Address	0x00000000_BEEF0000

- A user makes an call to fpga_send() to transfer 128 32-bit words of data on Channel 0.
- The RIFFA driver writes {32'd128} to Channel 0's RX Length register, and {31'd0,1'b1} to Channel 0's RX OffLast register. This notifies the FPGA that a new transfer is happening and will raise CHNL_RX for the user application. *Files: rxr_engine_*.v, registers.v, channel*.v, rx_port.v rx_port_gate.v, rx_port_reader.v*
- The RIFFA driver allocates an SGL with 1 element (4 32-bit words) at address {64'h0000_0000_BEEF_0000}. The driver fills the list with the length and address of the user data:
{32'd0,32'd128,64'h0000_0000_FEED_0000}.
- The RIFFA driver communicates the address and length of the SGL by writing {32'hBEEF0000} to to Channel 0's RX SGL Address Low register, {32'd0} to to Channel 0's RX SGL Address High register, and {32'd4} to to Channel 0's RX SGL Length register. Writing the RX SGL Length register notifies the RX SG Engine that a transfer has started, and the low and high portions of the 64-bit RX SGL Address are valid. *Files: rxr_engine_*.v, registers.v, channel*.v, rx_port.v, sg_list_requester.v*
- The SG List Requester on the FPGA issues a read request for 4 32-bit words of data starting at address 0xBEEF0000. The FPGA also issues an interrupt. The RIFFA driver reads the Interrupt Status Register of the FPGA and determines that Channel 0 has finished reading the RX SGL. *Files: sg_list_requester.v, rx_port_requester_mux.v, rx_port_*.v, channel*.v, tx_multiplexer.v, engine_layer.v, txr_engine_*.v, interrupt.v*
- The FPGA receieves a completion with 4 32-bit words. After being enqueued in the reorder buffer, the completion is delivered to Channel 0, and packed into the SGL RX Fifo. *Files: rxc_engine_*.v, engine_layer.v, reorder_queue*.v, fifo_packer_*.v*
- The RX Port Reader removes the SG element from the FIFO, and issues several read requests to receive all 128 32-bit words. *Files: rx_port_reader.v, rx_port_*.v, channel*.v, tx_multiplexer.v, engine_layer.v, txr_engine_*.v, tx_multiplexer.v*
- The completions return interleaved and are reordered in the reorder buffer. The reorder buffer releases the completions in order to the fifo packer, which puts them in the FIFO. The RX Port Channel Gate issues the data to the user. *Files: rxc_engine_*.v, engine_layer.v, reorder_queue*.v, fifo_packer_*.v, rx_port_reader.v, rx_port_channel_gate.v, channel*.v*

- The FPGA raises an interrupt with the last word of data is put into the Main Data Fifo. The RIFFA driver reads the Interrupt Status Register of the FPGA and determines that Channel 0 has finished the RX Transaction. The RIFFA driver reads the RX Words Read register to determine how many words were read during the transaction.
- Control is returned to the user.

6.4 TX Transfer

6.5 FPGA RX Transfer / Host Send

Parameter	Value
Data Transfer Length	128 (32-bit words)
Data Transfer Offsfet	0
Data Transfer Last	1
Data Transfer Channel	0
Data Page Address (DMA)	0x00000000_FEED0000
SGL Head Address	0x00000000_BEEF0000

- A user makes an call to fpga_recv() to transfer 128 32-bit words of data from Channel 0.
- The RIFFA driver allocates an SGL with 1 element (4 32-bit words) at address {64'h0000_0000_BEEF_0000}. The driver fills the list with the length and address of the user data: {32'd0,32'd128,64'h0000_0000_FEED_0000}.
- The user application independently raises CHNL_TX and starts writing data to CHNL_TX_DATA. RIFFA core logic reads transaction parameters from CHNL_TX_OFF, CHNL_TX_LAST, and CHNL_TX_LEN and acknowledges them with CHNL_TX_ACK. *Files: tx_port_channel_gate.v*
- An interrupt is raised by the FPGA. The RIFFA driver reads the Interrupt Status Register of the FPGA and determines that Channel 0 wishes to start a new TX Transaction. The driver ISR reads {32'd128} from Channel 0's TX Length register, and {31'd0, 1'b1} from Channel 0's TX OffLast register. Reading the OffLast register notifies the FPGA that the new transfer has been accepted. *Files: rxr_engine_*.v, riffa.v, registers.v, channel*.v, tx_port_*.v, tx_port_writer.v, tx_port_monitor_*.v, engine_layer.v, txc_engine_*.v*
- The RIFFA driver communicates the address and length of the SGL by writing {32'hBEEF_0000} to to Channel 0's TX SGL Address Low register, {32'd0} to to Channel 0's TX SGL Address High register, and {32'd4} to to Channel 0's TX SGL Length register. Writing the TX SGL Length register notifies the TX SG Engine that a transfer has started, and the low and high portions of the 64-bit TX SGL Address are valid. *Files: rxr_engine_*.v, registers.v, channel*.v, rx_port.v, sg_list_requester.v*
- The SG List Requester on the FPGA issues a read request for 4 32-bit words of data starting at address 0xBEEF0000. The FPGA raises an interrupt. The RIFFA driver reads the Interrupt Status Register of the FPGA and determines that Channel 0 has finished reading the TX SGL. *Files: sg_list_requester.v, rx_port_requester_mux.v, rx_port_*.v, channel*.v, tx_multiplexer.v, engine_layer.v, txr_engine_*.v, interrupt.v*
- The FPGA receieves a completion with 4 32-bit words. After being enqueued in the reorder buffer, the completion is delivered to Channel 0, and packed into the SGL TX Fifo. *Files: rxc_engine_*.v, engine_layer.v, reorder_queue*.v, fifo_packer_*.v*

- The TX Port Writer removes the SG element from the FIFO, and issues several write requests to write all 128 32-bit words. *Files: tx_port_monitor.v, tx_port_writer.v, tx_port_*.v, channel*.v, tx_multiplexer.v, engine_layer.v, txr_engine_*.v, tx_multiplexer.v*
- When the last write transaction has been accepted by the core, the FPGA raises an interrupt. The RIFFA driver reads the Interrupt Status Register of the FPGA and determines that Channel 0 has finished writing data. The RIFFA driver reads the TX Words Written register to determine how many words were written during the transaction (in case of early termination, or overflow). *Files: rxr_engine_*.v, riffa.v, interrupt.v, registers.v, channel*.v, tx_port_*.v, tx_port_writer.v, engine_layer.v, txc_engine_*.v*
- Control is return to the user because the TX_LAST signal was set to 1.