**NATIONAL AND KAPODISTRIAN
UNIVERISTY OF ATHENS
FACULTY OF PHYSICS
DEPARTMENT OF ELECTRONICS, COMPUTERS,
TELECOMMUNICATION AND CONTROLL**

Master Thesis

Interconnecting a Linux Host with a FPGA Board through PCI-Express

Angelos Kyriakos

Supervisor: Dionysios Reisis
Associate Professor

Athens
12/2016

NATIONAL AND KAPODISTRIAN
UNIVERISTY OF ATHENS
FACULTY OF PHYSICS
DEPARTMENT OF ELECTRONICS, COMPUTERS,
TELECOMMUNICATION AND CONTROLL

Master in Control and Computing
Thesis

Interconnecting a Linux host with a FPGA board through PCI-Express

Angelos Kyriakos

Student Registration Number: 2014511

Supervisor: Dionysios Reisis
Associate Professor

Examination Committee

Dimitrios Frantzeskakis	Hector Nistazakis	Dionysios Reisis
Professor	Associate Professor	Associate Professor

Athens
12/2016

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENT

I would first like to thank my thesis advisor Associate Professor Dionysios Reisis of the National and Kapodistrian University of Athens for the valuable help and guidance during my research. I would also like to thank Dr. Kostas Nakos for his assistance an guidance that provided me the foundation needed to begin my research.

Finally, I must express my very profound gratitude to my parents and to my friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author

Angelos Kyriakos

Introduction

The objective of this thesis is to provide a mean of communication between a FPGA and a PC host over PCI Express, a high-speed computer expansion bus. The whole thesis was part of the Nephele research project which is supported by the Horizon2020 Framework Programme for Research and Innovation of the European Commission [1]. NEPHELE is developing a dynamic optical network infrastructure for future scale-out, disaggregated datacenters. The Software-Defined Networking (SDN) controller, that is also being developed by the project, is going to be interfaced with all the switches with PCI Express.

Firstly, we studied the PCIe protocol and we came to the conclusion that is necessary to implement a PCIe driver along with a PCIe endpoint on the FPGA. Searching for other people's implementation of this idea, with came up with a company called Xillibus [2], but we choose to implement our own code because we couldn't afford this solution. Xilinx provides an IP core that handles some part of the protocol but a lot of functionality must be implemented by the user, moreover Xilinx does not provided any driver. Later on, as we were planning how we are going to implement the device driver, we discovered the project RIFFA (Reusable Integration Framework for FPGA Accelerators) [3] which is a simple framework for communicating data from a host CPU to a FPGA via a PCI Express bus. Then we focus our research on using this framework to communicate with the FPGA. The framework itself comes with hardware and software open-source code, which will make future addition of functionality to the code easier.

The thesis is structured in four chapters. The first chapter introduces the PCI Express interconnect, the goal of this chapter is to provide the necessary information of how PCIe works to someone who is not an expert on the subject. In the second chapter we present some points about modern PCIe drivers and how should be implemented. RIFFA comes along with an open-source driver that uses most of the functions presented here. The third chapter is focused on the hardware. The first half of the chapter is a summary of the functionalities provided by the Xilinx IP core, the second half is a summary of the RIFFA endpoint hardware architecture. The last chapter consists of test programs that we created using the RIFFA API, the hardware module that communicates with the RIFFA endpoint inside the FPGA, as well as measurements of the performance of the link.

Chapter 1

PCI Express Interconnect

This chapter presents an overview of the PCI Express architecture and key concepts. PCI Express is a high performance, general purpose I/O interconnect defined for a wide variety of computing and communication platforms. Key PCI attributes, such as its usage model, load-store architecture, and software interfaces, are maintained, whereas its parallel bus implementation is replaced by a highly scalable, fully serial interface. PCI Express takes advantage of recent advances in point-to-point interconnects, Switch-based technology, and packetized protocol to deliver new levels of performance and features. Power Management, Quality Of Service (QoS), Hot-Plug/Hot-Swap support, Data Integrity, and Error Handling are among some of the advanced features supported by PCI Express. [4]

1.1 PCIe Link

A Link represents a dual-simplex communications channel between two components. The fundamental PCI Express Link consists of two, low-voltage, differentially driven signal pairs: a Transmit pair and a Receive pair as shown in 1.1.

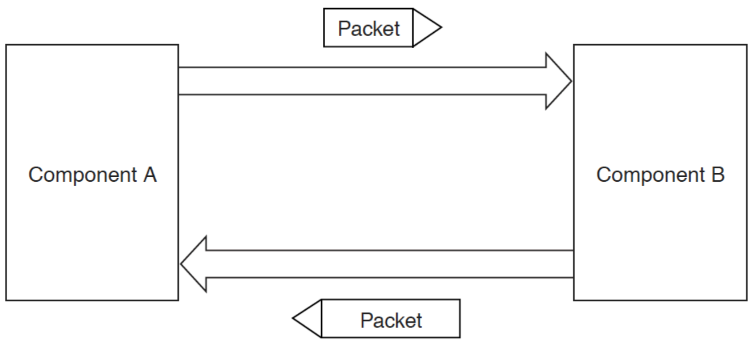


Figure 1.1: PCI Express Link

Some of the primary link's attributes are the following. The PCI Express link consists of dual unidirectional differential Links, implemented as a Transmit pair and a Receive pair. Once initialized, each link must only operate at one of the supported signaling levels. In the first generation of PCI Express technology, there was only one signaling rate supported, which provided an effective 2.5 Gigabits/second/Lane/direction of raw bandwidth. The second generation provides an effective 5.0 Gigabits/second/Lane/direction of raw bandwidth. The data rate is expected to increase in the future as the technology advances.

A link must support at least one lane and each Lane represents a set of differential signal pairs (one pair for transmission, one pair for reception). To scale bandwidth, a Link must aggregate multiple lanes denoted by xN where N may be any of the supported Link widths. An x8 link represents an aggregate bandwidth of 20 Gigabits/second of raw bandwidth in each direction. The implementation of the hardware in this thesis is of an x8 link. It should be noted that each link must support a symmetric number of Lanes in each direction, i.e., a x16 link indicates there are 16 differential signal pairs in each direction.

During hardware initialization, each PCI Express link is set up following a negotiation of lane widths and frequency of operation by the two agents at each end of the link. No firmware or operating system software is involved. [4]

1.2 PCIe Fabric Topology

A fabric is composed of point-to-point Links that interconnect a set of components, an example fabric topology is shown in Figure 1.2. This figure illustrates a single fabric instance referred to as a hierarchy, composed of a Root Complex, multiple Endpoints (I/O devices), a Switch, and a PCI Express to PCI/PCI-X Bridge, all interconnected via PCI Express links.

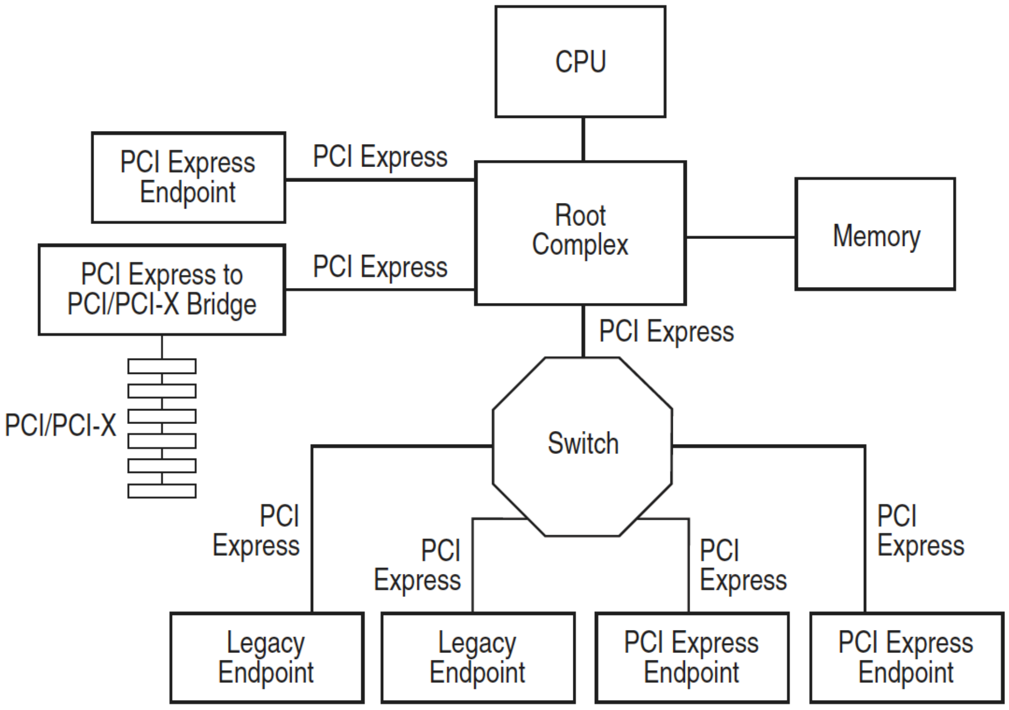


Figure 1.2: Example Fabric Topology

A Root Complex denotes the root of the I/O hierarchy that connectsthe CPU/memory subsystem to the I/O. As illustrated in Figure 1.2, a Root Complex may support one or more PCI Express ports. Each interface defines a separate hierarchy domain. Each hierarchy domain may be composed of a single Endpoint or a sub-hierarchy containing one or more Switch components and Endpoints. Endpoint refers to a type of Function that can be the Requester or the Completer of a PCI Express transaction either on its own behalf or on behalf of a distinct non-PCI Express device (other than a PCI device or Host CPU), e.g., a

PCI Express attached graphics controller or a PCI Express-USB host controller. Endpoints are classified as either legacy, PCI Express, or Root Complex Integrated Endpoints. [4]

1.3 PCIe Layering Overview

The architecture of PCI Express is specified in terms of three discrete logical layers: the Transaction Layer, the Data Link Layer, and the Physical Layer. Each of these layers is divided into two sections: one that processes outbound (to be transmitted) information and one that processes inbound (received) information, as shown in Figure 1.3.

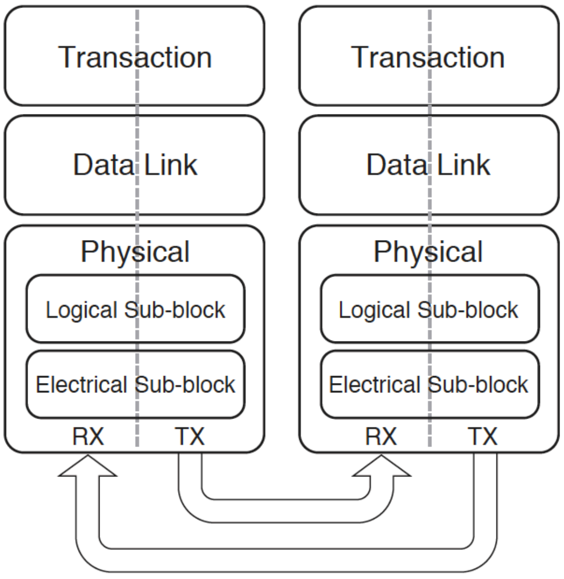


Figure 1.3: High Level Layering Diagram

PCI Express uses packets to communicate information between components. Packets are formed in the Transaction and Data Link Layers to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the other layers, they are extended with additional information necessary to handle packets at those layers. At the receiving side the reverse process occurs and packets get transformed from their Physical Layer representation to the Data Link Layer representation and finally (for Transaction Layer Packets) to the form that can be processed by the Transaction Layer of the receiving device, the procedure is similar to the encapsulation of packets in the network layers of the internet, such as the Transport layer(TCP), the Network layer (IP) and the Link layer.

1.3.1 Transaction layer

The upper layer of the architecture is the Transaction Layer. The Transaction Layer's primary responsibility is the assembly and disassembly of Transaction Layer Packets (TLPs). TLPs are the packets used to communicate transactions, such as read and write, as well as certain types of events. The Transaction Layer is also responsible for managing credit-based flow control for TLPs. Every request packet requiring a response packet is implemented as a split transaction. Each packet has a unique identifier that enables response packets to be directed to the correct originator. The packet format supports different forms of addressing depending on the type of the transaction (Memory, I/O, Configuration, and Message). The Packets may also have attributes such as No Snoop, Relaxed Ordering, and ID-Based Ordering (IDO). The Transaction Layer supports four address spaces: it includes the three PCI address spaces (memory, I/O, and configuration) and adds Message Space. [4]

The Transaction Layer, in the process of generating and receiving TLPs, exchanges Flow Control information with its complementary Transaction Layer implementations on the other side of the link. It is also responsible for supporting both software and hardware-initiated power management. Initialization and configuration functions require the Transaction Layer to store the link's configuration that is generated by the processor, and the link capabilities generated by the physical layer hardware negotiation, such as width and operational frequency. A Transaction Layer's Packet generation and processing services require it to generate TLPs from device core requests and convert received requests TLPs into request for the specific device core.

1.3.2 Data link layer

The middle layer in the stack, the Data Link Layer, serves as an intermediate stage between the Transaction Layer and the Physical Layer. The primary responsibilities of the Data Link Layer include link management and data integrity, including error detection and error correction. The transmission side of the Data Link Layer accepts TLPs assembled by the Transaction Layer, calculates and applies a data protection code and TLP sequence number, and submits them to Physical Layer for transmission across the link. The receiving Data Link Layer is responsible for checking the integrity of received TLPs and for submitting them to the Transaction Layer for further processing. On detection of TLP errors, this layer is responsible for requesting re-transmission of TLPs until information is correctly received, or the link is considered to have failed.

The Data Link Layer also generates and consumes packets that are used for Link management functions. To differentiate these packets from those used by the Transaction Layer (TLP), the term Data Link Layer Packet (DLLP) will be used when referring to packets that are generated and consumed at the Data Link Layer. [4]

Some of the services of the Data Link Layer regarding data protection, error checking and re-transmission are CRC generation, transmitted TLP storage for data link level retry, error checking, TLP acknowledgment are retry messages and error indication for error reporting and logging.

1.3.3 Physical layer

The Physical Layer includes all circuitry for interface operation, including driver and input buffers, parallel-to-serial and serial-to-parallel conversion, PLLs, and impedance matching circuitry. It includes also logical functions related to interface initialization and maintenance. The Physical Layer exchanges information with the Data Link Layer in an implementation-specific format. This Layer is responsible for converting information received from the Data Link Layer into an appropriate serialized format and transmitting it across the PCI Express Link at a frequency and width compatible with the device connected to the other side of the Link. The PCI Express architecture has "hooks" to support future performance enhancements via speed upgrades and advanced encoding techniques. The future speeds, encoding techniques or media may only impact the Physical Layer definition. [4]

Chapter 2

PCI Express Linux Driver

Device drivers take on a special role in the Linux kernel. They are distinct “blackboxes” that make a particular piece of hardware respond to a defined internal programming interface, additionally they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver. Mapping those calls to device-specific operations that act on real hardware is the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and then plugged in at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

One of the features of the Linux operating system is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running. Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

2.1 Classes of Devices and Modules

In Linux devices are distinguished between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one. The programmer can choose to build huge modules implementing different drivers in a single chunk of code. The best approach usually is to create a different module for each new functionality we need to implement, because decomposition is a key element of the scalability and extendability of the driver and the operating system. The three classes are the character, block devices and network interfaces, and they are presented on the following paragraphs, summarizing the most important points of [5].

A character (char) device is one that can be accessed as a stream of bytes (like a binary file in the C programming language). A char driver is in charge of implementing and supporting this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0) are some examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.

Similar to char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write data to a block device like a char device of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and consequently in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

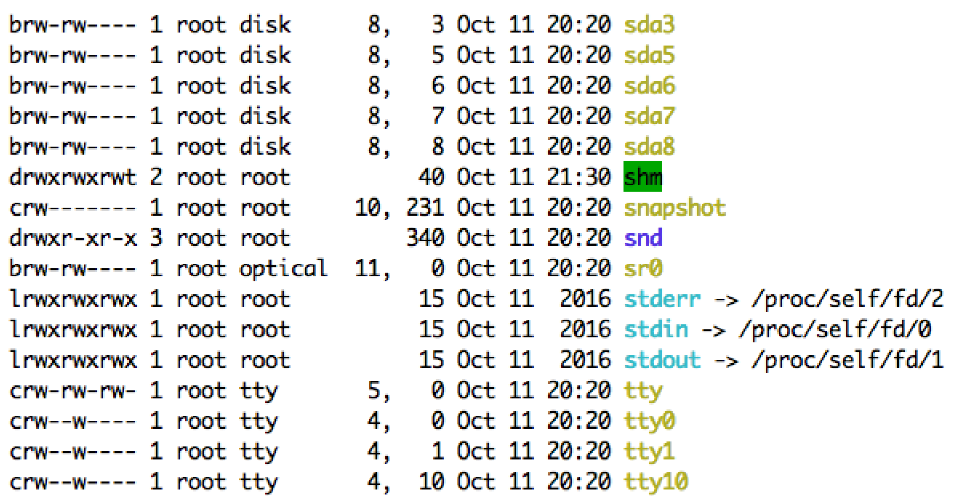
Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets. Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

2.2 Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files and they are conventionally located in the /dev directory. Special files for char drivers are identified by a "c" in the first column of the output of ls –l. Block devices appear in /dev as well, but they are identified by a "b." The following information applies to block and char devices.

If we issue the ls –l command on the /dev directory we get a result similar to the Figure 2.1. We can see there the two numbers (separated by a comma) in the device file entries, which are located just before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The Figure 2.1 shows a few devices as they appear on a typical system. Their major numbers are 8, 10, 11, 5, and 4, while the minors are 0, 1, 3, 5, 6, 7, 8, 64, and 231. The standard input/output files (stdin, stdout, stderr) does not have a major number, because they are not character devices but symbolic links to a more general mechanism, for instance /dev/stdin is a symbolic link to /proc/self/fd/0.

Traditionally, the major number identifies the driver associated with the device. For example, /dev/sda3, /dev/sda4 and so on, are all managed by driver 8, whereas /dev/tty devices are managed by driver 4. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle. The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how the driver is written, we can either get a direct pointer to the



```
brw-rw---- 1 root disk      8,   3 Oct 11 20:20 sda3
brw-rw---- 1 root disk      8,   5 Oct 11 20:20 sda5
brw-rw---- 1 root disk      8,   6 Oct 11 20:20 sda6
brw-rw---- 1 root disk      8,   7 Oct 11 20:20 sda7
brw-rw---- 1 root disk      8,   8 Oct 11 20:20 sda8
drwxrwxrwt 2 root root          40 Oct 11 21:30 shm
crw------- 1 root root     10, 231 Oct 11 20:20 snapshot
drwxr-xr-x 3 root root         340 Oct 11 20:20 snd
brw-rw---- 1 root optical  11,   0 Oct 11 20:20 sr0
lrwxrwxrwx 1 root root          15 Oct 11  2016 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root          15 Oct 11  2016 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root          15 Oct 11  2016 stdout -> /proc/self/fd/1
crw-rw-rw- 1 root tty       5,   0 Oct 11 20:20 tty
crw--w---- 1 root tty       4,   0 Oct 11 20:20 tty0
crw--w---- 1 root tty       4,   1 Oct 11 20:20 tty1
crw--w---- 1 root tty       4,  10 Oct 11 20:20 tty10
```

Figure 2.1: Example of major and minor numbers, after executing ls -l on /dev directory

device from the kernel, or we can use the minor number ourselves as an index into a local array of devices.

The classic way to register a char device driver is with:

```
int register_chrdev(unsigned int major, const char *name,struct file_operations
    *fops);
```

Here, major is the major number of interest, name is the name of the driver as it will appear in /proc/devices, and fops is the default file_operations structure. A call to register_chrdev registers minor numbers 0–255 for the given major, and sets up a default cdev structure for each. Drivers using this interface must be prepared to handle open calls on all 256 minor numbers (whether they correspond to real devices or not), and they cannot use major or minor numbers greater than 255. If you use register_chrdev, the proper function to remove the device(s) from the system is:

```
int unregister_chrdev(unsigned int major, const char *name);
```

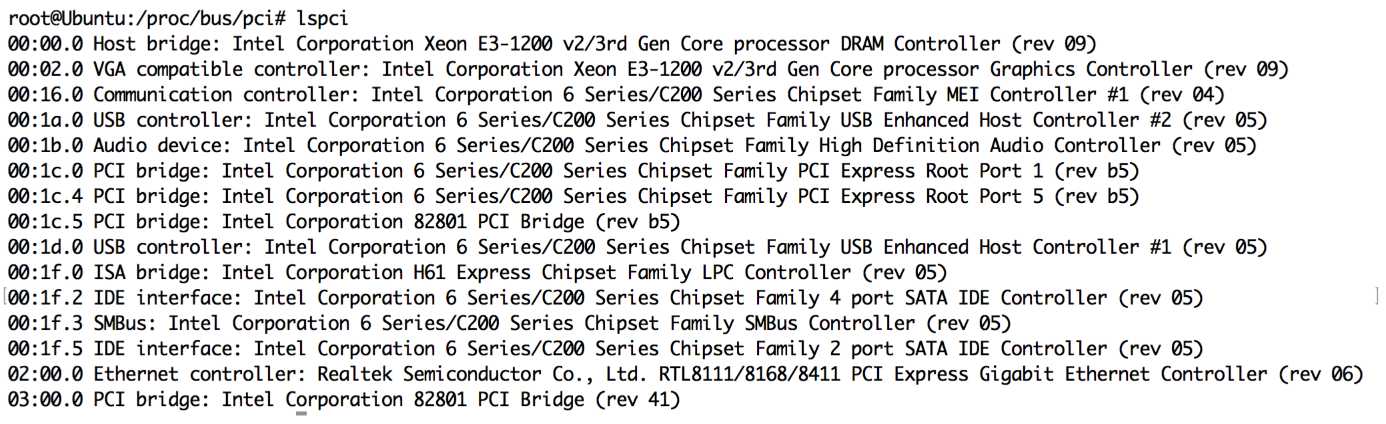
major and name must be the same as those passed to register_chrdev, or the call will fail.

2.3 PCI Addressing

Each PCI peripheral is identified by a bus number, a device number, and a function number. The PCI specification permits a single system to host up to 256 buses, but because 256 buses are not sufficient for many large systems, Linux now supports PCI domains. Each PCI domain can host up to 256 buses. Each bus hosts up to 32 devices, and each device can be a multifunction board (such as an audio device with an accompanying CD-ROM drive) with a maximum of eight functions. Therefore, each function can be identified at hardware level by a 16-bit address, or key. Device drivers written for Linux, though, don't need to deal with those binary addresses, because they use a specific data structure, called pci_dev, to act on the devices. [5]

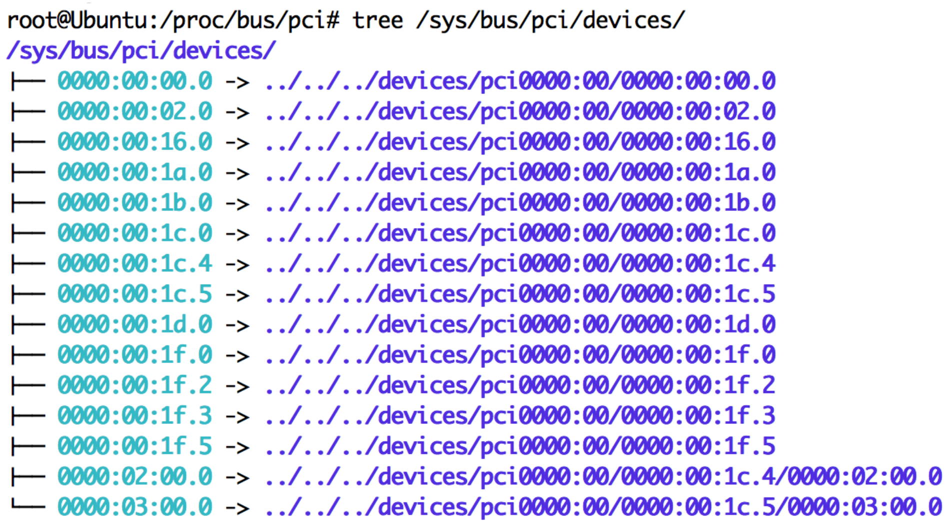
The 16-bit hardware addresses associated with PCI peripherals, although mostly hidden in the struct pci_dev object, are still visible occasionally, especially when lists of devices are being used. One such situation is the output of lspci (part of the pciutils package, available with most distributions). When the hardware address is displayed, it can be shown as two values (an 8-bit bus number and an 8-bit device and function number), as three values (bus, device, and function), or as four values (domain, bus, device, and function); all the values are usually displayed in hexadecimal.

For example we present the outputs of lspci and tree /sys/bus/pci/devices/ on figures 2.2,2.3. All three lists of devices are sorted in the same order, since lspci uses the /proc files as its source of information.



```
root@Ubuntu:/proc/bus/pci# lspci
00:00.0 Host bridge: Intel Corporation Xeon E3-1200 v2/3rd Gen Core processor DRAM Controller (rev 09)
00:02.0 VGA compatible controller: Intel Corporation Xeon E3-1200 v2/3rd Gen Core processor Graphics Controller (rev 09)
00:16.0 Communication controller: Intel Corporation 6 Series/C200 Series Chipset Family MEI Controller #1 (rev 04)
00:1a.0 USB controller: Intel Corporation 6 Series/C200 Series Chipset Family USB Enhanced Host Controller #2 (rev 05)
00:1b.0 Audio device: Intel Corporation 6 Series/C200 Series Chipset Family High Definition Audio Controller (rev 05)
00:1c.0 PCI bridge: Intel Corporation 6 Series/C200 Series Chipset Family PCI Express Root Port 1 (rev b5)
00:1c.4 PCI bridge: Intel Corporation 6 Series/C200 Series Chipset Family PCI Express Root Port 5 (rev b5)
00:1c.5 PCI bridge: Intel Corporation 82801 PCI Bridge (rev b5)
00:1d.0 USB controller: Intel Corporation 6 Series/C200 Series Chipset Family USB Enhanced Host Controller #1 (rev 05)
00:1f.0 ISA bridge: Intel Corporation H61 Express Chipset Family LPC Controller (rev 05)
00:1f.2 IDE interface: Intel Corporation 6 Series/C200 Series Chipset Family 4 port SATA IDE Controller (rev 05)
00:1f.3 SMBus: Intel Corporation 6 Series/C200 Series Chipset Family SMBus Controller (rev 05)
00:1f.5 IDE interface: Intel Corporation 6 Series/C200 Series Chipset Family 2 port SATA IDE Controller (rev 05)
02:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller (rev 06)
03:00.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 41)
```

Figure 2.2: Output of lspci



```
root@Ubuntu:/proc/bus/pci# tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
├── 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
├── 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
├── 0000:00:16.0 -> ../../../devices/pci0000:00/0000:00:16.0
├── 0000:00:1a.0 -> ../../../devices/pci0000:00/0000:00:1a.0
├── 0000:00:1b.0 -> ../../../devices/pci0000:00/0000:00:1b.0
├── 0000:00:1c.0 -> ../../../devices/pci0000:00/0000:00:1c.0
├── 0000:00:1c.4 -> ../../../devices/pci0000:00/0000:00:1c.4
├── 0000:00:1c.5 -> ../../../devices/pci0000:00/0000:00:1c.5
├── 0000:00:1d.0 -> ../../../devices/pci0000:00/0000:00:1d.0
├── 0000:00:1f.0 -> ../../../devices/pci0000:00/0000:00:1f.0
├── 0000:00:1f.2 -> ../../../devices/pci0000:00/0000:00:1f.2
├── 0000:00:1f.3 -> ../../../devices/pci0000:00/0000:00:1f.3
├── 0000:00:1f.5 -> ../../../devices/pci0000:00/0000:00:1f.5
├── 0000:02:00.0 -> ../../../devices/pci0000:00/0000:00:1c.4/0000:02:00.0
└── 0000:03:00.0 -> ../../../devices/pci0000:00/0000:00:1c.5/0000:03:00.0
```

Figure 2.3: Output of tree /sys/bus/pci/devices/

The hardware circuitry of each peripheral board answers queries pertaining to three address spaces: memory locations, I/O ports, and configuration registers. The first two address spaces are shared by all the devices on the same PCI bus (i.e., when you access a memory location, all the devices on that PCI bus see the bus cycle at the same time). The configuration space, on the other hand, exploits geographical addressing. Configuration queries address only one slot at a time, so they never collide. As far as the driver is concerned, memory and I/O regions are accessed in the usual ways via inb, readb, and so forth. Configuration transactions, on the other hand, are performed by calling specific kernel functions to access configuration registers. With regard to interrupts, every PCI slot has four interrupt pins, and each device function can use one of them without being concerned about how those pins are routed to the CPU. Such routing is the responsibility of the computer platform and is implemented outside of the PCI bus. [5]

The I/O space in a PCI bus uses a 32-bit address bus, while the memory space can be accessed with either 32-bit or 64-bit addresses. Addresses are supposed to be unique to one device, but software may erroneously configure two devices to the same address, making it impossible to access either one. But this problem never occurs unless a driver is willingly playing with registers it shouldn't touch. The firmware initializes PCI hardware at system boot, mapping each region to a different address to avoid collisions. The addresses to which these regions are currently mapped can be read from the configuration space, so the Linux driver can access its devices without probing. After reading the configuration registers, the driver can safely access its hardware. [5]

The PCI configuration space consists of 256 bytes for each device function (except for PCI Express devices, which have 4 KB of configuration space for each function), and the layout of the configuration registers is standardized. Four bytes of the configuration space hold a unique function ID, so the driver can identify its device by looking for the specific ID for that peripheral. In summary, each device board is geographically addressed to retrieve its configuration registers; the information in those registers can then be used to perform normal I/O access, without the need for further geographic addressing.

2.4 Module Device Table and Registration of Driver

This pci_device_id structure needs to be exported to user space to allow the hotplug and module loading systems know what module works with what hardware devices. The macro MODULE_DEVICE_TABLE accomplishes this. An example is:

```
MODULE_DEVICE_TABLE(pci, i810_ids);
```

This statement creates a local variable called __mod_pci_device_table that points to the list of struct pci_device_id. Later in the kernel build process, the depmod program searches all modules for the symbol __mod_pci_device_table. If that symbol is found, it pulls the data out of the module and adds it to the file /lib/modules/KERNEL_VERSION/modules.pcimap. After depmod completes, all PCI devices that are supported by modules in the kernel are listed, along with their module names, in that file. When the kernel tells the hotplug system that a new PCI device has been found, the hotplug system uses the modules.pcimap file to find the proper driver to load.[5]

The main structure that all PCI drivers must create in order to be registered with the kernel properly is the struct pci_driver structure. This structure consists of a number of function callbacks and variables that describe the PCI driver to the PCI core. In summary, to create a proper struct pci_driver structure, only four fields need to be initialized:

```c
static struct pci_driver pci_driver = {
  .name = "pci_skel",
  .id_table = ids,
  .probe = probe,
  .remove = remove,
};
```

To register the struct pci_driver with the PCI core, a call to pci_register_driver is made with a pointer to the struct pci_driver. When the PCI driver is to be unloaded, the struct pci_driver needs to be unregistered from the kernel. This is done with a call to pci_unregister_driver.

When this call happens, any PCI devices that were currently bound to this driver are removed, and the remove function for this PCI driver is called before the pci_unregister_driver function returns.

2.4.1 PCI Probe

In older kernel versions, the function, pci_register_driver, was not always used by PCI drivers. Instead, they would either walk the list of PCI devices in the system by hand, or they would call a function that could search for a specific PCI device. The ability to walk the list of PCI devices in the system within a driver has been removed from the 2.6 kernel in order to prevent drivers from crashing the kernel if they happened to modify the PCI device lists while a device was being removed at the same time. The ability to find a specific PCI device is supported by the pci_get_device function. This function scans the list of PCI devices currently present in the system, and if the input arguments match the specified vendor and device IDs, it increments the reference count on the struct pci_dev variable found, and returns it to the caller. After the driver is done with the struct pci_dev returned by the function, it must call the function pci_dev_put to decrement the usage count properly back to allow the kernel to clean up the device if it is removed.

An example is :

```
struct pci_dev *dev;
dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO, NULL);
if (dev) {
// Use the PCI device
...
pci_dev_put(dev);
}
```

In the probe function for the PCI driver, before the driver can access any device resource (I/O region or interrupt) of the PCI device, the driver must call the pci_enable_device function. This function actually enables the device. It wakes up the device and in some cases also assigns its interrupt line and I/O regions.

The driver then needs to call the pci_set_master, a function that asserts the bus master bit on the configuration registers. This is needed of the communication with the device is going to be implemented with direct memory access (DMA).

2.5 Direct Memory Access

Direct memory access, or DMA, is the advanced topic that completes our overview of how to create a modern PCI driver. DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated. [5]

Let's begin with the mechanism of how a DMA transfer takes place, considering only input transfers to simplify the discussion. Data transfer can be triggered in two ways: either the software asks for data (via a function such as read) or the hardware asynchronously pushes data to the system. In the first case, the steps involved can be summarized as follows:

1. When a process calls read, the driver method allocates a DMA buffer and instructs the hardware to transfer its data into that buffer. The process is put to sleep.
2. The hardware writes data to the DMA buffer and raises an interrupt when it's done.
3. The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

The second case comes about when DMA is used asynchronously. This happens, for example, with data acquisition devices that go on pushing data even if nobody is reading them. In this case, the driver should maintain a buffer so that a subsequent read call will return all the accumulated data to user space. The steps involved in this kind of transfer are slightly different:

1. The hardware raises an interrupt to announce that new data has arrived.
2. The interrupt handler allocates a buffer and tells the hardware where to transfer its data.
3. The peripheral device writes the data to the buffer and raises another interrupt when it's done.
4. The handler dispatches the new data, wakes any relevant process, and takes care of housekeeping.

The processing steps in all of these cases emphasize that efficient DMA handling relies on interrupt reporting. While it is possible to implement DMA with a polling driver, it wouldn't make sense, because a polling driver would waste the performance benefits that DMA offers over the easier processor-driven I/O. As far as interrupts are concerned, PCI is easy to handle. By the time Linux boots, the computer's firmware has already assigned a unique interrupt number to the device, and the driver just needs to use it. The interrupt number is stored in configuration register 60 (PCI_INTERRUPT_LINE), which is one byte wide.

2.5.1 DMA mappings

A DMA mapping is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device. The PCI code distinguishes between two types of DMA mappings, depending on how long the DMA buffer is expected to stay around, Coherent and Streaming DMA mappings.

Coherent DMA mappings usually exist for the life of the driver. A coherent buffer must be simultaneously available to both the CPU and the peripheral. As a result, coherent mappings must live in cache-coherent memory. Coherent mappings can be expensive to set up and use.

Streaming mappings are usually set up for a single operation. The kernel developers recommend the use of streaming mappings over coherent mappings whenever possible, and there are two reasons for this recommendation. The first is that, on systems that support mapping registers, each DMA mapping uses one or more of them on the bus. Coherent mappings, which have a long lifetime, can monopolize these registers for a long time, even when they are not being used. The other reason is that, on some hardware, streaming mappings can be optimized in ways that are not available to coherent mappings. [5]

2.5.2 Setting up streaming DMA mappings

These mappings expect to work with a buffer that has already been allocated by the driver and, therefore, have to deal with addresses that they did not choose. On some architectures, streaming mappings can also have multiple, discontiguous pages and multipart "scatter/-gather" buffers. For all of these reasons, streaming mappings have their own set of mapping functions.

When setting up a streaming mapping, you must tell the kernel in which direction the data is moving. The following symbols have been defined for this purpose:
DMA_TO_DEVICE
DMA_FROM_DEVICE
These two symbols should be reasonably self-explanatory. If data is being sent to the device, DMA_TO_DEVICE should be used; data going to the CPU, instead, is marked with DMA_FROM_DEVICE.
DMA_BIDIRECTIONAL
If data can move in either direction, use DMA_BIDIRECTIONAL.
DMA_NONE
This symbol is provided only as a debugging aid. Attempts to use buffers with this "direction" cause a kernel panic.

Some important rules apply to streaming DMA mapping. First the buffer must be used only for a transfer that matches the direction value that was given when mapped. Second once the buffer has been mapped it belongs to the device. Until the buffer has been unmapped, the driver should not touch its contents in any way. Only after dma_unmap_single has been called is it safe for the driver to access the contents of the buffer. Among other things, this rule implies that a buffer being written to a device cannot be mapped until it contains all the data to write. And thirdly the buffer must not be unmapped while the DMA is still active, serious system instability is guaranteed.

2.5.3 Scatter/gather mappings

Scatter/gather mappings are a special type of streaming DMA mapping. Suppose you have several buffers, all of which need to be transferred to or from the device. This situation can come about in several ways, including from a readv or writev system call, a clustered disk I/O request, or a list of pages in a mapped kernel I/O buffer. You could simply map each buffer, in turn, and perform the required operation, but there are advantages to mapping the whole list at once.

Many devices can accept a scatterlist of array pointers and lengths, and transfer them all in one DMA operation; for example, "zero-copy" networking is easier if packets can be built in multiple pieces. Another reason to map scatterlists as a whole is to take advantage of systems that have mapping registers in the bus hardware. On such systems, physically discontiguous pages can be assembled into a single, contiguous array from the device's point of view. This technique works only when the entries in the scatterlist are equal to the page size in length, but when it does work, it can turn multiple operations into a single DMA, and speed things up accordingly.

The first step in mapping a scatterlist is to create and fill in an array of struct scatterlist describing the buffers to be transferred. This structure is architecture dependent, and is described in <asm/scatterlist.h>. However, it always contains three fields: a page pointer corresponding to the buffer which will be used in the operation, the length of the buffer and it's offset within the page.

To map a scatter/gather DMA operation, the driver should set the page, offset, and length fields in a struct scatterlist entry for each buffer to be transferred. Then call:

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents, enum
    dma_data_direction direction)
```

where nents is the number of scatterlist entries passed in. The return value is the number of DMA buffers to transfer; it may be less than nents.

For each buffer in the input scatterlist, dma_map_sg determines the proper bus address to give to the device. As part of that task, it also coalesces buffers that are adjacent to each other in memory. If the system that the driver is running on has an I/O memory management unit, dma_map_sg also programs that unit's mapping registers, with the possible result that, from the device's point of view, we are able to transfer a single, contiguous buffer.

The driver should transfer each buffer returned by dma_map_sg. The bus address and length of each buffer are stored in the struct scatterlist entries, but their location in the structure varies from one architecture to the next. Two macros have been defined to make it possible to write portable code:

```
dma_addr_t sg_dma_address(struct scatterlist *sg);

unsigned int sg_dma_len(struct scatterlist *sg);
```

The first function returns the bus (DMA) address from this scatterlist entry, and the second one returns the length of this buffer.remember that the address and length of the buffers to transfer may be different from what was passed in to dma_map_sg.

Once the transfer is complete, a scatter/gather mapping is unmapped with a call to dma_unmap_sg:

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
int nents, enum dma_data_direction direction);
```

Note that nents must be the number of entries that you originally passed to dma_map_sg and not the number of DMA buffers the function returned.

2.5.4 A simple PCI DMA example

As an example of how the DMA mappings might be used, we present a simple example of DMA coding for a PCI device. The actual form of DMA operations on the PCI bus is very dependent on the device being driven. Thus, this example does not apply to any real device. A driver for hypothetical device might define a transfer function like this:

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
size_t count)
{
dma_addr_t bus_addr;
// Map the buffer for DMA
dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
dev->dma_size = count;
bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count,
dev->dma_dir);
dev->dma_addr = bus_addr;
// Set up the device
```

```c
writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
writel(dev->registers.addr, cpu_to_le32(bus_addr));
writel(dev->registers.len, cpu_to_le32(count));
// Start the operation
writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
return 0;
}
```

This function maps the buffer to be transferred and starts the device operation. The other half of the job must be done in the interrupt service routine, which looks something like this:

```c
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
struct dad_dev *dev = (struct dad_dev *) dev_id;
// Unmap the DMA buffer
dma_unmap_single(dev->pci_dev->dev, dev->dma_addr,
dev->dma_size, dev->dma_dir);
// Now is it safe to access the buffer
...
}
```

2.6 IOCTL

Most drivers need, in addition to the ability to read and write the device, the ability to perform various types of hardware control via the device driver. Most devices can perform operations beyond simple data transfers, user space must often be able to request, for example, that the device lock its door, eject its media, report error information, or self destruct. These operations are usually supported via the ioctl method, which implements the system call by the same name. In user space, the ioctl system call has the following prototype:

```c
int ioctl(int fd, unsigned long cmd, ...);
```

In a real system a system call can't actually have a variable number of arguments. System calls must have a well-defined prototype, because user programs can access them only through hardware. Therefore, the dots in the prototype represent not a variable number of arguments but a single optional argument, traditionally identified as char *argp. The dots are simply there to prevent type checking during compilation. The actual nature of the third argument depends on the specific control command being issued (the second argument). Some commands take no arguments, some take an integer value, and some take a pointer to other data. Using a pointer is the way to pass arbitrary data to the ioctl call, the device is then able to exchange any amount of data with user space. [5]

The unstructured nature of the ioctl call has caused it to fall out of favor among kernel developers. Each ioctl command is, essentially, a separate, usually undocumented system call, and there is no way to audit these calls in any sort of manner. Possible alternatives include embedding commands into the data stream (this approach is implemented in RIFFA for some commands) or using virtual filesystems (the Xillybus driver works this way), either sysfs or driverspecific filesystems. However, the fact remains that ioctl is often the easiest and most straightforward choice for true device operations.

Most ioctl implementations consist of a big switch statement that selects the correct behavior according to the cmd argument. Different commands have different numeric values, which are usually given symbolic names to simplify coding. The symbolic name is assigned by a preprocessor definition. Custom drivers usually declare such symbols in their header files

2.7 RIFFA Software Architecture

On the host PC is a kernel device driver and a set of language bindings. The device driver is installed into the operating system and is loaded at system startup. It handles registering all detected FPGAs configured with RIFFA cores. Once registered, a small memory buffer is preallocated from kernel memory. This buffer facilitates sending scatter gather data between the workstation and FPGA.

A user library provides language bindings for user applications to be able to call into the driver. The user library exposes the software interface described in Section 4.1. When an application makes a call into the user library, the thread enters the kernel driver and initiates a transfer. This is accomplished through the use of the ioctl function on Linux and with DeviceIoControl on Windows. [6]

At runtime, a custom communication protocol is used between the kernel driver and the RX Engine. The protocol is encoded in PCIe payload data and address offset. The protocol consists of single word reads and writes to the FPGA BAR address space. The FPGA communicates with the kernel driver by firing a device interrupt. The driver reads an interrupt status word from the FPGA to identify the conditions of each channel. The conditions communicated include start of a transfer, end of a transfer, and request for scatter gather elements. The protocol is designed to be as lightweight as possible. For example, a write of three words is all that is needed to start a downstream transfer. Once a transfer starts, the only communication between the driver and RIFFA is to provide additional scatter gather elements or signal transfer completion. [6]

2.7.1 Upstream transfers

A sequence diagram for an upstream transfer is shown in Figure 2.4. An upstream transfer is initiated by the FPGA. However, data cannot begin transferring until the user application calls the user library function fpga_recv. Upon doing so, the thread enters the kernel driver and begins the pending upstream request. If the upstream request has not yet been received, the thread waits for it to arrive. The user can set a timeout parametere upon calling the fpga_recv function. On the diagram, the user library and device driver are represented by the single node labeled “RIFFA Library.”

Servicing the request involves building a list of scatter gather elements that identify the pages of physical memory corresponding to the user space byte array. The scatter gather elements are written to a small shared buffer. This buffer location and content length are provided to the FPGA so that it can read the contents. Each page enumerated by the scatter gather list is pinned to memory to avoid costly disk paging. The FPGA reads the scatter gather data, then issues write requests to memory for the upstream data. If more scatter gather elements are needed, the FPGA will request additional elements via an interrupt. Otherwise, the kernel driver waits until all the data is written. The FPGA provides this notification, again via an interrupt.[6]

After the upstream transaction is complete, the driver reads the FPGA for a final count of

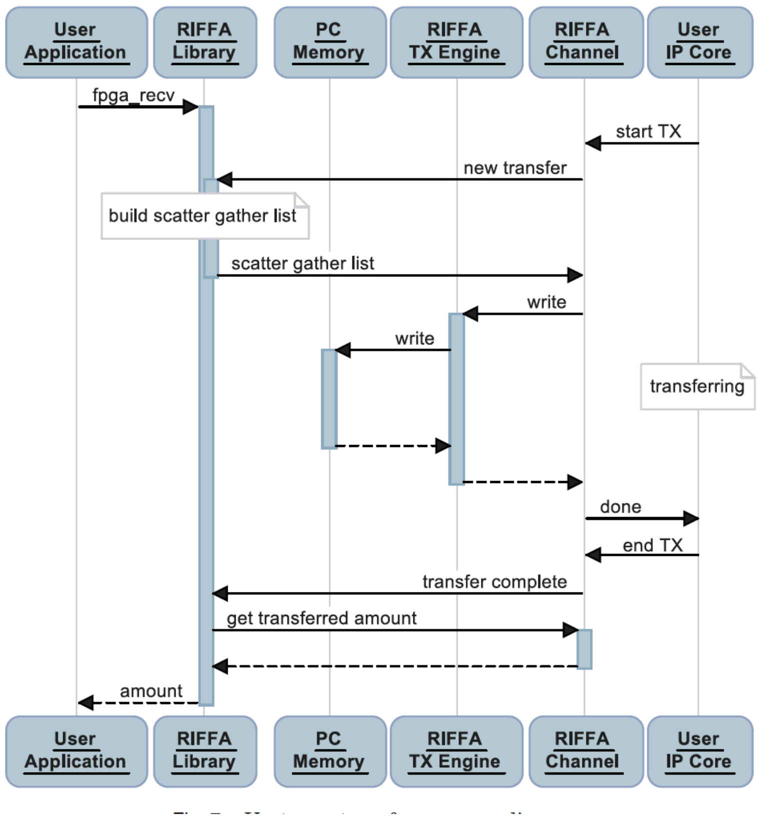


Figure 2.4: Upstream Data Transfer

data words written. This is necessary as the scatter gather elements only provide an upper bound on the amount of data that is to be written. This completes the transfer and the function call returns to the application with the final count. [6]

2.7.2 Downstream transfers

A similar sequence exists for downstream transfers. Figure 2.5 illustrates this sequence. In this direction, the application initiates the transfer by calling the library function fpga_send. The thread enters the kernel driver and writes to the FPGA to initiate the transfer. Again, a scatter gather list is compiled, pages are pinned, and the FPGA reads the scatter gather elements. The elements provide location and length information for FPGA issued read requests. The read requests are serviced and the kernel driver is notified only when more scatter gather elements are needed or when the transfer has completed. [6]

Upon completion, the driver reads the final count read by the FPGA. In error-free operation, this value should always be the length of all the scatter gather elements. This count is returned to the user application. The kernel driver is thread safe and supports multiple threads in multiple transactions simultaneously. For a single channel, an upstream and downstream transaction can be active simultaneously, driven by two different threads. But multiple threads cannot simultaneously attempt a transaction in the same direction. The data transfer will likely fail as both threads attempt to service each other's transfer events.

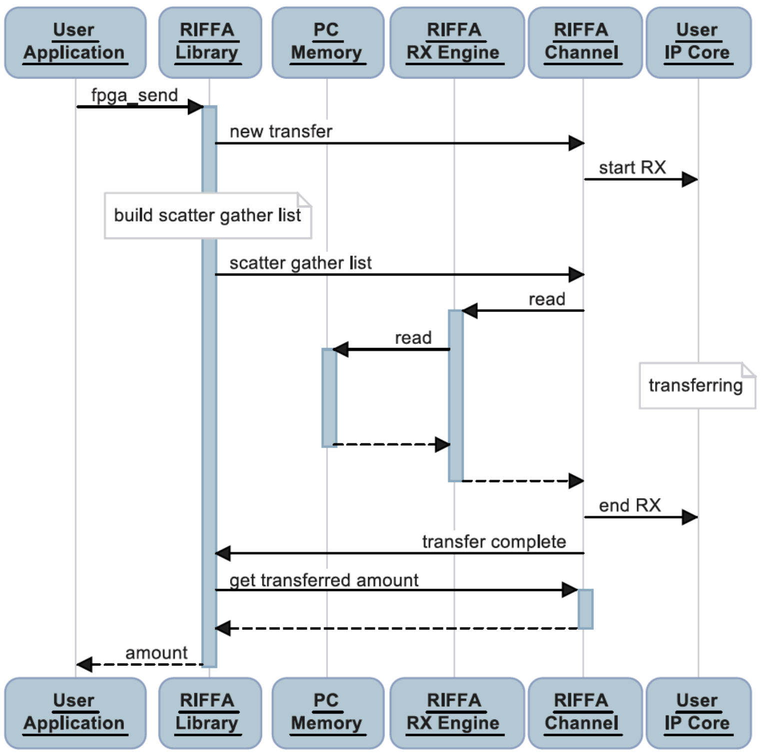


Figure 2.5: Downstream Data Transfer

Chapter 3

FPGA Implemetation of a PCI Express Endpoint

3.1 Xilinx PCI Express IP Core

The 7 Series FPGAs Integrated Block for PCI Express® core is a scalable, high-bandwidth, and reliable serial interconnect building block for use with Xilinx® Zynq®-7000 All Programmable SoC, and 7 series FPGA families. The 7 Series Integrated Block for PCI Express (PCIe®) solution supports 1-lane, 2-lane, 4-lane, and 8-lane Endpoint and Root Port configurations at up to 5 Gb/s (Gen2) speeds, all of which are compliant with the PCI Express Base Specification, rev. 2.1. This solution supports the AMBA® AXI4-Stream interface for the customer user interface. [7]

Although the core is a fully verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. The integrated block follows the PCI Express Base Specification layering model, which consists of the Physical, Data Link, and Transaction layers. The integrated block is compliant with the PCI Express Base Specification, rev. 2.1 [4]. The figure 3.1 illustrates the interfaces to the PCIe Core:

1. System (SYS) interface
2. PCI Express (PCI_EXP) interface
3. Configuration (CFG) interface
4. Transaction interface (AXI4-Stream)
5. Physical Layer Control and Status (PL) interface

The core uses packets to exchange information between the various modules. Packets are formed in the Transaction and Data Link Layers to carry information from the transmitting component to the receiving component. Necessary information is added to the packet being transmitted, which is required to handle the packet at those layers. At the receiving end, each layer of the receiving element processes the incoming packet, strips the relevant information and forwards the packet to the next layer. As a result, the received packets are transformed from their Physical Layer representation to their Data Link Layer and Transaction Layer representations.

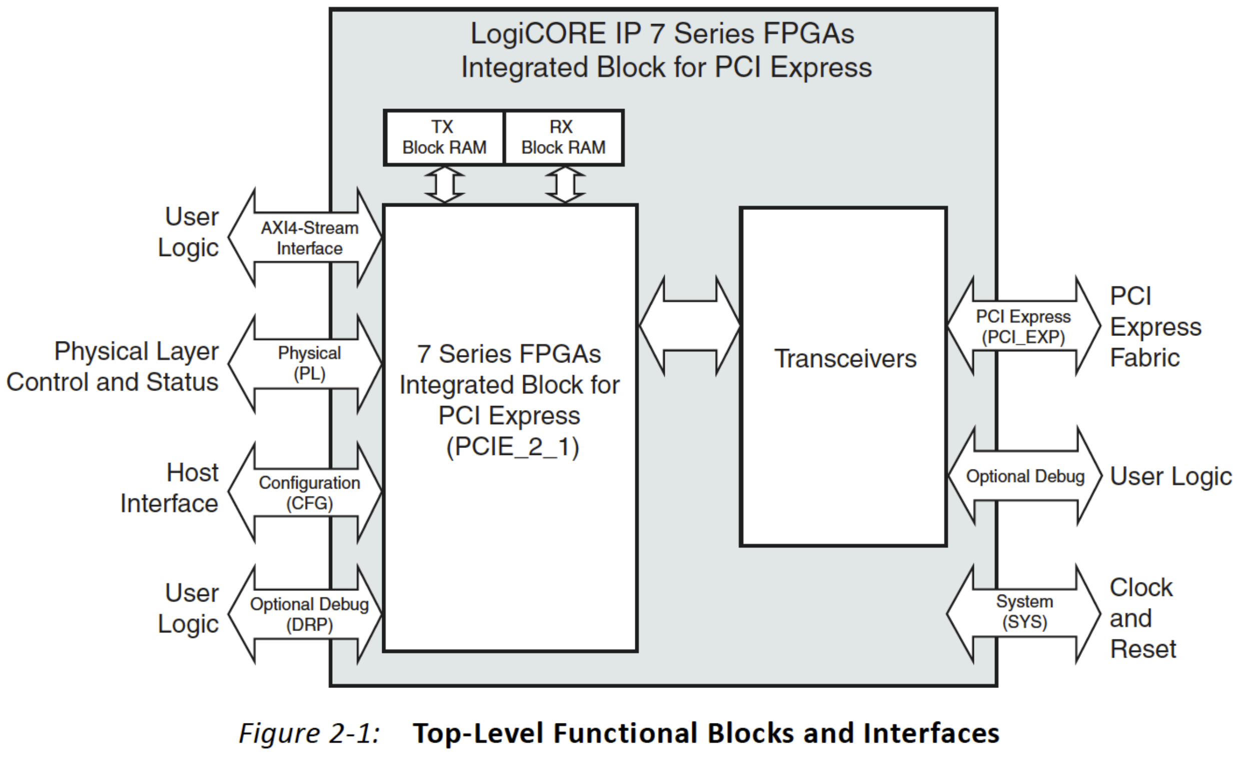


Figure 3.1: Top-Level Functional Blocks and Interfaces

Packets sent to the core for transmission must follow the formatting rules for transaction layer packets (TLPs) as specified in the PCI Express Base Specification. The user application is responsible for ensuring the validity of its packets. The core does not check that a packet is correctly formed and this can result in transferring a malformed TLP. The exact fields of a given TLP vary depending on the type of packet being transmitted. The PCIe Core automatically transmits these types of packets:

1. Completions to a remote device in response to Configuration Space requests.
2. Error-message responses to inbound requests that are malformed or unrecognized by the core.

The user application if responsible for constructing these types of outbound packets:

1. Memory, Atomic Ops, and I/O Requests to remove device.
2. Completions in response to requests to the user application, for example, a memory read request.
3. Completions in response to user-implemented Configuration Space requests, when enabled. These requests include PCI™ legacy capability registers beyond address BFh and PCI Express extended capability registers beyond address 1FFh.

To transmit a TLP, the user application must perform this sequence of events on the transmit transaction interface, which is using the AXI4-Stream interface [7]:

1. The user application logic asserts s_axis_tx_tvalid and presents the first TLP QWORD on s_axis_tx_tdata[63:0]. If the core is asserting s_axis_tx_tready, the QWORD is accepted immediately; otherwise, the user application must keep the QWORD presented until the core asserts s_axis_tx_tready.
2. The user application asserts s_axis_tx_tvalid and presents the remainder of the TLP QWORDs on s_axis_tx_tdata[63:0] for subsequent clock cycles (for which the core asserts s_axis_tx_tready).
3. The user application asserts s_axis_tx_tvalid and s_axis_tx_tlast together with the last QWORD data. If all eight data bytes of the last transfer are valid, they are presented on s_axis_tx_tdata[63:0] and s_axis_tx_tkeep is driven to 0xFF; otherwise, the four remaining data bytes are presented on s_axis_tx_tdata[31:0], and s_axis_tx_tkeep is driven to 0x0F.
4. At the next clock cycle, the user application deasserts s_axis_tx_tvalid to signal the end of valid transfers on s_axis_tx_tdata[63:0].

On the other hand if the user application want to receive inbound packets, it must perform this sequence of events on the receive AXI-4 Stream interface [7]:

1. When the user application is ready to receive data, it asserts m_axis_rx_tready.
2. When the core is ready to transfer data, the core asserts m_axis_rx_tvalid and presents the first complete TLP QWORD on m_axis_rx_tdata[63:0].
3. The core keeps m_axis_rx_tvalid asserted, and presents TLP QWORDs on m_axis_rx_tdata[63:0] on subsequent clock cycles (provided the user application logic asserts m_axis_rx_tready).
4. The core then asserts m_axis_rx_tvalid with m_axis_rx_tlast and presents either the last QWORD on s_axis_tx_tdata[63:0] and a value of 0xFF on m_axis_rx_tkeep or the last DWORD on s_axis_tx_tdata[31:0] and a value of 0x0F on m_axis_rx_tkeep.
5. If no further TLPs are available at the next clock cycle, the core deasserts m_axis_rx_tvalid to signal the end of valid transfers on m_axis_rx_tdata[63:0].

The Xilinx PCIe Core is used in every PCIe designed, the user needs to generate one in order other solutions, like Xillybus [2] and RIFFA [3], to work. Basically the core handles the Data Link Layer and the Physical Layer leaving most of the implementation of the Transaction Layer to the hardware designer.

3.2 RIFFA: Reusable Integration Framework for FPGA Accelerators

3.2.1 RIFFA hardware Architecture

On the FPGA, the RIFFA architecture is a scatter gather bus master DMA design connected to a vendor-specific PCIe Endpoint core, could be the core from Xilinx that we are using or an equivalent core from Altera. The PCIe Endpoint core drives the gigabit transceivers and exposes a bus interface for PCIe formatted packet data. RIFFA cores use this interface to translate between payload data and PCIe packets. A set of RIFFA channels provide read and write asynchronous FIFOs to user cores that deal exclusively with payload data.

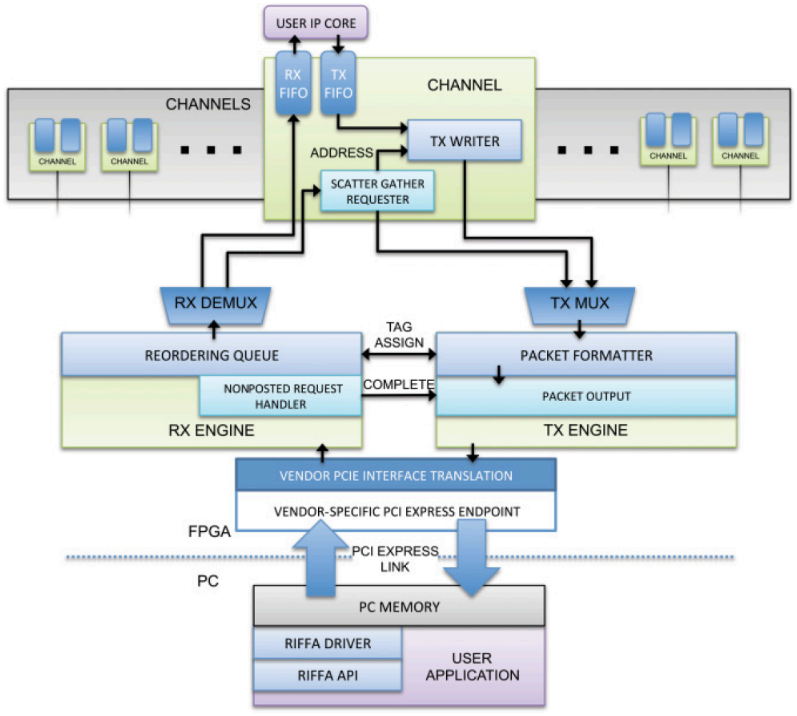


Figure 3.2: RIFFA Hardware Architecture

The RIFFA cores are driven by a clock derived from the PCIe reference clock. This clock's frequency is a product of the PCIe link configuration. It runs fast enough to saturate the PCIe link if data were sent every cycle. User cores do not need to use this clock for their CHNL_TX_CLK or CHNL_RX_CLK. Any clock can be used by the user core. The frequency of the clock is determined by the Xilinx core and is chosen by the configuration of the PCIe Link we are going to support, for example if we create an endpoint for a 8 lane setup we need to use 250 MHz clock.

The PCIe link configuration also determines the width of the PCIe data bus. This width can be 32, 64, or 128 bits wide. Writing a DMA engine that supports multiple widths requires different logic when extracting and formatting PCIe data. For example, with a 32-bit interface, header packets can be generated one 4-byte word per cycle. Only one word can be sent/received per cycle. Therefore, the DMA engine only needs to process one word at a time, containing either header or payload data. However, with a 128-bit interface, a single cycle presents four words per cycle. This may require processing three header packets and the first word of payload in a single cycle. It is possible (and simpler) to design a scatter gather DMA engine that does not perform such advanced and flexible processing. However, the result is a much lowerperforming system that does not take advantage of the underlying link as efficiently. There are many examples of this in research and industry. [6]

Upstream transfers are initiated by the user core via the CHNL_TX_* ports. Data written to the TX FIFO is split into chunks appropriate for individual PCIe write packets. RIFFA will attempt to send the maximum payload per packet. It must also avoid writes that cross physical memory page boundaries, as this is prohibited by the PCIe specification. In order to send the data, the locations in host PC memory need to be retrieved. This comes in the form of scatter gather elements. Each scatter gather element defines a physical memory address and size. These define the memory locations into which the payload data will be written. Therefore, each channel first requests a read of list of scatter gather elements from the host. Once the channel has the scatter gather elements, they issue write packets for each chunk of data. Channels operate independently and share the upstream PCIe direction. The TX Engine provides this multiplexing. [6]

The TX Engine drives the upstream direction of the vendor-specific PCIe Endpoint interface. It multiplexes access to this interface across all channels. Channel requests are serviced in a round-robin fashion. The TX Engine also formats the requests into full PCIe packets and sends them to the vendor-specific PCIe Endpoint. The TX Engine is fully pipelined and can write a new packet every cycle. Throttling on data writes only occurs if the vendorspecific PCIe Endpoint core cannot transmit the data quickly enough. The Endpoint may apply back pressure if it runs out of transmit buffers. As this is a function of the host PC's root complex acknowledgment scheme, it is entirely system dependent. [6]

Downstream transfers are initiated by the host PC via the software APIs and manifest on the CHNL_RX_* ports. Once initiated, the channel cores request scatter gather elements for the data to transfer. Afterward, individual PCIe read requests are made for the data at the scatter gather element locations. Care is also taken to request data so as to not overflow the RX FIFO. Each channel throttles the read request rate to match the rate at which the RX FIFO is draining. Channel requests are serviced by the TX Engine. When the requested data arrives at the vendor Endpoint, it is forwarded to the RX Engine. There the completion packet data is reordered to match the requested order. Payload data is then provided to the channel.

The RX Engine core is connected to the downstream ports on the vendor-specific PCIe Endpoint. It is responsible for extracting data from received PCIe completions and servicing various RIFFA device driver requests. It also demultiplexes the received data to the correct channel. The RX Engine processes incoming packets at line rate. It therefore never blocks the vendor-specific PCIe Endpoint core. Data received by the Endpoint will be processed as soon as it is presented to the RX Engine, avoiding the possibility of running out of buffer space. After extracting payload data, the RX Engine uses a Reordering Queue module to ensure the data is forwarded to the channel in the order it was requested.[6]

Chapter 4

RIFFA Tests and Measurements

4.1 Software Interface

The interface on the software side is consisted by a few functions. Data transfers can be initiated by both sides, PC functions initiate downstream transfers and hardware cores initiate upstream transfers. The function of the RIFFA 2.1 software interface is listed in Table 4.1 (for the C/C++ API). We omit the Java, Python, and Matlab interfaces but they are very similar.

Function	Arguments
fpga_list	fpga_info_list * list
fpga_open	int id
fpga_close	fpga_t * fpga
fpga_send	fpga_t * fpga, int chnl, void * data, int len, int destoff, int last, long long timeout
fpga_recv	fpga_t * fpga, int chnl, void * data, int len, long long timeout
fpga_reset	fpga_t * fpga

Table 4.1: Functions of RIFFA API

There are four primary functions in the API: open, close, send, and receive. The API supports accessing individual FPGAs and individual channels on each FPGA. There is also a function to list the RIFFA-capable FPGAs installed on the system. A reset function is provided that triggers the FPGA channel reset signal. All those function are very well documented on `http://riffa.ucsd.edu/node/10`, a quick summary follows.

The software API has only one function to send data and only one to receive data. This has been intentionally kept as simple as possible.[6] These function calls are synchronous and will block until the transfer has completed and both take arrays as parameters or byte arrays when using Java or any other object-oriented language. The byte arrays contain the data to send or serve as the receptacle for receiving data. In the send data function, the offset parameter is used as a hint for the FPGA. It specifies an offset for storing data at the destination. This can be used to support bulk memory copies between the PC and memory on the FPGA. The last parameter is used to group together multiple transfers. If last is 0, the destination should expect more transfers as part of the same group. The final transfer should have last set to 1. This grouping is entirely user specified and can be useful in situations where memory limitations require sending multiple partial transfers. Lastly, the timeout parameter specifies how many milliseconds to wait between communications during a transfer. Setting this value to an upper bound on computation time will ensure that RIFFA does not return prematurely. Setting a zero timeout value causes the software thread to wait for completion indefinitely.

Because the calls to fpga_send and fpga_recv are blocking, all the data must be transferred to the FPGA via the fpga_send call before the call to fpga_recv can be made. The upstream transfer cannot begin until the call to fpga_recv is made. This arrangement may be a problem if the hardware core attached to the channel is designed to start sending a response before it receives all the data. Many streaming-oriented designs will produce output in this fashion and attempt to start an upstream transaction while the downstream transaction is still running. To avoid a deadlock, users will need to use two threads, one for the call to fpga_send and one for the call to fpga_recv. This allows both calls to execute in a time overlapping manner as the hardware would expect. RIFFA would benefit from an expanded software API that supports programmed I/O and nonblocking function calls. [6]

4.2 Hardware Interface

A single RIFFA channel has two sets of signals, one for receiving data (RX) and one for sending data (TX). RIFFA has simplified the interface to use a minimal handshake and receive/send data using a FIFO with first word fall through semantics (valid+read interface). The clocks used for receiving and sending can be asynchronous from each other and from the PCIe interface (RIFFA clock). The tables 4.2, 4.3 below describes the ports of the interface. The input/output designations are from your our core's perspective. The interface is pretty similar to the AXI-4 interface provided by the Xilinx core, but it has added functionality for the current use case.

Name	Description
CHNL_RX_CLK	Provide the clock signal to read data from the incoming FIFO.
CHNL_RX	Goes high to signal incoming data. Will remain high until all incoming data is written to the FIFO.
CHNL_RX_ACK	Must be pulsed high for at least 1 cycle to acknowledge the incoming data transaction.
CHNL_RX_LAST	High indicates this is the last receive transaction in a sequence.
CHNL_RX_LEN[31:0]	Length of receive transaction in 4 byte words.
CHNL_RX_OFF[30:0]	Offset in 4 byte words indicating where to start storing received data (if applicable in design).
CHNL_RX_DATA[DWIDTH-1:0]	Receive data.
CHNL_RX_DATA_VALID	High if the data on CHNL_RX_DATA is valid.
CHNL_RX_DATA_REN	When high and CHNL_RX_DATA_VALID is high, consumes the data currently available on CHNL_RX_DATA.

Table 4.2: Hardware Interface Receive Ports

Name	Description
CHNL_TX_CLK	Provide the clock signal to write data to the outgoing FIFO.
CHNL_TX	Set high to signal a transaction. Keep high until all outgoing data is written to the FIFO.
CHNL_TX_ACK	Will be pulsed high for at least 1 cycle to acknowledge the transaction.
CHNL_TX_LAST	High indicates this is the last send transaction in a sequence.
CHNL_TX_LEN[31:0]	Length of send transaction in 4 byte words.
CHNL_TX_OFF[30:0]	Offset in 4 byte words indicating where to start storing sent data in the PC thread's receive buffer.
CHNL_TX_DATA[DWIDTH-1:0]	Send data.
CHNL_TX_DATA_VALID	Set high when the data on CHNL_TX_DATA valid. Update when CHNL_TX_DATA is consumed.
CHNL_TX_DATA_REN	When high and CHNL_TX_DATA_VALID is high, consumes the data currently available on CHNL_TX_DATA.

Table 4.3: Hardware Interface Transmit Ports

The timing diagram 4.1 shows the RIFFA channel receiving a data transfer of 16 (4 byte) words (64 bytes). When CHNL_RX is high, CHNL_RX_LAST, CHNL_RX_LEN, and CHNL_RX_OFF will all be valid. In this example, CHNL_RX_LAST is high, indicating to the user core that there are no other transactions following this one and that the user core can start processing the received data as soon as the transaction completes. CHNL_RX_LAST may be set low if multiple transactions will be initiated before the user core should start processing received data. Of course, the user core will always need to read the data as it arrives, even if CHNL_RX_LAST is low.

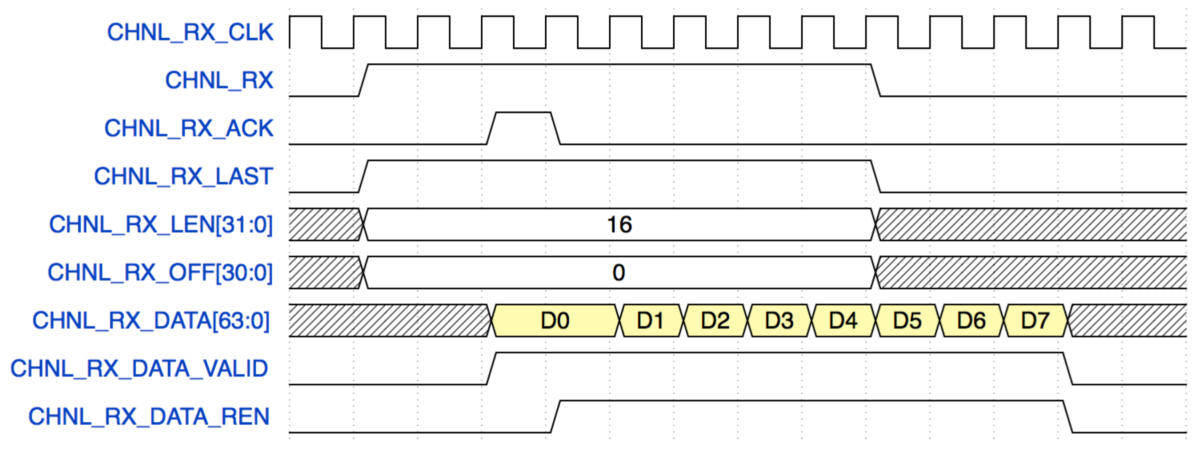


Figure 4.1: Receive Timing Diagram

In the example CHNL_RX_OFF is 0. However, if the PC specified a value for offset when it initiated the send, that value would be present on the CHNL_RX_OFF signal. The 31 least significant bits of the 32 bit integer specified by the PC thread are transmitted. The CHNL_RX_OFF signal is meant to be used in situations where data is transferred in multiple sends and the user core needs to know where to write the data (if, for example it is writing to BRAM or DRAM).

The user core must pulse the CHNL_RX_ACK signal high for at least one cycle to acknowledge the receive transaction. The RIFFA channel will not recognize that the transaction has been received until it receives a CHNL_RX_ACK pulse. The combination of CHNL_RX_DATA_VALID high and CHNL_RX_DATA_REN high consumes the data on CHNL_RX_DATA. New data will be provided until the FIFO is drained. Note that the FIFO may drain completely before all the data has been received. The CHNL_RX signal will remain high until all data for the transaction has been received into the FIFO. Note that CHNL_RX may go low while CHNL_RX_DATA_VALID is still high. That means there is still data in the FIFO to be read by the user core. Attempting to read (asserting CHNL_RX_DATA_REN high) while CHNL_RX_DATA_VALID is low, will have no affect on the FIFO. The user core may want to count the number of words received and compare against the value provided by CHNL_RX_LEN to keep track of how much data is expected. [3]

In the event of a transmission error, the amount of data received may be less than the amount expected (advertised on CHNL_RX_LEN). It is the user core's responsibility to detect this discrepancy if important to the user core.

The diagram 4.2 shows the RIFFA channel sending a data transfer of 16 (4 byte) words (64 bytes). It's nearly symmetric to the receive example. The user core sets CHNL_TX high and asserts values for CHNL_TX_LAST, CHNL_TX_LEN, and CHNL_TX_OFF for the duration CHNL_TX is high. CHNL_TX must remain high until all data has been consumed. RIFFA will expect to read CHNL_TX_LEN words from the user core. Any more data provided may be consumed, but will be discarded. The user core can provide less than CHNL_TX_LEN words and drop CHNL_TX at any point. Dropping CHNL_TX indicates the end of the transaction. Whatever data was consumed before CHNL_TX was dropped will be sent and reported as received to the software thread. [3]

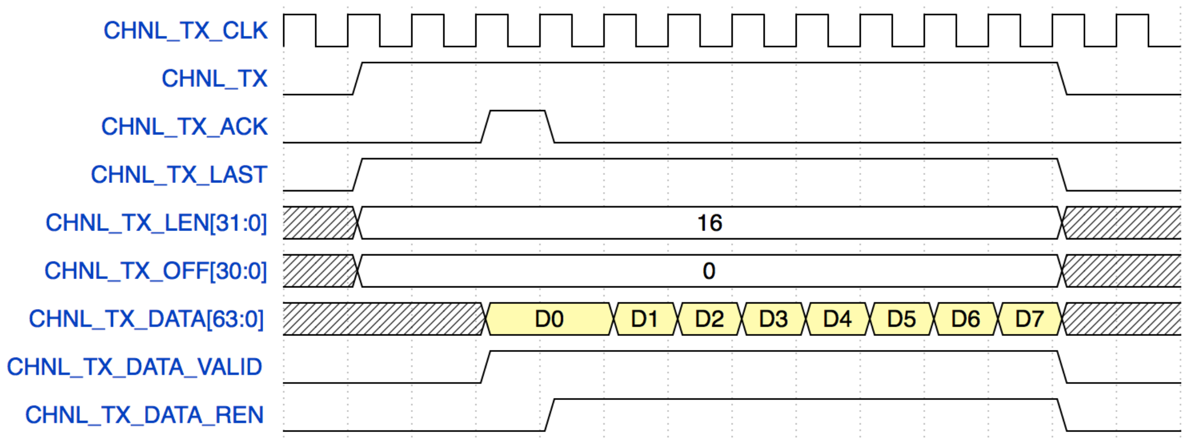


Figure 4.2: Transmit Timing Diagram

As with the receive interface, setting CHNL_TX_LAST high will signal to the PC thread to not wait for additional transactions (after this one). Setting CHNL_TX_OFF will cause the transferred data to be written into the PC thread's buffer starting CHNL_TX_OFF 4 bytes words from the beginning. This can be useful when sending multiple transactions and needing to order them in the PC thread's receive buffer. CHNL_TX_LEN defines the length of the transaction in 4 byte words. As the CHNL_TX_DATA bus can be 32 bits, 64 bits, or 128 bits wide, it may be that the number of 32 bit words the user core wants to transfer is not an even multiple of the bus width. In this case,CHNL_TX_DATA_VALID must be high on the last cycle CHNL_TX_DATA has at least 1 word to send. The channel will only send as many words as is specified by CHNL_TX_LEN. So any additional data consumed, past the last word, will be discarded.

Shortly after CHNL_TX goes high, the RIFFA channel will pulse high the CHNL_TX_ACK and begin to consume the CHNL_TX_DATA bus. The combination of CHNL_TX_DATA_VALID high and CHNL_TX_DATA_REN high will consume the data currently on CHNL_TX_DATA. New data can be consumed every cycle. After all the data is consumed, CHNL_TX can be dropped. Keeping CHNL_TX_DATA_VALID high while CHNL_TX_DATA_REN is low will have no effect.

4.3 Performance and Function Tests

In order to test the performance of the whole implementation we created the required hardware and software. For the tests we used Xilinx's VC707 development board and a Linux PC running Ubuntu 14.04 operating system. The FPGA board was connected to the PC via an extension PCI cable, this setup is optional because someone is able to plug the FPGA board directly to the PC motherboard. The PCI Express Link of our implementation had 8 lanes (x8) with a clock rate of 250 MHz. With this setup we must use the 128 interface with the core by using the 128 RIFFA endpoint. So the words that we are going to create/receive/send in the hardware level are going to be 128-bit wide.

RIFFA comes with some tests included in the source code. On the hardware side they provide CHNL_tester, a verilog module that first receives an incremental sequence of numbers and then creates the same sequence sending the data back, it doesn't save the received data to a ram. We created three test scenarios in C that suits our goals and are good for the evaluation. Finally we created a JAVA application similar to the bandwidth test program in C, in order to check JAVA functionality along with some problems that arise in an object oriented language.

4.3.1 Bandwidth test

First we needed to get a measure of the performance of the system, so we created a bandwidth test with more real functions than the test program from RIFFA had. The C application send an increment sequence of integers to the FPGA, the FPGA stores those integers to a distributed ram and then reads them back sending them back to the application. The application is measuring the running time of the function calls fpga_send and fpga_recv, calculating the bandwidth performance by how many words where sent/received in the time frame.

The design is straightforward we allocate two buffers of 32-bit integers, the send_buffer and the recv_buffer. After open the FPGA by calling fpga_open we initialize the data in the send_buffer and then we send them to the FPGA using fpga_send. After receiving the data back we implement a quick check of the data and we calculating the performance. We need to close the FPGA by calling fpga_close, either at the end of the program where we also deallocate the buffers, or after receiving the data as we no longer need the fpga, but is crucial not to forget this.

On the hardware side we created a distributed ram with the Xilinx distributed memory generation. We choose to use a distributed ram instead of a block ram in order to measure the peak performance of the system. When we receive the data we place them in the ram.

This approach is more realistic and accurate than the example of RIFFA, which is recreating a copy of the received sequence. Since we are accessing a ram we can have a rough estimate of the execution time by multiplying the number of elements in the ram with the clock period. In the next state of the FSM we send to the core the length of the upcoming TX transaction and then we change to the state that actually sends the data. In our tests the first 4 32-bit words, or differently the first transaction of one 128-bit word, was always corrupted, we were receiving the 4 last words sent to the FPGA, we concluded that was a small problem probably inherited by RIFFA or Xilinx core. The fix was quite easy, as we are able to send 1 more word in every tx transaction. The overhead that is imposed by this workaround is negligible.

We did two experiments regarding the number of words we were sending to the FPGA. In the first one we were sending 160 words to the FPGA, and on the other only 1 word. Those test derive from the two alternatives we have in the implementation of the interface with the agent of the Data Center. The SDN schedule of the Data Center communication can by transmitted at once for a scheduling period or the command for each slot can be transmitted separately. The amount of data for each scheduling period are similar to size with 160 32-bit words, that is the reason we choose the test this transaction, for the second experiment we send only one word so we actually testing the latency of the PCIe link, something that may be a problem if the transmittance of the schedule is decided to be divided in small commands.

The bandwidth we measured for 160 word transaction was 20,83 MB/s for the send function and 21,56 MB/s for the receive function. The running times were 0,032 ms and 0,043 ms respectively. The one word transaction had a bandwidth of 0,1 MB/s for the send function and 0,3 MB/s for the receive function, the running times were 0,039 ms 0,06 ms respectively. All the values mentioned are median values, the deviation wasn't much and is caused by other devices using the PCI bus. Our results are the same with the bandwidth values mention on the RIFFA cite and paper [3], [6], we present here a graph in figure 4.3 about the performance of RIFFA. We didn't test for higher amounts of data as it is not relevant to this thesis, but as we see on the graph the performance peaks for large transfers, something that is expected of the PCI Express interface.

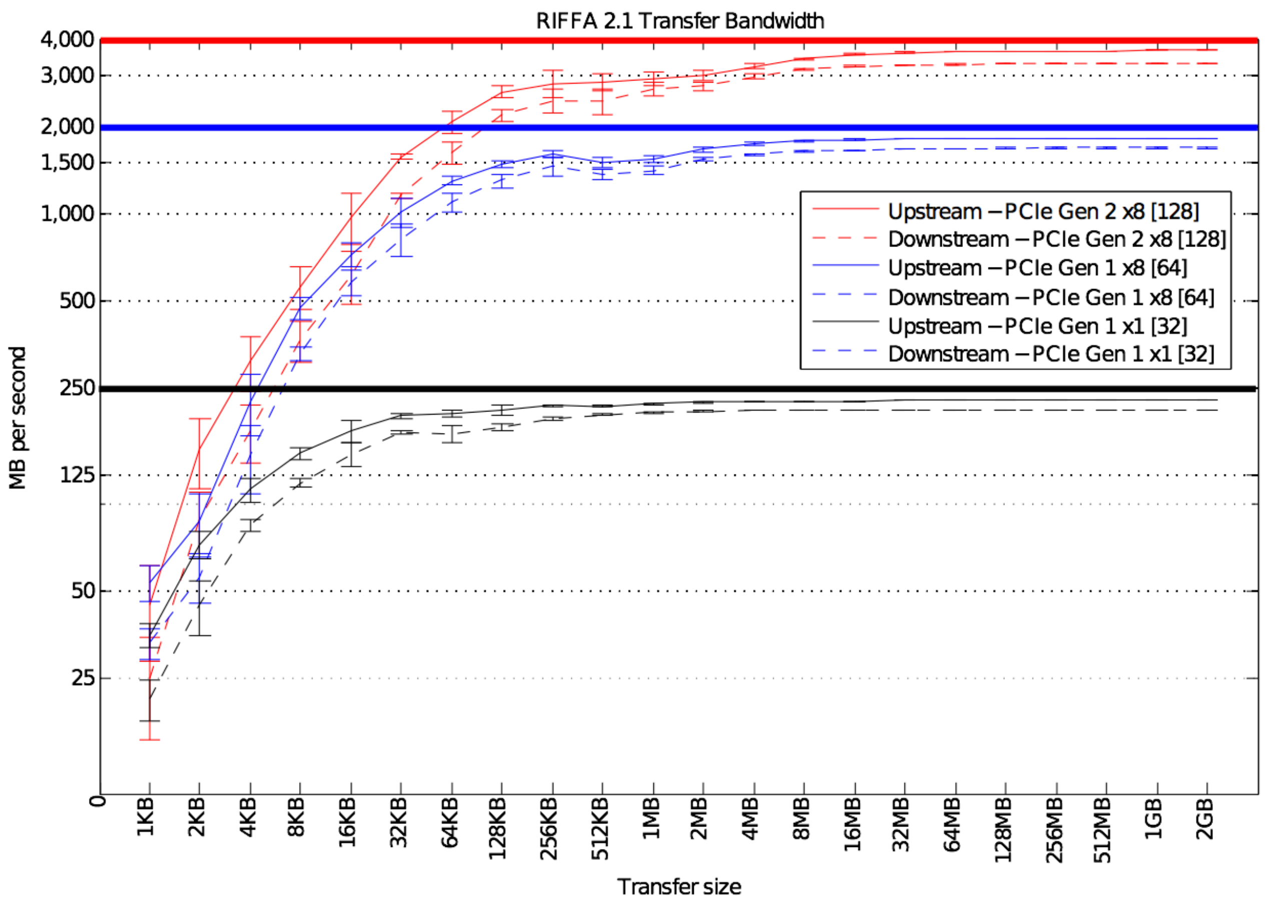


Figure 4.3: RIFFA bandwidth

4.3.2 FPGA starts the transaction test

Another function that was not straightforwardly mention was whether the FPGA could start a send transaction on its own, without receiving some command from the pc host. This functionality should be tested because there are scenarios which include buffer monitoring, were all the FPGA's in the Data Center are sending information about their buffers. So we created a program that only receives data from the FPGA an FSM that starts the transaction by sending some data to the pc. In this test we weren't measuring the time of the transaction because there is no reason that the results will differ from the previous tests. As we found out the FPGA is possible to start a transaction with the pc.

The transaction is triggered by a button on the FPGA board. When the button is pressed the send FSM is reseted and initiates a transaction by asserting CHNL_TX and giving value to CHNL_TX_LEN, if the interface responds by asserting CHNL_TX_ACK the transaction can begin. In order to receive the data to the host pc we need to run the C application before triggering the procedure on the FPGA, if we haven't started the program then we would not receive any data on the pc.

4.3.3 JAVA application

Up to this point we used C in order to check the functionalities and the performance of the interface, but the Data Center agent/scheduler is probably going to be implemented in JAVA.

The final test we done was to create a JAVA application similar to the C program created for the bandwidth test.

JAVA is an object oriented language in which everything is an object. The data we are going to send to the FPGA cannot reside in an object because we can't be sure of each structure. On of the advantages of object orientated language is that objects come along with the methods that operate on their data, in C++ for example each object at the beginning of the data has a table with VPTR pointers, that point to the location of the object's method's code. The location of the pointers is at the beginning in order to avoid a buffer overflow attack and alternating those pointers to the attackers code location in memory. In order to send correctly the data to the FPGA we need to create a JAVA ByteBuffer.

The JAVA ByteBuffer [8] is a class that operates on an a buffer similar to C. We need first to allocate the buffer in the heap memory and then we access is by index (pointer). Reading and writing to these buffers happen by transaction of bytes, but it is possible to handle them as integer buffers. By calling the method asIntBuffer on the allocated object ByteBuffer, we can handle the memory locations integers, we can now access the buffer with a put function, specifying the index and the integer value. On the opposite side when we receive the data from the FPGA we need to access the buffer as an IntBuffer the same way we did before. Then we are able to save data from inside the buffer to JAVA variable/objects. It should be noted that the aforementioned procedure can be implemented for any type of JAVA variable whether be it integer, float, double, char, etc.

Another problem we faced in the JAVA implementation was the endianess of the data. The JVM represents the data in big endian byte order, instead of little endian which is used in INTEL processors and XILINX FPGAs. It is very important to understand that the difference in endianess is in bytes and not in bits. When someone states that the data is in big endian means that the ordering of the bits starts with the most significant big and other bits follow in decreasing order, on the contrary little endian words start with the least significant bit first and the rest of the bit follow is increasing order. But in JVM big endian is used in the order of the bytes in each word, where each byte is represented in little endian. The solution to this problem is going to be fixed in the FPGA where in only one clock period we can reverse the order of the bytes with a simple concatenation operation. We need to send data to the JAVA application in the same way. Off course this opereation can be coded in the application but with the performance of the whole system in mind it would be faster to implement the reverse functionality on the FPGA.

The running time of the JAVA application where quite similar to the C program both send and receive function took 0,05 ms the deviations were very low. We also measured a mean bandwidth of 11,91 MB/s and 17,76 MB/s when sending and receiving 160 32-bit integers respectively. The same experiment was done for 20 integers and the bandwidth was 1,98 MB/s for sending the data and 3,05 MB/s for receiving them back. The bandwidth is in the same order of magnitude with the C program, but as we see the time of execution differs by an order of 2 in favor of the C program. In order to present a more accurate comparison between the two applications we need to send a larger amount of data.

Chapter 5

Conclusion and Future Work

The main goal of this thesis was to present to the reader the fundamental framework and the base knowledge that needs to work with the PCI Express interconnect in a FPGA accelerator use case. We presented the foundation that a programmer needs to create a PCI driver, but also some advance techniques that state of the art drivers use. Finally we demonstrated the RIFFA interconnection with detailed information on how a user can take advantage of this capabilities and include RIFFA in their designs. A master student can read this thesis and with very little effort he can start using the PCIe interconnect in his designs, and this is the important achievement of this thesis.

RIFFA proved to be one of the best implementations of PCIe endpoints, that is open-source and free to use, and in comparison with intellectual property implementations is more advanced and uses state of the art techniques in it's code. The bandwidth measured is pretty close to the peak limit of the PCIe link, especially for big transfers. The hardware interface is straightforward for most engineers as it is similar to the AMBA AXI-4 interface, that is used mostly on every XILINX core. The software interface supports most of the popular programming languages and is easy to use, the users might need to have some low level programming experience in order to debug and understand how to software is working.

Despite the fact the RIFFA is an exceptional solution some things need improvement but they belong to future research. For instance the driver must use the modern way of pci probing in order to enable hot plug support, a feature that is essential to the ease of use of the designs. Further customization can be added to the designs by tweaking the code, and users that try to implement their own hardware PCIe endpoint or a Linux PCI driver can use the RIFFA code as a reference or as a solution for the other side if they want to focus only on hardware or software.

Bibliography

[1] Nephele horizon 2020. http://www.nepheleproject.eu.

[2] Xillybus. http://xillybus.com.

[3] Riffa: Reusable integration framework for fpga accelerators. http://riffa.ucsd.edu.

[4] PCI SIG. *PCI Express® Base Specification Revision 2.1*. PCI SIG, 2009.

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[6] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 8(4):22:1–22:23, September 2015.

[7] XILINX. *7 Series FPGAs Integrated Block for PCI Express v3.2 LogiCORE IP Product Guide*, 2015.

[8] Class bytebuffer. https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.

[9] Eli Billauer. The anatomy of a pci/pci express kernel driver. http://haifux.org/lectures/256/.

[10] XILINX. *VC707 PCIe Design Creation*, November 2014.

Appendix A

Bandwidth Test Code

A.1 FPGA Verilog code

```
`timescale 1ns/1ns

module chnl_tester #(
  parameter C_PCI_DATA_WIDTH = 9'd128
)
(
  input CLK,
  input RST,
  output CHNL_RX_CLK,
  input CHNL_RX,
  output CHNL_RX_ACK,
  input CHNL_RX_LAST,
  input [31:0] CHNL_RX_LEN,
  input [30:0] CHNL_RX_OFF,
  input [C_PCI_DATA_WIDTH-1:0] CHNL_RX_DATA,
  input CHNL_RX_DATA_VALID,
  output CHNL_RX_DATA_REN,

  output CHNL_TX_CLK,
  output CHNL_TX,
  input CHNL_TX_ACK,
  output CHNL_TX_LAST,
  output [31:0] CHNL_TX_LEN,
  output [30:0] CHNL_TX_OFF,
  output [C_PCI_DATA_WIDTH-1:0] CHNL_TX_DATA,
  output CHNL_TX_DATA_VALID,
  input CHNL_TX_DATA_REN
);


reg [C_PCI_DATA_WIDTH-1:0] rData={C_PCI_DATA_WIDTH{1'b0}};
reg [31:0] rLen=0;
reg [31:0] rCount=0;
reg [1:0] rState=0;
reg [31:0] rLen_received=0;
wire [127:0] data_in;
wire [127:0] data_out;
reg [13:0] addressA = 0;
reg [13:0] addressB = 0;
reg enableA;
reg enableB;
reg write_enable;
reg [5:0] address = 0;
integer i;

reg [19:0] counter = 0;


wire [127:0] data_in1,data_in2;


dist_mem_gen_0 your_instance_name (
  .a(address),         // input wire [5 : 0] a
  .d(CHNL_RX_DATA),         // input wire [127 : 0] d
  .clk(CLK),     // input wire clk
  .we(CHNL_RX_DATA_VALID),       // input wire we
  .spo(data_out)  // output wire [127 : 0] qspo
);


assign CHNL_RX_CLK = CLK;
assign CHNL_RX_ACK = (rState == 2'd1);
assign CHNL_RX_DATA_REN = (rState == 2'd1);

assign CHNL_TX_CLK = CLK;
assign CHNL_TX = (rState == 2'd3);
assign CHNL_TX_LAST = 1'd1;
assign CHNL_TX_LEN = rLen_received; // in words
assign CHNL_TX_OFF = 0;
assign CHNL_TX_DATA = rData;
assign CHNL_TX_DATA_VALID = (rState == 2'd3);


always @(posedge CLK or posedge RST) begin
  if (RST) begin
    rLen <=  0;
    rCount <=  0;
    rState <=  0;
    rData <=  0;
  end
  else begin
    case (rState)

    2'd0: begin // Wait for start of RX, save length
      if (CHNL_RX) begin
        rLen <= CHNL_RX_LEN;
        rCount <=  0;
        rState <=  2'd1;
        enableA <= 1;
        address <= 6'd0;
      end
    end

    2'd1: begin // Wait for last data in RX, save value
      if (CHNL_RX_DATA_VALID) begin
        rCount <=  rCount + (C_PCI_DATA_WIDTH/32);
        address <= address + 6'd1;
      end
      if (rCount >= rLen) begin
        rState <=  2'd2;
        write_enable <= 0;
        enableA <= 0;
      end
    end
    2'd2: begin // Prepare for TX
        enableA <= 1;
        address <= 6'd0;
        rCount <=  (C_PCI_DATA_WIDTH/32);
        rState <=  2'd3;
        rLen_received <= rLen + 32'd4;
    end

    2'd3: begin // Start TX with save length and data value
      if (CHNL_TX_DATA_REN & CHNL_TX_DATA_VALID) begin
        rData <= data_out;
        address <= address + 6'd1;
        rCount <=  rCount + (C_PCI_DATA_WIDTH/32);
        if (rCount >= rLen_received 32'd1) begin
          rState <= 2'd0;
          enableB <= 0;
        end
      end
    end

    endcase
  end
end

(* KEEP = "true" *) reg [C_PCI_DATA_WIDTH-1:0] ila_rx_data;
(* KEEP = "true" *) reg ila_rx_valid;

always @(posedge CLK) begin
        ila_rx_data <= CHNL_RX_DATA;
        ila_rx_valid <= CHNL_RX_DATA_VALID;
end

endmodule
```

A.2 PC C code

```
#include <stdlib.h>
#include <stdio.h>
#include "riffa.h"
#include <stdint.h>
#include "timer.h"

int main(int argc, char** argv) {

  fpga_t * fpga;
  int id, chnl, recvd, sent;
  size_t numWords;
  int i;
  int32_t * recvBuffer;
  int32_t  * sendBuffer;

  id = 0;
  numWords = 160;
  chnl = 0;

    GET_TIME_INIT(3);

  // Get the device with id
  fpga = fpga_open(id);
  if (fpga == NULL) {
    printf("Could not get FPGA %d\n", id);
    return -1;
  }
  // Malloc the arrays
  sendBuffer = (int32_t *)malloc(sizeof(int32_t)*numWords);
  recvBuffer = (int32_t *)malloc(sizeof(int32_t)*(numWords+4));
  if (recvBuffer == NULL) {
    printf("Could not malloc memory for recvBuffer\n");
    fpga_close(fpga);
    return -1;
      }
    // Initialize the buffers
  for (i = 0; i < numWords; i++) {
    recvBuffer[i] = 0;
    sendBuffer[i] = i;
  }

  // Send the data
    GET_TIME_VAL(0);
  sent = fpga_send(fpga, chnl, sendBuffer, numWords, 0, 1, 25000);
  printf("words sent: %d\n", sent);
    GET_TIME_VAL(1);
  // Recv the data
  recvd = fpga_recv(fpga, chnl, recvBuffer, numWords, 25000);
  printf("words recv: %d\n", recvd);
    GET_TIME_VAL(2);
  // Done with device
    fpga_close(fpga);
    // Display the data
  // for (i = 0; i < numWords; i++) {
  //  printf("recvBuffer[%d]: %d\n", i, recvBuffer[i]);
  // }
  // Check the data
  if (recvd != 0) {
    for (i = 4; i < recvd+4; i++) {
      if (recvBuffer[i] != i){
        printf("recvBuffer[%d]: %d, expected %d\n", i, recvBuffer[i], sendBuffer
   [i-4]);
      }
    }
        printf("send bw: %f MB/s %fms\n",sent*4.0/1024/1024/((TIME_VAL_TO_MS(1)
   - TIME_VAL_TO_MS(0))/1000.0),(TIME_VAL_TO_MS(1) - TIME_VAL_TO_MS(0)) );

        printf("recv bw: %f MB/s %fms\n",recvd*4.0/1024/1024/((TIME_VAL_TO_MS(2)
    - TIME_VAL_TO_MS(1))/1000.0),(TIME_VAL_TO_MS(2) - TIME_VAL_TO_MS(1)) );
  }
  free(sendBuffer);
  free(recvBuffer);
    return 0;
}
```

Appendix B

FPGA starts the transaction test

B.1 FPGA Verilog code

```
`timescale 1ns/1ns

module chnl_tester #(
  parameter C_PCI_DATA_WIDTH = 9'd32
)
(
  input CLK,
  input RST,
  output CHNL_RX_CLK,
  input CHNL_RX,
  output CHNL_RX_ACK,
  input CHNL_RX_LAST,
  input [31:0] CHNL_RX_LEN,
  input [30:0] CHNL_RX_OFF,
  input [C_PCI_DATA_WIDTH-1:0] CHNL_RX_DATA,
  input CHNL_RX_DATA_VALID,
  output CHNL_RX_DATA_REN,

  output CHNL_TX_CLK,
  output CHNL_TX,
  input CHNL_TX_ACK,
  output CHNL_TX_LAST,
  output [31:0] CHNL_TX_LEN,
  output [30:0] CHNL_TX_OFF,
  output [C_PCI_DATA_WIDTH-1:0] CHNL_TX_DATA,
  output CHNL_TX_DATA_VALID,
  input CHNL_TX_DATA_REN,
  input manual_reset
);

reg [C_PCI_DATA_WIDTH-1:0] rData={C_PCI_DATA_WIDTH{1'b0}};
reg [31:0] rLen=32'd10000;
reg [31:0] rCount=0;
reg [1:0] rState=0;

assign CHNL_RX_CLK = CLK;
assign CHNL_RX_ACK = (rState == 2'd1);
assign CHNL_RX_DATA_REN = (rState == 2'd1);

assign CHNL_TX_CLK = CLK;
assign CHNL_TX = (rState == 2'd1);
assign CHNL_TX_LAST = 1'd1;
assign CHNL_TX_LEN = rLen; // in words
assign CHNL_TX_OFF = 0;
assign CHNL_TX_DATA = rData;
assign CHNL_TX_DATA_VALID = (rState == 2'd1);

always @(posedge CLK or posedge RST) begin
  if (RST | manual_reset) begin
    rLen <= #1 0;
    rCount <= #1 0;
    rState <= #1 0;
    rData <= #1 0;
  end
  else begin
    case (rState)

    2'd0: begin
      rCount <= #1 (C_PCI_DATA_WIDTH/32);
      rState <= 2'd1;
    end
    2'd1: begin
      if ( CHNL_TX_ACK ) begin
        rState <= 2'd2;
      end
    end
    2'd2: begin
      while (CHNL_TX_DATA_REN==1 && CHNL_TX_DATA_VALID==1 && rCount < rLen )
   begin
        rData <= #1 {rCount + 4, rCount + 3, rCount + 2, rCount + 1};
        rCount <= #1 rCount + (C_PCI_DATA_WIDTH/32);
      end
    end
    default: $display("Error in SEL");
    endcase
  end
end
endmodule
```

B.2 PC C code

```
#include <stdlib.h>
#include <stdio.h>
#include "riffa.h"

int main(int argc, char** argv) {

  fpga_t * fpga;
  int id,chnl,recvd;
  size_t numWords;
  int i;
  unsigned int *recvBuffer;

  id = 0;
  numWords = 16;
  chnl = 0;

  // Get the device with id
  fpga = fpga_open(id);
  if (fpga == NULL) {
    printf("Could not get FPGA %d\n", id);
    return -1;
  }
  // Malloc the receive buffer
  recvBuffer = (unsigned int *)malloc(numWords<<2);
  if (recvBuffer == NULL) {
    printf("Could not malloc memory for recvBuffer\n");
    fpga_close(fpga);
    return -1;
      }
    // Initialize the buffer
  for (i = 0; i < numWords; i++) {
    recvBuffer[i] = 0;
  }
  // Recv the data
  recvd = fpga_recv(fpga, chnl, recvBuffer, numWords, 25000);
  printf("words recv: %d\n", recvd);
  // Done with device
    fpga_close(fpga);
    // Display the data
  for (i = 0; i < numWords; i++) {
    printf("recvBuffer[%d]: %d\n", i, recvBuffer[i]);
  }
  // Check the data
  if (recvd != 0) {
    for (i = 4; i < recvd; i++) {
      if (recvBuffer[i] != i+1){
        printf("recvBuffer[%d]: %d, expected %d\n", i, recvBuffer[i], i+1);
      }
    }
  }
    return 0;
}
```

Appendix C

JAVA Application

C.1 JAVA Code

```
import edu.ucsd.cs.riffa.*;
import java.nio.ByteBuffer;

public class pc_starts {

        //private static final int numWords = 20;
        public static void main(String[] args) {
                int fid = 0;
                int channel = 0;
                int numWords = 160;
                int i, sent, recvd = 0;
                byte tmp0, tmp1;  //example byte declaration
                int temp1;

                //Get the device with id
                Fpga fpga = Fpga.open(fid);
                if(fpga == null){
                        System.out.println("Could not malloc memory for
   recvBuffer");
                        fpga.close();
                }

                //fpga.reset();

                // Malloc the arrays
                ByteBuffer sendbuf = ByteBuffer.allocateDirect(numWords*4);  //
   // *4, needs the number of bytes
                java.nio.IntBuffer intBufSend = sendbuf.asIntBuffer();
    //// Creates a view of this byte buffer as an int buffer.
                ByteBuffer recvbuf = ByteBuffer.allocateDirect(numWords*4+16);
   //// *4, needs the number of bytes--- +16 because the first 4 words(=16bytes)
    received from RIFFA are not actually valid
                java.nio.IntBuffer intBufRecv = recvbuf.asIntBuffer();       //
   // Creates a view of this byte buffer as an int buffer.

                if(sendbuf == null){
                        System.out.println("Could not malloc memory for
   sendBuffer");
                }
                if(recvbuf == null){
```

```
                                System.out.println("Could not malloc memory for
recvBuffer");
                        }

                        System.out.println("Data from PC to FPGA");
                        // initializa the data
                        for(i = 0; i< numWords; i++){
                                //sendbuf.putInt(i, i); //(index, value)
                                ///// Put an integer in the "integer view buffer"
intBufsend,but the integer is in BIG ENDIAN ////////
                                intBufSend.put(i, i+1);
                                System.out.printf("%d - %02x%02x%02x%02x\n", i+1,
                                sendbuf.get((i*4)+0), sendbuf.get((i*4)+1), sendbuf.get
((i*4)+2), sendbuf.get((i*4)+3));  // Prints the buffer in LITTLE ENDIAN
                        }

                        for(i = 0; i< numWords; i++){
                                recvbuf.putInt(i,0);
                        }

                        long startTime0 = System.nanoTime();
                        // sending data
                        sent = fpga.send(channel, sendbuf, numWords, 0, true, 0);
                        long endTime0 = System.nanoTime();
                        long duration0 = endTime0 - startTime0;
                        System.out.println("words sent:"+sent);

                        long startTime1 = System.nanoTime();
                        // receiving data
                        recvd = fpga.recv(channel, recvbuf, 0);
                        long endTime1 = System.nanoTime();
                        long duration1 = endTime1 - startTime1;
                        System.out.println("words recv:"+recvd);

                        //Done with the device
                        fpga.close();

                        System.out.println("Data from FPGA to PC");
                        for (i=0; i < numWords+4; i++){
                                        //System.out.printf("%d) - %02x%02x%02x%02x\n",
i+1,
                                        //recvbuf.get((i*4)+3), recvbuf.get((i*4)+2),
recvbuf.get((i*4)+1), recvbuf.get((i*4)+0)); //Prints the received buffer, as
 it was send in BIG ENDIAN
                                        if(i>3) {
                                                System.out.println((i-3) + ")   " +
intBufRecv.get(i)); // Prints the recv buffer as integer(in java);
                                                temp1 = intBufRecv.get(i); // example of
 how to transfers one word from the buffer to a variable
                                        }
                                }

                        System.out.printf("send bw: %f MB/s %fms\n",sent*4.0/1024/1024/(
duration0/1000000000.0),(duration0/1000000.0) );

                        System.out.printf("recv bw: %f MB/s %fms\n",recvd*
4.0/1024/1024/(duration1/1000000000.0),(duration1/1000000.0) );

                }
}
```