Bus Mastering PCI Express In An FPGA

Ray Bittner
Microsoft Research
One Microsoft Way
Redmond, WA 98052
1-425-703-9605
raybit@microsoft.com

ABSTRACT

This paper describes a bus mastering implementation of the PCI Express protocol using a Xilinx FPGA. While the theoretical peak performance of PCI Express is quite high, attaining that performance is a complex endeavor on top of an already complex protocol. The implementation is described and its performance is analyzed. Source code is offered for free download via the web.

Categories and Subject Descriptors

B.4.3 [**Interconnections (Subsystems)**] – Interfaces

General Terms

Design, Performance

Keywords

FPGA, PCI Express, PCIe, Bus Mastering, Design, Performance

1. INTRODUCTION

The PCI Express (PCIe) protocol has been prevalent in the PC industry for a few years, and the cores to implement it in FPGAs have been available for nearly as long. Offering raw bit rates of 2.5 GBit/Sec to 20 GBit/Sec to the FPGA, PCIe is the highest bandwidth interface available using PC-like platforms [3]. While major FPGA companies offer PCI Express implementations [1][2], the cores stop short of providing the Transaction Layer and leave that as an exercise to the user. This is not such a bad thing, since the Transaction Layer really defines the type of device that is being implemented and how it will behave; however, its implementation is not trivial. While it is not overly difficult to develop a programmed I/O Transaction Layer interface, such an implementation will not even come close to providing the full bandwidth that is available from PCI Express. In order to achieve higher bandwidth, a bus mastering interface is required, and the implementation of that interface is much more complex. This paper describes a real world bus mastering implementation and provides the associated Verilog source code with a Microsoft Windows WDM Driver and testing application for general use. In addition, the design is analyzed with an eye towards improvements.

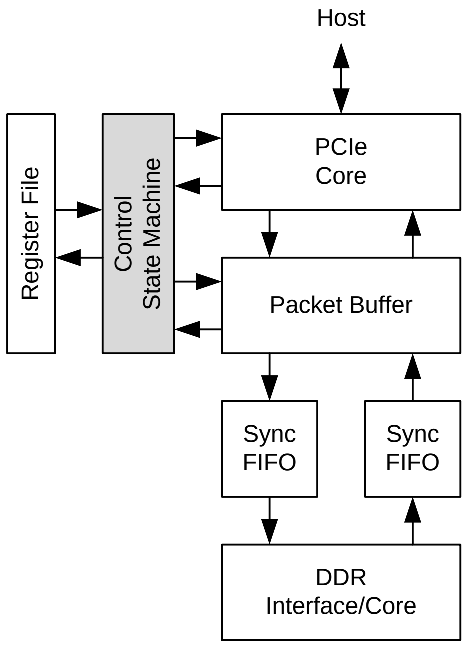

Figure 1. Design Overview

2. THE DESIGN

The test design is an interface between a PC host and DDR memory as shown in Figure 1, where the bulk of the design effort is involved in creating the shaded Control State Machine block. Of course, using this design it would be possible to easily interface other types of devices by treating the DDR interface as an address/data bus to which multiple targets could be attached. From the software point of view, the driver supports the Windows standard CreateFile(), ReadFile() and WriteFile() API to read and write the memory from C source code. The file pointer position is used to address DDR memory base addresses local to the development board. As this is a bus mastering design, all references to PCIe reads and writes below are from the perspective of the FPGA. That is, reads transfer data from the host to the FPGA and writes transfer data from the FPGA to the host.

2.1 DESIGN GOALS

At the PCIe level, the design supports bus mastering as an initiator; meaning that it is capable of initiating reads or writes onto the PCIe bus independent of the CPU or the system DMA controller. This is advantageous because it implies that the design can operate in a standalone system that may or may not have all of the hardware support amenities of a PC-like host. It also implies that the design could operate autonomously on the bus without the performance reducing effects of depending on software intervention from the host. However, it should be noted that while this version does act as a bus mastering initiator, the current implementation does wait for the host to use programmed I/O to set source and destination addresses as well as a length parameter for the transfer. All of the protocol machinery for independent operation is present in the Verilog, but this implementation internally polls the aforementioned registers as set by the host before acting.

2.2 DESIGN OVERVIEW

As illustrated in Figure 1, the PCIe core and the DDR core are supplied from the FPGA manufacturer; the rest of the blocks are user code as given by this design. The two synchronization FIFOs shown are used to cross the clock domains between the PCIe core sourced clock and the DDR core sourced clock. Everything above the FIFOs operates based on the clock dictated by the PCIe core, and everything below the FIFOs operates based on the clock given by the DDR core. The FIFOs are implemented using distributed RAM since there is no need for a large buffer, but block RAMs could have been used in order to save some slice resources.

The Packet Buffer shown is a block RAM that is used both to download received packets from the core and is also used as a buffer to assemble transmit packets for upload. Lastly, the Register File uses a block RAM to provide a set of control registers that are visible to both the Control State Machine and on the PCIe bus as a prefetchable memory range. The Register File is a 2K Byte range in the prefetchable memory space and is the only address range that is directly accessible from PCIe. The DDR memory is managed as an independent address range local to the card via DMA operations.

The PCIe core itself provides a 64-bit bi-directional data path that is used for packet transmit and receive. The operating frequency of the interface to the core is determined by the number of lanes that are negotiated with the PCIe link. This design runs at 62.5 MHz using the 1 lane (x1) configuration on the Xilinx ML505 development board with the Xilinx Endpoint Block Plus core.

2.3 CONTROL STATE MACHINE

The main loop of the state machine runs down the left side of Figure 2. A number of conditions are polled within the internal state and Register File until one of those conditions is triggered by an external event.

The first condition tests the two interrupt flags in the interrupt control register. The interrupt control register is contained within the Register File and these flags may be set whenever a DMA operation has completed.

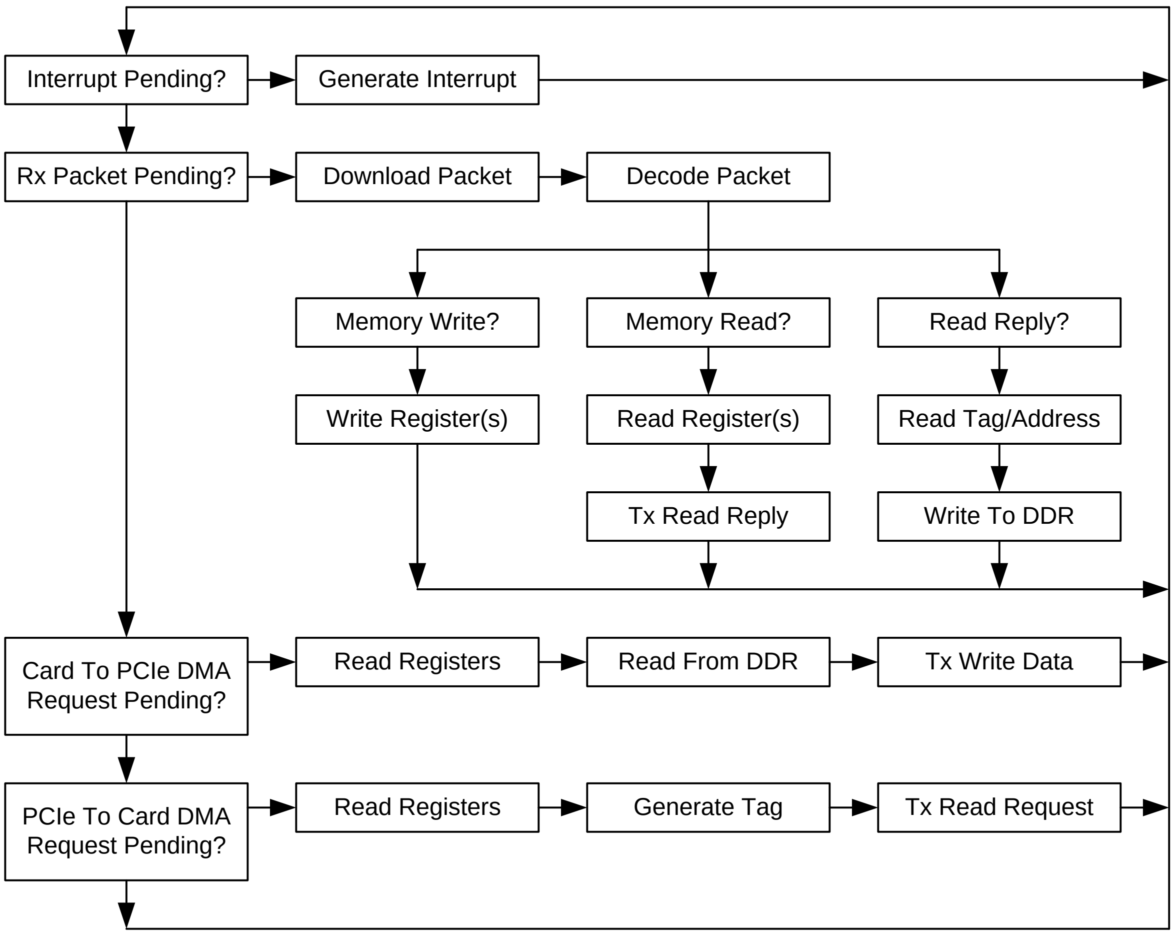

Figure 2. Control State Machine

The next check is for any inbound packets that may have arrived. These will be indicated by the flow control signals coming from the PCIe core. If present, these must be processed first in order to avoid a deadlock condition. Inbound packets can be one of several types: Memory Write, Memory Read or Read Reply. If they are Memory Write/Read, then it is assumed that they are accesses to the Register File and will be handled accordingly.

If the received packet is a read reply (completion), then it is assumed to be in response to a DMA read request that the card has initiated. The data from the reply is written into the DDR memory at the appropriate address. Since it is legal to have multiple read requests active on the bus at the same time, a tag in the PCIe packet header is used to associate the read completion with the original read request. The tag is used as an address into a small memory that contains the address of the original read request so that the data in the read reply can be written to the correct locations in the DDR.

The last two checks in the main loop of the state machine are for generating new DMA transactions on the bus. These are last in order to avoid stalling the host while waiting for the card to complete a DMA operation. For example, since received packet processing is given priority over DMA generation, it is possible for the host to poll the DMA count registers in order to watch the progress of the DMA. DMA in both directions is initiated by the host through the act of writing into the Register File. The host fills in the source or destination address in PCIe memory address space, the local DDR address for the transaction, and the number of DWORDs that are to be transferred. The Control State Machine reads the transfer counts and generates read or write requests as needed until the total number of DWORDs has been transferred at which point an interrupt will be generated for the host.

3. PERFORMANCE

The test system consists of the Xilinx ML505 reference board, which houses a Virtex 5 XC5VLX50T, a x1 PCIe interface and a SODIMM slot with 256 MByte DDR2 RAM standard. The PCIe core used is the Xilinx Endpoint Block Plus version 1.3. This is plugged into a Supermicro X6DAT-G motherboard with dual Xeon processors running at 3.4GHz and 4GByte of DDR2-666 RAM installed. The operating system is Microsoft Windows XP Service Pack 3.

There are many different variables to consider when attempting to achieve maximum performance on PCIe, especially when transferring data to/from system memory on a PC-like host. Certainly the DMA transfer itself is one possible bottleneck, but also important are the design of the device driver, and the method by which the host initiates transfers to/from the card.

3.1 Read Request Performance

Read requests refer to transfers from host RAM to the FPGA, as initiated by a bus master read request from the FPGA. The maximum payload that can be requested is the minimum of several system parameters set by the host and the PCIe core. The host polls these parameters and then writes the actual value into each device's Configuration Space at boot time. Using the Xilinx PCIe core and the test system, the maximum transfer request size is 512 Bytes.

Figure 3 shows the completion header and data credit counts updating as a 512 Byte transfer is initiated (shown by the X marker) until it is complete (shown by the O marker). These hex counts start off at their maximum value on the left side of the timing diagram. At the start of the transfer, the total required credits are deducted from each and then the counts increase back to their respective maximums as completions return from the host. A total of 513 clock cycles elapse in between. That translates to 59.49 MByte/Sec maximum throughput if no driver were involved using the x1 PCIe core clock rate of 62.5 MHz.

In practice, the transfer rate as measured from the Window's test application written in C is much less, coming in at between 11 MByte/Sec and 15 MByte/Sec, depending on driver transfer size settings and run to run variance. The "real" numbers are so much lower because of the software overhead of acknowledging the interrupt and programming new values into the registers after each contiguous transfer. Since this is running under a real operating system (Windows) using paged virtual memory, the physical addresses for a large transfer will be non-contiguous and so many transfer requests from the driver are necessary.

In fact, the main performance bottleneck with this implementation is Window's interrupt processing rate. The average contiguous block length was found to be approximately 2KBytes, implying that the maximum interrupt service rate is 15MByte/Sec / 2KBytes = 7680 interrupts per second. Since this PCIe implementation depends on the software to respond to an interrupt from the card after each contiguous transfer, this becomes problematic. This model should be changed so that the host does not need to respond to an interrupt after every contiguous transfer. Typically this is solved by creating a linked list or circular queue of transfer requests that the driver can fill up and then leave the hardware alone while it services each contiguous transfer in turn.

It can also be observed in Figure 3 that the header and data credits are strictly increasing after the read request, which indicates that no read requests are being issued concurrently with the return of the completions. Although the Control State Machine contains the logic to generate these requests, the completions are returning faster than the Control State Machine can process them and so it

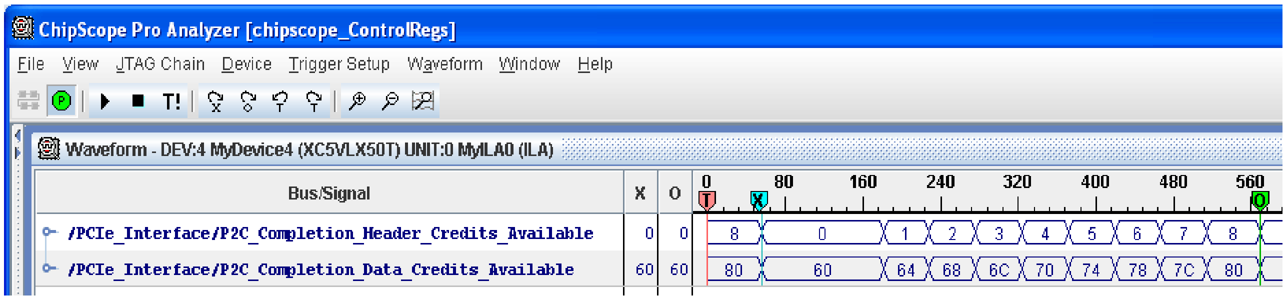
ChipScope Pro Analyzer [chipscope_ControlRegs]
File View JTAG Chain Device Trigger Setup Waveform Window Help
Waveform - DEV:4 MyDevice4 (XC5VLX50T) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
0
80
160
240
320
400
480
560
/PCIe_Interface/P2C_Completion_Header_Credits_Available
0
0
8
0
1
2
3
4
5
6
7
8
/PCIe_Interface/P2C_Completion_Data_Credits_Available
60
60
80
60
64
68
6C
70
74
78
7C
80

Figure 3. Read Request Timing (Credit Values In Hexadecimal)

never gets far enough in the polling loop to generate another read request until all of the completions for a given request have been processed. 128 clock cycles elapse from the time that the credits are deducted until the first completion returns. If these requests were being issued in parallel with the completions, that latency could be hidden, possibly only requiring 385 clock cycles for the 512 Byte request, and giving a software-free throughput of 79.3 MByte/Sec. A parallel state machine design could alleviate this problem.

A more subtle slowing effect is introduced by the Packet Buffer shown in Figure 1. As mentioned previously, every packet that is received is clocked into the packet buffer before it is processed. Since the completions are returning so quickly, it is entirely possible that the latency of buffering the packet is adding to the apparent completion processing time. Unfortunately, it is impossible to measure the effect of this added latency. Since there is no way to judge how many completions are buffered within the PCIe core, measuring the latency of the Packet Buffer alone would not allow an adequate estimation of throughput if the Packet Buffer were not present. However, this implies that the peak throughput number could be higher than calculated. The Packet Buffer was included to allow easy access to any part of the packet and to ease flow control issues. As written, random packet access was not needed, and the potential slow down is not worth the ease of flow control, so the Packet Buffer should be eliminated.

Lastly, the maximum number of completion header credits allowed by the PCIe core could be a limitation in that more credits could allow for larger read requests in some systems. This would allow a longer period during which read replies were returning when new read requests could be transmitted to the host. Of course, this parameter is fixed by the PCIe core and is out of the designer's control. Using this implementation, it is not possible to determine if a larger number of completion header credits would be beneficial in the test system.

3.2 Write Request Performance

Write requests refer to transfers from the FPGA to the host, as initiated by a bus master write request from the FPGA. Write requests are posted, meaning that a header is transmitted with data and no reply is required from the target. This should eliminate some overhead time as seen with read requests where a request header is transmitted and then the device waits for read replies (completions). The transfer size for write requests is subject to the same minimization process used for reads, with a different set of rules and parameters. The allowed maximum transfer size is 128 Bytes for the test system. Running at full burst, these 128 Byte packets require 108 clock cycles back to back for transmission, giving a peak throughput of 74.1 MByte/Sec when not encumbered by software. The smaller allowed burst size (and greater header overhead) is likely hurting the write requests, resulting in lower peak bandwidth vs. read requests. It is also possible that the added latency of the Packet Buffer is slowing the peak rate, but it is still impossible to determine to what extent this affects the transfer rate using this implementation. Software overhead takes its toll on final throughput; and the overhead of interrupt processing and register programming drags performance down to between 11 MByte/Sec and 15 MByte/Sec as seen from the C test application. As with reads, it is expected that lowering interrupt processing overhead and removing the Packet Buffer would allow this to be much higher.

It is important to recognize that all of these performance numbers are an amalgamation of software and hardware overhead. Sources of software overhead come mainly from interrupt response times and driver efficiency. Possible sources of hardware overhead are the test system's chipset, the design itself and the PCIe core. Likely these numbers would improve or degrade in another system using a different motherboard chipset, or faster processor.

Table 1. PCIe x1 Link Measured Performance

	Sustained Software Performance	Peak Hardware Performance
Read Request	11-15 MByte/Sec	79.3 MByte/Sec
Write Request	11-15 MByte/Sec	74.1 MByte/Sec

4. PRIOR WORK

After the time of writing it was discovered that Xilinx [6] had produced a similar reference design for bus mastering PCIe DMA one month prior. Performance issues were not analyzed, but it seems to suffer from the same hobbling problem of interrupt service rate as this design since the x1 lane configuration has roughly the same throughput and a similar limitation for contiguous transfers as this design.

Altera [7] also provides a reference design for bus mastering DMA in which they analyze theoretical maximum transfer rates and achieve much better actual transfer rates using a linked list of transaction descriptors, thus lowering interrupt overhead.

5. CONCLUSION

PCI Express is a complex interface that requires a sophisticated state machine to implement. While there are simpler interfaces that can be used, the raw speed offered by PCIe, and its ubiquity, make it a good choice for many potential projects that require high performance. This paper has presented one implementation of PCIe using the Xilinx PCIe core, and analyzed it for possible improvements. The net effect of the suggested changes could lead to measured improvements in read and write transactions that more closely approximate the hardware peak performance numbers shown in Table 4, and possibly approach the theoretical limit of 200 MByte/Sec for a x1 link. The source code for the current design is available on the web at:

http://research.microsoft.com/people/raybit/

6. REFERENCES

[1] *LogiCORE Endpoint Block Plus v1.3 for PCI Express User Guide*. Xilinx Corporation, May 17, 2007.

[2] *PCI Express Compiler User Guide v8.0*. Altera Corporation, May 2008.

[3] Budruk, R., Anderson, D. and Shanley, T. 2006. *PCI Express System Architecture*, Addison Wesley.

[4] *Virtex-5 User Guide*. Xilinx Corporation, February 2, 2007.

[5] *PCI Local Bus Specification Revision 2.1*. PCI Special Interest Group, June 1, 1995.

[6] Wiltgen, J. *Bus Master DMA Reference Design for the Xilinx Endpoint Block Plus Core for PCI Express*. Xilinx Application Note 1052, Xilinx Corporation, August 22, 2008.

[7] *PCI Express High Performance Reference Design*. Altera Application Note 456, Altera Corporation, May 2007.