

An Efficient Communication Controller Architecture for FPGA-Based Hardware Acceleration of Signal Processing Applications

Petersen F. Curt
EM Photonics, Inc.
51 East Main St.
Newark, DE USA 19711
302-456-9003
pcurt@emphotonics.com

James P. Durbano
EM Photonics, Inc.
durbano@emphotonics.com

Fernando E. Ortiz
EM Photonics, Inc.
ortiz@emphotonics.com

John R. Humphrey
EM Photonics, Inc.
humphrey@emphotonics.com

Abstract

Both academic and commercial researchers are developing systems that utilize FPGAs to accelerate signal-processing applications. These systems typically off-load computationally intensive portions of software algorithms to FPGAs in order to gain a parallel processing speedup. Because the problem data are stored in the host PC, this approach requires communication between the coprocessor and host PC. When problems require frequent transfers and/or large data sets, the benefits of the FPGA may be lost to data transfer overheads and inefficiencies. In this paper we present the infrastructure for an efficient communication controller capable of providing the necessary bandwidth to sustain computational throughput. Important points presented include: layered abstraction, direct memory access, and clock de-coupling techniques.

1. Introduction

Software implementations of scientific methods benefit directly from improvements in general-purpose processor technology. Unfortunately the steady rate of performance increase has begun leveling off as the improvements are becoming less effective. Thus, researchers have shifted their design methodology from a purely software approach to a hybrid one using specialized hardware coprocessing.

Efficient communication between the host workstation and hardware coprocessor is absolutely necessary in order to sustain the throughput necessary to make the approach worth-while. For instance, a high-throughput, parallel FFT can be easily implemented in customized hardware, but unless the communication channel can supply the unit with a constant data stream, the computational pipelines will be starved for input, forcing artificial slowdowns. As a result, the overall performance of the hybrid approach could actually become slower than a pure software approach. Thus, we

must ensure that hardware-based acceleration platforms contain highly efficient communication controllers.

Although commercially available communication controllers provide basic protocol support, their overall functionality is generally very limited. For instance, Xilinx provides an excellent user's guide for their PCI core, with example code demonstrating its implementation and use, but much more work is necessary to take full advantage of the communication resources PCI provides. We have developed a full-featured communication controller that includes both hardware and software interfaces. This allows the system to manage communication with the host at the virtual channel level, thus abstracting the details of the physical interface from the algorithm's implementation and enabling a clear upgrade path for future technologies. Further, the architecture provides a simple user interface and bus utilizations in excess of 99%.

The power of such a controller has already been demonstrated in the successful implementation of a Finite-Difference Time-Domain (FDTD) hardware accelerator [1], which rivals the speed of a 100-node PC cluster (3.0 GHz, 2GB PC3200 DDR, 100 Mb Ethernet). This generalized communication architecture is applicable to numerous applications, including digital filters, FFTs, and linear systems. In the next section, we begin with an overview of a common communication interface – the PCI bus. We describe the basic elements of the architecture, the protocol devices must obey, and the different communication modes available, including benchmarking results.

2. The PCI Bus and Transfer Modes

PCI is a shared bus, which uses a transaction-based protocol to allow two devices to communicate with one another. At the physical level, a single set of wires is shared between all devices on the bus, and each device can selectively drive, or sense, the state of each signal. Once the PCI bus arbiter grants a device ownership of the

bus, that device initiates a transaction to the target device by driving a subset of the bus signals. When the target device signals it is ready, data may be bursted until the transfer is complete.

The CPU and main memory are not directly connected to this bus. Instead, they must communicate with devices through a PCI host bridge (specifically the south bridge in x86 architecture). This architecture allows the CPU and main memory to communicate very quickly, but can hamper communication with the PCI bus because all accesses must go through the bridge (see Figure 1, below). There are two possible transfer modes to move data through the bridge. The first method uses the CPU as initiator and the PCI device as the target. The PCI host bridge is an intermediate entity, physically initiating the transactions on the PCI bus. In the second case, the roles of the host bridge and device are reversed; the PCI device initiates the transaction to the target host bridge.

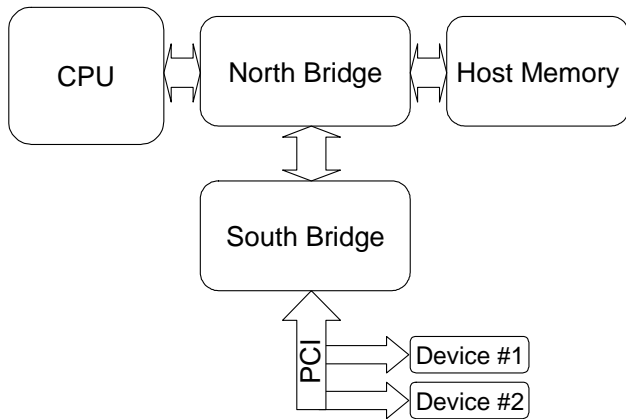


Figure 1. Typical x86 PC Architecture This figure shows the basic components used by the PCI hardware accelerator to communicate with the CPU and host memory. In order to communicate with either of these entities, the data stream must pass through both bridges before it reaches the device.

Before designing the communication controller, testing was performed to determine the best method for device communication. We found that using the former approach (CPU is initiator, device is target) on the PCI bus was highly inefficient for large data transfers (see Table 1).

To achieve increased efficiency for large data transfers, we can use the second transfer method, where the PCI device is the initiator and the main memory is the target. This method is typically referred to as direct memory access (DMA), because it does not involve the CPU. Using this technique, the device can sustain longer bursts (on the order of thousands), effectively utilizing up to 99% of the bandwidth (see Table 2).

Because of the tremendous increase in efficiency, DMA transfers were chosen for the advanced communication controller. When implemented in our accelerator, this reduced the initial download time from four minutes to less than 8 seconds: a 30-fold speedup. However, implementing a full-featured DMA engine is not a trivial task, considering that it must interact with both the PCI physical interface and the acceleration engine. Further, these two variables may change, depending on the platform and solver combination being targeted. Thus, the simplest way to implement this functionality, without having to restrict its reusability, was to include it in a new layer, which we call the session layer.

Table 1. Benchmark Results for Bridge Initiator & Device Target This table contains observed communication throughput when using a CPU initiator scheme for targeting the hardware accelerator. Read and write bursts of up to 15000 words (~60KB) in size were tested, but were found to be impractical for use in sustaining large data transfer throughputs for the accelerator engine.

Target Direction	Transaction Size (# words)				
	10	100	1000	10000	15000
READ	1.09 (3%)	0.99 (3%)	0.98 (3%)	0.97 (3%)	0.98 (3%)
WRITE	6.95 (21%)	5.97 (18%)	5.49 (17%)	5.46 (17%)	5.45 (17%)

* all values in words/ μ s and (% efficiency of 32b/33MHz PCI)

Table 2. Benchmark Results for Device Initiator & Bridge Target This table contains observed communication throughput when using the direct memory access (DMA) technique. This technique yielded a tremendous advantage in efficiency over the previous case studied, sustaining bursts of data with 99% efficient bus utilization.

DMA Direction	Transaction Size (# words)				
	10	100	1000	10000	15000
READ	15.8 (48%)	29.8 (90%)	32.4 (98%)	32.7 (99%)	32.6 (99%)
WRITE	15.7 (48%)	29.7 (90%)	32.4 (98%)	32.6 (99%)	32.6 (99%)

* all values in words/ μ s and (% efficiency of 32b/33MHz PCI)

3. The Session Layer

This section introduces the additional layer provided by our communication controller, the session layer. Unlike networking protocols, such as the Open System

Interconnection (OSI) model, PCI communication is typically not described using layers [2]. However, the same logical tasks required for network endpoints to communicate effectively can be applied to the communication required between a host PC and a hardware accelerator device (see Figure 2, below). Although defining multiple layers is not necessary for simple PCI communication tasks (e.g., the programming of several registers), we noted in the previous section that large data transfers require the use of DMA, a more complex transfer method. This complexity is due in part to the shift of communication state; for simple transfers, a software driver on the host PC is responsible for maintaining communication state, whereas a DMA solution must share much of that state with the hardware. Whether in hardware or software, we propose that this state can be abstracted from the other tasks and stored in a separate “session” layer. The session layer’s primary purposes are to provide the additional functionality required by DMA transfers and a means for data verification, leaving the vendor-specific PCI core intact.

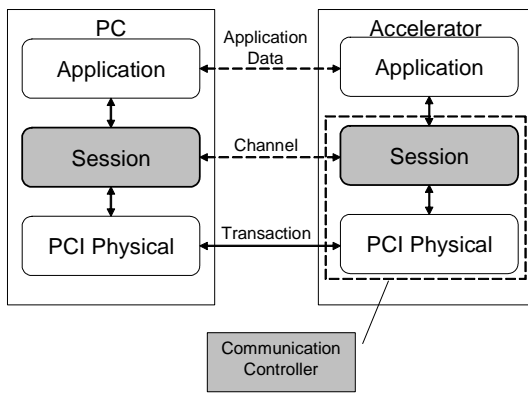


Figure 2. The PC/Hardware Communication Stack Here we see a layered model of the PC/Hardware communication medium. Note that the communication controller contains both session and physical layers. The session layer provides additional functionality needed for DMA transfers, which are necessary for increased communication efficiency.

We begin this section by describing the session layer’s interfaces to its nearest neighbors on the communication stack: the user’s application and PCI physical layers. Next, we discuss the important internal elements necessary for connecting the layers, including first-in first-out (FIFO) buffers.

3.1 Physical Layer – Session Layer Interface. The session layer’s communication with the PCI bus is indirect, utilizing a physical layer to exercise the bus signals properly. The session layer must present the

proper interface to the physical layer so that transactions between the two can be understood. In effect, they must be glued together in order for the communication stack to function properly. This glue logic is located inside the session layer, which allows it to be tailored to any physical layer implementation, such as the Xilinx PCI core (see Figure 3, below).

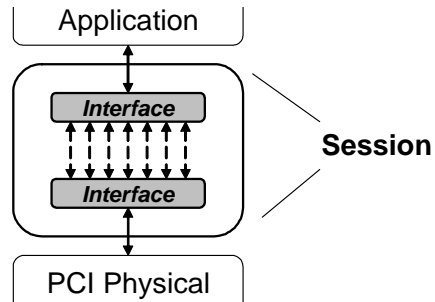


Figure 3. Session Layer Interfaces The interfaces to the session layer determine how the added functionality the layer provides interacts with the PCI physical core and the user application. Each interface is flexible enough to interact with most user applications and PCI Physical cores with little integration effort.

The session layer’s interface with the physical layer includes two important signal groups: target and initiator interface. The target signals are used when the physical layer receives a transaction request from another device on the bus, which it cannot fulfill itself (e.g., host PC reads a status register in the acceleration hardware). The session layer must decode the signals from the physical layer properly, pass the necessary information to the application layer, and then respond to the physical layer when the transaction is complete. The initiator group is used when the controller needs to initiate a DMA transaction on the bus (e.g., problem dataset needs to be initially transferred from PC main memory to the hardware accelerator’s memory). The session layer requests a transfer from the physical core, providing the information needed for the transaction. When the physical core initiates transactions on the PCI bus, it provides a ready signal to the session layer, at which point the data transfer can begin.

The session layer’s interface with the physical layer is not tied to any specific implementation, so it can be easily assimilated into existing designs that use anything from a Xilinx LogiCORE to an external ASIC accelerator bridge. This particular implementation of the controller was targeted to the PLX9656 I/O accelerator [3]. It is an external ASIC, which communicates with a PCI 64-bit, 66MHz bus, and presents a simplified, 32-bit “local bus” to the FPGA. This ASIC was especially useful for prototyping this design because it moved the

PCI physical layer interface off the FPGA, and allowed for a faster development cycle. Because the new layer is not specifically tied to PLX's proprietary local bus interface, the Xilinx PCI LogiCORE could replace the PLX.

3.2 Application Layer – Session Layer Interface. On the other side of the session layer is the application layer interface, which is responsible for communicating with the user application. This interface is used when a PCI transaction request is received and cannot be fulfilled by either the physical or session layers (target mode), or when the user application must interact with data during a DMA transaction (initiator mode).

In order to be both robust and simple to interact with, the interface was designed as two separate groups of signals. The high-speed group interfaces with the session layer's DMA components and consists of several ports that operate as a set of FIFOs. Status signals define when the channel is ready and data can be transferred in and out using a synchronous enable. The second signal group provides a lightweight bus interface, which is used to map registers in the user application to the device's memory space for direct access. Although slower than the high-speed ports, this group provides the application's software controller the freedom to access internal register banks as needed. These groups are shown in Figure 4, below.

Each group's signals are synchronous with respect to the provided computational clock, making it possible to clock the user application at a much higher frequency than the communication bus. For example, our group uses floating-point arithmetic units, pipelined for operating frequencies in excess of 150 MHz. If these units were limited to the 33 MHz PCI clock, no overall speedup would be achievable.

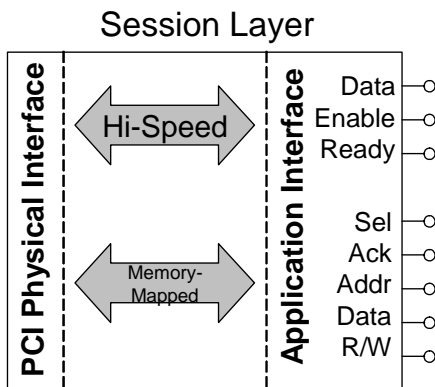


Figure 4. Session Layer's Application Interface This interface consists of two groups of signals: high-speed signals for fast DMA transactions and memory-mapped signals for providing host software direct access to user application registers.

3.3 Custom DMA Controller. A customizable DMA controller is used to transfer data on the high speed ports of the application layer interface. The DMA controller can initiate transactions on the PCI bus, targeting any other memory-mapped device in the system with optimal efficiency. Our particular group's applications most often target PC main memory and other FPGA sibling cards on the bus. Using such a controller allows the efficient transfer of gigabytes of data at a sustainable rate of over 128 MB/s. The controller maintains its own state throughout the transaction, so it can be programmed once by the software, and requires no intervention until the transfer is complete.

By itself, the DMA controller is limited to buffers of contiguous physical memory. This is because the state that is stored in hardware describes a block of memory by 1) its starting address and 2) its size. As these blocks become larger, it may become the case where they span multiple virtual pages and pieces of a block become segmented by the operating system's virtual memory manager. To solve this problem, the DMA controller uses FPGA cache blocks (Xilinx SelectRAM) to store a self-updating memory paging table. Without this page table the DMA controller would be forced to stall when it reached one of these segment boundaries and request the next chunk's address and size information from the host PC application. This latency tends to be very long and can occur very often, depending on the size of the transfer. The page table allows the host software to provide the DMA controller with many chunks of memory at the beginning of a transfer, instead of just one. In this way, the DMA controller can transfer a portion of a large buffer in one page, and then switch to the next page using back-to-back transfers. The last entry in the table indicates a physical memory address for the next set of pages so the table can refill itself with fresh values, again using back-to-back transfers.

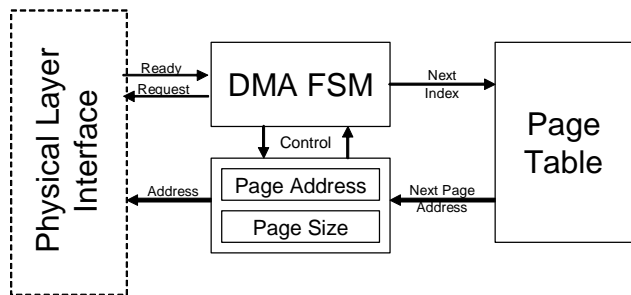


Figure 5. Session Layer DMA Infrastructure The session layer DMA infrastructure employs the use of a finite-state machine (FSM), coupled with a lookup table of memory page addresses. The DMA FSM's control logic manages all components to ensure that the correct

target address is presented to the physical layer interface.

3.4 Buffering. Because the DMA controller can sustain transfer rates up to 128 MB/s, the data bus needs a comparably efficient technique to keep data moving through the communication stack. Specifically, using a FIFO buffer helps regulate traffic between two entities that may have differing bandwidth requirements, or logical interfaces, and can provide temporary storage in case the consumer is not immediately ready to process the data. The FIFO interface can be viewed as flow control for the high-speed link providing direct access to a buffer in PC main memory.

The application layer transfers data over the link using an enable, and monitors the link's state with a status signal. The direction of the FIFO is specified in the controller's instantiation in the HDL, and the explicit meaning of the signals "enable" and "status" are appropriately assigned based on that direction. In the case of a main memory-to-hardware link, the enable is a read-enable, and the status is an empty signal. For write, the enable is used for writing and the status denotes full. This interface provides a simple, familiar technique for any user application to follow, and allows for reuse regardless of the particular functionality of the user application.

Many of the designs that reuse this functionality are targeted for operating frequencies over 100 MHz. The PCI bus is limited to a much slower speed, so we typically use at least two clocks in our designs: one for communication and another for computation. The FIFOs used in the communication layer are asynchronous from one another [4], meaning that the read and write interfaces can be synchronized to different clocks. This way, the FIFO can bridge the two clock domains efficiently, without the user having to consider the boundary at all.

Oftentimes the width of the bus across the clock boundary is not the same as the word size of the communication bus. For instance, in one of our accelerator architectures, new runtime results become available every cycle on a 192-bit wide bus. These results must be partitioned into 32-bit words suitable for PCI before they can be transferred to PC main memory for storage. Instead of further buffering this data for width conversion with more FIFOs in the application layer, the session layer can provide this functionality at very little additional cost. Thus, it is best to allow the width of the FIFO to be arbitrarily specified when the communication controller is instantiated in the HDL. All necessary logic for width conversion can be automatically generated and the logic pushed to the communication clock domain, where the predetermined communication bus word size is used. Incorporating this

flexibility into the communication controller allows further design reuse and increased speed, as width conversion logic is implemented in the PCI clock domain.

In a similar spirit of reuse, additional steps have been taken to make the application interface even more flexible for the user application, specifically for applications that burst data for efficiency. In the case of the DRAM download described earlier, the accelerator architecture bursts data from the FIFO for a prolonged time, depending on the memory controller design. Currently, our DRAM modules on the platform are configured for a burst length of at least four 128-bit words per access. However, this is not possible using the existing interface, because it is not possible to ensure the 4-word burst requirement of the memory transaction using only a "not empty" status. To overcome this at the application layer, the design could expand the FIFO's 128-bit port four times to read each segment in succession, therefore guaranteeing the data availability for a burst of four. But this requires routing a much larger bus, which is used only 25% as often. It is much more efficient to implement the 128-bit port and use the bus more often. For this reason, threshold and status information is included in the communication controller, built into its session layer. The FIFO has a programmable threshold, which the user application can program to receive a signal when the FIFO has at least a certain number of entries available before initiating a burst. This allows processing larger chunks of data, without the waste of using of a wider data bus. The runtime programmable nature of these bursts allows us to tweak the burst lengths in order to achieve maximum efficiency, end-to-end.

Using this flexibly coupled FIFO-DMA approach to implementing a high-speed link is the most powerful aspect of the advanced communication controller. Once the software driver completes the association between an allocated chunk of PC main memory and the session layer's DMA registers, data will be transferred as flow permits automatically, with no hardware intervention whatsoever.

3.5 Multiple Virtual Channels. A single FIFO transferring data in one direction may not be enough for all user applications. For instance, the FDTD hardware accelerator continuously transfers data in both directions (to and from PC main memory) throughout the computation process. Thus, multiple virtual channels were incorporated into the communication layer for added flexibility. Up to eight separate FIFO buffers can be included, simply by changing parameters when instantiating the layer. Figure 7, below, shows such an implementation including three virtual channels.

Each FIFO shares the single session layer's DMA engine, but has its own set of DMA registers, and a separate page table to maintain concurrent transfer state. Additional options exist to set the number of page table entries and optimize the engine for each transfer's characteristics. For instance, when communicating from one hardware accelerator platform to another, the transfer is FIFO buffer to FIFO buffer. Because the remote FIFO buffer is a single memory-mapped location, neither an address counter nor a page table is needed for such a transfer.

Each virtual channel/FIFO pair can be given its own session-side programmable threshold for bursting data (separate from the application threshold described earlier). When the prescribed amount of data is available, the FIFO will signal its availability to the DMA engine. The DMA engine then transfers data to or from each available FIFO in a fair, round-robin fashion.

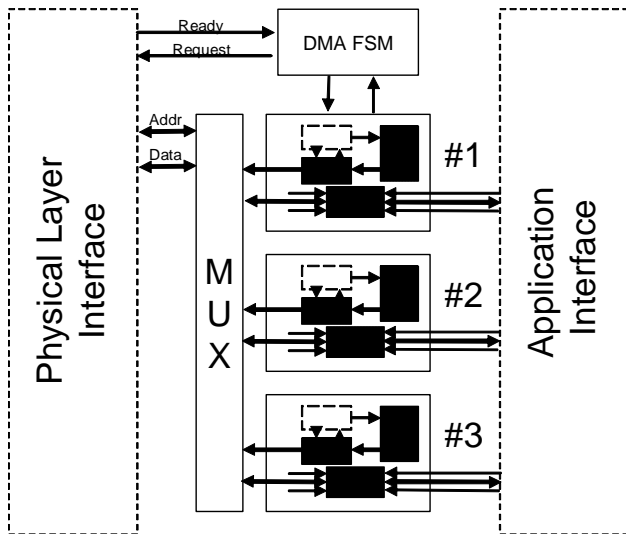


Figure 7. Three Virtual Channels *The previously described logic is replicated inside the session layer to support up to eight virtual channels, each sharing a single DMA FSM. This figure depicts an implementation with three such channels.*

4.0 Conclusion

This paper provides a more adequate path to high-bandwidth communication for FPGA-based hardware accelerator designers. The additional session layer presented in this paper bridges the gap between PCI physical core interfaces and the user application to provide an efficient, reusable design pattern that is both powerful and easy to integrate. Final results show over

99% bus utilization in benchmarks, and only minimal speed sacrifice when cooperating with the hardware accelerator at maximum throughput. Our future work includes improving speed and functionality at both the physical and application layers of our designs.

At the physical layer, one remaining, severely limiting factor is the 133 MB/s PCI bus. In an effort to overcome this limitation, the communication controller was ported to a new accelerator platform with a 32-bit/66 MHz bus provided by a PCI-X bridge, effectively doubling the maximum throughput to 266 MB/s. This important step was made with very little redesign to the communication controller's physical interface and allowed twice as much result data to be stored. Further steps are being planned to target a 64-bit PCI-X bus that can offer data throughputs over 1 GB/s. This port should require no significant changes to the communication controller other than a physical layer swap. By next year, our group has plans to interface the same communication controller to a PCI-Express implementation, which supports an aggregate bandwidth of almost 250 MB/s per lane. The final implementation planned is 16-lanes for almost 8 GBps of bandwidth.

On the application side, only two of the eight possible channels are currently being used by any of our hardware architecture solvers. We plan to increase the number of used channels in order to demonstrate the power of such a controller when it is used to link multiple hardware accelerators sharing the same PCI bus. This would allow the platforms to work in tandem, communicating directly with one another, without CPU intervention. Our projected result is a single PC-based accelerator system capable of FDTD computational throughputs rivaling a 150-node cluster.

References

- [1] J. P. Durbano, P. F. Curt, J. R. Humphrey, F. E. Ortiz, and D. W. Prather, "FPGA-based Acceleration of the Three-Dimensional Finite-Difference Time-Domain Method for Electromagnetic Calculations," presented at Global Signal Processing Expo & Conference (GSPx), Sep 2004.
- [2] International Organization for Standardization.
- [3] PLX-Technology, "PCI 9656BA Data Book," 2003.
- [4] Xilinx, "FIFOs Using Virtex-II Block RAM," 2005.