A Versatile UDP/IP based PC ↔ FPGA Communication Platform

Nikolaos Alachiotis, Simon A. Berger, Alexandros Stamatakis
The Exelixis Lab, Scientific Computing Group
Heidelberg Institute for Theoretical Studies
Heidelberg, Germany
Emails: {Nikolaos.Alachiotis,Simon.Berger,Alexandros.Stamatakis}@h-its.org

***Abstract*—We present a substantially improved version of our popular UDP/IP core for simple and fast PC ↔ FPGA communication over Gigabit Ethernet. We provide a novel feature to automatically configure (previously hard-coded) internal settings on the FPGA. Thereby, we substantially reduce the installation overhead when a FPGA shall communicate with several different PCs. The UDP/IP core is designed to occupy a minimum amount of hardware resources on the FPGA. On the PC side, this new automatic configuration protocol can be used and invoked via a C software interface which provides convenient functions for setting up the connection to the FPGA device and sending/retrieving arrays of common C data types to/from the UDP/IP core on the FPGA. The initial UDP/IP core version is available under the LGPL license at http://opencores.org/project,udp_ip__core while the improved version of the core, including the C software interface (also under LGPL), is available at http://opencores.org/project,pc_fpga_com.**

***Keywords*-FPGA; UDP/IP; PC-FPGA communication;**

I. Introduction

FPGAs are commonly deployed as accelerator devices or for verifying architectures. Irrespective of their particular use, a machinery for moving data to/from the device is required. Recently, we presented a proof-of-concept implementation of a highly efficient UDP/IP core architecture that is optimized for point-to-point communication [1]. We found that combining the IPv4 and UDP protocols represents an efficient solution for establishing a connection between a PC and a FPGA, with respect to reconfigurable hardware utilization. We achieve low resource utilization because the architecture employs a simplified method for calculating the IPv4 Header Checksum based on a pre-calculated template.

Here, we present an improved UDP/IP core implementation *and* a comprehensive Ethernet-based communication platform. A key property of the UDP/IP core is that all static header fields required for point-to-point communication are stored in a lookup table. In the proof-of-concept implementation, this required the user to manually initialize the lookup table before configuring the device. The improved architecture we present here entails additional control logic that retrieves all static fields of an incoming initialization packet from the PC. This data is then stored in the respective lookup table. This versatility of the new UDP/IP core comes at a cost: the additional logic occupies approximately 56% more hardware area than the proof-of-concept core. Nonetheless, this novel and more flexible UDP/IP core still outperforms all competing UDP/IP core implementations in terms of speed *and* resource requirements.

Since February 2010, the proof-of-concept implementation has been downloaded over 6,000 times from OpenCores.org (http://opencores.org/project,udp_ip__core) and from our software page (http://www.exelixis-lab.org/countIPv4.php). Note that both Xilinx [2] and Altera [3] provide PCI Express-based solutions for PC ↔ FPGA communication. However, the interest our initial core has generated underlines the apparent lack of a standardized, open-source Ethernet-based communication mechanism. Thus, in addition to the improved UDP/IP core architecture, we also present an implementation of a comprehensive open-source PC ↔ FPGA communication platform that relies on Gigabit Ethernet. This platform deploys the versatile version of the UDP/IP core and implements a simplified communication protocol that facilitates usage and integration of the send/receive paradigm for common C data types and arrays. Our platform hides the complexity inherent to establishing a PC ↔ FPGA connection and to synchronizing data transmissions in both directions. The versatile UDP/IP core and the communication platform are available for download at http://opencores.org/project,pc_fpga_com.

The remainder of this paper is organized as follows: Section II provides a short overview of related work on UDP/IP and TCP/IP cores as well as PC ↔ FPGA communication platforms. In Section III, we describe the design of the novel and more versatile UDP/IP core. The PC ↔ FPGA communication platform is introduced in Section IV. Section V covers implementation and verification details as well as a performance evaluation of the UDP/IP core and the PC ↔ FPGA platform. Section V also includes application examples to demonstrate the functionality of the proposed solution. We conclude in Section VI and address directions of future work.

II. Related Work

In [4], Löfgren *et al.* present three different (commercially available) stand-alone UDP/IP cores and denote them as *minimum*, *medium*, and *advanced* implementations. The *medium* and *advanced* IP cores offer additional protocols which are not necessary for establishing a direct PC ↔

FPGA communication. Thus, we do not compare these heavy-weight cores with our implementation because such a comparison would be unfair. The *minimum* IP core offers the same functionality as our Gigabit-speed UDP/IP core but requires 2.8 times more hardware resources while operating at 10/100 Mbps only.

In [5], Kühn *et al.* used UDP/IP to implement PC ↔ FPGA communication over Gigabit Ethernet. To implement the communication mechanism, the authors deployed an operating system (Linux) that was running on an embedded PowerPC processor. While using a processor that handles the communication represents an elegant and straightforward approach, not all reconfigurable devices contain such hard-coded processors. When such a processor is not required for conducting additional computations, deploying a soft-coded processor can lead to excessive hardware utilization on the device. Our novel communication platform essentially offers the same functionality but does not require an embedded processor.

Dollas *et al.* [6] presented an architecture for an open TCP/IP core that supports all necessary protocols (ARP, ICMP, UDP) typically required for real-world applications. The architectural complexity that is induced by the large number of supported protocols would yield a comparison to our light-weight design unfair. Note that a full TCP/IP core is not necessary for direct PC ↔ FPGA communication since this can be implemented using less complex protocols and a lower amount of resources.

In [7], Lin *et al.* presented a PC ↔ FPGA based communication platform for verification and fast prototyping. The platform communicates via a PCI interface and is intended to facilitate verification of multimedia applications. There exist similar platforms as the one presented by Lin *et al.* that target specific application types [8], [9]. Our PC ↔ FPGA communication platform offers a mechanism for integrating communication routines into any kind of application and allows for transferring arrays of the basic IEEE-754 data types (e.g., characters, short integers, integers, floats, long integers, and doubles).

Recently, Lieber and Hutchings [10] presented an open-source PC ↔ FPGA communication framework for Xilinx FPGA boards achieving a maximum throughput of 504 Mbps. Our PC ↔ FPGA communication platform provides similar functionality and can operate at Gigabit speed (1000 Mbps) while requiring 50% less reconfigurable resources.

III. Versatile UDP/IP Core Architecture

In the following, we briefly review the basic concept of the initial UDP/IP core and describe the enhancements in the new version. As already mentioned, all static header fields required for direct point-to-point communication are stored in a lookup table (henceforth denoted as HLUT). The fields stored in HLUT form part of the 802.3 MAC frame, the IPv4 header section, and the UDP header section. The main fields

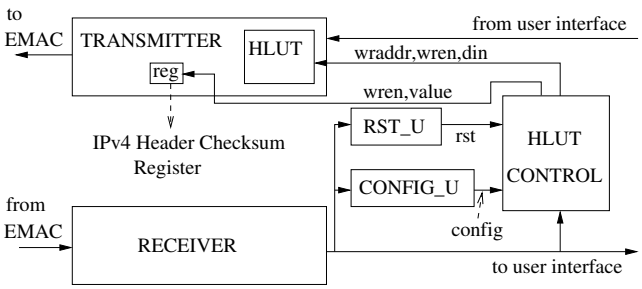

Figure 1. Top-level design of the advanced UDP/IP core.

of the 802.3 MAC frame are the *Destination Address*, the *Source Address*, and the *Ethernet Type*. The IPv4 header section contains the following fields: *Version*, *Header Length*, *Differentiated Services*, *Total Length*, *Identification*, *Flags*, *Fragment Offset*, *Time to Live*, *Protocol*, *Header Checksum*, *Source Address*, and *Destination Address* [11]. The UDP header section consists of the *Source Port*, the *Destination Port*, the *Length*, and the *Checksum* [12]. When only a direct point-to-point connection between a PC and a FPGA is considered, all of the above header fields are constant with the exception of the IPv4 *Total Length* and *Header Checksum* fields as well as the UDP *Length* field. The idea of storing constant fields of UDP point-to-point connections in lookup tables has been used before [4]. However, the required hardware resources are further reduced in our design because the calculation of the IPv4 checksum field can be implemented by a single subtraction.

During a transmission of an Ethernet packet from the FPGA to the PC, all static/constant fields are retrieved directly from the HLUT. The three variable fields (*Total Length, Header Checksum, Length*) can be calculated at low cost using two additions and one subtraction. The initial architecture contained three hard-coded values that were used as the first operand of the two adders and the subtracter, while the second operand in all three operations was the length of the user data. The hard-coded values were set to the minimum IPv4 *Total Length* and UDP *Length* values, that is, the offsets to the respective header sections and the IPv4 Header *Checksum* field of an Ethernet packet without any user data attached to it. Based on the amount of data (number of bytes) transferred with the packet, the corresponding length fields are calculated by two additions. The subtraction is used to calculate the correct IPv4 *Header Checksum* as a function of the IPv4 *Total Length* field. A more detailed description of this procedure and a proof that a single subtraction is sufficient can be found in [1].

Figure 1 illustrates the versatile UDP/IP core architecture. The block diagram of the TRANSMITTER and a description of the RECEIVER components are provided in [1]. The new RST_U, CONFIG_U, and HLUT CONTROL components have been integrated to increase the flexibility of the core.

The RST_U component monitors the first byte of the user data in every incoming packet. The purpose of this unit is to detect a dedicated byte-code that triggers a reset of the HLUT CONTROL. A similar function (also using a dedicated byte-code) is implemented in the CONFIG_U component for detecting a *configuration enable code* that triggers the HLUT initialization process. The demand for receiving the *reset* and *configuration enable* codes in distinct packets represents a less error-prone solution, especially with respect to the fact that the UDP/IP core might not always be used with the provided software interface.

The HLUT CONTROL is implemented as a 5-state FSM (Finite State Machine). When the device is configured, the FSM transitions from the *default reset state* to the *idle state*. It will remain in *idle state* until a *configuration enable* signal arrives from the PC. Upon arrival, the FSM transitions to *pre-configuration state* and waits for the next incoming packet. This packet is used as reference packet to retrieve the required static fields and must not contain any user data since the IPv4 *Header Checksum* value of the packet will be stored in a respective 16-bit register. When this empty packet has arrived, the FSM then switches to *configuration state*. In this state, all incoming header fields are redirected to the HLUT that resides in the TRANSMITTER along with write addresses and write-enable signals. In addition to controlling the HLUT initialization process, the HLUT CONTROL component also stores the value of the IPv4 *Header Checksum* (see above). After these initialization steps, the FSM transitions to the *lock state* which then allows the TRANSMITTER to transmit regular user data. To prevent the transmission of faulty packets, the TRANSMITTER can not initiate any transmissions to the PC if the HLUT CONTROL is not in the *lock state*.

These additional components, allow to setup and use the UDP/IP core for communication with any PC in a seamless manner, that is, without requiring manual code changes. The only requirement for the platform to work properly is that the PC must transmit three packets to the FPGA that contain the reset code, the configuration code, and one packet that does not contain any data. This can be easily encapsulated in an initialization function. We describe the implementation of such a general communication platform that hides the complexity of the underlying design (based on this improved version of the UDP/IP core) in the following section.

IV. PC ↔ FPGA Communication Platform

In the following, we describe the software/hardware interface and a simple communication protocol that provide (in conjunction with the UDP/IP core) a complete PC ↔ FPGA communication solution. The key design objective is to hide the complexity of (i) setting up a PC ↔ FPGA connection and (ii) transmitting and receiving variable data types. In Section IV-A, we introduce a simplified protocol for configuring the UDP/IP core and transmitting user data.

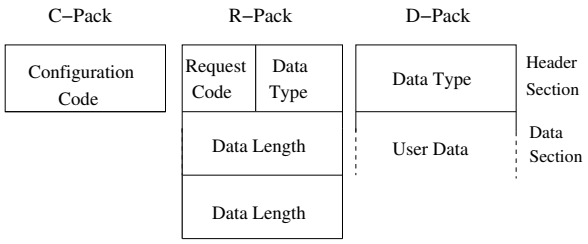

Figure 2. Supported packets in the Configuration and Data Protocol.

In Section IV-B, the hardware interface is described, while Section IV-C covers the software interface.

A. Configuration and Data Protocol

We propose a *Configuration and Data Protocol* (henceforth denoted as CDP) which offers a simple communication model for configuring the system *and* transferring variable size user data. However, in analogy to the User Datagram Protocol (UDP [12]), it does not provide reliability and data integrity mechanisms. There exist three types of packets: configuration packets, data packets, and data request/retrieval packets. In CDP, we henceforth denote a configuration packet as *C-Pack*, a data request packet as *R-Pack*, and a data packet as *D-Pack*. Figure 2 illustrates the three packet formats.

A *C-Pack* consists of a header field with a length of a single byte that contains a configuration code. Two configuration codes are supported in order to reset and enable the configuration process of the UDP/IP core (see Section III). A *R-Pack* consists of a three-byte-long header field that contains a data request code, a data type, and a data length field. Using the *R-Pack*, the software on the PC can initiate a transmission of data from the FPGA back to the PC. Finally, the *D-Pack* consists of a byte-long Data Type header field followed by the user data.

B. Hardware Interface

On the FPGA side, our implementation (Figure 3) provides a well-defined interface that allows the reconfigurable architecture to transmit and receive all supported data types. Furthermore this interface provides all the required connections to the Media Access Controller (MAC).

For receiving packets, two active-high signals indicate the start and end of data. Within the time-frame defined by these signals, a valid data signal is deployed to denote the clock cycles during which the data buses contain valid user data. There exist four data buses with different lengths that allow for receiving character data over the 8-bit bus, short integer data over the 16-bit bus, integers and floats over the 32-bit bus, and long integers and doubles over the 64-bit bus. An additional 3-bit data type signal indicates which of the four data buses contains valid data at each point in time. Three bits are required to select among buses *and* to distinguish

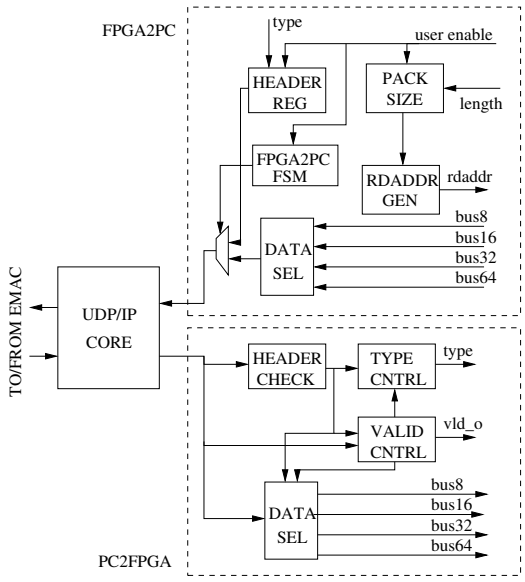

Figure 3. Hardware design of the communication platform.

between the respective data types (e.g., 32-bit integers or floats on the 32-bit bus).

To transmit data from the reconfigurable architecture to the PC, a transmission enable pulse is required. In synchrony with the enable pulse, the input type port must contain the respective *Data Type* code (e.g., character, short integer, integer, etc.) and the length port must contain the number of data items to be sent. The address generator component will then compute respective read addresses on the *rdaddr* bus that must be connected to on-chip memory with a latency of one clock cycle. The *Data Sel* component selects between the four data input buses to transmit the specified data type. Depending on the data transmission type (e.g., character, short integer, integer, etc.), the read addresses are generated at different speeds. Therefore, the user design (reconfigurable architecture) is notified when the transmission has been completed by a *transmission over* signal. For the sake of simplicity, the *transmission over* signal as well as the *start* and *end of data* signals have been omitted from Figure 3.

C. Software Interface

The software interface offers three categories of functions that are used for configuration, sending data, and receiving data. The configuration (initialization) functions are used to establish an initial connection with the FPGA by transmitting *C-Packs*. The data transmission functions are used for sending arrays of C data types (characters, short integers, integers, floats, long integers, and doubles) to the FPGA. This is accomplished by transmitting *D-Packs*. The third and final class of functions allows for receiving basic data type arrays. *R-Packs* can be transmitted using the respective data reception function.

For sending data from the PC to the FPGA, the software interface splits up the arrays provided by the user into separate UDP packets and can optionally also perform endian swapping [our current target PC platform (x86 32- or 64-bit Linux) uses little endian]. For receiving data, the software implementation can concatenate multiple UDP packets into larger arrays (including endian conversion) which are then returned to the application that invoked the receive function. In contrast to the functions for sending data to the FPGA, the receive functions make use of a FIFO receive buffer and a background reader thread to improve performance (see below). The background reader thread constantly reads incoming UDP packets and stores them in a FIFO list.

The array splitting and re-assembly part works as follows: The Ethernet standard defines a maximum packet size (MTU=maximum transmission unit) that has to be supported by standardized Ethernet equipment. Thus, in the worst case, an IP packet can carry a maximum payload of only 1,500 bytes. Note that, for specific hardware interfaces, the MTU can potentially be larger. The Intel network interface we used in our tests supports a MTU of 9,000 bytes. Because the MTU is hardware-dependent, we provide an option to explicitly set the appropriate MTU. If the user intends to transmit an array that exceeds the MTU (e.g., 1,000 double values, which require 8,000 bytes while the MTU is 1,500 bytes), the data is split into separate UDP packets which are sent one-by-one to the FPGA. Conversely, if the user requests to receive 1,000 double values, the library expects the data to be split into sufficiently small UDP packets by the FPGA. In this case, the PC interface will read as many UDP packets as required to complete the receive request and re-assemble the array from the individual packets.

The background reader thread has been introduced to alleviate problems that can occur when the PC is not able to receive UDP packets that are sent by the FPGA fast enough. This can happen in the following scenario (assuming a single-threaded implementation on the PC): The PC sends a UDP packet to the FPGA for retrieving a data array that can consist of multiple UDP packets. After the request packet (*R-Pack*) has been sent, a system call for receiving the first reply packet is issued on the PC. Our UDP/IP core has very short latencies between receiving a *R-Pack* and sending back the reply packets. Therefore it is possible, and indeed quite common according to our experiences, that the reply packets from the FPGA are already on their way *before* the PC is ready to receive them. When using a standard Linux configuration and consumer network hardware, only limited buffer space for incoming UDP packets is typically available. Thus, it is likely that some of the reply packets from the FPGA are discarded *before* they can actually be read by the user process on the PC. To solve this problem, it is necessary to already issue the system call of the receive function prior to sending an *R-Pack* to the FPGA. The background reader approach solves this problem as follows:

A dedicated thread is started by the background reader mechanism upon initialization of the communication library. This thread constantly reads incoming packets and stores them in a sufficiently large FIFO buffer. When the user sends a *R-Pack*, the background reader thread will already have issued the receive call and can therefore reliably receive the reply packets. The user program can then retrieve the packets from the FIFO buffer in the same order as they were received by the background reader thread. The background reader thread works transparently in the background and does therefore not require any multi-threaded programming in the application that uses the communication interface.

The background reader is implemented in C++ using the boost (www.boost.org) libraries for multi-threading. To ensure portability, we provide an ANSI-C standard-compliant application interface for the background reader. The remaining functions for FPGA initialization and sending data packets are directly implemented in ANSI-C.

V. Experimental Results

Initially, we verify the functionality of the enhanced UDP/IP core and the PC $\leftrightarrow$ FPGA hardware interface (Section V-A). Thereafter, we provide a performance and resource utilization comparison (Section V-B) with the previous version of our UDP/IP core and the *minimum* commercial UDP/IP core [4]. Finally, we describe three application test cases that demonstrate the functionality of our communication platform (Section V-C).

A. Verification

To verify the correctness of the enhanced UDP/IP core implementation, we performed post-place-and-route simulations *and* tests on a real FPGA board (HTG-V5-PCIE development platform) connected to a DELL Latitude e4300 notebook running Linux via a standard CAT5 twisted-pair Ethernet cable. We used Chipscope Pro Analyzer to monitor the input and output ports of the UDP/IP core which were connected to the EMAC Local Link Wrapper. In addition, we monitored the incoming and outgoing signals of the PC2FPGA and FPGA2PC components (see Figure 3) to verify correctness of the hardware interface on the FPGA.

B. Resource Utilization

The communication system was mapped on a Virtex 5 SX95T-1 FPGA. Table I shows the resources occupied by the proof-of-concept and enhanced versions of our UDP/IP core. The table shows that the enhanced UDP/IP core occupies 56% more FPGA slices than the initial implementation. The additional hardware resources required reflect the price that has to be paid for increasing flexibility. Nevertheless, Table II also shows that our improved UDP/IP core still outperforms the commercial solution [4] with respect to hardware resource utilization. To conduct a fair comparison between our implementation and the one described in [4], we mapped the UDP/IP core on the same FPGA (Spartan 3 XC3S200-4) that was used by Löfgren *et al.* in their paper. On this FPGA, our UDP/IP core architecture, unlike the commercial design, can operate at Gigabit speed while occupying almost 2.8 times less hardware resources.

	UDP/IP Core		PC $\leftrightarrow$ FPGA
	Initial	Versatile	Platform
Slice Registers	79	127	408
Slice LUTs	155	195	562
Occupied Slices	67	105	272
Frequency (MHz)	261	262	181

Table I
Resources occupied on a Virtex 5 SX95T-1 FPGA by the initial and versatile UDP/IP core implementations and the PC $\leftrightarrow$ FPGA platform hardware part (incl. UDP/IP core).

	Löfgren core	our UDP/IP core
Xilinx Slices	517	184
Xilinx BRAMS	3	0
FMax(MHz)	90.7	128.8
Duplex Mode	FULL	FULL
Length(Bytes)	256	1472*
Speed(Mbps)	10/100	10/100/1000

Table II
Performance comparison on a Spartan3 XC3S200-4 FPGA: UDP/IP core [4] VS our improved implementation. *Limitation imposed by the EMAC configuration (Xilinx EMAC wrapper files generated by Core Generator).

C. Test Applications

To demonstrate the potential of our communication platform, we implemented the following three test cases:

Basic Test: In the first experiment, we offload a simple `for`-loop that calculates the natural logarithm of the values stored in an array of floats to the FPGA. We used the respective platform function to send the input array of floats to the FPGA. On the FPGA, we connected the 32-bit output bus to a LAU (Logarithm Approximation Unit [13], [14]) for computing the logarithms on the FPGA. The output values of the LAU are stored in a memory block on the FPGA. Finally, the respective software function for requesting an array of floating-point values was used to retrieve this array from the FPGA.

Phylogenetic Alignment Kernel: In the second experiment, we focus on verifying a more complex reconfigurable architecture that we designed for accelerating a phylogeny-aware short read alignment kernel [15]. Here, we used the respective communication functions for sending character arrays in order to transmit bit-encoded DNA sequences from the PC to the FPGA. The results of this phylogeny-aware short read alignment process are 16-bit integers/scores that are subsequently retrieved by the PC through the 16-bit

FPGA2PC data bus.

Phylogenetic Parsimony Kernel: The reconfigurable architecture we used for the third experiment was designed to accelerate the parsimony kernel [16] for building phylogenetic trees as implemented in the open-source `Parsimonator` code [17]. The data transfers to the FPGA are similar to those in the alignment kernel (see above), since we need to transfer bit-encoded DNA character arrays to the FPGA. Thus, the 8-bit bus provided by the PC2FPGA component was used to receive nucleotide sequence data as well as to trigger parsimony computation requests on the FPGA. The resulting parsimony scores (32-bit integer values) are then retrieved by the parsimonator software on the PC through the FPGA2PC 32-bit bus.

In these test scenarios (as for the phylogenetic kernels), thousands of input arguments and hundreds of scores were reliably transmitted back and forth within a few seconds using our communication platform, thereby allowing for extensive real-world testing of our reconfigurable systems.

VI. Conclusion

We presented a significantly enhanced version of our widely-used open-source UDP/IP core for efficient direct PC $\leftrightarrow$ FPGA communication. The improved version allows for automatic configuration of the UDP/IP core. In addition, we introduce a light-weight communication protocol and provide an appropriate software/hardware interface and communication library implementation. This library allows for easy integration with C or C++ codes. Our versatile hardware/software interface completely hides the complexity inherent to establishing communication (configuring the UDP/IP core) and transmitting arrays of variable size containing standard C data types such as characters, short integers, integers, floats, long integers, and doubles.

Future work will focus on providing a more generic implementation that can be mapped to any device by any vendor. We also plan to extend the communication protocol to compensate for packet loss and/or corruption by means of automatic packet re-transmission (e.g., using time-outs).

References

[1] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Efficient PC-FPGA Communication over Gigabit Ethernet," in *CIT*, 2010, pp. 1727–1734.

[2] Xilinx, "Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions." [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf

[3] Altera, "PCI Express Compiler: x1, x4, and x8 MegaCore Functions." [Online]. Available: http://www.altera.com/products/ip/iup/pci-express/m-alt-pcie8.html

[4] A. Löfgren, L. Lodesten, S. Sjöholm, and H. Hansson, "An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity," in *Proceedings of the 23rd IEEE NORCHIP Conference*, November 2008, pp. 94–97.

[5] W. Kühn, C. Gilardi, D. Kirschner, J. Lang, S. Lange, M. Liu, T. Perez, S. Yang, L. Schmitt, D. Jin, L. Li, Z. Liu, Y. Lu, Q. Wang, S. Wei, H. Xu, D. Zhao, K. Korcyl, J. Otwinowski, P. Salabura, I. Konorov, and A. Mann, "Fpga based compute nodes for high level triggering in panda," in *Journal of Physics: Conference Series*, vol. 119, 2008, pp. 22–27.

[6] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris, "An open tcp/ip core for reconfigurable logic," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, 2005, pp. 297–298.

[7] Y.-L. Lin, C.-P. Young, Y.-J. Chang, Y.-H. Chung, and S. A.W.Y, "Versatile pc/fpga based verification/fast prototyping platform with multimedia applications," in *Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE*, vol. 2, May 2004, pp. 1490–1495.

[8] D.-S. Kang, S. Y. Hwang, K.-S. Jhang, and K. Yi, "A low cost and interactive rapid prototyping platform for digital system design education," *Microelectronics Systems Education, IEEE International Conference on/Multimedia Software Engineering, International Symposium on*, vol. 0, pp. 95–96, 2007.

[9] P. Schumacher, M. Mattavelli, A. Chirila-Rus, and R. Turney, "A software/hardware platform for rapid prototyping of video and multimedia designs," *System-on-Chip for Real-Time Applications, International Workshop on*, vol. 0, pp. 30–33, 2005.

[10] P. Lieber and B. Hutchings, "FPGA Communication Framework," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 69–72.

[11] J. Postel, "Internet protocol," RFC 791 (Standard), Internet Engineering Task Force, Updated by RFC 1349, September 1981.

[12] ——, "User datagram protocol," RFC 768 (Standard), Internet Engineering Task Force, August 1980.

[13] N. Alachiotis and A. Stamatakis, "Efficient floating-point logarithm unit for FPGAs," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[14] ——, "A Vector-Like Reconfigurable Floating-Point Unit for the Logarithm," *International Journal of Reconfigurable Computing*, 2011.

[15] N. Alachiotis, S. Berger, and A. Stamatakis, "Accelerating Phylogeny-Aware Short DNA Read Alignment with FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 226–233.

[16] N. Alachiotis and A. Stamatakis, "FPGA Acceleration of the Phylogenetic Parsimony Kernel?" in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*. IEEE, 2011, pp. 417–422.

[17] A. Stamatakis, *Parsimonator*, http://www.exelixis-lab.org/software.html.