UNIVERSITÉ DE MONTRÉAL

IMPLEMENTATION OF ULTRA-LOW LATENCY AND HIGH-SPEED COMMUNICATION CHANNELS FOR AN FPGA-BASED HPC CLUSTER

ROBERTO SANCHEZ CORREA
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
MAI 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IMPLEMENTATION OF ULTRA-LOW LATENCY AND HIGH-SPEED COMMUNICATION CHANNELS FOR AN FPGA-BASED HPC CLUSTER

présenté par : SANCHEZ CORREA Roberto
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. SAVARIA Yvon, Ph. D., président
M. DAVID Jean Pierre, Ph. D., membre et directeur de recherche
M. BOIS Guy, Ph. D., membre

DEDICATION

To my wife Ely and my parents,
Thank you for all.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, professor Jean Pierre David, for giving me the possibility to be part of his research lab and seeing in me the potential to complete this work. I am very thankful for all of his advice and support in both professional and personal areas. His professionalism and research skills are worthy of inspiration. Professor Jean Pierre is the kind of professor that tells you : ≪*"if you have problems or you are stuck, don't hesitate to knock on the door of my office"*≫, and he means it ; and I am very thankful for that.

I also would like to thank the "Groupe de Recherche en Microélectronique et Systèmes" (GR2M) and the Electrical Engineering Department for funding this research. It has been an amazing journey at Poly.

To my family and my friends, thank you guys for your support and concerns. To my beautiful wife, thank you for being a constant source of support and encouragement, for waking always by my side and for helping me during the challenges of life. To my parents, thank you for all you have done for me. To all of you, this work is as mine as yours.

Finally, I would like to thank Federico Montano, Ph.D. student and friend from the lab. Thank you for your help and your advice, and for the great times we had at the lab.

RÉSUMÉ

Les clusters basés sur les FPGA bénéficient de leur flexibilité et de leurs performances en termes de puissance de calcul et de faible consommation. Et puisque la consommation de puissance devient un élément de plus en plus importants sur le marché des superordinateurs, le domaine d'exploration multi-FPGA devient chaque année plus populaire.

Les performances des ordinateurs n'ont jamais cessé d'augmenter mais la latence des réseaux d'interconnexion n'a pas suivi leur taux d'amélioration. Dans le but d'augmenter le niveau d'abstraction et les fonctionnalités des interconnexions, la complexité des piles de communication atteinte à nos jours engendre des coûts et affecte la latence des communications, ce qui rend ces piles de communication très souvent inefficaces, voire inutiles. Les protocoles de communication commerciaux existants et les contrôleurs d'interfaces réseau FPGA-FPGA n'ont la performance pour supporter ni les applications à temps critique ni un partitionnement étroitement couplé des systèmes sur puce. Au lieu de cela, les approches de communication personnalisées sont souvent préférées.

Dans ce travail, nous proposons une implémentation de canaux de communication à haut débit et à faible latence pour une grappe de FPGA. Le système est constitué de deux BEE3, chacun contenant 4 FPGA de la famille Virtex-5 interconnectés par une topologie en anneau. Notre approche exploite la technologie à transducteur à plusieurs gigabits par seconde pour l'obtention d'une bande passante fiable de 8Gbps. Le module de propriété intellectuelle (IP) de communication proposé permet le transfert de données entre des milliers de coprocesseurs sur le réseau, grâce à l'implémentation d'un réseau direct avec capacité de routage de paquets. Les résultats expérimentaux ont montré une latence de seulement 34 cycles d'horloge entre deux nœuds voisins, ce qui est un des plus bas parmi ceux rapportés dans la littérature.

En outre, nous proposons une architecture adaptée au calcul à haute performance qui comporte un traitement extensible, parallèle et distribué. Pour une plateforme à 8 FPGA, l'architecture fournit 35.6Go/s de bande passante effective pour la mémoire externe, une bande passante globale de réseau de 128Gbps et une puissance de calcul de 8.9GFLOPS. Un solveur matrice-vecteur de grande taille est partitionné et mis en œuvre à travers le cluster. Nous avons obtenu une performance et une efficacité de calcul concurrentielles grâce à la faible empreinte du protocole de communication entre les éléments de traitement distribués.

Ce travail contribue à soutenir de nouvelles recherches dans le domaine du calcul parallèle intensif et permet le partitionnement de système sur puce à grande taille sur des clusters à base de FPGA.

ABSTRACT

An FPGA-based cluster profits from the flexibility and the performance potential FPGA technology provides. Since price and power consumption are becoming increasingly important elements in the High-Performance Computing market, the multi-FPGA exploration field is getting more popular each year.

Network latency has failed to keep up with other improvements in computer performance. Complex communication stacks have sacrificed latency and increased overhead to achieve other goals, being in most of the time inefficient and unnecessary. The existing commercial off-the-shelf communication protocols and Network Interfaces Controllers for FPGA-to-FPGA interconnection lack of performance to support time-critical applications and tightly coupled System-on-Chip partitioning. Instead, custom communication approaches are preferred.

In this work, ultra-low latency and high-speed communication channels for an FPGA-based cluster are presented. Two BEE3s grouping 8 FPGAs Virtex-5 interconnected in a ring topology, compose the targeting platform. Our approach exploits Multi-Gigabit Transceiver technology to achieve reliable 8Gbps channel bandwidth. The proposed communication IP supports data transfer from coprocessors over the network, by means of a direct network implementation with hop-by-hop packet routing capability. Experimental results showed a latency of only 34 clock cycles between two neighboring nodes, being one of the lowest in the literature.

In addition, it is proposed an architecture suitable for High-Performance Computing which includes performing scalable, parallel, and distributed processing. For an 8 FPGAs platform, the architecture provides 35.6GB/s off-chip memory throughput, 128Gbps network aggregate bandwidth, and 8.9GFLOPS computing power. A large and dense matrix-vector solver is partitioned and implemented across the cluster. We achieved competitive performance and computational efficiency as a result of the low communication overhead among the distributed processing elements.

This work contributes to support new researches on the intense parallel computing fields, and enables large System-on-Chip partitioning and scaling on FPGA-based clusters.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

10GbE	10-Gigabit Ethernet
BEE	Berkeley Emulation Engine
BER	Bit Error Ratio
CDR	Clock Data Recovery
CRC	Cyclic Redundancy Check
FMC	FPGA Mezzanine Card
HPC	High Performance Computing
IBERT	Integrated Bit Error Ratio Tester
IP	Intellectual Property
MAC	Media Access Controller
MGT	Multi-Gigabit Transceiver
MPSoC	MultiProcessor System-on-Chip
NIC	Network Interface Controller
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PCS	Physical Coding Sublayer
PE	Processing Element
PHY	Physical Layer Device
PMA	Physical Medium Attachment
PRBS	Pseudo-Random Bit Sequences
SATA	Serial Advanced Technology Attachment
SerDes	Serializer/Deserializer
TCP	Transmission Control Protocol
XAUI	Ten Gigabit Attachment Unit Interface
XGMII	Ten Gigabit Media Independent Interface

LIST OF APPENDIXES

CHAPTER 1 INTRODUCTION

Numerous applications like mining exploration, weather forecasting, molecular dynamics modeling, financial computing, among others, demand increasing amounts of processing capability. In many cases, the computation cores and data types are suited to field-programmable gate arrays (FPGA). A solution is thus hardware acceleration which increases processing with application-specific coprocessors.

FPGAs offer tremendous performance potential and on-chip facility for diverse applications. Modern high-end FPGAs have not only millions of configurable "gates" and interconnections, but also large numbers of hard-wired components. In addition, they have hundreds of general-purpose I/O pins and tens of transceivers for high-speed serial interconnect. FPGAs offer speed-up by hardware and software co-design environments and parallelization, with reduced energy power consumption over other platforms like GPUs or CPUs. Endless design alternatives are possible and may go from pipelines structures with hundreds of stages to soft/hard configurable processors, systolic arrays, etc. Their flexibility enables the designers to realize almost any computer configuration that can be imagined and use any form of concurrent execution to suit the applications. FPGAs work well when applications contain enough parallelism to compensate for their relatively low clock speeds.

Parallel computing is a computational technique where many calculations or the execution of processes are carried out simultaneously. Parallelism has been employed for many years, especially in the High-Performance Computing (HPC) field. Recently, such paradigm has aroused a renewed interest due to the physical constraints preventing frequency scaling.

Advanced research in HPC has moved beyond the processing capacity of single computers to supercomputers with different architectures. Figure 1.1 shows the top 500 supercomputers' architecture tendency from the performance and the system points of view. The last trend points out to clustering of commodity devices as the predominant architecture over others like Massively Parallel Processing (MPP) and Symmetric Multiprocessing (SMP). Clustering has been gaining more and more popularity each year since about 2000. Nowadays, 86% of the top 500 supercomputers around the world utilize clustering architecture while the rest focus on Massively Parallel Processing. Moreover, it can be seen that clustering has proven higher performance compared to others. For instance, from the total amount of floating point operations per second (expressed in GFLOPS in Figure 1.1 left) achievable by all 500 supercomputers together, 58.89% corresponds to cluster architectures.

In this context, it is natural that FPGA-based clusters have been focused by the HPC

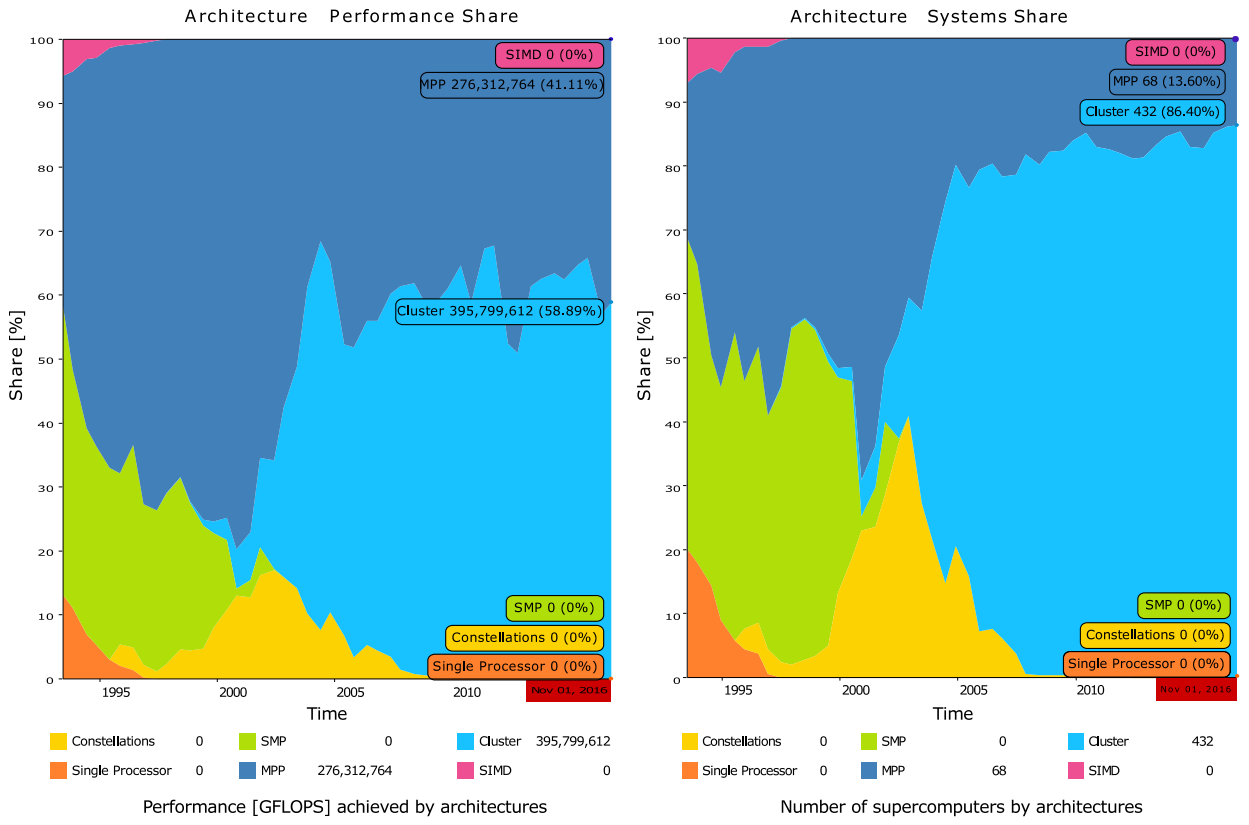

Figure 1.1 High-Performance Computer architectures over time [5] (Nov2016).

community. In addition, since price and power consumption are becoming increasingly important elements in the HPC market, the multi-FPGA exploration field is growing rapidly. An FPGA-based cluster profits from the flexibility and the performance potential this technology accommodates. It also enables large System-on-Chip partitioning which provides a great deal to create ideal machines for specific applications.

In an HPC cluster, there are three main ways of increasing computational performance: incrementing the number of processing units, improving application algorithm and reducing nodes' interconnections overhead. Loosely coupled computing nodes generally don't communicate too much, so they are not affected by the interconnection metrics. In opposition, tightly coupled nodes do require low latency and high-bandwidth message passing, otherwise, the network becomes the bottleneck of the entire system and the overall performance of the machine may be compromised. Hence, cluster computing scalability and the degree of parallelism is limited by the performance of the interconnect network.

The network's latency property is very important for round-trip communication patterns.

Despite that, network latency has failed to keep up with other improvements in computer performance [9]. Complex communication protocol stacks have affected the latency and increased the overhead, being most of the time inefficient and unnecessary. That is the case of commercial off-the-shelf communication protocols and Network Interfaces Controllers (NIC) when they are used for FPGA-to-FPGA interconnection. It has been demonstrated their lack of performance to support time-critical applications and tightly coupled SoC partitioning [10]. Therefore, custom communication approaches are preferred when targeting high-performance interconnections. This statement is broadened beyond the multi-FPGA field to the supercomputers field where custom interconnection deployments are being widely used.

Figure 1.2 presents the interconnection families of the top 500 supercomputers all around the world. As can be seen, the custom interconnects represents the third most utilized interconnect family behind Infiniband and 10Gbps Ethernet. 14.2% of the supercomputers has adopted customizable approaches for communicating their nodes. In terms of performance (Figure 1.2 right), the custom interconnect family takes credits for 47.7% of the global combined performance in the top 500 list. In other words, most of the achievable computing power presently relies on custom approaches.

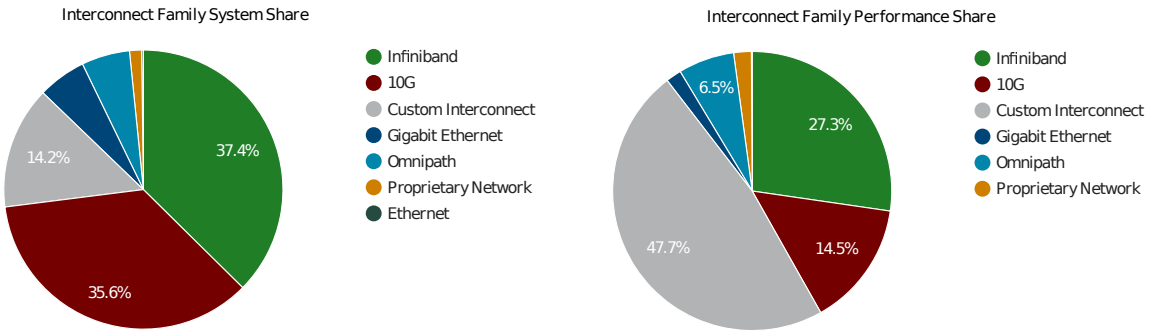

Figure 1.2 HPC Interconnect families [5] (Nov2016).

Standard and Proprietary interconnects, i.e. 10Gbps Ethernet and Infiniband, are rapidly deployed compared to custom. However, there is a performance penalty to pay and to be considered when building supercomputers. Custom interconnects benefit from a personalized solution for a specific application and/or machine but at the expenses of an increased effort and time delay in the building process. There are different levels on which a custom solution may stand depending on the degree of customization. A fully custom interconnect system personalizes all the layers in the communication stack including the physical device and

the transmission medium. On the other hand, a partially custom interconnect selects the components, functionalities, and layers to be personalized while using commodity when it satisfies the project's needs.

Many challenges have to be faced when designing high-performance custom interconnect network. Each of them is associated with a level of abstraction (layer) in the communication stack. For instance, at the physical level, the link's power consumption, bandwidth, latency, reliability and fault tolerance should be considered. Similarly, at higher communication levels, the network topology and its scalability, the routing algorithm, the congestion control and the programming models have to be addressed as well. Handling these aspects while maintaining high-performance features (i.e. low end-to-end latency, high channel bandwidth and small network diameter) issues a tremendous challenge for the designers.

Once a High-Performance Computing cluster is built, the next step is testing and benchmarking. Running a standard benchmark or a standard parallel application in an FPGA cluster is not straightforward. Since they are written based on a parallel programming model, they need hardware and software support implemented in the FPGAs. This means an operating system should run on a microprocessor and a parallel programming library should be adapted to work properly with the custom interconnect's drivers. Rather than this complicated solution for testing and benchmarking, a simpler approach could be implementing a low-level version of the benchmark or the application instead of its standard counterpart.

A closer look at most common scientific and engineering applications, entailing huge amount and intensity of computing power, shows that they generally have the same basic algebraic operations. This is extended to standard benchmarks in which the computing time elapsed when solving systems of linear equations is measured for estimating the machine performance. Therefore, the implementation of an algebraic kernel partitioned and distributed all across an FPGA cluster allows correctness testing of the cluster's computational units and the interconnect network while still giving an approximate idea of its performance potential. In this context, the matrix-vector multiplication raises as one of the preferred low-level applications to be run on a multi-FPGA machine.

This research is framed as the acceleration of the communication system for the *Goliath* supercomputer located at Polytechnique Montréal. Since *Goliath* is an heterogeneous machine with different types of computational units and interconnect networks, we focus on the 10Gbps FPGA-to-FPGA communication. This research stands on the aforementioned facts that the commercial off-the-shelf communication solutions lack performance to support time-critical applications and tightly coupled System-on-Chip (SoC) partitioning for multi-FPGA machines. In addition, custom communication approaches are being preferred

over standard deployments when high-performance interconnects are required. In *Goliath*'s FPGA-to-FPGA interconnect network, the utilization of a custom communication Intellectual Property (IP) module with low latency and high-bandwidth properties will reduce the overhead associated with data movement and will increase the processing efficiency of the machine. Moreover, it will adapt the communication to satisfy *Goliath*'s needs.

Our main objective is the implementation of communication channels for an FPGA-based High-Performance Computing (HPC) cluster targeting ultra-low latency and high-speed interconnections. We aim at tight integrating of the FPGAs as a result of this research, allowing large SoC partitioning. At the end of this work, a communication IP will be delivered as well as a hardware and software system for HPC support.

The specific objectives are:

1. Design and implement a communication IP that serves as a 10Gbps Network Interface Controller with low overhead features and direct network support.
2. Integrate the IP into a hardware and software codesign environment providing network accesses from both systems.
3. Build and test a tightly coupled FPGA-based cluster supported by the communication IP.
4. Accelerate an application by parallelizing and distributing workloads on the cluster.

This thesis is organized as follows.

— Chapter 2 provides a brief background on this work as part of the *Goliath* supercomputer research program.
— Chapter 3 presents a literature review of the existing implementations on high-speed serial communication between FPGAs.
— Chapter 4 refers to the proposed Intellectual Property module for FPGA-to-FPGA communication, its architecture, implementation and experimental results.
— Chapter 5 explores the inter-FPGA communication channels under a High-Performance Computing environment. A large matrix-vector SoC is partitioned across an FPGA-based cluster for accelerating by parallelization.
— Chapter 6 presents the conclusions of this work and a state-of-the-art comparison.
— Appendix A provides the transceivers' wrapper source file modification after the serial links tune up process.
— Appendix B shares a VHDL description of the Native Port Interface controller for 32 double word burst read transfers from a Multi-Port Memory Controller.

CHAPTER 2 BACKGROUND

This chapter describes the environment on which this research stands. It also provides the basic knowledge for the comprehension of the rest of the document and the work carried out.

2.1 Project background: The Goliath supercomputer

VESI (Figure 2.1) is an architecture born at Politechnique Montréal. Its creators, Professor Jean Pierre David and Professor Yvon Savaria, aim at the integration of different high-performance computing platforms for scientific and engineering researches. In this context, the Goliath supercomputer emerges as the materialization of VESI.

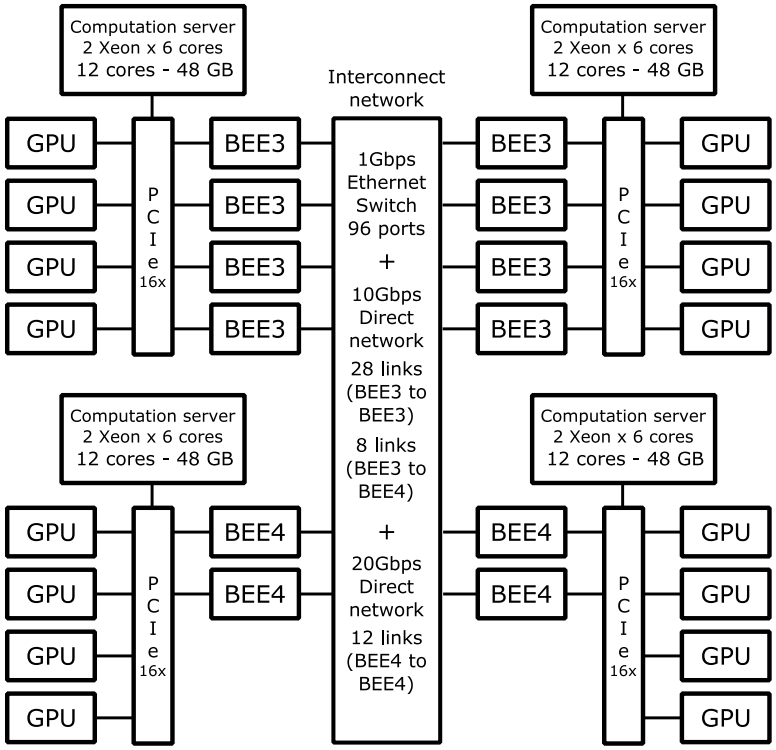

Figure 2.1 VESI architecture proposed by Professor Jean Pierre David et Yvon Savaria at Polytechnique Montréal.

The Goliath supercomputer is an heterogeneous machine that integrates several computational servers equipped with two Intel® Xeon six-core CPUs each. The servers are connected, throughout a PCI-Express 16x bus, to four NVidia® Tesla C2050 GPUs. There are also two types of multi-FPGA machine, each of them hosting four FPGAs: the Berkeley Emulation

Engine 3 (BEE3) and its next generation the BEE4. All of the computational units in the architecture communicate by means of a combination of interconnect networks, i.e. the Ethernet 1Gbps, the 10Gbps and 20Gbps direct links, and the PCIe bus. The tremendous computing potential and the flexibility of Goliath supercomputer make it an ideal machine for High-Performance Computing applications. However, having a highly effective interconnect system supporting the communication between the computational units remains imperative to avoid limiting the overall architecture performance.

This research is framed as the acceleration of the communication system in Goliath. It is the continuity of the Ethernet 1Gbps interconnect project developed for the Goliath's FPGAs at Professor David's research lab.

The 1Gbps Ethernet project relies on an indirect network supported by a centralized switch. As shown in Figure 2.2, the switch provides full connectivity to all of the computational units having an RJ-45 interface. From the FPGAs point of view, the communication stack is accelerated with a UDP/IP hardware module and a 1Gbps Ethernet MAC.

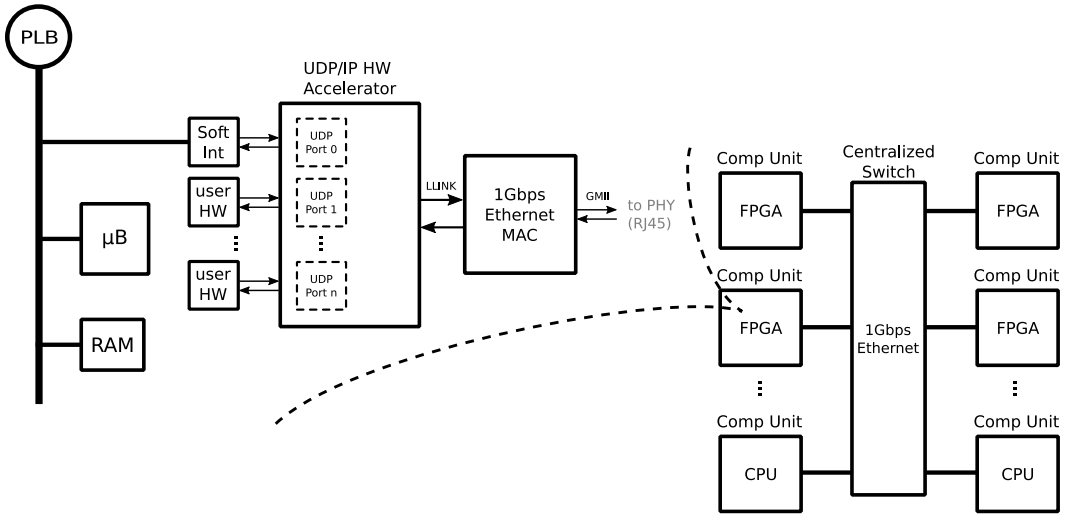

Figure 2.2 Goliath's 1Gbps Ethernet communication solution.

The UDP/IP module provides several UDP ports for transmitting, receiving and queueing UDP packets across the 1Gbps network. Each port has an independent interface. They are used to indicate the UDP/IP protocol fields and to give access to the internal transmission and reception buffers (FIFOs). The ports' buffers are multiplexed and demultiplexed, dividing the physical channel into several logical channels. Therefore, any number of user modules along Goliath's FPGAs can be assigned to a UDP port for enabling accessing time to the network. A software interface module and its associated driver facilitate the communication from a microprocessor environment. Within the software interface module, there is a Direct

Memory Access controller which bridges the UDP/IP port to the main memory system.

Closer to the network side, the Ethernet MAC module interfaces the PHY and implements the standard Ethernet MAC function. With this interconnection system, a user can have both hardware and software facilities to communicates across Goliath. However, the overall performance of the supercomputer is compromised because of the well-known limitations of Ethernet solutions. More specifically, the reduced channel bandwidth along with the high-overhead of the Ethernet protocol and its latency penalty, affect directly the system performance by becoming the bottleneck. This drawback may not be important if loosely coupled computational units are intended in Goliath. But, since high-communicating parallel and distributed computational units are in fact intended, then there is clearly a limitation on this interconnection system.

To overcome the presently communication bottleneck in Goliath supercomputer, the next step is the 10Gbps FPGA-based direct network development, on which this research finds its foundation. The Ethernet 1Gbps environment sets the basis and the starting point for pushing further Goliath's interconnection and for solving its aforementioned limitation.

2.2 Basic interconnection network concepts

Some of the basic concepts mentioned all along this document are following clarified [11].

— *Channel Bandwidth*
  Maximum capacity of a network channel (in bit/s). The bandwidth of a link is the theoretical amount of data that could be sent over a channel without regard to practical consideration (latency, protocols) during a specified time period.

— *Aggregate Bandwidth*
  The aggregate bandwidth is defined as the channel bandwidth multiplied by the number of ports.

— *Throughput*
  It refers to the rate of information arriving or being transmitted at a particular point in a network system. It is the rate of successful payload data delivered over a communication channel (payload transmitted per second). In other words, the throughput is how much payload data actually does travel through the channel. The overhead reduces throughput and increases latency. It can be measured at any layer in the OSI model.

— *Overhead*
  Sending a payload of data (reliably) over a communication network requires sending

more than just the desired payload data. It also involves sending various control data required to achieve the reliable transmission of the desired data. The control information is the overhead.

— *Latency*

The time required to deliver a unit of data through the network, measured as the elapsed time between the injection of the first bit at the source to the ejection of the same bit at the destination.

— *Radix*

Refers to the number of ports of a switch.

— *Hop count*

The number of links traversed between source and destination.

— *Network Diameter*

It is the longest of the shortest path (distance) between any two nodes in a network. In other words, the network diameter is the largest number of nodes which must be traversed in order to travel from one node to another when only direct paths are considered (no detour or loops). It also indicates how long it will take at most to reach any node in the network [12]. Sparser networks will generally have greater diameters. Therefore, the network diameter should be relatively small if the latency is to be minimized.

Table 2.1 Popular Interconnection Networks Diameters [1].

Network	Number of Nodes	Network Diameter
Linear Array	k	$k-1$
Ring	k	$k/2$
2D mesh	k^2	$2k-2$
2D torus	k^2	k
3D mesh	k^3	$3k-3$
3D torus	k^3	$3k/2$
Binary tree	2^l-1	$2l-2$
Hypercube	2^l	l

— *Indirect Network*

The Endpoints connect through a central switch unit. It has intermediate routing-only nodes. There is no direct connection between the Endpoints. Most modern networks are indirect network, i.e. Ethernet, Infiniband.

— *Direct Network*

The Endpoints provide multiple links towards the network side and integrate the switching unit. No centralized switching resource is required. Instead, distributed

switches as part of the network interface are used. Most common topologies are mesh, torus, and hypercube. The main drawback of direct networks is that it increases the number of hops to reach the destination. Latency increases linearly with the number of hops. Hop-based routing is used so that a fully connected network is not required.

— *Bit Error Ratio (BER)*

Bit Error Ratio of a physical link is given in $errors/bit$. It refers to the number of bit errors divided by the total number of transferred bits during the studied time interval. The Bit Error Ratio can be converted to Bit Error Rate by multiplying by the link bandwidth in bit/s.

— *Bit Error Rate (also BER)*

The Bit Error Rate is the number of bit errors of a physical link per unit of time.

CHAPTER 3 LITERATURE REVIEW

The multi-FPGA exploration field is getting more popular each year. The High-Performance Computing (HPC) community, for example, has been focusing on FPGA technology for acceleration at small scale i.e. task or algorithms with FPGA development boards, and at big scale i.e. complex system simulation with powerful FPGA-based clusters. Network deployment solutions have been applying FPGA technologies as well. The high-speed serial protocol support from FPGA vendors has offered to the Network Interface Controller (NIC) manufacturer a new exploration field in the growing market of communications. In this chapter, we review the state-of-the-art of high-speed interconnect networks technology on FPGAs. For that purpose, we have categorized the reviewed articles into two main groups based on similar approaches: pure multi-FPGA machines and FPGA-based network accelerators on heterogeneous machines. Later in this chapter, the recent trends in high-speed serial technology supported by FPGAs vendors are discussed. Finally, for our high-performance computing test, we review some implementations of FPGA-based matrix-vector multiplication kernels.

3.1 Pure multi-FPGA machines

In this section, we review some pure multi-FPGA machine implementations, their interconnect technologies, protocols and topologies; putting under the scope metrics as point-to-point latency, communication bandwidth, network diameter, network efficiency and the scalability of the solution.

Cube [13] is a 512-FPGA machine built of eight boards carrying 64 Xilinx's Spartan 3 FPGAs each. The machine is assembled as an 8x8x8 cluster, where each FPGA is considered a Processing Element (PE) of a systolic array interconnection. The communication between PEs is done through parallel lines on the Printed Circuit Board (PCB), which were designed to achieve a theoretical bandwidth of 6.4Gbps. The high cost and complexity due to the parallel routes in such a PCB assembly make this solution unpopular. Systolic arrays offer low latency communication between neighboring PEs but they lack flexibility.

The Reconfigurable Computing Cluster Project at the University of North Carolina at Charlotte [14] has as the main goal to investigate the feasibility of using FPGA platforms in building cost-effective HPC systems that will scale to a PetaFLOP. As part of the project, the *Spirit* cluster was built of 64 Xilinx ML-410 development boards hosting Virtex-4 FPGAs. An Architecture Independent REconfigurable Network (AIREN) addresses the question of

how to efficiently communicate among compute nodes, putting under the scope a high-speed, low latency and low overhead communication channels [15,16]. Figure 3.1 presents AIREN's block diagram for a Data Link hardware core and its switching module implementations. The Link layer core wraps around a Xilinx's Aurora protocol module. It uses two buffers for supporting back pressure flow control and for handling short interruptions in the data flow. Finite state machines monitor the buffers and the Aurora core prior proceeding to data transfers.

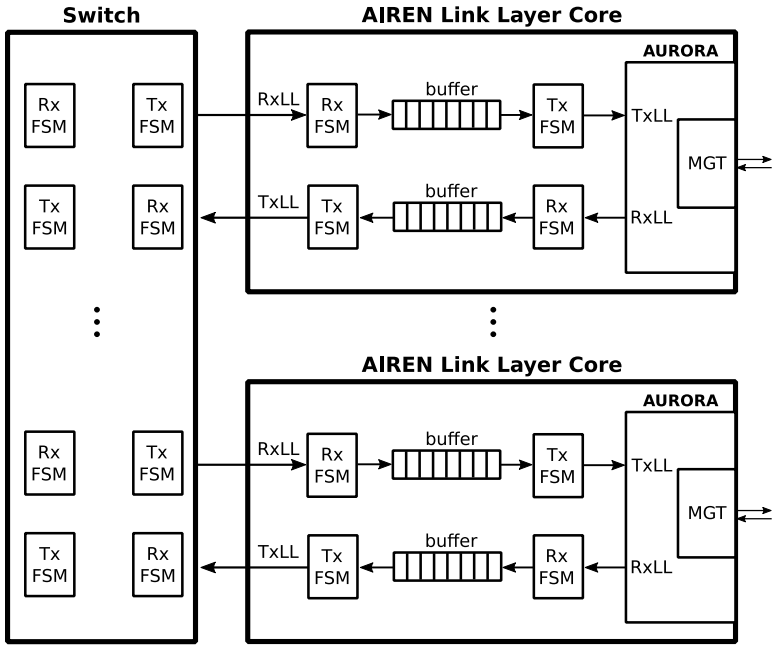

Figure 3.1 AIREN block diagram for a single communication channel on the *Spirit* FPGA-based cluster.

The authors also implements a configurable (software addressable configuration registers) full crossbar switch with broadcast capabilities. Here again, the state machines at each input port control data flow by applying back pressure to pause transfers. Xilinx's LocalLink (LL) interface is used as the standard communication interface.

The *Spirit* cluster uses Aurora to interface eight independent Multi-Gigabit Transceiver (MGT) along with a custom Serial Advanced Technology Attachment (SATA) breakout board, providing up to eight connectivity channels per node at a bit rate of 3.2Gbps. Such a high radix (8 connectivity channels) has the advantage of better network topology exploration, hence, smaller network diameter but at the expenses of higher FPGA resources usage due to the on-chip packet switching. A drawback on this approach is the I/O bounds related to the switch implementation, which limits the bandwidth to 3.2Gbps per channel even with channel bonding.

This approach implements a custom packet-based protocol over Aurora for data transfers between nodes. This allowed them to reduce overhead and utilize 95% of available channel bandwidth with 16KB payload packet size. The inter-nodes latency reported for the AIREN network was 0.8μs.

A quad-FPGA cluster implementation on a Berkeley Emulation Engine (BEE) 3 multi-FPGA prototyping platform is presented in [17]. The author's intention is the study of application-level network processing, for what they implement and accelerate all required protocols in high-speed internet connexion. At the Physical and Link level of the Open Systems Interconnection (OSI) reference model, they used Xilinx's Ten Gigabit Attachment Unit Interface (XAUI) and 10 Gigabit Ethernet (10GbE) Media Access Controller (MAC) modules pair. The system only leverages one high-speed serial port used as the system entry of eight available, bounding the platform aggregate bandwidth to only 10Gbps. Internal PCB parallel lines interconnect the four FPGAs in a ring topology. As expected, the high overhead and complexity of internet protocol implementation resulted in around 45% FPGA resources utilization. Even if network interface latency was not reported, it can be is easily deduced from the Physical and Link layer implementations, that it is above 1μs. Similar internet protocol acceleration approach have been presented in [18] where a full TCP/IP stack was implemented, reporting 5.5μs as the lowest stack latency. Figure 3.2 shows a block diagram example of an internet protocols implementation over 10GbE. The complexity associated with each functional block inevitably leads to latency penalties. In addition, internet protocols implementation does not support direct network, hence, an external switching device is mandatory.

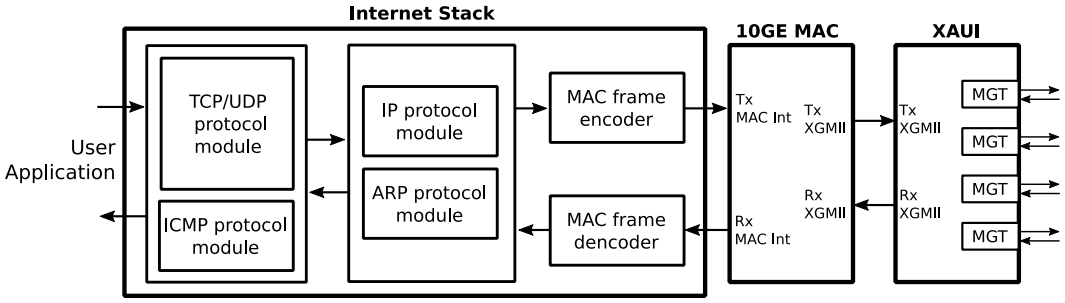

Figure 3.2 Internet protocols stack implementation example over 10 Gigabit Ethernet.

In [19], eight BEE3 platforms are arranged to form a 32-FPGA cluster. This approach proposes a latency optimized hybrid interconnection network composed of high-speed serial inter-BEE3 and parallel PCB intra-BEE3 links. Two protocols are implemented for the serial links: 10 Gigabit Ethernet and Aurora. To extend platform connectivity beyond the eight

CX4 ports, the authors use, for Aurora implementation, CX4-to-SATA breakout cables for breaking apart intrinsic CX4 channel bonding. This allows them to explore different topologies in the cluster. Similar to [17] the 10 Gigabit Ethernet approach instanciates Xilinx's XAUI and 10GbE MAC modules pair. All of the FPGAs in the cluster were connected to an external 10 Gigabit Ethernet switch to achieve full connectivity. The reported channel latency was 840ns and 1.56μs for the network diameter.

The Aurora communication approach is similar to [16] (depicted in Figure 3.1). In this case, the switching module architecture is based on multiplexers which are controlled by a mix of round robin and priority schedulers. They also used port controllers at each individual switch port for determining the packet's destination path (routing capability), requesting access to the switch and controlling data flow. Figure 3.3 depicts the switch block diagram from [19]. The latency reported associated with the switch implementation was 16 clock cycles running at 250Mhz.

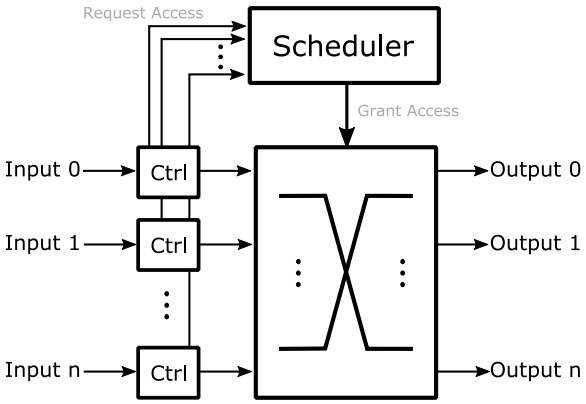

Figure 3.3 Switch module architecture.

Aurora interconnection showed more resources utilization employing many thin channels than fewer, wider channels because of the routing costs. The switch implementation on a high connectivity configuration, adds a significant delay to the packet routing. As a consequence, the system shows almost no improvement in the network diameter (845ns) compared to low connectivity. In terms of Bit Error Ratio (BER), this approach does not implement any error detection/correction mechanism. However, to reduce channel faults, the transceivers are forced to operate at a line rate of 1.95Gbps from the initially set 3.75Gbps. The overall network efficiency peak, the lowest FPGA resources utilization and the channel latency reported are 86%, around 10%, and 541ns respectively.

Bluehive is a custom 64-FPGAs machine assembled at the University of Cambridge [10,20,21]. It is composed of 64 DE4 boards from Terasic, hosting Altera's Stratix IV FPGAs. Each

board has twelve SATA3 channels connected to twelve multi-gigabit transceivers in the FPGA. A PCIe-to-SATA breakout board is used to provide independent access to eight transceivers in the PCIe connector. At the FPGA-level, a custom interconnection Intellectual Property (IP) module (BlueLink) wraps up the transceivers to implement a custom communication protocol. Figure 3.4 shows the BlueLink core layers and its protocol fields. Cyclic Redundancy Check (CRC)32 is used for error detection, which is appended to the packet structure. The authors proposed a reliability layer to ensure reliable communications over the link. When an error is detected, the packet is retransmitted by means of a window-based retransmission buffer and a packet sequence number. The packets are removed from the buffer and the retransmission window is shifted, only after a delivery acknowledgment has been received. A Start of Day field is used for link reinitialization. The Ack field contains an extra bit for back pressure data flow control which tells the sender to stop or start transmission. Reliable links are achieved but at the expenses of less bandwidth efficiency. A major drawback of this approach is that the custom protocol can only send a maximum of 64 bits data payload packed in 128 bits. This represents 50% overhead which combined with the 8b/10b transceiver encoding, reduces available bandwidth to around 40%. The reported effective data rate and latency, for a single FPGA-to-FPGA link at a transceiver operating rate of 3.125Gbps, are 1.2Gbps and 400ns.

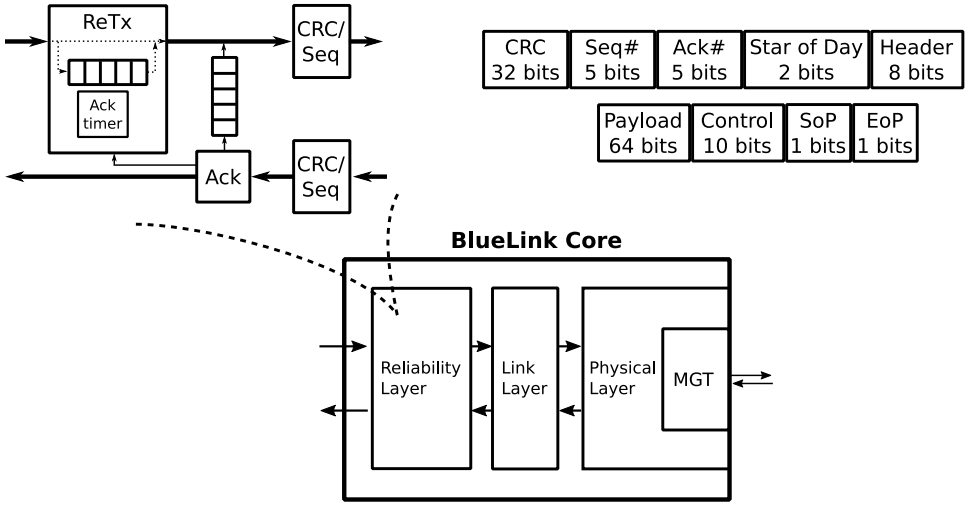

Figure 3.4 BlueLink communication approach.

3.2 FPGA-based network accelerators and heterogeneous machines

Other research works have pointed to heterogeneous computing cluster rather than pure FPGA cluster. Many of these solutions integrate FPGA-based expansion boards to CPU

systems for accelerating communication of nodes. The PCIe connector is generally used to achieve tight integration between the NIC on the boards and the hosting CPUs. This involves complex protocol transformation resulting in a higher NIC footprint and latency than pure cluster solutions. Since the use of the entire FPGA for a NIC implementation is out of scope, we have focused only on the FPGA-to-FPGA communication part for the reviewed literature.

CusComNet in [22] is an FPGA-based interconnection network implementation for heterogeneous computing clusters. Sixteen nodes composed of a CPU system hosting a Virtex 5 LX330T FPGA board each, are interconnected in a 4x4 2D torus topology. Similar to [23], each node has four Infiniband 1x cables connected to four multi-gigabit transceivers, allowing a maximum theoretical bandwidth of 1.6Gbps per link after 8b/10b encoding. At the low communication level, CusComNet wraps the four transceivers individually, creating four independent full duplex channels. However, there are no signs of any channel bonding support for increasing channels aggregate bandwidth. A customizable communication protocol developed by the authors permits them to set, in design time, a variety of parameters for the protocol, such as the number of bits of the source and destination IDs and the payload size to be used. This results in a better utilization of the available bandwidth and the FPGA resources. An up to 97% efficiency per link of 1.6Gbps is reported for 64 bytes payload. Despite that, we estimate up to 91% effective data rate approximately, for the same payload size due to the CRC32 error detection mechanism appended to the protocol which it seems like it wasn't considered. Packets routing is achieved by means of a simple round robin scheduler module, a route module that contains routing information base on the network topology and a cross-bar switch. CusComNet implementation latency (FPGA-to-FPGA) reported was 830ns for 0 bytes data payload packets.

Some research projects like EXTOLL [24–26], APEnet+ [27] and NetFPGA [28] have addressed and commercialized high-speed and low latency FPGA-based communication solutions, as high-performance boards for clusters. While these solutions have good network performance, they intended to use almost the entire FPGA fabric letting little room left for user applications. EXTOLL project, for instance, introduces an IP core for low latency message exchange in an interconnection network. The architecture is divided into three main layers: at the top layer is the host interface implementing protocol for the hosting machine (PCIe or HyperTransport); the middle layer implements translations and interfaces between the top and the bottom layers; and at the bottom layer a cross-bar switch with a round robin arbiter connects the network interface to six FPGA transceivers running at 6.24Gbps. The authors implement a credit-based flow control mechanism for avoiding reception buffer overflow, a CRC error detection, and a packet retransmission mechanism. A prototype on a Xilinx's Virtex 4 FX100 FPGA achieved a half-round-trip latency of 1.16μs for 8 bytes

payload messages. Reported device utilization shows the high complexity of this approach where 85% of the FPGA resources were utilized even after floorplanning.

Similar to EXTOLL, APEnet+ ([27]) proposed a three-layer custom architecture for an FPGA-based NIC card. A central switch at the middle layer interconnects and controls flow between a PCIe interface for the hosting PC and the Multi-Gigabit Transceivers (MGT) at the Physical layer. Virtual channel technique is used to improve accesses to the physical links and buffering. Twenty-four transceivers on an Altera Stratix IV FPGA are used in a four-lane configuration to provide six QSFP+ links with a maximum theoretical bandwidth of 34Gbps per link. Once again, maintaining the maximum transceivers bit rate is difficult because of the errors on the links, so the authors reduced the rate to 28Gbps. The reported device utilization was 42% but no details about protocol or latency were presented.

In an effort to abstract and simplify FPGA cluster designs, [6] presents a framework for multi-FPGA interconnection using multi-gigabit transceivers. The authors proposed a FIFO-like interface to the NIC that receives a continuous stream of data and subdivide it into packets for transmission. The framework uses CRC for error detection and an acknowledgment mechanism to signal each received packet integrity to the sender. The Acknowledgment mechanism is well-known for limiting exploitation of the available bandwidth even with piggybacking technique. A better solution for a framework would be the negative acknowledgment for signaling thus the occurrence of an error. Packet sequence identifier is also proposed to detect missing and not in order packets. A retransmission buffer at the sender side provides error correction features. Figure 3.4 depicts the interconnection framework block diagram usign Multi-Gigabit Transceivers (MGTs).

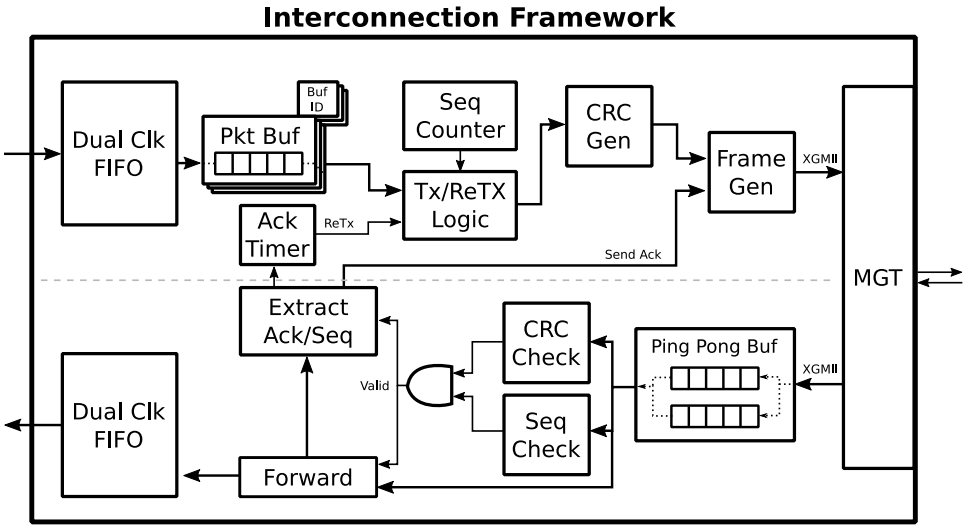

Figure 3.5 Interconnection framework block diagram for multi-FPGA cluster [6].

The packet structure defined for the interconnection framework has 256 bits protocol overhead and fixed (defined in design time) payload length. Packets are assembled in 64-bits chunks to match XGMII interface at the transceivers user side. The framework evaluation was done on two Xilinx ML605 development boards carrying two Virtex-6 LX240T FPGAs. The device utilization reported was around 6%. No other details of implementation can be presented for this approach because of the lack of clarity in the report. The framework refers to neither packet routing nor transceivers grouping for channel bonding support.

NetFPGA [28] is an open source FPGA-based platform designed for research and teaching of high-speed interconnection networks. NetFPGA SUME is PCI Express board with I/O capabilities for 10 and 100 Gbps operation. At its core, a Xilinx's Virtex-7 FPGA empowers the platform and provides one of the last generation transceivers in the market: the 7 Series GTH (13.1 Gbps). To achieve 100Gbps aggregate bandwidth and increase the board connectivity to the network, the platform supports expansion through an FPGA Mezzanine Card (FMC). A total of 14 high-speed serial links among SPF+ and FMC interfaces are available per board. As part of the open source project, various reference projects can be accessible for implementation on the board. All of the reviewed network reference projects use Xilinx's XAUI - 10GE MAC IPs at the physical and link layers. As we have seen before Ethernet's full stack performance is not too attractive for FPGA-to-FPGA communication.

Aurora is a link-layer protocol of the OSI model [29]. In [30] the authors present a custom network and transport layer implementation over Aurora, aiming high-layer functionalities for FPGA interconnection networks with low latency and lightweight characteristics. The network layer proposed implements a router with fixed packet routing path to achieve lossless and in order features. Endpoint modules (virtual channel) are connected to the router to allow information exchange with them-self throughout the network. The transport layer is implemented at each individual endpoint, to support individual credit-based end-to-end flow control. For prototyping, the authors used twenty Xilinx Virtex-7 VC707 FPGA development boards on an Intel Xeon-based heterogeneous machine. An FMC pinned out eight GTX transceivers configured as two-per-lane to provide four 20Gbps serial links per node. The reported Aurora per-hop latency was 0.48μs deduced from 75 clock cycles at 156.25MHz. Packet and flow control overhead yielded 85% channel efficiency, which means 17Gbps effective throughput. The device utilization for entire network stack using two Endpoint modules was less than 4%.

3.3 Recent trends in FPGA interconnect technology

In the race for high-performance serial I/O, FPGA vendors are reaching impressive achievement in the transceivers technology field. In 2010 Xilinx announced its 7 Series FPGAs family introducing a new high-k metal gate (HKMG) variant of 28nm process technology [31]. The scaling process from the previous 45nm family resulted in an increase in cell density and system performance whilst reducing power consumption. High-speed serial connectivity also benefited great improvement. The 7 series introduced up to 96 built-in multi-gigabit transceivers blocks capable of operating at 13.1Gbps maximum rate, and 16 others transceivers blocks operating at up to 28.05Gbps. Devices I/O bandwidth raised to 2.8Tbps.

In 2013 Xilinx introduced the UltraScale series manufactured in a 20nm planar process and the UltraScale+ families based on 16nm FinFET+ 3D transistors technology. Once again transceivers were boosted, now to support 100G Ethernet, 150G Interlaken, and the next to come PCIe Gen4 protocols [32]. Transceivers density went up to 128 per FPGA at a maximum operating speed of 32.75Gbps for 8384Gbs total I/O bandwidth. Table 3.1 summarizes the latest trends in transceivers technology from Xilinx's FPGAs.

Table 3.1 UltraScale and ultraScale+ architecture transceiver portfolio [2]

Family	Type	Max Performance	Max Transceivers	I/O Peak Bandwidth
Virtex US+	GTY	32.75Gbps	128	8384Gbps
Kintex US+	GTH/GTY	16.3/32.75Gbps	44/32	3530Gbps
Zynq US+	PS-GTR/GTH/GTY	6.0/16.3/32.75Gbps	4/44/28	3316Gbps
Virtex US	GTH/GTY	16.3/30.5Gbps	60/60	5616Gbps
Kintex US	GTH	16.3Gbps	64	2,086Gbps

Protocols scalability has been related to transceivers performance and flexibilities. Nowadays FPGA vendors offer massive bandwidth standard IP solutions for a high range of networking applications. Table 3.2 summarizes some of the serial IP protocols supported by Xilinx's latest FPGAs.

Xilinx's 7 series architecture introduced integrated hard IP for PCIe solutions. The UltraScale family has extended this hardened approach to 100G Ethernet and 150G Interlaken protocols. The number of hard IPs available in each FPGA varies from one family to the other, but it does not exceed one tenth. This still represents a great improvement on the FPGAs interconnection technology especially in terms of the latency because now the high complexity and overhead of these standard protocols profit from specialized fabric.

Table 3.2 Xilinx's latest trends in high-speed serial solution [3].

Protocol	LogiCORE IP solution	Lane Rate(Gbps)	Configuration
10G Ethernet	XAUI/RXAUI/	3.125/6.25/	4 Lanes/2 Lanes/
	10G Ethernet PCS-PMA	10.3125	1 Lane
40G Ethernet	XLAUI[a]/	10.3125	4 Lanes
	40G Ethernet PCS-PMA[a]		
100G Ethernet	CAUI-10[a]/100G Ethernet	10.3125/	10 Lanes/
	PCS-PMA[a]/CAUI-4[b]	25.78125	4 Lanes
Interlaken	Interlaken 150G	3.125-12.5/	1-12 Lanes/
		12.5-25.78125	1-6 Lanes
Aurora	Aurora 8b/10b	up to 6.6/	1-16 Lanes/
	Aurora 64B/66B	up to 25.7813	1-16 Lane
PCI Express	Gen2/Gen3	5/8	x1/x2/x4/x8

a. 40G/100G Ethernet LogiCORE IP.
b. UltraScale Devices Integrated Block for 100G Ethernet LogiCORE IP.

In the first trimester of 2016, Xilinx announced the first 56Gbps transceiver using the 4-level Pulse Amplitude Modulation (PAM4) transmission scheme, setting thus the foundations for the 400GE standardization to come. Intel (Altera) FPGA materialized 56Gbps PAM4 transceivers with the release of the Stratix 10 family [33]. Stratix 10 FPGAs have been built on the Intel 14nm Tri-Gate process offering a powerful platform for networking applications. Hard IP solutions for PCIe and 100G Ethernet are supported too in the Stratix 10, as well as a broad range of soft standards IP solutions for high-speed serial protocols.

Both main FPGA vendors offer high integration and flexibility in their latest devices. FPGA designers have now a broad range of possibilities for FPGA interconnection to choose from, i.e. soft or integrated hard IPs and custom or standard protocols. The hard IP solution for some standard protocols available in the newest FPGA architectures offers high potentialities in the communication field, by managing protocol complexity with specialized fabric, and reducing communication footprint. The high cost of this technology and the no yet free access to products information makes performance studies still difficult for the research community.

3.4 Matrix-vector multiplication for scientific applications

The Basic Linear Algebra Subprograms (BLAS) [34] describes low-level routines that provide standard building blocks for solving algebra problems and equations. It is widely used as the benchmarks' core in supercomputers. BLAS has three different categories according to the complexity of the algorithm. BLAS level 1 comprises vector-level math, i.e. dot prod-

uct. Level 2 and level 3 increases complexity to matrix-vector and matrix-matrix operations respectively. The BLAS libraries serve as basic blocks for many numerical linear algebra applications. Any BLAS level implementation benefits from the wide variety of published implementations to compare with, giving a close idea of the computing power performance of any targeted machine. Moreover, a BLAS implementation performance may be extrapolated to higher level scientific and engineering application performance estimations.

Matrix-vector multiplication (BLAS level 2) is an ideal low-level application for FPGA-based computing platforms. Because of its highly parallel nature, the computations may be done in parallel by distributing multipliers and adders all across the FPGA machine. Many authors have addressed matrix-vector multiplication seeking acceleration of scientific applications. Sparse matrices are more popular for architecture researches than dense matrices since the matrix traversal, the indirect memory accesses and the matrix compression methods are more challenging. However, most of the sparse matrix-vector multiplication implementations on FPGAs include a dense matrix test that generally up bounds the performance estimation. Thus, our literature review focuses on matrix-vector multiplication implementations on FPGAs with dense matrices performance results.

In [7, 35], a Basic Linear Algebra Subroutines (BLAS) FPGA implementation using double-precision floating point is presented. This approach is a row-major Gaxpy design that uses various $PIPE$ operators for parallel computing dot-product of a row of the matrix A with the vector X. $PIPE$ operator performs several multiplications in parallel and accumulates them throughout an adder tree structure and a dual stage accumulator. The operator is stall-free, being capable of accepting new data sets every clock cycle. Figure 3.6 shows the authors approach for matrix-vector multiplication. It includes multiple $PIPEs$ in parallel, each operating on a different row of the matrix. The vector X output data is broadcasted to all the operators. The partial sums has been reduced in the accumulator module, the corresponding Y vector element is updated. The arithmetic operators are double-precision floating point Xilinx IP cores. The arrangement in parallel of $PIPE$ operators allows inter-row and intra-row parallelism. The design is implemented on a Virtex-5 lx155t and a Virtex-6 lx240t FPGAs with 16 and 32 processing element respectively. The authors reported 3.1GFLOPS peak and 0.92GFLOPS sustained for the Virtex-5, as well as 6.4GFLOPS peak and 1.14GFLOPS sustained for the Virtex-6 platforms.

Similar to the above no-stall multiply-accumulator implementation, in [8, 36] the authors propose the $R3$ processing element architecture with an intermediator module between a multiplier and an adder core to compose a single processing element. To address a stall-free operator despite the incoming data flow rate, their approach works on multiple intermediate

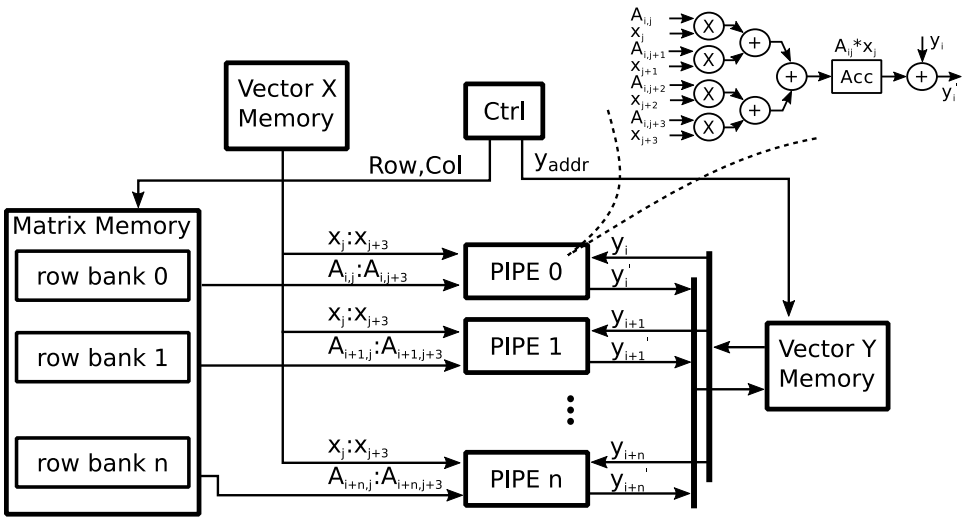

Figure 3.6 Row-major BLAS gaxpy architecture [7].

Y vector values and does the additions out of order. The Intermediator module allows the matrix to be traversed in a loosely row-major traversal in a matrix block set of 16 rows, by storing 32 intermediate Y vector values. It takes in two values, one from the multiplier's result and one from the adder's result and outputs a pair of values to be added. Internally is has a dual-port Block RAM module that stores intermediate values until an element in the same row appears. Figure 3.7 depicts the a single matrix-vector processing element and its distribution on a multi-FPGA platform.

The authors managed to fit 64 processing elements in a Convey HC-1 platform with four Virtex-5 lx330 FPGAs and 40GB/s of sustainable global memory bandwidth. They achieved 13.6GFLOPS peak and 7.6GFLOPS average for a dense 2000×2000 matrix. The more recent work of the same authors shows an implementation on a Convey HC-2ex platform hosting 64 processing elements on four Virtex-6 lx760 FPGAs with 19GB/s memory bandwidth per device. This new improved system reach up to 16.3GFLOPS peak processing power and 11.9GFLOPS average. Both distributed solutions don't address node communication.

Authors from [37] present a single-precision floating point processing elements architecture for matrix-vector multiplication based on a dual-port RAM, a multiplier and an adder. The targeting device is a Virtex-5 sx95t FPGA with 35.74GB/s of memory bandwidth hosted by a Reconfigurable Open Architecture Computing Hardware (ROACH) platform. For a 2000×2000 dense matrix and 64 processing elements, this approach reaches a computational performance of 19.16GFLOPS peak and 17.64GFLOPS average.

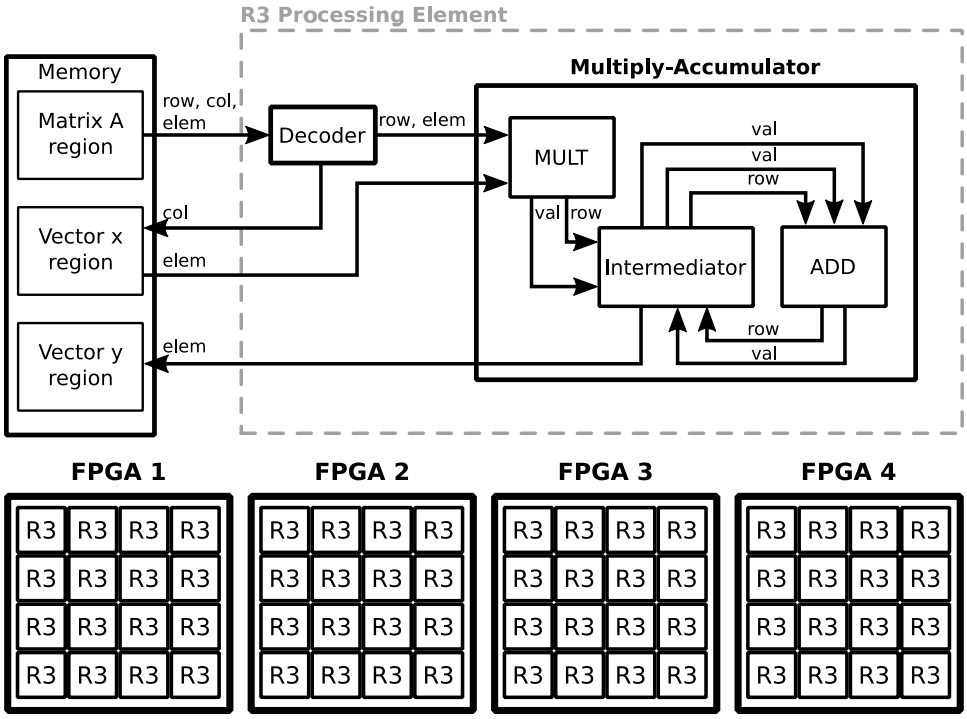

Figure 3.7 Parallel matrix-vector multiplication with $R3$ processing elements [8].

3.5 Conclusions

Computational units and interconnect network are the two typical main components in today's High-Performance Computing (HPC) systems. In order to increase performance by parallelization, large parallel systems with many nodes are built. Nonetheless, HPC is compromised, among others, by the interconnect network. The interconnect network should meet requirements like low latency, high-bandwidth, fault tolerance, and scalability to allow high-performance communication between the computational units.

From the reviewed articles, it is possible to observe that the utilization of complex features on high-speed serial protocols yields reliable communication channels. However, there is latency penalty and resource over-utilization cost to pay. We have seen that often the authors relied on protocols, originally designed for different applications, which results in an unnecessary overhead and a reduced effective throughput.

Ethernet is the most popular solution since its broad utilization on internet networks. The full Ethernet stack implementation has demonstrated this solution is not effective for time-critical applications. [10] has shown for FPGA-to-FPGA communication, custom solutions are preferable over standards off-the-shelf approaches. Nevertheless, a custom approach does not preclude standard IP cores wherever they are useful, i.e they may be components in a

communication stack.

In terms of FPGA connectivity for a fixed physical bandwidth, we have seen that a high fan-out leads to better direct-network topologies (2D mesh, torus, etc.) and reduced network diameter. But as a consequence, a high-radix router implementation is generally necessary thus causing high resource consumption and latency. A trade-off between the fan-out and the channel bonding must be evaluated in order to achieve the best performance.

Most popular error detection mechanisms found in the literature are Checksum, CRC, and parity. Most popular error correction mechanism is the Automatic Repeat Request (ARQ) with its different variants like stop-and-wait ARQ, Go-Back-N, and Selective Repeat ARQ [6, 20–22, 25]. Data acknowledgment (Ack) or negative acknowledgment (NaK) is very used too. It is generally combined with a sliding window technique for the retransmission buffer implementation and/or an acknowledgment timer for timeout signaling when packet loss occurs. Flow control mechanism is used to allocate the different resources in the network such as the buffers and the channels as the data progresses from the source to its destination. Most popular implementations of flow control regarding FPGA interconnection networks utilizes the credit-based or the back pressure (ON/OFF) techniques [10, 14, 26].

For concluding this chapter, the consulted researches are summarized hereafter. Table 3.3 summarizes the main results and features of the reviewed implementations for high-performance interconnections on FPGAs. We should highlight that all of the custom protocols showed better end-to-end latency results than standard implementations.

Table 3.3 Summary on high-performance communication implementations using FPGAs.

Implementation	Fanout (Channels)	Bandwidth per Channel	Protocol	Latency	Efficiency peak	Resources	Platform
[15]	8 one-lane	3.2Gbps	Custom over Aurora	800ns	95% [a]		Virtex-4
[17]	1 four-lanes	10Gbps	Internet stack 10GE (XAUI+MAC)	>1μs		45%	Virtex-5
[19]	2 four-lanes	10Gbps	10GE	840ns		18%	Virtex-5
[19]	2 four-lanes	6.24Gbps	Aurora	541ns	86%	10%	Virtex-5
[10, 21]	12 one-lanes	3.125Gbps	Custom	400ns	40%	-	Stratix IV
[10, 21]	12 one-lanes	10Gbps	Custom	200ns	40%	18%	Stratix V
[22]	4 one-lane	1.6Gbps	Custom	830ns	91%	40%	Virtex-5
[30]	4 two-lanes	17Gbps	Custom over Aurora	480ns	85%	4%	Virtex-7

a. with 16KB payload.

Table 3.4 organizes the reviewed implementation results of matrix-vector multiplication (MVM) on FPGAs, GPUs and CPUs platforms.

Table 3.4 Summary on computational performance on different MVM implementations with dense matrices results.

Work	Peak Perf. [GFLOPS]	Total PE	Platform	Mem. BW [GB/s]
[35]	3.1	16	single FPGA Virtex-5 lx155t (BEE3)	6.4
[7]	6.4	32	single FPGA Virtex-6 lx240t (ML605)	51.2
[37] [a]	19.2	64	single FPGA Virtex-5 sx95t (ROACH)	35.74
[8]	13.6	64	four FPGAs Virtex-5 lx330 (Convey HC-1)	80
[36]	16.3	64	four FPGAs Virtex-6 lx760 (Convey HC-2ex)	80
[8]	23	N/A	GPU Tesla M2090	177.6
[37] [a]	9.3	N/A	GPU GTX 660	144.2
[37] [a]	21.7	N/A	GPU GTX Titan	288.4
[37] [a]	2.5	N/A	Intel Core i7-2600	21
[37] [a]	5.2	N/A	Intel Core i7-4770	25.6

a. single-precision floating point.

CHAPTER 4 THE COMMUNICATION IP

Time critical applications on multi-FPGA platforms and the lack of performance of the commercial off-the-shelf communication protocols have motivated us to propose a custom implementation of a communication IP optimized for latency. For its design, we have carefully selected the necessary functions to be implemented as part of the protocol to successfully achieve low latency and high throughput links. This chapter presents the communication IP, its architecture, main blocks, functional features and implementation results on an FPGA platform. The Open Systems Interconnection (OSI) scheme is used to aid presenting our architecture.

4.1 Communication requirements

Given that the main goal for this work is a high-speed and low latency inter-FPGA communication, the architecture requirements for the communication IP is following presented. These requirements have driven the development time and will be part of the evaluation and comparison criteria.

1. *High-Speed*

   The IP should exploit the maximum bit rate capacity from its physical components.

2. *Low latency*

   The IP logic and its functionalities should be chosen carefully and optimized for latency. The aim is to achieve competitive half-round trip time.

3. *Efficient*

   The protocol should have reduced overhead to maximize the effective throughput.

4. *Small footprint*

   The communication IP should consume as few resources as possible to provide enough room for user application.

5. *Abstraction*

   The IP should simplify communication among other FPGA devices in an interconnection network.

6. *Reliability*

   The network interface must be aware of bit errors during the communication.

4.2 Architecture

The communication IP maps into the Physical, Data Link and Network layers of the OSI reference model because of its functionalities. A simplified description of the IP architecture is a protocol logic driving a Serializer/Deserializer (SerDes) module (Figure 4.1).

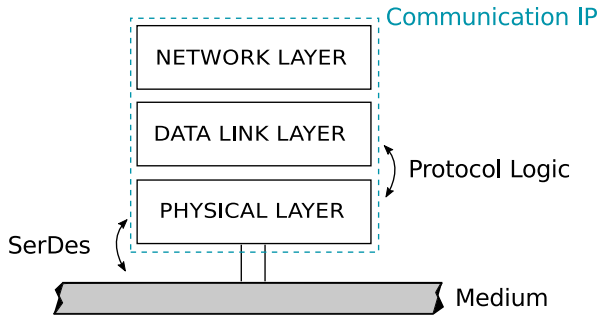

Figure 4.1 The communication IP layers based on the OSI model.

Figure 4.2 depicts the communication IP block diagram. From a top-level view, the inter-FPGA communication is mainly composed of Sockets, switches, state machines, and XAUIs. The IP provides multiples user application (endpoint) accesses to the PHY by means of virtual channels. The user endpoints connect to the communication IP through the Sockets which can be of the access type needed: Transmission or Reception Socket. A FIFO-like interface offers a friendly and common interface for the hardware modules at each end of the communication channel. A custom packet-based protocol with routing and broadcast support is also offered. The communication IP abstracts applications from the under-hood complexity. The main modules and features are discussed in more detail hereafter.

4.2.1 Physical layer

The Physical layer contains dedicated hardware to transmit raw bits over a physical link. The Multi-Gigabit Transceivers (MGTs) compose the core of the Physical layer in the communication IP. A XAUI is used to bond four transceivers in a single wide channel running at 10Gbps. The need for a XAUI stems from the fact that channel bonding does perform better than single-lane channels (refer to Chapter 3). In addition, having Ethernet compatibility at the Physical layer could be useful for future implementations. A custom approach to a high-speed, low latency communication channel does not preclude the utilization of commercial off-the-shelf IPs whenever they have interesting properties [10]. In that situation, we found

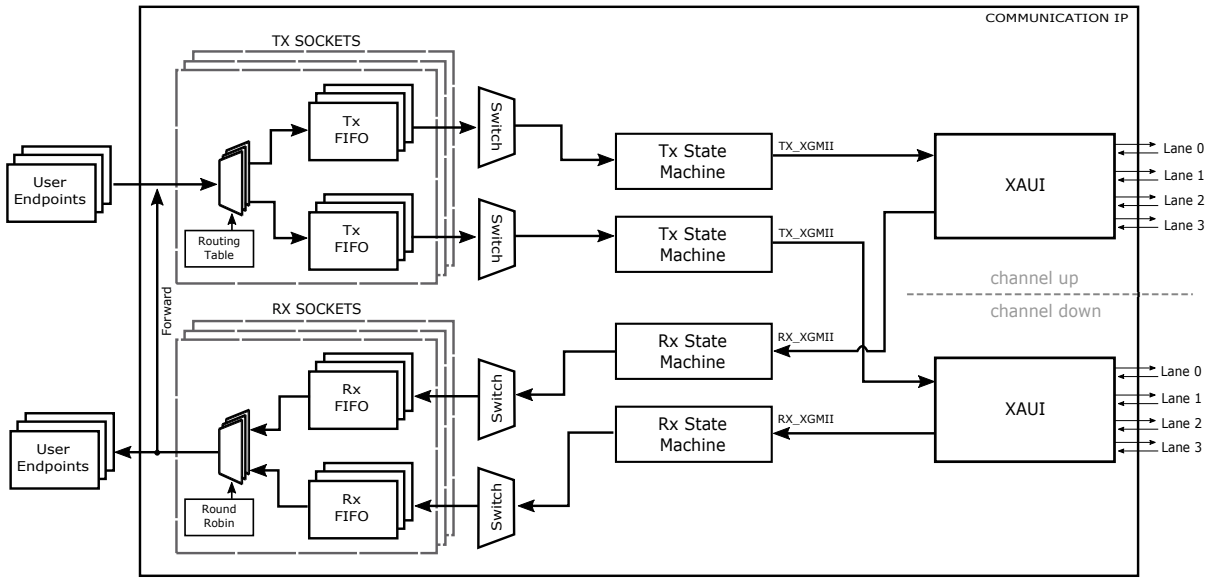

Figure 4.2 The communication IP block diagram.

Xilinx's XAUI solution [38] which is used in our architecture.

A four-lanes channel implementation reduces the communication IP connectivity to a fanout of one rather than a fanout of four when using, for example, four single-lane channels. This cancels up all variety of network topology implementations that do not involve an external high-speed switch (indirect network). To overcome this problem, the communication IP implements two coupled XAUI cores to provide two wide links towards the network side, doubling thus the fanout and the bandwidth. At the Physical layer boundary, two independent XGMII interfaces allow accesses to their corresponding XAUI channel.

Instantiating two XAUI cores is not a simple drag-and-drop process. Some modifications have to be done in the source codes especially because of the clocking requirements. A single XAUI core comes with a Digital Clock Manager (DCM) module. Its source clock input ($CLKIN$) is fed from the dedicated DCM at a transceiver Tile. XAUI's DCM purpose is to supply the other clock signals (reference, transmission and reception clocks) required by the transceivers and the user logic. Keeping both DCMs, one per XAUI, when instantiating two XAUIs in a design would lead to two clock domains of the same frequency and eventually, they have to be handled like that. To avoid this, the source codes are modified to provide a single, external and shared DCM for both XAUIs and the user logic. Figure 4.3 shows the clock distribution, used in our approach, for two coupled XAUIs in a Virtex-5 FPGA. Four GTP Dual Tiles are used consecutively to provide eight transceivers toward the network side.

Only one reference clock signal ($REFCLKOUT$) coming out from the transceivers is used for the rest of the design. The DCLK port can be ignored since no dynamic reconfiguration is required.

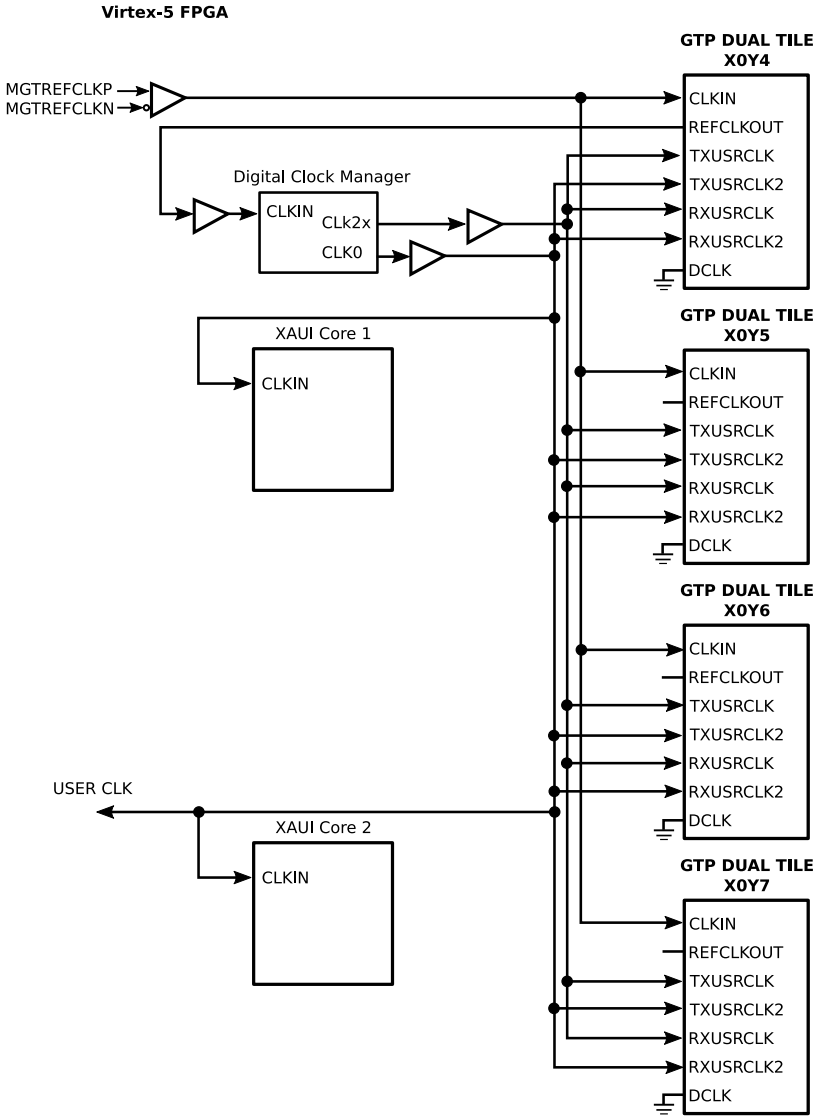

Figure 4.3 Two coupled XAUIs clocking scheme.

The XAUI core is not intended to work at a different clock frequency than the nominal 156.125MHz, which is required for 10Gbps operation. Xilinx Core Generator tool does not support any other clock frequency for XAUI. However, by modifying the source code manually and configuring properly the shared XAUI DCM and the transceivers' dedicated DCMs, it is possible to achieve different operation frequencies. This technique will be lately used in our approach.

The error detection mechanism in the communication IP is based on the 8b/10b encoding scheme being located here at the Physical layer. The decoder engine at the transceiver reception side tracks the incoming data to detect errors. When the data arrives with wrong disparity or illegal 10-bit codes (out-of-table) the decoder immediately signals an error. The out-of-synchronization error in any lanes is also signaled. The XAUI core monitors the transceivers error signals and generates its own flags which can be accessed from the status and control ports. The integration of these mechanisms provides thus a significant reliability level on the communication channel.

4.2.2 Link and network layers

In order to increase the abstraction level of the communication channels, a Link and Network layer are designed. In a direct network implementation, every node is an endpoint and a switch. Distributing switches as part of the nodes allows full connectivity without the need of a centralized switch. However, the use of direct network increases the number of hops required to reach a destination and the overall network latency and diameter. Hence, one of the major tasks of this work is to minimize the Link and Network layer latencies.

Custom protocol

The Link layer implements a packet-based protocol to append information to raw data bytes (Figure 4.4 XGMII stream). A packet consists of six fields: Start character, Source ID, Destination ID, Control, Payload, and Tail.

— *Start and tail*
The Start character (0xfb) and the Tail are imposed fields by the XGMII protocol. They are used by the XAUI core as Control Characters to align, synchronize, and manage the links. The Tail is a concatenation of an End character (0xfd) and four Idle characters (0x07070707). They are used to ensure the minimal necessary interframe gap.

— *Source and destination IDs*
The source and destination IDs are 16-bits wide fields each. They represent endpoints and nodes addresses in the network. The lower byte corresponds to an endpoint ID and the higher byte to a node ID (an FPGA ID). This allows a total addressing range of 256 FPGAs and 65536 endpoints.

— *Control*
The control field is currently reserved for future utilization. It intends to hold information in future feature implementations like automated retransmission on error detection and flow control.

— *Payload*

The Link layer supports variable payload length without restrictions on the maximum length. This is an interesting feature to maximize channels efficiency.

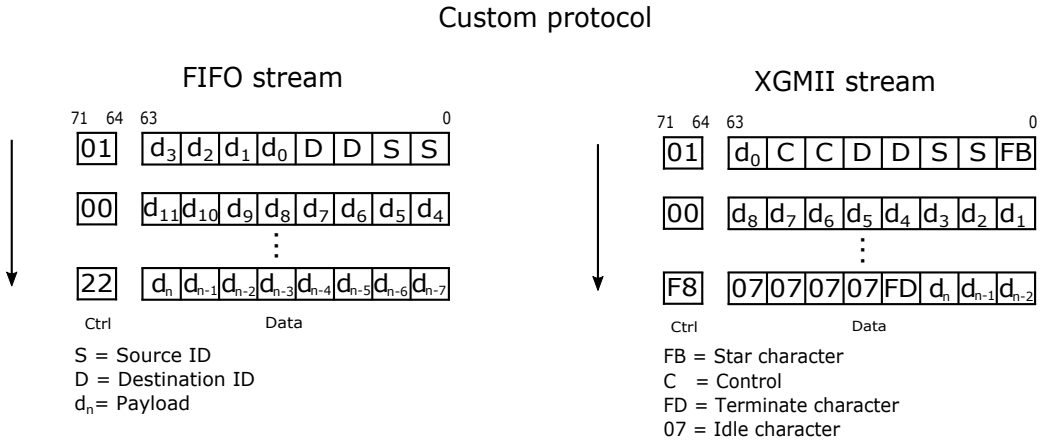

Figure 4.4 A custom protocol example at the communication IP user side (FIFO stream) and at the XAUIs user side (XGMII stream).

From the user point of view (endpoint), a FIFO-like interface with a simple custom protocol abstracts the communication process (Figure 4.4 FIFO stream). An endpoint that wants to send a packet has to specify the source and destination IDs along with the payload and the associated control byte. The FIFO-like interface is a 72-bit wide bus grouped in 64 bits ($<63:0>$) for data and 8 bits ($<71:64>$) for control. The control byte is used to signals the Start, Body, and End of the packet being transmitted. The Start of packet is the first FIFO beat of the stream. It is signaled by the bit $<64>$ on the control bus and it indicates that source and destination IDs are available at the first half of the data bus. The Body is a set of FIFO beats holding just payload information. Bits $<65:64>$ should be driven to '0' to signal the Body. As mentioned above, there is not a restriction on the payload length for a packet, hence, the Body size is not restricted either. When the End of a packet is asserted (bit $<65>$), the data bus may hold not all of the bytes with payload information, so bits $<69:66>$ indicate how many of them do hold it. Figure 4.4 shows the custom protocol used at the communication IP user side (FIFO stream) and at the XAUIs user side (XGMII stream).

Transmission and reception sockets

Virtual channel is a well-known technique to increase accesses to a physical channel. For that purpose, the communication IP implements an independent parameterizable number

of transmission (Tx) and reception (Rx) Sockets. A Tx Socket, depicted in Figure 4.5, is basically composed of two FIFOs in parallel and a router. Its input is demultiplexed to the FIFOs depending on the packet's destination ID and the routing decision stored in a routing table. The table must be filled up beforehand, otherwise, the packet will be routed through the default channel. The routing feature is generally part of the Network layer on the OSI reference model. This is the only feature from this layer which is implemented in the communication IP.

The Endpoints normally push/pop data to/from the Sockets in the form of 72-bit beats. The FIFO-like interface uses full (txready) and empty (rxready) signals to indicate when it is possible to send or retrieve data from the network. On transmission, the routing decision is held until a new Start of packet is pushed, so the Body and the Tail follow the same channel.

In order to save logic resources and to improve performance, the FIFOs exploit all potentialities that Virtex-5 technology offers. The FIFO built-in primitive is used enabling the dedicated hard-controller for FIFO implementations in every BRAM block. In addition, the best memory aspect ratio is also used (512×72 bits) to provide a full utilization of the BRAM resources (36Kb per block).

The txready signal indicates to Endpoints the availability of both channels for transmitting packets. The main drawback of this solution is that if a channel collapses and the associated transmission FIFO goes full, the whole Socket goes unavailable. In others words, the opposite channel cannot be used as an alternative path. A simple solution could be to multiplex the full signal from each Tx FIFO depending on the state of the routing decision. Like that, the ready signal for the transmission Socket would be associated to the destination Node ID. However, Tx Endpoints would be aware of the channel availability after signaling the destination Node ID.

A one-clock-cycle routing table is designed for each Tx Socket. The routing decision is stored in a 256×1-bits distributed memory which is implemented in Configurable Logic Blocks (CLB). An appropriated HDL coding style was necessary to properly guide the synthesis tool in inferring distributed memories [39].

The 72-bits FIFO-stream protocol signal (Figure 4.5) is split to separate control and destination ID fields. The left-most byte (most significant) of the packet's destination ID, indicating the destined FPGA, feeds the read address port of the routing memory and consequently, the one-bit routing decision can be read. The state of the routing decision controls the multiplexing of the push signal, driving thus the packet upstream or downstream. The route path must be held for the following beats. For that, a register stores the routing decision and updates itself at every Start of packet.

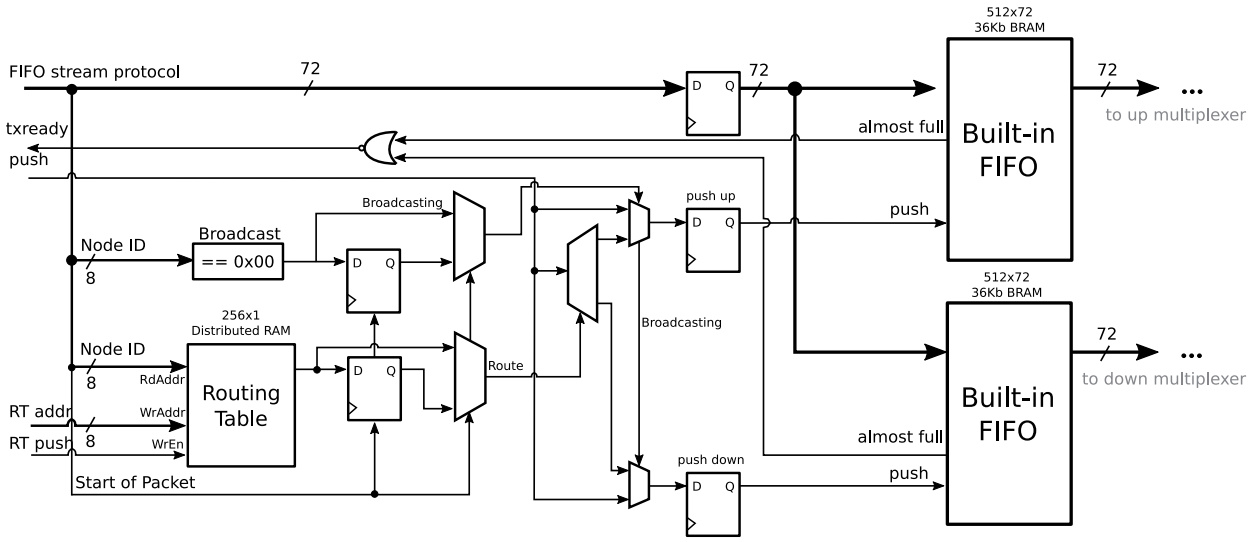

Figure 4.5 Transmission socket architecture.

The main advantages of using a distributed memory for the routing table implementation are a fast looking-up process (asynchronic reads) and the BRAM resource saving. To alleviate the critical path of the combinatorial logic, especially in large designs with several Sockets, a pipeline stage is used, from there the one-clock-cycle latency.

There is one router per Tx Socket. For the moment, all of them share the same routing information because their write port is connected together to a single write interface, *RT addr* and *RT push*. However, creating separated write interfaces for each router is straightforward and enables independent routing information for each Socket. This is a great feature to balance transmission loads over the network.

Unlike the Tx Socket, the Rx Socket does not need a routing table. Instead, it uses a 16-bit wide identification number that obeys the following rules:

— The most significant byte represents the node ID (FPGA ID).
— The most significant byte must to be the same for all of the Rx Socket in an FPGA and unique in the network.
— The least significant byte represents the endpoint ID (virtual channel ID).
— The least significant byte has to be unique in an FPGA.

These rules are adopted to simplify the router implementation and save FPGA resources. The Rx Socket IDs are used to match with the packet destination field on reception and aiding to reach the packet's destination.

Default sockets and forward feature A default Tx and Rx Socket is used to support a for-

warding feature. These Sockets are internal and not accessible from the user application (endpoint). Figure 4.6 shows the architecture for a default Tx Socket (upper half) and a default Rx Socket (lower half) interconnected for packet forward support. When a packet is received and its destination does not match with any local Rx Socket IDs, it is transferred to the default Rx Socket. Once it is on either of the reception FIFOs, a round robin scheduler monitors the state of their empty signals and stalls when there is data available. The FIFOs used are First-Word-Fall-Through type where the pop signal works as a read acknowledgment. The packet data beats are multiplexed from both default Rx FIFOs to the routing table and finally to either of the default Tx FIFOs depending on the routing decision. This means, for example, that a packet arriving from the down channel can be looped back through the same down channel or forwarded to the opposite up channel. There is a risk of routing loop in closed network topologies like ring, thus special attention must be taken to the routing table. A solution could be using the available control field in the packet structure to implement a time-to-live mechanism, similar to IP protocol.

Switches To multiplex the virtual channels (Sockets), it is necessary a switch implementation with a scheduling politic. In transmission, the round robin scheduling politic is chosen to provide equal chances in accessing to the physical channels. Two $(1+number\ of\ txsocket)-to-1$ multiplexers are used, one per PHY. As can be seen in Figure 4.7, the channel multiplexer interconnects the Tx Sockets (including the default one) to the Tx state machines. From the Tx state machine point of view, the multiplexers and demultiplexers abstract the fact that many Tx Sockets are using the same physical channel. The round robin scheduler selects the correct signals for communication and stalls whenever there is a packet for sending. The Figure also shows a packet header storage mechanism per Tx Socket. This is useful for the packet partitioning and composition feature supported by the Tx state machines. Channel multiplexing in transmission has from 0 to NB_TX_Socket clock cycles latency because of the round robin scheduling.

In reception, two $1-to-(1+number\ of\ rxsocket)$ demultiplexers use the packet's destination ID for comparing to the Rx Socket IDs. If a match is found the packet is pushed to its destination Socket, otherwise it is forwarded to the next node through the Default Sockets. The channel demultiplexing process is fully combinational. Figure 4.8 shows the channel demultiplexing architecture.

Broadcast

Broadcast is a useful feature supported by the communication IP. In many applications, the nodes need to send information to all other nodes in the network. Without a dedicated broadcasting mechanism, the current design has to send the same information separately.

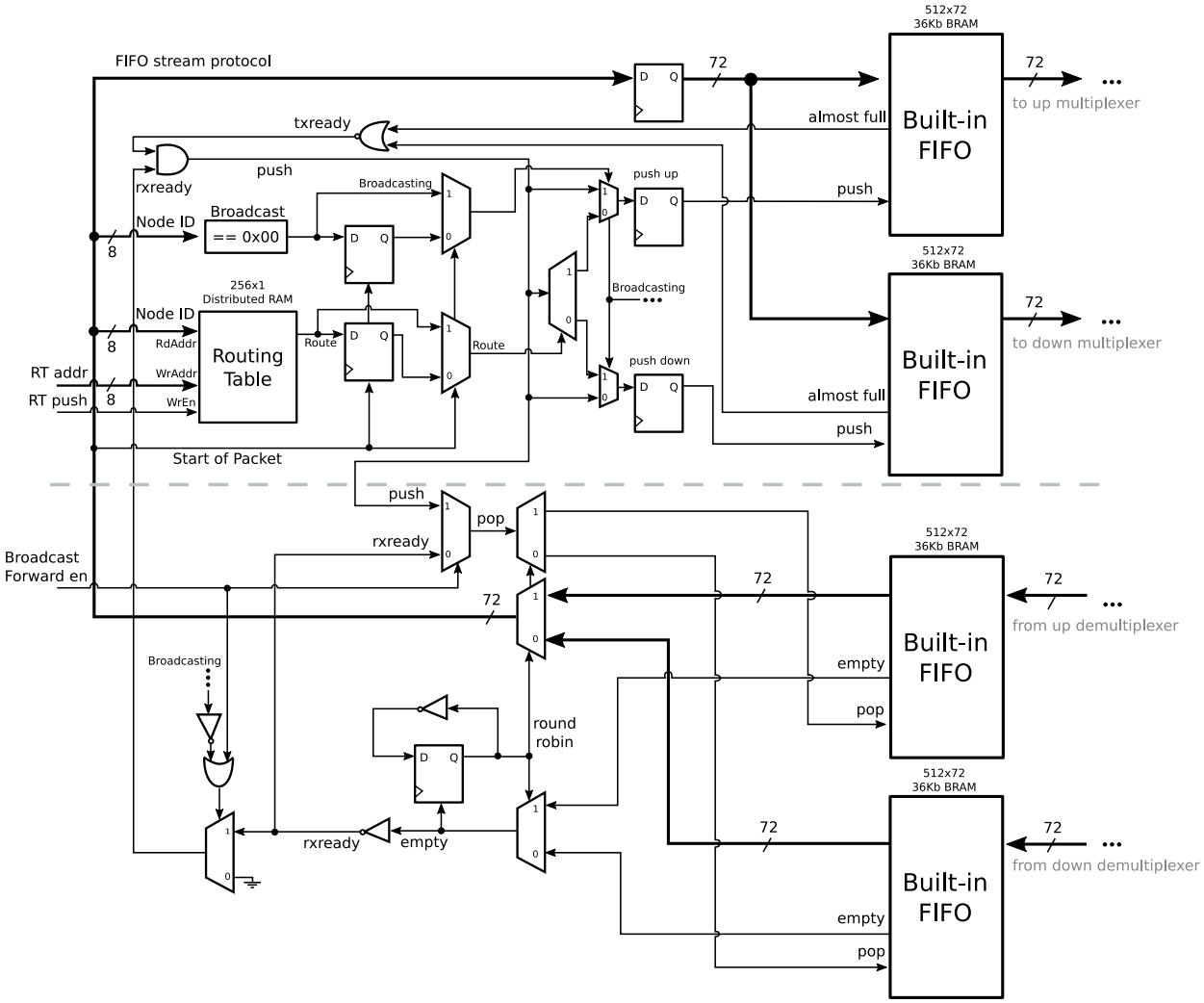

Figure 4.6 Default Tx and Rx sockets architecture interconnected for packet forwarding.

This overhead can be reduced with the implemented broadcast protocol. In transmission, whenever a packet has destination ID equal to zero, it is forwarded to both communication channels, channel up and down. In reception, whenever the detected destination ID is zero, the packet is pushed to all of the Rx Sockets, including the Default Rx Socket. Therefore the broadcasted packet is received in all of the local Endpoints and is forwarded to the next node (FPGA). To control the broadcast propagation over the network, the communication IP implements two control interfaces, one per channel (*Accept Broadcast* signal in Figure 4.8). The control interface is a 2-bits in port which manages the broadcast configuration. Among the configuration states are accept, reject, forward and not forward broadcast packets.

Figure 4.9 shows an example of a broadcast packet across the network. A Tx Endpoint at

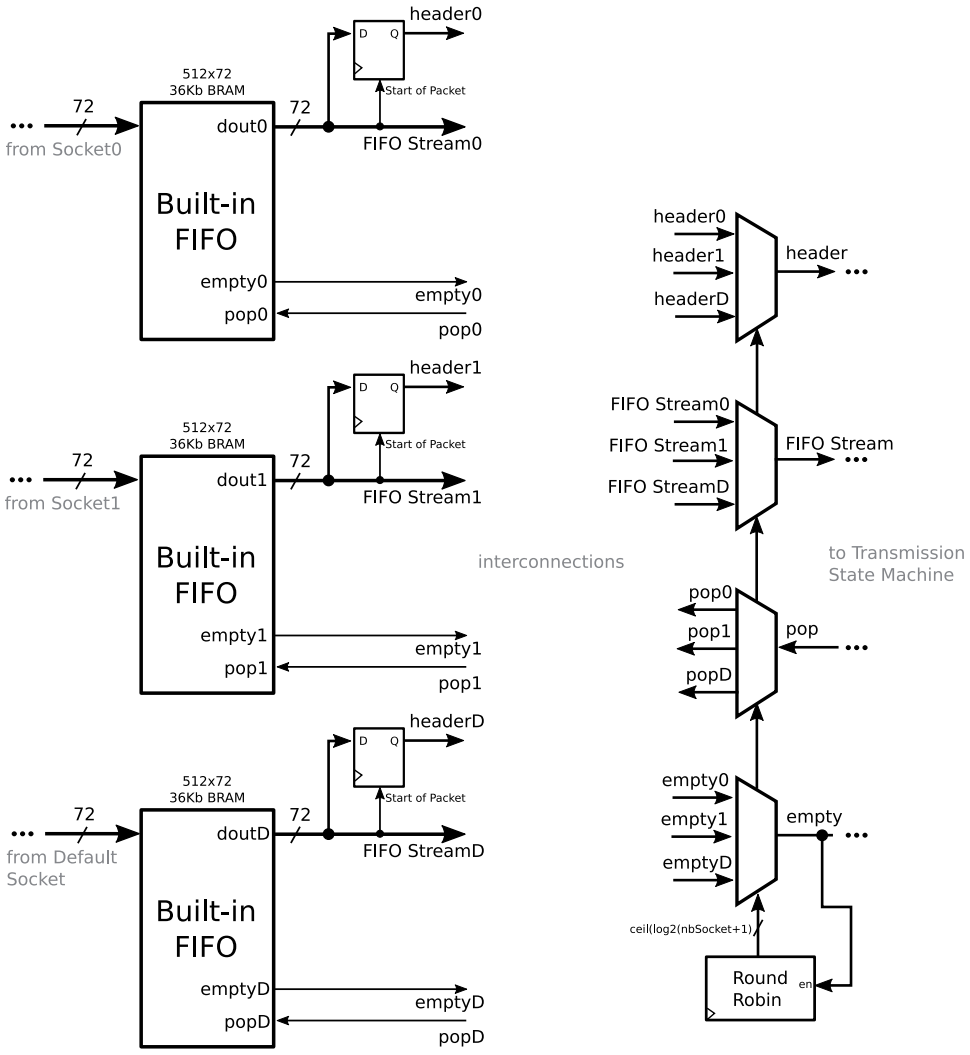

Figure 4.7 Transmission sockets multiplexing. Virtual channelling.

the node 1 injects a packet with the destination address $0x0000$ (broadcast). The packet is sent through both channels to the next upstream and downstream neighboring nodes 4 and 2. Nodes 2 and 4 are configured to accept and forward broadcast packets from upstream and downstream respectively. In node 3, the packet has to be accepted only one time and not forwarded at all. For that, the communication IP is configured properly to block the broadcast packets incoming from channel UP and to accept and not forward those from channel down. This simple and effective design allows reducing data propagation time over the network. The configurations may be modified dynamically.

Transmission and reception state machines

The transmission and reception state machines are at the boundary of the Link layer, inter-

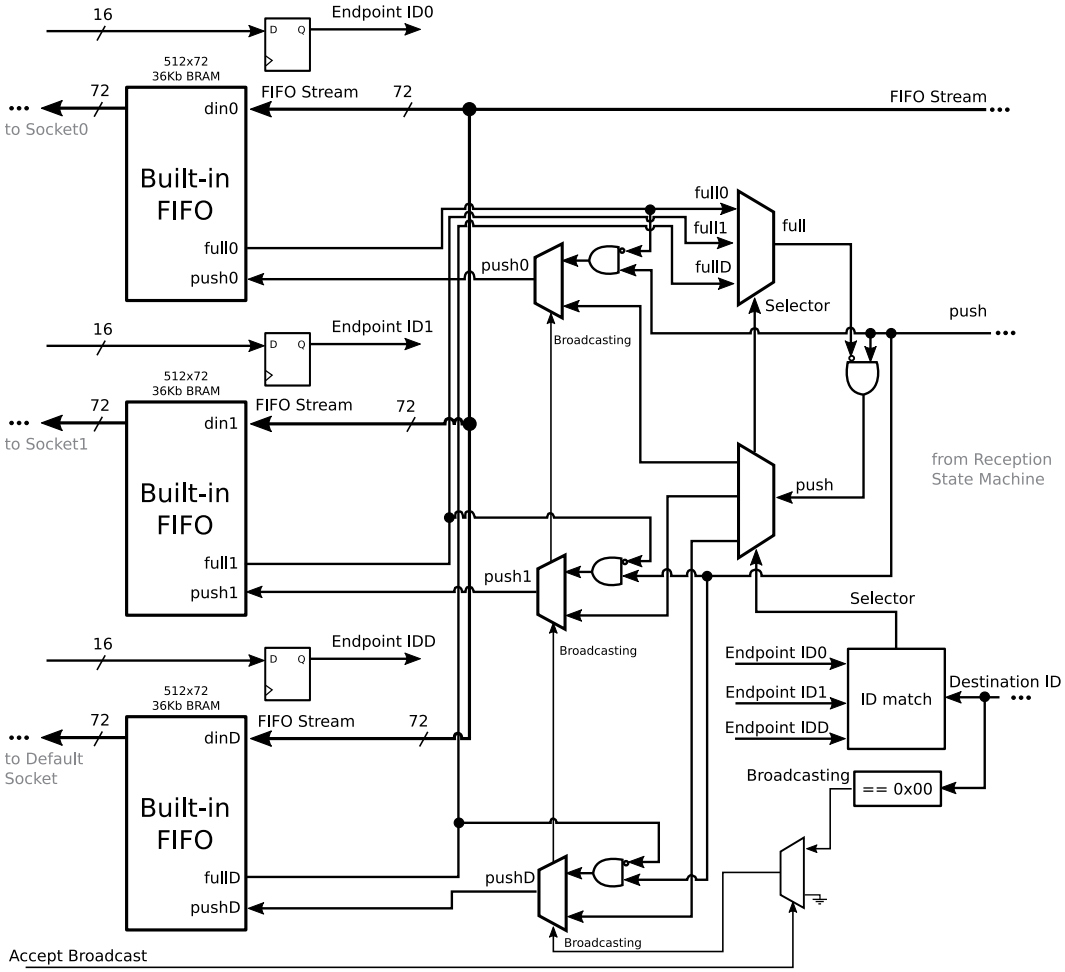

Figure 4.8 Reception sockets demultiplexing. Virtual channelling.

facing the Physical layer. They transform a FIFO-like stream of words from/to the Sockets into a continuous stream of words (XGMII) for the XAUIs. The Tx state machine keeps a track of the source and destination IDs of the last packet transmitted for each Tx Socket. This is used to support interrupted transmissions in case an endpoint cannot maintain the cadence in a long packet. In such a situation, the packet is split and the header information is stored. It will be used subsequently whenever the endpoint restarts transmission. The Rx state machine monitors the XGMII signals for incoming packets. After the destination ID is identified from the header, the Rx state machine transforms the incoming data stream to the FIFO-stream needed at the Rx Sockets. Out-of-format packets are ignored as well as the incoming data afterward an error is signaled by the XAUIs.

Flow control is not supported by the state machines. However, an easy estimation could prevent applications to saturate the network and therefore lose packets. Suppose a network

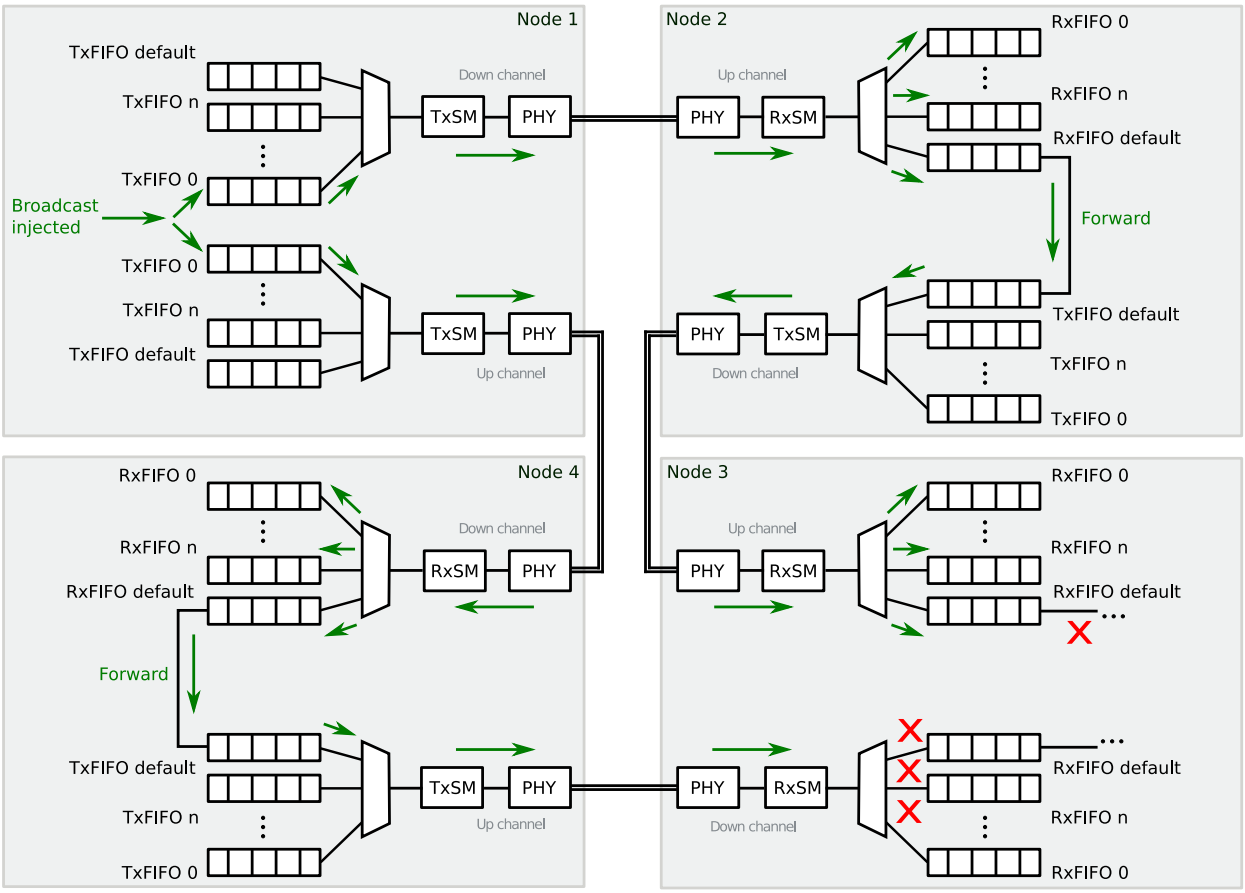

Figure 4.9 Broadcast packet propagating across the network.

of 8 nodes connected in a ring topology like is shown in Figure 4.10. All of the nodes are sending data to a single common node A, including itself, and the routers are configured to balance the network load and to route the packets through the shortest path. Considering the node A has 10Gbps bandwidth per channel, the total data injection rate distributed equally among the nodes is 2.5Gbps. If 8-bytes payload length is used per packet, the channel efficiency would be 33.3%. Thus, the maximum effective data transfer rate (throughput) per node before saturating the network is 833Mbps. Flow control is no longer necessary whenever the application running on the cluster does not exceed the saturation threshold.

An application that requires more effective data transfer rate per node than an estimated threshold eventually will stall, even if a flow control system were implemented, bounding the system for communication rather than for processing power or for memory bandwidth which is generally the cases.

User Endpoint modules are responsible for keeping the buffers in a not full state otherwise,

they risk of packet loss.

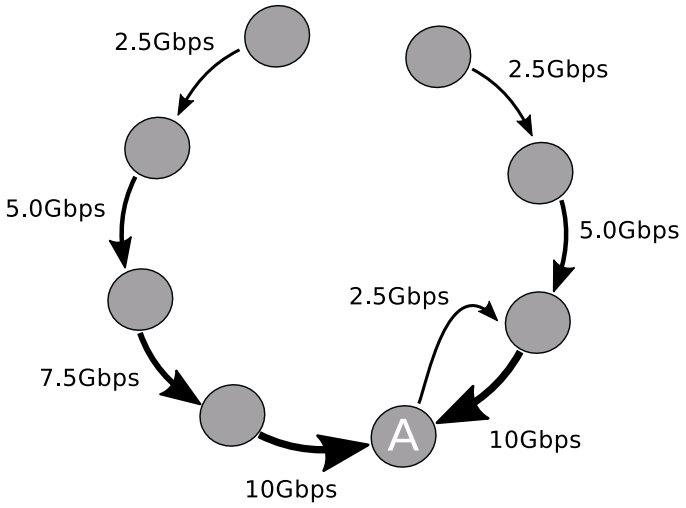

Figure 4.10 Effective data rate estimation example to avoid flow control.

4.3 Software interface

Software interfaces are necessary to support microprocessor transfers through the network. Our approach utilizes a MicroBlaze soft processor and a PLB bus for peripherals interconnection. Each Socket at the communication IP may connect to the PLB bus through a transmission (Tx) interface module and a reception (Rx) interface module depending on the Socket's type (Figure 4.11).

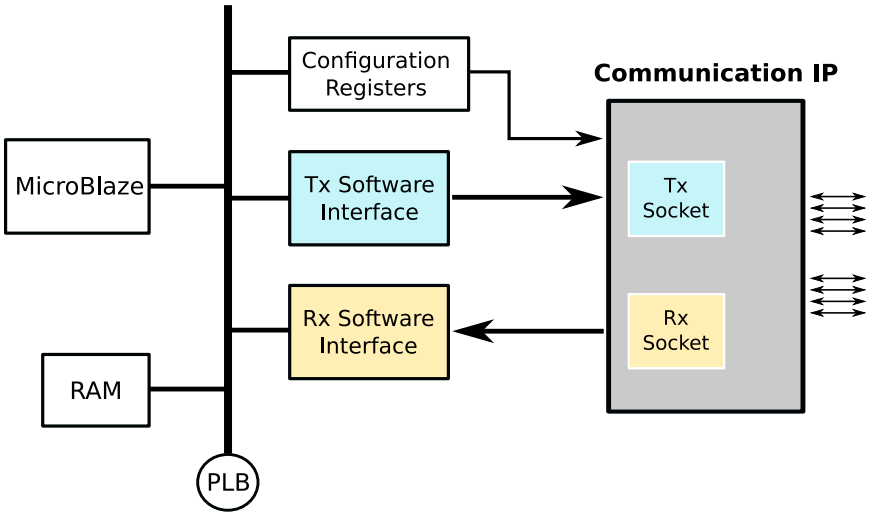

Figure 4.11 Software interface for the communication IP.

4.3.1 Transmission software interface module

The Tx software interface module uses three address spaces to provide accesses to three register banks. Each register bank holds a portion of the 72-bit width interface of the communication IP user side. To inject a packet on the network, the microprocessor or any other coprocessor needs to execute three write operations since PLB bus width is only 32 bits. Writing to the register bank 3, which is associated with the Control bits in the FIFO-Stream protocol, makes the module to push the data into the Tx Socket.

The MicroBlaze and the PLB bus are not intended to work at the communication IP clock frequency. They rather work at lower frequencies, making thus necessary a synchronizer for handling clock domains. Figure 4.12 shows the architecture of the software interface module for Tx Sockets. The *write enable* signal for the register bank 3 (WE_Add2) is synchronized with 2 flip flops for resolving cross clock domain issues. Two stabilizer registers keep the synchronized signal steady for at least 2 clock cycles. Since it is possible that the synchronized signal stays asserted for more than 1 clock cycle, depending on how metastability is being resolved, a rising edge level detector is used for avoiding pushing the same data twice. A masking register is also used to hide data changes from the communication IP. The transmission ready signal needs to be synchronized too. Since we do not expect that the communication IP goes full too frequently, we can consider this signal as a very low-frequency signal. Hence, only 2 flip-flops are necessary.

4.3.2 Reception software interface module

Our Rx software interface is designed to provide Direct Memory Accesses (DMA) through a PLB master interface and store the received packet in any memory region in an organized manner. Memory organization is based on the Source ID of each incoming packet. A memory segment is assigned to each Source ID, which in turn it is associated with an Endpoint in the network. A Firs-In-First-Out hardware support is also provided to the software system. In other words, the Rx software interface is a peripheral device that handles many FIFO controllers operating on memory segments and allows DMA write operations for packet storage and organization.

Since supporting a FIFO controller with a specific memory region for all 65535 possible Endpoints (Source ID $< 15 : 0 >$) is too costly in terms of FPGA resource consumption, our approach uses a parameterizable design where the number of Endpoint actually used can have a FIFO controller support. The architecture of the Rx software interface module, presented in Figure 4.13, utilizes a generic parameter to configure itself for as many Source

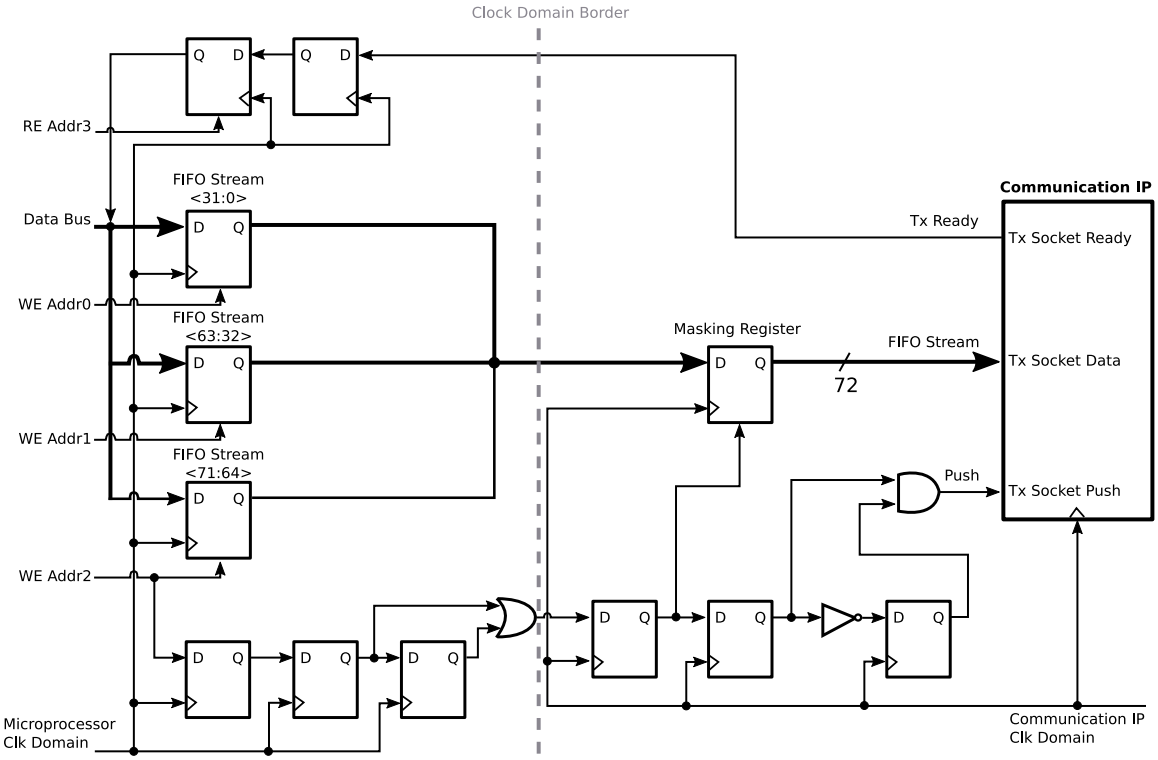

Figure 4.12 Transmission software interface module.

Endpoints as needed (currently for 32 Endpoints).

Unlike the Tx software interface module, the Rx counterpart uses a higher bandwidth synchronizer for crossing clock domains; a dual clock FIFO. After entering on the microprocessor clock domain, the Source ID field from the 72-bit FIFO-stream signal is matched to a set of Source Endpoint IDs stored in a look-up table. When a match is found, the corresponding position in the table is extracted and used as the address selector for a set of RAM memory blocks. The memory blocks hold the FIFO controller and the memory segment information for each Endpoint. The look-up table is composed of register banks providing fast access and ID match. It must be populated beforehand by the software system with the identification number of every Tx Endpoint on the network.

The memory set is composed of true dual-port Block RAMs (BRAM). One port gives access to the microprocessor system and the other is used by the module itself. For each segment allocated on a memory peripheral, the memory set holds a byte count for the stored data, a read/write pointer for packet's payload data accessing, the segment's size and base address, and the number of occupied bytes in the current 4-bytes writing location (Figure 4.14). All these information are arranged in groups (Memory Segment Data Set) to properly minimize the BRAM utilization by considering BRAM aspect ratio.

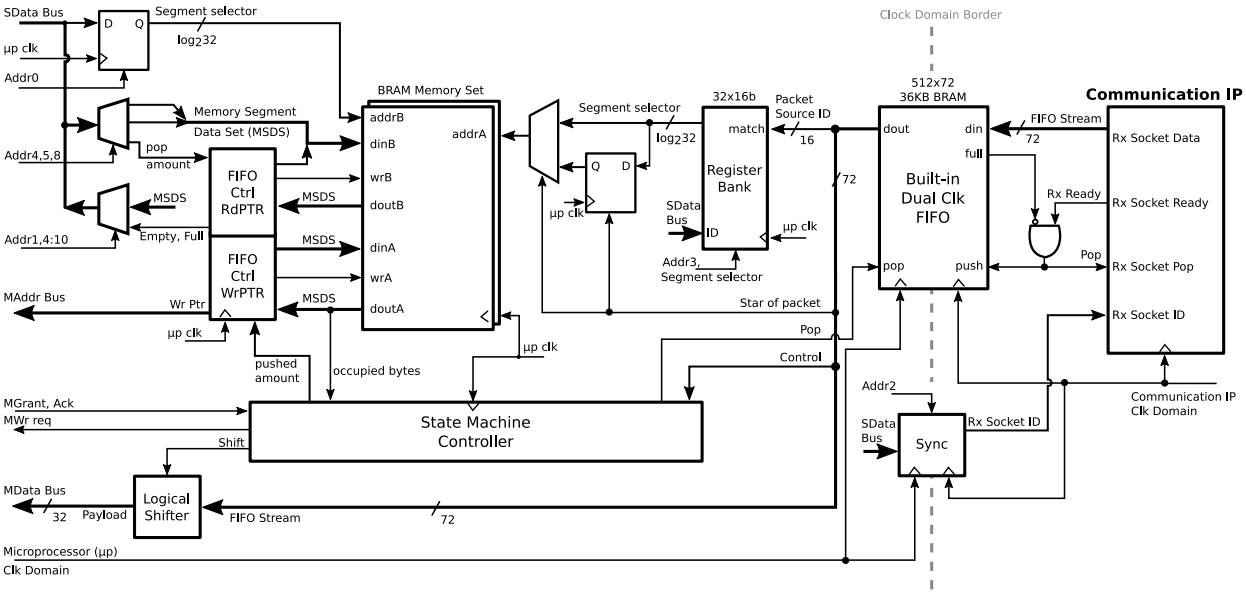

Figure 4.13 Reception software interface module.

The microprocessor should allocate memory segments for each Endpoint on the network, and the associated parameters like the memory base address and the maximum buffer size should be specified to the Rx software interface module. Our approach defines a default memory segment for those packets that do not find a memory segment for its sender.

The module has a state machine controller to drive the PLB master interface when writing into a peripheral memory. Using the segment information and a logical shifter, the controller can organize the incoming payload data consecutively with a byte resolution. Byte resolution is supported, when writing into a peripheral memory, for extending the variable payload feature of the custom communication protocol to the Rx software interface module.

The microprocessor has different access permission to the registers and the Memory Segment Data Set on the Rx software interface module. The register address space is presented hereafter.

There is a FIFO controller module operating segments read and write pointers. Any time a transfer is completed by the state machine in charge of master bus write operations, the amount of byte transferred is notified to FIFO controller which in turns updates the segment's write pointer and full and empty signals. When the microprocessor reads the stored data from the peripheral memory, it has to notify the number of bytes read to the Rx software interface module. The FIFO controller updates once again the segment's pointers and the state. There is a risk of data collision when simultaneous write operations are performed

to the same memory segment. To prevent it, the module monitors write operations and reschedule B port's write, one clock cycle after.

Figure 4.14 shows an example of a memory segment allocated for an Endpoint in the network. All of the received packets which its Source ID matches to the segment ID will be stored consecutively in the associated memory location. In the example, the segment has 6 bytes stored which is informed to the software system by reading the appropriated register (Rx segment data count). The write pointer shifts when a location is all full. The occupied byte information serves to this purpose. When the write pointer reaches the last memory location (base address plus segment size), it loops to the star of the memory segment only if the read pointer is not placed on the base address, otherwise, the FIFO controller signals full. The read pointer shifts depending on the notified amount of bytes being read by the software.

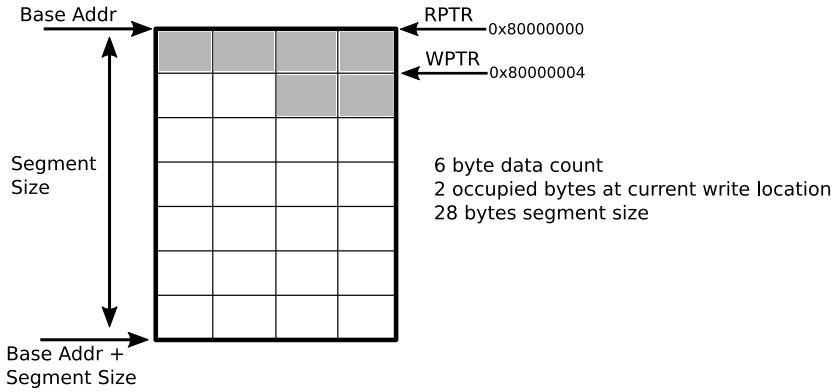

Figure 4.14 An example of a memory segment state controlled by the Rx software interface module.

Table 4.1 summarizes the register address space for both software interface modules. The physical address for each register can be computed as $C\_BASEADDR + REG\_OFFSET \times 0x4$. The register types depend on the permission granted to the software system for a safe utilization of the software interface modules.

4.3.3 IP configuration and utilization by software examples

Setting the communication IP and the Rx software interface module ready for operation requires some configuration steps to be executed before a packet can travel through the network. This can be done by software by accessing to the Configuration Registers interface module (Figure 4.11) and the Memory Segment Data Set at the Rx software interface module.

Configuration of the reception software interface module

Listing 4.1 shows the source code written C for a MicroBlaze microprocessor configuring the

Table 4.1 Register address space for the transmission and reception software interface modules.

Register Offset	Rx Software Interface Description	Type	Tx Software Interface Description	Type
0	Memory segment selector	W/R	FIFO stream LSW Data	W/R
1	Rx Socket status	R	FIFO stream MSW Data	W/R
2	Rx Socket ID	W/R	FIFO stream Control	W/R
3	Rx segment ID	W/R	Tx Socket status	R
4	Rx segment base address	W/R	-	-
5	Rx segment size	W/R	-	-
6	Rx segment wptr	R	-	-
7	Rx segment rptr	R	-	-
8	Rx segment data count	W/R	-	-
9	Maximum number of segment	R	-	-
10	Maximum segment size	R	-	-

reception software interface module through the configuration registers. Initially, a unique ID must be provided for the reception Socket. The identification number follows the rules previously mentioned in Section 4.2.2 *Transmission and reception Sockets*. It should be later used in the destination field of a packet targeting this Endpoint.

The second step is allocating the memory segments for each transmission Endpoint in the network. The segment size should not exceed the maximum size allowed by the Rx software interface module. If a larger size is needed, then simply modify the hardware module by changing the MAX_BUFFER_SIZE generic parameter in the vhdl code. The look-up table (register bank) is following populated with the memory segment IDs which should match the transmission Endpoint IDs for data organization in memory. The register bank depth is equal to the number of nodes (FPGAs) in the network multiplied by the number of transmission Endpoints per FPGA, plus the default segment location. The population of the default memory segment is not shown in the listing.

Listing 4.1 Example code in C for configuring the Rx software interface module.

```
1 // pointers declaration
2 Xuint32 * rx_socket_updown = XPAR_RXSOCKET_INTERFACE_0_BASEADDR;
3
4 // configuring rx socket id. this id is used for both channels
5 *(rx_socket_updown + socket_id) = ((this_fpga)<<8)|endpoint_id ;
6
7 // setting memory segment parameters for each endpoint on the network.
8 s = sizeof(Xuint8) * segment_size;
9 if (s > max_seg_size){
```

```
10   xil_printf("segment size incorrect. %d bytes max allowed.
11         current is %d bytes\r\n", max_seg_buffer_size, s);
12   while (1);  }
13
14 // source id has the following format this_fpga<<8|endpoint_id
15 for (i=0; i< nb_nodes; i++){
16   xil_printf("configuring memory segment %d\r\n", i);
17   for (r=0; r< nb_endpoint; r++){
18   *(rx_socket_updown + socket_seg_selector) = (nb_endpoint*i)+r;
19   *(rx_socket_updown + socket_seg_source_id)  = ((i+1)<<8)|(r);
20   buff_updown = malloc(s);
21   if (buff_updown == NULL){
22     xil_printf("buffer allocation problem\r\n");
23     while(1);}
24   else
25     xil_printf("buffer allocation success \r\n");
26     *(rx_socket_updown + segment_mem_base) = buff_updown; // seg base addr
27     *(rx_socket_updown + segment_mem_size) = s;  // seg size
28     }
29   }
```

Configuration of the communication IP

Listing 4.2 shows an example of the routing table configuration for the communication IP working as the Network Interface Controller in a ring topology interconnection. For accessing the configuration ports in the IP, it is used the Configuration Registers module (Figure 4.11), which basically performs clock domain transitions. The example code uses two for loops to go upstream and downstream across the ring, ensuring all the time the shortest possible path base on the number of hops. Initially, the routing starts with the current node (`this_fpga`) which is set to be routed upstream. This means that if a node sends itself a packet, the route used will be up. However, when the packet arrives at the upstream neighbor, it will be routed back downstream. Writing to the routing table needs the following format `<route_decision><destination_fpga_id><endpoint_id>` where the Endpoint and the destination IDs are 8 bits width each and the route decision is only 1-bit width. Since we use hop-per-hop routing, the Endpoint ID is not considered. Only the destination node (FPGA) ID is of interest for the routing table configuration. Hence, changing the number of Endpoints on the network does not affect it. At the low level, the destination ID is used as the address selector in the 256×1 distributed ram memory where the routing decision will be stored.

A similar technique is used for configuring the broadcast and the forward modes. Depending

on the current node position (FPGA executing the C code) and the broadcast sender position on the network, it would be the configuration. Two for loops go across the network node IDs, from the broadcast sender ID to the opposite node in the ring. The broadcast and forward configuration will depend on where the current node is (downstream or upstream).

Listing 4.2 Example code in C for configuring the communication IP.

```
1  // pointers declaration
2  Xuint32 * ctrl_regs = XPAR_SYSTEM_CONF_REGS_0_BASEADDR;
3
4  // configuring routing table and the broadcast mode.
5  // routing table format route_decision|destination_fpga_id<<8|endpoint_id
6  xil_printf("THIS FPGA ROUTING TABLE -->\r\n");
7  // routing upstream. ring topology. IDs decrease
8  for (i=0; i< nb_nodes/2; i++){
9    if((int)(this_fpga - i) >= 1){
10     *(ctrl_regs + route_table_data_offset)  = (Xuint32) ROUTE_UP|(((int)(
       this_fpga - i))<<8)|(0); // set dest IDs and route decision.
11     xil_printf("\t\t DESTINATION %3d  %3x   ROUTE UP  \r\n", (int)(
       this_fpga - i), (Xuint32) ROUTE_UP|(((int)(this_fpga - i))<<8)|(0));}
12   else{
13     *(ctrl_regs + route_table_data_offset)  = (Xuint32) ROUTE_UP|(((
       nb_nodes) + (int)(this_fpga - i))<<8)|(0);
14     xil_printf("\t\t DESTINATION %3d  %3x   ROUTE UP  \r\n", (nb_nodes) +
       (int)(this_fpga - i), (Xuint32) ROUTE_UP|(((nb_nodes) + (int)(this_fpga
        - i))<<8)|(0));}
15   }
16 // routing downstream. ring topology. IDs increase
17 for (i=1; i<= nb_nodes/2; i++){
18   if((int)(this_fpga + i) <= nb_nodes){
19     *(ctrl_regs + route_table_data_offset)  = (Xuint32) ROUTE_DOWN|(((int)
       (this_fpga + i))<<8)|(0);
20     xil_printf("\t\t DESTINATION %3d  %3x   ROUTE DOWN  \r\n", (int)(
       this_fpga + i), (Xuint32) ROUTE_DOWN|(((int)(this_fpga + i))<<8)|(0));}
21   else{
22     *(ctrl_regs + route_table_data_offset)  = (Xuint32) ROUTE_DOWN|(((int)
       (this_fpga + i) - (nb_nodes))<<8)|(0);
23     xil_printf("\t\t DESTINATION %3d  %3x   ROUTE DOWN  \r\n", (int)(
       this_fpga + i) - (nb_nodes), (Xuint32) ROUTE_DOWN|(((int)(this_fpga + i
       ) - (nb_nodes))<<8)|(0));}
24   }
```

Sending and receiving data

Once the software interface modules are connected to the PLB bus, the sending and receiving process remain quite simple. The software just needs to reproduce the FIFO stream protocol by using the three available registers `lsw_data`, `msw_data`, and `ctrl`. Before sending, it should check the transmission Socket status to verify space availability for each protocol beats. Listing 4.3 presents a utilization example of the software interface module for sending 8 bytes data. Every time the software writes to the `ctrl` register, the entire FIFO-stream beat is pushed into the transmission Socket at the communication IP. Since the microprocessor's clock frequency is lower than the module's and each write operation over the PLB bus takes several clock cycles, the Tx software interface module cannot maintain one FIFO-stream beat per clock cycle. When that happens, two main features developed enter to play, the routing track memorization and the packet split and reconstruction at the transmission state machine. In the example code, the first protocol beat sets the routing path and the header information for all remaining beats. The transmission state machine detects that the rest of the packet is not following coming and splits and terminate the packet with the corresponding tail. Whenever the second protocol beat is pushed, the header information is appended and a packet is reconstructed with the new payload information. This operation may be repeated as long as the `ENDBIT` is not signaled.

Listing 4.3 Example code in C for injecting 8 bytes data into the network through the Tx software interface.

```
1  Xuint32 * tx_socket = XPAR_TXSOCKET_INTERFACE_0_BASEADDR;
2  Xuint64 payload_data64;
3  Xuint32 NB_BYTE = 4 ; // this is only used at the last fifo stream beat
4  Xuint32 HEADER = ((1<<24)|(0<<16))|((this_fpga<<8)|endpoint_id);
5  // header <node ID><endpoint ID><this node ID><this endpoint ID>
6
7  while(((*(tx_socket + TXStatus))) == 0);          // tx socket status
8  payload_data64.Lower = 326651;                    // any random data
9  payload_data64.Upper = 5511236;                   // any random data
10 *(tx_socket + lsw_data)  = HEADER;                // packet's header
11 *(tx_socket + msw_data)  = payload_data64.Lower;  // lsw data
12 *(tx_socket + ctrl)  = STARTBIT;                  // sending flit
13 while(((*(tx_socket + TXStatus))) == 0);          // tx socket status
14 *(tx_socket + lsw_data)  = payload_data64.Upper;  // msw data
15 *(tx_socket + ctrl)  = NB_BYTE<<2|ENDBIT;         // sending flit
```

The reception software interface module does Direct Memory Access operations as packets are arriving at the Rx Socket. It organizes them in memory segments according to the sender's IDs. The software just needs to gather segments information and read the data

from the associated memory location. To find the right segment selector, the software has to consider the number of transmission endpoints per node on the network. Listing 4.4 shows an example of 8 bytes being received from a Tx Endpoint identified as "0" which belongs to the node "`master_id`". The FIFO controller in the Rx software interface module (Figure 4.13) controls all memory pointer operations, so the software just has to access to the read pointer information and then notify the amount read. The read amount notifications let the module know how many bytes can be removed from the segment by automatically updating the read pointer and the segment data count. For the moment, MicroBlaze endianness issues have to be solved by software.

Listing 4.4 Example code in C for receiving 8 bytes data through the Rx software interface.

```
1  Xuint32 data_swapped0;
2  Xuint32 data_swapped1;
3  Xuint32 *rx_ptr;
4  Xuint32 *rx_socket_updown = XPAR_RXSOCKET_INTERFACE_0_BASEADDR;
5
6  xil_printf("Waiting for data to arrive from the node "master_id"\r\n");
7  // selecting segment information set
8  *(rx_socket_updown + socket_seg_selector) = (master_id-1)*nb_endpoint + 0;
9  while (*(rx_socket_updown + segment_data_count) == 0);
10 xil_printf("segment has %d B\r\n",*(rx_socket_updown + segment_data_count)
       );
11 //reading 4 bytes data
12 rx_ptr  = (Xuint32*) (*(rx_socket_updown + segment_mem_base) +
                *(rx_socket_updown + segment_read_ptr));
13 data_swapped0 = ((*rx_ptr>>24)&0xff) | ((*rx_ptr<<8)&0xff0000) | ((*rx_ptr
       >>8)              &0xff00) | ((*rx_ptr<<24)&0xff000000);
14 *(rx_socket_updown + segment_data_count) =  4;  //notifying read amount
15 //reading 4 bytes data
16 rx_ptr  = (Xuint32*) (*(rx_socket_updown + segment_mem_base) +
                *(rx_socket_updown + segment_read_ptr));
17 data_swapped1 = ((*rx_ptr>>24)&0xff) | ((*rx_ptr<<8)&0xff0000) | ((*rx_ptr
       >>8)              &0xff00) | ((*rx_ptr<<24)&0xff000000);
18 *(rx_socket_updown + segment_data_count) =  4;  //notifying read amount
19 xil_printf("Data received is  %d %d \r\n",data_swapped1, data_swapped0);
```

4.4 Implementation and results

This section describes the communication IP implementation and its results on a multi-FPGA platform.

4.4.1 Platform description

Two Berkeley Emulation Engine 3 (BEE3) platforms are used to test the communication IP. The BEE3 [40] has four Xilinx FPGAs Virtex-5 lx155t, each of which hosts up to 16 GB of DDR2 DRAM and connects to two 10GBASE-CX4 interfaces on the front panel. The eight FPGAs are interconnected in a ring topology using three meters long CX4-to-CX4 passive copper cables (Figure 4.15). The dual 10GBASE-CX4 interfaces per FPGA are driven by eight Multi-Gigabit Transceivers (MGTs) at the corresponding FPGA and they are designed to shoot out data at 10Gbps. There is currently installed two 2GB DDR2 DRAM memories per FPGA in a dual channel configuration (two independent channels).

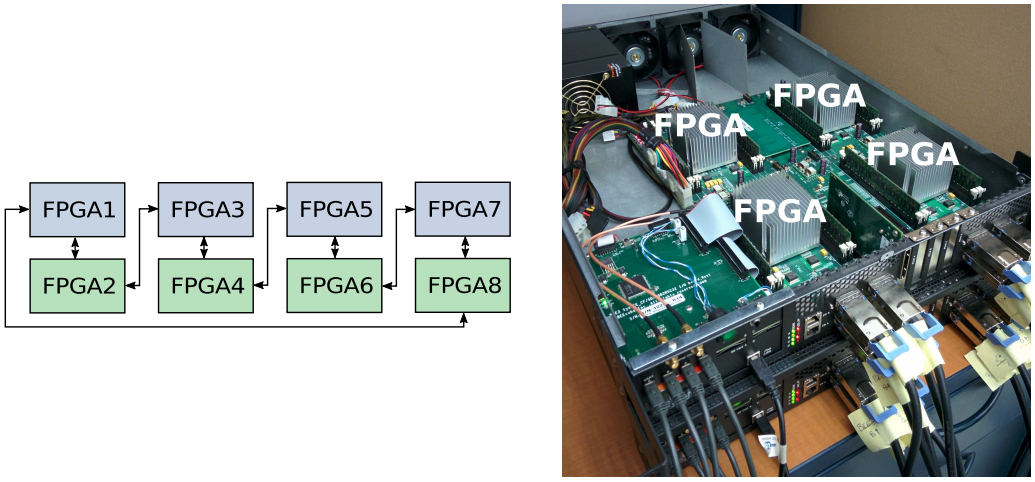

Figure 4.15 Multi-FPGA platform for testing the communication IP. Eight-FPGA machine based on two BEE3 systems.

4.4.2 Implementation

The communication IP's transceivers are mapped to the dual 10GBASE-CX4 interfaces, bringing out two high-speed communication channels per FPGA. Each channel directly connects a neighboring node. A direct network of FPGA devices in a ring topology is now supported by the IP. Figure 4.16 shows a technology view of the implementation system.

For testing purposes, a pseudo-random generator/checker is connected to a transition and reception Socket as an Endpoint. This generator/checker is a vhdl module designed to inject packets with variable payload length. The packet's source and destination IDs were set as the same for each node, so each FPGA sends itself pseudo-random packets. A MicroBlaze-based

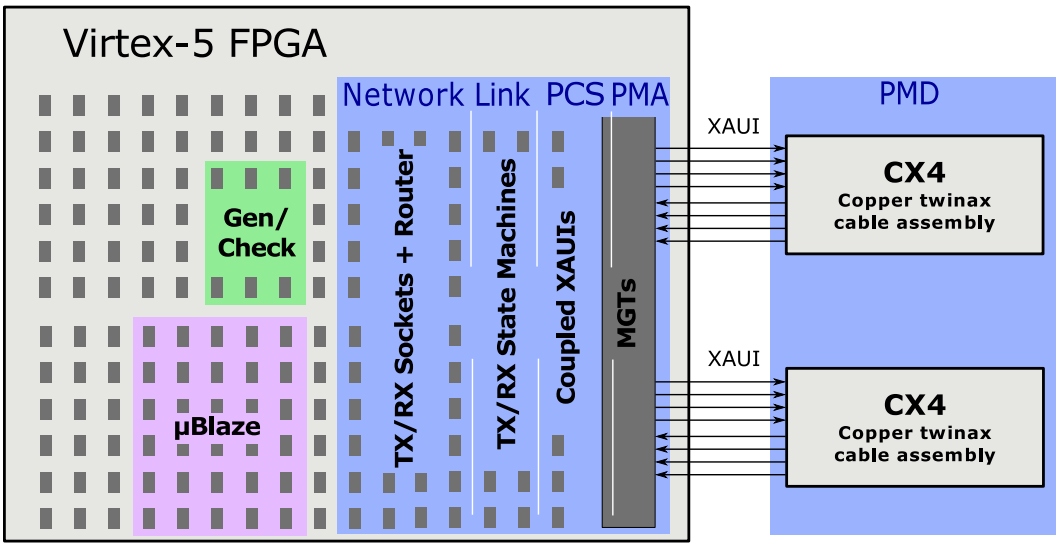

Figure 4.16 The implementation system. A technology view.

system aids the IP configuration for the routing table, which is set up to loop back all of the received packets. It also connects to the communication IP through the software interface modules (Figure 4.11). Figure 4.17 shows the packet generator and checker architecture. The module is capable of injecting as many packets as the transmission Sockets can handle. In other words, it stresses the communication channel by keeping the transmission buffer full. The pseudo-random generator circuit uses 64 registers accessible in parallel to conform packet's payload. The generator's seed is specified at design time as well as the packet header and the tail (constants). The module allows variable payload length packet injection, going from 1 to n bytes.

The checker circuit is similar to the generator. When a packet arrives at the Rx Socket, the FIFO Stream beats from the network are compared to those from the checker. An error is signaled when there is not a match.

Bit error ratio (BER)

At 10Gbps speed, some of the channels show errors in the received data. The XAUI modules often signal loss of synchronization and in some cases, the links are completely down. To debug the channels and identify the source of errors, the transceivers were tested in all four possible loopback-path configurations. Each configuration tests a different transceiver section in order to discard faulty components.

1. *Near-end PCS loopback*

   The data path loops back at the PCS layer in the local FPGA. The data never gets

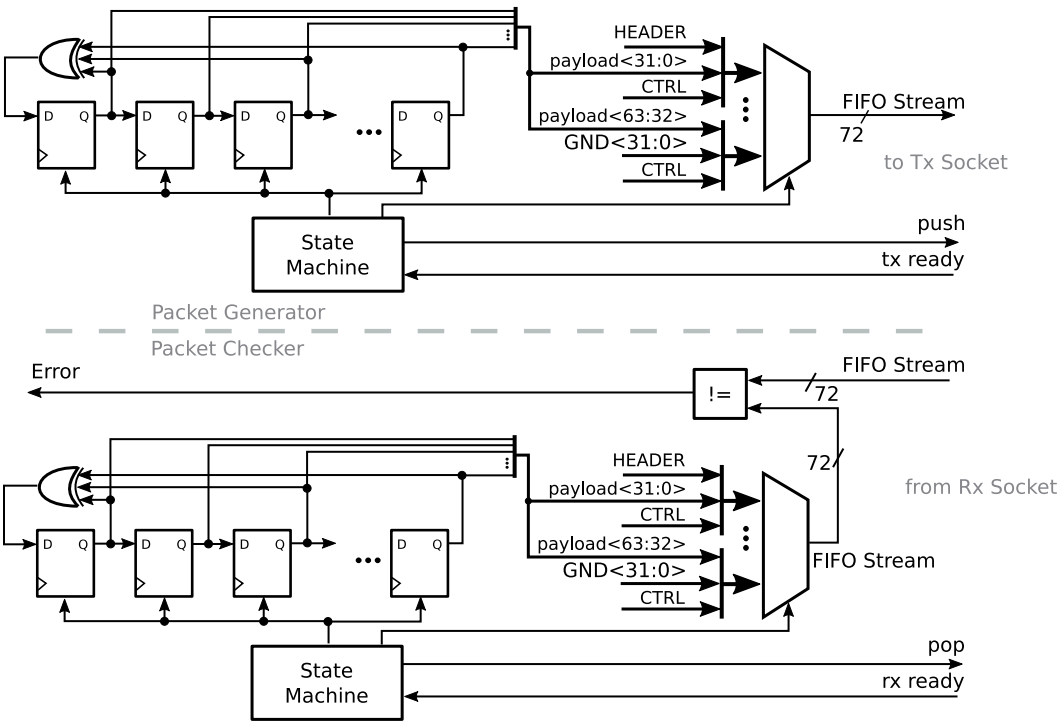

Figure 4.17 Packet generator and checker architecture.

out of the device. The PMA section is not involved.

2. *Near-end PMA loopback*

   The data path loops back at the transceivers' PMA layer, just before the line drivers. The data remains at the FPGA's internal boundary.

3. *Far-end PMA loopback*

   The data path loops back at the PMA layer in a remote FPGA. The CX4-to-CX4 passive copper cable is part of the path. The data gets out of the local FPGA. The remote PCS section is not involved.

4. *Far-end PCS loopback*

   The data path loops back at the PCS section in a remote FPGA. The data travels from the source to the destination device and it is automatically looped back at the PCS. The received data is also presented to the remote user logic.

The two Far-End loopback configurations indicated that the source for the faulty links could be caused by imperfections in the BEE3's PCB or in the CX4 cable connections. This was concluded since the two Near-End configurations didn't show errors and the Far-End ones did.

The Integrated Bit Error Ratio Tester (IBERT) tool from Xilinx is used to tune up the links. This tool allows real-time Multi-Gigabit Transceivers monitoring and configuration through Chipscope. It utilizes the embedded Pseudo-Random Bit Sequence (PRBS) engine at each transceiver to generate and check bit sequences over the link. A real-time report of the Bit Error Ratio (BER) is displayed whilst changing the transceiver block configurations. The sweep feature is used to test a range of configurations. The report shows that some links reach down to 10^{-9} BER which is worse than the BER operation defined for XAUI standard (10^{-12}) [41].

To overcome this problem, the transceivers speed is reduced to 2.5Gbps of the 3.125Gbps initially set for XAUI operation. This reduces the XAUI channel bandwidth to $2.5 \cdot 4 \cdot 0.8 =$ 8Gbps after 8b/10b encoding. A BER of 10^{-12} is now reached in all of the links. The new set of parameters for the transceivers configuration is extracted from the IBERT tool and used in the eight-FPGA machine. The Xilinx XAUI IPs are also configured to operate at the reduced speed. Both new transceiver and XAUI configurations are specified manually by modifying the IP source files. Figure 4.18 shows a screenshot of the tuning process with Xilinx IBERT tool. From there, is it possible to see the 8 transceivers running at 2.5Gbps, sending and receiving Pseudo-Random Sequences of 7 bits (PRBS-7) which are similar to the 8b/10b encoding scheme. The Figure only shows the window panel for one FPGA, that one configuring the transceivers to shoot out raw data bits (no loopback). However, another window panel associated to the immediate neighbor FPGA would show the transceivers configured in loopback so that the links can be tested. The transceivers' status "Linked" is only reached after a certain level of good Bit Error Ratio.

Finding a good and common configuration set for all of the transceivers in the eight-FPGA machine was a long process. Listing A.1 shows the initial transceiver configuration from Xilinx Core Generator at 10Gbps operation rate, and the new configuration obtained after IBERT tuning up for 8Gbps line speed. The code displayed comes from the `xaui_rocketio_wrapper_tile.v` source file which contains the attributes and configuration port states for the transceiver dual-tile. Initially, the shared Phase Locked Loop (PLL) module at the transceiver dual tile is set properly for 125Mhz reference clock and 2.5Gbps line rate. Then, the receiver equalization circuit is enabled by tying `RXENEQB0` port to ground and its parameters configured according to IBERT tune up. For more details about the receiver equalization circuit refer to [42].

XAUI's new attributes are related to the Digital Clock Manager module configuration (Figure 4.3). They are updated in the same way as the PLL attributes are specified to the transceiver dual-tile.

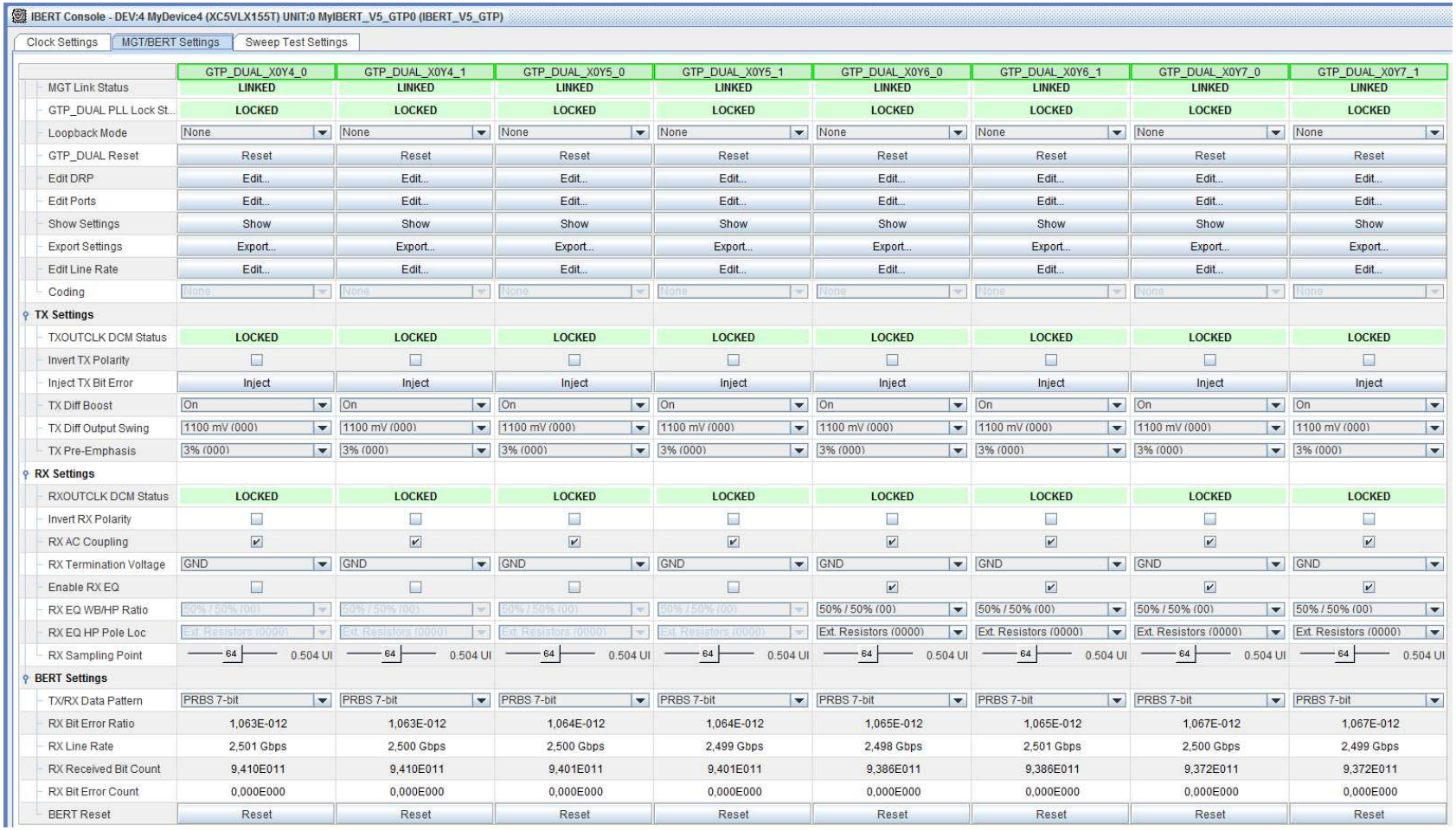

Figure 4.18 Xilinx IBERT screenshot for link tune up.

Maximum clock frequency

After implementation, the synthesis tool reports a maximum clock frequency for the communication IP of 220MHz. However, for XAUI configuration (10Gbps), the IP most be driven at 156.25MHz. For the reduced speed configuration (8Gbps), a clock frequency of 125MHz should be used.

Device utilization

Table 4.2 summarizes the resource utilization for the communication IP configured for two Tx/Rx Sockets. A two Tx/Rx Sockets configuration actually uses three transmission and reception Sockets because of the Default Socket. Each of them consumes two Block RAMs for buffering (FIFO), one per channel. The Virtex-5 FPGA contains 16 MGTs structured as eight GTP Dual-Tiles, of which the communication IP utilizes the half. The small footprint of the proposed architecture ensures enough room for user applications.

End-to-end latency The minimum theoretical latency for the Link and Network layers, and 32 bits packet's payload length is 4 clock cycles in transmission and 3 clock cycles in reception. Figure 4.19 presents the minimum latency per functional block. Some block's latency depends on the configuration used for the communication IP, especially on the number of Tx and Rx Sockets. As explained above, in the router module and the FIFOs (First-Word-Fall-Through type) have only 1 and 2 clock cycles latency respectively. The switching module's latency on transmission depends on the number of Tx Sockets and the round robin scheduler. Its latency can go from 0 to NB_TX_Socket including the scheduling time for the Default Tx Socket. The Rx and Tx state machines have 1 clock cycle minimum latency each. However, the XGMII alignment at the receiving XAUI interface may affect the Rx state machine in 1 clock cycle. Demultiplexing the incoming packets is fully combinational, so no latency is incurred at this stage. Finally, the round robin scheduler at the Rx Socket adds 0 or 1 cycle to the entire communication process.

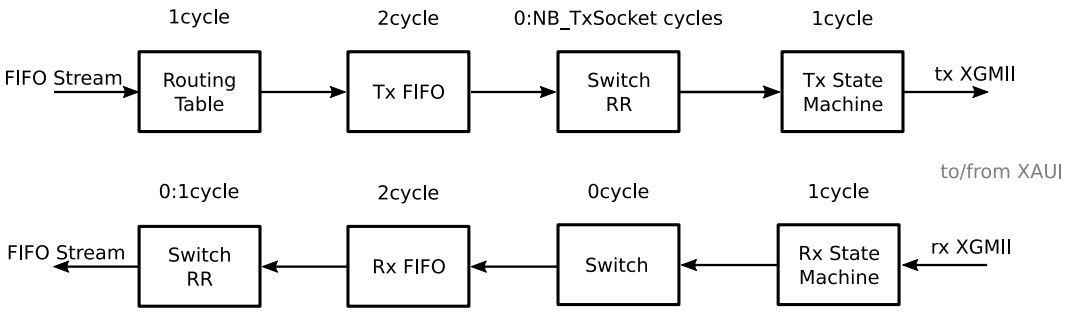

Figure 4.19 Latency estimation per block on the communication stack.

Table 4.2 Resource utilization for communication IP in a Virtex-5 xl155t FPGA.

Resource	LUTs	FFs	Slices	Block RAMs	GTP Dual
Utilization	2289	2777	1135	12	4
%	2	2	4	5	50

To verify the minimum theoretical latency estimation, the communication IP is simulated. Latency is measured from the moment an endpoint pushes the first packet beat into a Tx Socket until it is available to pop out at the destination Rx Socket. The testbench configures the IP for two Tx/Rx Sockets and loops back the channels at the network side (XAUI lanes in the channel up are connected to XAUI lanes in the channel down). The delays incurred due to the length of the serial connection (cable) are not considered in simulation.

Table 4.3 presents the minimum accumulative latency at four points across the data path. The Tx/Rx endpoint and the Tx/Rx XAUI regions refer to the communication IP user-side ports and the XGMII interface boundaries with the Physical layer respectively. The IP reach a minimum end-to-end latency of 34 clock cycles which is equivalent to 272ns at 125Mhz clock frequency. If the links were able to work at 10Gbps then at the nominal clock frequency for XAUI, the communication IP would have 217.6ns end-to-end latency.

Table 4.3 Latency report in clock cycles for the communication IP.

Region	Tx endpoint	Tx XAUI	Rx XAUI	Rx endpoint
Min. Acc. Latency	0	4	31	34

Figure 4.21 shows the waveform simulation associated with the reported latency measurements. The packet's payload length chosen for the stimulus is four bytes which fit in only one FIFO-stream beat. The main events during the transmission are summarized hereafter.

— At time T0, two broadcast packets with 4 bytes payload are pushed in parallel into two Tx Sockets. Only one FIFO-Stream beat is necessary for that.
— At time T1, the routing path is solved and the packet is ready to be driven through the adequate Tx FIFO.
— At time T2, the packet is pushed into a Tx FIFO.
— At time T3, the packet comes out from the data out port on a Tx FIFO being accessible from the Tx state machines.
— At time T4, the first XGMII beat is driven to a XAUI module. A single FIFO-Stream beat requires two XGMII beats. Following this point, it is possible to see in the waveform the two packets pushed at the beginning, being now transmitted consecutively. This actually shows a nice operation from the round robin scheduler at

the switching module and the Tx state machine.

— At time T5, the first XGMII beat is received at a XAUI module.

— At time T6, after the second XGMII beat arrives, the packet is fully received and it is pushed into an Rx FIFO.

— At time T7, finally the packet is ready to be read from an Rx Socket.

Throughput and efficiency

Protocols overhead is always a concern in any communication system. In the communication IP, the custom protocol implemented over the XGMII introduces twelve bytes overhead per packet (Figure 4.4 XGMII stream). Additionally, each new packet always starts in the next XGMII data beat. This means that in the worst case, seven bytes must be also added as overhead. Depending on the packet length, the protocol overhead can go from twelve to nineteen bytes producing a sawtooth behavior on the communication IP's throughput chart (Figure 4.20).

The available bandwidth is 16Gbps split into two 8Gbps channels. Because of the variable payload length feature, the actual throughput depends on the payload size. The worst throughput is 0,9Gbps for one-byte payload packets, which represents 5,56% channel efficiency. This poor behavior of the communication channel is rapidly improved by increasing the payload size. With only 100 bytes payload, less than 11% of the available bandwidth is utilized for the protocol (overhead).

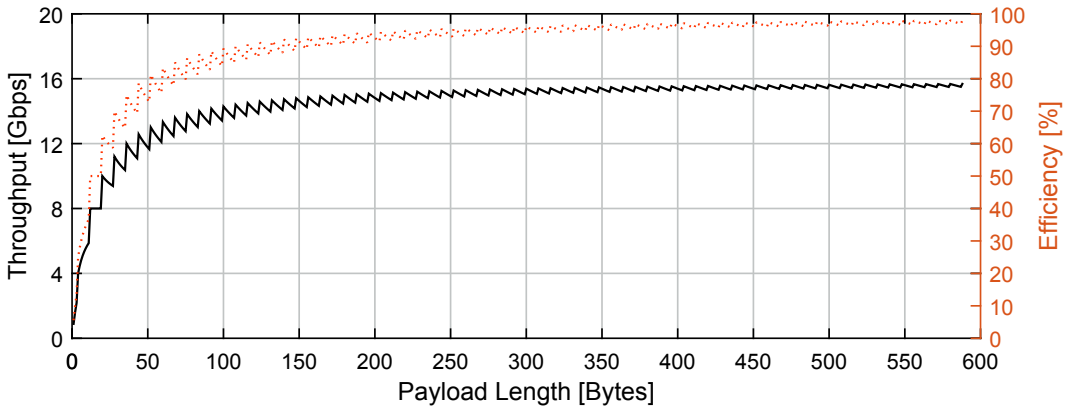

Figure 4.20 Custom protocol throughput.

Upgrading and boosting the communication IP

Upgrading the communication IP to work on modern FPGAs is straightforward. The XGMII standard interface decouples the Physical layer from the Link layer allowing an easy migration to newer transceiver technologies. For example, a Virtex-6 FPGA on a BEE4 multi-FPGA

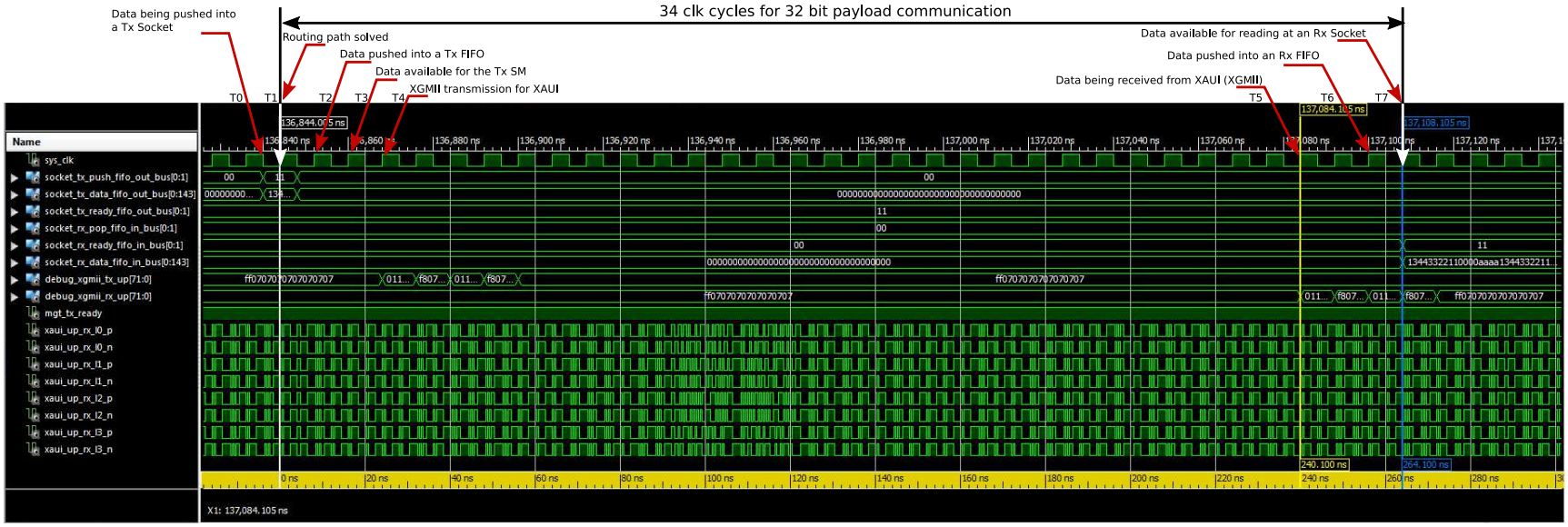

Figure 4.21 Minimum latency for 32-bit payload packets. Communication IP waveform simulation.

platform uses GTX transceivers capable of running at a line rate of 6.6Gbps. Depending on the FPGA speed grade, the XAUI cores on the communication IP can be driven at a clock frequency of 312,5MHz to reach DXAUI (Double XAUI) mode. In DXAUI configuration, the IP is capable of shutting out 20Gbps per channel data to the network; 40Gbps in total.

Another possible upgrading is replacing the XAUI cores by two RXAUI (Reduced XAUI) cores. The RXAUI cores are very similar to XAUI with the slight difference that they use only two transceivers to reach 10Gbps bandwidth. This is a tremendous improvement in terms of serial high-speed PHY utilization. If maintaining the initially four transceivers is not a problem, then four RXAUI cores could be implemented to provide a fanout of four at 10Gbps each. A small modification has to be done to the switching blocks in the Link layer to support four XGMII interfaces instead of two.

For higher bandwidth like 40Gbps and 100Gbps, some major modification has to be done. At the Physical layer, the XAUI cores have to be replaced with XLAUI (40 Gigabit Attachment Unit Interface) or CAUI (100 Gigabit Attachment Unit Interface) cores. At the Link layer, the Tx/Rx state machines have to be modified to support XLMII (40 Gigabit Media Independent Interface) or CMII (100 Gigabit Media Independent Interface). The XGMII and XAUI legacy would make this modification effortless.

4.5 Verification

The verification process for the entire design was separated in units and system verification. Units verification included tests to particular functions or low-level blocks. Integration tests were also carried out where the micro scale blocks were combined together. White-box and gray-box (because of the XAUI's protected netlist and built-in FIFOs) testing were mainly used for units verification with Xilinx ISIM as the main simulator. Among the techniques used there were basic and semi-automated testbenches with assertions and waveform monitoring.

System level verification was carried out directly on the FPGA platform. The software system (MicroBlaze and Software interface modules) and the Chipscope Analyzer tool were used for injecting stimulus, monitor behaviors, checking results and printing reports.

Table 4.4 highlights the main tests applied to the communication IP for verification. Only a summary is presented for illustrating de verification process.

Figure 4.22 shows an example of system level verification where the variable payload, the routing table, and the broadcast features are tested from the software interface. In the Figure, there are three console windows displaying information from the software application running on node 4, 5, and 6. Nodes 4 and 6 are the nearest neighboring nodes to 5, connecting

upstream and downstream respectively. The node 5 is used as the master device in a ring network topology. Initially, it sends 2 bytes (0x2211) to node 4 and 4 bytes (0x44332211) to node 6. Later, it injects a 4-bytes payload (0x88776655) broadcast packet to all nodes. The console windows show the transmission and reception process for both the master and the slave applications. It is possible to see part of the initial configuration process where the memory segments are allocated and bound to transmitter Endpoints, the routing table is populated and the broadcast mode is set.

Table 4.4 – Communication IP test plan summary for the transmission section.

Section	Description	Method
	Transmission	
FIFO Stream and XGMII protocol	The FIFO Stream protocol is well translated into XGMII protocol. The Tx state machine behaves according to the Control field in the FIFO Stream protocol. All the protocol fields are arranged properly (packet tail is well inserted in the XGMII protocol with the adequate length). Variable payload length is supported.	ISIM simulator: Testbench with from-file stimulus and reference model; waveform monitoring.
Packet split and recomposition	After an unterminated FIFO Stream packet injection, the transmitter truncates and appends the appropriated packet tail. Whenever the injection restarts, a new packet is assembled by appending the last packet's header.	ISIM simulator: Testbench and waveform monitoring. On-Chip test: Software interface stimulus and monitoring; Chipscope monitoring.
Broadcast	Packets with destination ID equal to 0x00 are pushed into both communication channels.	ISIM simulator: Testbench and waveform monitoring. On-Chip test: Software interface stimulus and monitoring; Chipscope monitoring.

Routing	The routing table is accessible from the software interface. The packets are forwarded to the communication channel according to its destination ID and the routing decision stored on the table. The router can loop back packets. The packet's body and tail follow the same path as the header.	ISIM simulator: Testbench and waveform monitoring. On-Chip test: Software interface stimulus and monitoring; Chipscope monitoring.
Tx software interface	The microprocessor environment injects packets into the network throughout the communication IP.	ISIM simulator: Testbench and waveform monitoring. On-Chip test: Software interface stimulus and monitoring; Chipscope monitoring.

The waveform displayed in Figure 4.22 associates to node 5. It captures outgoing packets from both channels at the XAUIs' XGMII interfaces. The waveform verifies the correctness of the custom protocol (Figure 4.4) and the features implemented. It shows the first packet targeting node 4, coming out from channel UP with only 2 bytes payload. The second captured packet targets node 6 with 4 bytes payload and goes throughout channel DOWN. Finally, the third packet traverses both channels at the same time, targeting all nodes in the network.

4.6 Conclusion

The presented architecture for an FPGA-to-FPGA high-speed communication was designed for low latency. Its' functionalities and main blocks were carefully selected to fit the basic needs in an FPGA network while maintaining a small footprint, low overhead, and high throughput. The IP was tested in an eight-FPGA network connected in ring topology. Some of the links showed constant bit errors at maximum speed, therefore a lower bit rate was used to compensate for it. The IP is highly efficient in terms of bandwidth utilization thanks to the variable payload length feature and the reduced protocol control fields. It should be noticed that the custom protocol implemented has unused fields for future functionalities support which were also included in the overhead and efficiency estimations. A friendly FIFO-like user interface and a customizable design (vhdl generics) abstract the communication process and make the IP easy to use.

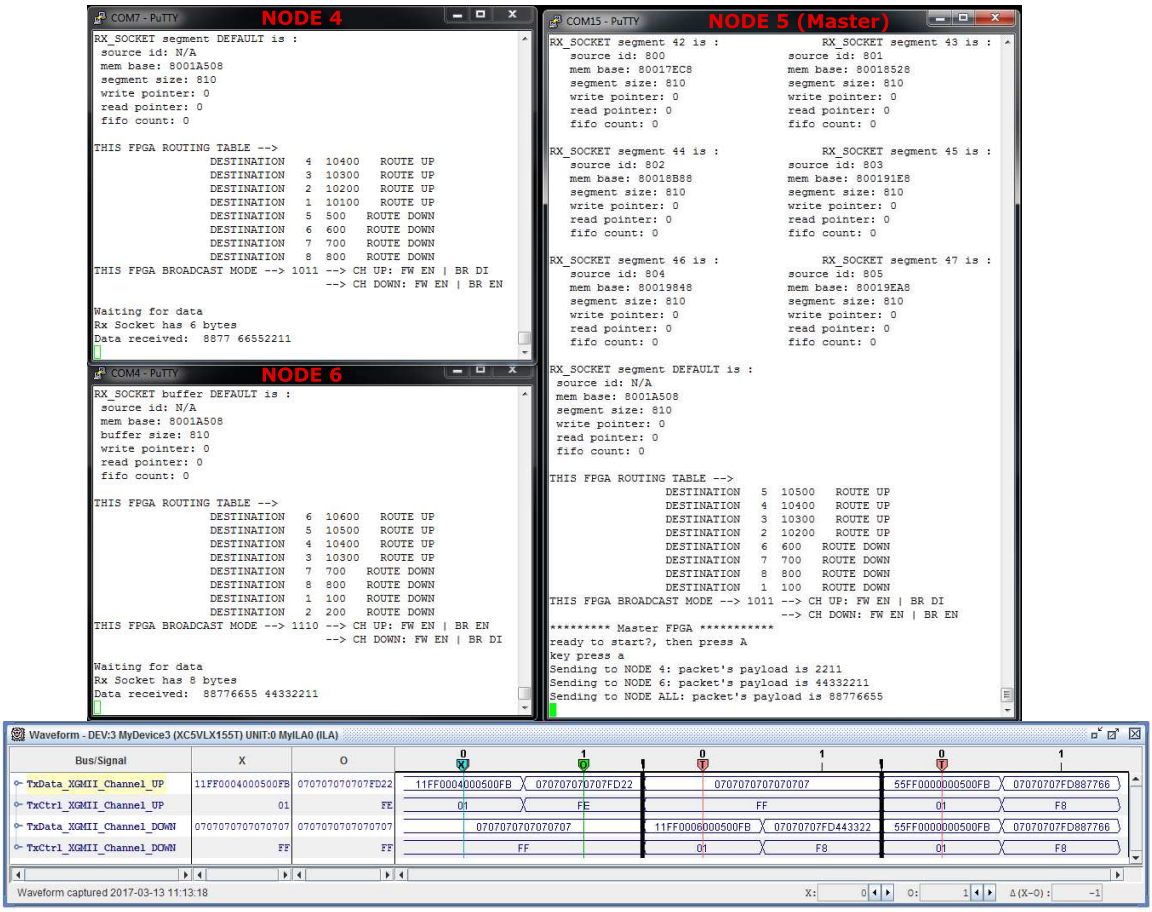

Figure 4.22 Features verification example with software stimulus and Chipscope monitoring.

A software interface has been designed for supporting full data transfer and IP configuration from a microprocessor environment. In packet reception, a memory organization system is implemented for storing the incoming packets y separated memory segments according to the senders' IDs.

The proposed architecture is capable of scaling to hundreds of nodes (FPGAs) and Endpoints, being the only limitation the targeting platform. The end-to-end latency on an FPGA network utilizing the communication IP is expected to scale linearly. Two communication channels interface the network side for the moment. This low connectivity limits the network topology to ring or linear array which isn't the ideal topologies for low diameter networks.

Time critical and distributed applications involving data transmission via high-speed links can be supported by the communication IP. Its low latency, high-bandwidth features as well as its direct network architecture, fit perfectly in the scheme of High-Performance Computing systems. Packet routing and broadcasting are supported allowing tight large System-on-Chip

partitioning over an FPGA-based cluster.

The architecture requirements presented in Section 4.1 are following reviewed to conclude this chapter.

1. *High-speed communication*

   The XAUI cores in the architecture bond four transceivers to reach 10Gbps bandwidth per channel. However, a reduced speed version of XAUI was used to overcome bit errors in some of the links. The reasons for high BER could be associated with jitter accumulation, signal noise and attenuation due to BEE3's long PCB routes, power supply noise, thermal noise and/or poor electrical connections in the CX-4 assembly. The operating aggregate bandwidth is then 16Gbps.

2. *Low latency communication*

   Basic functionalities for distributed FPGA-based applications and an optimized design have led to a low latency of 272ns (34 clock cycles).

3. *Efficient communication*

   The channel efficiency depends on the payload length. For that, the communication IP implements a variable payload length feature which allows reaching 89% efficiency with only 100 bytes payload (14 XGMII-stream beats including protocol overhead).

4. *Small footprint*

   The FPGA utilization is directly associated with the number of Sockets configured for the IP. Increasing the number of Sockets would lead to higher resource consumption due to more buffering and routing components. Despite that, the proposed architecture has a small footprint with only 5% device utilization for two Rx/Tx Sockets (two Rx/Tx endpoints can have access to the Network Interface Controller).

5. *Abstraction*

   A FIFO-like stream at the user side interface abstracts the communication process for all of the endpoints.

6. *Reliability*

   Reliability is achieved by means of the 8b/10b encoding scheme on the transceivers. When data arrives with wrong disparity or illegal 10-bit codes (out-of-table), the transceivers signal an error which is monitored by the XAUI core. This in turn checks for out-of-synchronization and misalignment lanes errors, providing thus a significant level of reliability on the communication channel.

CHAPTER 5 HIGH-PERFORMANCE COMPUTING TEST ON A MULTI-FPGA PLATFORM

In the previous chapter, it has been implemented a custom high-speed and low latency communication IP, suitable for High-Performance Computing applications on multi-FPGA platforms. The IP reduces the overhead associated with data movement and allows tight integration in large System-on-Chip (SoC) partitions. To demonstrate it, this chapter describes an HPC test implementation on an eight-FPGA machine utilizing the Communication IP as the Network Interface Controller.

5.1 HPC test: Dense matrix-vector multiplication

The highly parallel nature of matrix-vector multiplication makes it an ideal low-level application for using in an FPGA-based computing platform. Eventually, its implementation may be later extrapolated to higher level scientific and engineering application, giving a good idea of what the platform is capable of. Matrix-vector multiplication computations may be done in parallel by distributing multipliers and adders across several FPGAs, because of the loose dependency of the operations.

Consider the following dense (with only non-zero elements) $m \times n$ matrix multiplying a dense vector of length n (Equation 5.1). The resulting vector is the dot-product between each row of the matrix and the vector. The multiplications and the additions can be performed concurrently through MAC operators.

There are several matrix partitioning, organization and compressing methods to aid improve performance when calculating. Since implementing complex techniques for matrix-vector parallel multiplication is out of our goal and is time-consuming in a pure hardware approach, a simple row-major and block partitioning method, also known as "rowwise block-striped with replicated vector", is chosen for the HPC test. A block is n columns by m/num_nodes rows when possible, to guaranty the same processing amount per computing unit. Equation 5.1 shows an example of $2 \times n$ partitioned blocks, each of which will be assigned to a unique unit.

$$\begin{bmatrix} \begin{bmatrix} A_{00} & A_{01} & \cdots & A_{0n} \\ A_{10} & A_{11} & \cdots & A_{1n} \end{bmatrix} \\ \begin{bmatrix} A_{20} & A_{21} & \cdots & A_{2n} \\ A_{30} & A_{31} & \cdots & A_{3n} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} A_{m-10} & A_{m-11} & \cdots & A_{m-1n} \\ A_{m0} & A_{m1} & \cdots & A_{mn} \end{bmatrix} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} A_{00}x_0 + A_{01}x_1 + \cdots + A_{0n}x_n \\ A_{10}x_0 + A_{11}x_1 + \cdots + A_{1n}x_n \end{bmatrix} \\ \begin{bmatrix} A_{20}x_0 + A_{21}x_1 + \cdots + A_{2n}x_n \\ A_{30}x_0 + A_{31}x_1 + \cdots + A_{3n}x_n \end{bmatrix} \\ \vdots \\ \begin{bmatrix} A_{m-10}x_0 + A_{m-11}x_1 + \cdots + A_{m-1n}x_n \\ A_{m0}x_0 + A_{m1}x_1 + \cdots + A_{mn}x_n \end{bmatrix} \end{bmatrix} \tag{5.1}$$

Execution sequence

The execution sequence for the HPC test is based on a scatter and gather parallel pattern. It is explained hereafter.

1. *Matrix partitioning*

   The matrix is partitioned into equal-size blocks and stored locally for all FPGAs.

2. *Vector broadcasting*

   The vector is distributed from a master device to all other nodes throughout the interconnection network supported by the Communication IP (Chapter 4).

3. *Vector receiving and storing*

   The vector elements are received and stored locally for all FPGAs.

4. *Parallel computing*

   From the moment the vector is available at a computing node (FPGA), the computation starts. Elements from a row-major matrix traversal and the broadcasted vector are taken into MAC operators for computing dot-product. A full run is completed when all of the elements in a row are computed to all the elements in the vector and a final resulting element is obtained.

5. *Resulting vector gathering*

   Each computing unit sends back the results to the master device.

Number of operations

The HPC test is computed in floating-point double precision (64 bits). The number of mathematical operations involved is expressed in floating-point operations which can be obtained through the Equation 5.2. The speed of which these operations are solved is used as a parameter for measuring the performance of any computing machine. For example, for a 4000×2000 matrix multiplying a 2000 elements vector, the amount of floating-point

operations is 15996000. Supposing that it can be computed in $1ms$, then the performance of the machine is about $16GFLOPS$.

$$FLOPs = (2n - 1) \times m \tag{5.2}$$

5.2 Platform architecture

The target platform is the eight-FPGA machine described in Subsection 4.4.1. The High-Performance Computing (HPC) test consists of a single hardware/software design (bitstream) for all FPGAs in the network. Its design integrates a memory system, several processing units, a microprocessor and a Network Interface Controller (the Communication IP). Figure 5.1 presents the architecture for the HPC test.

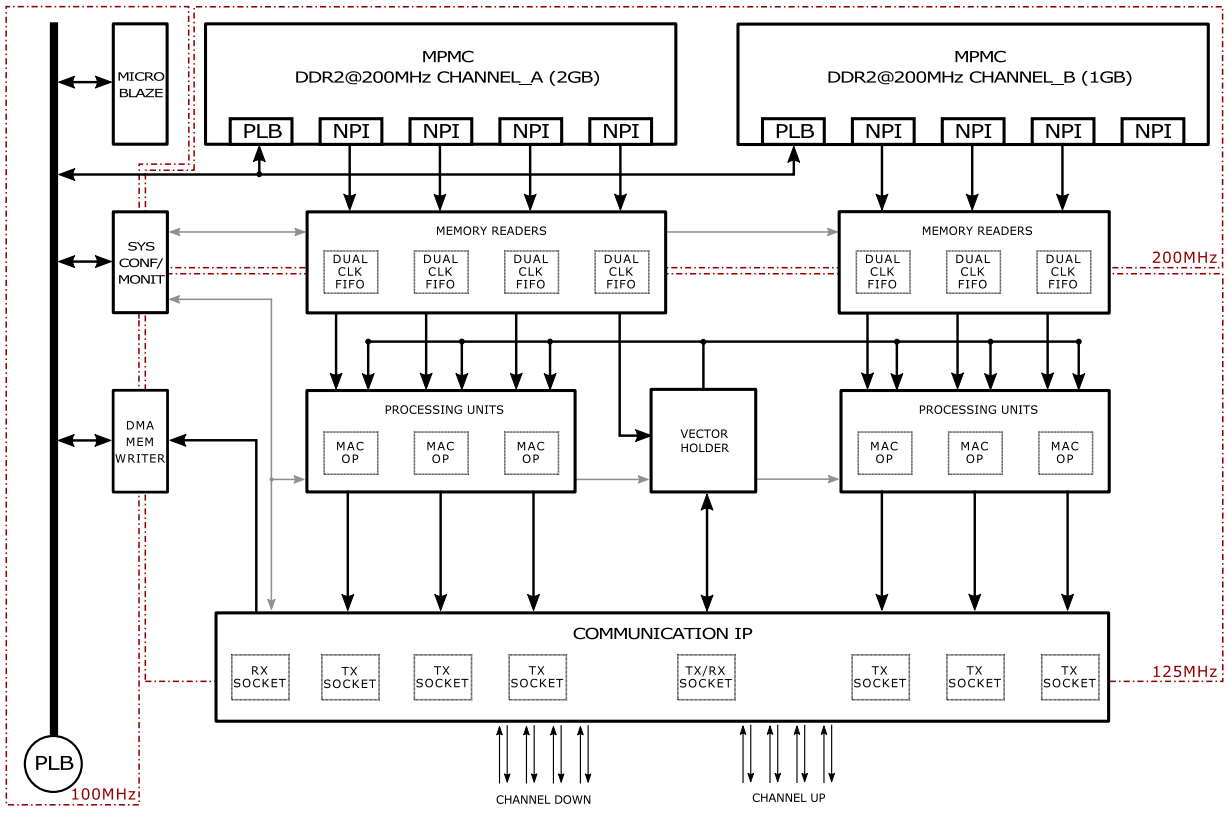

Figure 5.1 Architecture for the High-Performance Computing test.

There are three main clock regions. Synchronizers and dual-clock FIFOs are used when needed to ensure safe clock domains crossing.

1. *The 200MHz clock region*

This clock region includes the external memory system which operates at DDR2@200MHz 64-bits data interface.

2. *The 125MHz clock region*

   This clock region includes the hardware processing units and the Communication IP. The IP is driven at this frequency to ensure reliable high-speed communication, as presented in Chapter 4.

3. *The 100MHz clock region*

   This clock region includes the MicroBlaze microprocessor system. The MicroBlaze couldn't operate at 200MHz because of timing failures neither at 125MHz because is not compatible with the external memory system. The external memory controller in the BEE3 is configured to operate at 200MHz to achieve DDR2 400. The devices interfacing the controller may operate at the same or at the half of the memory controller's operating frequency [4]. Hence, 125MHz remains as an incompatible clock frequency when directly interfacing the off-chip memory.

5.2.1 Memory system

As many embedded systems, the platform has on-chip and off-chip memories. Both are used in the HPC test to fully exploit the platform capabilities.

Off-chip memory

The BEE3 platform has installed two 2GB DDR2 DRAM memories per FPGA in a dual channel configuration. To fully exploit this memory capability, two independent memory controllers are instantiated. Xilinx's Multi-Port Memory Controller (MPMC) solution is used along with a Memory Interface Generator (MIG)-based PHY interface.

The MPMC [4] has eight independent interface configurable ports to the user application, providing multiple accesses to the off-chip memory, which means that several cores could be directly connected to the MPMC and independently issue requests to and from memory. The MPMC provides the Native Port Interface (NPI) as a custom interface for high bandwidth and low latency transfers to memory. It allows a direct connection to the MPMC without arbitration, and hardware cores to be driven at the memory controller's operating frequency. Other configurable interfaces are also supported for more standard solutions i.e. PLB interface.

The memory system configures one port on both MPMC for 32-bits data PLB interface and the remaining ports for 64-bits data NPI. The PLB interfaces are used to provide memory accesses through a shared bus interface connecting a MicroBlaze processor and other periph-

erals. The NPI interfaces attend requests from hardware cores. The arbitration between the ports is set fixed with the highest priority on the $PORT0$. This simplifies the arbitration logic, saving resources and helping to meet timing constraints. The memory controllers are driven at a clock frequency of 200MHz as well as all of the accessing ports excepting the PLB configured ports which are driven at 100MHz because of the MicroBlaze processor frequency limit.

Xilinx does not directly support NPI integration into a hardware core through generated templates, thus the configuration has to be done manually by modifying the source files [43]. Xilinx neither supports the BEE3 board during the Memory Interface Generator (MIG) GUI flow, hence the PHY connexions and constraints must be specified manually using BEE3 documentation.

The NPI has the highest performance on the MPMC. Table 5.1 shows the NPI performance estimation for the proposed memory system. This estimation depends on the interface configuration and on the hardware core behavior driving it.

Table 5.1 NPI latency and throughput for the implemented memory system [4].

Number of Ports	Memory Interface	NPI Width	NPI Burst Type	Initial Transaction Latency	Max Total Data Throughput
3-8	DDR2@200MHz 64 bits	64 bits	32 DWord	23 Clk cycles	17.8Gbps

With two MPMC interfacing two RDIMM memories, the platform counts with 35.6Gbps off-chip data throughput per FPGA, 284.8Gbps global.

NPI memory readers

To guarantee the maximum possible data throughput from the MPMC, the modules (memory readers) connected to the NPI ports implement back-to-back transfers, also known as double buffering. The memory readers are basically composed of a dual clock FIFO and a control unit that manages the NPI protocol. Each reader receives, through an input port, the memory base address and the amount of data to be read from the memory. After a triggering event (ready signal), the data is requested to the MPMC and stored into the MPMC's internal FIFO. Thus, NPI interface controller leverages the buffering capabilities at every MPMC port. Once the data is ready to be popped out (NPI_empty deasserted), the NPI read latency has to be considered for validating the data at the NPI_Rd_Data port. Since the rest of the design cannot work at the same clock frequency as the Multi-Port Memory Controller, a built-in dual clock FIFO is used for synchronizing. Figure 5.2 present the architecture of the Native Port Interface memory reader described above. The NPI read controller keeps the MPMC's internal FIFOs as full as possible by queueing read requests.

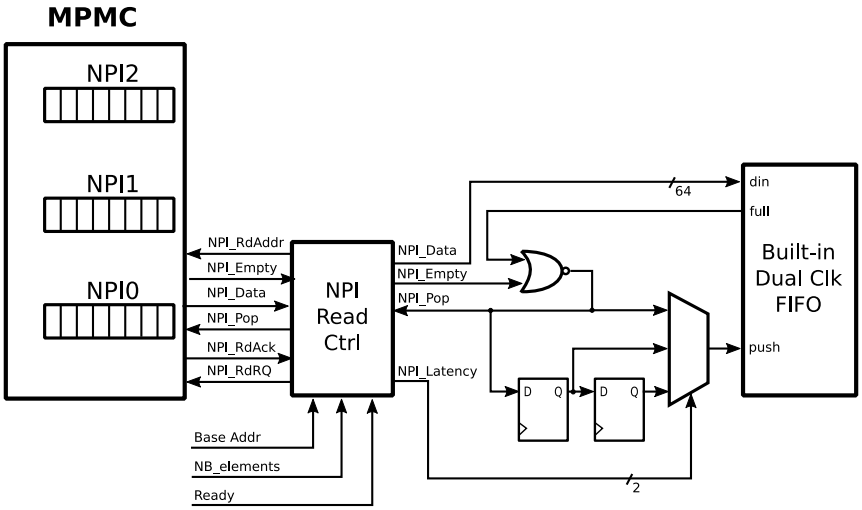

Figure 5.2 Architecture of the native port interface memory reader.

It monitors the FIFO's state to avoid overflow and data corruption. Listing B.1 presents the hardware description code of the NPI read controller for 32-double-word burst read transfers.

Data allocation and memory population

For the matrix-vector multiplication application, the matrix and the vector must be allocated on the external memory as well as the results of the calculation (golden or reference) for verification purposes. Since the BEE3 platform has two DDR2 RDIMM memories installed per FPGA, each of them in an independent channel, two MPMCs are used to increase the global memory bandwidth. Data is then stored in both memories through a PLB interface port. Both MPMCs are connected to a PLB bus providing access to a MicroBlaze microprocessor and some others peripherals as needed. The MicroBlaze is a 32-bits RISC microprocessor, so its addressable capabilities are $2^{32} = 4$GB. Then 2GB of the total address space is assigned to the $DDR2\_CHANNEL\_A$, 1GB to the $DDR2\_CHANNEL\_B$ and the remaining to peripheral devices and local on-chip memory. Only 3GB storage are available of the physical 4GB (two 2GB RDIMM), however, the memory system counts with a powerful 35.6Gbps external data throughput per FPGA.

The memory population process is done during the downloading process of the executable file (elf) to the MicroBlaze. The elf file contains the data which is going to be placed on the off-chip memories. For that, the memory regions have to be specified beforehand by means of the linker script. Data alignment has to be also specified since NPI imposes a data alignment constraint on burst transfers. Data must be aligned to the transfer size, that is, each address must be on the boundary of the transfer length. For 32-dwords transfer size, the alignment must be 256-bytes boundary (starting address must be 0x000 or 0x100).

On-chip memory

The platform exploits on-chip memory for local and fast storage. To optimize on-chip memory utilization and save logic resources, different design techniques are considered. Distributed and Block RAM memories are used depending on the size needed. A smart utilization of its aspect ratio based on the primitives (the basic building blocks) saves the proposed design from resource misuses i.e. half a BRAM wasted. Built-in FIFO primitives are also used when necessary.

The $VECTOR\_HOLDER$ module (Figure 5.3) is a FIFO/circular buffer implemented on BRAM memory. Its purpose is to hold the vector data and to loop on it for every row in the matrix. Initially, it behaves like a regular First-Word-Falls-Through FIFO allowing data processing as vector arrives. After the entire vector has been received, it changes its behavior to a circular buffer. It has two input sources for receiving the vector data; one connected to an NPI memory reader which is used when the platform is configured as a master device, and the other connected to the Network Interface Controller (the Communication IP) for receiving the vector from the network (slave device). The operating mode is changeable in runtime through a configuration port ($MASTER\_config$). The $VECTOR\_HOLDER$ module feeds one of the operand's inputs at the processing units (MAC cores). On master mode, the vector is forwarded to the slave devices using the broadcast feature of the Communication IP. The module holds up to 4000 floating point double precision vector elements.

Largest possible matrix-vector multiplication application

The architecture for testing has 3GB off-chip memory and 32KB on-chip memory for holding the matrix and vector respectively. Considering that the master device has to store more data than the slave devices because of the verification and the scattering/gathering process, the largest possible matrix and vector estimation is done for the master. Table 5.2 summarizes the main constraints that limit the matrix and the vector sizes in the HPC test.

Table 5.2 Maximum matrix-vector data set size for matrix-vector multiplication on the BEE3 platform. Architecture constraints.

Constraint	
Max_VSize (vector size) 32KB on-chip memory capacity	4000 elements
NB_Nodes (number of FPGAs)	8 (1 master, 7 slaves)
$Offchip\_Mem$ 3GB off-chip memory capacity	402 653 184 elements

Equation 5.3 describes the relation between the off-chip memory capacity and the maximum application data allowed, where Max_Rows is the maximum number of matrix rows and Max_VSize is the maximum vector size constrained by the $VECTOR\_HOLDER$ module

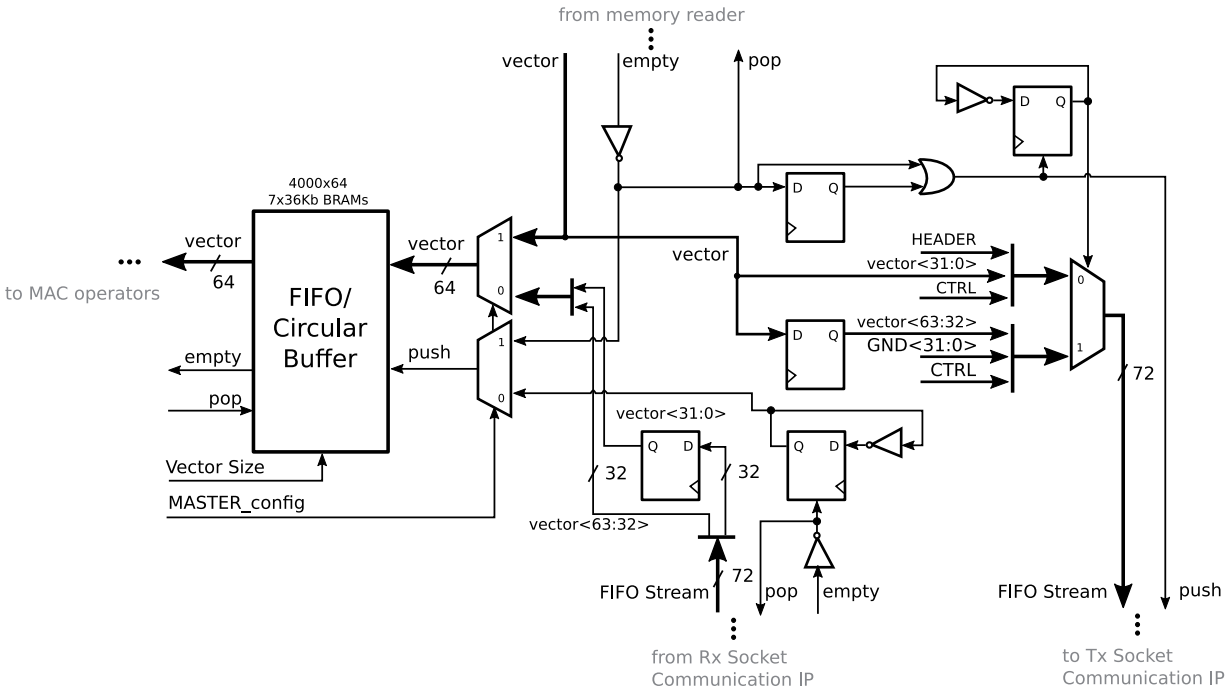

Figure 5.3 Vector holder architecture.

depth. After applying Table 5.2's constraints, the maximum matrix size that can be computed on the platform is 803688×4000 elements which can be decomposed into eight 100461×4000 submatrices, one per node (FPGA). No matrix compression techniques are considered.

$$\begin{aligned} Max\_Rows \times (Max\_VSize + NB\_Nodes) + \\ \Rightarrow Max\_VSize = Offchip\_Mem \end{aligned} \tag{5.3}$$

Matrix organization into memory

In computing, row-major order and column-major order describe methods for arranging multidimensional arrays in linear storage such as memory. The matrix organization in the memory system is done in row-major order where consecutive elements of the array are continuous in memory. This works accordingly with the matrix decomposition and parallelization strategy "row-wise block-striped with replicated vector".

Direct memory access and PLB memory writer

Writing in memory in an organized manner is very important for any application. For that, the architecture counts with a memory writer module connected to the PLB bus (rx software

interface from Section 4.3.2). This module performs Direct Memory Accesses (DMA) to the off-chip memories through the PLB interface port at the MPMCs. The PLB interface is chosen for the writer instead of the NPI interface because the result writing rate is less than the reading and processing rates. Hence, it is unnecessary and inefficient in terms of resource utilization to employ a new interface for this only purpose, while there is a shared PLB interface that meets the requirements.

The memory writer (rx software interface) at the master device receives the resulting data from all of the nodes in the network. The data must be organized in the memory as it arrives at a Reception Socket at the Communication IP.

The writer uses memory pointers for all of the processing units on the network. Each pointer points to a memory region defined by a base address and a maximum buffer size. When the resulting data arrives at the Reception Socket, the packet's Source field is utilized for selecting the associated memory pointer. Figure 5.4 illustrates the memory map for the off-chip memories.

The maximum buffer size allowed by the writer hardware is 1MB per processing unit which is enough to store the largest possible result based on the largest possible matrix-vector multiplication application that can run on the platform.

5.2.2 Processing units

The architecture in Figure 5.1 has six processing unit capable of computing six dot-product operations in parallel. The operators' inputs are connected to an independent NPI memory reader ($OPERAND\_A$) and to the $VECTOR\_HOLDER$ module output ($OPERAND\_B$). The NPI memory readers feed one operand input with a fraction of the local block-striped matrix while the other operand is the replicated vector. The dot-product operators are data flow controlled modules, operating only and immediately when there is valid data at each operand input. Figure 5.5 presents the dot-product operator architecture. A FIFO-like interface simplifies and unifies modules' interconnections.

The operator is structured in three main stages.

1. *The multiplication stage*

   This stage is composed of a floating point double precision multiplication operator pipelined in 6 stages and a control unit that monitors for valid data at each operand input. The operator is implemented on 13 Virtex-5 FPGA DSP48E slices. Xilinx Core Generator tool is used to generate the multiplier primitive.

2. *The partial sums computation stage*

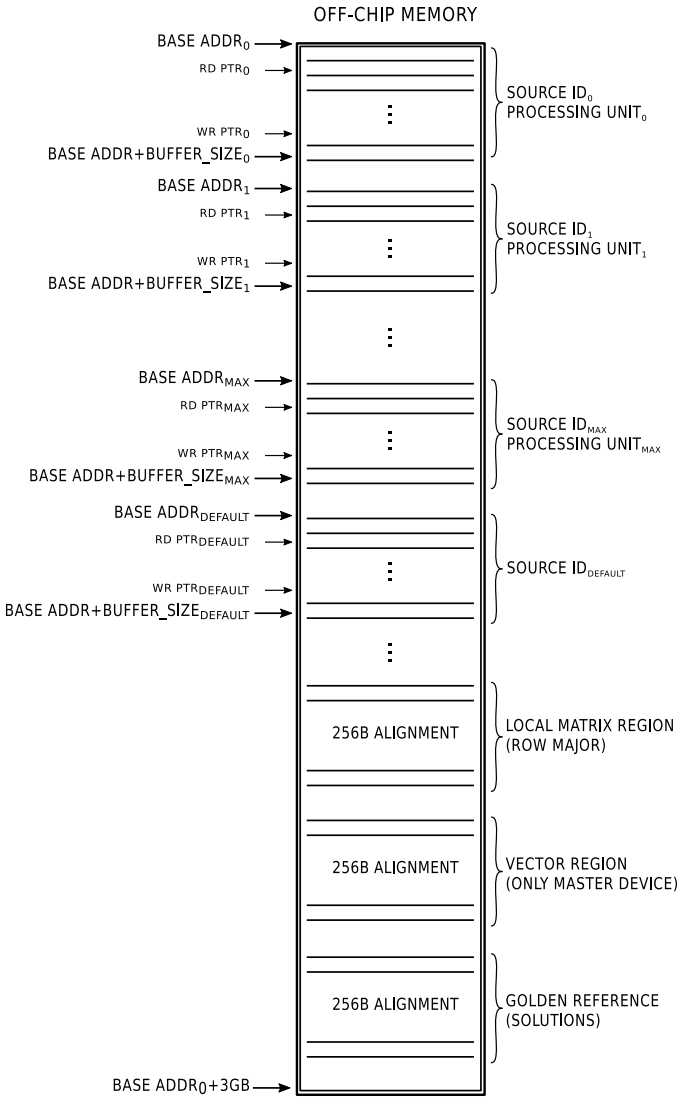

Figure 5.4 Off-chip memory map and memory writer module look-up table.

A floating point double precision accumulator integrates this stage. Partial sums are computed and accumulated from the previous stage results. The operator has 8 cycles latency, thus 8 partial sums are stored in pipeline. A local control unit monitors for valid data at the inputs and keeps track of the processed column number for reinitialization purposes. When an entire row and vector have passed, the control unit throws out the pipelined partial sums to the next stage for reduction while can keep accepting new income data.

3. *The partial sums reduction stage*

   The main purpose of the third stage is to reduce the partial sums computed in the previous stage without stalling the incoming data flow. Similar to the second stage, an

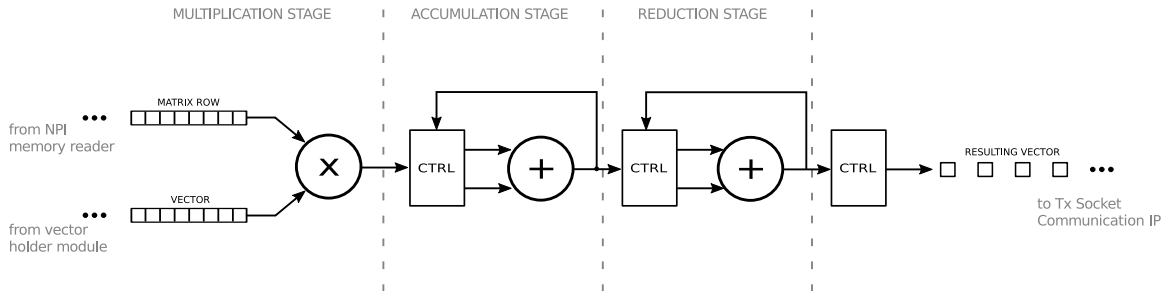

Figure 5.5 Dot-product operator architecture.

accumulator is implemented in 8 pipeline stages on 3 Virtex-5 FPGA DSP48E slices. The partial sum reduction process takes $Lat_{add} \times \log_2(Lat_{add})$ clock cycles to produce a final result. Since the adder operator has a latency of 8 cycles ($Lat_{add} = 8$), the reduction process takes 24 clock cycles. After partial sums reduction completion, the result is then packed and sent to the master FPGA through the Communication IP.

Our dot-product operator implementation is similar to the dot-product operator for a gaxpy BLAS level-2 routine implementation in [35]. The differences are in the dual-stage accumulator which in our case it is not implemented in a binary adder tree. Instead, we use a partial sums reduction stage with control logic and feedback. This allows us to reduce logic resources associated to the adder tree.

Each dot-product operator is connected to a Transmission Socket at the Communication IP. The Source, Destination, and Control fields of the Custom protocol (Figure 4.4 FIFO-stream) are appended to the dot-product result similar to the vector packing circuit in Figure 5.3. All of the operators have a different and unique ID which is used in combination with the FPGA (node) ID as the Source field. This will be lately used to organize the received data at the master device.

The operators are driven at a clock frequency of 125MHz and can accept a new pair of operands every clock cycle. Since the operators are data flow controlled, the operand supplier's rate must match the processing rate to prevent operators from stalling. That is 8Gbps data rate per operand port. The $VECTOR\_HOLDER$ module provides a half of the processing rate while the other half comes from the MPMCs. Both MPMCs combined have a maximum data throughput of 35.6Gbps which is less than the required 48Gbps from matrix operands ports (8Gbps operand bandwidth $\times$ 6 operators). This means that the dot-product operators are working at 74% capacity because of the external memory bandwidth limitation and 26% of the time, the operators will be stalled.

The main advantages of the proposed three-stage dot-product operator are that the size of

the computed matrix-vector can change without affecting the resource utilization on the targeting FPGA platform. Hence, the matrix can scale. The operators are data flow controlled modules computing only and immediately there is valid data at their inputs. High processing performance is achieved thanks to the dual-stage accumulator, which is capable of accepting a new set of (matrix_row,vector) data elements every clock cycle (125MHz). Only 30 clock cycles are needed to fully reduce the partial sums and produce a result after the entire (matrix_row,vector) arrays have entered the operator. The total latency estimation equation for the proposed dot-product operators in a $matrix\_row \cdot vector$ application is described by the Equation 5.4.

$$Total\_Latency = Lat_{mult} + Lat_{add} \times \log_2(Lat_{add}) + Vector\_Size \tag{5.4}$$

Table 5.3 summarizes the resource utilization for one and six floating point double precision dot-product operators implemented on the Virtex-5 xl155t FPGA.

Table 5.3 Resource utilization for the floating point double-precision dot-product operators on a Virtex-5 xl155t FPGA.

Resource	LUTs	FFs	Block RAMs	DSP48E
		One operator		
Utilization	2078	1693	0	19
%	2	1	0	14
		Six operators		
Utilization	12872	10760	0	114
%	13	11	0	89

Computing power and data solution rate

Parallel processing is achieved across the eight-FPGA platform as a result of the distribution of the processing units. In total 48 dot-product operators are implemented. From Equation 5.2, for an $m \times n$ matrix, it is needed $(2n-1) \times m$ floating point operations ($FLOPs$) which can be solved by a single operator in $30 + m \times n$ cycles (assuming no external memory bandwidth limitation). This can be approximated to 2 $FLOPs$ per clock cycle. At 125MHz, a single dot-product operator has a theoretical computing power of approximately $0,25GFLOPS$. Hence, the global theoretical computing power is about $12GFLOPS$. However, as all memory bound application, the memory controllers cannot supply the needed 6GB/s memory throughput per FPGA to the operators. Instead, they can only reach 4.45GB/s which reduce the platform maximum computing power to $8.9GFLOPS$.

The results producing rate of the processing units depends on the vector size. This rate has

to be low enough to be handled by the Communication IP. The bigger the vector size, the lowest the data solution rate. Equation 5.5 presents a pessimist (upper bound) estimation for the processing units' data solution rate where no external memory bandwidth limitation is considered. For the present use case application, the Sol_Width is 64 bits (floating point double precision) which incurs in 33.3% channel efficiency. The Clk_Freq and the $NB\_Processing\_Units$ are 125MHz and 6 respectively, and considering a small vector size ($Vector\_Size$) of 100 elements, the processing units' data solution rate estimation is about 480Mbps which when packed for transmission extends to 1.44Gbps per FPGA. This rate can be easily handled by the Communication IP.

$$Sol\_Rate = (\frac{Sol\_Width}{Vector\_Size}) \times Clk\_Freq \times NB\_Processing\_Units \tag{5.5}$$

5.2.3 Microprocessor system

A MicroBlaze microprocessor is integrated to the architecture for system configuration, monitor, and verification. It is connected to several coprocessors (hardware peripherals) through the PLB bus (Figure 5.1). The MicroBlaze and the PLB run at 100MHz clock frequency which entails changing clock domain any time there is an access to the Multi-Port Memory Controllers (MPMCs), the memory readers and writer, the processing units, and the Communication IP. For the MPMCs, no special circuitry has to be added to safety manage clock domain transitions. This is because the PLB interface at the MPMC's ports can run at $1:1$ or $1:2$ the MPMC clock rate, i.e. 100MHz PLB interface clock frequency and 200MHz MPMC clock frequency. This is only supported by the PLB interface, not by the NPI or others. When accessing the memory readers clock region (200MHz), no additional circuitry is used either, since the Electronic Design Automation (EDA) tools can manage the domain changes. For 100MHz to 125MHz clock region transitions and vice-versa, synchroniser or dual clock FIFOs are used.

The MicroBlaze is generated using Base System Builder (BSB) wizard from Xilinx Embedded Development Kit (EDK) tool. It is optimized for saving logic resources so its footprint is small enough to not considerably affect the rest of the design. As a consequence, the microprocessor doesn't have data or instruction cache, nor Memory and Exception Management Units.

Embedded software

The microprocessor system executes a standalone application. The application runs on each node without the need of a parallel-programming Operating System (OS) neither a Message Passing Interface. It is written in C language and developed on Xilinx's Software Develop-

ment Kit (SDK) platform for the embedded software. Figure 5.6 presents the standalone application flow chart for the system test. First, the initialization process is executed where all of the configurable registers are set according to the application needs. This includes memory segments allocation and binding to all transmission Endpoints (processing units) in the network, as well as base addresses and matrix-vector sizes setting for the memory readers. It also includes routing table population, vector holder module configuration (vector size) and IDs setting for the processing units. After the system initialization, the applications distinguish between a master and a slave node. A slave node waits for the vector to arrive from the Network Interface Controller (NIC) and then verifies for data corruption during the network transfer process. Local matrix-vector computation is done in parallel by the hardware coprocessors. On the other hand, a master node waits for user signal to trigger the timer monitor (for performance evaluation) and the computing system. Afterward, the software checks if all solutions have arrived, and searches for errors between the reference model and the computed results. Finally, a report containing the time elapsed and the number of errors is displayed in a window shell through the UART-USB link.

The matrix, vector and result segments are created beforehand using a MATLAB script. The script uses as input parameters the number of nodes and processing units of the global system and the desired matrix and vector sizes. Then, it generates a source file (.c) with a random vector and several matrix segments for each processing unit. The golden reference model is also generated with the computation results for the verification process. All of these data arrays are included in the main application as external variables. Listing 5.1 shows a code fragment of the generated file by the MATLAB script containing a portion of the matrix data for the FPGA 1. The array `matrix_segment0_a []` is placed in the `matrix_section_channel_a` memory section and aligned 256 bytes.

Listing 5.1 Code fragment of the MATLAB generated file for random matrix-vector data set tests.

```
1 ...
2  #if THIS_FPGA == 1
3  Xuint64 __attribute__((section("matrix_section_channel_a")))
     matrix_segment0_a [] __attribute__((aligned(0x100))) = {
4 {0x403f8dca,0x0d10320d},
5 {0x40451196,0xbeb2edcc},
6 {0x404d6004,0x8fa3ce8f},
7 ...
```

The linker script file is modified manually to create memory segments with 256-bytes boundary alignment as required by the NPI 32-dwords burst transfer constraint. It is also modified

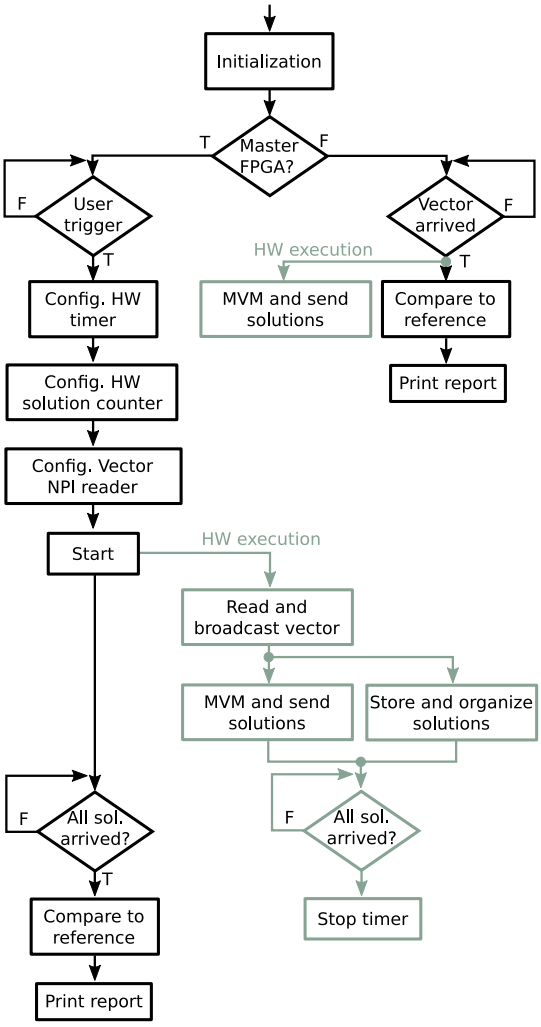

Figure 5.6 Test application flow chart.

to address the application heap to the external memory region and to enlarge its capacity for fitting in the generated arrays. After compilation, the executable file (elf) can be verified to ensure that data regions and alignment were set properly. Listing 5.2 shows a fragment of the MicroBlaze elf file after the software application compilation where it can be seen the matrix, the vector and the golden reference sections in memory. It also shows its base addresses, sizes and data alignments. The `matrix_section_channel_a` and `matrix_section_channel_b` sections are placed into two different off-chip memories to increase memory bandwidth.

Listing 5.2 MicroBlaze elf file fragment after software compilation.

```
1 Sections:
2 Idx Name          Size      VMA       LMA       File off  Algn
3   0 .vectors.reset 00000008  00000000  00000000  000000d4  2**2
```

```
4                   CONTENTS, ALLOC, LOAD, READONLY, CODE
5    1 .vectors.sw_exception 00000008  00000008  00000008  000000dc  2**2
6                   CONTENTS, ALLOC, LOAD, READONLY, CODE
7    2 .vectors.interrupt 00000008  00000010  00000010  000000e4  2**2
8                   CONTENTS, ALLOC, LOAD, READONLY, CODE
9  ...
10  13 matrix_section_channel_a 00249f00  80000000  80000000  00263600  2**8
11                  CONTENTS, ALLOC, LOAD, DATA
12  14 matrix_section_channel_b 00249f00  40000000  40000000  00019700  2**8
13                  CONTENTS, ALLOC, LOAD, DATA
14  15 vector_section 00003f00  80249f00  80249f00  004ad500  2**8
15                  CONTENTS, ALLOC, LOAD, DATA
16  16 golden_section 00004b00  8024de00  8024de00  004b1400  2**8
17                  CONTENTS, ALLOC, LOAD, DATA
18  17 .heap          00501170  80252900  80252900  004b5f00  2**0
19                  ALLOC
20  18 .stack         00002804  000196d4  000196d4  00019620  2**0
21 ...
```

The MATLAB reference result is not exactly the same as the one computed by the BEE3 parallel and distributed system. This is because Xilinx's floating point operators deviate slightly from IEEE-754 Standard to provide a better trade-off of resources against functionality [44]. In other words, the way Intel's Floating-Point Unit (FPU) executes floating-point instructions is different to the way Xilinx's operators compute floating point arithmetic. The main deviation from the IEEE-754 Standard regarding Xilinx's operators is related to the denormalized number handling and the rounding modes, which are not supported and partially supported respectively. In addition, when working with floating-point representation, the simple fact of changing the order of the elements involved in a sum or a multiplication could lead to different results. This is attributed to the rounding-to-the-nearest-representable-number process which is natural in floating-point arithmetic. Even if the matrix-vector traversal exhibits the same order in both platforms, the MATLAB's compiler directly impacts in how the computations are translated to machine code, thus the order may change. The inaccuracy possibly introduced is propagated through several calculations, leading to a small difference at the end of the multiply-accumulate process. Hence, an error gap is accepted during the verification process.

5.2.4 Communication IP

The communication IP (Chapter 4) works as a low latency, high bandwidth Network Interface Controller (NIC) for the coprocessor units that want to communicate over the network. It

allocates seven Transmission and two Reception Sockets to fit the application needs which are assigned to the processing units, the vector holder, and the memory writer module. Round Robin scheduling scheme is used among them. This module offers up to 20Gbps aggregate bandwidth (10Gbps per channel) with a fanout of two. Since some link errors were detected at maximum transceivers speed, the latter was conveniently reduced to achieve virtually no link errors. Thus, a 16Gbps communication channel is used to handle the processing unit's communication needs.

The Communication IP holds a routing table, which is populated by the software and supports broadcast feature for fast vector distribution among the nodes. Data forwarding feature is also supported for hop transitions and broadcast control when packets are moving across the network nodes. In terms of error handling on the proposed architecture, when the Communication IP detects an error signaled by the transceivers or the XAUI cores, the incoming data is ignored. This is then noticed by the software part as a missing element/solution.

5.3 Tightly coupled FPGA cluster

Figure 5.7 shows the computing system described above all replicated across the cluster. Forty-eight dot-product hardware operators are tightly integrated into a large Matrix-Vector Multiplication kernel. The Communication IP enables a fast and direct interconnect network for low latency and high-speed communication between all of the nodes. One of the most interesting properties of this architecture is its flexibility to adapt to almost any engineering and scientific application. The hardware operators can be replaced or adapted for any other functionality by exploiting FPGAs' reconfigurability feature. In a rapidly way, the entire cluster can change for a totally different application. The system depicted in Figure 5.7 offers globally 24GB external memory capacity with 35.6GB/s sustained bandwidth and 128Gbps full-duplex network aggregate bandwidth. Such a powerful system is used hereafter for testing and benchmarking the FPGA cluster.

5.3.1 Programming and diagnostic

Programming the eight-FPGA platform requires an automatic approach to avoid tedious and repetitive procedures. This is achieved with python scripting. There are several operations that must be done for getting the platform ready.

1. *Hardware synthesis and bitstream generation*
   The hardware platform is the same for all the nodes in the system.
2. *Software compilation and executable file (.elf) generation*

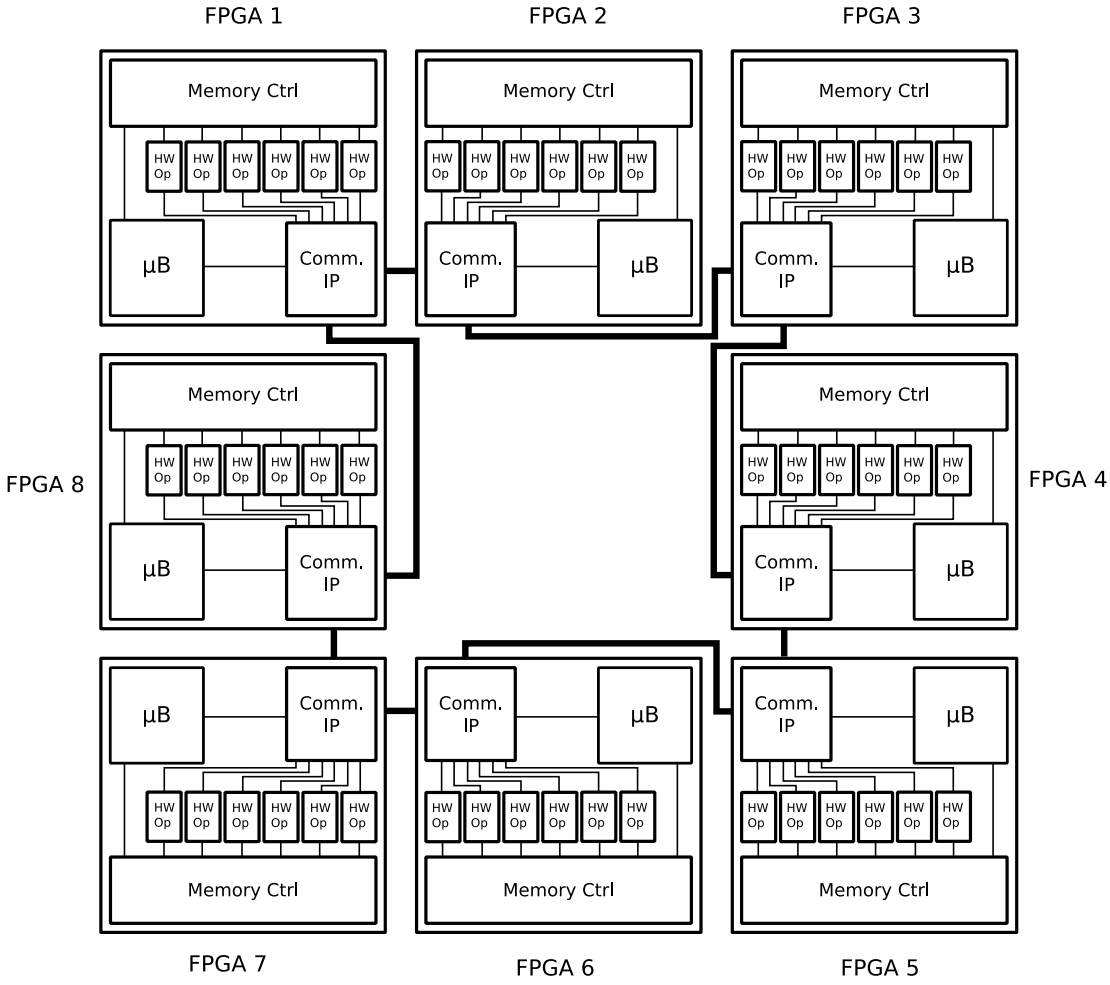

Figure 5.7 Global view of the high-performance computing system.

During the software compilation process, the behavior of each individual node is differentiated from the rest of the system. For that, some conditional description in the application code enables and disables different code regions.

3. *FPGA programming*

   This process has to be done for all of the nodes in the system. The programming order does not affect the system behavior.

4. *Software download and microprocessor restart*

   This process has to be done for all of the nodes in the system. At this point, the software behavior has to be different. Once the executable file is downloaded and the microprocessor is restarted, the nodes start executing the application code. The programming order does affect the system behavior which can easily lead to system failure. If the master node starts operating before all of the slaves are ready, then a

deadlock situation occurs. To avoid programming order problems, the master node always waits for a user trigger action performed over the serial port interface (UART-USB).

The python scripts use the number of nodes as an input parameter to generate an identity header file (.h) unique for each FPGA. This file contains the node's ID which is used during the software compilation time to identify the associated code and data regions in the application code. The scripts automatize the entire platform programming process. Once the platform is all set, the user must trigger the application from the master node.

During the execution process, each node sends monitoring information through the local serial port interface (UART-USB). There are eight window shells on a host computer, displaying run-time nodes states. Final reports after computation completion are also displayed by each node, providing information associated with the number of errors between the reference data and the real data.

Figure 5.8 shows a screenshot of the platform being ready for matrix-vector computing. From the console windows, it is possible to see the software application verifying data alignment on the off-chip memories. Since this process is common for all of the FPGAs, the slave devices have a local vector segment. However, it is only used for verification.

Besides the software monitor mechanism, a hardware monitor approach is also used. Xilinx Chipscope Analyzer displays run-time signals and registers states for the associated hardware probes. These probes are strategically inserted to fully capture important system states. Critical monitor parameters like those related to the performance evaluation are double checked, by software and by hardware.

5.3.2 Performance evaluation and results

To evaluate the performance of the distributed parallel processing system presented in this chapter and to see how a low latency and high-speed inter-node communication channel contributes to the overall system performance, we run a set of matrix-vector tests with different aspect ratios.

The matrix set is tested on two different platforms for comparison purposes.

1. *single CPU*

   An Intel Core i7-4510u running at a base frequency of 2.00Ghz and a turbo frequency of 2.60Ghz with 2 physical cores and 4 Threads [45]. The first level cache (L1) is 32KB per core and the main memory system is 8GB DDR3L SDRAM @1600 MHz.

Figure 5.8 Eight window shells, one per FPGA. Screenshot on platform ready.

The Core i7-4510u processor has a peak memory bandwidth of 25.6GB/s and a peak performance of 32GFLOP/s[1]. Windows 8.1 64-bit operating system is used.

2. *FPGA cluster*

   An eight-FPGA (2 BEE3s) platform hosting Virtex-5 lx155t devices. Due to the aforementioned limitations, the system has 8×3GB DDR2-400 DRAM, 8×4.45GB/s global sustained memory bandwidth, 8×6 processing units (dot-product operators), and 16Gbps full-duplex network interface bandwidth (the Communication IP).

For the Intel Core i7 processor implementation, a MATLAB (R2014b 64-bit) script generates pseudorandom floating point double-precision dense matrices and vectors, and computes the multiplication among them. The Intel Math Kernel Library (MKL) 11.1.1 runs behind MATLAB optimizing computation over the Intel architecture. To measure the time elapsed during the matrix-vector computation, we used the TIC and TOC MATLAB functions, just before and after the multiplication instruction. However, since MATLAB is an interpreter (it interprets the code on the fly rather than compile it), the first time the script is executed,

1. Obtained after running the Intel Optimized LINPACK Benchmark 10.3.4 for Windows [46].

the measured elapsed time is always longer than the followings. For that reason, the first estimation is never considered.

The execution time on a CPU may vary depending on the operating system workload and on the microprocessor temperature. Therefore, to get the best results from it, we run each test on cold and settled conditions, which allows the operating frequency to boost up to its turbo range.

For the eight-FPGA implementation, the matrix is split into 8 blocks which are distributed among the nodes and stored in the off-chip memory. When the system is triggered, the vector is copied and stored locally in each node's on-chip memory. The computation takes place in the processing units. They receive data flows from the on-chip (vector) and off-chip (matrix) memories. As the results are been generated, they are packed and sent through the network to the master node. The master stores the solutions and checks for errors. A hardware timer on the master node is used to estimate the computational performance. The timer is triggered just before the master starts scattering the vector, and is stopped just after all of the solutions were received. The computing power is then calculated as the ratio of the number of floating point operations over the measured elapsed time.

Figure 5.9 presents a screenshot of the platform's window shells after computing completion in a 2400×2000 matrix test. The shells display, in hexadecimal format, the verification process of the broadcasted vector and the multiplication results. A final error report and the computation time are displayed as well at the master node. The Figure also shows a Chipscope console monitoring by hardware the number of solutions received and the number of clock cycles elapsed during the entire process. The hardware computation time differs from the software one in the clock frequency; 125Mhz and 100Mhz respectively. There is a slight difference between the results obtained with the FPGA platform and the reference model computed on MATLAB, which is associated with the deviation of Xilinx's floating point operators from the IEEE-754 Standard and the computation order.

Figure 5.10 shows the performance achieved after running the HPC test on the CPU and FPGA platforms. Our FPGA approach achieved 6.38GFLOPS peak and 5.44GFLOPS sustained computing power over the entire matrix set, against 4.25GFLOPS peak and 3.35GFLOPS sustained on the CPU. The CPU system was not able of computing the largest matrix set because it has not enough memory.

Considering that the maximum computing power after the memory bandwidth limitations is about $8.9 GFLOPS$, the proposed platform went from 50% to 72% computational efficiency including data propagation, distribution, and collection through the ring network. Hence, utilizing the Communication IP as the Network Interface Controller (NIC) when clustering

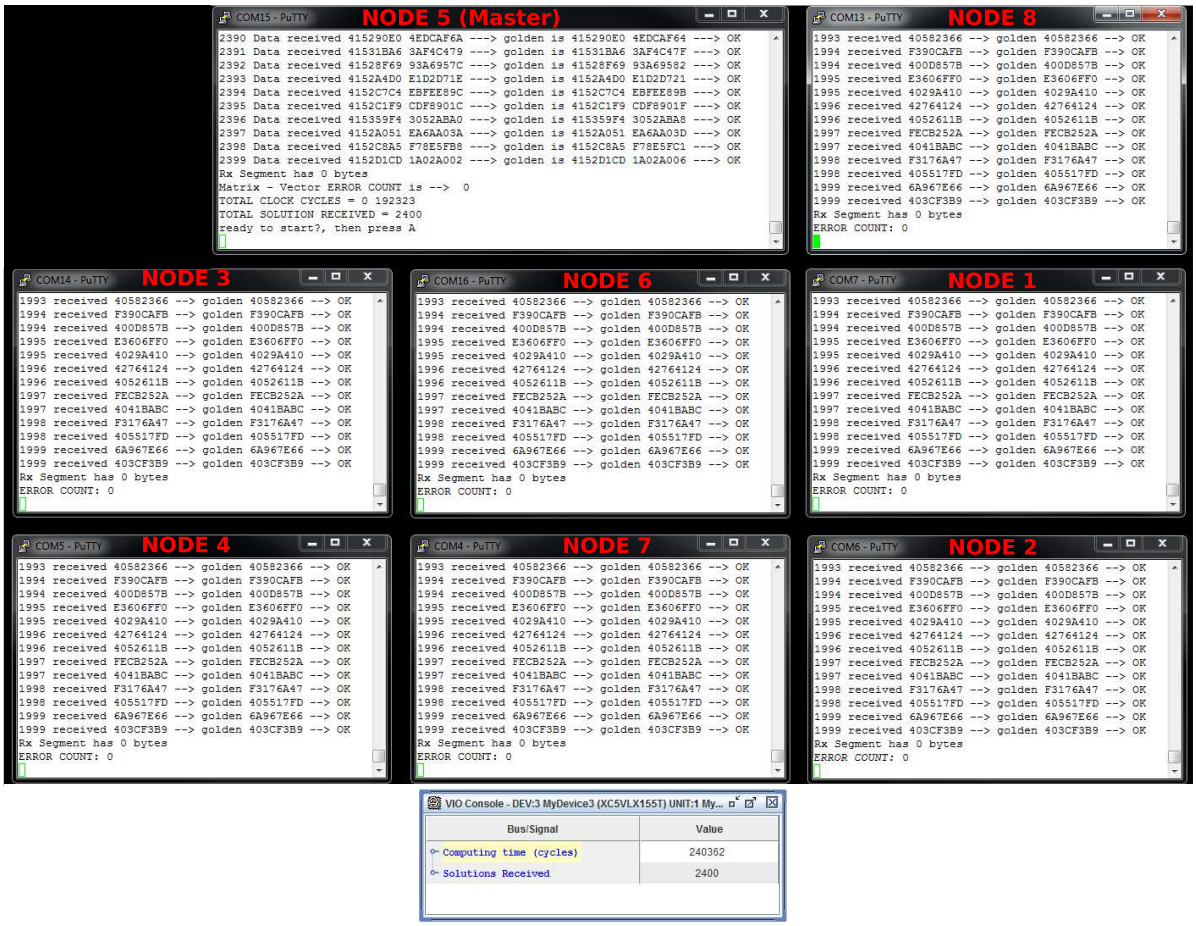

Figure 5.9 Computing completion screenshot.

FPGAs has shown great results by reducing communications overhead. Current channel fanout could be increased in future works for allowing different network topologies with a better diameter which results in a higher overall computational efficiency.

The resources utilization for the HPC test architecture implementation on a Virtex-5 xl155t FPGA is following presented in Table 5.4

Table 5.4 Resource utilization for the HPC test on a Virtex-5 xl155t FPGA.

Resource	LUTs	FFs	Block RAMs	DSP48E
Utilization	25685	30973	175	117
%	26	31	82	91

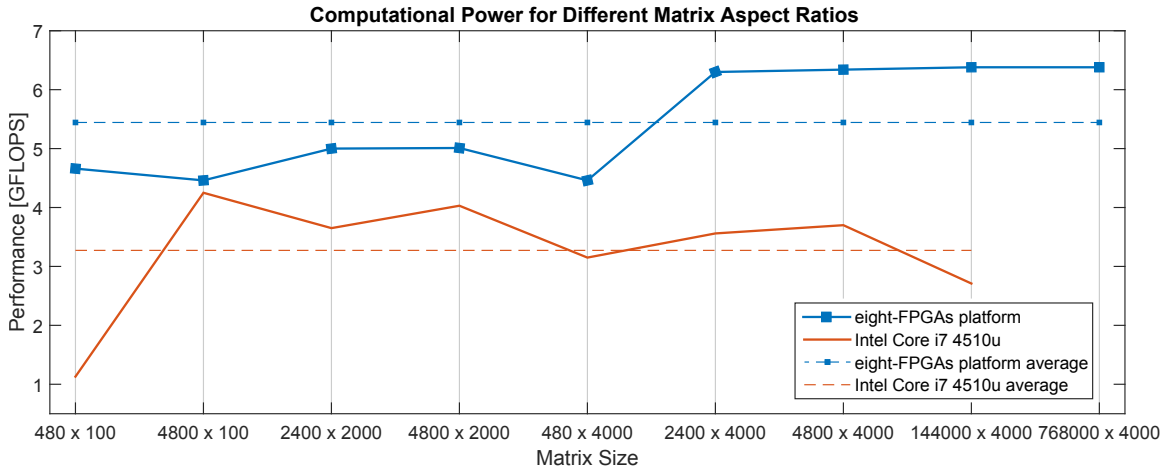

Figure 5.10 Experimental results for performance estimation using dense matrix-vector multiplication on the eight-FPGA Virtex-5 platform and the Intel Core i7-4510u microprocessor.

5.4 Conclusion

We have assembled an eight-FPGA machine hosting 48 distributed coprocessors (MAC operators) for computing Matrix-Vector Multiplication in parallel, and for testing the Communication IP as the Network Interface Controller in an HPC environment. The system has a theoretical computational power of $8.9 GFLOPS$ due to off-chip memory bounds. A set of dense matrices and vectors were used with different aspect ratio for evaluating the platform performance. The experimental results showed 72% computational efficiency including data propagation time over the network. The communication overhead is less than 30% even though the network topology used was one of the largest diameters.

In an HPC system, there are three main ways of increasing performance: increase the number of processing units, improve algorithm and reduce interconnection overhead. We have demonstrated that a better utilization of the interconnection network leads to higher efficiency in terms of the computational power. Consequently, the proposed Communication IP may support FPGA-based HPC clusters.

CHAPTER 6 CONCLUSION

In this work, ultra-low latency and high-speed communication channels for reconfigurable hardware have been presented. Our approach exploits the FPGA Multi-Gigabit Transceiver technology to achieve reliable 8Gbps channel speed. At the Physical layer of the communication stack, four transceivers were bonded per channel using the 10 Gigabits Attachment Unit Interface (XAUI) standard protocol. This provides Ethernet compatibility at the Physical layer which can be eventually used in a 10Gbps Ethernet deployment.

The proposed architecture separates Link and Physical layers by a 10 Gigabit Media Independent Interface (XGMII) standard for easy decoupling when migrating to other FPGA families. Our Link layer implementation proposes a custom simple packet-based protocol for appending information to raw data bits. The protocol fields were selected carefully to provide only the basic functionalities to operate the communication channels on an FPGA-based network. It reached 89% bandwidth efficiency with only 100B payload packets. Such a simple communication stack led us to achieve one of the lowest end-to-end latencies (272ns) of the state-of-the-art. The FPGA resource utilization was kept as low as possible with only 5% footprint and enough room left for user applications.

A direct network is supported by our implementation. A parameterizable number of user Endpoints (coprocessors) may have access to the Physical layer by means of the virtual channels technique. Packet routing and broadcast features were used for Endpoint communication across the network.

Table 6.1 summarizes some of the main results obtained in similar implementations that were found in the literature review. There are a few aspects to consider, regardless the design itself, when comparing to other works. First are the FPGA generation and family. This aspect determines the transceiver and the fabric technologies which in turn affect their operational rates (frequency). Both rates are directly associated with the overall latency of an implementation. The second aspect to consider is the channel's lane combination. Single lane channels do not require channel bonding logic nor deskew, so better latency results may be achieved compared to multi-lane channels. However, single lane channels occupy more resources than its counterpart.

For testing purposes, our IP is used as the Network Interface Controller in an eight-FPGA parallel processing cluster. We have designed a complete High-Performance Computing cluster with features like 35.6GB/s external memory throughput, 8.9GFLOPS computing power, and 128Gbps network aggregate bandwidth. For that, two Multi-Port Memory Controllers,

Table 6.1 The communication IP features and a state-of-the-art comparison.

Implementation	Fanout (Channels)	Bandwidth per Channel	Protocol	Latency	Efficiency peak	Resources	Platform
[15]	8 one-lane	3.2Gbps	Custom over Aurora	800ns	95% [a]		Virtex-4
[17]	1 four-lanes	10Gbps	Internet stack 10GE (XAUI+MAC)	>1μs		45%	Virtex-5
[19]	2 four-lanes	10Gbps	10GE	840ns		18%	Virtex-5
[19]	2 four-lanes	6.24Gbps	Aurora	541ns	86%	10%	Virtex-5
[10, 21]	12 one-lanes	3.125Gbps	Custom	400ns	40%	-	Stratix IV
[10, 21]	12 one-lanes	10Gbps	Custom	200ns	40%	18%	Stratix V
[22]	4 one-lane	1.6Gbps	Custom	830ns	91%	40%	Virtex-5
[30]	4 two-lanes	17Gbps	Custom over Aurora	480ns	85%	4%	Virtex-7
This work	2 four-lanes	8Gbps	Custom over XAUI	272ns	89% [b]	5%	Virtex-5

a. with 16KB payload.
b. with 100B payload.

six floating point double-precision Multiply-Accumulate operators, a MicroBlaze microprocessor, and the proposed Communication IP were integrated into each FPGA. Our FPGA cluster achieved competitive performance and computational efficiency: 6.38GFLOPS peak and 72% respectively, as a result of a low communication overhead among the distributed processing elements and supported by this work.

Table 6.2 compares our experimental results to other Matrix-Vector Multiplication (MVM) implementations running dense matrices on FPGA, CPU, and GPU. The MVM performance is mainly determined by the memory bandwidth and the number of processing elements (PE) in the system. Taking that into consideration, we can see that our implementation achieved competitive results even though the collective data movement times for parallel processing were included in the experiments.

Table 6.2 Computational performance on different MVM implementations with dense matrices results.

Work	Peak Perf. [GFLOPS]	Total PE	Platform	Mem. BW [GB/s]
[35]	3.1	16	single FPGA Virtex-5 lx155t (BEE3)	6.4
[7]	6.4	32	single FPGA Virtex-6 lx240t (ML605)	51.2
[37] [a]	19.2	64	single FPGA Virtex-5 sx95t (ROACH)	35.74
[8]	13.6	64	four FPGAs Virtex-5 lx330 (Convey HC-1)	80
[36]	16.3	64	four FPGAs Virtex-6 lx760 (Convey HC-2ex)	80
This work	6.38	48	eight FPGAs Virtex-5 lx155t (BEE3)	51.2
[8]	23	N/A	GPU Tesla M2090	177.6
[37] [a]	9.3	N/A	GPU GTX 660	144.2
[37] [a]	21.7	N/A	GPU GTX Titan	288.4
[37] [a]	2.5	N/A	Intel Core i7-2600	21
[37] [a]	5.2	N/A	Intel Core i7-4770	25.6
This work	4.25	N/A	Intel Core i7-4510u	25.6

a. single-precision floating point.

6.1 Advancement of knowledge

The standard high-speed communication solutions are inefficient and unnecessary for FPGA-to-FPGA communications. Custom solutions are preferable over standards off-the-shelf approaches [10]. This work has made its contribution in an FPGA-to-FPGA communication IP with high-speed and ultra-low latency features, which makes it suitable for FPGA clustering in the High-Performance Computing fields. A distributed parallel platform is also implemented and the experimental results showed great potentialities for further researches.

6.2 Limits and constraints

Current limitations of the proposed Communication IP are related to the targeting platform. The number of available Multi-Gigabit Transceivers (MGT) limits the fanout toward the network. At the same time, this limits the network topology to just ring or linear array which are not good topologies in terms of the diameter (the longest shortest paths between any two nodes in the network). Since a direct network is used, the longest the diameter, the lower the overall performance of the network. Hence, increasing the IP's fanout could lead to less communication overhead on the HPC cluster and higher computational efficiency.

6.3 Recommendations

Message Passing is the dominant programming model for distributed parallel computers, and Message Passing Interface (MPI) a standard library used for developing and running parallel programs. In order to be able to execute standard parallel programs and performance estimation benchmarks on the FPGA cluster, it is our recommendation to implement and support parallel programming model like MPI. For that, the MicroBlaze soft processor or the PowerPC hard processor should be used as well as the communication channels already proposed in this work. For the implementation, a portable MPI library is recommended. Some modifications have to be done to adapt it to the current platform. MPI support would allow mixed platform parallel execution, i.e. a PC-based cluster with an FPGA-based cluster.

REFERENCES

[1] B. Parhami, *Introduction to parallel processing: algorithms and architectures*. Springer Science & Business Media, 2006.

[2] J. Brandon, "WP458 v2.0: Leveraging ultrascale architecture transceivers for high-speed serial io connectivity bandwidth growth and high-speed serial interfaces," White Paper, Xilinx, pp. 1–24, 2015.

[3] "High Speed Serial," Xilinx. [Online]. Available: https://www.xilinx.com/products/technology/high-speed-serial.html

[4] Xilinx, *DS643 v6.06.a: LogiCORE IP Multi-Port Memory Controller: Product specification*, February 2013.

[5] "Top500 Supercomputer Sites." [Online]. Available: https://www.top500.org

[6] M. Dreschmann, J. Heisswolf, M. Geiger, J. Becker, and M. HauBecker, "A framework for multi-fpga interconnection using multi gigabit transceivers," in *2015 28th Symposium on Integrated Circuits and Systems Design (SBCCI)*, Aug 2015, pp. 1–6.

[7] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal fpga matrix-vector multiplication architecture," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 9–16.

[8] K. Townsend and J. Zambreno, "Reduce, reuse, recycle (r3): A design methodology for sparse matrix vector multiplication on reconfigurable platforms," in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, June 2013, pp. 185–191.

[9] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991611

[10] A. T. Markettos, P. J. Fox, S. W. Moore, and A. W. Moore, "Interconnect for commodity fpga clusters: Standardized or customized?" in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–8.

[11] J. Dong, *Network Dictionary*. Javvin Technologies Inc., 2007.

[12] J. Buford, H. Yu, and E. K. Lua, "Chapter 2 - peer-to-peer concepts," in *P2P Networking and Applications*. Boston: Morgan Kaufmann, 2009, pp. 25 – 44. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123742148000027

[13] O. Mencer, K. H. Tsoi, S. Craimer, T. Todman, W. Luk, M. Y. Wong, and P. H. W. Leong, "Cube: A 512-fpga cluster," in *2009 5th Southern Conference on Programmable Logic (SPL)*, April 2009, pp. 51–57.

[14] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, and P. Beeraka, "Reconfigurable computing cluster (rcc) project: Investigating the feasibility of fpga-based petascale computing," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, April 2007, pp. 127–140.

[15] A. G. Schmidt, W. V. Kritikos, R. R. Sharma, and R. Sass, "Airen: A novel integration of on-chip and off-chip fpga networks," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, April 2009, pp. 271–274.

[16] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Investigation into scaling i/o bound streaming applications productively with an all-fpga cluster," *Parallel Computing*, vol. 38, no. 8, pp. 344 – 364, 2012, {APPLICATION} {ACCELERATORS} {IN} {HPC}. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001712

[17] S. Mühlbach and A. Koch, "A scalable multi-fpga platform for complex networking applications," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2011, pp. 81–84.

[18] U. Langenbach, A. Berthe, B. Traskov, S. Weide, K. Hofmann, and P. Gregorius, "A 10 gbe tcp/ip hardware stack as part of a protocol acceleration platform," in *2013 IEEE Third International Conference on Consumer Electronics ¿¿ Berlin (ICCE-Berlin)*, Sept 2013, pp. 381–384.

[19] T. Bunker and S. Swanson, "Latency-optimized networks for clustering fpgas," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, April 2013, pp. 129–136.

[20] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 133–140.

[21] P. J. Fox, A. T. Markettos, and S. W. Moore, "Reliably prototyping large socs using fpga clusters," in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, May 2014, pp. 1–8.

[22] S. Denholm, K. H. Tsoi, P. Pietzuch, and W. Luk, "Cuscomnet: A customisable network for reconfigurable heterogeneous clusters," in *ASAP 2011 - 22nd IEEE International*

Conference on Application-specific Systems, Architectures and Processors, Sept 2011, pp. 9–16.

[23] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell - a 64 fpga supercomputer," in *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, Aug 2007, pp. 287–294.

[24] M. Nüssle, B. Geib, H. Fröning, and U. Brüning, "An fpga-based custom high performance interconnection network," in *2009 International Conference on Reconfigurable Computing and FPGAs*, Dec 2009, pp. 113–118.

[25] H. Fröning, M. Nüssle, H. Litz, and U. Brüning, "A case for fpga based accelerated communication," in *2010 Ninth International Conference on Networks*, April 2010, pp. 28–33.

[26] M. Nüssle, H. Fröning, S. Kapferer, and U. Brüning, "Accelerate communication, not computation!" in *High-Performance Computing Using FPGAs*. Springer, 2013, pp. 507–542.

[27] R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, P. Paolucci, D. Rossetti, A. Salamon, F. Simula, L. Tosoratto, and P. Vicini, "A 34 gbps data transmission system with fpgas embedded transceivers and qsfp+ modules," in *2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC)*, Oct 2012, pp. 872–876.

[28] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sept 2014.

[29] "Aurora Link-layer Protocol," Xilinx. [Online]. Available: https://www.xilinx.com/products/design_resources/conn_central/grouping/aurora.htm

[30] S. W. Jun, M. Liu, S. Xu, and Arvind, "A transport-layer network for distributed fpga platforms," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–4.

[31] "Xilinx Multi-Node Product Portfolio," Xilinx. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga.html

[32] *DS890 v2.10: UltraScale Architecture and Product Overview*, Xilinx, November 2016.

[33] "Intel FPGA Stratix 10," Intel. [Online]. Available: https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html

[34] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990. [Online]. Available: http://doi.acm.org/10.1145/77626.79170

[35] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE Computer Society Annual Symposium on VLSI*, July 2010, pp. 288–293.

[36] K. R. Townsend, "Computing spmv on fpgas," Ph.D. dissertation, 2016. [Online]. Available: https://search.proquest.com/docview/1798479304?accountid=40695

[37] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554785

[38] Xilinx, *PG053 v10.4: LogiCORE IP XAUI: Product Guide*, July 2012.

[39] Xilinx, *UG627 v12.4: XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices*, December 2010.

[40] *BEE3*, BeeCube. [Online]. Available: http://www.beecube.com/products.html

[41] "Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications," *IEEE Std 802.3-2005 (Revision of IEEE Std 802.3-2002 including all approved amendments)*, pp. 1–2695, Dec 2005.

[42] Xilinx, *UG196 v2.1: Virtex-5 FPGA RocketIO GTP Transceiver*, December 2009.

[43] "AR#24912: 12.4 EDK, MPMC v6.00.a - How do I create a custom NPI core and connect it to MPMC in EDK?" Xilinx, 2016. [Online]. Available: https://www.xilinx.com/support/answers/24912.html

[44] Xilinx, *DS335 v5.0: LogiCORE IP Floating-Point Operator: Product specification*, March 2011.

[45] "Intel® Core™ i7-4510U Processor," Intel. [Online]. Available: https://ark.intel.com/products/81015/Intel-Core-i7-4510U-Processor-4M-Cache-up-to-3_10-GHz

[46] "Intel® Math Kernel Library Benchmarks," Intel. [Online]. Available: https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite

ANNEXE A Modifications to the transceivers' wrapper source file.

Listing A.1 Source code modifications to the `xaui_rocketio_wrapper_tile.v` file after the tuning up process with IBERT tool at 8Gbps speed.

```
1  //_____________________________ Shared Attributes ____________________________
2  //----------------------- Tile and PLL Attributes ----------------------
3  //configuration for 156.125Mhz refclk and 3.125Gbps rate lane
4  //         .CLK25_DIVIDER               (10),
5  //         .CLKINDC_B                   ("TRUE"),
6  //         .OOB_CLK_DIVIDER             (6),
7  //         .OVERSAMPLE_MODE             ("FALSE"),
8  //         .PLL_DIVSEL_FB               (2),
9  //         .PLL_DIVSEL_REF              (1),
10 //         .PLL_TXDIVSEL_COMM_OUT       (1),
11 //         .TX_SYNC_FILTERB             (1),
12 //configuration for 125Mhz refclk and 2.5Gbps rate lane
13           .CLK25_DIVIDER               (5),
14           .CLKINDC_B                   ("TRUE"),
15           .OOB_CLK_DIVIDER             (4),
16           .OVERSAMPLE_MODE             ("FALSE"),
17           .PLL_DIVSEL_FB               (2),
18           .PLL_DIVSEL_REF              (1),
19           .PLL_TXDIVSEL_COMM_OUT       (1),
20           .TX_SYNC_FILTERB             (1),
21 //---------------------------------------------------------------------
22 //               ...
23 //-----------------------Port mapping---------------------------------
24 //----- Receive Ports - RX Driver,OOB signalling,Coupling and Eq.,CDR -
25          .RXCDRRESET0           (RXCDRRESET0_IN),
26          .RXCDRRESET1           (RXCDRRESET1_IN),
27          .RXELECIDLE0           (rxelecidle0_i),
28          .RXELECIDLE1           (rxelecidle1_i),
29          .RXELECIDLERESET0      (tied_to_ground_i),
30          .RXELECIDLERESET1      (tied_to_ground_i),
31 //     .RXENEQB0              (tied_to_vcc_i),
32 //     .RXENEQB1              (tied_to_vcc_i),
33 //     .RXEQMIX0              (tied_to_ground_vec_i[1:0]),
34 //     .RXEQMIX1              (tied_to_ground_vec_i[1:0]),
35 //     .RXEQPOLE0             (tied_to_ground_vec_i[3:0]),
36 //     .RXEQPOLE1             (tied_to_ground_vec_i[3:0]),
37           .RXENEQB0              (tied_to_ground_i),
```

```
38          .RXENEQB1               (tied_to_ground_i),
39          .RXEQMIX0               (2'b11),   //37.5% wideband,
40          .RXEQMIX1               (2'b11),   //62.5% high-pass
41          .RXEQPOLE0              (4'b1100), //+12.5%
42          .RXEQPOLE1              (4'b1100),
43          .RXN0                   (RXN0_IN),
44          .RXN1                   (RXN1_IN),
45          .RXP0                   (RXP0_IN),
46          .RXP1                   (RXP1_IN),
47 //-------------------------------------------------------------------------
48                ...
```

ANNEXE B Native Port Interface source code

Listing B.1 VHDL description of the Native Port Interface controller for 32 double word burst read transfers.

```
1  --  npi_size_i --> NATIVE PORT INTERFACE DATA TRANSFER SIZE ENCODING
2  --  0000       8   BYTES TRANSFERRED (1  BEAT  FOR 64 BIT DATA WIDTH BUS)
3  --  0001       16  BYTES TRANSFERRED (2  BEATs FOR 64 BIT DATA WIDTH BUS)
4  --  0010       32  BYTES TRANSFERRED (4  BEATs FOR 64 BIT DATA WIDTH BUS)
5  --  0011       64  BYTES TRANSFERRED (8  BEATs FOR 64 BIT DATA WIDTH BUS)
6  --  0100       128 BYTES TRANSFERRED (16 BEATs FOR 64 BIT DATA WIDTH BUS)
7  --  0101       256 BYTES TRANSFERRED (32 BEATs FOR 64 BIT DATA WIDTH BUS)
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.NUMERIC_STD.ALL;
11
12 entity NPI_read_interface is
13 generic(
14     C_PI_ADDR_WIDTH     :integer := 32;
15     C_PI_DATA_WIDTH     :integer := 64;
16     C_PI_BE_WIDTH       :integer := 8;
17     C_PI_RDWDADDR_WIDTH :integer := 4
18     );
19   port(
20     -- User interface signals
21     BASE_ADDR_IN: in  std_logic_vector(31 downto 0):= (others => '0');
22     -- IMPORTANT BASEADDR MUST HAVE THE LSB EQUAL TO 0X000 OR 0X100 TO BE
23     -- ALIGNED WITH 256 BYTES BURST TRANSFER
24     NUM_ELEMENTS: in  std_logic_vector(31 downto 0):= (others => '0');
25     READY: in  std_logic:= '0';
26     DONE: out std_logic:= '0';
27     POP: in  std_logic:= '0';
28     EMPTY: out std_logic:= '0';
29     LATENCY: out std_logic_vector(1 downto 0):= (others => '0');
30     DATA_OUT: out std_logic_vector(C_PI_DATA_WIDTH-1 downto 0):= (others
       => '0');
31     -- MPMC Port Interface - Bus is prefixed with NPI_
32     XIL_NPI_Addr: out std_logic_vector(C_PI_ADDR_WIDTH-1 downto 0):= x"
       A0000000";
33     XIL_NPI_AddrReq: out std_logic:= '0';
34     XIL_NPI_AddrAck: in  std_logic:= '0';
35     XIL_NPI_RNW: out std_logic:= '0';
```

```
36      XIL_NPI_Size: out std_logic_vector(3 downto 0):= (others => '0');
37      XIL_NPI_WrFIFO_Data: out std_logic_vector(C_PI_DATA_WIDTH-1 downto 0)
        := (others => '0');
38      XIL_NPI_WrFIFO_BE: out std_logic_vector(C_PI_BE_WIDTH-1 downto 0):= (
        others => '0');
39      XIL_NPI_WrFIFO_Push: out std_logic:= '0';
40      XIL_NPI_RdFIFO_Data: in  std_logic_vector(C_PI_DATA_WIDTH-1 downto 0)
        := (others => '0');
41      XIL_NPI_RdFIFO_Pop: out std_logic:= '0';
42      XIL_NPI_RdFIFO_RdWdAddr: in  std_logic_vector(C_PI_RDWDADDR_WIDTH-1
        downto 0):= (others => '0');
43      XIL_NPI_WrFIFO_Empty: in  std_logic:= '0';
44      XIL_NPI_WrFIFO_AlmostFull: in  std_logic:= '0';
45      XIL_NPI_WrFIFO_Flush: out std_logic:= '0';
46      XIL_NPI_RdFIFO_Empty: in  std_logic:= '0';
47      XIL_NPI_RdFIFO_Flush: out std_logic:= '0';
48      XIL_NPI_RdFIFO_Latency: in  std_logic_vector(1 downto 0):= (others =>
        '0');
49      XIL_NPI_RdModWr: out std_logic:= '0';
50      XIL_NPI_InitDone: in  std_logic:= '0';
51      Clk: in  std_logic:= '0';
52      Rst: in  std_logic:= '0'
53      );
54  end NPI_read_interface;
55
56  architecture Behavioral of NPI_read_interface is
57     constant MAX_READ_REQ_QUEUE: integer := 4;
58     constant BURST_TRANSFER_SIZE: integer := 32;
59     signal total_num_element: std_logic_vector (31 downto 0):= (others
       =>'0');
60     signal num_element_counter: std_logic_vector (31 downto 0):= (others
       =>'0');
61     signal num_ele_NPI_fetched_counter: std_logic_vector (31-5 downto 0):=
       (others=>'0');
62     signal ext_num_ele_NPI_fetched_counter: std_logic_vector (31 downto 0)
       := (others=>'0');
63     signal address_counter: std_logic_vector (23 downto 0):= (others=>'0');
64     signal ext_address_counter: std_logic_vector (31 downto 0):= (others
       =>'0');
65     signal counter_32: std_logic_vector (4 downto 0) := (others=>'0');
66     signal burst_in_process_counter: std_logic_vector (2 downto 0) := (
       others=>'0');
67     -- this is a counter for tracking the number of burst transfer that
68     -- have been buffered it means that the MPMC have acknowledged then
```

```
69    -- and they are holded on the NPI internal fifo the size of the NPI
70    -- fifo is 1024 bytes so it can hold up to 4 - 32 double word burst
71    -- tranfers without risk of overflow.
72    signal next_num_element_counter: std_logic_vector (31 downto 0):= (
      others=>'0');
73    signal next_num_ele_NPI_fetched_counter: std_logic_vector (31-5 downto
      0):= (others=>'0');
74    signal next_address_counter: std_logic_vector (23 downto 0):= (others
      =>'0');
75    signal next_counter_32: std_logic_vector (4 downto 0) := (others=>'0');
76    signal next_burst_in_process_counter: std_logic_vector (2 downto 0) :=
      (others=>'0');
77    signal num_element_load: std_logic:= '0';
78    signal num_element_en: std_logic:= '0';
79    signal num_element_reached: std_logic:= '0';
80    signal num_element_zero: std_logic:= '0';
81    signal num_element_zero_reg: std_logic:= '0';
82    signal num_ele_NPI_fetched_en: std_logic:= '0';
83    signal address_counter_en: std_logic:= '0';
84    signal counter_32_en: std_logic:= '0';
85    signal counter_32_overflow: std_logic:= '0';
86    signal counter_32_overflow_z1: std_logic:= '0';
87    signal burst_in_process_counter_en_up: std_logic:= '0';
88    signal burst_in_process_counter_en_down: std_logic:= '0';
89    signal spop: std_logic:= '0';
90    signal spop_z1: std_logic:= '0';
91    signal spop_z2: std_logic:= '0';
92    signal spop_mux: std_logic:= '0';
93    signal empty_ctrl: std_logic:= '1';
94    signal sXIL_NPI_AddrReq: std_logic:= '0';
95    signal sXIL_NPI_AddrAck_z1: std_logic:= '0';
96    signal ready_z1: std_logic:= '0';
97    type state_type is (IDLE, FETCH, LAST_FETCH, DONE_STATE);
98    signal SM_STATES, NEXT_SM_STATES: state_type;
99  begin
100   process (Clk)
101   begin
102     if Clk'event and Clk = '1' then
103       if Rst = '1' then
104          num_element_counter<= (others=>'0');
105          num_ele_NPI_fetched_counter<= (others=>'0');
106          address_counter<= (others=>'0');
107          burst_in_process_counter<= (others=>'0');
108          counter_32<= (others=>'1');
```

```
109          num_element_zero_reg<= '0';
110          SM_STATES<= IDLE;
111          spop_z1<= '0';
112          spop_z2<= '0';
113          ready_z1<= '0';
114          counter_32_overflow_z1<= '0';
115          sXIL_NPI_AddrAck_z1<= '0';
116          num_element_reached<= '0';
117       else
118          num_element_counter<= next_num_element_counter;
119          num_ele_NPI_fetched_counter<= next_num_ele_NPI_fetched_counter;
120          address_counter<= next_address_counter;
121          burst_in_process_counter<= next_burst_in_process_counter;
122          counter_32<= next_counter_32;
123          num_element_zero_reg<= not num_element_zero;
124          SM_STATES<= NEXT_SM_STATES;
125          spop_z1<= spop;
126          spop_z2<= spop_z1;
127          ready_z1<= READY;
128          counter_32_overflow_z1<= counter_32_overflow;
129          sXIL_NPI_AddrAck_z1<= not XIL_NPI_AddrAck;
130          num_element_reached<= num_element_zero and num_element_zero_reg;
131       end if;
132     end if;
133   end process;
134   process (num_element_load, num_element_en, num_element_counter,
       total_num_element )
135   begin
136     next_num_element_counter <= num_element_counter;
137     if num_element_load = '1' then
138         next_num_element_counter  <= total_num_element;
139     elsif num_element_en = '1' then
140       if (num_element_counter > (num_element_counter'range=>'0')) then
141         next_num_element_counter  <= std_logic_vector (unsigned (
       num_element_counter)-1);
142       end if;
143     end if;
144   end process;
145   process (num_element_load, num_ele_NPI_fetched_en,
       num_ele_NPI_fetched_counter, total_num_element)
146   begin
147     next_num_ele_NPI_fetched_counter <= num_ele_NPI_fetched_counter;
148     if num_element_load = '1' then
```

```
149         next_num_ele_NPI_fetched_counter  <= total_num_element(31 downto
    5);
150     elsif num_ele_NPI_fetched_en = '1' then
151       if (num_ele_NPI_fetched_counter > (num_ele_NPI_fetched_counter'range
    =>'0')) then
152         next_num_ele_NPI_fetched_counter  <= std_logic_vector (unsigned (
    num_ele_NPI_fetched_counter)-1);
153       end if;
154     end if;
155   end process;
156   process (num_element_load, address_counter_en, address_counter,
     BASE_ADDR_IN )
157   begin
158     next_address_counter <= address_counter;
159     if num_element_load = '1' then
160         next_address_counter  <= BASE_ADDR_IN(31 downto 8);
161     elsif address_counter_en = '1' then
162       next_address_counter  <= std_logic_vector (unsigned (address_counter
    ) + 1);
163       -- EACH BURST TRANSFER WILL BE OF 256 BYTES (32 DOUBLE WORDS)
164       -- THE ADDR HAS TO BE ALIGNED SO THE LSB MUST ALWAYs BE 0x000 OR
165       -- 0x100
166     end if;
167   end process;
168   process (burst_in_process_counter_en_up,
     burst_in_process_counter_en_down, burst_in_process_counter)
169   begin
170     next_burst_in_process_counter <= burst_in_process_counter;
171     if burst_in_process_counter_en_up = '1' then
172         next_burst_in_process_counter  <= std_logic_vector (unsigned (
    burst_in_process_counter)+1);
173     elsif burst_in_process_counter_en_down = '1' then
174         if (unsigned (burst_in_process_counter) > 0 ) then
175             next_burst_in_process_counter  <= std_logic_vector (unsigned (
    burst_in_process_counter)-1);
176         end if;
177     end if;
178   end process;
179   process (counter_32_en, counter_32)
180   begin
181     next_counter_32 <= counter_32;
182     if counter_32_en = '1' then
183       next_counter_32  <= std_logic_vector (unsigned (counter_32) - 1);
184     end if;
```

```
185  end process;
186  process (SM_STATES, ready_z1, burst_in_process_counter,
      ext_num_ele_NPI_fetched_counter, XIL_NPI_AddrAck, num_element_reached,
      XIL_NPI_InitDone)
187  begin
188    NEXT_SM_STATES <= SM_STATES;
189    num_element_load <= '0';
190    sXIL_NPI_AddrReq <= '0';
191    case (SM_STATES) is
192      when IDLE  =>
193        if (ready_z1 = '1' and XIL_NPI_InitDone = '1') then
194         NEXT_SM_STATES <= FETCH;
195         num_element_load <= '1';
196        end if;
197      when FETCH =>
198          if (unsigned (burst_in_process_counter) < MAX_READ_REQ_QUEUE)
     then
199          -- if there is at least BURST_TRANSFER_SIZE double word free
200          -- space on the internal NPI read fifo
201             -- read 32 double words
202             sXIL_NPI_AddrReq <= '1';
203             if (unsigned (ext_num_ele_NPI_fetched_counter) >
     BURST_TRANSFER_SIZE ) then
204             -- if there is still data to read
205                NEXT_SM_STATES <= FETCH;
206             else
207                NEXT_SM_STATES <= LAST_FETCH;
208             end if;
209          end if;
210      when LAST_FETCH =>
211            sXIL_NPI_AddrReq <= '1';
212            if (XIL_NPI_AddrAck = '1') then
213            -- if command received
214               NEXT_SM_STATES <= DONE_STATE;
215            end if;
216      when DONE_STATE =>
217            if (num_element_reached = '1') then
218              NEXT_SM_STATES <= IDLE;
219            end if;
220      when others =>
221             NEXT_SM_STATES <= IDLE;
222   end case;
223  end process;
```

```
224   counter_32_overflow <= '1' when counter_32 = (counter_32'range=>'0') and
        counter_32_en = '1' else '0';
225   burst_in_process_counter_en_down <= counter_32_overflow_z1;
226   burst_in_process_counter_en_up <= XIL_NPI_AddrAck;
227   num_ele_NPI_fetched_en <= XIL_NPI_AddrAck;
228   address_counter_en <= XIL_NPI_AddrAck;
229   num_element_zero <= '1' when unsigned(num_element_counter) = 1 and spop
       = '1' else '0';
230   XIL_NPI_RdFIFO_Flush <= num_element_reached;
231   XIL_NPI_Addr <= ext_address_counter;
232   XIL_NPI_Size <= "0101";
233   XIL_NPI_RNW <= '1';
234   total_num_element <= NUM_ELEMENTS;
235   ext_address_counter <= address_counter & "00000000";
236   ext_num_ele_NPI_fetched_counter <= num_ele_NPI_fetched_counter &
       total_num_element(4 downto 0);
237   DATA_OUT <= XIL_NPI_RdFIFO_Data(7 downto 0) & XIL_NPI_RdFIFO_Data(15
       downto 8) & XIL_NPI_RdFIFO_Data(23 downto 16) & XIL_NPI_RdFIFO_Data(31
       downto 24) & XIL_NPI_RdFIFO_Data(39 downto 32) & XIL_NPI_RdFIFO_Data(47
        downto 40) & XIL_NPI_RdFIFO_Data(55 downto 48) & XIL_NPI_RdFIFO_Data
       (63 downto 56);
238   num_element_en <= spop;
239   counter_32_en <= spop;
240   XIL_NPI_RdFIFO_Pop <= spop;
241   DONE <= num_element_reached;
242   spop <= POP;
243   EMPTY <= XIL_NPI_RdFIFO_Empty;
244   LATENCY <= XIL_NPI_RdFIFO_Latency;
245   XIL_NPI_AddrReq <= sXIL_NPI_AddrReq and sXIL_NPI_AddrAck_z1;
246   -- one cloclk cycle rest for arbitation
247 end Behavioral;
```