Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

WP2 New accelerator designs exploiting mixed precision

D2.9 IP for low-latency internode communication links, part 2

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

TEXTAROSSA

Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

Grant Agreement No.: 956831

Deliverable: D2.9 IP for low-latency internode communication links, part 2

Project Start Date: 01/04/2021 **Duration**: 36 months

Coordinator: *AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE - ENEA, Italy.*

Deliverable No	D2.9
WP No:	WP2
WP Leader:	CINI-UNIPI
Due date:	M30
Delivery date:	30/11/2023

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

DOCUMENT SUMMARY INFORMATION

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	01/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP2 New accelerator designs exploiting mixed precision

Deliverable number:	D2.9					
Deliverable title:	IP for low-latency inter-node communication links, part 2					
Due date:	M30					
Actual submission date:	02/12/2023					
Editor:	Francesca Lo Cicero					
Authors:	F. Lo Cicero, A. Lonardo, C. Rossi, P. Vicini					
Work package:	WP2					
Dissemination Level:	Public					
No. pages:	49					
Authorized (date):	30/11/2023					
Responsible person:	Francesca Lo Cicero					
Status:	Plan	Draft	Working	Final	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	2023-11-03	Francesca Lo Cicero	Draft structure
0.2	2023-11-03	Cristian Rossi	Performance Tests
0.3	2023-11-09	Alessandro Lonardo	Finalization of draft for internal rev.
0.4	2023-11-10	Piero Vicini	Internal review.
0.5	2023-11-29	Francesca Lo Cicero	Changes suggested by internal review.
0.6	2023-11-30	Alessandro Lonardo	

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Cosimo Gianfreda	2023-11-22
	Ariel Oleksiak	
Checked by Task Leader	Francesca Lo Cicero	2023-11-30
Checked by WP Leader	Sergio Saponara	2023-11-30
Checked by Project Coordinator	Massimo Celino	2023-12-01

COPYRIGHT

ACKNOWLEDGEMENTS

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no 956831. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Germany, France, Spain, Poland.

Please see http://textarossa.eu for more information on the TEXTAROSSA project.

The partners in the project are AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE (ENEA), FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V. (FHG), CONSORZIO INTERUNIVERSITARIO NAZIONALE PER L'INFORMATICA (CINI), INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), BULL SAS (BULL), E4 COMPUTER ENGINEERING SPA (E4), BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK (PSNC), ISTITUTO NAZIONALE DI FISICA NUCLEARE (INFN), CONSIGLIO NAZIONALE DELLE RICERCHE (CNR), IN QUATTRO SRL (in4). Linked third parties of CINI are POLITECNICO DI MILANO (CINI-POLIMI), Università di Torino (CINI-UNITO) and Università di Pisa (CINI-UNIPI); linked third party of INRIA is Université de Bordeaux; in-kind third party of ENEA is Consorzio CINECA (CINECA); in-kind third party of BSC is Universitat Politècnica de Catalunya (UPC).

The content of this document is the result of extensive discussions within the TEXTAROSSA © Consortium as a whole.

DISCLAIMER

Table of contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
IP	Intellectual Property
FPGA	Field Programmable Gate Array
HLS	High-Level Synthesis
ASIC	Application Specific Integrated Circuit
VCT	Virtual Cut-Through
DOR	Dimension Order Routing
FSM	Finite State Machine
API	Application Programming Interface
BRAM	Block Random Access Memory
DDR (SDRAM)	Double Data Rate (Synchronous Dynamic Random Access Memory)
UDP	User Datagram Protocol
ARP	Address Resolution Protocol
LFSR	Linear Feedback Shift Register
NIC	Network Interface Card

Executive Summary

This document reports on the activities done by TEXTAROSSA partner INFN with reference to the design of the internode Communication IP in WP2.

The INFN Communication IP, developed in VHDL, allows data transfers between processing tasks hosted in the same node (intra-node communications) or in different nodes (inter-node communications), implementing a direct network for FPGA accelerators and enabling the distributed implementation of streaming applications in the APEIRON framework.

The Communication IP was implemented as a Xilinx Vitis RTL kernel that can be automatically integrated with HLS computing kernels by the APEIRON framework to generate the design to be deployed on a multi-FPGA system.

Major improvements to the preliminary release of the IP have been implemented in the design to boost performance and to add functionalities:

- A 256-bit internal data path version of the IP has been developed besides the 128-bit one.
- The clock frequency of the internal logic has been increased from 100 MHz to 150 MHz (and 200 MHz for the 256-bit version).
- The maximum number of serial lanes for the inter-node channels has been increased from 2 to 4.
- The maximum number of intra-node ports has been increased from 2 to 4.
- One network ports of the card can be configured to work as 10G/25G Ethernet port supporting UDP/IP transport layer offloading.

We performed tests on two/four Xilinx Alveo U200 cards and on two Xilinx Alveo U280 cards connected by QSFP+ cables in a ring topology configuration, measuring the performance in terms of end-to-end latency and one-way bandwidth of the Communication IP.

This document is part of deliverable D2.9 along with the IP project database synthesizable both on the Xilinx Alveo U200 and U280 platforms and is publicly available on the deliverable section of the TEXTAROSSA project website (https://textarossa.eu/dissemination/deliverables/).

The synthesizable IP project database is also available in the APEIRON framework git repository (https://github.com/APE-group/APEIRON).

1 Introduction

The INFN Communication IP implements a direct network for FPGA accelerators, allowing low-latency data transfer between processing tasks deployed on the same FPGA (intra-node communication) and on different FPGAs (inter-node communication) and enabling the distributed implementation of real-time dataflow applications in the APEIRON framework.

This document describes the final version of the Communication IP in detail and shows data for its synthesis on the two reference platforms (Xilinx Alveo U200 and U280), along with results of tests developed to validate the newly introduced features of the design and assess its current performance.

Section 1 shows how our IP achieves TextaRossa project's objectives and compares its performance with some solution proposed for multi-FPGAs clusters.

Section 2 introduces the Communication IP architecture, describing in detail the Ethernet port added in the new release, with simulation and implementation results.

Section 3 shows results of latency and bandwidth tests performed connecting two boards (Alveo U200 or U280) and improving internal logic clock (from 100 MHZ to 150 MHz), datapath (from 128 to 256 bit) and number of transceiver's lanes (from 2 to 4).

Section 4 highlights the Communication IP best case implementation (256 bit @ 200 MHz).

Section 5 reports on the FPGA resource usage, for both the Alveo U200 and U280 platforms, of the performance test design integrating two different configurations of the Communication IP (2 serial lanes per inter-node port/128-bit internal datapath and 4 serial lanes per inter-node port/256-bit internal datapath).

Finally, section 6 concludes the report.

1.1 Relationship with project objectives

The Communication IP is the key component enabling the deployment of real-time scalable dataflow applications on a multi-FPGAs system via the APEIRON streaming programming model inspired by Kahn processing networks. Starting from a simple configuration file, the APEIRON framework creates all the files required for the FPGA bitstream generation linking the Communication IP and the application's computational HLS kernels, unburdening the HLS developers from the task of writing a top-level design. Our IP design idea was motivated by the following considerations:

1. The direct communication between computing tasks deployed on FPGAs avoids the involvement of the host CPUs and system bus resources in the data transfers, improving the energy efficiency of the execution platform.
2. Bypassing the intervention of the host network stack, communication latency is reduced while bandwidth for small massages is increased.
3. Since communication operations are implemented on a completely "hardware" path, deterministic latency is achieved, in accordance with the real-time requirements.

These considerations are strictly related to the TEXTAROSSA project objectives:

- **Objective 1 - Energy efficiency**. APEIRON addresses this objective enabling the complete offload of the streaming processing to FPGA devices [Qasaimeh2019, Nguyen2020, Goz2020].

Furthermore, avoiding the involvement of the CPUs and system bus resources in data transfers improves the energy efficiency of the multi-FPGA execution platform.

- **Objective 2 - Sustained application performance**. The sustained applications' performance of distributed streaming applications, such as the RAIDER use case, are strongly affected by the performance of the network system. Implementing a direct FPGA to FPGA interconnect and bypassing the host network stack, allows to keep the communication latency in the sub-microsecond range and to increase the bandwidth for small messages.
- **Objective 4** - Seamless integration of reconfigurable accelerators. The APEIRON framework leverages the Vitis HLS workflow, extending it to a multi-FPGA execution platform through a lightweight communication library (HAPECOM) at programming level, and through a simple configuration system for the deployment of the distributed application to the multi-FPGA execution platform.
- **Objective 5** - Development of new IPs. The INFN Communication IP is the key enabling technology behind the APEIRON framework, allowing direct low-latency intra/inter FPGA communications between HLS kernels.

The objectives are also related to the strategic goals of the project:

- **Strategic Goal #2**: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The APEIRON framework software is developed following the open-source model and is freely available in its GitHub repository (https://github.com/APE-group/APEIRON).
- **Strategic Goal #3**: Opening of new usage domains. The APEIRON framework aims at offering hardware and software support for running real-time dataflow applications on a network of interconnected FPGAs, leveraging on the Vitis HLS tool. We believe that it has the potential to ease the development and to support the efficient execution of a wide class of applications suited to be executed on a multi-FPGA platform, such as but not limited to real-time HPDA ones.

1.2 Comparison with the state-of-the-art

As of today, FPGA represents one of the main accelerator architectures for HPC applications. In addition to this, this type of accelerator is well suited to develop customized algorithms, combining the processing capability of an Application Specific Integrated Circuit (ASIC) with the reconfigurability feature characterizing this kind of device.
In modern development, and in dedicated networks, multiple FPGAs clusters are emplaced to map large HPC kernels by exploiting the low-latency communication capability of these accelerators. However, despite FPGAs high-speed transceiver links, a certain network flexibility with very large clusters could be required to map applications' workloads and to strategically maximize resource utilization and performance. In this direction, many solutions of scalable switched FPGA cluster have been developed, where, for example, the transceiver links are physically connected to ports of high-speed Ethernet switches in an indirect network setup implementing FPGAs as Network Interface Cards (NICs), as for example in the Corundum [1] open-source network interface, in the Virtual Circuit-Switching Network (VCSN) [2], and in the EasyNet [3] open source networking stack.

Corundum is an open-source, high-performance FPGA-based NIC and platform for in-network compute. Features include a high performance datapath, 10G/25G/100G Ethernet, PCI express gen 3, a custom, high performance, tightly-integrated PCIe DMA engine, many (1000+) transmit, receive, completion, and event queues, scatter/gather DMA, MSI interrupts, multiple interfaces, multiple ports per interface, per-port transmit scheduling including high precision TDMA, flow hashing, RSS, checksum offloading, and native IEEE 1588 PTP timestamping. A Linux driver is included that integrates with the Linux networking stack. Development and debugging are facilitated by an extensive simulation framework that covers the entire

system from a simulation model of the driver and PCI express interface on one side to the Ethernet interfaces on the other side. While being a very advanced NIC design, Corundum is a NIC for PCI express endpoints that includes many features that are not needed or are redundant for our reference application scenario of scalable real-time dataflow processing on FPGA.

VCSN is a design of a FPGA-based PCI express NIC targeted at achieving a higher flexibility and scalability in HPC systems compared to those obtained using direct networks, like the one implemented by our Communication IP, while maintaining the same level of performance in terms of latency and bandwidth. While the project demonstrates that this objective can be achieved to a certain extent, it also shows that this is done at the cost of additional dedicated resources in the FPGA design. So, also stripping away the features dedicated to PCI express interfacing that are redundant in our context, the additional cost in terms of FPGA resources, that would limit those dedicated to computing HLS kernels, and in terms of additional hardware for external switches make this approach not effective for our design targets.

EasyNet is a design aiming at reducing the programming effort for FPGA applications on distributed systems. In this system, a 100 Gbps open-source TCP/IP stack is integrated into Xilinx Vitis framework to enable HLS network programming. The network stack instantiation is hidden for the user, so as in the original design flow of a Vitis application, and a set of MPI-like communication primitives and collective operation have been developed to hide the interaction and control management within the network layer and to be easily invoked from an HLS C library.
Corundum and EasyNet are both FPGA-based designs of indirect networks using a 100 Gbps TCP/IP stack, but they present a difference in what concern their implementation: in fact, Corundum FPGA NIC Verilog HDL design doesn't allow the integration of a user computing kernel, while EasyNet design, whose building blocks (CMAC and Network kernels) are developed in Xilinx Vitis framework, allows implementing a HLS user kernel connected to the Network kernel managing the intercommunication between FPGAs. Connecting HLS kernels through the network via communication primitives callable as HLS library functions, EasyNet is similar to the APEIRON framework. Comparing performance, with our Communication IP we assessed a round trip time (RTT) of 1.5 us measured as a ping-pong benchmark of 64 Bytes between two FPGAs, EasyNet report a value of 4.3 us in the same conditions.
In our reference application scenario of real-time dataflow distributed applications on FPGA, latency for small messages (<= 4kB) is a critical feature. A comparison of direct and indirect networks for FPGA clusters [4] shows that for small messages a higher bandwidth and a lower latency are obtained using the direct network, confirming our architectural choice.

To conclude this section, we report on a recent proceedings paper [5] describing a very low-latency multi-FPGA interconnection framework aimed at distributed processing applications. The proposed solution is claimed to allow efficient communication between different processing elements distributed among the FPGAs. To evaluate it, authors built a multi-FPGA system composed of five Zynq ZC706 FPGA boards capable of hosting a diverse number of coprocessors distributed over our custom network. To handle high-speed transceivers, they incorporated the LogiCORE™ IP Aurora 8b1Ob core into the physical layer working at a clock frequency of 250 MHz. With an aggregate bandwidth of up to 25 Gbps per FPGA board, the authors declare the interconnection framework reaches a latency of only 200.36 ns, one of the lowest reported in Electronics Engineering literature and corresponding to roughly one third of what we measure. Although our latency measurement also takes into account the reading of message content from BRAM memory at source node and the writing of the message content in destination node, this is not enough to justify this huge difference. Unfortunately, the authors do not provide a way to reproduce their results, but these results stimulate us to improve further the performance of our Communication IP in terms of latency.

2 Basic design

The Communication IP allows data transfers between processing tasks hosted in the same node (intra-node communications) or in different nodes (inter-node communications), as shown in Figure 2-1.
In the context of the APEIRON framework, processing tasks are implemented by HLS kernels with Xilinx

Vitis. The details of the interface between HLS kernels – the endpoints of the communication – and the Communication IP are described in Section 2.1.

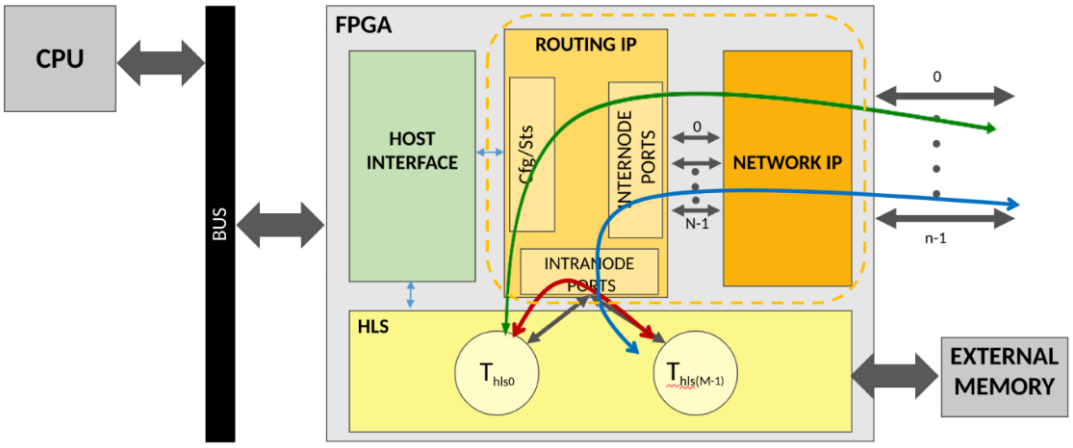

Figure 2-1: Example of intra-node (in red) and inter-node (in blue/green) data transfers between tasks.

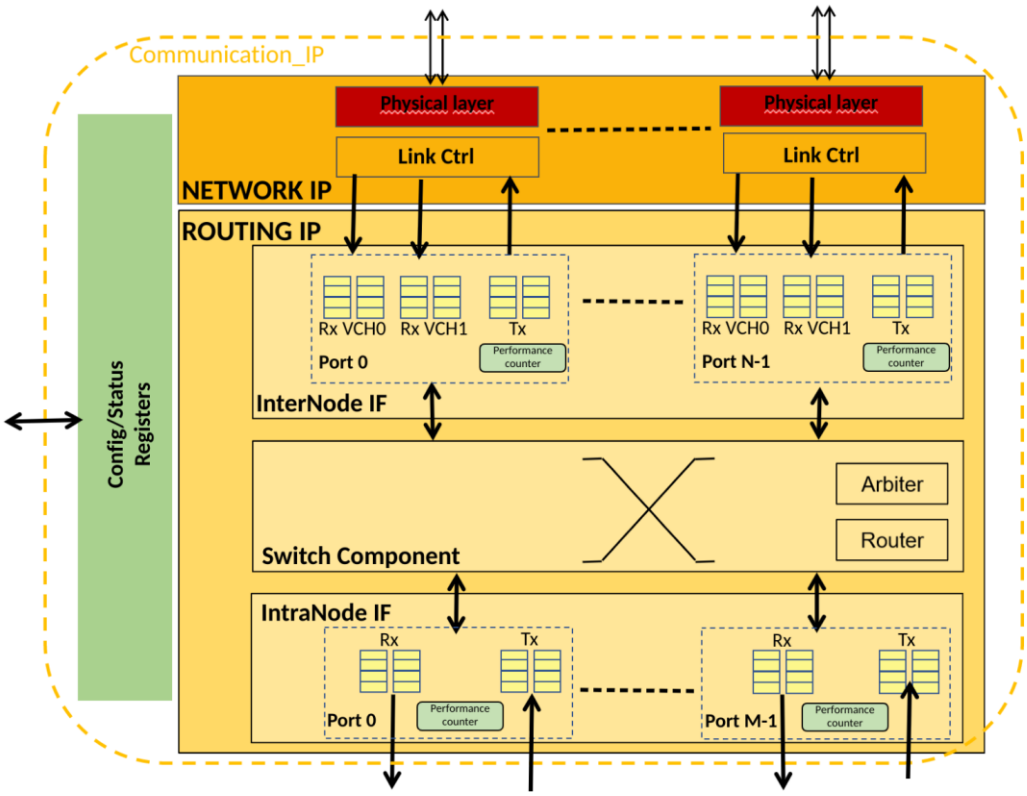

Figure 2-2: Architectural partitioning of the Communication IP

The hardware block structure can be split into a **Network_IP** and a **Routing_IP** (Figure 2-22).

The **Routing_IP** defines the switching technique and routing algorithm; its main components are the Switch_component Block, the Configuration/Status Registers and the InterNode and IntraNode Interfaces.

The Switch component dynamically interconnects all ports of the IP, implementing a communication channel between source and destination ports.

Dynamic links are managed by routing logic together with arbitration logic: the Router configures the proper path across the switch while the Arbiter is in charge of solving contentions between packets requiring the same port.

For inter-node communications, the routing policy applied is the dimension-order (DOR) one: each FPGA is uniquely identified by its coordinates in a N dimensional torus, the DOR consists in reducing the offset along one dimension to zero before considering the offset in the next dimension in anti-lexicographic order.

The employed switching technique — i.e., when and how messages are transferred — is Virtual Cut-Through (VCT) [6]: the router starts forwarding the packet as soon as the algorithm has picked a direction and the buffer used to store the packet has enough space. The deadlock-avoidance of DOR routing is guaranteed by the implementation of two virtual channels for each physical channel (with no fault-tolerance guaranteed) [7].

The transmission is packet-based, meaning that the Communication IP sends, receives and routes packets with a header (Figure 23), a variable size payload and a footer.

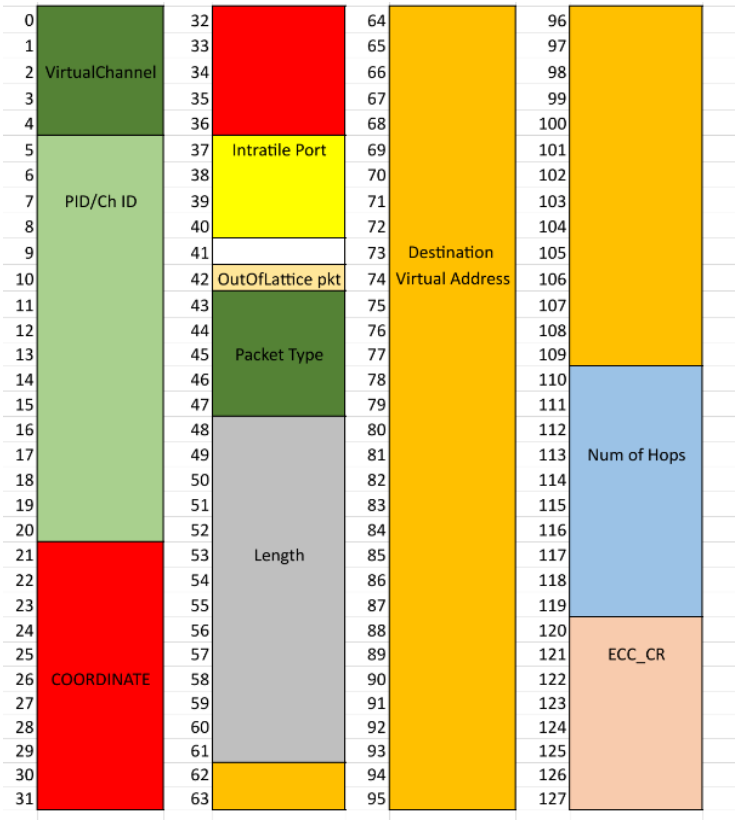

Figure 2-3: TEXTAROSSA packet's header format

The Communication IP exposes two sets of interfaces, i.e., IntraNode and InterNode IF; the number of ports within these interfaces (M and N) can be customized at design time. The IntraNode IF manages data flow to (RX) and from (TX) local tasks; each port consists of two FIFOs for

each direction, so that header/footer and data use a dedicated FIFO, and a **Performance_counter** block (containing an **internal_generator** block to generate packets and an i**nternal_consumer** block for performances monitoring and analysis). The InterNode IF, with the Network_IP block, oversees managing data flow over the serial links between FPGAs.

In the final release of the Communication_IP, the number of IntraNode ports can be customized at design time between 1 and 4, while InterNode ports are set to 2 (according to TEXTAROSSA target devices - Xilinx Alveo U200 and U280).

In the **Network_IP**, the physical layer blocks define the data encoding scheme for the serialization of the messages over the cable and shape the network topology. They provide point-to-point bidirectional, full-duplex communication channels of each node with its neighbors along the available directions.

Link_Ctrl blocks instead establish the logical link between nodes and guarantee reliable communication, eventually performing error detection and correction.

To transfer data between each node with its neighbors we used Xilinx Aurora 64B/66B cores for the serialization of the messages over the cable, and INFN APElink IP [8] to guarantee reliable communication, enabling error detection and correction for critical protocol sections.

In the first release of the Communication_IP we implemented Aurora-based data link layer with 2 lanes with the Routing_IP having a 128-bit internal datapath width.

In order to support applications requiring higher network performance, in the final release of the IP we also support a configuration using Aurora transceivers with 4 bonded lanes associated with a Routing IP datapath width of 256-bit.

For the same reason we increased the frequency (from 100 to 150/200 MHz) of the external RTL kernel clock (ap_clk), used for clocking the internal logic.

2.1 Integration of the Communication IP in the APEIRON Framework

The APEIRON framework developed by the INFN APE Lab [9] aims at offering hardware and software support for running real-time dataflow applications on a network of interconnected FPGAs. The main motivation for the design and development of the APEIRON framework is that the currently available HLS tools do not natively support the development and deployment of applications over multiple FPGA devices, which severely chokes the scalability of problems that this approach could tackle. To overcome this limitation, we envisioned APEIRON as an extension of the Xilinx Vitis framework able to support a network of FPGA devices interconnected by a low-latency direct network as the reference execution platform. Developers can define scalable applications, using a streaming programming model inspired by Kahn Process Networks, that can be efficiently deployed on a multi-FPGAs system.

In this scenario the communication IP must guarantee low-latency communication between processing tasks deployed on FPGAs, even if they are hosted on different computing nodes. Thanks to the use of HLS tools in the workflow, processing tasks are described in C++ as HLS kernels, while communication between tasks is expressed through a lightweight C++ API based on non-blocking *send()* and blocking *receive()* operations:

```
size_t send (msg, size, dest_node, task_id, ch_id);
size_t receive (ch_id, recv_buf);
```

Where:

- `msg` is the message to be sent and `size` is its size in Bytes;
- `dest_node` is the n-Dim coordinate of the destination node (FPGA) in the n-Dim torus network;
- `task_id` is the local-to-node receiving task (kernel) identifier (0-3);
- `ch_id` is the local-to-task receiving FIFO (channel) identifier (0-127).
- `recv_buf` is the receive buffer of the destination HLS kernel.

This simple API allows the HLS developer to perform communications between kernels, either deployed on the same FPGA (intra-node communication) or on different FPGAs (inter-node communication) without knowing the details of the underlying network stack. The software communication Library leverages AXI4-Stream (https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI4-Stream-Interfaces-with-Side-Channels) Side-Channels to encode all the information needed to forge the packet header.

Two APEIRON HLS IPs defined in the software communication library manage the adaptation toward/from IntraNode ports of the Routing IP: they are AGGREGATOR and DISPATCHER, as shown in Figure 24. The DISPATCHER receives incoming packets from the Routing IP and forwards them to the right input channel, according to the relevant fields of the header. The AGGREGATOR receives outgoing packets from the task and forges the packet header, then filling the header/data FIFOs of the Routing IP IntraNode port.

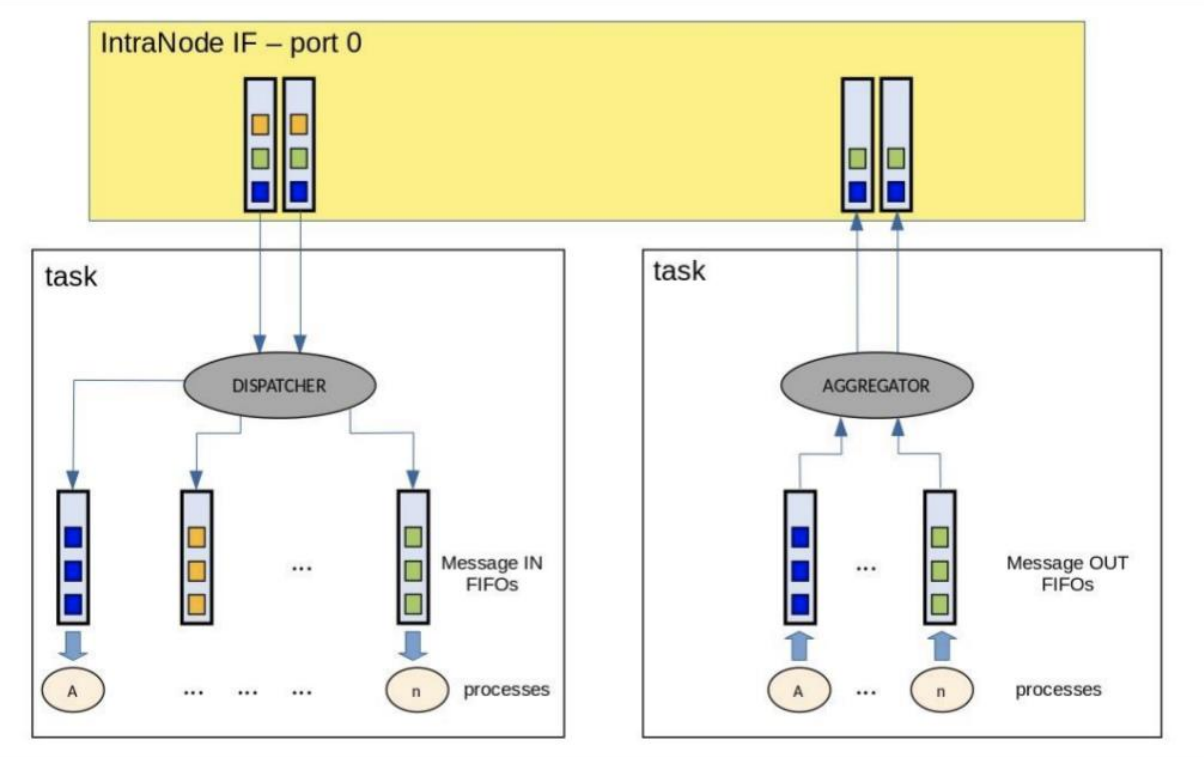

Figure 2-4: Interface between IntraNode Port 0 and the attached HLS computing Task mediated by the AGGREGATOR and DISPATCHER components.

For more details on the APEIRON framework refer to deliverable D4.7 – HLS Flow.

2.2 Ethernet port

In the final release of the Communication_IP, we also implemented a port using the Xilinx® 10G/25G High Speed Ethernet Subsystem, which implements the 25G Ethernet Media Access Controller (MAC) with a Physical Coding Sublayer (PCS). The main purpose of this new feature is to include a standard network port in the Communication IP design to be used as I/O channel by APEIRON dataflow applications.

The architecture of this additional port is depicted in Figure 25.

This port, as well as each port of the Communication_IP, consists of two AXI-FIFOs for each direction (TX/RX), so that header/footer and data use different FIFO.

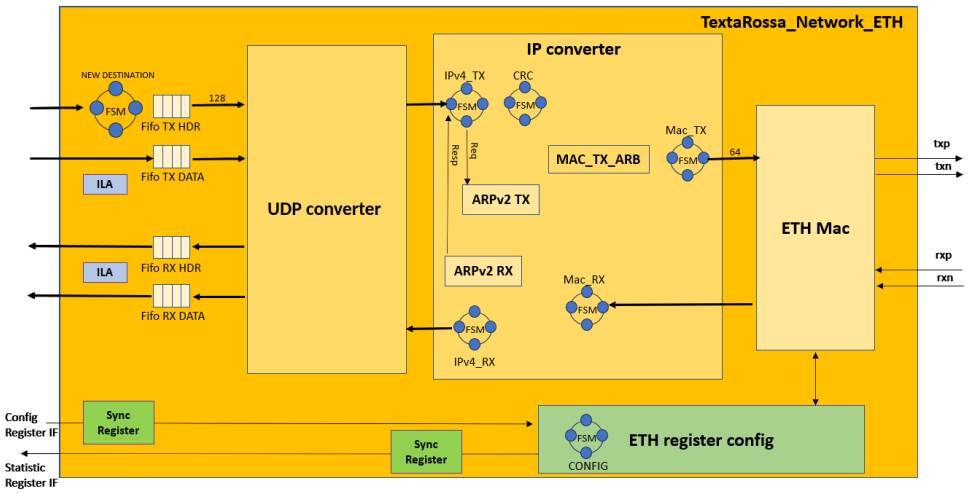

Figure 2-5: Ethernet port architecture

The end-to-end communication can be organized in four abstraction layers: data are encapsulated at each level as shown in Figure 26.

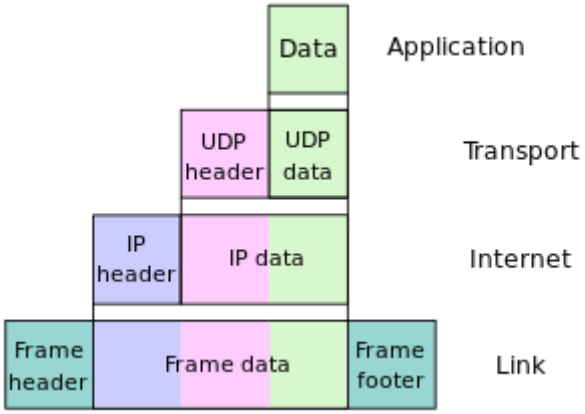

Figure 2-6: Encapsulation of data descending through abstract layers.

The transport layer performs host-to-host communication on either the local network or remote network.

The User Datagram Protocol (UDP) is a simple message-oriented transport layer protocol, which uses a connectionless communication. It also provides a checksum for data integrity and port numbers (source/destination) to identify the process.

Figure 27 shows its header format.

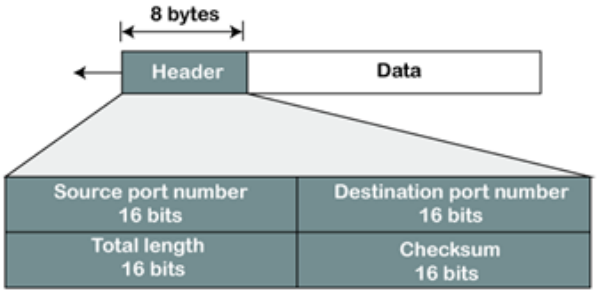

Figure 2-7: UDP header format

The **UDP converter** block is in charge of encapsulating transmitted TEXTAROSSA packets in UDP packets (on TX side) and to extract TEXTAROSSA packets from received UDP payload (on RX side).

On the TX side, the UDP packets are converted into IPv4 packets by the IPV4_TX state machine in the IP converter block; an IPv4 packet consists of a header (shown in Figure 28) and data.

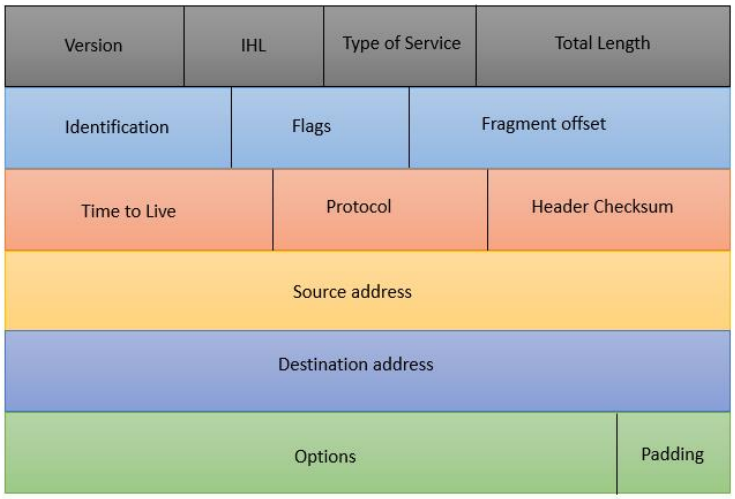

Figure 2-8: IPV4 header format

Version (4 bit): For IPv4, this is always equal to 0x4.

Internet Header Length (IHL) (4 bit): The IPv4 header is variable in size due to the optional 14th field (options). The HL field contains the size of the IPv4 header, specifying the number of 32-bit words in the header (in our implementation 0x5).

Type of Service (TOS) (1 Byte): this field has various purposes, but in our implementation, it is ignored (0x00)

Total Length (2 Byte): this field defines the entire packet size in bytes, including header and data. The minimum size is 20 bytes (header without data), and the maximum is 65,535 bytes.

Identification (2 Byte): This field is primarily used for uniquely identifying the group of fragments of a single IP datagram.

Fragment flags (3 bit): These bits are used to control or identify fragments:

- bit 0: Reserved; must be zero
- bit 1: Don't Fragment (DF)
- bit 2: More Fragments (MF)

Fragment offset (13 bit): This field specifies the offset of a particular fragment relative to the beginning of the original unfragmented IP datagram. Fragments are specified in units of 8 bytes. Fragmentation is not handled in this Communication_IP release (fragment fields are ignored).

Time to live (TTL) (1 Byte): This field limits a datagram's lifetime to prevent network failure in the event of a routing loop. It is specified in seconds (in our implementation is set to 0x80).

Protocol (1 Byte): This field defines the protocol used in the data portion of the IP datagram (0x11 corresponds to UDP).

Header checksum (2 Byte): This field is used to detect corruption in the header. When a packet arrives at a router, the router calculates the checksum of the header and compares it to the checksum field. If the values do not match, the router discards the packet. Errors in the data field must be handled by the encapsulated protocol. UDP has separate checksums that apply to their data.

Source address (4 Byte): IPV4 address of the sender of the packet.

Destination address (4 Byte): IPV4 address of the receiver of the packet.

Options: Not used in our implementation

The **IPV4_TX** FSM (Figure 29) first checks the length of UDP received packet (it has to be less of 1480, otherwise an error arises), then it examines the destination IP address: if it is not equal to 0xFFFFFFFF (BROADCAST ADDRESS) it needs to request the mac address to the **ARPv2 TX** block and to wait for ARP[1] response (WAIT MAC state).

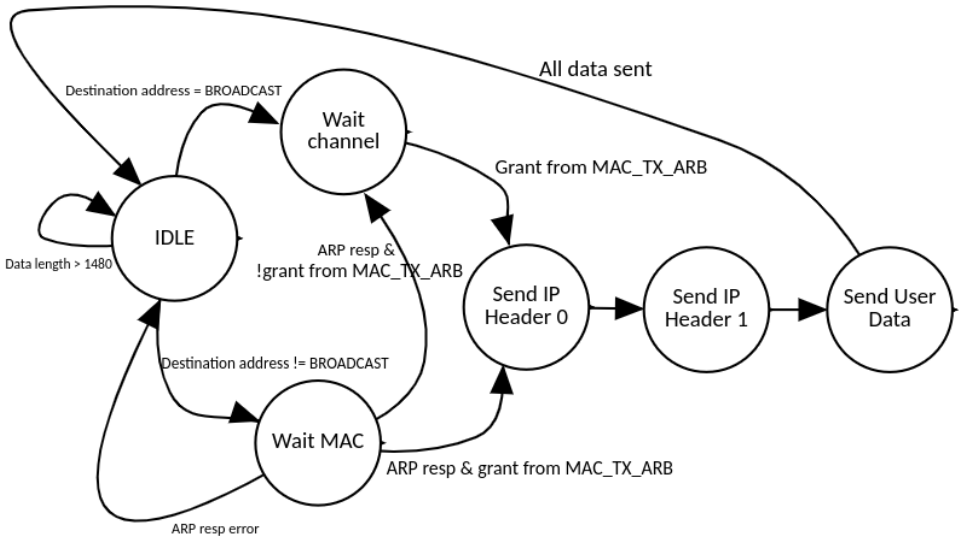

Figure 2-9: IPV4 TX FSM

After grant received from **MAC_TX_ARB** block (in charge of solving contentions between packets requiring ETH port – I.e. IPV4_TX or ARPV2), **IPV4_TX** FSM generates IP packet and sends it to **MAC_TX** FSM. In parallel, the CRC FSM computes CRC of the IPV4 header.

In case of BROADCAST packets, we do not need to look up the MAC address, so grant from **MAC_TX_ARB** is enough.

The **ARPv2 TX** block responds to ARP requests using a cached ARP table or searching external ARP (sending a broadcast ARP request message).

[1] The Address Resolution Protocol (ARP) is a communication protocol used for discovering the link layer address, such as a MAC address, associated with a given internet layer address, typically an IPV4 address.

In the second case, after grant received from **MAC_TX_ARB**, the **ARPV2 TX** block sends an ARP request packet to the **MAC_TX** FSM specifying the destination address 0xffffffffffff (broadcast request).

Figure 210 shows ARP packet's structure.

<table>
<tr><td colspan="2">Hardware Type</td><td>Protocol Type</td></tr>
<tr><td>Hardware length</td><td>Protocol length</td><td>Operation</td></tr>
<tr><td colspan="2" rowspan="2">Sender hardware address</td><td></td></tr>
<tr><td>Sender protocol address</td></tr>
<tr><td colspan="2"></td><td rowspan="2">Target hardware address</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="3">Target protocol address</td></tr>
</table>

Figure 2-10: ARP packet (32-bit words)

Hardware type (HTYPE): This field specifies the network link protocol type, in our implementation Ethernet (0x0001).

Protocol type (PTYPE): This field specifies the internetwork protocol for which the ARP request is intended. For IPv4, this has the value 0x0800.

Hardware length (HLEN): Length (in octets) of a hardware address. Ethernet address length is 0x06.

Protocol length (PLEN): Length (in octets) of internetwork addresses. IPv4 address length is 0x04.

Operation: Specifies the operation that the sender is performing: 0x0001 for request, 0x0002 for reply.

Sender hardware address (SHA): Media address of the sender. In an ARP request this field is used to indicate the mac address of the host sending the request. In an ARP reply this field is used to indicate the address of the host that the request was looking for.

Sender protocol address (SPA): Internetwork address of the sender.

Target hardware address (THA): Media address of the intended receiver. In an ARP request this field is ignored. In an ARP reply this field is used to indicate the address of the host that originated the ARP request.

Target protocol address (TPA): Internetwork address of the intended receiver.

When **ARPv2 RX** block receives the ARP response with its own address, it updates arp entry data with MAC address associated to the IP address and communicate it to the **IPV4_TX** block.

If **ARPv2 RX** block receives an ARP request, it compares destination IP address with its own IP: in case of matching, this block delivers a unicast response to the sender, filling in the target MAC address field with its MAC address.

MAC_TX FSM is in charge of creating ETH frame, a data link layer protocol data unit with 14-byte ETH header, payload and CRC-checksum (Figure 211).

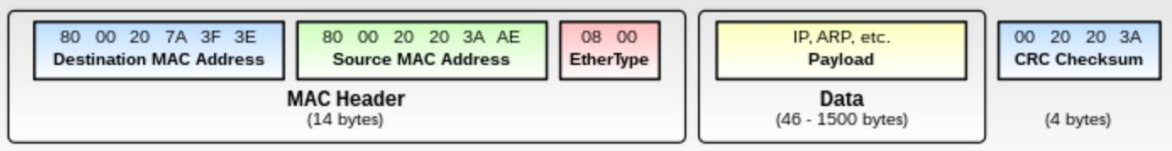

Figure 2-11: Ethernet type II frame

Ether Type field is 0x0800 in case of IPV4 transmission and 0x0806 in case of ARP transmission.

The minimum frame size is 64 bytes, consisting of 14-Byte MAC Header, 4-Byte CRC Checksum and 46-Byte Payload; if the actual data to be sent is less than 46 bytes, it must be padded.

The maximum payload that can be sent is instead of 1500 Bytes (therefore maximum frame size is 1518 Bytes).

On Rx side, data flow is the same: the received ETH frame is sent to IPV4 RX FSM in case of IPV4 packets (i.e. Ether type = 0x0800) otherwise to the **ARP RX** block.

In the first scenario, the IPV4 packet is de-encapsulated in UDP packet and, at the end, UDP payload is written in **FIFO RX HEADER/DATA** by the **UDP Converter** block.

In the second case, as mentioned before, the **ARP RX** block checks if the received packet is an ARP request or an ARP response and behaves accordingly.

To implement the Ethernet Media Access Controller (MAC) with a Physical Coding Sublayer (PCS) we included an AMD 10G/25G High Speed Ethernet Subsystem (**ETH MAC**).

The following figure shows the block diagrams of the 10G/25G High Speed Ethernet Subsystem (GT serial transceivers are not shown).

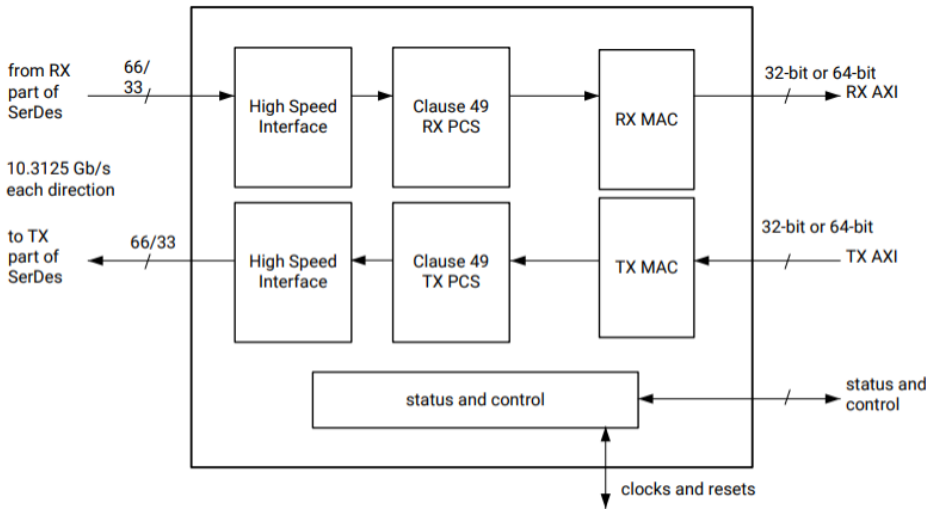

Figure 2-12: Block diagram of 10G/25G High Speed Ethernet Subsystem (without GT transceivers).

User Interfaces available are:

- AXI4 -Stream for datapath interface (TX AXI and RX AXI)
- AXI4-Lite for control and statistics interface

and differential serial interface feature can be configured during the instantiation.

In the Communication_IP we instantiated a single core configuration @10.3125Gbps with single lane GT transceiver inside the IP core, and a datapath interface of 64 bit.

Block **ETH register config** configures the IP core for our use case, accessing AXI4-Lite registers [10].

This configuration is controlled by the **CONFIG** FSM (Figure 213).

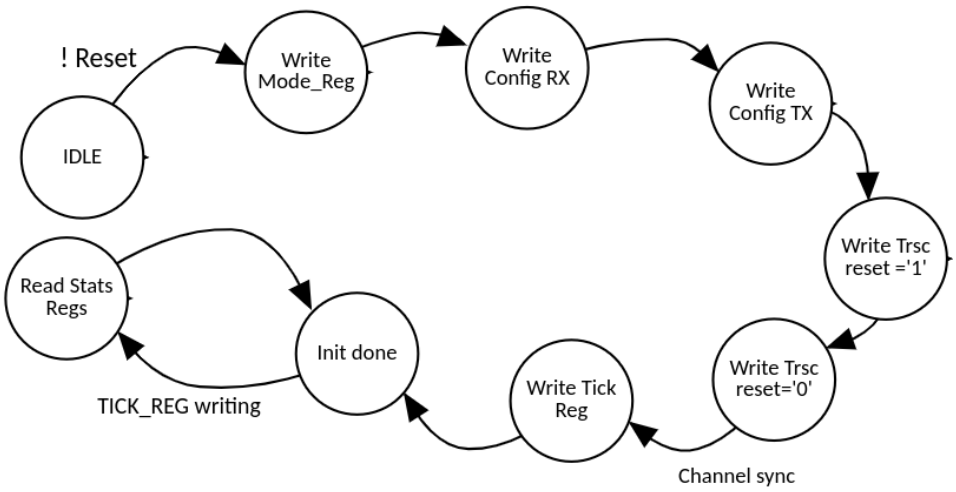

Figure 2-13: ETH register block FSM

Register MODE_REG (0x0008) is set to 0x40000000: bit 30 (tick_reg_mode_sel) equal to '1' allows us to read the statistics counters (which provide histograms of the classification of traffic and error counts) by writing a 1 to the TICK_REG register, while the 0th and 1st bit set to '0' suppress the reporting of slv error.

Bits	Default	Type	Signal
0	1	RW	en_wr_slverr_indication
1	1	RW	en_rd_slverr_indication
30	1	RW	tick_reg_mode_sel
31	0	RW	ctl_local_loopback

Table 2-1: MODE_REG

Register CONFIGURATION_RX_REG1 (0x0014) is set to 0x00000033 and CONFIGURATION_TX_REG1 (0x000C) to 0x00003003.

Bits	Default	Type	Signal
0	1	RW	ctl_rx_enable
1	1	RW	ctl_rx_delete_fcs
2	0	RW	ctl_rx_ignore_fcs
3	0	RW	ctl_rx_process_lfi
4	1	RW	ctl_rx_check_sfd
5	1	RW	ctl_rx_check_preamble
6	0	RW	ctl_rx_force_resync
7	0	RW	ctl_rx_test_pattern
8	0	RW	ctl_rx_test_pattern_enable
9	0	RW	ctl_rx_data_pattern_select
10	-	-	Reserved
11	0	RW	ctl_rx_custom_preamble_enable
12	0	RW	ctl_rx_prbs31_test_pattern_enable

Table 2-2: CONFIGURATION_RX_REG1

Bits	Default	Type	Signal
0	1	RW	ctl_tx_enable
1	1	RW	ctl_tx_fcs_ins_enable
2	0	RW	ctl_tx_ignore_fcs
3	0	RW	ctl_tx_send_lfi
4	0	RW	ctl_tx_send_rfi
5	0	RW	ctl_tx_send_idle
13:10	12	RW	ctl_tx_ipg_value
14	0	RW	ctl_tx_test_pattern
15	0	RW	ctl_tx_test_pattern_enable
16	0	RW	ctl_tx_test_pattern_select
17	0	RW	ctl_tx_data_pattern_select
18	0	RW	ctl_tx_custom_preamble_enable
23	0	RW	ctl_tx_prbs31_test_pattern_enable

Table 2-3: CONFIGURATION_TX_REG1

Register GT_RESET_REG (0x0000) is used to properly reset the transceiver startup Finite State Machine (FSM). We used different timing for simulation and synthesis.

Bits	Default	Type	Signal
0	0	RW	ctl_gt_reset_all This is a clear on write register.
1	0	RW	ctl_gt_rx_reset
2	0	RW	ctl_gt_tx_reset

Table 2-4: RESET_REG

After core lock (the receiver has detected and locked to the word boundaries), a first writing in the TICK_REG register (0x0020) allows to set to 0 readable STAT_*_MSB/LSB registers (which are not resettable).

The initialization of the core is then completed (signal channel_ok ='1') and the FSM, in INIT_DONE state, waits for the writing of the TICK_REG (achieved by writing Communication_IP register 70) which causes the accumulated counts to be pushed to the readable STAT_*_MSB/LSB registers and simultaneously clear the accumulators.

Therefore, to offer to the users the possibility to set at run time a few key features and to read status information of the new ETH port, we added some registers to the Communication_IP.

2.3 Ethernet port simulation and test

For the ETH_port we primarily tested the UDP/MAC blocks interaction using the configuration shown in figure 214 (this configuration can be yet implemented with VHDL parameter test_mac='1' in Textarossa_switch_synt entity).

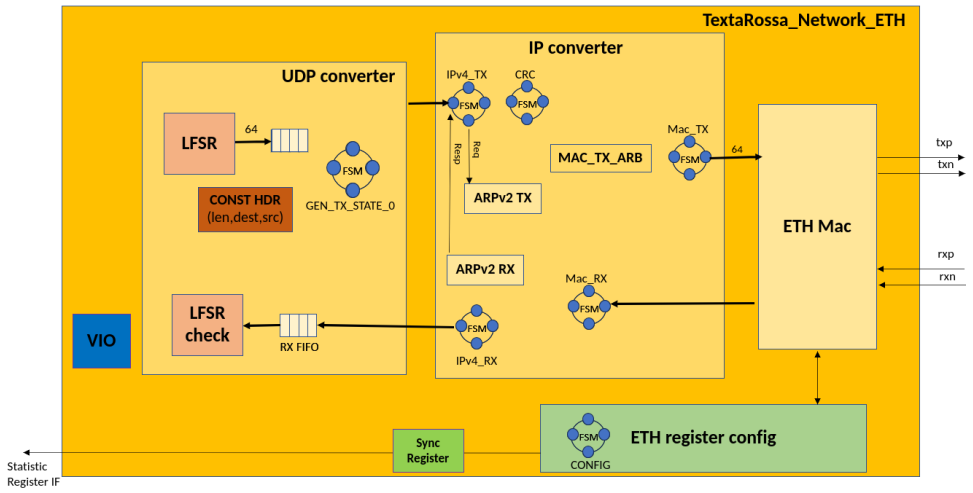

Figure 2-14: ETH port's architecture in TEST_MAC configuration

The **GEN_TX_STATE_0** FSM (enabled by parameter test_mac_gen='1') generates c_tx_pkt_cnt UDP packets with constant header:

- Destination port: 0xfa62;
- Source port: 0xfa62;
- Total length in byte: 0x0040;
- Checksum: 0x0000

and pseudo-random payload generated by LFSR block (a generator based on linear feedback shift register).

Destination_IP_address is set to x"c0a80002".

The **LSFR_check** block flushes the RX FIFO, checking payload of received packets.

The Virtual input/output core (**VIO**) drives reset and tick_reg (to read statistics registers of ETH MAC) signals in real-time.

ILA IP allows us to monitor ETH MAC statistics registers and **LFSR_check** block signals (received data and identified errors).

Figure 215 shows the simulation of two connected Communication_IPs in TEST_MAC configuration:

- Textarossa 0: parameter test_mac_gen='1', c_tx_pkt_cnt =1 (512 Byte); MAC_ADDRESS: d00bacc0aaaa; IP_ADDRESS: c0a80002
- Textarossa 1: parameter test_mac_gen='0'; MAC_ADDRESS: d00dacc0aaaa; IP_ADDRESS: c0a80002

The waveform shows that Textarossa 1 correctly receives 622 Byte (rx_good_data) - 62 Byte for ARP request and 560 Byte for IPV4 transmission - and sends 64 Byte (rx_good_data) for ARP response.

The outputs of Textarossa 1 **LFSR_check** block (in orange) show that packet was received (64 words @64bit) without errors.

Figure 2-15: ETH port simulation in TEST_MAC configuration

The design in TEST_MAC configuration was loaded into the FPGA, and the debugger internal signals were monitored using an Integrated Logic Analyzer (ILA).

Also in this case, as shown in Figure 216, data were received (Rx_data_cnt=64) without errors (Rx_err_cnt=0).

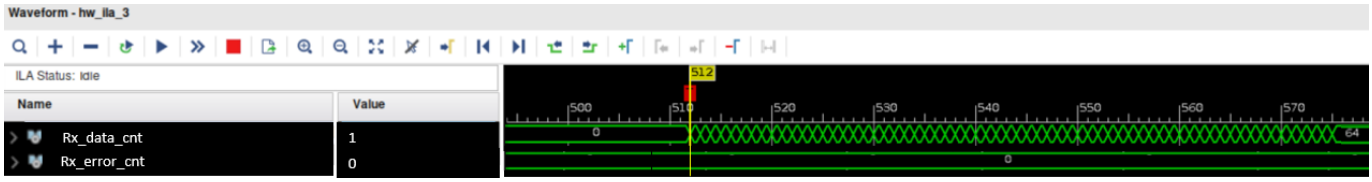

Figure 2-16: ILA of board in TEST_MAC configuration

In the second test setup configuration (NORMAL_MODE, Figure 2-17), on the TX side, the **UDP_converter** block generates UDP packets starting from information sent by the **ROUTING_IP** (header+payload+footer).

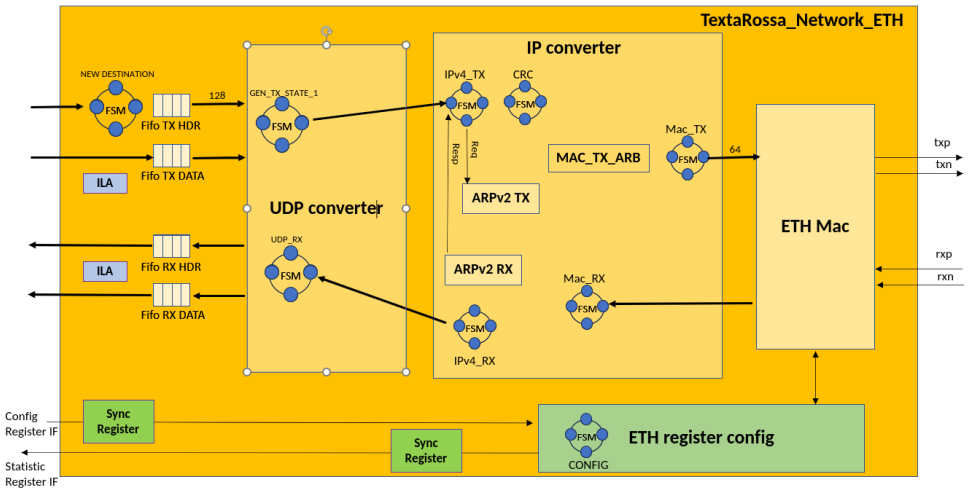

Figure 2-17: ETH port's architecture in NORMAL_MODE configuration

The **GEN_TX_STATE_1** FSM generates the UDP header from information contained in **FIFO TX HEADER** and encapsulating Textarossa packet header (shown in Figure 23) and footer in the UDP payload (with the Textarossa packet payload read from **FIFO TX DATA**), going from 128 to 64-bit interface.

Figure 218 shows the simulation of a Communication_IPs in NORMAL_MODE configuration, connected in loopback: we used **internal_generator** block to generate packets and fill the transmitting FIFOs, and an **internal_consumer** block to flush the receiving FIFOs and check payload of received packets (if all packets are received the signal test_ok arises).

Figure 2-18: ETH port simulation in NORMAL MODE

The design was also tested, and internal signals were monitored using Integrated Logic Analyzers (Figure 219).

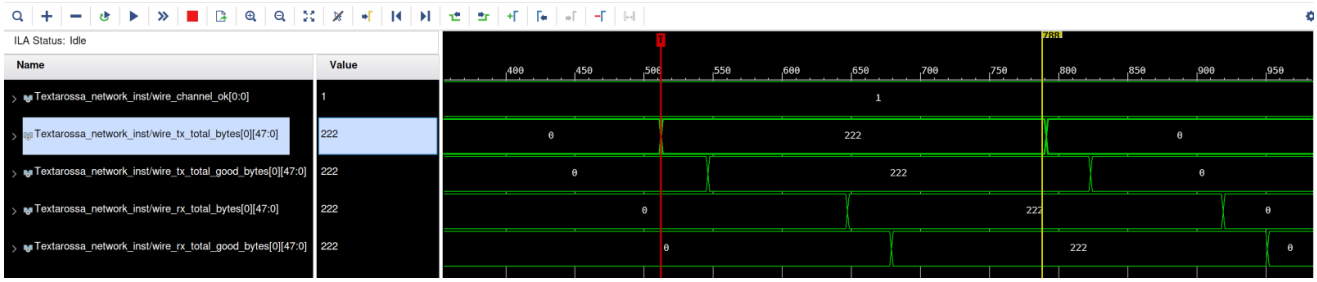

Figure 2-19: ILA of board in NORMAL MODE configuration

This port is also used connected to a switch with an internal datapath of 256 bit width.

2.4 Configuration and Status Registers

This is the list of configuration and status registers of the Communication IP along with the associated address (in green the registers related to ETH port):

	Offset	Name	Description	Default value	
4	0x00000010	RESET_REG	Bit 0: write '1' to reset; Self-clear ('0' after 200 clock's cycle)	0x0000000	RW
5	0x00000014	REVISION_REG	Bit 15 downto 0: Revision ID Bit 31 downto 16: Version ID	0x0000000	RO
6	0x00000018	COORDME_REG	3D node's coordinates Bit 5 downto 0: X coordinate	0x0000000	RW
8	0x00000020	LATTICESIZE_REG	Lattice size Bit 5 downto 0: X direction	0xfffffff	RW
12	0x00000030	PERF_INTRANODE_CF	Perf_Block configuration register: Bit 7 downto 0: IntraNode Packet_Generator enable Bit 15 downto 8: IntraNode Consumer enable	0x0000000	RW

			Bit 23 downto 16: IntraNode BW test enable		
13	0x00000034	PERF_INTERNODE_CF	Perf_Block configuration register: Bit 7 downto 0: InterNode Packet_Generator enable Bit 15 downto 8: InterNode Consumer enable	0x00000000	RW
14	0x00000038	PKTGEN_CONFIG_0	Packet_generator configuration register: Bit 15 downto 0: number of packets generated Bit 29 downto 16: packet length (in byte) Bit 31: header only packet generated	0x00000000	RW
16	0x00000040	PKTGEN_CONFIG_1	Destination of packet Bit 5 downto 0: X direction Bit 10 downto 6: Y direction Bit 15 downto 11: Z direction	0x00000000	RW
20	0x00000050	PERF_INTRANODE_ST	Bit 3 downto 0: packet_generator status (intraNode 0) Bit 7 downto 4: packet_checker status (intraNode 0) Bit 11 downto 8: packet_generator status (intraNode 1) Bit 15 downto 12: packet_checker status (intraNode 1) Bit 19 downto 16: packet_generator status (intraNode 2) Bit 23 downto 20: packet_checker status (intraNode 2) Bit 27 downto 24: packet_generator status (intraNode 3) Bit 31 downto 28: packet_checker status (intraNode 3) *Packet_generator status: "0000" SM_STATE = OFF "0001" SM_STATE = IDLE "0010" SM_STATE = TX_HEADER "0011" SM_STATE = TX_PAYLOAD "0100" SM_STATE = TX_FOOTER **Packet_checker status		RO

			Bit 0 = Test ok! (All packets received with correct payload) Bit 3 downto 1: SM STATE "000" SM_STATE = OFF "001" SM_STATE = IDLE "010" SM_STATE = COUNT		
21	0x00000054	PERF_INTERNODE_ST	Bit 3 downto 0: packet_generator status (see register PERF_INTRANODE_STS) Link 0 Bit 7 downto 4: packet_checker status Link 0 Bit 11 downto 8: packet_generator status Link 1 Bit 15 downto 12: packet_checker status Link 1		RO
22	0x00000058	PERF_INTRANODE_CNT0	TxRx clock counter (first packet written, last packet read) IntraNode 0		RO
23	0x0000005C	PERF_INTRANODE_CNT1	TxRx clock counter (first packet written, last packet read) IntraNode 1		RO
24	0x00000060	PERF_INTRANODE_CNT2	TxRx clock counter (first packet written, last packet read) IntraNode 2		RO
25	0x00000064	PERF_INTRANODE_CNT3	TxRx clock counter (first packet written, last packet read) IntraNode 3		RO
26	0x00000068	PERF_INTERNODE_CNT0	TxRx clock counter (first packet written, last packet read) InterNode 0		RO
27	0x0000006C	PERF_INTERNODE_CNT1	TxRx clock counter (first packet written, last packet read) InterNode 1		RO
28	0x00000070	INTRANODE_FIFO_STS_RX_0	Bit 31 downto 16: Fifo IntraNode 0 Data Rx UsedWord Bit 15 downto 0: Fifo IntraNode 0 Header Rx UsedWord		RO
29	0x00000074	INTRANODE_FIFO_STS_TX_0	Bit 31 downto 16: Fifo IntraNode 0 Data Tx UsedWord Bit 15 downto 0: Fifo IntraNode 0 Header Tx UsedWord		RO
30	0x00000078	INTRANODE_FIFO_CNT_HD_TX_RD_0	Fifo IntraNode 0 Header Tx read counter		RO
31	0x0000007C	INTRANODE_FIFO_CNT_HD_TX_WR_0	Fifo IntraNode 0 Header Tx write counter		RO
32	0x00000080	INTRANODE_FIFO_CNT_HD_RX_RD_0	Fifo IntraNode 0 Header Rx read counter		RO
33	0x00000084	INTRANODE_FIFO_CNT_HD_RX_WR_0	Fifo IntraNode 0 Header Rx write counter		RO

34	0x00000088	INTRANODE_FIFO_CNT_DT_TX_RD_0	IntraNode 0 Data Tx read counter		RO
35	0x0000008c	INTRANODE_FIFO_CNT_DT_TX_WR_0	IntraNode 0 Data Tx write counter		RO
36	0x00000090	INTRANODE_FIFO_CNT_DT_RX_RD_0	IntraNode 0 Data Rx read counter		RO
37	0x00000094	INTRANODE_FIFO_CNT_DT_RX_WR_0	IntraNode 0 Data Rx write counter		RO
38 47	0x00000098- 0x000000BC	INTRANODE_FIFO_*_1	Fifo counter register IntraNode 1		RO
48 57	0x000000C0- 0x000000E4	INTRANODE_FIFO_*_2	Fifo counter register IntraNode 1		RO
58 67	0x00000098- 0x0000008C	INTRANODE_FIFO_*_3	Fifo counter register IntraNode 1		RO
68	0x00000110	LINK_0_CONFIG_0	Bit 31 downto 28: Edac enable InterNode 1 "0000" NO EDAC "1111" EDAC Bit 27 downto 24 = Edac enable InterNode 0 Bit 17 = Use new destination in InterNode 1 Bit 16 = Use new destination in InterNode 0 Bit 15 downto 0: New destination (15–11: Z; 10–6: Y; 5–0:X).	0x00000000	RW
69	0x00000114	LINK_0_CONFIG_1	Bit 25 downto 16: Red data threshold Bit 7 downto 0: Red header threshold	0x00000000	RW
70	0x00000118	LINK_0_CONFIG_2	Bit 6: the accumulated counts are pushed to the readable ETH_* registers andl simultaneously the accumulators are cleaned Bit 15 downto 8: Tx new credit cycle Bit 7 downto 0: Tx waiting cycle	0x00000000	RW
71	0x0000011C	LINK_0_CONFIG_3	Header error gen		RO
80	0x00000140	LINK_0_STATUS_0	Bit 15 downto 12: Rx status; Bit 11 downto 8: Tx footer status Bit 7 downto 4: Tx payload status Bit 3 downto 0: Tx header status		RO
81	0x00000144	LINK_0_ERROR	Bit 31 downto 16: Rx header error counter Bit 15 downto 0: Rx header fatal error counter		RO
82	0x00000148	LINK_0_TX_MAGIC	Tx magic counter		RO

83	0x0000014C	LINK_0_TX_START	Tx start counter		RO
84	0x00000150	LINK_0_TX_HDR	Tx header counter		RO
85	0x00000154	LINK_0_TX_FTR	Tx footer counter		RO
86	0x00000158	LINK_0_RX_MAGIC	Rx magic counter		RO
87	0x0000015c	LINK_0_RX_START	Rx start counter		RO
88	0x00000160	LINK_0_RX_HEADER	Rx header counter		RO
89	0x00000164	LINK_0_RX_FOOTER	Rx footer counter		RO
90 99	0x00000168- 0x00000185	LINK_1_REGISTERS			
110	0x000001B8	LINK0_RD_WR_CNT_0	Header read counter Link TX0		RO
111	0x000001BC	LINK0_RD_WR_CNT_1	Data read counter Link TX0		RO
112	0x000001C0	LINK0_RD_WR_CNT_2	Header write counter Link TX0		RO
113	0x000001C4	LINK0_RD_WR_CNT_3	Data write counter Link TX0		RO
114	0x000001C8	LINK0_RD_WR_CNT_4	Data write counter Link RX0 VCH0		RO
115	0x000001CC	LINK0_RD_WR_CNT_5	Header write counter Link RX0 VCH0		RO
116	0x000001D0	LINK0_RD_WR_CNT_6	Data read counter Link RX0 VCH0		RO
117	0x000001D4	LINK0_RD_WR_CNT_7	Header read counter Link RX0 VCH0		RO
118	0x000001D8	LINK0_RD_WR_CNT_8	Data write counter Link RX0 VCH1		RO
119	0x000001DC	LINK0_RD_WR_CNT_9	Header write counter Link RX0 VCH1		RO
120	0x000001E0	LINK0_RD_WR_CNT_10	Data read counter Link RX0 VCH1		RO
121	0x000001E4	LINK0_RD_WR_CNT_11	Header read counter Link RX0 VCH1		RO
122- 133	0x000001E8- 0x00000214	LINK1_RD_WR_CNT_*			RO
150	0x00000258	FIFO_INTRANODE_EXC	Bit 7 downto 0 = IntraNode TX HD write exception Bit 15 downto 8 = IntraNode TX DT write exception Bit 23 downto 16 = IntraNode RX HD write exception Bit 31 downto 24 = IntraNode RX DT write exception		RO

151	0x00000260	FIFO_REGISTER	Bit 31 downto 24: Fifo Header Rx exp width Bit 23 downto 16: Fifo Data Rx exp width Bit 15 downto 8: Fifo Header Tx exp width Bit 7 downto 0: Fifo Data Tx exp width		RO
152	0x00000264	TRANSCEIVER_STATUS	Bit 0: InterNode 0 channel up Bit 1: InterNode 1 channel up Bit 16: InterNode 0 transceiver's error Bit 17: InterNode 1 transceiver's error		RO
200	0x00000320	IP ADDRESS	ETH IP address	0xc0a80002	RW
201	0x00000324	MAC_REG_LOW	MAC address low	0xacc0aaaa	RW
202	0x00000328	MAC_REG_HIGH	MAC address high	0x0000d00b	RW
203	0x0000032C	ETH_TX_GOOD_BYTE_LSB	Number of good bytes sent (LSB)	0x00000000	RO
204	0x00000330	ETH_TX_GOOD_BYTE_MSB	Number of good bytes sent (MSB)	0x00000000	RO
203	0x00000334	ETH_RX_GOOD_BYTE_LSB	Number of good bytes received (LSB)	0x00000000	RO
204	0x00000338	ETH_RX_GOOD_BYTE_MS	Number of good bytes received (MSB)	0x00000000	RO

Table 2-5: Communication_IP configuration and status registers (in green the registers related to ETH port):

3 Performance tests

Latency and bandwidth tests were conducted to validate the performance of the final version of the Communication IP integrating the improvements described in the previous sections. In the setup used for tests, the Communication IP is implemented as an RTL-IP Xilinx fashioned free running kernel connected to the global system/board clock of 150 MHz (improved wrt the preliminary release which featured a clock frequency of 100MHz). Performance tests have been reported and compared for the two developed versions of Communication IP (128-bit internal datapath width/2 lanes inter-node channels versus 256-bit internal datapath width/4 lanes inter-node channels).

Figure 31 shows the general test setup with Communication IP featured with four intranode ports and two internode ports, and four couples of dispatcher/aggregator.

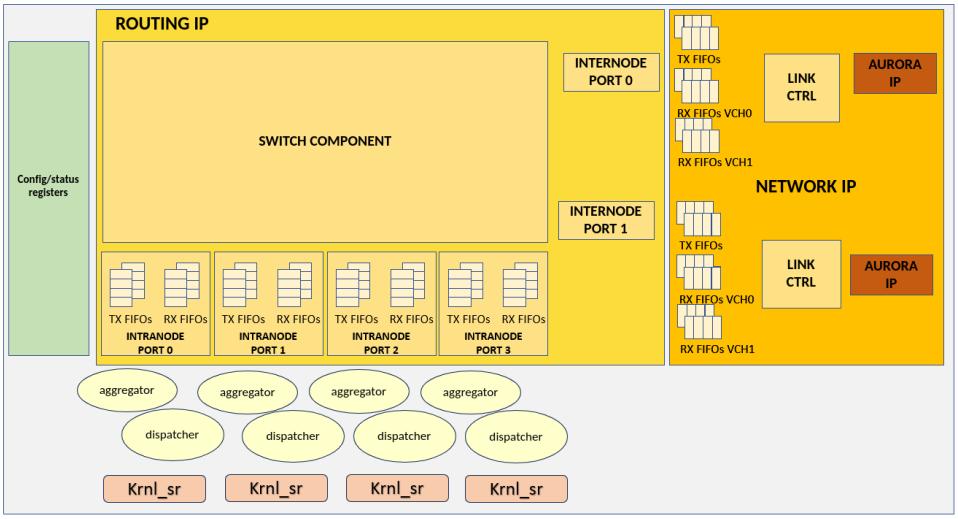

Figure 3-1: Setup used to assess the performance of the Communication IP

Tests described in this deliverable have been performed on two different testbench: 1) a dual server system, integrating a single Xilinx Alveo U200 FPGA per server, interconnected via the internode ports of the Communication IP; 2) a similar system hosting Xilinx Alveo U280 FPGAs.

The pseudocode, which describe the summary of tests' execution, is reported in Appendix. A.

The complete code of the test setup can be accessed on the APEIRON framework github repository (https://github.com/APE-group/APEIRON).

3.1 Latency test

Latency test is performed using an HLS kernel which reads a payload (of max 4096 Bytes) data item from the memory (either BRAM or DDR) of the "initiator" FPGA and sends and receives it through/from the Communication IP to/from a second interconnected FPGA. An HLS kernel, embedded in this receiver FPGA, is in charge of getting a single packet and bouncing it back to the initiator FPGA (as shown in Figure 32), allowing the measurement of inter-node latency.

The "send_receive" HLS kernel on the initiator FPGA is issued via host code while the "pipe" HLS kernel is free running. In order to minimize the contribution of the host call overhead to the latency measure, one million send_receive operations are launched and the overall time elapsed from the start of the first packet

send to the completion of the last packet receive is measured on the host. The end-to-end latency is then obtained as the half the overall elapsed time measured divided by the number of repetitions.

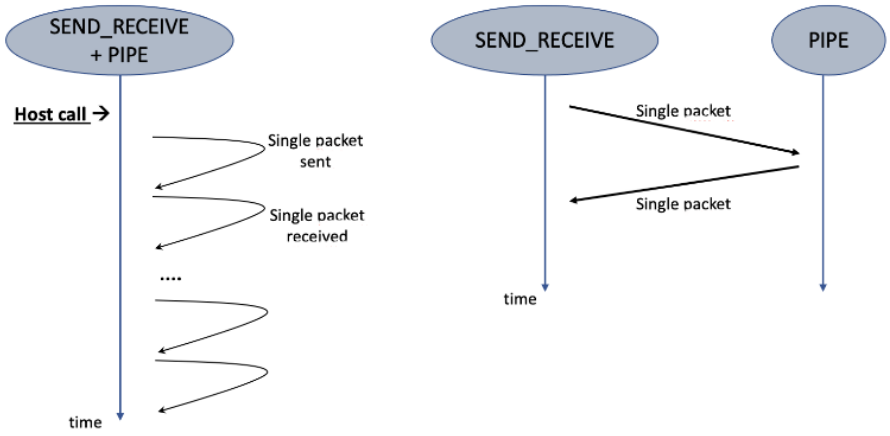

Figure 3-2: Latency test scheme

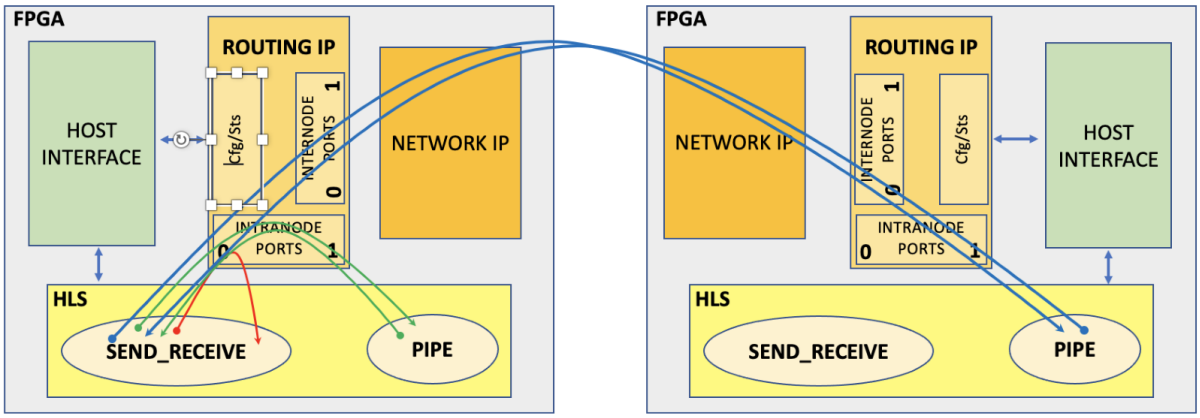

Figure 3-3: Testbench design illustration. The arrows describe different flows of data depending on the test performed: "Local-loop" (red arrow), "Local-trip" (green arrows), "Roundtrip" (blue arrows)

We conducted several kinds of latency tests, covering all the possible paths involved in the communication within the same FPGA and on different FPGAs.

In detail, to stress and validate the performance of the intra-node communication, "local-loop" and "local-trip" tests were performed. These two, as can be seen in Figure 33, differ for what concerns the intra-node ports where the communication takes place. In the local-loop case, packets are sent on the port 0 and then routed back to the same port (red line), while in the local-trip test, packets are sent to the port 1 through the port 0, and then they are received and sent back by the kernel "pipe" connected to the port 1 (green lines). The "roundtrip" test (blue lines) is suitable to validate performance of the inter-node communication between 2 different FPGAs (respectively named as "node 0" and "node 1"). For all these test configurations we report the time taken to complete a packet traversal between source and destination node, i.e. the end-to-end communication latency.

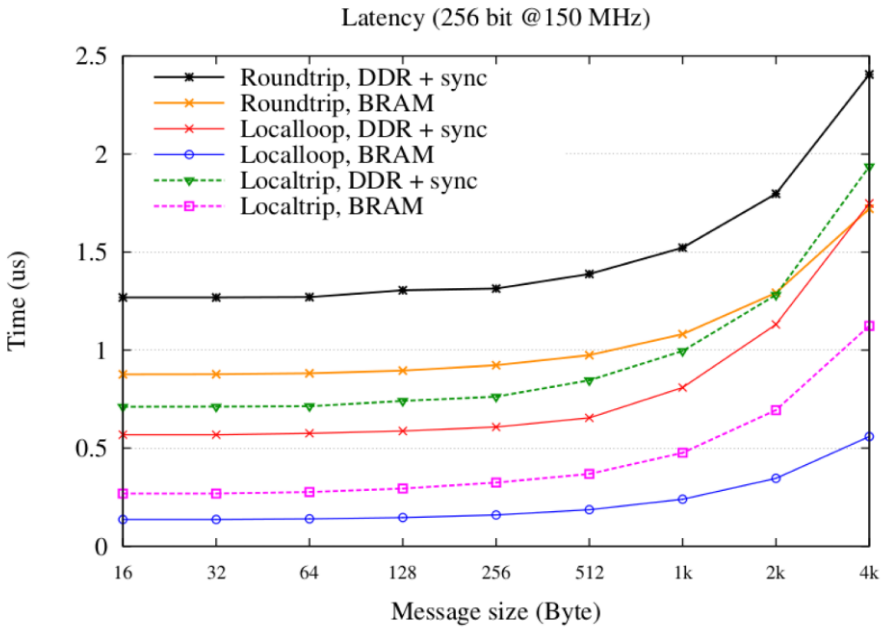

Figure 3-4: Measured latency of HLS Kernels intra-node (localloop, localtrip) communication and inter-node (roundtrip) communication using BRAM and DDR to allocate send/receive buffers.
These results refer to the 256-bit internal datapath width @150 MHz configuration of the Communication IP.

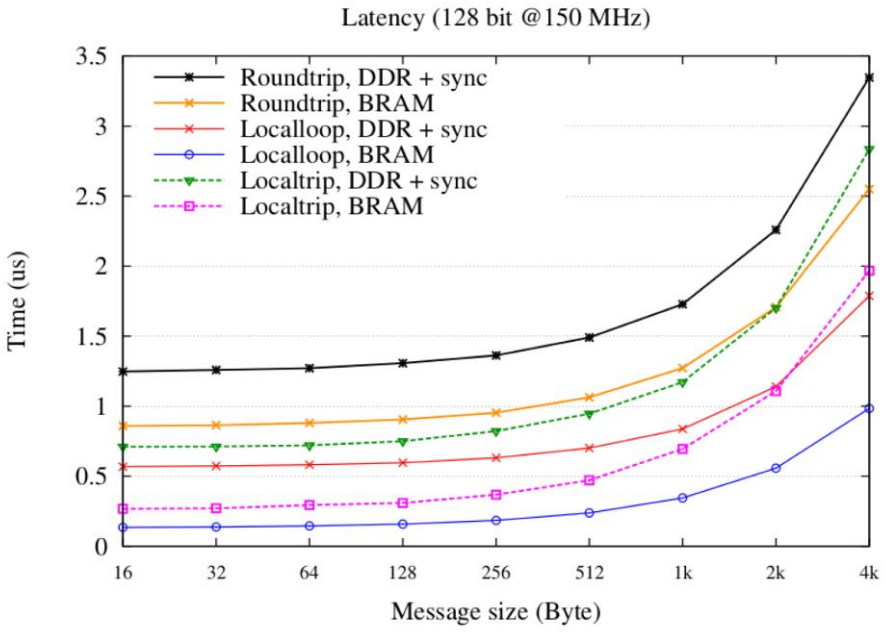

Figure 3-5: Measured latency of HLS Kernels intra-node (loopback, localtrip) communication and inter-node (roundtrip) communication using BRAM and DDR to allocate send/receive buffers.
These results refer to the 128-bit internal datapath width @150 MHz configuration of the Communication IP

In Figure 3-4 and Figure 3-5, the results of latency test are reported differentiated by the specific type of test performed and what kind of FPGA memory was used. It's possible to notice how latency values increase when working with the onboard DDR memory, which introduces an overhead due to its access latency and to the time required for the data synchronization between the CPU and FPGA (what we indicate as "sync" in the graph). Unsurprisingly, using the BRAM to store the payload on source and destination endpoints yields the lower latency values reached by the setup in each of the of communication configurations.

In the intra-node communication (local-trip configuration, sender and receiver on different intra-node ports) we measure the contribution of the Routing IP, of the Aggregator/Dispatcher, and of the BRAM or DRAM access (and sync operation for the latter) to the overall end-to-end communication latency. In this setup the latency to transfer a 16B message varies from 267 ns (BRAM) to 710 ns (DDR+sync). In the inter-node configuration (Roundtrip setup), the latency measurement takes into account also the delay introduced by the Network IP (mainly due to serialization/deserialization stages) and the latency ranges between 858 ns (BRAM) and 1240 ns (DDR+sync). It is worth noticing that when using the BRAM (yellow line), the end-to-end latency remains below 1 us for packet payload sizes up to 512B using the 256-bit datapath version of the Communication IP.

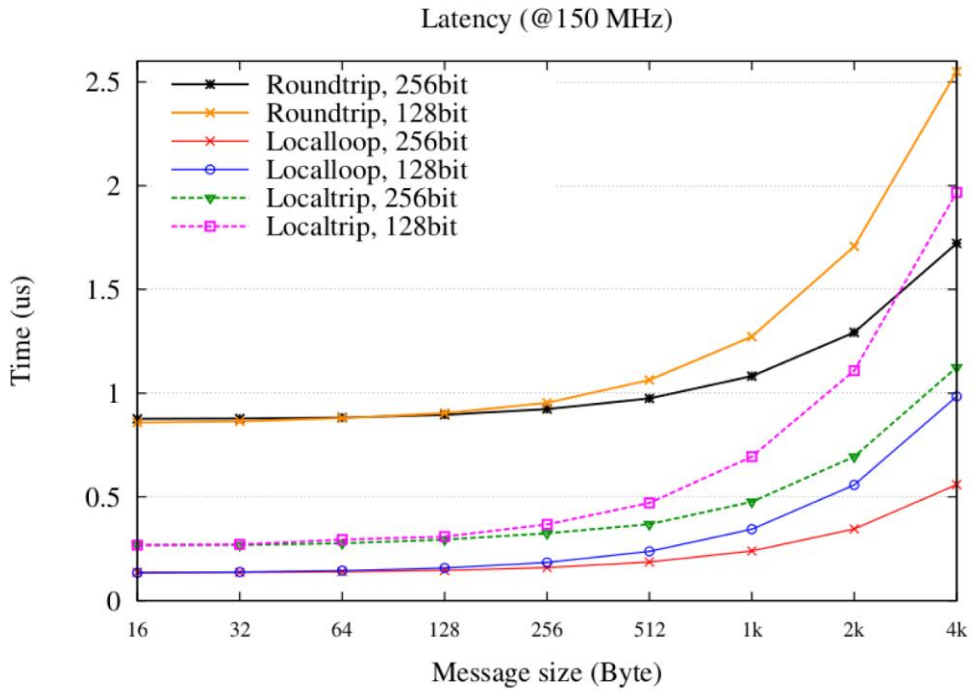

Figure 3-6:Measured latencies on the 128-bit and the 256-bit internal datapath setups. Send and receive buffers allocated on BRAM.

In the test setup integrating the 256-bit datapath for the Communication IP, the forwarding of packet payload to the network occurs by sending half of the words required for the 128-bit datapath case: we can notice from Figure 3-6 how every latency value obtained for a given message size in the 128-bit datapath setup is comparable to the latency value obtained in the 256-bit one for a message of twice the size. This behaviour clearly improves the performance of the 256-bit setup wrt the 128-bit one for message sizes larger than 32B.

3.2 Bandwidth test

As shown in Figure 3-7, bandwidth test is carried out by transferring multiple data packets with fixed payload size from a “sender” HLS kernel which reads data from the source buffer in FPGA memory (either DDR or BRAM). The sender HLS kernel forwards them through the Communication IP to another FPGA where a “receiver” HLS kernel writes data into the destination buffer in memory. After receiving the number of data packets whose integrated payload adds up to the size of the receive buffer, the second FPGA send back a single “ACK” packet with minimal payload to confirm the reception (one-way mode). In loop-back mode sender and receiver are two tasks on the same FPGA. The total data sent through the

network is summed and then divided by the time (measured on the sender node) elapsed between the start of the multiple packets send and the completion of the receive operation of the ACK packet.

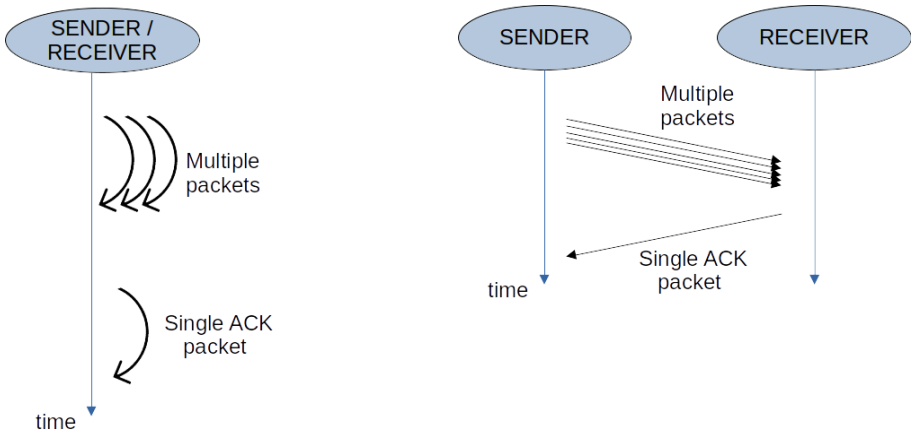

Figure 3-7: Bandwidth test scheme

For the bandwidth, we will not dwell on commenting in detail values obtained in the DDR test cases where it is capped by the overhead due to the memory access that also explain the lack of measured bandwidth differences between intra-nodes and inter-node tests.

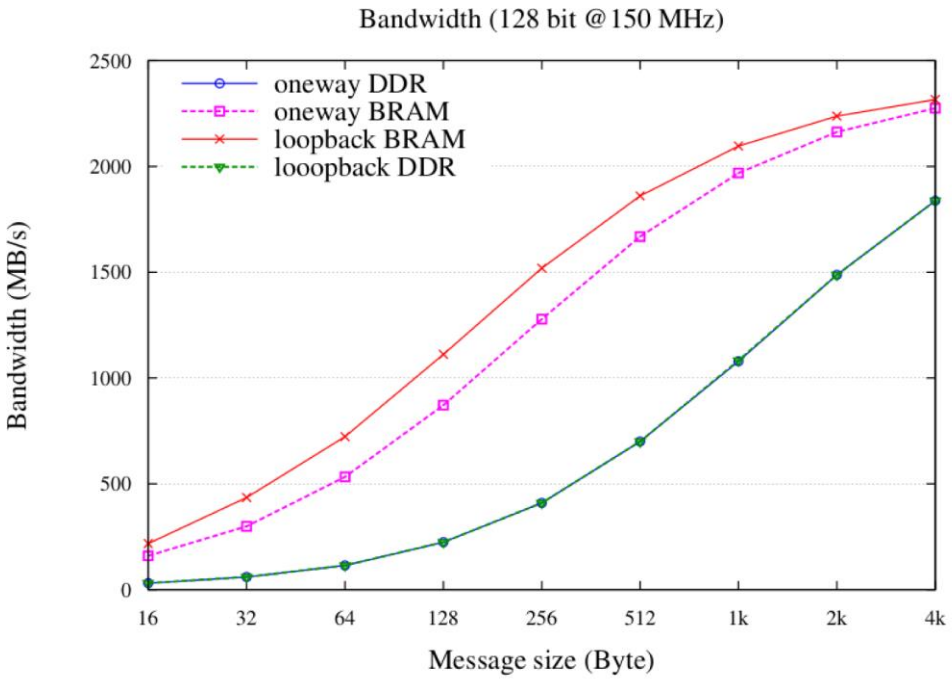

Figure 3-8: Measured bandwidth in HLS Kernels intra-node (loopback) communication and inter-node (oneway) communication using BRAM and DDR to allocate send/receive buffers.
These results refer to the 128-bit internal datapath setup.

Referring to the BRAM case, in Figure 3-8 we notice that the bandwidth tends to saturate while increasing the size of sent messages. For the 128-bit datapath setup, the bandwidth reaches a value of ~18.1 Gbps for both intra-node loopback BRAM case (red line) and for the inter-node BRAM case (fuchsia line) compatible with the maximum theoretical value of raw bandwidth, equal to 19.2 Gbps, due to the data injection rate

at the router port (128bit@150MHz). The difference is mainly due to the packet protocol overhead and to the effect of the 64b/66b encoding on the intra-node channels.

Figure 3-9 shows the bandwidth measurements for the same test modes but for the 256-bit datapath where, with a 150 MHz clock frequency, the maximum raw bandwidth theoretical value is 38.4 Gbps. At the maximum payload size of 4kB and using BRAM both the intra-node loopback (red line) and the inter-node one-way (fuchsia line), bandwidths still do not saturate, reaching 35.8 Gbps and 34.6 Gbps respectively.

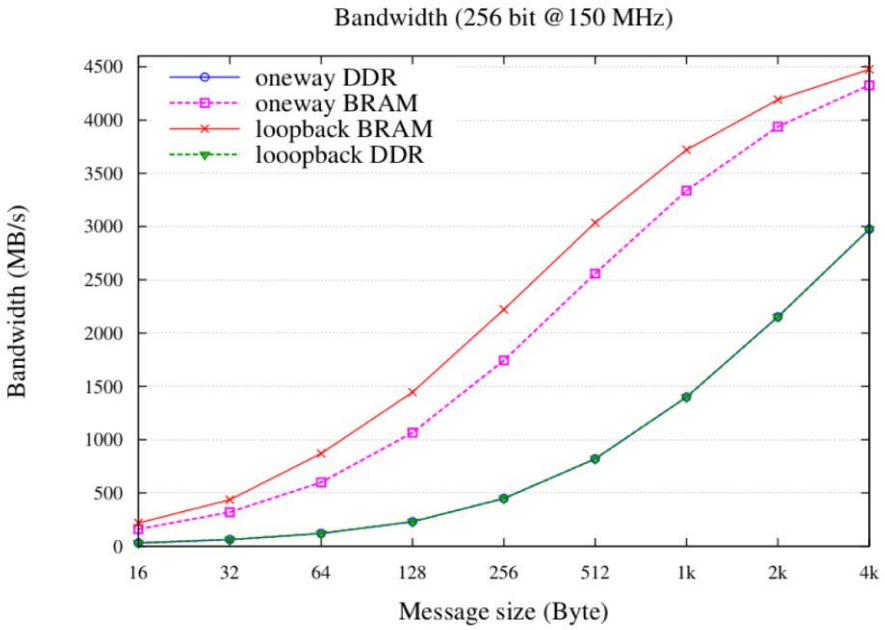

Figure 3-9*:* Measured bandwidth in HLS Kernels intra-node (loop-back) communication and inter-node (one-way) communication using BRAM and DDR to allocate send/receive buffers.
These results refer to the 256-bit internal datapath setup.

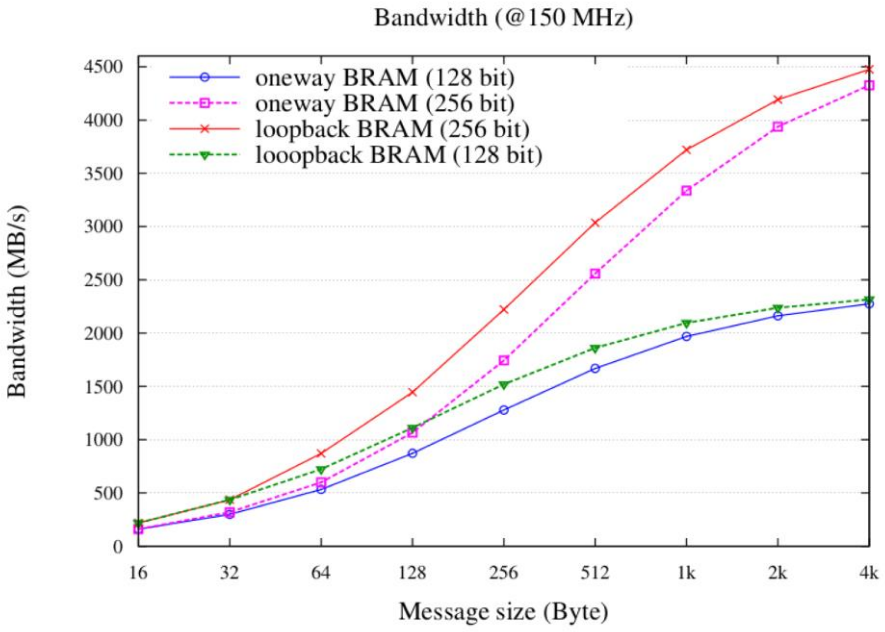

Figure 3-10: Comparison between measured bandwidth with the 256-bit and the 128-bit internal datapath setups

The differences in terms of bandwidth between the two setups can be appreciated in Figure 3-10 where tests results are shown for both used internal datapaths. As expected, the bandwidth for 256-bit setup tends to saturate at a value equal to twice the one reached in the 128-bit setup.

3.3 Multi-node test: 4 Alveo U200 boards

To evaluate the performance of the Communication IP in a multi-node setup, we used a testbed composed by 4 Xilinx Alveo U200, as shown in Figure 3-11, connected in a ring topology, measuring the additional latency introduced by the presence of an intermediate node between the source (node 0) and destination (node 2) communication endpoints.

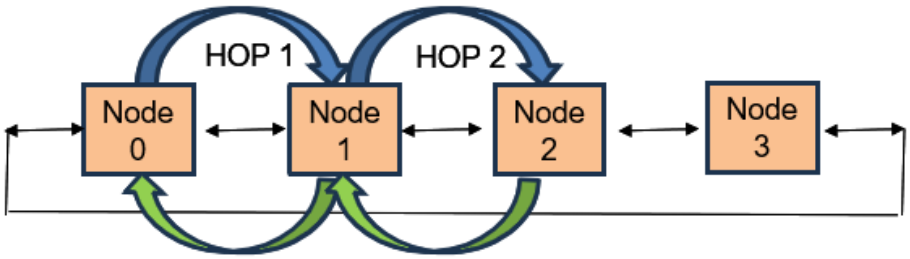

Figure 3-11: Testbed composed by 4 Xilinx U200 connected in a ring topology.

Since the crossing of an intermediate peer of a direct network (we refer to this passage as a “hop” operation) will result in additional operations to route packets to the destination node, we decided to perform a latency test, similar to the one described in Section 3.1, sending packets from node 0 to node 2. In the following we report the measured one and two hops latency.

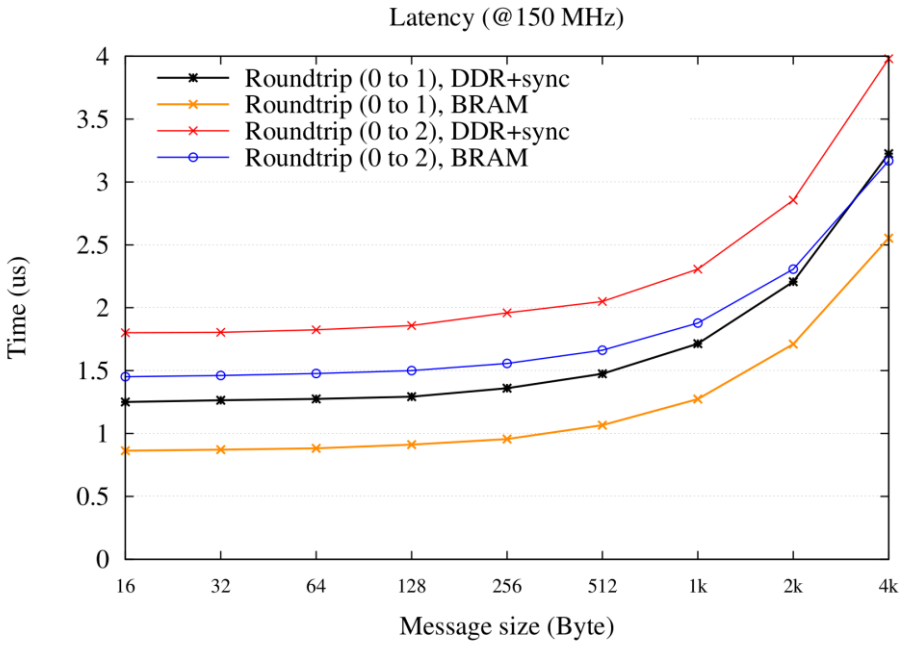

Figure 3-12: Measured latency of HLS Kernels inter-node (roundtrip) communication performed between nodes 0 and 1 (1 hop) and between nodes 0 and 2 (2 hops).
These results refer to the 128-bit internal datapath setup with a global clock of 150 MHz

As can be seen from Figure 3-12, inter-node communication latencies measured in the two hops case are slightly higher with respect to the single hop one. As we expect, the differences between the values obtained in the two configurations are mostly constant and equal to ~600 ns, the latency cost introduced by one hop in the network.

4 High Performance IP Configuration

In this section we describe the performance of the Communication IP in the most performant configuration we have been able to deploy in the TEXTAROSSA project. This configuration is characterized by an internal datapath width of 256 bit and an operating frequency of 200 MHz of the IP core logic, with inter-node channels implemented with 4 bonded serial lanes.

Referring to the performance, Figure 4-1 shows latency measurements obtained with the same test configurations described in Section 3.1. They are coherently better (so, lower) than those obtained using the 256-bit datapath, @150 MHz clock configuration (see Figure 3-4).

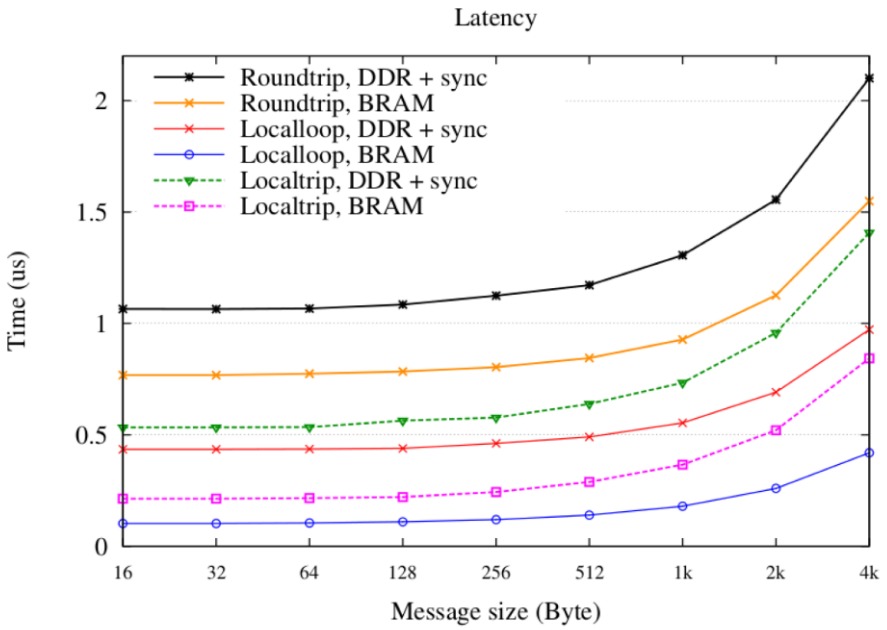

Figure 4-1: Measured latency of HLS Kernels intra-node (loopback, localtrip) communication and inter-node (roundtrip) communication using BRAM and DDR to allocate send/receive buffers.
These results refer to the 256bit internal datapath setup with a global clock of 200 MHz and 4 lanes channels.

Table 4-1 summarizes end-to-end latency values for intra-node and inter-node communications measured using packets with 16/32B payload size.

	DDR+Sync (ns)	**BRAM (ns)**
Intra-node (localtrip)	533	213
Inter-node (roundtrip)	1065	768

Table 4-1: End-to-end latency values for intra-node and inter-node communications using packets with 16/32B payload size. These results are referred to the 256bit internal datapath setup with a global clock of 200 MHz and 4 lanes channels.

In this configuration of the IP the maximum theoretical internal raw bandwidth corresponds to the 51.2 Gbps data injection rate at intra-node router port. It can be noticed from Figure 4-2Figure 4-2 that it still has not reached saturation in the intra-node “loopback” BRAM case using packets with the maximum packet payload size of 4kB.

In the inter-node “one-way” BRAM test, the maximum achievable one-way bandwidth is capped by that of inter-node channel. The inter-node ports are implemented using “Gt*_serial_ports” bonded transceivers in the Communication IP Vivado project. In this configuration the single transceiver switching frequency was set to a value of 156 MHz, corresponding to a channel data injection rate of 256 bit@156 MHz, coherent with a raw bandwidth of 39.9 Gbps neglecting the effect of the 64b/66b encoding. The inter-node measured bandwidth using 4kB payload packets is 37.3 Gbps, rather close to this maximum, as shown in Figure 4-2.

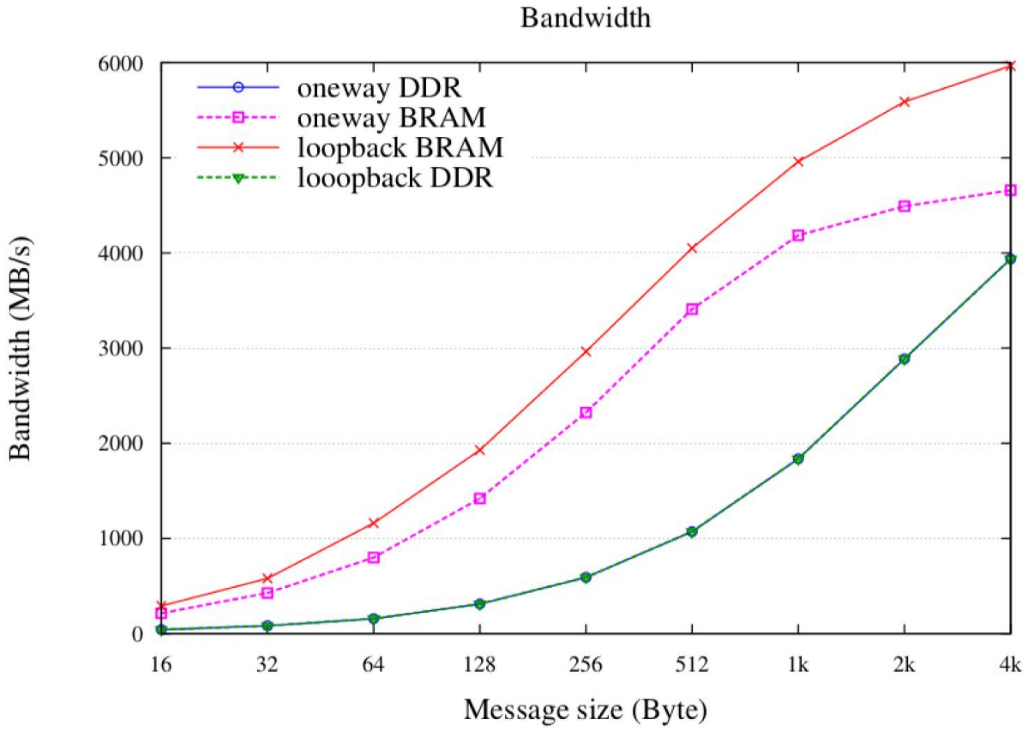

Figure 4-2: Measured bandwidth of HLS Kernels intra-node (loopback) communication and inter-node (one-way) communication using BRAM and DDR to allocate send/receive buffers.
These results refer to the 256-bit internal datapath setup with a global clock of 200 MHz and 4 lanes.

Table 4-2 summarizes measured bandwidth values for intra-node and inter-node communications measured using packets with 4 kB payload size.

	DDR+Sync (MB/s)	BRAM (MB/s)
Intra-node (loopback)	3938	5967
Inter-node (oneway)	3938	4658

Table 4-2 Bandwidth values for intra-node and inter-node communications using packets with 4kB payload size. These results refer to the 256-bit internal datapath setup with a global clock of 200 MHz and 4 lanes channels.

5 Resource usage

Figure 5-1 and Figure 5-2 show the usage report generated for the performance test design (represented in Figure 3-1) implemented with Xilinx Aurora IP 2 lanes design and 128-bit internal datapath version of the Communication IP for both Alveo U200 and Alveo U280 cards.

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	259466 [21.95%]	44381 [7.50%]	350538 [14.83%]	448 [20.74%]	20 [2.08%]	7 [0.10%]
User Budget	922774 [100.00%]	547459 [100.00%]	2013942 [100.00%]	1712 [100.00%]	940 [100.00%]	6833 [100.00%]
Used Resources	30661 [3.32%]	2396 [0.44%]	47896 [2.38%]	76 [4.44%]	0 [0.00%]	24 [0.35%]
Unused Resources	892113 [96.68%]	545063 [99.56%]	1966046 [97.62%]	1636 [95.56%]	940 [100.00%]	6809 [99.65%]
aggregator_0	1185 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_0_1	1185 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1	1185 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1_1	1185 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2	1184 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2_1	1184 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3	1185 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3_1	1185 [0.13%]	0 [0.00%]	2560 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0_1	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1_1	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2_1	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3_1	946 [0.10%]	0 [0.00%]	1758 [0.09%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
krnl_sr	22138 [2.40%]	2396 [0.44%]	30624 [1.52%]	76 [4.44%]	0 [0.00%]	24 [0.35%]
krnl_sr_1	5534 [0.60%]	599 [0.11%]	7656 [0.38%]	19 [1.11%]	0 [0.00%]	6 [0.09%]
krnl_sr_2	5535 [0.60%]	599 [0.11%]	7656 [0.38%]	19 [1.11%]	0 [0.00%]	6 [0.09%]
krnl_sr_3	5534 [0.60%]	599 [0.11%]	7656 [0.38%]	19 [1.11%]	0 [0.00%]	6 [0.09%]
krnl_sr_4	5535 [0.60%]	599 [0.11%]	7656 [0.38%]	19 [1.11%]	0 [0.00%]	6 [0.09%]

Figure 5-1: Resource usage report for 4 intra-node and 2 inter-node (2 lanes) ports @128 bit for U200 card

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	181156 [13.90%]	24106 [4.01%]	241977 [9.28%]	255 [12.65%]	0 [0.00%]	4 [0.04%]
User Budget	1122524 [100.00%]	576854 [100.00%]	2365383 [100.00%]	1761 [100.00%]	960 [100.00%]	9020 [100.00%]
Used Resources	30442 [2.71%]	2388 [0.41%]	45015 [1.90%]	76 [4.32%]	0 [0.00%]	24 [0.27%]
Unused Resources	1092082 [97.29%]	574466 [99.59%]	2320368 [98.10%]	1685 [95.68%]	960 [100.00%]	8996 [99.73%]
aggregator_0	1184 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_0_1	1184 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1	1185 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1_1	1185 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2	1184 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2_1	1184 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3	1184 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3_1	1184 [0.11%]	0 [0.00%]	2560 [0.11%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0	933 [0.08%]	0 [0.00%]	1744 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0_1	933 [0.08%]	0 [0.00%]	1744 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1	933 [0.08%]	0 [0.00%]	1744 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1_1	933 [0.08%]	0 [0.00%]	1744 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2	933 [0.08%]	0 [0.00%]	1744 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2_1	933 [0.08%]	0 [0.00%]	1744 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3	933 [0.08%]	0 [0.00%]	1745 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3_1	933 [0.08%]	0 [0.00%]	1745 [0.07%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
krnl_sr	21973 [1.96%]	2388 [0.41%]	27798 [1.18%]	76 [4.32%]	0 [0.00%]	24 [0.27%]
krnl_sr_1	5494 [0.49%]	597 [0.10%]	6949 [0.29%]	19 [1.08%]	0 [0.00%]	6 [0.07%]
krnl_sr_2	5488 [0.49%]	597 [0.10%]	6950 [0.29%]	19 [1.08%]	0 [0.00%]	6 [0.07%]
krnl_sr_3	5495 [0.49%]	597 [0.10%]	6950 [0.29%]	19 [1.08%]	0 [0.00%]	6 [0.07%]
krnl_sr_4	5496 [0.49%]	597 [0.10%]	6949 [0.29%]	19 [1.08%]	0 [0.00%]	6 [0.07%]

Figure 5-2: Resource usage report for 4 intra-node and 2 inter-node (2 lanes) ports @128 bit for U280 card

The usage reports were generated for the same system integrating the Communication IP configured with 4 lanes channels and internal datapath width of 256 bit (Figure 5-3 and Figure 5-4).

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	259204 [21.92%]	43519 [7.35%]	346168 [14.64%]	448 [20.74%]	20 [2.08%]	7 [0.10%]
User Budget	923036 [100.00%]	548321 [100.00%]	2018312 [100.00%]	1712 [100.00%]	940 [100.00%]	6833 [100.00%]
Used Resources	39170 [4.24%]	2644 [0.48%]	73640 [3.65%]	92 [5.37%]	0 [0.00%]	24 [0.35%]
Unused Resources	883866 [95.76%]	545677 [99.52%]	1944672 [96.35%]	1620 [94.63%]	940 [100.00%]	6809 [99.65%]
aggregator_0	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_0_1	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1_1	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2_1	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3_1	1644 [0.18%]	0 [0.00%]	3968 [0.20%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0	1271 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0_1	1271 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1	1271 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1_1	1271 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2	1272 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2_1	1272 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3	1271 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3_1	1271 [0.14%]	0 [0.00%]	3164 [0.16%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
krnl_sr	27509 [2.98%]	2644 [0.48%]	45112 [2.24%]	92 [5.37%]	0 [0.00%]	24 [0.35%]
krnl_sr_1	6879 [0.75%]	661 [0.12%]	11278 [0.56%]	23 [1.34%]	0 [0.00%]	6 [0.09%]
krnl_sr_2	6875 [0.74%]	661 [0.12%]	11278 [0.56%]	23 [1.34%]	0 [0.00%]	6 [0.09%]
krnl_sr_3	6877 [0.75%]	661 [0.12%]	11278 [0.56%]	23 [1.34%]	0 [0.00%]	6 [0.09%]
krnl_sr_4	6878 [0.75%]	661 [0.12%]	11278 [0.56%]	23 [1.34%]	0 [0.00%]	6 [0.09%]

Figure 5-3: Resource usage report for 4 intra-node and 2 inter-node (4 lanes) ports @256 bit for U200 card

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	211485 [16.22%]	37670 [6.27%]	268898 [10.31%]	347 [17.21%]	0 [0.00%]	4 [0.04%]
User Budget	1092195 [100.00%]	563290 [100.00%]	2338462 [100.00%]	1669 [100.00%]	960 [100.00%]	9020 [100.00%]
Used Resources	38298 [3.51%]	2640 [0.47%]	67665 [2.89%]	92 [5.51%]	0 [0.00%]	24 [0.27%]
Unused Resources	1053897 [96.49%]	560650 [99.53%]	2270797 [97.11%]	1577 [94.49%]	960 [100.00%]	8996 [99.73%]
aggregator_0	1538 [0.14%]	0 [0.00%]	3995 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_0_1	1538 [0.14%]	0 [0.00%]	3995 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1	1538 [0.14%]	0 [0.00%]	3994 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_1_1	1538 [0.14%]	0 [0.00%]	3994 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2	1539 [0.14%]	0 [0.00%]	3995 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_2_1	1539 [0.14%]	0 [0.00%]	3995 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3	1538 [0.14%]	0 [0.00%]	3994 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
aggregator_3_1	1538 [0.14%]	0 [0.00%]	3994 [0.17%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_0_1	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_1_1	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_2_1	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
dispatcher_3_1	1268 [0.12%]	0 [0.00%]	3150 [0.13%]	0 [0.00%]	0 [0.00%]	0 [0.00%]
krnl_sr	27073 [2.48%]	2640 [0.47%]	39087 [1.67%]	92 [5.51%]	0 [0.00%]	24 [0.27%]
krnl_sr_1	6767 [0.62%]	660 [0.12%]	9772 [0.42%]	23 [1.38%]	0 [0.00%]	6 [0.07%]
krnl_sr_2	6769 [0.62%]	660 [0.12%]	9772 [0.42%]	23 [1.38%]	0 [0.00%]	6 [0.07%]
krnl_sr_3	6766 [0.62%]	660 [0.12%]	9772 [0.42%]	23 [1.38%]	0 [0.00%]	6 [0.07%]
krnl_sr_4	6771 [0.62%]	660 [0.12%]	9771 [0.42%]	23 [1.38%]	0 [0.00%]	6 [0.07%]

Figure 5-4: Resource usage report for 4 intra-node and 2 inter-node (4 lanes) ports @256 bit for U280 card

All the synthesis reports show that dispatchers/aggregators occupy a small percentage of the total resources employed (indicated as *Platform* in the figures above).

Furthermore, comparing the results for the same cards, increasing the datapath width from 128 bit to 256 bit causes a very limited increase in resources' occupancy.

As expected, the resource's occupancy is in percentage smaller in the U280 card, but in either cases the occupancy is low, allowing the implementation of Communication_IP with more intra-node ports, possibly adding new features, and much more complex HLS kernels compared to those used in the performance test design.

6 Conclusions

In this deliverable we described the TEXTAROSSA Communication IP, showing in detail the implementation of a new inter-node port based on the Xilinx® 10G/25G High Speed Ethernet Subsystem and able to support the UDP transport protocol.

Simulation and test of the ethernet port demonstrated the correct behaviour of the port and of its interconnection with the Routing_IP, both for internal datapath equal to 128 bit and 256 bit.

Furthermore, we described changes made in the new version of the Communication_IP: in order to increase bandwidth and lower latency we increased internal datapath from 128 to 256 bit, logic clock from 100 MHz to 200 MHz, and number of channel's lane from 2 to 4.

Synthetic tests, developed to validate the design and assess its performance, have shown significant improvements of the new Communication_IP's performance in terms of bandwidth and latency achieved in both U200 and U280 cards.

7 References

[1] *Corundum: An open-source 100-Gbps NIC*. Forencich, A.; Snoeren, A.C.; Porter, G.; Papen, G. In Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 3–5 May 2020; pp. 38–46.

[2] *VCSN: Virtual Circuit-Switching Network for Flexible and Simple-to-Operate Communication in HPC FPGA Cluster.* Tomohiro Ueno and Kentaro Sano. ACM Trans. Reconfigurable Technol. Syst. 16, 2, Article 25 (June 2023), 32 pages. https://doi.org/10.1145/3579848

[3] *EasyNet: 100 Gbps Network for HLS*. He, Z.; Korolija, D.; Alonso, G. In Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL 2021), Dresden, Germany, 30 August 30–3 September 2021.

[4] *Comparison of Direct and Indirect Networks for High-Performance FPGA Clusters.* F. Rincon et al. (Eds.): ARC 2020, LNCS 12083, pp. 314–329, 2020. https://doi.org/10.1007/978-3-030-44534-8_2

[5] *A custom interconnection multi-FPGA framework for distributed processing applications. C. Salazar-García et al.*, 2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI), Porto Alegre, Brazil, 2022, pp. 1-6, doi: 10.1109/SBCCI55532.2022.9893238.

[6] A New Computer Communication Switching Technique. P. Kermani and L. Kleinrock, Virtual Cut-Through: Comput. Networks 3 (1979) 267.

[7] Deadlock-free message routing in multiprocessor interconnection. Seitz, W. J. Dally, and C. L. 1987. 5: Computers, IEEE Transactions on, 1987, Vol. C.36, p. 547–553.

[8] APEnet+ 34 Gbps data transmission system and custom transmission logic. R. Ammendola, A. Biagioni, O. Frezza, A. Lonardo, F. Lo Cicero, P. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, P. Vicini J. Instrum., 8 (12) (2013), p. C12022, 10.1088/1748-0221/8/12/C12022

[9] *APEIRON: composing smart TDAQ systems for high energy physics experiments*. R. Ammendola, A. Biagioni, C. Chiarini, A. Ciardiello, P. Cretaro, O. Frezza, F. Lo Cicero, A. Lonardo, M. Martinelli, P. S. Paolucci, C. Rossi, F. Simula, M. Turisini, P. Vicini, Under review in Journal of Physics: Conference Series (ACAT 2022), arXiv:2307.01009 [cs.DC]

[10] 10G/25G High Speed Ethernet Subsystem Product Guide (PG210) https://docs.xilinx.com/r/en-US/pg210-25g-ethernet/Product-Specification?tocId=7GkMtn4e6VqMZW3Klq7QhA.

Appendix A. Relevant source codes

Bandwidth test host pseudocode

```
device.load_xclbin(bitstream);
Allocate_recv_buffer(device, buf_size);
Allocate_send_buffer(device, packet_size);
Fill_send_buffer();
Send_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
switch.write_register(auto-toggle reset);
kswitch.write_register(local_coord);
(only for localloop test):  //kswitch.write_register(overwrite destination);
kswitch.write_register(threshold);
kswitch.write_register(credit);
If node_sender:
        Run_kernel_receiver(recv_buffer, 1);
gettimeofday(&startTime,NULL);  //start time measurement
        run_kernel_sender (receiver_coord, npackets, packet_size, send_buffer);
        ksender_run.wait();
        kreceiver_run.wait();
gettimeofday(&endTime,NULL); //stoptime measurement
elapsedTime = elapsed(startTime,endTime);
        BW = (npackets*packet_size)/elapsedTime);
 If node_receiver:
Run_kernel_receiver(recv_buffer, npackets);
kreceiver_run.wait();
recv_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
        Run_kernel_sender(sender_coord, 1, 16, send_buffer); //send back 1 packet of size 16B
        ksender_run.wait();
```

Bandwith test "kernel sender" pseudocode (example for DDR test)

```
int nword = packet_size / sizeof(word_t);
Foreach (packet){
        Header = Fill_header;
        Hdr_fifo_out.write(Header);
        foreach (word) {
        data_fifo_out.write(data_word);
}
        Footer = fill_footer()
Hdr_fifo_out.write(footer);
}
```

Bandwith test "kernel receiver" pseudocode (example for DDR test)

```
Foreach (packet){
hdr_fifo_in.read(hdr);
len = hdr.packet_size;
N_words = len/sizeof(word)
        Foreach(word in N_words){
                word[j] = data_fifo_in.read();
}
        header_fifo_in.read(footer)
}
```

Latency test host pseudocode

```
device.load_xclbin(bitstream);
If !bram_usage:
        Allocate_recv_buffer(device, buf_size);
        Allocate_send_buffer(device, packet_size);
        Fill_send_buffer();
        Send_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
switch.write_register(auto-toggle reset);
kswitch.write_register(local_coord);
kswitch.write_register(threshold);
kswitch.write_register(credit);
If initiator FPGA:
gettimeofday(&startTime,NULL);  //start time measurement
        run_kernel_sender_receiver (destination_coord, npackets, packet_size, send_buffer, recv_buffer,
                                    bram_usage);
        ksender_receiver_run.wait();
        If !bram_usage:
                recv_buffer.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

gettimeofday(&endTime,NULL); //stoptime measurement
elapsedTime = elapsed(startTime,endTime);
Latency = (elapsedTime/2)/npackets;
```

Latency test "kernel sender_receiver" (krnl_sr) pseudocode

```
Foreach (packet){
        If bram_usage:
                memory_in = local_BRAM_buffer_in;
                memory_out = local_BRAM_buffer_out;
```

*send(memory_in, packet_size, **coord**, task_id, ch_id, data_fifo_out); //Communication Library*
receive(ch_id, memory_out, data_fifo_in);
}

Latency test "kernel pipe" (krnl_pipe) pseudocode

Foreach (packet){
receive(ch_id, local_memory, data_fifo_in); //Communication Library APIs
*send(local_memory, packet_size, **coord**, task_id, ch_id, data_fifo_out);*
}