Design and Implementation of the Send Part of an Advanced RDMA Engine

Xirouchakis Pantelis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete

School of Sciences and Engineering Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Manolis G.H. Katevenis*

Thesis Co-*Advisor Dr. Nikolaos Chrysos*

This work has been performed at and was supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Computer Architecture and VLSI Systems (CARV) Laboratory, within the ExaNeSt project, funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 671553.

March 2019 (Published in March 2020)

University of Crete

Computer Science Department

Design and Implementation of the Send Part of an Advanced RDMA Engine

Thesis submitted by
Xirouchakis Pantelis
in partial fulfillment of the
requirements for the Masters' of
Science degree in Computer Science

THESIS APPROVAL

Author: _____
Xirouchakis Pantelis

Committee approvals: _____
Manolis G.H. Katevenis

Professor, Thesis Supervisor

_____

Angelos Bilas

Professor, Committee Member

_____

Polyvios Pratikakis

Assistant Professor, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, March 2019(Published in March 2020)

Design and Implementation of the Send Part of an Advanced RDMA Engine

Abstract

In High Performance Computing (HPC), low latency communication between remote processes is crucial to application performance. InfiniBand and other off-the-shelf networks can reduce the latency but require special and costly network interface cards, which are loosely coupled with the CPU. In this work, we describe the design and implementation of an advanced RDMA engine developed within the ExaNeSt EU project, which has a number of advantages over Infinibad: i) We segment RDMA transfers in blocks, and support block-level multipathing of RDMA transfers on a per-block basis. ii) We perform selective end-to-end retransmissions. iii) We do not need to pin the regions of RDMA transfers in memory, while at the same time we support accessing the full virtual address space of processes, using the ARM SMMU. Additionally, we provide a number of virtual channels able to work simultaneously with many outstanding transfers. Our advanced RMDA engine is designed to support multi-pathing in order to be able to utilize the rich parallel links found in HPC networks. In this work, we describe the hardware implementation of the RDMA engine on the Zynq Ultrascale+. The hardware design has been optimized to meet timing requirements of up to 200 MHz while consuming little resources, leaving plenty of space to be used by accelerators. We have also designed and integrated the interconnect required, as well as the Network Interface (NI) in order to utilize the large Global Virtual Address Space (GVAS) provided by our hardware prototype. We have implemented our advanced RDMA on multiple interconnected FPGAs and have run HPC benchmarks and applications in order to verify and evaluate our design. The results show great improvement over 10G Ethernet, as well as over our previous RDMA implementations. Finally, our RDMA has been designed to easily accommodate many more features, such as congestion management.

Σχεδίαση και Κατασκευή του Κομματιού Αποστολής μίας Προηγμένης Μηχανής Άμεσης Προσπέλασης Μνήμης

Περίληψη

Στις εφαρμογές που απαιτούνε υπολογιστές υψηλών επιδόσεων, η χαμηλή καθυστέρηση επικοινωνίας ανάμεσα σε απομακρυσμένους κόμβους είναι καίριας σημασίας για την απόδοση των εφαρμογών. Το InfiniBand και άλλες έτοιμες επιλογές δικτύων μπορούν να μειώσουν αυτήν την καθυστέρηση αλλά απαιτούνε εξειδικευμένες και ακριβές κάρτες δικτύου που στη γενική περίπτωση είναι απομακρυσμένες από τον επεξεργαστή. Σε αυτήν την εργασία, θα περιγράψουμε τη σχεδίαση και κατασκευή μίας προηγμένης Μηχανής Άμεσης Προσπέλασης Μνήμης (RDMA) την οποία κατασκευάσαμε στο ITE μέσα στα πλαίσια του ευρωπαϊκού έργου ExaNeSt, η οποία υπερτερεί σε πολλά σημεία σε σχέση με το InfiniBand. i) Κόβουμε τις μεγάλες μεταφορές σε πολλές μικρότερες, έτσι ώστε να χρησιμοποιούμε ταυτόχρονα πολλές διαδρομές μέσα στο δίκτυο (multi-pathing) σε επίπεδο υπό-μεταφορων. ii) Υποστηρίζουμε αναμεταδόσεις, σε επίπεδο υπό-μεταφορών. iii) Χρησιμοποιώντας την μονάδα εικονικής μετάφρασης περιφερικών (SMMU) της ARM, δεν χρειάζεται να καρφιτσώνουμε περιοχές μνήμης ενώ παράλληλα έχουμε πρόσβαση σε όλη την εικονική μνήμη του συστήματος. Επιπρόσθετα παρέχουμε έναν αριθμό από εικονικά κανάλια τα οποία είναι σε θέση να δουλεύουν ταυτόχρονα έχοντας χιλιάδες εκκρεμείς μεταφορές. Η προηγμένη Μηχανή Άμεσης Προσπέλασης Μνήμης μας έχει σχεδιαστεί ώστε να μπορεί να υποστηρίξει μεταφορές από πολλαπλά μονοπάτια έτσι ώστε να είναι σε θέση να εκμεταλλευτεί τα πλούσια σε παράλληλα μονοπάτια δίκτυα από τα οποία αποτελούνται οι μοντέρνοι υπολογιστές υψηλών επιδόσεων. Σε αυτήν την εργασία παρουσιάζουμε την κατασκευή αυτής της RDMA στο Zynq Ultrascale+ MPSoC. Το υλικό έχει σχεδιαστεί έτσι ώστε να μπορεί να δουλέψει σε ταχύτητες έως και 200MHz, έχοντας καθυστερήσεις τόσο μικρές όσο ένα μίκρο-δευτερόλεπτο, ενώ παράλληλα καταναλώνει ελάχιστους πόρους, αφήνοντας αρκετό χώρο να χρησιμοποιηθεί από άλλες μορφές επιταχυντών. Επίσης, σχεδιάσαμε και ενώσαμε έναν καινούργιο μεταγωγέα πακέτων καθώς και την διεπαφή δικτύου που χρειάζεται έτσι ώστε να εκμεταλλευτούμε τη μεγάλη εικονική μνήμη που παρέχεται από το πρωτότυπο μας. Εφαρμόσαμε την RDMA μας σε πολλαπλές συνδεδεμένες μεταξύ τους FPGAs και τρέξαμε διάφορα προγράμματα αναφοράς, έτσι ώστε να μπορέσουμε να αξιολογήσουμε τις επιδόσεις της κατασκευής μας. Τα αποτελέσματα δείχνουνε τεράστια βελτίωση έναντι στο κλασικό 10G Ethernet καθώς και προηγούμενες RDMA μηχανές μας.

Acknowledgments

I would like to thank my Advisor Dr. Nikos Chrysos for his guidance and support during my studies and this work. I would also like to thank my Supervisor, Professor Manolis GH Katevenis for his overall assistance and interesting discussions all these years. Furthermore, I would like to thank Professor Angelos Bilas and Professor Polyvios Pratikakis for being in the committee for the evaluation of this work. Special thanks to Nikos Dimou and Fabien Chaix that helped with the writing of my master thesis.

I need to also express my appreciativeness to Konstantinos Harteros for training and introducing me to the world of hardware design. Finally, I would also like to thank all the members of the CARV Laboratory team for their support and collaboration.

Finally, I would like to thank the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), for the funding of this work, which was done for the ExaNeSt project, funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 671553.

Contents

List of Tables

List of Figures

1 Introduction

Large HPC systems rely on efficient interconnects to accommodate a constantly increasing number of end-nodes, while offering low latency and high bandwidth communication among the end nodes, independent of their spatial orientation. At the same time, ongoing efforts from research and industry aims at replacing the power hungry, high-end servers of today with simpler, RISC-like servers, possibly tightly coupled with accelerators, in order to reduce the energy consumption, the system cost and allow flexibly tailoring new system to new workload requirements. Along this direction, the ExaNeSt EU-funded project develops and prototypes a system composed of ARM-based processors, tightly coupled with FPGAs. Traditional end-host network stacks, like Ethernet (TCP/IP) networks, greatly limit the extent to which applications can benefit from the high bandwidth and the low-latency of the hardware interconnect. In addition, these systems tend to consume precious processor cycles to serve the I/O path. For this reason, HPC usually use (custom or InfiniBand-based) RDMA (remote direct memory access) interconnects, which offload several layers of the network, and, compared to traditional software transports, greatly improves the throughput and the latency performance of communications. In our system, we leverage the FPGA to implement a custom low-latency RDMA-capable interconnect that connects computing nodes with each other, as well as with memories and fast, non-volatile storage devices. The core of our RDMA interconnect is implemented in network interface (hardware) engines that offload the software transport by offering reliable communication services, and allowing processes to benefit from hardware-class latencies and throughput.

1.1 RDMA overview

RDMA (Remote Direct Memory Access) allows one process to directly access the memory of a remote process with very high throughput and low latency, by using specialized NIC (network interface chip) that minimizes the CPU overhead. Traditionally such networks have been used by high performance computing applications, with somewhat custom hardware networks that are hard to be programed and were usually application specific. Datacenters on the other hand kept using traditional commodity hardware and TCP/IP for their networking. Lately however, as fast and reliable such networks come into the market (i.e InfiniBand , RoCE, iWarp), many datacenters decide to use RDMA instead of conventional TCP/IP. Meanwhile, there is significant ongoing research to build improved RDMA networks that can perform even better and provide extra functions, such as congestion management and multi-pathing.

The inherent inefficiency that RDMA networks try to solve is the fact that, in traditional networks, the kernel is invoked in network transfers, increasing the latency and the processing overhead, and that data have to be copied on many intermediate buffers before finally arriving on the destination, increasing both latency and power consumption. Along these lines, we describe in this work the development of a zero-copy, user-level initiated RDMA engine used in the ExaNeSt project prototype. In our system, the goal is to achieve the ideal RDMA operations depicted in Figure 1-1. For this to work, the data must be addressed at the source and the destination using virtual addresses. Therefore, the network must provide a mechanism to translate virtual to physical addresses.

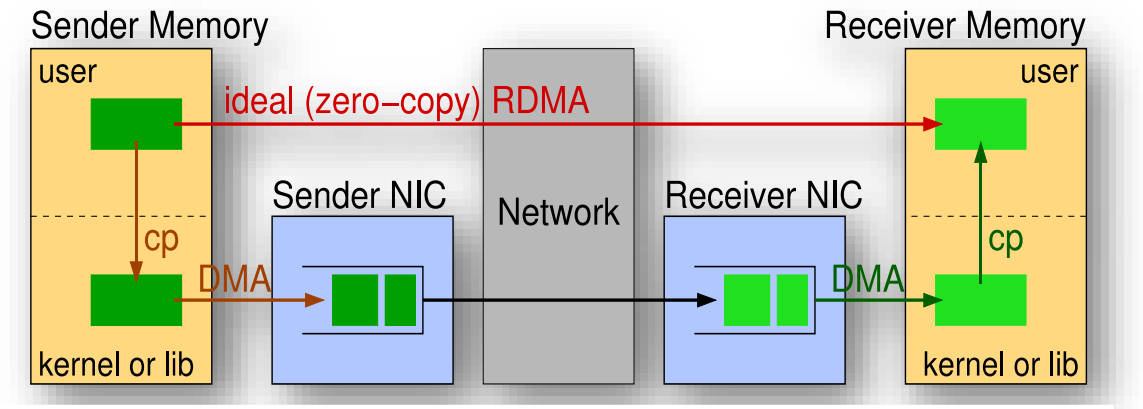


Figure 1-1 Ideal Zero Copy RDMA transfer

Different networks approach this problem in different ways. InfiniBand for instance tackles this issue by keeping a (network) TLB on the NI, and forcing the applications to pin its pages pertaining to transfers. In this way, there is no need to copy data to intermediate (I/O) buffers inside the kernel, which avoids copying of the buffers but still adds latency due to the kernel involvement for pining. In our protocol, we obviate this step, allowing network memory accesses to produce page-faults, which can trigger network retransmissions.

To this day, RDMA has mainly been used as a single-path transport, which is prone to failures and falls short to utilize the rich parallel paths found on HPC/Datacenter systems. Attempts to implement multi-pathing RDMA have been performed but with no great result, as they usually require excessive amounts of metadata, greatly increasing the cost of the NIC (Network Interface Chip). Moreover little-to-no care has been given to handling congestion situations that arise in the network. Infiniband does feature a congestion control mechanism, which, however is very hard to tune. Coupling congestion management with multi-path routing would open new possibilities for routing algorithms and can increase the effective bandwidth of RDMA even further.

In addition, the resiliency features typically implemented by RDMA interconnects are focusing on functionality instead of performance, in order to economize silicon area. InfiniBand, for instance, provides a very crude end-to-end resiliency scheme, with per-packet acknowledgements and full message retry. This results in inefficient use of bandwidth, large flow completion times, and is also prone to livelocks [1]. The proposed RDMA protocol and its implementation coalesces acknowledgements (one per block) and performs selective retransmissions, greatly minimizing the Flow Completion Time (FCT) and the network bandwidth overhead.

For large networks with millions of nodes, one node may wish to perform RDMA transfers to potentially all the other nodes, or to a large subset of them, at the same time. We do not want to necessarily serialize these transfers in time. Instead, being able to multiplex many of these transfers on the ingress network path(s), on a per-packet basis, can improve the network utilization, when some of these transfers are congested. In order to do so, the RDMA engine has to provide many channels to its users, which increases the cost of bookkeeping and the complexity of the control unit that supervises their parallel transfers.

At the same time, as silicon manufacturing techniques improve, and the number of transistors inside a die increases, manufacturers have the ability to add more peripherals within the die such as accelerators and smaller, real-time co-processor units (e.g. following the big-little architecture). In our work, we leverage one such co-processor available in the Zynq Ultrascale+ FPGA to assist in RDMA operations. The co-processor is *responsible for block-based operations*, which are infrequent enough, thus not strictly requiring hardware-speed pipelines. We build the hardware part of the RDMA engine on the programmable logic segment of the FPGA that resides in the same die with the ARM CPUs, inside the programming system (PS) for the Zynq Ultrascale+ ARM. This very tightly coupled network interface further decreases latency, as we do not need to cross for example the PCI bus, as e.g. in Intel-based InfiniBand networks.

The RDMA protocol we have implemented in this thesis is similar in spirit with InfiniBand and RDMA over Converged Ethernet (RoCE) networks, but it differs in the following ways:

- We segment RDMA transfers in blocks, and support *block-level multi-pathing* of RDMA transfers on a per-block basis.
- We perform *selective end-to-end re-transmissions*, whereas InfiniBand networks typically perform *Go-back-N* on the end-to-end path, which may involve retrying full transfers.
- We do not need to pin the regions of RDMA transfers in memory, while at the same time accessing the full virtual address space of processes, using ARM's SMMU.
- We coalesce RDMA responses (1 response per segment), whereas InfiniBand sends one Response per packet.
- Our RDMA engine can be coupled with accurate congestion control, by spacing of packets at the network sources (rate limiting transfers), whereas InfiniBand is still in search for an appropriate congestion control method.
- We provide advanced scheduling techniques on NI (Network Interface) egress path, which can easily be modified in software.
- We provide hooks to users for controlling transfers order, and influence routing.
- We designed and implemented a novel and efficient mechanism for fast completion notifications at the receiver described on Appendix A.

1.2 Contributions

This thesis has contributed to the design of a new Remote Direct Memory Access (RDMA) engine within the ExaNeSt project, suitable for user-level initiated zero-copy transfers in a system of ARM cores, and tightly coupled Network Interface (implemented in FPGA) nodes working on a global virtual address space (GVAS). Our RDMA supports advanced quality-of-service (QoS) and resiliency features, such as multi-pathing, fast notifications, and selective retransmissions, which we report in this thesis, together with performance evaluation results. In our ongoing work, we are adding congestion management support.

The design of the new RDMA engine is split into a *software-programmable part* and a *hardware part*. The hardware part is the core of this thesis and is implemented and tested in the Programable logic of the FPGA. The design has been implemented on Zynq Ultrascale+ Xilinx MPSoC and is now functional, running on the ExaNeSt-project prototype. Our RDMA design offers low-latency/ high-bandwidth user-level read/write transfers.

The author of this thesis has designed and built the hardware implementation of the RDMA send unit (ExaDMA). More specifically:

- **ExaDMA Send Unit**: The hardware core of the RDMA engine which is responsible for executing the RDMA transfers. This block has a large array of pending transactions and a round robin scheduler that iterates amongst them. For each transaction, it reads the data directly from the applications memory, by accessing through the SMMU of the system. It implements output buffering and has one output buffer per output link of the FPGA. Among other things, it is tasked with performing all the shifting required (byte level) in order for the data being send to be aligned at destination. It is designed to be fully compatible with the software running on the Real-Time processor (software-hardware co-design) in order to maximize the effectiveness of the RDMA.
- **ExaNet Switch:** This is a 16x16 buffer-less ExaNet switch, used to connect each component of the ExaNet network. The routing algorithm uses both the destination coordinates and the destination address of the packets. For simplicity and flexibility, the routing can be configured in the Vivado block diagram, without needing to change source-code, repackage and redistribute.
- **ExAurora:** This is a block used as an interface that connects the ExaNet Datapath, with the transceiver's AXI-STREAM Datapath. It implements input and output cross-clock domain FIFOS, and is also responsible for the flow control of the Link.
- **Intra-node interconnect:** The network design and implementation of the QFDB (Quad FPGA Daughter Board) interconnect, which connects the network interfaces and the accelerators of four (4) intra-node Ultrascale+ interconnected FPGAs with each other and the external gateway router.
- **Platform Integration and verification:** The implementation on the Zynq Ultrascale+ MPSoC of all the ExaNet blocks as well as the verification and debugging. Given the great amount of different hardware blocks and functionality that the ExaNet network has, the interaction between various agents proved to be very challenging requiring many hours of debugging.

2 ExaNeSt Platform

2.1 The ExaNeSt project

ExaNeSt develops, evaluates, and prototypes the physical platform and architectural solution for a unified Communication and Storage Interconnect and the physical rack and environmental structures required to deliver European Exascale Systems. The consortium brings technology, skills, and knowledge across the entire value chain from computing IP to packaging and system deployment; and from operating systems, storage, and communication to HPC with big data management, algorithms, applications, and frameworks. Building on a decade of advanced R&D, ExaNeSt will deliver the solution that can support exascale deployment in the follow-up industrial commercialization phases. Using direction from the ETP4HPC roadmap and soon-available high density and efficiency compute, we will model, simulate, and validate through prototype, a system with:

- High throughput, low latency connectivity, suitable for exascale-level compute, their storage, and I/O, with congestion mitigation, QoS guarantees, and resilience.
- Support for distributed storage located with the compute elements providing low latency that non-volatile memories require, while reducing energy, complexity, and costs.
- Support for task-to-data SW locality models to ensure minimum data communication energy overheads and property maintenance in databases.
- Hyper-density system integration scheme that will develop a modular, commercial, European-sourced advanced cooling system for exascale in ~200 racks while maintaining reliability and cost of ownership.
- The platform management scheme for big-data I/O to this resilient, unified distributed storage compute architecture.
- Demonstrate the applicability of the platform for the complete spectrum of Big Data applications, e.g. from HPC simulations to Business Intelligence support.

All aspects have be steered and validated with the first-hand experience of HPC applications and experts, through kernel turning and subsequent data management and application analysis.

2.2 Zynq Ultrascale+ MPSoC

The ExaNeSt design has been implemented on the Xilinx Zynq Ultrascale+ MPSoC devices. These chips contain a Processing System (PS) that consists of four A53 ARMv8 cores operating at 1.333 GHz and two R5 Real time processors operating at 600Mhz. Additionally the MPSoC contains Programmable Logic (PL) part that has 274K LUTS and 912 RAMB32 (SRAM blocks with 32-bit data interface) available for development. Crucial to the project is the connectivity that the PS provides to the resources found at the PL. For this purpose, the PS provides 2 Full AXI-4 Master Cache Coherent interfaces and 2 Slave interfaces.

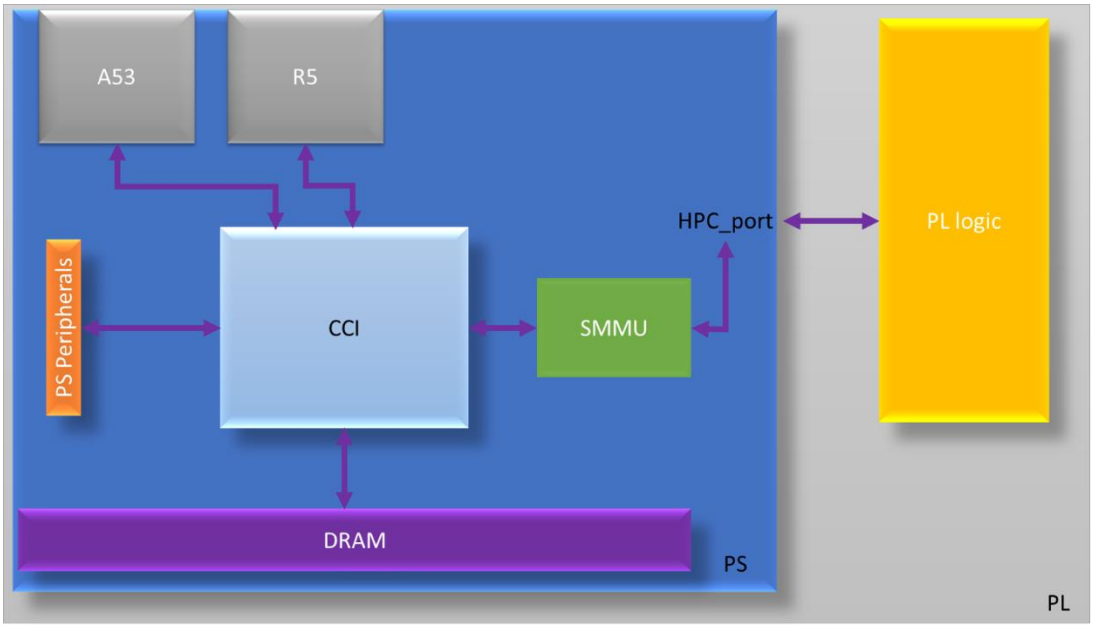


Figure 2-1: Ultrascale+ MPSoC Simplified block diagram

Accesses from and to those ports pass through the CCI (Cache Coherent Interconnect) allowing the PL to do (I/O) coherent accesses to the PS. Additionally, the PS provides an ACE-Lite port (AXI coherent extensions) from which the PL can access directly the L2 cache of the A53 core, greatly reducing the required latency. Finally, the PS provides 6 high throughput, non-coherent AXI-4 Slave ports. A simplified view, as well as the detailed block diagram of the MPSoC can be seen on the Figure 2-1 and Figure 2-2[2].

The prototype consists of up to 112 nodes (will reach up to 250+) all being able to address the same GVAS (Global Virtual Address Space). In this direction, the SMMU (I/O mmu) plays a central role in translating all process-level virtual addresses to physical main memory locations. It features 8 different context banks, where each context bank can hold a TLB cache and a pointer to a page table of a particular process.

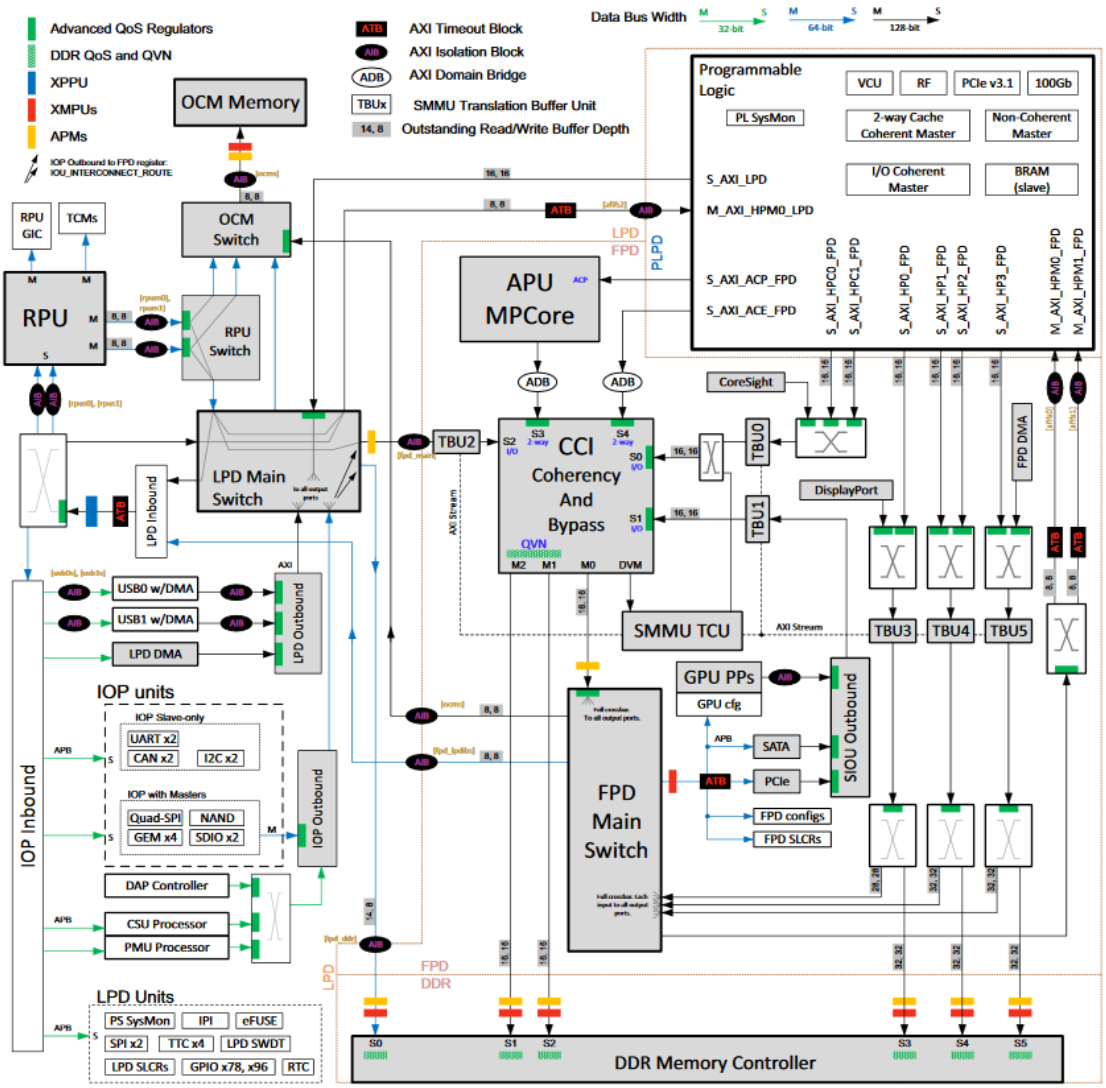


Figure 2-2: Ultrascale+ MPSoC detailed block diagram, Xilinx UG1085

2.3 Hardware Prototype

The basic compute node is called Quad-FPGA Daughterboard (QFDB). It contains four (4) Ultrascale+ MPSoCs connecting each other hardwired with two High Speed Serial Links (HSS) in an all-to-all mesh topology as shown in Figure 2.3. Each FPGA features two 16MB QSPI and 16GB DRAM so that one (1) QFDB aggregates 64 GB of DRAM as well as 512GB SSD storage. Moreover, each QFDB provides a connector with ten (10) bidirectional HSS links (10 x 16Gbit/s = 160Gbit/s = 20GB/s) for high-throughput communication with other devices. Four (4) of those links are used to connect neighboring QFDBs hosted on the Blade. The remaining six (6) HSS links are attached to the external link cages (SFP+), mainly for connection with other blades, e.g., within the same Chassis.

Four (4) QFDBS are connected to a mezzanine to form one Blade. The mezzanine provides the QFDBS with 1G Ethernet, and connects the network FPGA of each QFDB in a torus topology of max dimension 4.

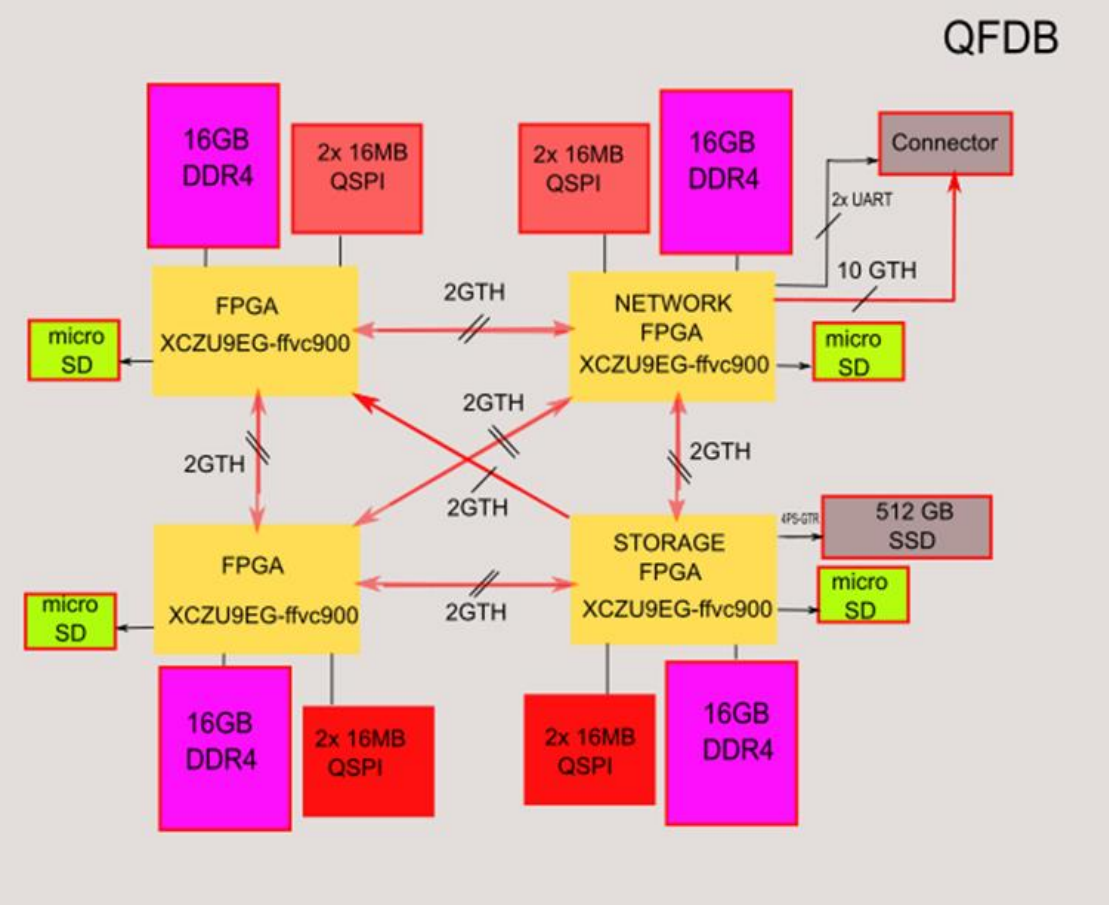


Figure 2-3: Quad FPGA daughter board overview

3 ExaNet Network

The ExaNet network uses a custom header/payload/footer packet-based protocol, developed within ExaNeSt for inter-processor (or accelerator) payload traffic, and for network interface (NI) end-to-end control messages. It supports the basic UNIMEM operations[3] such as owner-only page caching and shared-memory semantics, while extending UNIMEM with address translation at the destination, advanced protection for cluster virtualization (HPC applications co-scheduling), and protocol security and reliability features.

The ExaNet packet format was preferred against AXI, the protocol used by ARM processors to access the main memory and peripheral devices, because AXI is suitable for on-chip systems, but does not scale well to system-level HPC interconnects (because of many parallel channels used that increase the complexity in handshaking therefore utilization).

Inside the FPGAs, the ExaNet packets is carried by a 128-bit datapath (equal to the maximum PS ←→ PL AXI interface width), and currently runs at 150 MHz. Thus, each ExaNet on-chip link offers a throughput of 19.2 Gb/s, which can easily saturate the 10 Gb/s High-Speed Serial (HSS) links of the platform, while offering a significant internal speedup inside the switches.

Thus, the ExaNet protocol supports 80-bit global virtual addresses, split into a 16-bit PDID field discussed above, a 22-bit destination coordinates field used to identify a node in the cluster, and a 42-bit virtual memory address to locate an object inside an FPGA and its local DRAM Figure 3-1.

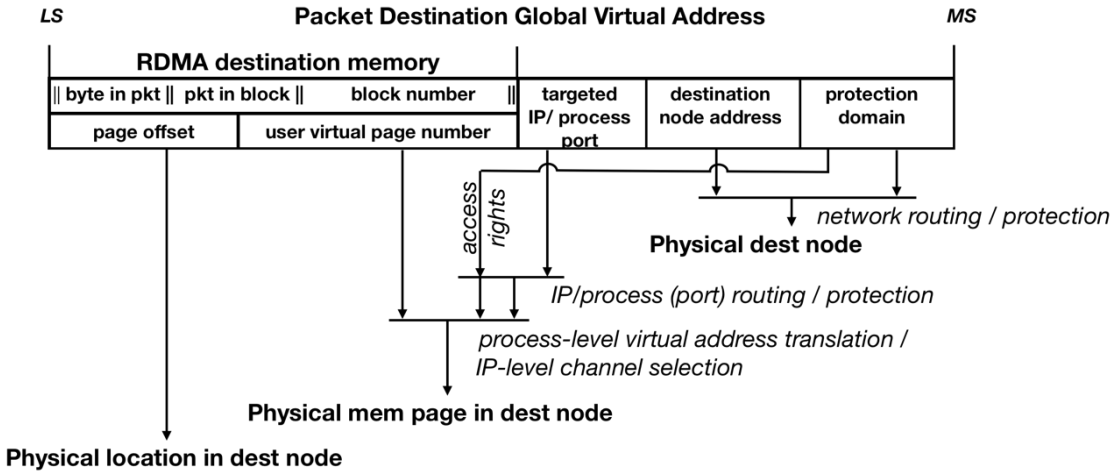


Figure 3-1: ExaNet GVAS breakdown.

The ExaNet packet can contain: 1) header only packets (16Byes), 2) zero-payload (header-footer) packets (32Bytes), and 3) variable payload (header-N*payload-footer) packets (48 – 288 Bytes) with a max payload of 256Bytes. ExaNet packets are covered by:

- *16-bit error detection code*, protecting the fields in the ExaNet packet headers,

- *8-bit footer CRC,* protecting critical fields in the ExaNet packet footer
- *32-bit payload CRC*, protecting the payload of ExaNet packets (Figure 3-2).

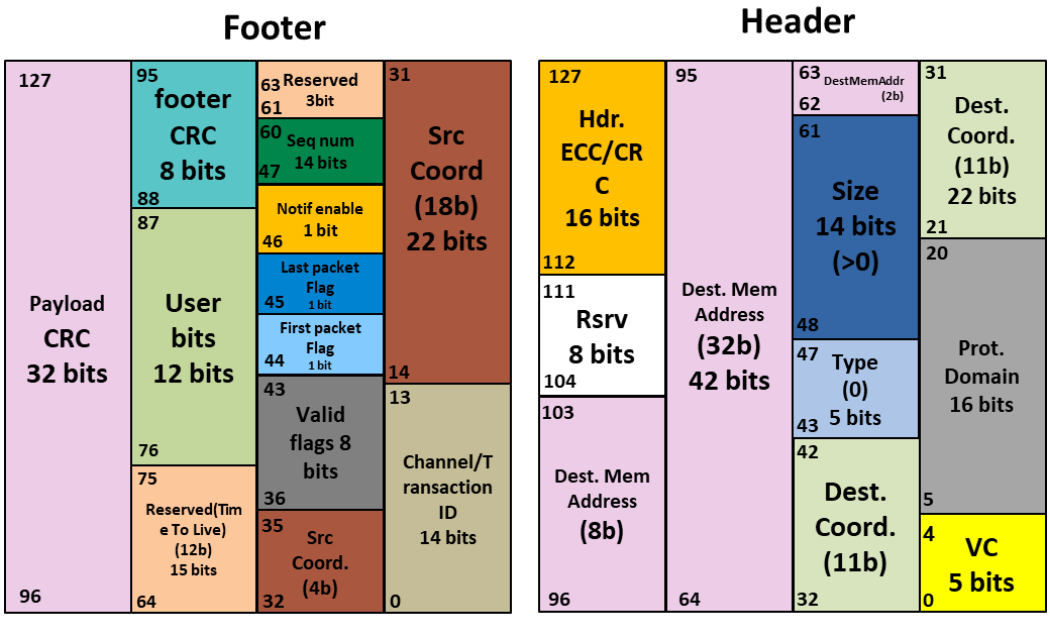


Figure 3-2: ExaNet RDMA packet header format

Within ExaNeSt, we have defined the following ExaNet packet types. These packets are used to carry packets with payload (RDMA and shared-memory primitives) as well as NI control information over the ExaNet fabric. Here is a list of our current packet types:

1. **Remote Write Data Packet**
   - from ExaNet RDMA Send or ExaNet Packetizer to ExaNet RDMA Receive or ExaNet Mailbox
2. **Remote Read Request**
   - from ExaNet Packetizer to ExaNet RT Mbox
3. **DMA Control Packet for Transaction Completion Notification**
   - from ExaNet RDMA Sender to ExaNet RDMA Receiver
4. **DMA Transaction Completion Notification Packet**
   - from ExaNet RDMA Receive to ExaNet Mailbox or ExaNet RDMA Receive
5. **zDMA Remote Read Request Packet**
   - from ExaNet AXI2EXA to ExaNet EXA2AXI
6. **zDMA Remote Write Data Packet**
   - from ExaNet AXI2EXA to ExaNet EXA2AXI
7. **zDMA Remote Read Response Packet**
   - from ExaNet EXA2AXI to ExaNet AXI2EXA
8. **Acknowledgment / negative acknowledgment**
   - Error codes for packet errors and destination memory or NI responses
9. **Release Context Packet (*not yet implemented*)**
   - from ExaNet Packetizer or ExaNet RDMA Send to ExaNet Mailboxes or ExaNet RDMA RX
10. **Flow Rate Packet (not yet implemented)**
    - ExaNet Congestion Control Notification

3.1 ExaNet Network Interface

In ExaNeSt project, we have built a Network Interface that contains:

- A low-latency multi-channel Remote Direct Memory Access (RDMA) engine that leverages both the SMMU for secure memory accesses and the R5 co-processor for book-keeping and handling of the transactions. In our design, we use the System Memory Management Unit (SMMU) in order to translate process virtual addresses to physical memory locations as packets go in and out of memory and avoid copies from user to kernel space, allowing zero-copy, user-initiated RDMA transfers. The memory accesses issued from our RDMA engine pass through the ARM's Cache-Coherent Interconnect (CCI) thus obviating the need to flush or bypass the caches before triggering RDMA transfers. The network that we use to transfer data between nodes is the ExaNet interconnect, which we describe next.
- A fast multi-channel *packetizer* that allows processes to write small, latency-critical messages to arbitrary positions in a global address space, by writing a set of registers in the network interface, which is implemented in the programmable logic. We have also implemented fast *mailboxes* that write messages to per-process queues hosted in the L2 cache and DRAM memory hierarchy of the ARM subsystem. The packetizer and mailbox primitives are typically used together, allowing user or system processes running on CPUs or acceleration units to exchange fast messages that go to mailboxes and memory, allowing partitioned global address space applications (PGAS) to run on the prototype.
- A low-latency network interface switch, which also connects the FPGAs within the same QFDB, with each other and also with the QFDB-level gateway, which is implemented by the APENET router[4] in the network FPGA.

The NI interface additionally contains IPs for Ethernet-based communication and additional functional blocks.

3.2 The Network Interface switches and QFDB-Level Interconnect

Prior to the quad-FPGA QFDB boards the ExaNeSt interconnect designs and FPGAs firmware were tested on (single-FPGA) Trenz-Based boards, in 2D Torus topologies. In these designs, every FPGA of the prototype had the NI from FORTH and the APErouter mentioned earlier. Going from single-FPGA nodes to four-FPGA ones, we had two options. One was to treat every FPGA as a single node of the Torus topology, and each QFDB as a dimension of the Torus. However, for the reasons listed below, we opted to take a different path:

- The QFDB internally has more physical HSS links, and can support topologies with smaller diameter and higher bisection bandwidth than a ring of (one dimension of a Torus). The topology that we eventually chose is a *single-hop all-to-all topology* connecting each FPGA with every other FPGA inside the QFDB.
- Using an All-to-All topology inside every QFDB coupled with a Torus Topology to connect QFDBs, results in an interesting hybrid-Torus topology, with no routing deadlocks, as long as we use single-path routing inside the QFDBs, and the more complex APErouter resolves deadlocks using different VCs.
- The hybrid Torus topology does not need to have the Torus-routing capabilities (e.g. an APErouter) in every FPGA of the QFDB, thus saving on complexity and FPGA resources.

The hybrid-Torus topology essentially treats each *QFDB as a single node*, with local SSD, lots of DRAM, computing capacity, and 10 x 16 Gb/s network I/O. This QFDB node needed a new simplified crossbar for intra-node communication.

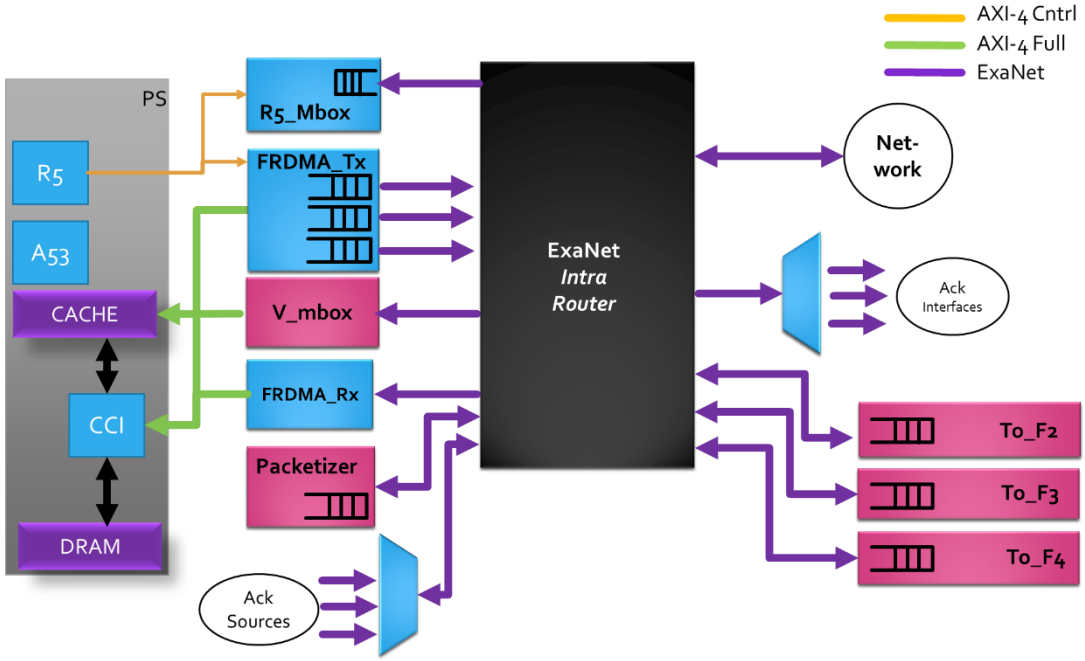


Figure 3-3: ExaNet NI Switch located in FPGAs F1-F4 of the QFDB and its interface to the APErouter (Network), which serves inter-QFDB traffic that is routed through a 2D/3D Torus (on Mezzanine or single-feeder) topology.

3.2.1 Routing inside the QFDB and its impact to firmware development

Routing in NI blocks is common source of bugs and delays. When the first QFDBs became fully-functional, early in 2018, we needed to port the ExaNeSt interconnects from Trenz-based prototypes to QFDB-based ones, while in parallel we were developing and testing new ExaNet packet types in order to improve the resiliency of the ExaNet network (end-to-end acknowledgments for different network interface IPs) and to support new NI-to-NI control messages. In this work, we observed two inter-related obstacles:

- *routing packets to the proper NI block had become a frequent cause of "soft-bugs" as designs needed to support many different ExaNet packet types across many new NI blocks,*
- *changing the routing of (new or old) packet types to NI blocks required a new switch version, which means that engineers had to change their Verilog code.*

Uniform address-based routing for all packet types was needed in order to move on. At the same time, the existing routing of packets and acknowledgment, to NI blocks, being packet-type based, was rather inflexible. For instance, the packetizer block at that time could only send packets and receive acknowledgements from mailboxes, although in principle we had designed the primitive in order to send packets to any memory location. To support this, we needed a *flexible and uniform way to access NI blocks and memory address locations.*

To uniformize accesses across memory and NI peripherals, we decided to *add the destination virtual address field in all ExaNet packet types*. With the new packet format, all packets, even acknowledgements or control messages that target mailboxes or any other NI peripherals, have a virtual destination address field in their header, which starts at the same bit position with the destination virtual memory address of packets that target a specific location in DRAM.

In particular, the *new simplified NI switch which is present in the ExaNet prototype uses the following rules to route packets to DRAM and peripheral devices:*

- If the 3-bit prefix of the destination virtual memory address equals '111', then the packet is targeting a peripheral, which can be indexed using the next 7 bits of the destination address.
- If the 3-bit prefix of destination virtual memory address is not equal '111', then the packet is targeting the DRAM, which we can address using all address bits (after going through translation in the SMMU).

In this way, we consume about 1/8 of the ExaNeSt global memory address space, which is a reasonable trade-off. In practice, we don't even consume that much, because not all 42 bits in the destination virtual address field of ExaNet are translated by the SMMU but only 39 bits

To further homogenize designs, and to simplify platform developments, we needed to have the same design for as many FPGAs in the QFDB as possible in order to reduce the number of firmware designs that we have to implement and test each time that we add a new feature or correct a bug on the platform firmware. In practice, the QFDB Network FPGA, F1, needs a different design, because it also contains the APErouter, and the Ethernet NIC. However, we could still consolidate the designs of FPGA F2, F3, and F4.

At this point, we considered to add software-defined routing that we can modify at runtime on a per FPGA basis and to improve the turn-around latency of creating a new (set of) FPGA firmware that corrects a bug or adds new routing rules. However, with the following two features, which we added into the NI switches inside the QFDB, software-define routing was not yet needed

- **Uniform routing across QFDB FPGAs routing**: we added *dead ports* in the NI crossbar, in order to have the same rules for all FPGA when we route packets inside the FPGA. In practice, the dead port on each FPGA is connected to a loop-back port, so that, for instance, the second inter-FPGA crossbar port is connected to QFDB FPGA F2, regardless on which QFDB FPGA uses this port.

- **Selectable routing ranges at design-integration time:** we added the capability to select the routing to NI ports at FPGA firmware compile time -- which destination virtual address range will map to which NI block -- using the graphical settings of the Vivado design, thus without having to modify the Verilog code.

To simplify the routing, and avoid the need for deadlock avoidance mechanisms, we currently use single path routing inside the QFDB, leaving multipath capabilities using the two links available per FPGA pair inside the QFDB for next versions of the ExaNeSt prototype or for follow-up projects.

3.2.2 ExaNet NI crossbar

To route packets inside the QFDB, we have implemented a NI crossbar switch augmented with simple but effective routing rules to route packets among the QFDB FPGAs and the NI blocks, following the principles described in the previous section. This ExaNet NI-crossbar switch supports a configurable number of uniform ExaNet interfaces, and can be connected to a number of ExaNet agents, such as NI endpoint hardware blocks, and to custom transceivers for inter-FPGA traffic.

ExaNeSt-AURORA transceivers: We currently use AURORA IP from Xilinx to implement reliable communication inside the QFDB. In particular, using the AURORA transceivers from Xilinx, we have implemented basic, but resilient ON-OFF flow control to prevent buffer overflows when sending packets across links.

The ExaNet NI-crossbar is *bufferless crossbar,* with no input or output buffers (and queueing). Given that in the ExaNet NI, all the hardware sender and receiver blocks have some form of input or output buffers, respectively, and that the transceivers also have buffers of their own, in order to *a) save FPGA BRAMs, b) reduce the in-fabric backlog, c) minimize the number of packet copies (saving energy), and d) decrease the in-network queuing delay, we decided not to put additional buffers inside the NI switch/crossbar.*

The main features of the ExaNet NI switch are listed below:

- It offers a cut-through latency of two cycles, thanks to its simple and uniform routing rules.
- It inserts two idle cycles between consecutive packets on the link, thus offering nearly full link efficiency. Given that the datapath will run slightly faster than the links, the switch is able to saturate links nearly for all packet sizes.
- It supports all ExaNet packet sizes (header-only, header-footer, and header-payload-footer).
- We can configure its routing to NI-attached ports at compile time using the Vivado GUI.

In principle, the new NI switch allows us to flexibly connect many peripheral devices, and to define how packets are routed to them, based on the destination memory address field of the packets. In order to reduce the crossbar area, we can merge multiple low-throughput interfaces (e.g. ACKs generated by different NI IPs) on a single ExaNet switch port using special ExaNet multiplexors and demultiplexor

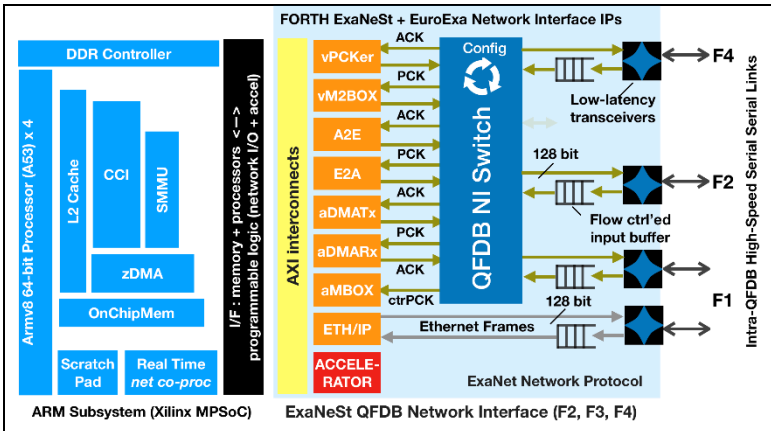
3.2.3 QFDB interconnect and firmware inside the QFDB FPGAs F2-F4

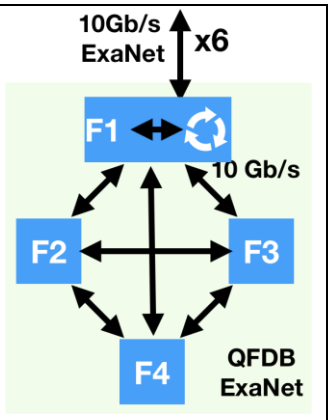
Inside the QFDB, we have designed and implemented a custom *distributed QFDB-internal ExaNet interconnect,* to connect the NI IPs with the ExaNet network, and the FPGAs with each other and the gateway to the Torus QFDB-level network inside the F1. The firmware that we currently use in FPGAs F2-F4 is depicted in figure 3.3. It consists of eight (8) NI blocks, which

are connected to our ExaNet NI switch, and from there, to low-latency transceivers that realize the all-to-all topology within the QFDB.

The firmware used in FPGA F1 additionally hosts the Ethernet NIC from and the unmanaged 10G Ethernet switch. In addition, F1 hosts the router used to connect to other QFDBs, namely APErouter, using a Torus topology.

Routing among FPGAs inside the QFDB and to/from the APErouter: Packets are routed by NI switches based on the *offset* field of the destination coordinates, or to the network FPGA if the X,Y,Z destination coordinates indicate a different QFDB target. Inside the network FPGA, there is another instance of this switch as well as the APErouter board-switch, which are connected through 4 local ports available – see also Figure 3.3. The routing is done such that one such local port of the board board-level switch is reserved for every FPGA in the QFDB.





(a) Firmware for QFDB FPGAs, F2, F3 and F4, i.e. all FPGAs except the network FPGA, F1.

(b) All-to-all topology among QFDB FPGAs. Inside F1, the NI switch is connected to APErouter and from there to the outside world.

Figure 3-4: The ExaNeSt QFDB-based firmware and network topology inside the QFDB.

A simplified view of the network blocks can be seen on Figure 3-5

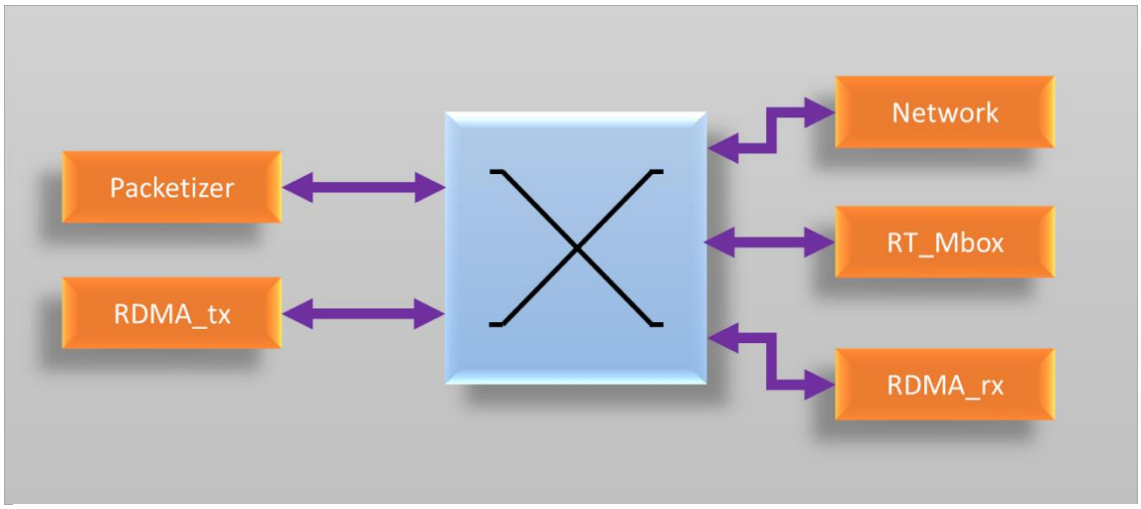


Figure 3-5: RDMA Network Interface simplified overview

The hardware blocks used for RDMA operations in the NI, also shown in Figure 3-5, are a packetizer, a mailbox, a switch transceiver and the RDMA engine send (TX) and receive (RX) modules. Each of the above hardware modules plays a crucial role in the ExaNet network. A more detailed description can be found in Chapter 6.

3.2.4 ExaNet data path protocol

The ExaNet data path consists of the following signals:

- Exa_Data [127:0]
- Exa_Header_Valid and Exa_Header_Ready
- Exa_Payload_Valid and Exa_Payload_Ready
- Exa_Footer_Valid and Exa_Footer_Ready

The “Valid” signals are driven by the transmitter/master of the packet and the ready signals are driven by the receiver/slave of the packet. The master can assert the Valid signal, anytime the appropriate correct data are driven on the Exa_Data bus, but is not allowed to change the value of that data until the appropriate Ready signal has been asserted by the slave.

4 Overview of RDMA Engine

The RDMA engine, is split into a programmable part that currently runs using custom code on the dedicated network interface co-processor running on the ARM Real-Time (Cortex-R5) core, and a hardware part that runs on the programmable logic (PL) of the Xilinx Ultrascale+ MPSoC. The RDMA supports advanced quality-of-service (QoS) and resiliency features that will be described below.

We have implemented the RDMA engine using three main modules, as shown in Figure 4-1.

- The first module is responsible for transfer segmentation and scheduling, as well as for the retransmissions of transfer blocks (segments). These are relatively complex functions but they run on the timescale of many packets (64 packets for 16KB), thus relatively infrequently compared to the processor and the NI clock (and the packet time on a link). This fact allows us to build these functions on the NI co-processor, which we run on the Real-Time (Cortex-R5) of the Xilinx FPGA that resides inside the processing system of the FPGA. Our software implementation decreased the turn-around implementation time of this module, while increasing flexibility, in terms of policies that can be implemented, along the spirit of Software-Defined Networking (SDN).
- The second module, running on special hardware inside the PL, is responsible for executing the transfer segments at the source. It schedules, spaces and transmits packets, reading data from memory through the SMMU, aligning the packets' payload to destination addresses, and sending packets to the interconnect using the custom ExaNet packet format. It can also detect page-faults and stop transmission of such packets. This module is the main focus of this thesis.
- The third hardware module is responsible for bookkeeping the transfer segments at the receiver, monitoring their execution, generating and coalescing the negative/positive acknowledgments, as well as generating our fast completion notifications. It consists of a 16-way set associative cache, that holds the context of each active transaction. Each context has a 64bit bitmap for the packets that have arrived, along with other information required by the NI.

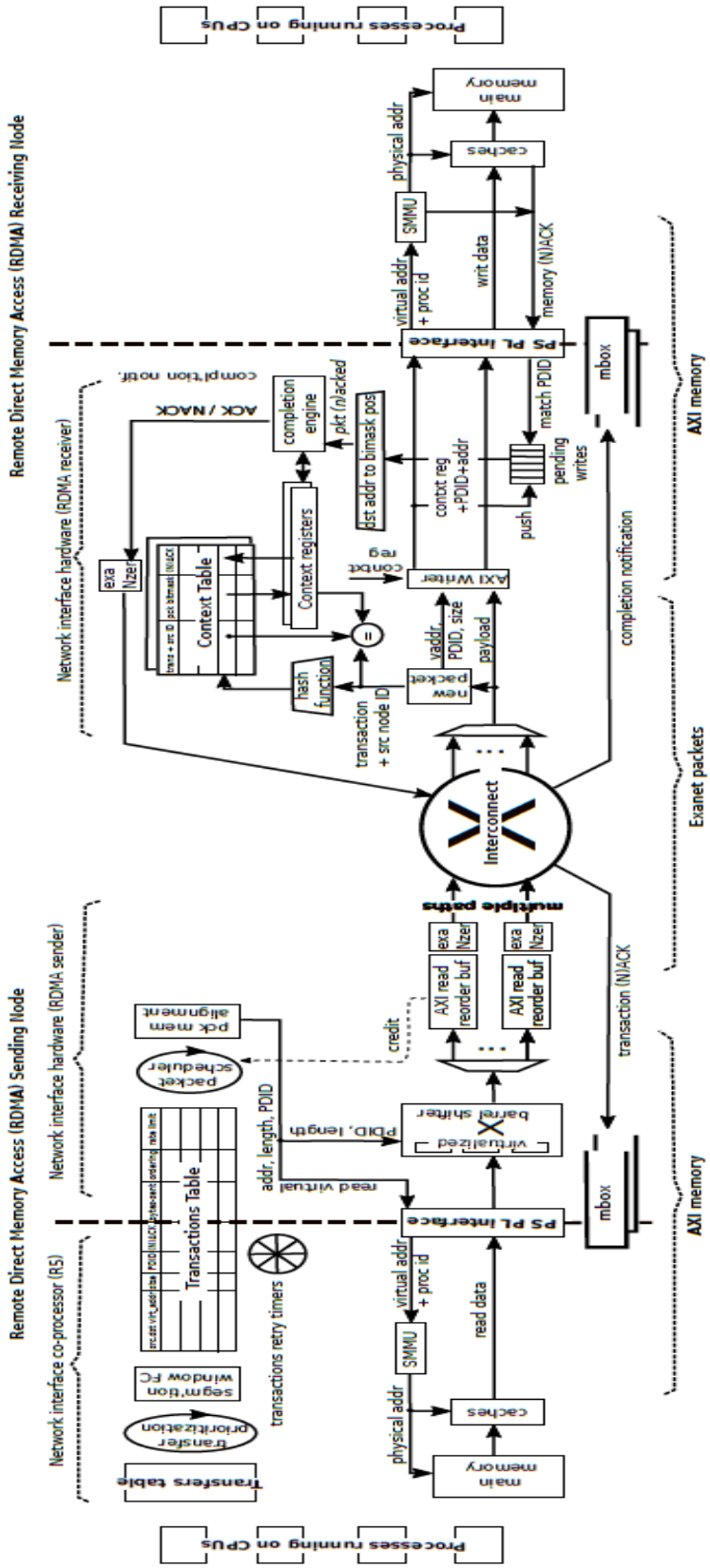


Figure 4-1: Network Interface advanced overview

The smaller entity in the ExaNet network is a packet, which consists of a header a footer and a payload part. The Maximum Transfer Unit (MTU) for this packet is chosen to be 256 bytes since this size has shown to be more efficient at doing network congestion work. Apart from this, smaller MTU also helps keeping the network buffers small, saving utilization space and therefore cost. As mentioned earlier, the receiver of the RDMA holds a bitmap for each active transaction, that shows how many packets have arrived. In combination with the MTU being 256 bytes, this limits the total transfer size of a transaction. We chose 64bits for the context, which results in maximum transaction size of up to 16KB. This means that no transaction should cross 16KB boundaries at destination address, a functionality which is handled by the R5 co-processor. The RDMA receiver has to be able to receive the packets of a transaction delivered out-of-order since the network is built to support multipathing. This means that a bitmap should be utilized to mark down the packets that have arrived. If we chose the maximum transaction size to be 32Kb then the bitmap would have to be 128Bits and so on, meaning that we would have to sacrifice a great amount of resources (RAMB32) in order to achieve smaller and smaller gains. In addition, our RDMA protocol supports retransmissions at transaction level but not at packet level. This means that larger transactions sizes, apart from inherently having more chances to require a retransmission (crc errors etc) they would also require a lot of information to be retransmitted. Taking all those factors in consideration 16KB seems like a goldilocks choice.

4.1 RDMA Channels and Transactions / Blocks Units

As mentioned earlier, in order to avoid system calls and the latency associated with them our system uses user-level initiation of RDMA transfers. We achieve this by leveraging the SMMU which can hold up to 16 different translation contexts at any time, i.e 16 Protection domains id's (PDID). A different protection ID is assigned to each application running on the system, or to each process within an MPI application (if the application choses so)

The R5 scratchpad memory is split into 16 4K pages, each one capable of holding up to 64 RDMA transfer descriptors, and each page is dynamically allocated to a specific PDID by the driver. From those 64 channels, 32 are allocated for write operations and 32 for read operations. This means that each application can have up to 32 outstanding Read RDMA and 32 Write RDMA transfers at any time.

Furthermore, the ExaNet RDMA transmit unit (ExaDMA) has 1024 available transaction IDs (transaction descriptor), where each transaction ID can hold the descriptor required for one RDMA transaction. Those transaction IDs can be used by the software running on the R5 co-processor at any manner it chooses, allowing for great flexibility when it comes to researching scheduling algorithms etc.

The RDMA will end with a completion notification (usually used for read RDMA) and a response, which shall be an ack or nack, or just with a response. If everything went well, the response will be a positive acknowledgment.

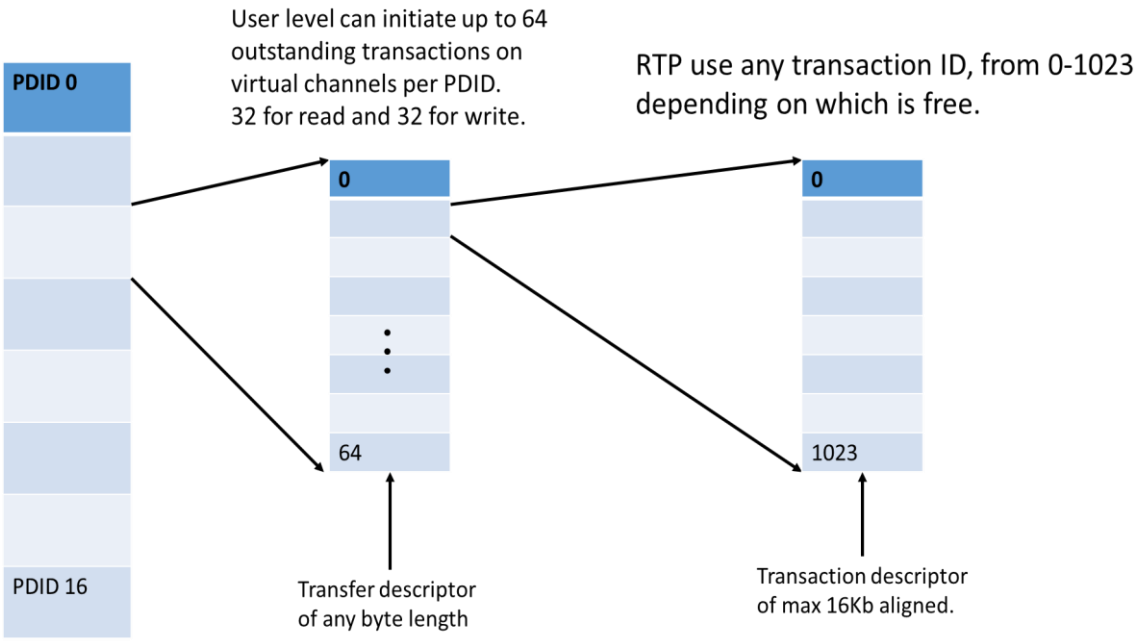


Figure 4-2: Virtual Channel allocation

If, however the RDMA encounters any error, then the response will be negative. Conditions that can generate such a response are:

- Page fault at destination
- No context found at destination
- Packet received with a CRC error
- AXI Decode error on destination

In all of these cases, our RDMA supports retransmission. Furthermore, if a response does not arrive at all (either because the response was lost or because a number of packets were lost all together) then the RDMA transaction will receive a timeout and will be re-initiated. Our network deals with duplicates by tagging each packet of a transaction with the appropriate "retransmission number" which indicates how many times this transaction has been re-transmitted.

4.1.1 RDMA Write

When a user level application requires an RDMA Write operation it informs the R5 by writing on its scratchpad. The write contains all the required information for the transfer, including source/destination virtual address and bytes to send. The R5 continuously polls the scratchpad for new transfers and when a new one is written it will enqueue that transfer on a list of active transfers. Among other things the R5 will segment the transfer into a number of 16KB transactions. After that, in order to initiate a transaction for execution the R5 configures the descriptors of the ExaDMA, with the appropriate information for the transaction by writing on the PL 4x 64bit words. When the last word is written, the ExaDMA

module begins reading the data from the memory hierarchy and sending packets on the network.

On the Receiver side, a context table is kept that keeps track of how many packets have been received (see chapter 7.1 for a more detailed analysis). When all the packets have been received, the receiver will send a response ack/nack on the initiating R5's mailbox. When R5 receives this response from the mailbox, it will consider the transaction as done and continue with initiating the next one until all the transactions of the transfer have been done. The R5 can choose to have any amount of transactions active per transfer at the same time in order to avoid latency resulting on the round-trip time of the response and can also use advanced double buffering techniques provided by the hardware of the ExaDMA.

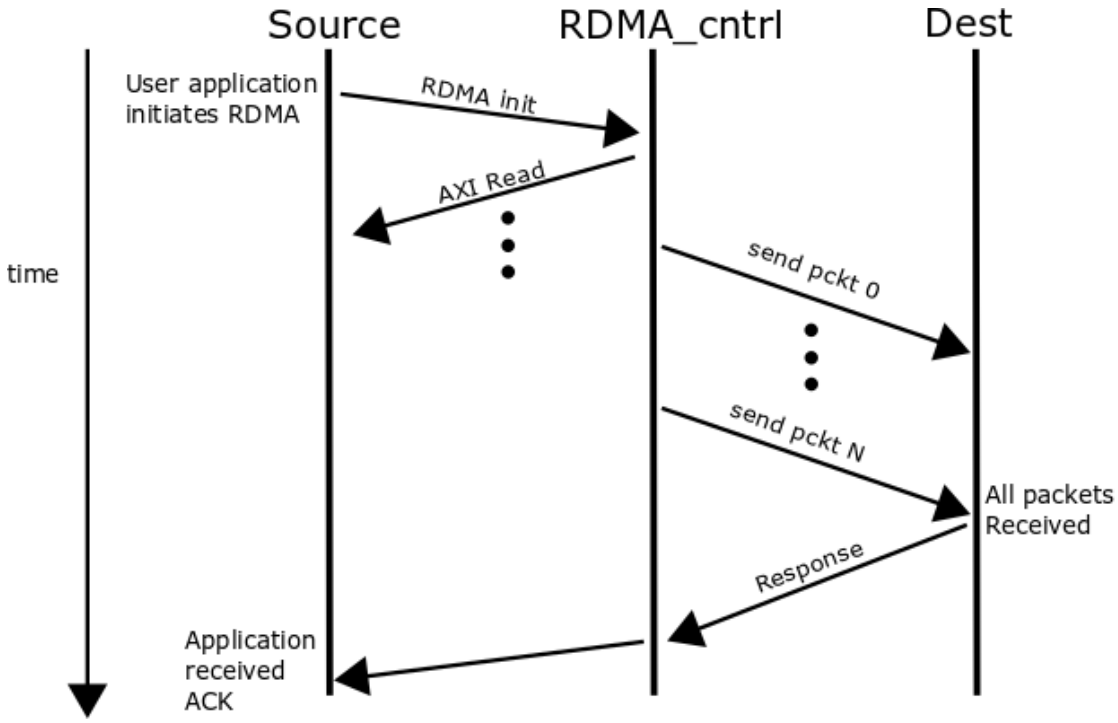


Figure 4-3: RDMA Write timing diagram

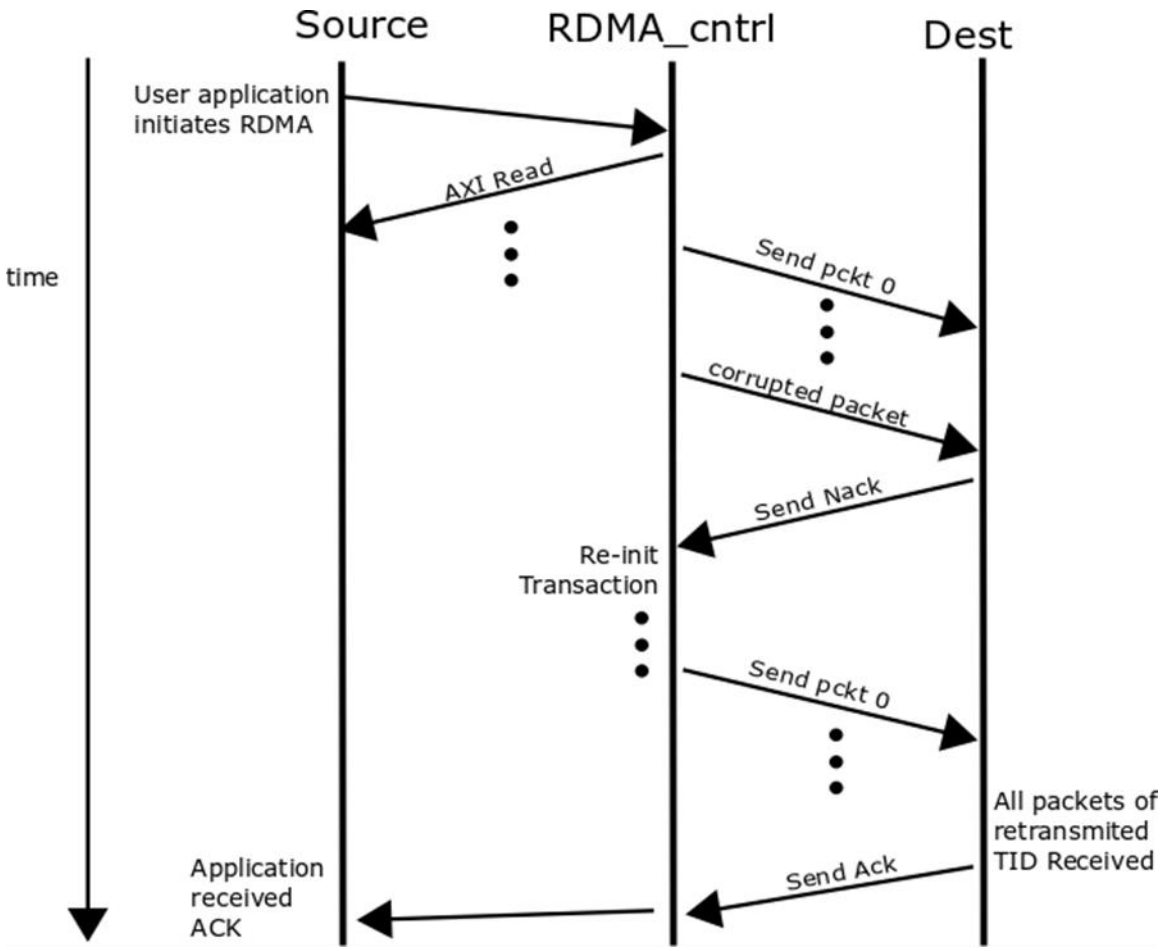


Figure 4-4: RDMA Write with error and retransmission timing diagram

The application can also configure the transfer to receive a completion notification message, a procedure which is described on Appendix A.

4.1.2 *RDMA Read*

Similarly, an RDMA Read operation is similar in spirit with an RDMA Write. But in order to achieve an RDMA Read the packetizer of the system is utilized. When the application issues an RMDA Read operation to the user level library, the library configures and sends a control message from the system's packetizer. This message is destined to the R5 Mailbox of the node from which the application wants to read the data. This message carries all the appropriate information like source/destination and bytes to transfer, and also is tagged by the hardware with the same PDID as the application.

When the R5 dequeues this message, it registers it as a new outstanding transfer, taking up one channel of the 32 "Read channels". From there on the transaction is the same as a RDMA Write but with opposite direction.

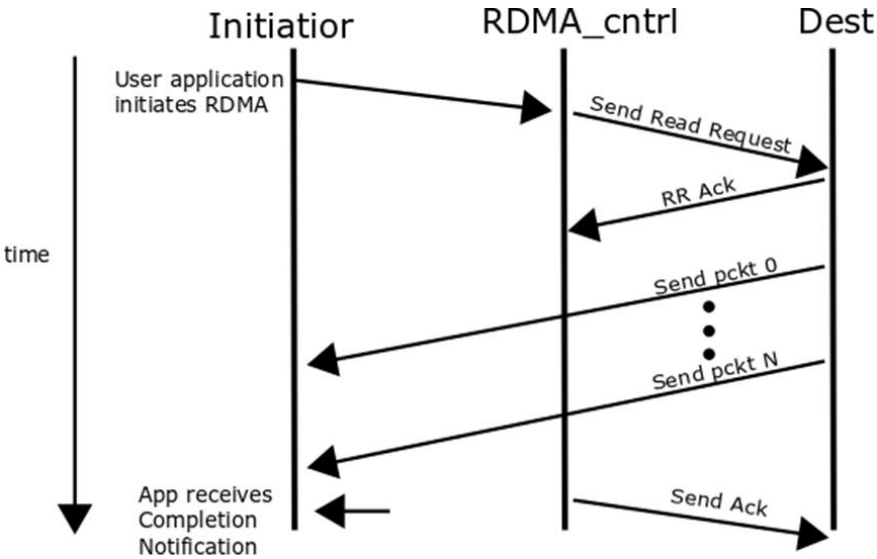


Figure 4-5: RDMA Read timing diagram

This scenario has the added reliability that the packet sent by the packetizer to the remote destination's mailbox can be lost or corrupted. In that case, the remote destination will not enqueue any read requests. Additionally, if the acknowledgment sent from the mailbox to that packetizer is lost, the initiator will not know if the read request is done. The latter is solved by polling the completion notification address and the first requires that the packetizer re-transmits the packet.

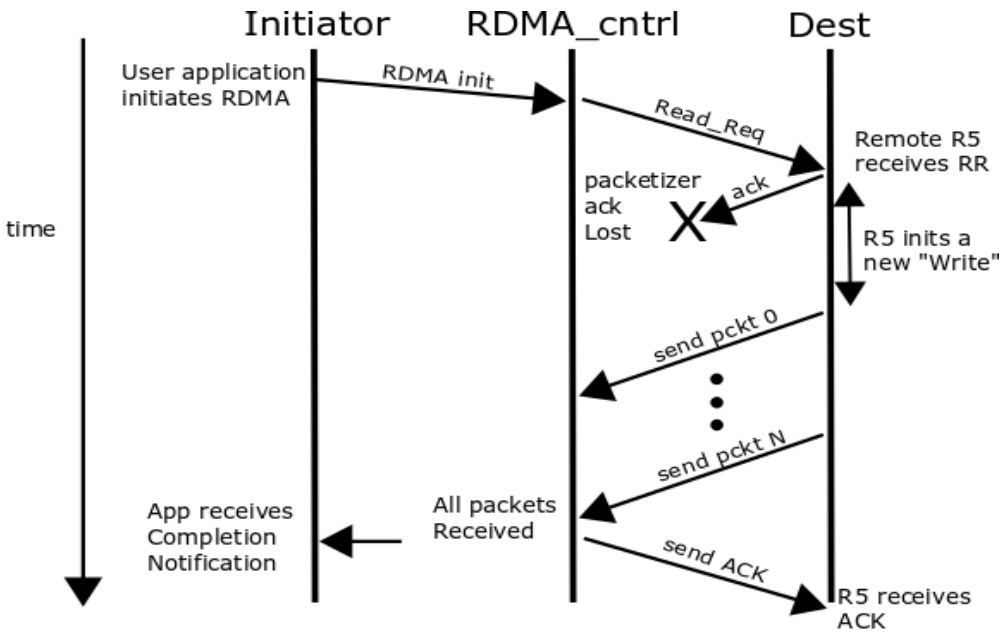


Figure 4-6: RDMA Read with packetizer acknowledgment lost and recovery

The initiating node will be informed that the data have arrived, by polling the completion notification address. This address can be a system mailbox (different from the one that the R5 uses), or an address directly on DRAM.

5 RDMA Send Unit Description

In this chapter, we describe the functionality and top-level architecture of the ExaNet RDMA sender unit, which we call ExaDMA.

The *ExaDMA* engine is used to initiate and handle RDMA transactions over the ExaNet network. It is designed to work in connection with the Real Time Processor of the ZYNQ Ultrascale+ MPSoC. Therefore, some of the functions needed for a complete RDMA transfer, are done by the R5 as described in section 4.1.

The ExaDMA provides 1024 transaction descriptors which can be outstanding concurrently. A round robin scheduler is implemented that iterates through all the active transactions, and initiates one packet from each active one. The maximum packet size of the ExaNet network is 256 Bytes, and so most of the packets in a large RDMA transfer will be of that size. ExaDMA supports transaction sizes that range from 1 Byte up to 16KB, completely unaligned both at source and at destination address.

Apart from the above, the *ExaDMA* module has been designed to be able to work with the congestion management of the ExaNet network as well as the reliable communication additions, meaning that the scheduler can be replaced by a priority-heap scheduler instead of the existing round-robin mechanism.

5.1 Functionality list:

- **ExaDMA is designed to work with R5 co-processor**, which segments transfers as described in sections 4.1.1, 4.1.2.
- **ExaDMA can be used to send RDMA block transactions from 1 byte to 16KB**, potentially unaligned both at source and at destination. When used together with our RDMA receiver, the transfers should not cross 16KB memory boundaries. This is currently managed by the R5.
- **ExaDMA can also act as a simple packetizer:** By writing a small number of words to the ExaDMA controller, an agent, such as a hardware block the R5 or a software running on a nearby processor, can send a prioritized message, “control packet”, to a destination. We use this feature to implement in-band, end-to-end protocols between the RDMA endpoints.
- **ExaDMA supports up to 8 Protection Domain IDs (PDID).** When the Scheduler issues a read transaction to the memory subsystem, it sets the AXI ID to the PDID of the transaction. The AXI-4 protocol states that when different AXI-IDs are used by a master, the slave can respond completely out of order, and in read requests the data can be interleaved. For this reason, the internal barrel shifter is virtualized, and has 8 channels.
- **ExaDMA has a Double Buffering - Chaining functionality.** This allows the user agent e.g. the R5, to configure a set of transactions descriptors, in such a way so that they are all sent one by one in a predetermined order, without any further intervention. The first packet of the Nth transaction is scheduled to depart right after the last packet of the N-1 transaction is send. This can be used as an offloading mechanism of the R5 in order to avoid having to schedule transactions, and potentially increasing the

initiation latency for a new transfer, or can be used in order to define other sub-transfer entities like “congestion managed flows” that may be bigger than 16KB.

- **ExaDMA has an arbitrary number of Output buffers**, for instance one per outgoing transceiver. We need output buffering, in order to avoid bubbles in our network (also required to implement congestion management [5]). When the scheduler issues an AXI read transaction, the PS takes up to 40 clock cycles to respond. For this reason, each buffer has a number of slots (256 bytes payload + 32 bytes Header/Footer), such that in case of no network congestion, the scheduler can issue read requests without having to halt because of all the output slots being full.
- **ExaDMA drives in the footer of each packet the flags “first”, “last”, “notify”**. Flags are used for in-band communication with the receiver, and are used to indicate whether this is the first/last packet of a transaction (in case of out of order delivery because of multi-pathing) and if the transaction will have a completion of notification issued.
- **In order to reduce the critical path of the receiver, the transmitter is also tasked with making sure that no packet crosses 4k Boundaries on the destination address.** For this reason, one packet (the first) can be less than 256 Bytes, in order to assure 256Byte boundary alignment at the destination. In this way, the receiver logic is simplified by not needing to do checks for 4K boundaries on the AXI-Write. The scheduler also makes sure not to cross 4K boundaries in the AXI-Read transactions.
- **Finally, the ExaDMA can detect if a virtual address has prompted a page fault in the AXI-Read interface** and if so, it stops the packet from being sent since the data would be non-valid. Additionally, a mechanism is implemented which completely halts the transaction and waits for the R5 to re-issue when the page fault has been dealt with.

5.2 Descriptor Description /Register space

In order for a transaction to be initiated, the R5 needs to write 4 64bit transaction descriptors. These descriptors fully describe the transaction, and cannot be modified after the transaction is started. A transaction is considered started when the 4th word is written. Following is a description of these transaction descriptors:

63:0
Source Virtual Address: starting source address of the transaction

63:0
Destination Virtual Address: starting destination address of the transaction

63 (1b)	59 (1b)	58 (1b)	57 (1b)	56:42 (15b)	(41:32)	31 (1b)	30:16 (15b)	15:0 (16b)
DB	Send notify	Acked	Done	Bytes send	Dependency id	chained	Transfer length	Prot domain

63:19	18:5 (13b)	4:0 (5b)
Unused	Sequence #	Path

- **Source Virtual Address (R/W):** The virtual Address from which the ExaDMA will read the data. This address should be in the local node (since the data are read locally via the AXI-4 protocol) and is byte aligned.
- **Destination Virtual Address (R/W):** The virtual address of the destination. The 22 MS bits of this address should be the coordinate of the destination node, and the 42 LS the virtual address on the destination nodes memory hierarchy.
- **Protection Domain (R/W):** The protection domain used by both the source and the destination SMMU in order to translate the addresses.
- **Transfer Length (R/W):** The length in bytes of the transaction. Should be from 1 to 16384 Bytes.
- **Double Buffered (R/W):** This bit indicates if this transaction should not be started, before some other transaction is finished being send. The register space provides the user with the ability to initialize a chain of transactions, and enforce the order in which they will be served.
- **Dependable (R/W):** Bit that indicates if this transaction is before another transaction, in a chain of dependable transactions
- **Dependency ID (R/W):** The transaction ID of the transaction that is waiting for this one to end before starting.
- **Bytes Send (R/):** The number of bytes that the engine has sent so far.
- **Done (R/):** Bit indicating if all the packets of this transaction have been sent.
- **Acked (R/):** reserved for later use.
- **Send Notify (R/W):** Bit used by the in-band messages of the RDMA to inform the receiver that this transaction will be followed by a control packet.
- **Path (R/W):** indicates from which output of the ExaDMA the packets of this transaction should be send.
- **Sequence # (R/W):** used for the retransmission protocol, this field indicates how many times this transaction has been retransmitted.

Control Packet Initialization: Control packets are sent as in-band messages in order to achieve the completion notification at destination. In order for the receiver to send a completion notification, the ExaDMA has to send to it a control packet first, containing the payload of the notification, as well as some other control information. In order for such a packet to be sent, the address space of the ExaDMA is increased by 1024 x 128 x n bits, where n is the number of outputs. In order to initiate a control packet, the R5 should do 3 consecutive writes, at the same address, within this range. Both the transaction ID regarding the control packet, and the output from which the packet will be send are selected by the address. For instance, if we have n = 3 outputs, then the address space will be extended by three (3) extra segments. The output is selected by the segment from which the write address resides, and the transaction id is selected by the 128bit word within that segment. The 3 writes made by the software should contain the payload of the notification message. All other information required is found from the internal pending list of the ExaDMA, drastically reducing the number of writes required for such an initialization. This means that the transaction Descriptor regarding a transaction ID should always be initialized before any control packets are issued. These packets completely bypass the output buffers and are send to the network as soon as they are initiated. This creates the possibility of the software wanting to send a new control packet, while the previous one has not been sent again because of backpressure from a congested

network. In this case, the backpressure will be also exerted to the AXI interface, back pressuring the AXI-Writes and potentially the R5 software.

In case of small RDMA transfers in which the completion notification is a significant portion the total completion time, the R5 can reduce the latency by issuing the transaction descriptor first, and then the control packet. That way, the RDMA send unit does not stay idle while waiting for the AXI-Read bus to respond but instead sends the control packet.

5.3 RDMA Send Unit Submodules Description

In this chapter, we describe in detail each of the sub-modules contained in the ExaDMA. Figure 5-1 depicts the top-level architecture of the ExaDMA module, with all the submodules shown.

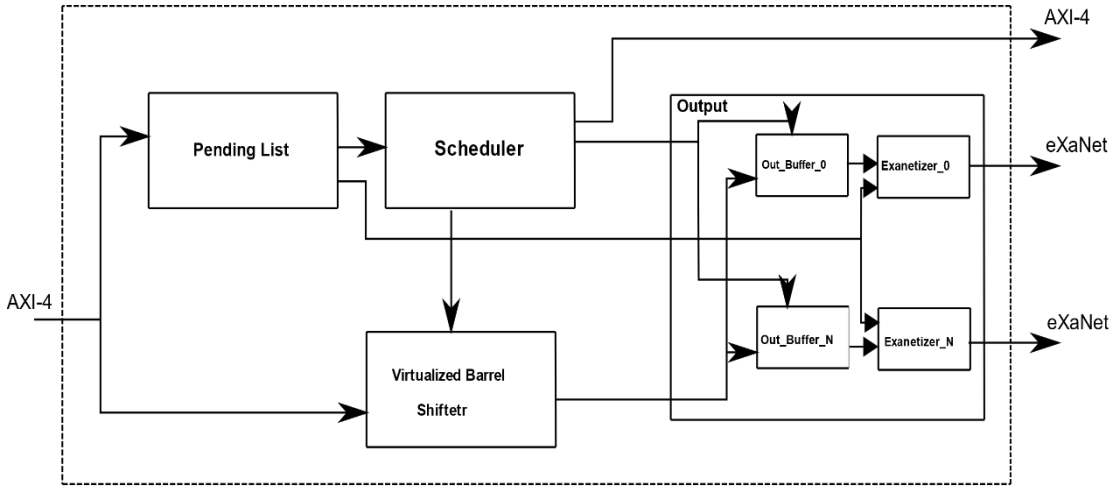


Figure 5-1: ExaDMA top-level architecture

The ExaDMA transmitter block consists of the following sub-blocks:

- Pending List
- Packet Scheduler
- Virtualized Barrel Shifter
- Output Buffers
- Exanetizers

In order for a transaction to be issued, the R5 has to configure the *Pending_List* module via the AXI-Slave Bus. The Pending list is essentially a table keeping the outstanding block transactions, indexed by their transaction ID, selected by the R5. The Scheduler reads transaction descriptors from the Pending List and issues AXI-Read transactions at the AXI-Master interface. The AXI-read responses pass through the virtualized barrel shifter which takes care of the required re-alignments and pass the data to the output buffers. When an output buffer is filled, the ExaNetizer block creates an ExaNet packet.

5.3.1 Pending_List

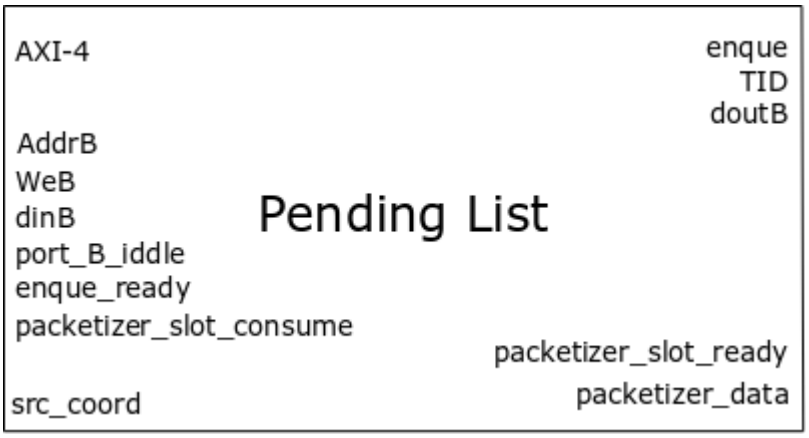


Figure 5-2: Pending List submodule hardware instantiation

The Pending_List submodule is responsible for keeping the array with all the 1024 transaction descriptors. It is configurable via an AXI interface.

5.3.1.1 *Descriptor List*

In order to initiate a RDMA, AXI-4 transactions are generated from the PS to the DMA AXI-Slave interface. Those transactions are handled by the Pending_List module and can be either 32, 64 or 128bit writes. All the ExaDMA register space is both writable and readable but only via AXI single read/writes, since adding the FSM for AXI bursts would increase the complexity for something that is not really used for such situations. This module consists of a large BRAM array, 1024 x 32x4 x 2 in size (262144 bits total). These BRAMs are divided into 1024 transaction descriptors (also referred to as transaction ID's). Each transaction requires 2 x 128 bit writes for its descriptor (256 bits in total). Because the PS can generate smaller than 128bit words, each 128bit word is saved on 4x32bit wide BRAM and the write enable signal of each BRAM is driven by the appropriate Strobe signal of the bus.

This memory is implemented as a true dual-port BRAM, because multiple masters can read/write to it at the same time: 1) the PS when writing/reading a descriptor, 2) the scheduler when reading/modifying a descriptor, 3) and finally the Pending_List module itself, when a new packet is being created. For this reason, great care has been taken, in order to avoid race conditions that could lead to corruption of data.

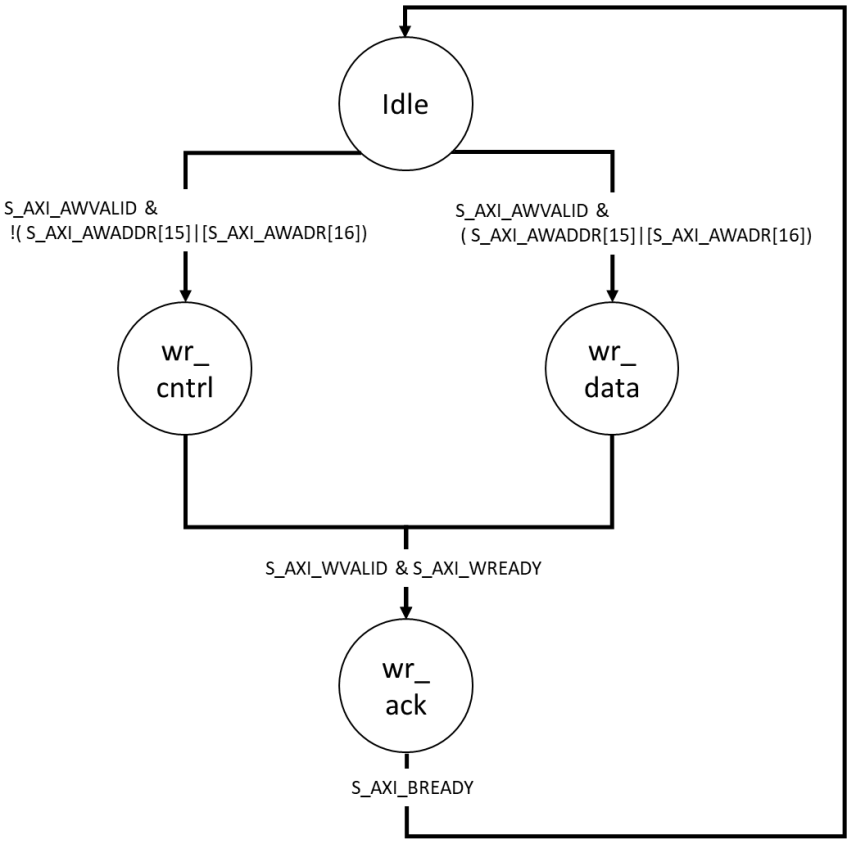


Figure 5-3: AXI-4 write FSM

iddle	**AWWREADY = 1** **WREADY = 0** **BVALID = 0**
Wr_cntrl	AWWREADY = 0 WREADY = 1 BVALID = 0
Wr_data	AWWREADY = 0 WREADY = 1 BVALID = 0
Wr_ack	AWWREADY = 0 WREADY = 0 BVALID = 1

Table 5-1: AXI-4 Write FSM signals

A descriptor consists of 4x64 bit words. When the software writes the 3d word, the transactions is triggered. For this reason ,if retransmissions or multipath are used, the software should write the forth word beforehand, otherwise it can skip writing it altogether. The transaction is triggered by enqueueing a pointer to that descriptor in the scheduling FIFO. At that point, the transaction is considered as active. If, however, the descriptor indicates that

this transaction should wait for another one to end before being sent (DB), then the pointer is not enqueued at all.

In order to handle AXI transactions, this module uses 2 separate FSMs as required by the protocol in order to avoid deadlocks. In the handshake step of a transaction (AWVALID/ ARVALID), the address is latched (as well as the axi_AWID or axi_ARID) and is given to the BRAMs via port A. Since both AXI channels use the same port, priority is given to writes, via a MUX.

For read transactions, the address is given to the BRAM, and assuming that no AXI write is pending, one clock cycle later, the correct data are latched to the AXI_RDATA field.

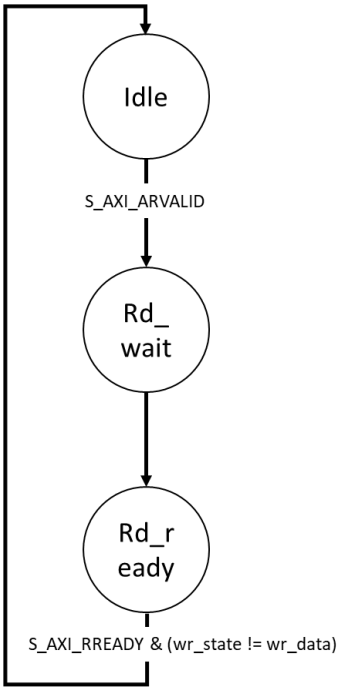


Figure 5-4: AXI-4 Read FSM

iddle	ARREADY = 1 RVALID = 0
rd_wait	ARREADY = 0 RVALID = 0
rd_ready	ARREADY = 0 RVALID = 1

Table 5-2:AXI-4 Read FSM signals

Each 128bit word is saved in a bank of four 32bit BRAM, and the bank is chosen via the LS bit of the descriptor address. The software can write to the pending list via 32/64 or 128 bit writes, so in order to drive the correct Write_Enable signal to each BRAM, the *AXI_strobe* signal along with the bank_select signal (generated by the address LS) is used. We also have to check if the address range does not belong to a control packet initialization, and that the scheduler is ready to enqueue the new transaction descriptor (via the *i_enque_r* signal). In case the scheduler cannot enqueue a new transaction, then backpressure will be exerted to the AXI interconnect, by not asserting the WReady signal.

The signal o_enque is used to inform the scheduler module that the data of the o_TID bus should be enqueued in the scheduling fifo. o_TID is simply a pointer to the address of the newly configured, active descriptor. So, to drive o_eqnue we use the signal we_Bram_11 (which indicates that a write has been made to the MS 32 bits of the third 64bit word) along with inverted MS bit of the Wdata channel. This has to do with the ability to issue chained transactions: if this transaction descriptor is part of chain, we don't want it enqueued yet in the scheduler. The third word is chosen instead of the fourth one, because, from the R5's perspective, each after 3 x 64 bit write, there will be a large latency (due to store buffer) before the fourth word arrives; effectively, the R5 can save latency by not writing anything on the fourth word if this is not necessary. Apart from the above, other signals like *addr_b dout/din_b* are used by the scheduler in order to read and modify existing transaction descriptors

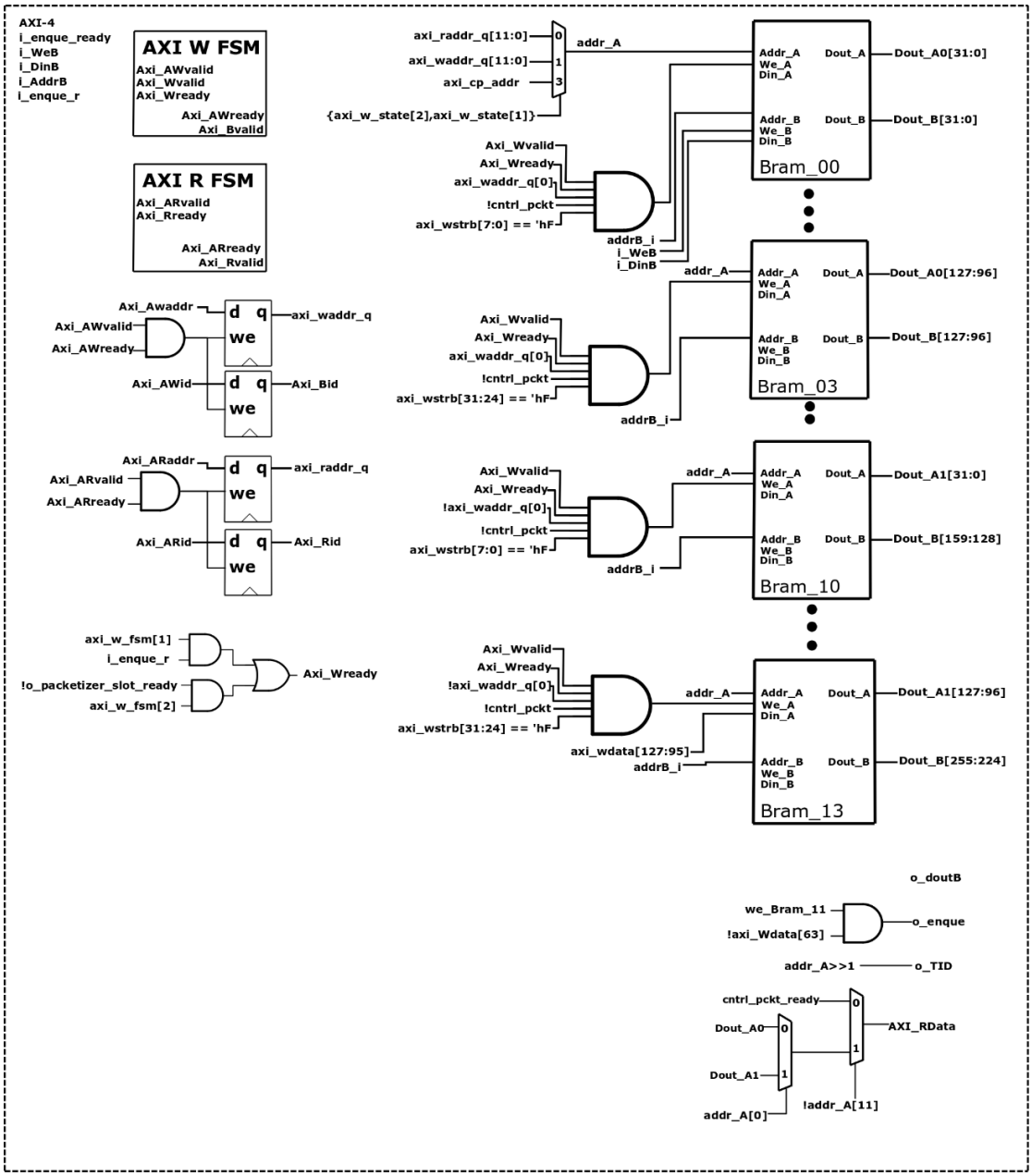


Figure 5-5: Descriptor list detailed schematic. Connections between wires have been omitted for simplicity

5.3.1.2 *Packetizer messages*

Apart from the descriptor list, this module (*Pending_List*) also includes 4 x 128bit wide register, per output port of the DMA, used for sending packetized messages. Those registers hold the information of the message to be send as is (i.e header, two words of payload and a footer). In order to minimize the amount of writes necessary for a message to be sent, the following scheme is used:

The packetizer function of the *ExaDMA* has a large address space, 32KB per output port. The software only needs to do 3x 64bit writes, all at the same address, and the rest of the information will be read by the descriptor BRAM, using the TID as pointer. The software selects the TID by selecting the address of the writes within this address space as in: "packetizer_configuration_base + TID + output_port x 1024". Those three writes should contain the payload of the message. The module can then read the header and footer information from the Descriptor BRAM. This can be done because the messages sent by this packetizer are control messages related to already active RDMA transactions that exist in the descriptor list. Hence, this functionality cannot be used to send general purpose messages.

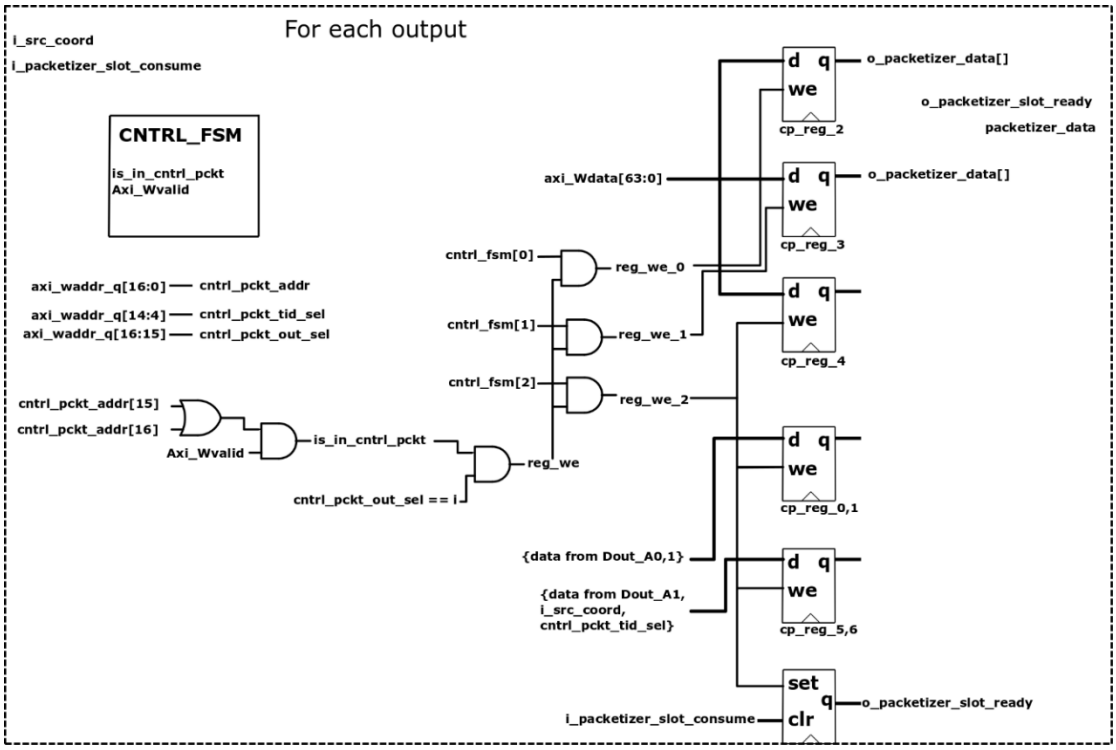


Figure 5-6: Detailed schematic of control packet generation

The above logic is used to generate control packets and is replicated once for each output. It is built around an FSM that has 3 stages, and iterates through them each time the R5 software does a write, in the address range of the control packet configuration. On each stage, the appropriate WE signal is driven so that the correct register latches the payload. When the third payload is latched, the header and the footer of the control packet are also latched by reading all the appropriate information from the BRAM. At the third write, the signal packetizer_slot_ready is also asserted indicating to the output module that a control packet is ready for sending. After sending that packet, the output module will assert the signal *packetizer_slot_consume*. If in the meantime the software tries to write another word on the

registers, then the module will not assert the AXI_WReady signal, back-pressuring the R5, in order to avoid corruption of the data being sent.

When the third write is done, the scheduler is notified and will send the packet as soon as the network is available, bypassing the output buffers. Note that because there is no buffer for these control packets, if the network is congested and the software tries to send a second message before the first one is sent, then this module will backpressure the AXI Interface, and potentially the software.

5.3.2 Scheduler

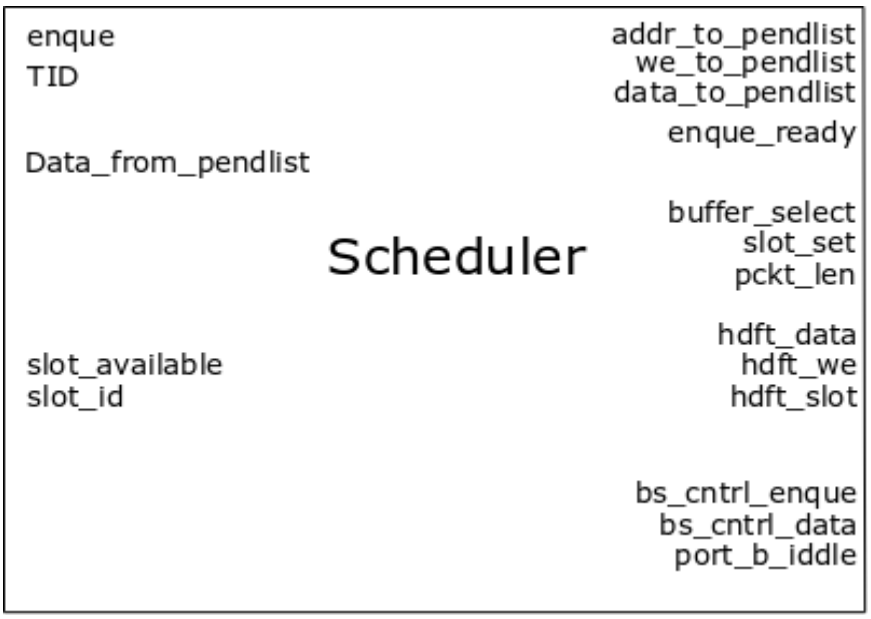


Figure 5-7: Scheduler submodule hardware instantiation

The core of the scheduler is a FIFO that is used for round-robin scheduling. This module also drives the Read-Address channel of the AXI-Master interface of the ExaDMA, and also gives the configuration commands to the Virtual Barrel Shifter.

The FSM of the scheduler uses the *fifo_empty* signal, from the round robin scheduling FIFO, to determine if there are transactions that need to be served. If so, the pointer of the descriptor is dequeued from the FIFO, and is used to read the descriptor data from the *Pending_List* module. The descriptor is read from a 256-bit wide bus, and takes one clock cycle to arrive. In the next cycle, the scheduler determines the “path” of the transactions (which output port of the DMA it will be send from) and looks on the appropriate output buffer for any free slot. If no slots are found, then the scheduler writes-back the descriptor and re-enqueues the pointer, starting the scheduling round all-over, in order to avoid head-of-line (HOL) blocking.

If a free-slot is found, then the scheduler allocates this slot to the transaction, and marks the slots as used. It then proceeds to calculate the information needed for the packet transmission. In order to avoid 4k Boundary checks on the receiving side, the packets should not cross 4k boundaries on the destination address. Since each packet is up to 256 bytes, the

scheduler can purposely issue the first packet with less than 256 bytes, in order to be aligned on 256 boundaries of the destination address. That way, there are no 4k Boundary crosses on the receiving side, thus simplifying its operations, as discussed in more detail in section 7.1

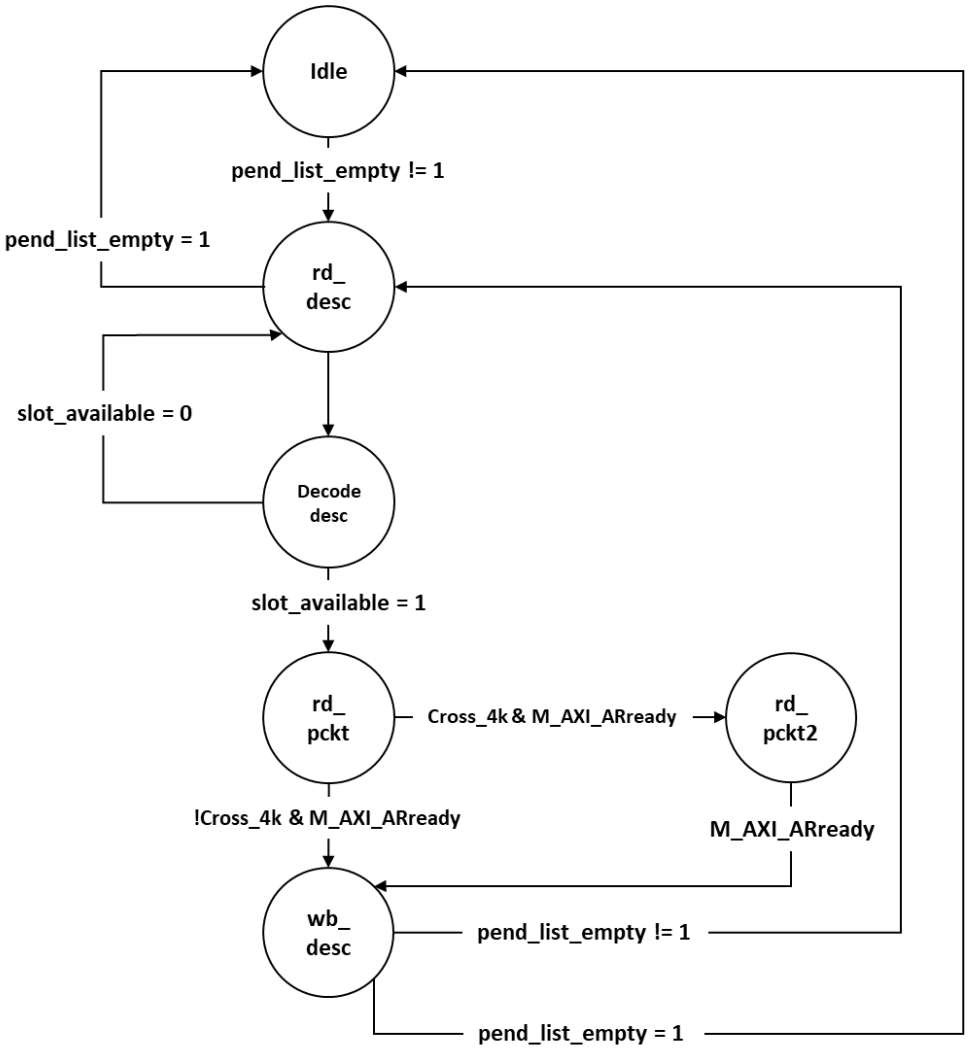


Figure 5-8: RR scheduling FSM

The AXI-Reads also should not cross 4K boundaries. In order to not send two small packets just for this reason, the scheduler can sometimes issue two AXI-Reads per scheduling cycle instead of one. In normal operation, the scheduler will typically issue reads with the size of 256 bytes, or whatever size is left (for the start or the end of blocks). Byte alignment is also done on the sending side, using the virtualized barrel shifter, as discussed later in this chapter. Hence, the final AXI read Address and Size takes into account that the barrel shifter might need one extra word depending on the source-destination address alignment. After the above information is calculated, the scheduler enqueues the appropriate alignment command on one of the 8 command FIFOS of the VBS, depending on the PIDID of the descriptor. The PDID is also used as the AXI-ID, and the AXI Address Read request is issued.

In parallel, the scheduler calculates the information needed by the EXA header/footer and writes it on the output buffer indicated by the slot acquired at the start of the scheduling cycle.

If this was the last packet, then the scheduler marks the “Done” field of the descriptor indicating that the transaction is fully send, and writes-back without re-enqueueing the pointer to the scheduling FIFO. Otherwise it also re-enqueues the pointer.

5.3.3 Virtualized Barrel Shifter

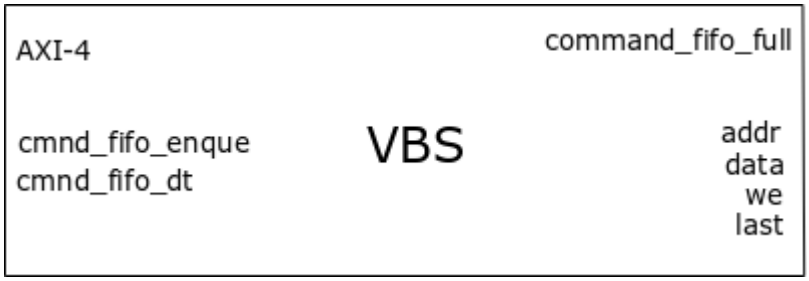


Figure 5-9: VBS submodule hardware instantiation

In order to achieve byte level unaligned transfers both at source and destination, a virtualized barrel shifter is used. The ExaDMA module is connected to the AXI-HCP0 port of the PS, which is a fully IO-coherent port of the PS. In addition, accesses to this port pass through the systems SMMU. The SMMU Driver has 8 contexts allocated for the incoming AXI transactions on this port, which can be mapped to 8 protection domain IDs [6]. The AXI-Read ordering model demands that same ID transactions will be responded in the order they were issued. However, for different ID the protocol allows for both out-of-order and Interleaving of the Read data.

This factor can greatly increase the response time of some AXI-Read Requests since some requests can be found on the DRAM and some others on the caches of the APU. Having support for Out-Of-Order responses allows it to be able to receive data found in the cache, with much smaller latency than if they had to wait for other data, found in DRAM and requested earlier in time. Additionally, the PS interconnection system works at 600MHz but the PL fabric can only get up to 250Mhz. By having support for interleaving delivery of read data, we allow the PS to take advantage of its internal speed up, and further decrease the latency required for the read responses to complete.

The context of the SMMU is selected using the AXI-ID. This means that the barrel shifter should expect to receive a different request datum each clock cycle, and do the appropriate shifting on it. For all the above reasons the barrel shifter is virtualized, and has eight small control FIFOS (one per channel, or per PDID) and registers that hold the commands and state of each read request. Each clock cycle that the AXI-Rvalid signal is asserted, the Barrel shifter selects the correct FSM that needs to be used using the AXI-Rid signal. The barrel shifter is also pipelined in order to be able to reach higher clock speeds.

5.3.4 Output Buffer

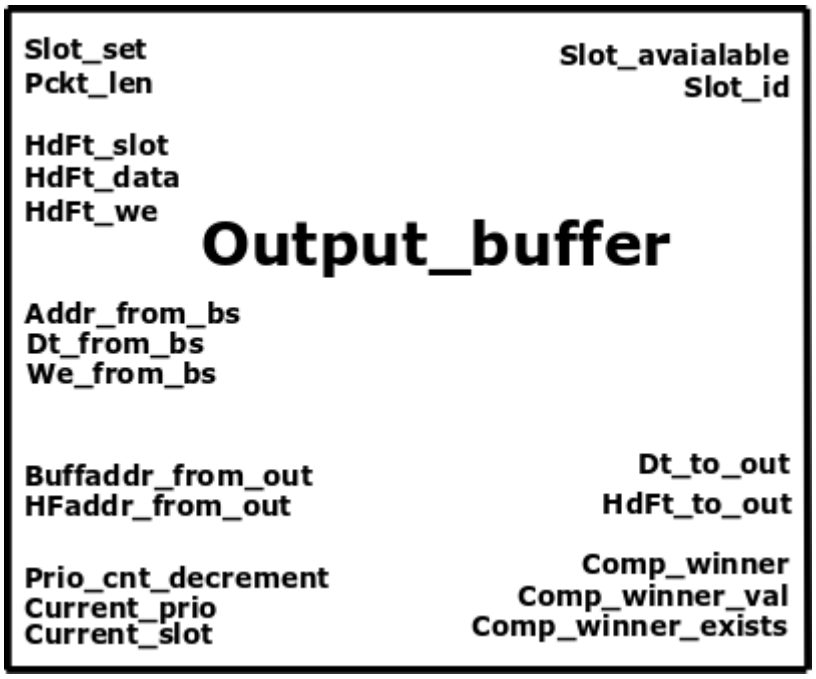


Figure 5-10: VBS submodule hardware instantiation

The DMA can be configured to have any amount of Output Buffers, typically one per path of the multi-path network. The scheduler allocates a free slot, and also sets up a register that indicates how many bytes this slot will receive (any amount from 1 to 256). The number of slots required in order to achieve maximum Read throughput can be calculated by calculating the amount of clock cycles required by the DRAM to respond to a Read request, and taking into account how often the scheduler can issue a read request at full operation. In our design, this is calculated to be eight slots where each slot is 256 bytes. Those slots are directly accessed and written by the barrel shifter, after the shifting is done.

Apart from the payload slots, the buffer also has slots allocated for the Header and footer of the packet, implemented on registers and written by the scheduler at the time of the allocation.

When a word is written on a slot, a counter is decremented, and when it overflows it signals to the output stage that a slot is ready for sending, giving the address of the slot along.

Since the slots can be filled at any order regardless of the order of issuing (because of out of order response when using different read IDs), a mechanism is implemented that ensures that in the case of many slots simultaneously filling, it will enforce the order of issuing when deciding which slot to send first.

Finally, this block is tasked with detecting page faults. The AXI-4 states that if something goes wrong on AXI-Read burst transaction, then the AXI-Master interface that received it should still be able to receive it all. AXI-4 Faults are distinguished by the RRESP signal, which in case of a page fault will have the value 3 or 2. In that case, the barrel shifter will still receive the whole burst, and write in on the output buffer. After the slot is fully written, the *output_buffer* module will not assert the slot ready signal to the *ExaNetizer* block. Instead, it will clear the “used” flag, and raise an “interrupt” to the *scheduler* module indicating that a

packet has received page fault, along with the TID of that packet. The *scheduler* will then go on with effectively killing the transaction, and setting the "error" bit to 1. Meanwhile the page fault handling mechanism of the system will inform the R5 of the problem when the page fault is dealt with, in order for a retransmission to be issued. That way we avoid sending unnecessary garbage information in the network, reducing unnecessary bandwidth consumption and congestion.

5.3.5 Output Stage

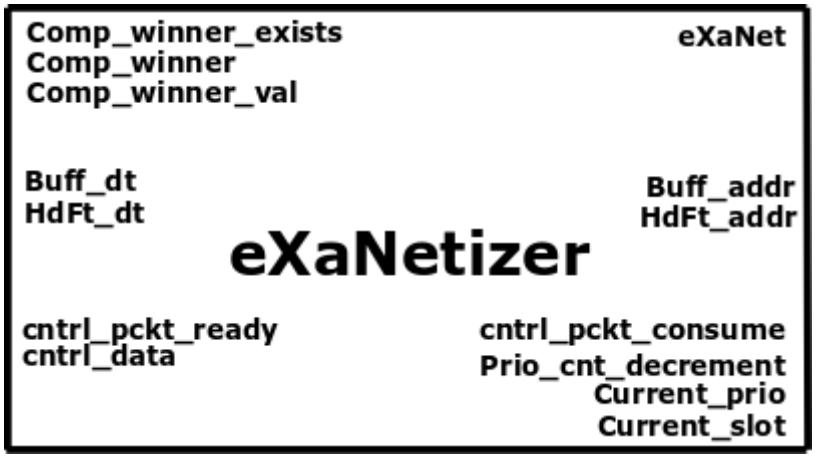


Figure 5-11: ExaNetizer submodule hardware instantiation

The ExaNetizer module is responsible for creating the ExaNet packet and sending it to the network. ExaDMA uses one ExaNetizer per output buffer and so is able to send to multiple destinations at the same time. A packet will be sent when an output buffer slot is completely filled or when a control packet message has been written by the software. If both are true, then the control packet messages have priority over the normal payload packets.

When a control packet message is written by the software, a register is latched that does not allow the software to configure a second message until the first one is send. This is done by not asserting the $Wready$ signal on the AXI transactions, and therefore backpressures the R5. This means that the software can write control packet messages without first checking if the prior ones have been sent, which typically is bad because it adds latency.

The inputs used to determine when a payload packet is ready for transmission are *comp_winner_exists* and *comp_winner_val*, which indicate if a buffer slot is filled, and which slot it is. The FSM of this submodule can be seen Figure 5-12. First, the FSM waits until a slot is filled in the output buffers or a control message is ready, giving priority to the control messages. If the first is true, the FSM latches the *comp_winner* signal because this signal can change at any moment while the barrel shifter writes data to the output buffers and uses this signal as address to the header/footer and payload buffers, while proceeding to send the header of the packet. If this is a control header, then it sends the header directly from the *cntrl_data* bus. After the *header_ready* signal is asserted, the FSM jumps to the next state.

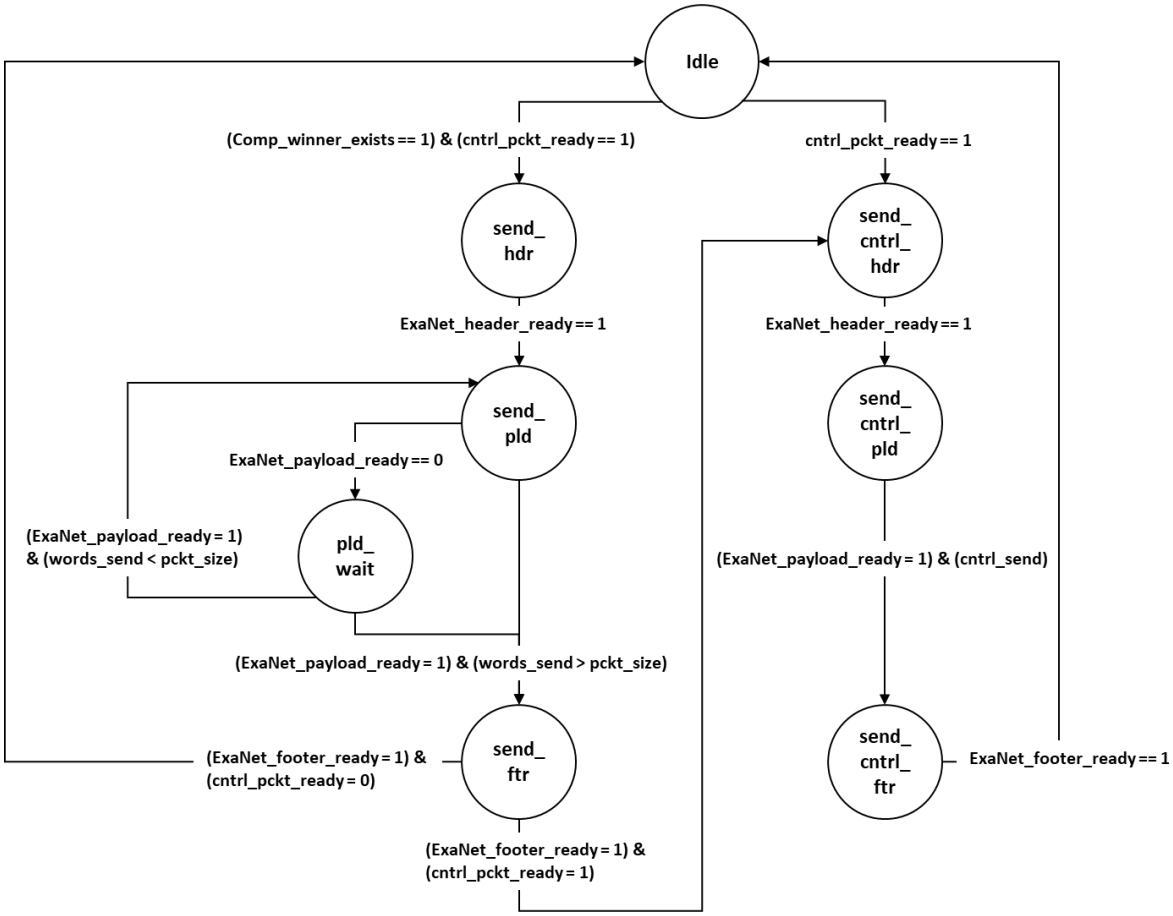


Figure 5-12: ExaNetizer Submodule FSM

The second stage of the FSM is the control payload or standard payload. In this stage, the FSM sends payload words, advancing the buffer address, until the words send are equal to the size of the packet (which has been latched on the *send_header* state. In case that the *payload_ready* signal is not asserted the FSM jumps to an intermediate state, *payload_wait* and latches the payload until the *payload_ready* signal is asserted, because the buffer address has been changed, and new data will arrive on the *Buff_dt* bus.

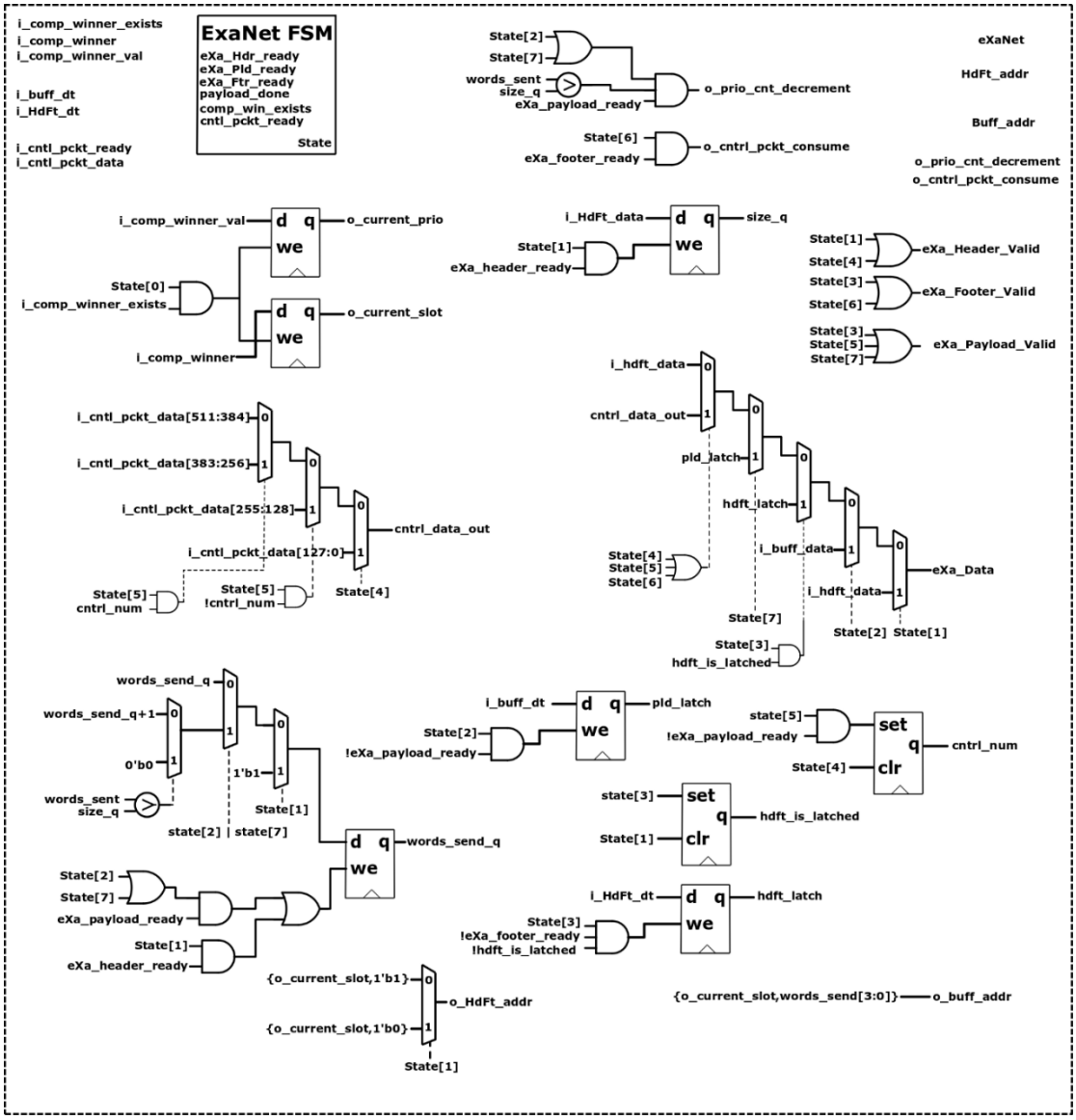


Figure 5-13: ExaNetizer submodule detailed schematic. In order for the schematic to be clear, connections between signals have been omitted and are indicated by same name.

If the FSM was at *send_cntrl_pld* state, then, after sending two words, it will jump to state *send_cntrl_ftr*. If the state was *send_pld* and the last word of the payload is sent, the FSM asserts the output signal *o_prio_decrement* that signals the output buffer that all existing slots should decrement their priority, and also sets the used slot as freed so that the scheduler can re-issue a packet on it, then jumps to *send_ftr* state. If the network is congested it might not assert the *footer_ready* signal for a long time. This is a problem because the slot currently being used has been marked as freed in the previous transmission cycle, and the scheduler can write a new footer/header on it. For this reason, the data for the footer are registered

after the first clock cycle that the *footer_Valid* signal is asserted when the *footer_ready* signal is zero.

Finally, after the footer has been sent, the FSM will jump either to state *idle*, *send_hdr* or *send_cntrl_hdr*, depending on the starting state and what signals are asserted as shown in Figure 5-12.

6 RDMA Modules Description

6.1 RDMA Receiver

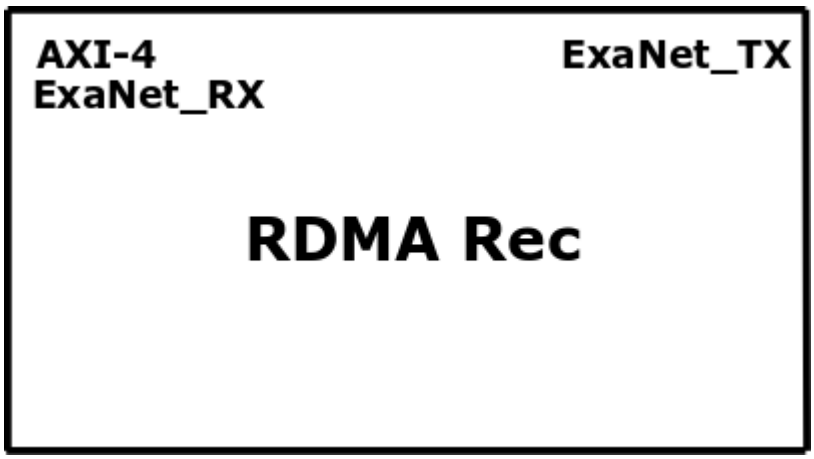


Figure 6-1: RDMA Receiver hardware instantiation

The RDMA Receiver module is responsible for receiving ExaNet packets, allocating them into contexts and keeping track of the transaction progress. It has been developed at FORTH by a colleague in parallel with the ExaDMA, and I overview its inner workings for completeness.

As shown in Figure 6-1 the RDAMA receiver has 3 interfaces. An AXI-4, an ExaNet_Rx used to receive ExaNet packets, and an ExaNet_Tx used to transmit responses.

The core of the RDMA Receiver is a large BRAM array, called “context table” which is used to do the bookkeeping about all the active transactions. It consists of 256 contexts each one holding an 64bit bitmap to mark down all the packets that have arrived, as well as fields to hold the transaction ID, source node coordinate and other fields used for the completion notification mechanism. This context is implemented as a 16-way associative cache, in order to be able to serve active transactions faster.

The RDMA Receiver also acts as an ExaNet to AXI-4 converter. When a packet arrives, it is immediately written to the DRAM (cut through operation), after passing the virtual address to the SMMU of the system. When the footer of the packet arrives, which contains the transaction ID and the source node ID (both used for hashing inside the context table) the receiver begins the mark down of the packet into the appropriate context slot. This packet can either belong to an active transaction, in which case we will have a hit, or to a new transaction, in which case we will have to allocate a new slot to it. If no slots are available then the receiver will send a properly marked negative response to the source. The Receiver uses the address of the packet to determine the correct bit of the bitmap to markdown, i.e. the position of the packet in the bitmap, and knows what bits should be completed by looking at the footers of the packets for the “first” and “last” flags. When all awaited bits are completed, the receiver will send a positive ExaNet response back to the source.

If the SMMU responds negatively (BRESP != 0) , or if the packet has a CRC error, then the receiver will send a negative response with the appropriate error code back to the source. Packets carry a “sequence number” field, which indicates the number of times this transaction has been retransmitted. The Receiver keeps in its bitmap this sequence number and if at any time it sees a sequence number packet lower than the one currently active, it ignores it

(doesn't write down the bitmap field). However, if it sees a sequence number larger than the saved one, then it completely flushes the bitmap in order to be sure that all the packets of the retransmission have arrived. Finally, all the responses generated by the receiver carry the same sequence number as the transaction they are meant for, which helps the source identify if they are the correct responses, or are from "older" packets that have been retransmitted.

With all the above features, the receiver allows our RDMA network to be able to handle completely out of order delivery of packets, enabling multipathing, and also allows for end-to-end resiliency features such as retransmissions.

Finally, the AXI interface is solely for debug purposes. The software can read all the contexts, using a program that decodes the data returned by the context BRAM, and get a complete view of the current situation in the context table, i.e. what slots are active, how many packets have arrived etc. This has greatly helped in debugging, since before implementing this, we had no way of knowing what had gone wrong.

6.2 RT mailbox

Figure 6-2: RDMA Receiver hardware instantiation

This block is the mailbox used by the R5 processor in order to receive transaction responses and Read-Requests. It consists of two FIFOS, one for read requests and one for ExaNet transaction responses. ExaNet packets arriving from the network are written on those FIFOs via an ExaNet interface, and the R5 can read those FIFOs via a local AXI protocol. Those FIFOs are dequeued accordingly based on how many reads the software needs to do from the AXI interface in order to read all the required information. Since the R5 can only read using 32bit words, this means that a word is dequeued from the response FIFO for each AXI-Read on it, and one from the response FIFO every four. The requirement for two different FIFOs arises from the fact that the R5 software has an internal queue for Read-Requests that can get full , making the R5 unable to receive more Read-Requests. If both read requests and responses were in the same FIFO, and this FIFO is dequeued when the software reads from it, and since the R5 does not know whether what it will read is a response or a read request, a situation could occur that the R5 cannot read from this FIFO even for responses, because the Read-Request queue is full, leading to protocol deadlocks. For this reason, the R5 software can choose from which FIFO to read, by changing the offset address of the read request.

An optimization to the above is that when the R5 reads from the response FIFO, apart from all the payload related to the response, a special bit is returned that indicates whether or not the read-request FIFO is empty or not. That way the R5 can skip reading it altogether, reducing the loop time required since the PL reads are very time consuming.

Finally, in order to serve a remote Read-Request, the R5 has to know the protection domain of the application requesting the read. This information is on the ExaNet header of the packet, which is created on hardware (from the requesting nodes packetizer), based on configuration made by the kernel on the packetizer channel used to carry the read request. When this packet arrives to the R5 mailbox, some of the payload is the original payload chosen by the user, while protected payload, like the protection domain, is constructed by the mailbox itself, based on the packet's header. In this way we don't have to worry about malicious users trying to read from protection domains other than their own.

6.3 ExAurora

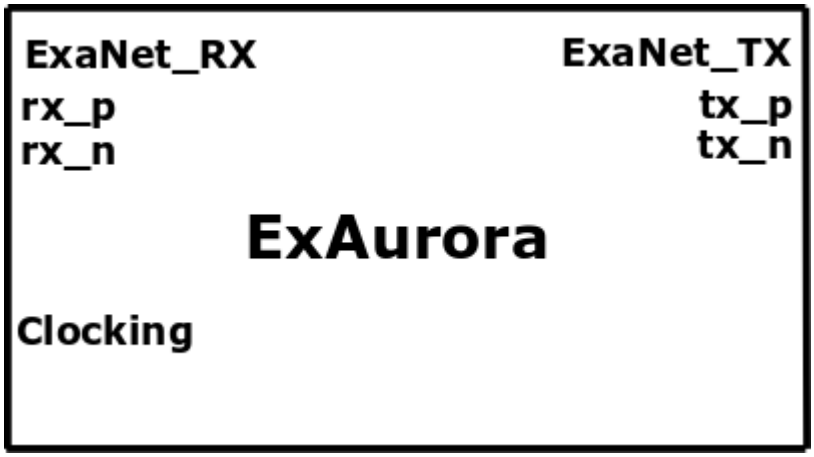


Figure 6-3: ExAurora hardware instantiation

The *ExAurora* IP is the transceiver of our system. It is a wrapper that instantiates the IP provided by Xilinx, *Aurora 64B/66B* which in turn uses a SERDES (serializer de- serializer) capable of achieving transmission speed on High Speed Serial (HSS) links up to 16Gb/s, with the ability to bond lanes and achieve even higher speeds. In our system, *Aurora* is configured to work with 10Gb/s links, although the PCB can get up to 16Gbit. *Aurora* provides two AXI-Stream interfaces (one for incoming and one for outcoming traffic) which use the TVALID, TREADY, TLAST, TDATA [63:0] signals of the bus (TREADY signal is omitted on the receiving side). It also provides a second AXI-Stream interface which can be used for flow-control using NFC (Native flow control). When the NFC-Tvalid signal is asserted, the *Aurora* stops sending data and instead sends the NFC-TData. This is an 8-bit bus which holds information regarding flow control. The 7 LS bits can be used to send pause commands and indicate how many cycles we want the pause to last and the MS bit is used to send XON/XOFF commands.

ExAurora IP utilizes two cross-clock FIFOS; one for Rx and one for Tx, which are low latency (only 3CCs from first enqueue until not empty signal is asserted). The Rx FIFO is 128bit wide and 256 words deep. It uses programmable full and programmable empty functionality to implement the XON/XOFF watermarks and the required high/low watermarks are calculated to be 72 words, based on our link rate and datapath. An FSM monitors the programmable full signal and when it is asserted, it uses the NFC-TVALID to send the appropriate XON/XOFF commands.

The Tx FIFO is also 128 bits wide and 256 words deep. This FIFO has the programmable Full signals configured in such a way that the FIFO drives the eXa_Header_Valid signal only if the FIFO can fit a full packet length (18 words). This helps with congestion management and routing algorithms, and also has greatly helped with debugging.

Since only one FIFO is used per transmission side, the FIFOS are also accompanied by the appropriate logic that decodes the header in order to know how many words are payload and which are header/footer.

Last but not least, since each FPGA has many transceivers scattered across many banks of the FPGA, ExAurora comes with the appropriate logic to configure many auroras to work using the same GTH clock.

6.4 ExaNet intra-Switch

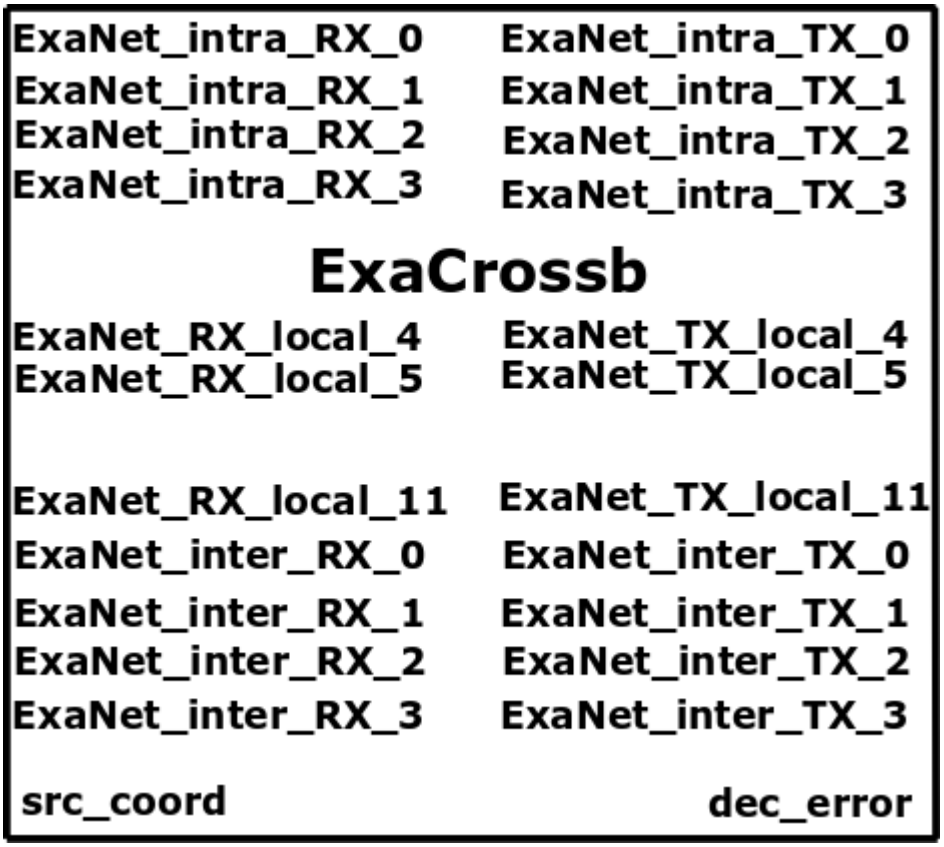


Figure 6-4:ExaCrossb hardware instantiation

Routing in our network is done in total by two switches. The one is the Inter-QFDB router that manages the connections going out of the QFDB through the F1 (Network FPGA) and uses only the destination coordinates to route. It has in total 4 local ports and 8 remote ports and is provided as a black box by a partner of the project.

The other switch is used for intra-QFDB and NI address-based routing (ExaCrossb shown in Figure 6-4). It is a 16x16 port ExaNet buffer-less crossbar. The switch is buffer-less because all the IPs involved in the RDMA network either have output buffers (ExaDMA / packetizer) or have input buffers (ExAurora), so this helps with lowering the utilization and minimizing the amount of FIFO latency.

Out of those 16 ports, four (4) are used for routing into the intra-QFDB transceivers and another 4 are used for inter-QFDB traffic, connecting directly to the 4 local ports of the network router as shown in Figure 6-5. ExaCrossb has as input the source coordinate and uses a 2-level routing algorithm.

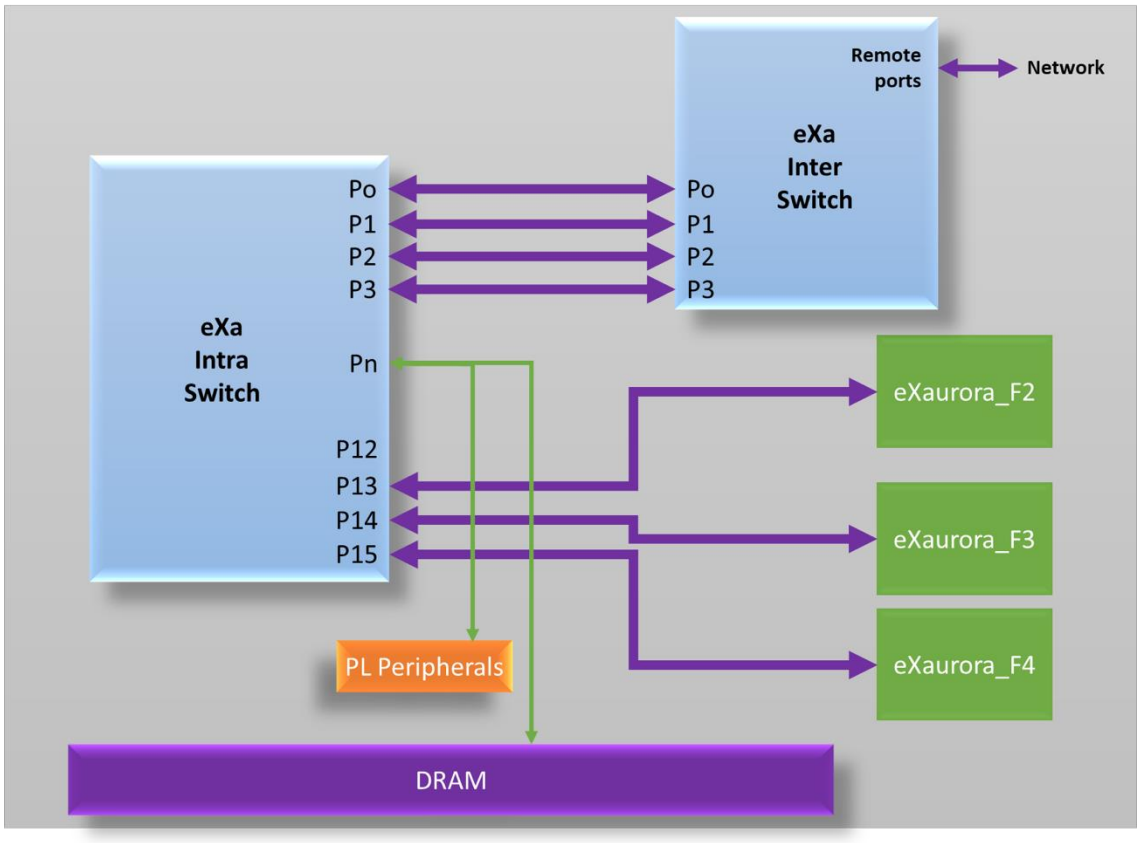


Figure 6-5:ExaNet network routing

The first level checks if the destination coordinates and the source coordinates do not match. IF the fields X,Y,Z of the coordinates (which QFDB in the network) do not much, then it routes the packet to the F1(Network FPGA). If, however the packet is already in the F1 FPGA then it routes the packet to one of the local ports of the Network router. Which of the 4 ports is chosen depends on what input port the packet comes from. If it comes from one of the local ports, then it routes it on port_0, if it comes from the input port that connects to the F2 FPGA it routes it on port_1 and so on. This allows for maximum bandwidth while simultaneously avoiding head-of line-blocking, in case that multiple intra-FPGAs need to communicate with remote QFDBs.

If only the "offset" coordinate (which FPGA within a QFDB) does not match, then it routes the packet to the appropriate transceiver that connects to that FPGA. Since the FPGAs within the QFDB are connected in an all-to-all topology, there is no need to check for deadlocks.

If the source and destination coordinates match, then this means that this is the destination node of the packet. In this case, the packet is routed based on its address to the appropriate ExaNet peripheral (mailbox packetizer etc.) or to DRAM (ExaNet RDMA Receiver)

6.5 Network Utilization report

Following are the results of the implementation of our RDMA network interface on the Xilinx Ultrascale+ FPGA.

	LUTS	RAMB32
RDMA Tx	4447 (1.6%)	10.5 (1.1%)
RDMA Rx	12494 (4.5%)	9 (1%)
RT mailbox	274 (0.1%)	2 (0.2%)
ExaCrossb	8199 (3%)	0
ExAurora(3x)	2253 (1%)	15 (1.6%)
Total	27667 (10.2%)	36.5 (3.9)

Table 6-1:Network Utilization Report

As shown on table 7.1, the whole RDMA network requires very little resources allowing for extra space to be used for other forms of accelerators.

The “buffers” of the system are located in the ExAurora blocks, which account for all their 15 BRAM, and at the output buffers of the ExaDMA (RDMA Tx). Some of the BRAM used on the RDMA Tx as well as all the buffers used at RDMA Rx are for state-keeping with the Rx side having to keep all the active contexts (256), and the Tx side having to keep all the active transaction descriptors (1024).

7 Experimental Evaluation and Results

In this Chapter, we present the average flow completion time and the average throughput of our new advanced user level initiated read/write RDMA for various transfer sizes and network hops.

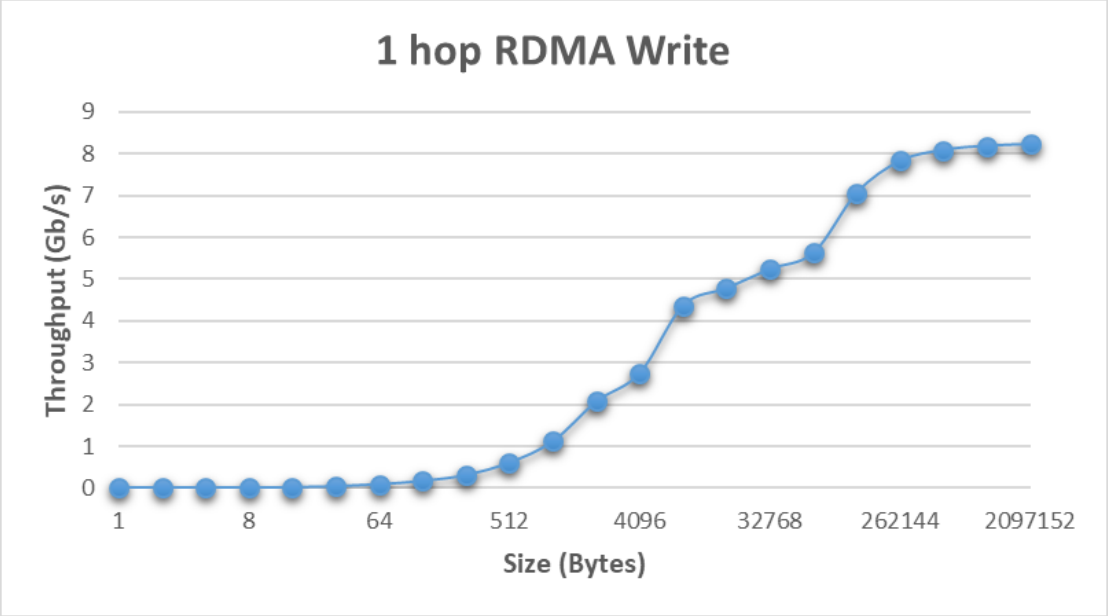


Figure 7-1: 1-hop RDMA Write throughput - Size. User level application

The design on which we tested was implemented with an ExaNet datapath running at 150Mhz, meaning that intra-FPGA the maximum throughput can get up to 19Gbps. However, the transceivers connecting the FPGAS run at 10Gbps .As shown on Figure 7-1, the max throughput achieved on one hop (intra-qfdb) transfers is 8.23Gbps. As already mentioned, the maximum packet size is 256 bytes, or 18 datapath cycles (16 payload + 2 header/footer). This results in each packet having up to ~11% header/footer overhead. Hence if the theoretical maximum of the transceiver is 10Gbps then the max that an RDMA transaction can achieve is ~8.9Gbps. If we take into account routing latencies within the FPGA that can account for extra 3 clock cycles, then we can see that our RDMA engine achieves the maximum theoretical throughput of the system. The benchmarks used where the MPI OSU microbenchmarks suite, with a custom MPI library implementation which does not yet support the eager protocol, and uses only the RDMA Read.

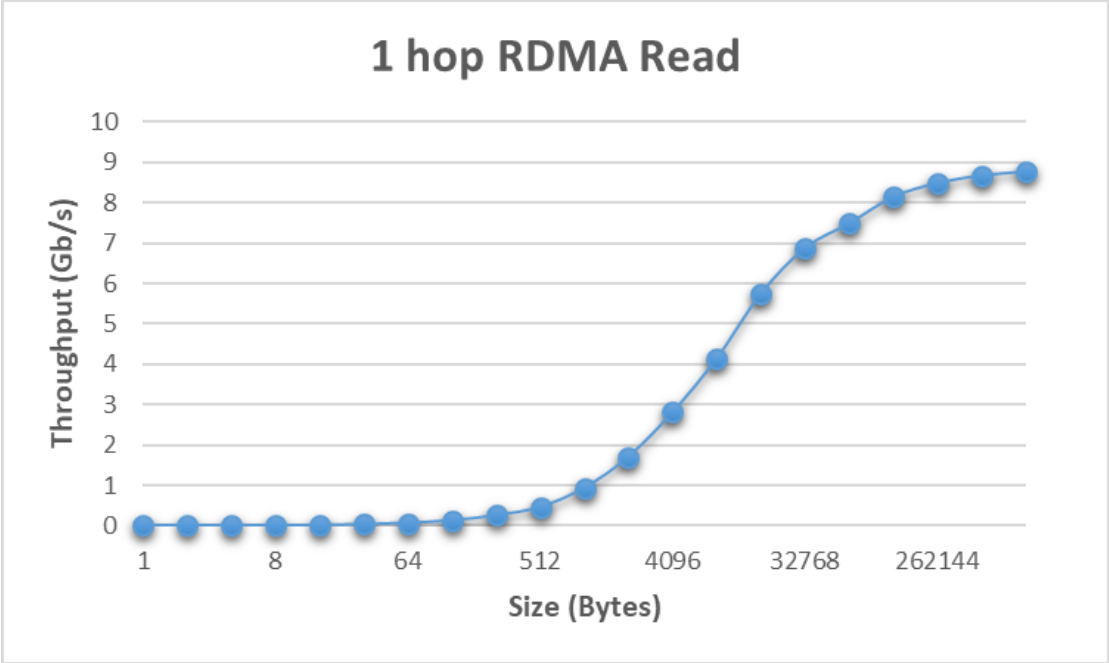


Figure 7-2 :1 Hop RDMA Read throughput – Size. OSU Bandwidth

On Figure 7-2, we can see the situation for RDMA Read operations. The test used was the MPI application OSU microbenchmarks, latency. As expected, the performance is generally the same regarding throughput.

We showed results for one hop, intra-qfdb transactions. When the hop is inter-qfdb, then the maximum theoretical throughput further reduces, since the routing latency of the Inter-qfdb switch is significantly higher. The topology used for the ExaNet network can be seen on Figure 7-9.

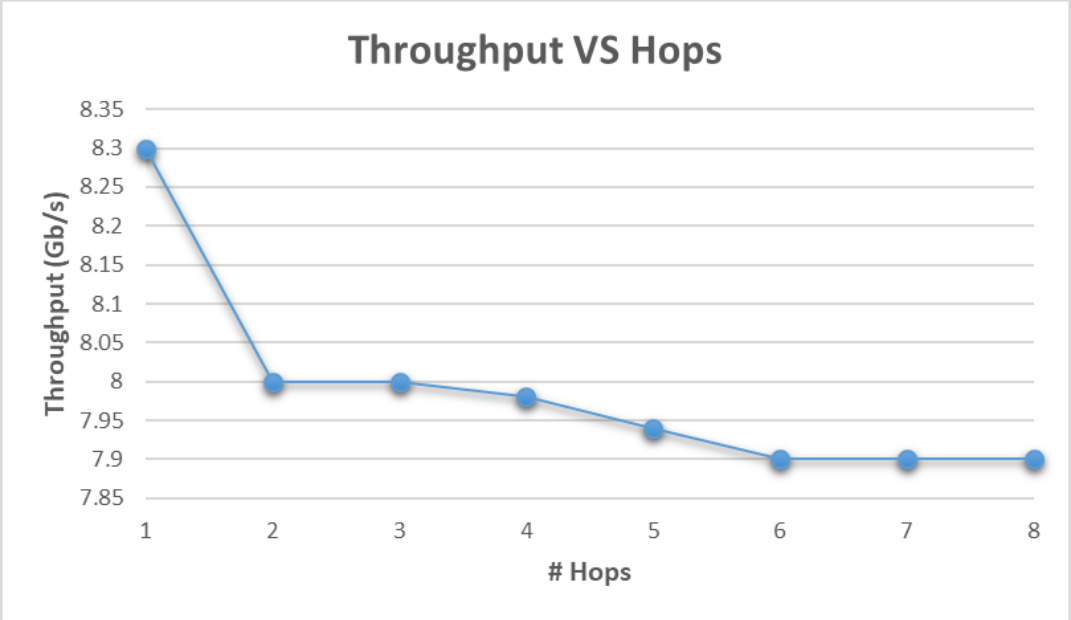


Figure 7-3: Throughput - Hops. Note that the drop is from 8.3 to 7.9 Gb/s

One of the most important features that our advanced RDMA engine provides is the scalability as the prototype becomes larger. This can be clearly seen on Figure 7-3 where we can see that the throughput remains constant as the number of hops increases.

At this point, we will provide some comparison with the RDMA engine used in the project so far, mainly for application development and prototyping. This engine (referred to as zDMA for sort) had a lot of limitations, the main being its low performance, due to the small packet size (64bytes) and the small number of outstanding transactions (6).

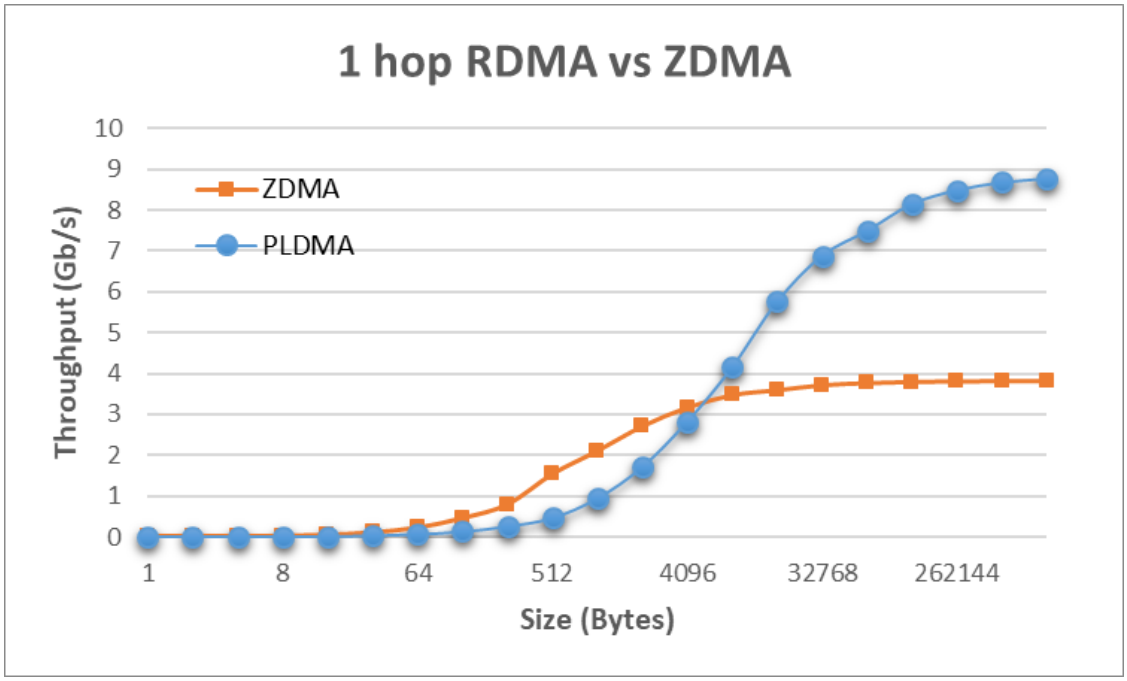


Figure 7-4: 1 Hop PLDMA/ZDMA - Size OSU, Bandwidth

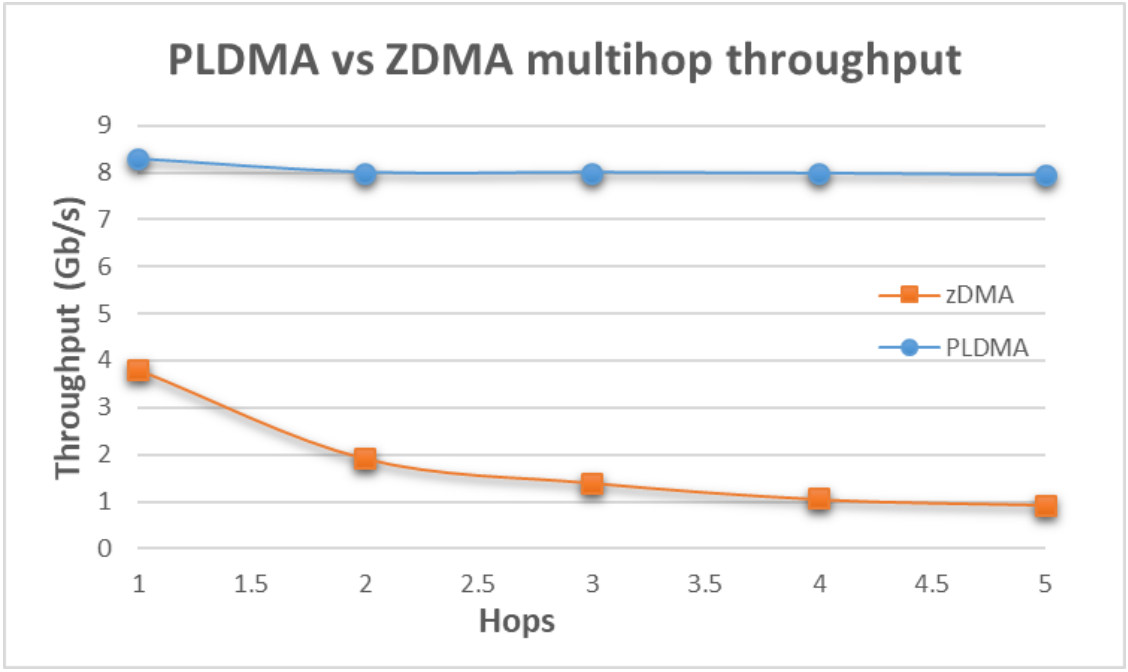


Figure 7-5: PLDMA/ZDMA - #Hops multi-hop throughput. OSU, Bandwidth

We also run various tests to compare with our tcp/ip over 10G . Our 10G network is a custom hardware implementation build around the ethernet 10G MAC ip , provided by Xilinx. The maximum throuput this implementation can achieve is ~3Gbit as shown below. Again we used the OSU microbenchmarks for evaluation.

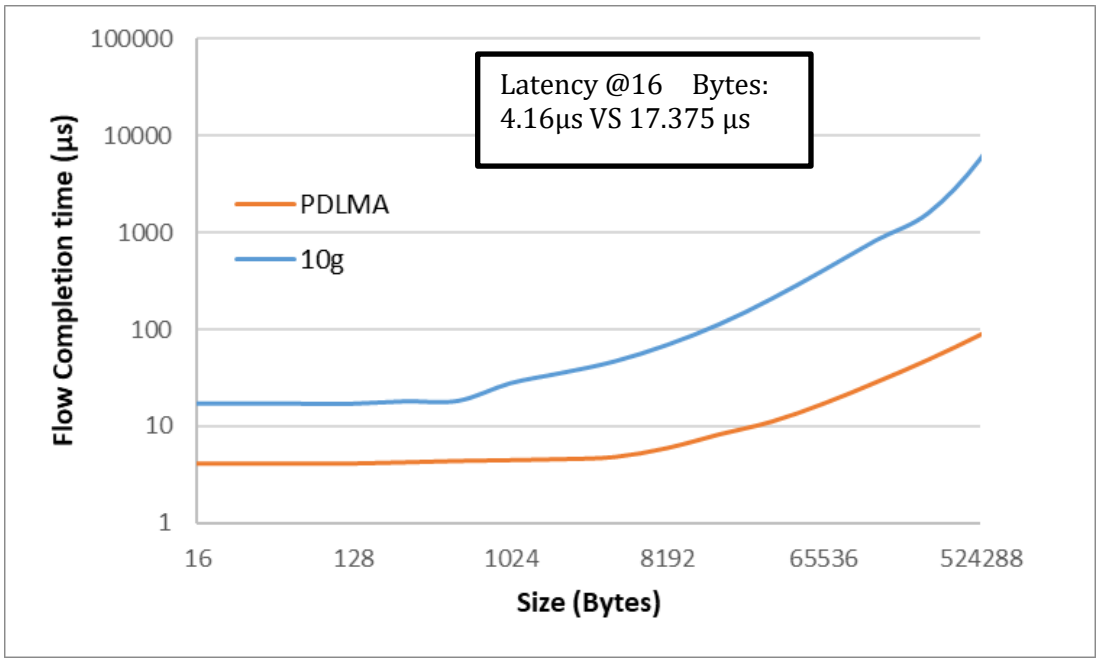


Figure 7-6: Flow Completion Time 10G vs PLDMA. OSU, Latency

As we can see from Figure 7-6 our implementation provides ***4.16 µs*** one-way communication latency whereas the 10G implementation has ***17.3µs.*** Further breakdown of this latency, shown in Figure 7-7 shows that only 0.7µs of this latency accounts to hardware latency, and from that, almost 0.5µs accounts to transceiver latency and not functions related to our RDMA. The Rest of the latency can be accounted on operations done by the R5 co-processor. As the size of the transfer increases, our implementation is almost two order of magnitudes better, and that is due to the much higher throughput that our RDMA provides.

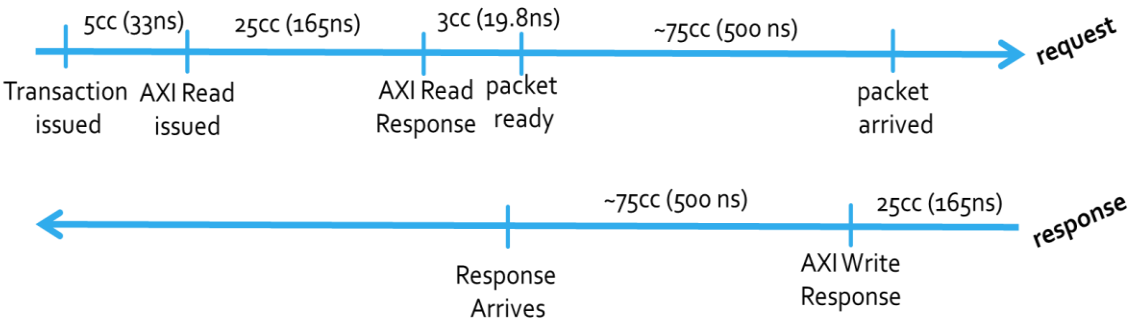


Figure 7-7: Hardware latency breakdown

On Figure 7-8 we can see a bandwidth comparison between the TCP/IP over 10G and our advanced RDMA engine. We can see that the 10G implementation tops near 2.5Gbit and then has a minor drop in bandwidth. This is because the implementation does not have any tcp/ip

hardware offloading, and many functions of the stack have to be done by the processor which is not so fast and results in packets being dropped.

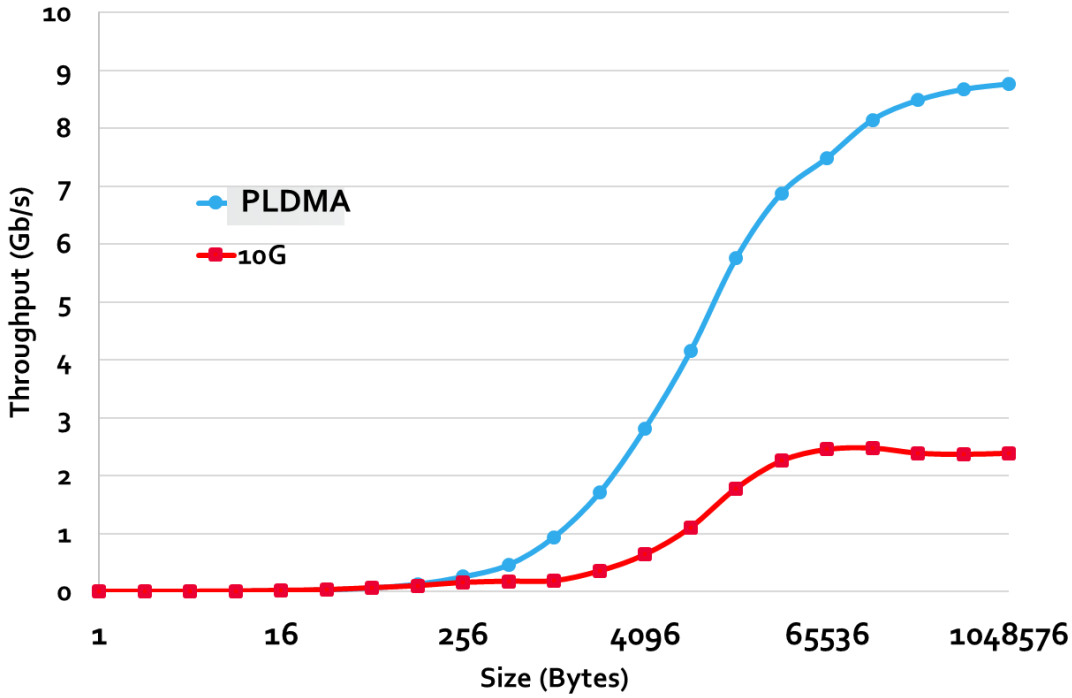


Figure 7-8: TCP/IP over 10G vs PLDMA throughput. OSU ,Bandwidth

7.1 Application-level Performance

In order to further evaluate our advanced RDMA engine we also run LAMMPS application. LAMMPS is a classical molecular dynamics (MD) code that models ensembles of particles in a liquid, solid, or gaseous state. It can model atomic, polymeric, biological, solid-state (metals, ceramics, oxides), granular, coarse-grained, or macroscopic systems using a variety of interatomic potentials (force fields) and boundary conditions. It can model 2d or 3d systems with only a few particles up to millions or billions. The application consists of many control messages, as well as large transfer sizes. The MPI implementation used uses the RDMA Read protocol.

Each FPGA contains four CPUS so the max CPU count of our measurements is 128. For every doubling of the execution nodes, we also double the problem size of the application so if the application where to scale perfectly we would expect to see no difference in the execution times (weak scaling). As show in Figure 7-8 already at eight nodes our advanced RDMA engine does better in comparison with 10G ethernet. That difference grows even larger as the nodes increase. The measurements indicate that our RDMA engine is at *8/16/32* nodes ***x1.6, x3.1 and x7.6*** times better than conventional 10G Ethernet.

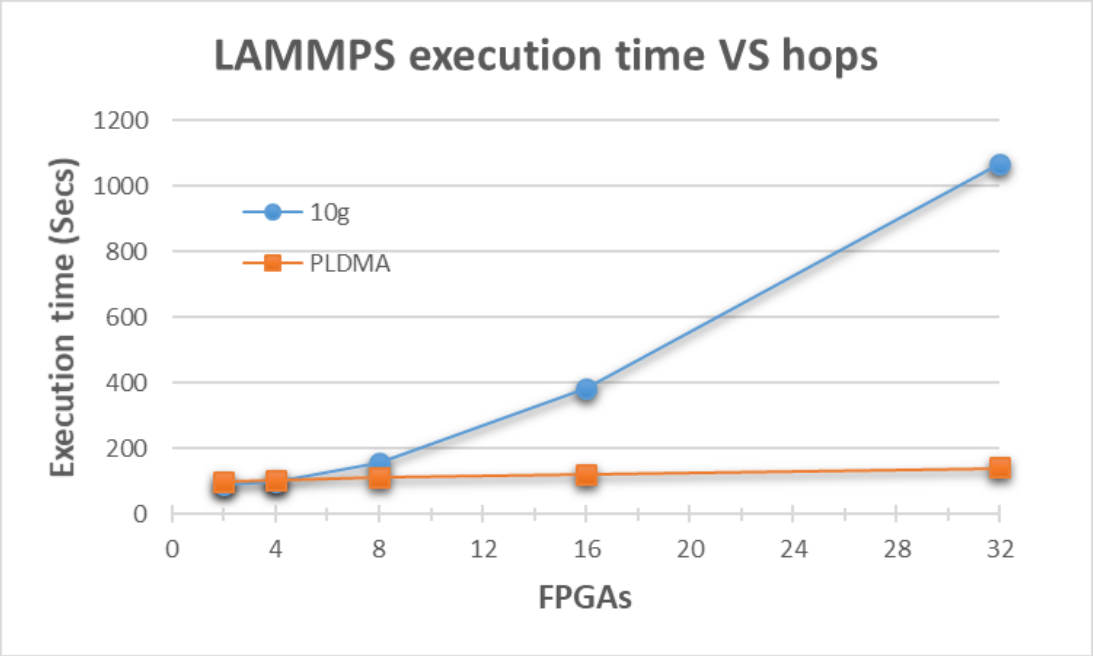


Figure 7-8: LAMPS application execution time PLDMA vs 10G

What is important to note is that the application seems to be scaling very well as the nodes increase and the execution time seems to more or less stay the same.

Our better performance when compared to 10G Ethernet is due to many reasons. Even for small nodes:

- Better throughput, Our RDMA engine can reach the full bandwidth allowed by the transceivers, while 10G ethernet can only get up to ~3Gbit.
- Lower latency, since our implementation is user-level zero copy, we do not have all the latency inherit from kernel involvement, and multiple data copies.

Furthermore, as the nodes increase the performance of ethernet greatly decreases while our RDMA engines remains the same. This is because TCP/IP over 10G requires allot of CPU involvement, while our RDMA engine completely offloads the CPUs. As the traffic coming from multiple nodes increases, so does the work that the CPU has to do for the TCP/IP stack. When traffic increases enough, the CPU is not able to respond fast enough, resulting in packet drops and retransmissions which further, congest the 10G network.

Another issue that we have to take into account when comparing with tcp/ip over 10G Ethernet is the “unfairness” in regards of network aggregate throughput.

As shown in Figure 7-9 and explained in detail in Chapter 6.4, each QFDB has 4 links of 10Gbit each, connected on a 2D torus. Furthermore, inside each QFDB the FPGAs are connected all-to-all and the Crossbar that routes the traffic inside the F1 FPGA can use 4 paths, on for each connected FPGA, to connect to the main network router. This means that each QFDB can potentially generate up to 40Gbit of traffic.

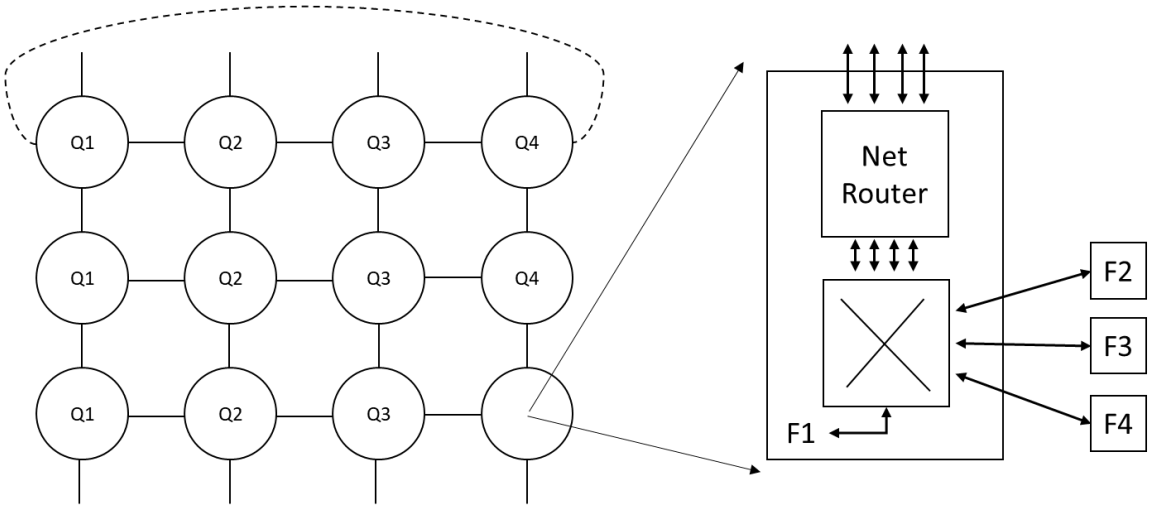


Figure 7-9: ExaNet Network Topology

10G Ethernet is also connected in all-to-all within the QFDB and uses a small Ethernet switch within the network (F1) FPGA to connect to the 10G MAC block. Even though each FPGA can generate ~3Gbit of traffic outgoing from the MAC to the network can still reach 10Gbit due to oversubscribing from the FPGAs within the QFDB if all FPGAs have traffic for the network.

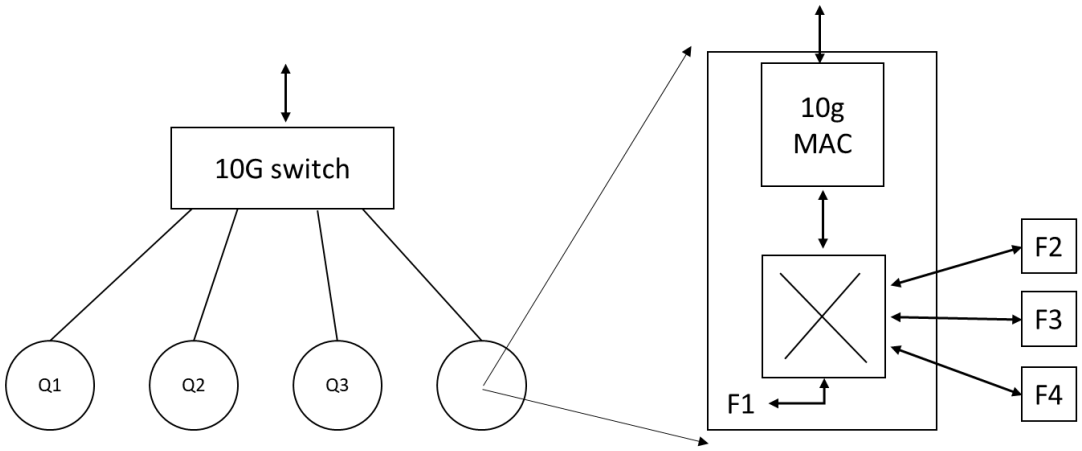


Figure 7-10: 10G Ethernet Topology

As shown in Figure 7-10 each QFDB directly connects to a 10G network switch and each switch can have up to 12 QFDBs connected to it. This means that using the ExaNet network, one application can have up to 40Gbit traffic going to the neighbors, while with our 10G Ethernet implementation, only 10Gbit. If we wanted to have a fairer comparison, we should modify the intra-qfdb crossbar to only use one path within the FPGA instead of four. That way the aggregate throughput going out of the FPGA would still be 10Gbit in the ExaNet network, regardless of how much traffic is generated within the QFDB .By Doing this we would be able to compare the actual topologies and the network interfaces in a more accurate way.

8 Conclusion and future work

Modern computing clusters consist of many heterogeneous computing units that work collectively in order to fulfill tasks that require high performance. Low latency communication between the remote processes that run on these servers is a critical factor for achieving high performance. In this work, we described the hardware implementation of an advanced RDMA engine send path, as well as the complete RDMA network that is used in the ExaNeSt prototype. Our advanced RDMA engine provides high throughput coupled with advanced Resiliency features.

Our Performance evaluation demonstrates great improvements in throughput and in overall application execution time when compared with traditional TCP/IP over 10G and previous RDMA implementations within ExaNeSt. Furthermore, our implementation allows for page-fault handling, multipathing and congestion management features.

One weakness of our RDMA engine when compared with our previous implementation (zDMA) is the increase of latency in small transfer sizes. This increase is expected when considering the great number of added features that were non-existent in zDMA. Many of those features are served by the R5 co-processor, which itself suffers from limited performance. One immediate update in our system would be to offload some of the work done by the R5 co-processor to the ExaDMA (RDMA send unit). Some of those features would be the receiving of acknowledgments and handling of Nacks / retransmissions.

One feature of our RDMA which we did not exploit in this work is multi-pathing. All the hardware blocks of the RDMA are multi-pathing capable, and the only thing missing is the support from the network, which we currently are working at.

Finally, work has been done to change the round robin scheduler of the RDMA send unit, with a priority heap that takes information from our congestion management infrastructure and serves each transaction according to the congestion of its destination.

Bibliography

[1]. Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (SIGCOMM '16). ACM, New York, NY, USA, 202-215. DOI: https://doi.org/10.1145/2934872.2934908

[2]. Xilinx. Zynq Ultrascale+ Technical Reference Manual, UG1085 January 17,2019

[3]. Durand Y, Carpenter PM, Adami S, Bilas A, Dutoit D, Farcy A, Gaydadjiev G, Goodacre J, Katevenis M, Marazakis M, Matus E. Euroserver: Energy efficient node for european micro-servers. In2014 17th Euromicro Conference on Digital System Design 2014 Aug 27 (pp. 206-213). IEEE.

[4]. Ammendola R, Guagnelli M, Mazza G, Palombi F, Petronzio R, Rossetti D, Salamon A, Vicini P. APENet: a high speed, low latency 3D interconnect network. Incluster 2004 Sep 20 (p. 481).

[5]. Dimitris Giannopoulos, Nikos Chrysos, Evangelos Mageiropoulos, Giannis Vardas, Leandros Tzanakis, and Manolis Katevenis. 2018. Accurate congestion control for RDMA transfers. In Proceedings of the Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS '18). IEEE Press, Piscataway, NJ, USA, Article 3, 8 pages.As

[6]. Pantelis Xirouchakis, Panagiotis Peristerakis, Michalis Gianoudis, Antonis Psistakis, Giorgos Kalokerinos, Nikos Chrysos, Vassilis Papaefstathiou, and Manolis G.H. Katevenis. Low Latency RDMA for High-Performance Computing on ARM Platforms. Fiuggi, Italy: HiPEAC ACACES 2017