# Efficient Network Interface design for low cost distributed systems

**Vasileios Amourgianos Lorentzos**

Professor Dionisios Pnevmatikatos

Associate Professor Yannis Papaefstathiou

Dr. Dimitris Theodoropoulos

School of Electronics and Computer Engineering

Technical University of Crete

October 2017

# Acknowledgements

I would like to thank my supervisor, Professor Dionysios Pnevmatikatos for his guidance and support, as well as for the opportunity to work on cutting edge technology development, knowing that my work will be used for further advances in the field.

I would also like to express my gratitude to Dr. Yannis Papaefstathiou and Dr. Dimitris Theodoropoulos for their interest in my work and for contributing to its evaluation as members of the thesis committee.

Finally, I would like to thank my family for their support over the years.
This thesis is dedicated to them.

This work is partially supported by the European Union H2020 program through the AXIOM project (grant ICT-01-2014 GA 645496).

# Abstract

Cyber-Physical Systems (CPSs) are widely used in many applications that require physical inputs and outputs. A CPS usually combines a set of hardware-software components to achieve optimal application execution in terms of performance and energy consumption. An important ability and requirement of CPSs is to be scalable through modularity, making them cost effective and providing enough computational power for the assigned tasks.

This thesis is the result of our efforts to design and implement a cost-effective yet efficient Network Interface (NI) for the AXIOM CPS board and system to achieve said modularity. The NI has to achieve high throughput, with remote access to the memory, and several different transfer types to accommodate the project's needs. It also has to be cost effective, with the minimum possible usage of the board's available resources and follow certain guidelines for easy connectivity.

# Table of contents

# List of figures

# List of tables

# Nomenclature

**Acronyms / Abbreviations**

| | |
|---|---|
| ACK | Acknowledged |
| BRAM | Block Random-Access Memory |
| BSP | Board Support Package |
| CPS | Cyber-Physical System |
| DMA | Direct Memory Access |
| DST Addr | Destination Address |
| E | Error |
| FIFO | First In First Out |
| FPGA | Field-Programmable Gate Array |
| GIC | Generic Interrupt Controller |
| IDE | Integrated Development Environment |
| ID | Identifier |
| ILA | Integrated Logic Analyzer |
| INIT | Initialize |
| IP | Intellectual Property |
| JTAG | Joint Test Action Group |
| LUT | Look-Up Table |

| | |
|---|---|
| N/U | Not Used |
| NACK | Not Acknowledged |
| NI | Network Interface |
| PS | Processing System |
| RDMA | Remote Direct Memory Access |
| RO | Read Only |
| RW | Read - Write |
| Rx | Receive |
| SDK | Software Development Kit |
| SRC Addr | Source Address |
| S | Sender |
| Tx | Transmit |
| UART | universal asynchronous receiver-ransmitter |
| cc | clock cycle |
| GUI | graphical user interface |
| IDE | Integrated Design Environment |
| Tcl | tool command language |
| VC | Virtual Channel |
| WO | Write Only |
| XSCT | Xilinx Software Command-Line Tool |

# Chapter 1

# Introduction

In recent years, a key research area are Cyber-Physical Systems, as they are tightly integrated with the Internet and their users, allowing a rapid and close interaction not only between themselves, but also between them and humans. In CPSs, physical and software components are deeply intertwined, with the expectation that such systems will at least react in real-time, provide enough computational power for the assigned tasks, consume the least possible energy, scale up through modularity and allow for an easy programmability across performance scaling. Various projects focus on CPSs. One of them, the AXIOM project aims at researching new software and hardware architectures for CPSs to meet the above expectations.

## 1.1 Motivation

This thesis was conceived as part of the AXIOM project, AXIOM being the abbreviation for Agile, eXtensible, fast I/O Module. The AXIOM project aspires to create a European-designed and -manufactured single board computer [1.1] that has great flexibility, energy efficiency and modularity, because of a low-power CPU, an integrated low-power FPGA, and a fast-and-cheap interconnect, the subject of this thesis, using the USB type-C connectors. The programming model for the AXIOM board is the OmpSs from Barcelona Supercomputing Center, offering improved thread management, using the BSC Mercurium compiler, and running on Linux, with drivers provided as open-source by the project's partners. The AXIOM board also has support for Arduino pluggable boards, called shields, to extend it's functionality and is provided by SECO.

This thesis is the result of our efforts to design and implement a cost-effective yet efficient Network Interface (NI) for the AXIOM CPS board and system to achieve said modularity. The NI has to achieve high throughput, with remote access to the memory, and several different transfer types to accommodate the project's needs. It also has to be cost effective, with the minimum possible usage of the board's available resources and follow certain guidelines for easy connectivity.

Fig. 1.1 AXIOM board

## 1.2 Thesis Contributions

The Network Interface we designed along with a Router, also designed as part of the FORTH AXIOM program, constitute the Axiom Interconnect. The Network Interface is inspired by the FORMIC project, but includes the necessary functionality to facilitate the needs of the OmpSs programming model. Those features include several transfer types, that include Remote Direct Memory Access (RDMA) transfers. Another important feature is the well thought out Control and Status Registers and the Interrupt system.

To implement this design, we create three different message controllers, an interrupt controller, as well as the needed interconnects. We utilize the AXI4 and AXI4-Stream interfaces provided by Xilinx as well as the provided IPs to implement the FIFOs, the registers and the DMA engine for our design.

The final result is a working, efficient Network Interface, using a lot of the available bandwidth, with low usage of the available FPGA resources.

Notably, the NI:

- Supports multiple transfer types
- Supports RDMA transfers
- Has a fully fledged interface to interact with the PS
- Is pipelined for minimal latency and parallel functions

## 1.3 Thesis Outline

The remainder of this work is organized as follows. In Chapter 2 we review similar work in whole, or in part to this thesis, and find out what comparisons can be made to our work. In Chapter 3 we describe the Network Interface Architecture we designed, as well as the intended functionality. Chapter 4 provides a description of the tools used, the implementation details of our design, and the resources utilized. In Chapter 5 we analyze the hardware used and the results we attained. Finally, Chapter 6 entails the conclusions drawn from this work, as well as proposals for improvements to the efficiency and cost of our work.

# Chapter 2

# Related work

As multi- and many-core platforms are becoming an important research subject with scalability in mind, there has been a need for efficient network interfaces. Two works that research network interfaces in multi-chip platforms are the FORMIC project and the SARC project.

The SARC project's [4] approach to network interfaces for many-core platforms is to merge the cache controller and the network interface functions by relying on a single hardware primitive. Depending on the access type and the state of the addressed line, different responses may be triggered like coherence actions, RDMA, synchronization, or configurable event notifications. To test the design, benchmark kernels were simulated on a full-system simulator to compare speedup and network traffic against cache-only systems with directory-based coherence and prefetchers. The results were 10 to 40% higher speedup on 64 cores, and a reduction of network traffic by factors of 2 to 4, thus economizing on energy and power.

S. Kavadias et Al [5][6] propose the use of scratch-pad memories, since they allow direct inter-core communication, with latency and energy advantages over coherent cache-based communication. They designed a cache-integrated network interface to be used in multicore architectures, that combine the flexibility of caches and the efficiency of scratchpad memories. To create the NI they shared the SRAM configurably among caching, scratchpad, and virtualized NI functions. Their architecture provides local and remote scratchpad access, to either individual words or multi-word blocks through RDMA copy. It also supports event responses as a mechanism for software configurable synchronization primitives. To test their design, they implemented it in a four-core FPGA

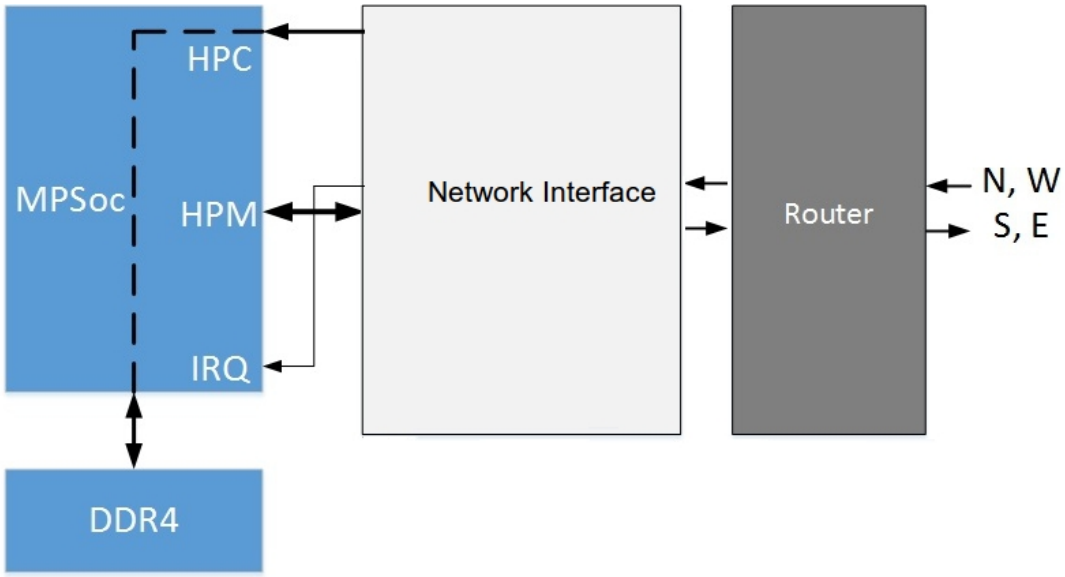
prototype, and evaluated the on-chip communication performance on the prototype as well as on a CMP simulator with up to 128 cores. The result was efficient synchronization, low-overhead communication, and amortized-overhead bulk transfers, which allow parallelization gains for fine-grain tasks, and efficient exploitation of the hardware bandwidth.

Another approach is the FORMIC [8] project. FORMIC is a novel hardware prototype board designed specifically to be a cost-efficient building block for scalable systems. It is minimal in concept, small, has both SRAM and DRAM memories, features convenient SATA connectors and is optimized to be a part of a larger system. The Formic board has a 35% bigger FPGA, three times more SRAM, four times more links and costs at most half as much when compared to the popular Xilinx XUPV5 prototyping platform. Each FORMIC board consists of a Xilinx Spartan-6 LX150T FPGA, three Cypress 9-Mbit 166-MHz ZBT SRAMs and a single Micron 1-Gbit 400-MHz DDR2 SDRAM chip. Each SRAM offers a raw bandwidth of 5.3 Gbps and the DRAM has a peak bandwidth of 12.8 Gbps. To test the Formic board, a hardware design project based on the SARC project was used. A full network-on-chip centered around a 22-port crossbar was designed for scalability. A 64-board system with 512 cores that is MicroBlaze-based, non-coherent, with DMA capabilities and with full network-on-chip in a 3D-mesh topology was built and tested. This hardware architecture was used in the ENCORE project, as the basis of a manycore, task-based runtime system.

# Chapter 3

# NI architecture

The NI we designed and implemented has to communicate with the Processing System (PS) of the AXIOM board, through four FIFOs, two Rx and two Tx, to send descriptors that the NI transforms to packets, or receive descriptors of the incoming packets. It also communicates with the Router, by sending or receiving packets.




Two basic functions are supported, the transfer of data from and to the PS, from now on referred to as Raw transfer category, and the transfer of data directly from and to the memory of the PS with the use of a DMA engine, from now on referred to as RDMA transfer category, as shown in figure 3.1. To initiate a transfer, the PS sends the corresponding descriptor to the NI. The Raw transfers are single packet transfers with a max payload of 255 Bytes. The RDMA transfers allow for a max payload of

64 KBytes per transfer, as they are comprised of multiple packets. Another advantage RDMA transfers have over Raw transfers is that the payload is read and written without interaction with the PS, other than writing a Tx Descriptor to start the transfer, and reading an Rx Descriptor after completion. The RAW transfer category includes the RAW_NEIGHBOUR and RAW_DATA transfers, and the RDMA transfer category includes the LONG_DATA, RDMA_WRITE and RDMA_READ transfers.

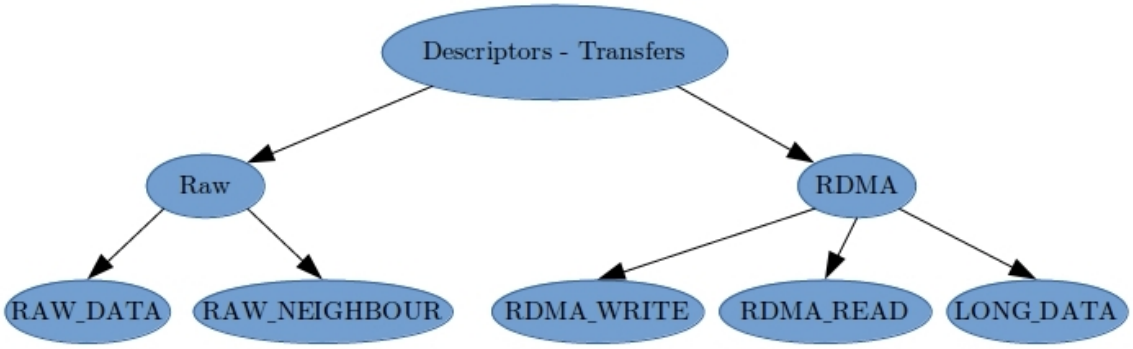



Fig. 3.1 Available transfers - descriptors

For every transfer, the PS communicates with the NI by writing a Tx descriptor in the corresponding Tx FIFO to start a transfer, or reading one from the corresponding Rx FIFO to learn about completed transfers. The NI is responsible for transforming those descriptors to packets on the sender side, and re-transforming them to descriptors on the receiver side. For RAW transfers the payload is included in these descriptors, whereas for RDMA transfers only the address offset and size of the payload in the memory is included. This way the PS does not have to send the payload to the NI to start the transfer, and after a completed transfer can read the payload from the memory when needed.

To control the network traffic three different priorities are used, high, medium and low, each with its own Virtual Channel (VC). High priority Packets are assigned to VC 2, Medium priority Packets are assigned to VC 1 and Low priority Packets are assigned to VC 0.

The NI is comprised of the Tx channel, the Rx channel, the Interrupt Handler, a DMA engine, the Control Registers and the Status Registers. Figure 3.2 shows the architecture

of the NI we designed. The Tx Controllers receive the Tx descriptors, transform them to packets and send them to the Vc FIFOs and then to the Router. The Rx Controller receives the packets and generates the Rx descriptors. Whenever one of the Tx FIFOs, while previously full, gets to have empty space, or one of the Rx FIFOs, while previously empty, gets to have data, an interrupt is generated to inform the PS. Finally, the DMA engine is responsible for reading and writing the payload for RDMA transfers.

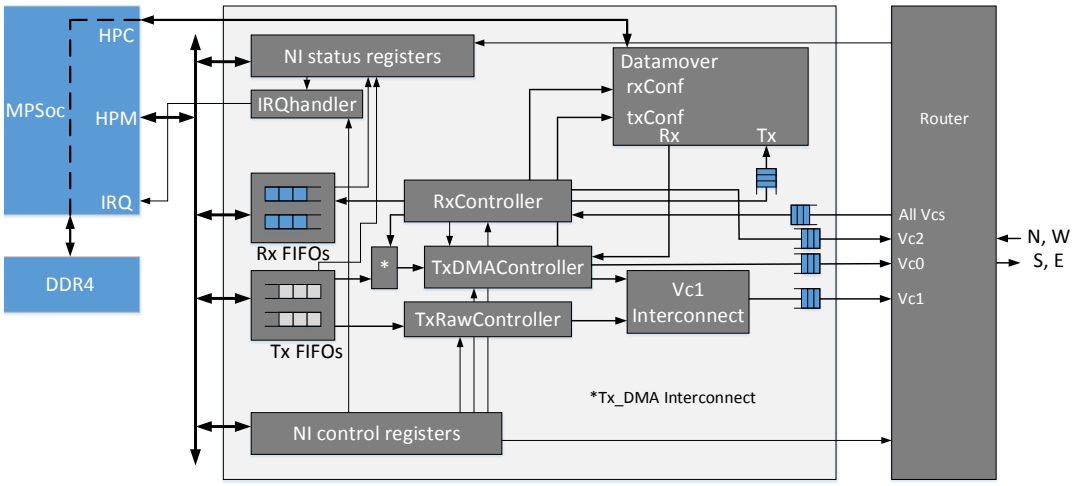



Fig. 3.2 NI architecture

## 3.1 Supported transfers

To start a transfer, the PS has to send a descriptor to the NI. These descriptors are shown in table 3.1. For the RAW transfers, the Tx and Rx descriptors have the same structure. The Raw descriptors have a header with the necessary information for the transfer, and the payload to be sent. The Tx descriptors have a DESTINATION field, while the Rx descriptors have a SOURCE field. For the RDMA transfers, the Tx descriptor contains not only the header, to start the transfer, but also the information needed to get the payload from the memory, while the descriptor itself does not contain any payload. The Rx descriptor that is written in the receiver node contains the necessary information for the PS to get that payload from the memory. An Rx descriptor is also written in the sender node, and is used to inform the PS if the transfer was successful or not. For the LONG_DATA transfers, it is not needed for the sender PS to specify where to write the transmitted payload because that is handled by the receiver PS. There are six main fields in the descriptor of each packet:

1. The TYPE field is 3 bits wide and its value is the packet's type.
2. The PORT field is 3 bits wide and is used by the PS, for example, to encapsulate Ethernet packets.
3. The DESTINATION field is 8 bits wide and its value is either the Physical Interface it will be sent from for RAW_NEIGHBOUR transfers or the receiving Node id for all the other transfers.
4. The SOURCE field is 8 bits wide and its value is either the Physical Interface it will be received from for RAW_NEIGHBOUR transfers or the sending Node id for all the other transfers. This value is written in the SOURCE register of the NI after a discovery protocol is complete and is read by the Tx Controllers to generate the Headers.
5. The MESSAGE ID field is 8 bits wide and its value is an id given by the PS that together with the SOURCE and DESTINATION fields create a unique id for each transfer.
6. The PAYLOAD SIZE field is 8 bits wide for RAW descriptors and 16 bits wide for RDMA and LONG_DATA descriptors and its value is the size of the payload that will be transmitted in Bytes.

LONG_DATA and RDMA descriptors have four extra fields depending on whether they are written or read by the PS:

1. The SRC Addr field is 32 bits wide and its value is the address offset of the memory that the DMA transfer will start from in the local node.
2. The DST Addr field is 32 bits wide and its value is the address offset of the memory that the DMA transfer will start from in the remote node.
3. The S field, short for Sender, is 1 bit wide and specifies if this descriptor's recipient is the sender of the transfer or not.
4. The E field, short for Error, is 1 bit wide and specifies if the transfer completed with an error or not.

The packets, as shown in table 3.2, contain the information extracted from the corresponding descriptors, but include two other fields necessary for the transfer and the payload:

1. The VC field is 2 bits wide and its value is the priority of the packet depending on the type. ACK packets are of High priority, RDMA_READ packets are of Low priority and all the other Packets are of Medium priority.

The ACK and INIT packets are not data packets, but control packets, and do not contain payload. The ACK packet also contains the ACK Type field with valid types: RDMA_WRITE, RDMA_READ and LONG_DATA, and the NACK field which specifies if it is a NACK or an ACK. Finally, the INIT packet contains the INIT Type field with valid types: RDMA_WRITE, RDMA_READ, and LONG_DATA.

Table 3.1 descriptors

RAW_NEIGHBOUR / RAW_DATA descriptor

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| N/U | PORT | TYPE |
| SRC / DST | | |
| MESSAGE ID | | |
| PAYLOAD SIZE | | |
| N/U | | |
| PAYLOAD Byte 0 | | |
| ... | | |
| PAYLOAD Byte n | | |

LONG_DATA Tx descriptor

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| N/U | PORT | TYPE |
| DESTINATION | | |
| MESSAGE ID | | |
| PAYLOAD SIZE LOW | | |
| PAYLOAD SIZE HIGH | | |
| SRC Addr 0 | | |
| SRC Addr 1 | | |
| SRC Addr 2 | | |
| SRC Addr 3 | | |

RDMA_WRITE / RDMA_READ Tx descriptor

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| N/U | PORT | TYPE |
| DESTINATION | | |
| MESSAGE ID | | |
| PAYLOAD SIZE LOW | | |
| PAYLOAD SIZE HIGH | | |
| SRC Addr 0 | | |
| SRC Addr 1 | | |
| SRC Addr 2 | | |
| SRC Addr 3 | | |
| DST Addr 0 | | |
| DST Addr 1 | | |
| DST Addr 2 | | |
| DST Addr 3 | | |

LONG_DATA / RDMA_WRITE / READ Rx descriptor receiver side

| ⟵ 1 Byte ⟶ | | | |
|---|---|---|---|
| S | E | PORT | TYPE |
| SOURCE | | | |
| MESSAGE ID | | | |
| PAYLOAD SIZE LOW | | | |
| PAYLOAD SIZE HIGH | | | |
| DST Addr 0 | | | |
| DST Addr 1 | | | |
| DST Addr 2 | | | |
| DST Addr 3 | | | |

LONG_DATA / RDMA_WRITE / READ Rx descriptor sender side

| ⟵ 1 Byte ⟶ | | | |
|---|---|---|---|
| S | E | PORT | TYPE |
| DESTINATION | | | |
| MESSAGE ID | | | |
| PAYLOAD SIZE LOW | | | |
| PAYLOAD SIZE HIGH | | | |

Table 3.2 packets

RAW packet

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| SOURCE | | |
| DESTINATION | | |
| MESSAGE ID | | |
| VC | PORT | TYPE |
| PAYLOAD SIZE | | |
| PAYLOAD Byte 0 | | |
| ... | | |
| PAYLOAD Byte n | | |

ACK packet

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| SOURCE | | |
| DESTINATION | | |
| MESSAGE ID | | |
| VC | PORT | TYPE |
| PAYLOAD SIZE LOW | | |
| PAYLOAD SIZE HIGH | | |
| NACK | ACK Type | |

INIT packet

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| SOURCE | | |
| DESTINATION | | |
| MESSAGE ID | | |
| VC | PORT | TYPE |
| PAYLOAD SIZE LOW | | |
| PAYLOAD SIZE HIGH | | |
| INIT Type | | |

LONG_DATA packet

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| SOURCE | | |
| DESTINATION | | |
| MESSAGE ID | | |
| VC | PORT | TYPE |
| PACKET PAYLOAD | | |
| PAYLOAD Byte 0 | | |
| ... | | |
| PAYLOAD Byte n | | |

RDMA_WRITE / RESPONSE packet

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| SOURCE | | |
| DESTINATION | | |
| MESSAGE ID | | |
| VC | PORT | TYPE |
| PACKET PAYLOAD | | |
| DST Addr 0 | | |
| DST Addr 1 | | |
| DST Addr 2 | | |
| DST Addr 3 | | |
| PAYLOAD Byte 0 | | |
| ... | | |
| PAYLOAD Byte n | | |

RDMA_READ packet

| ⟵ 1 Byte ⟶ | | |
|---|---|---|
| SOURCE | | |
| DESTINATION | | |
| MESSAGE ID | | |
| VC | PORT | TYPE |
| PAYLOAD SIZE LOW | | |
| PAYLOAD SIZE HIGH | | |
| SRC Addr 0 | | |
| SRC Addr 1 | | |
| SRC Addr 2 | | |
| SRC Addr 3 | | |
| DST Addr 0 | | |
| DST Addr 1 | | |
| DST Addr 2 | | |
| DST Addr 3 | | |

## RAW_NEIGHBOUR

The RAW_NEIGHBOUR transfer is mainly used along with a discovery protocol that is run by the main FPGA of the ring or 2-d mesh to locate all the other FPGAs, assign ids to them and initialize their Routing Tables. Since at initialization the Routing Table for each FPGA cannot be used, RAW_NEIGHBOUR packets get their DESTINATION in the form of the Physical Interface they are going to be sent by. When received by the next FPGA the Router assigns to them the Physical Interface they were received by as SOURCE and forwards them to the NI. RAW_NEIGHBOUR transfers have a maximum payload of 255 Bytes, and are of Medium priority.

The RAW_NEIGHBOUR transfer is fairly straightforward. Figure 3.3 is an example of a RAW_NEIGHBOUR transfer. The PS writes the descriptor in the Tx_RAW FIFO, the Tx_RAW Controller reads the descriptor and creates the packet. The packet is forwarded to the Router, through which it is sent to the next node. The receiving router then sends the packet to the Rx Controller, which creates the Rx descriptor and writes it to the Rx_RAW FIFO. The receiving PS is informed with an interrupt that the Rx_RAW FIFO is not empty and can at any time read the descriptor. RAW_NEIGHBOUR transfers always go from one node to nodes directly connected to it and cannot hop from node to node.

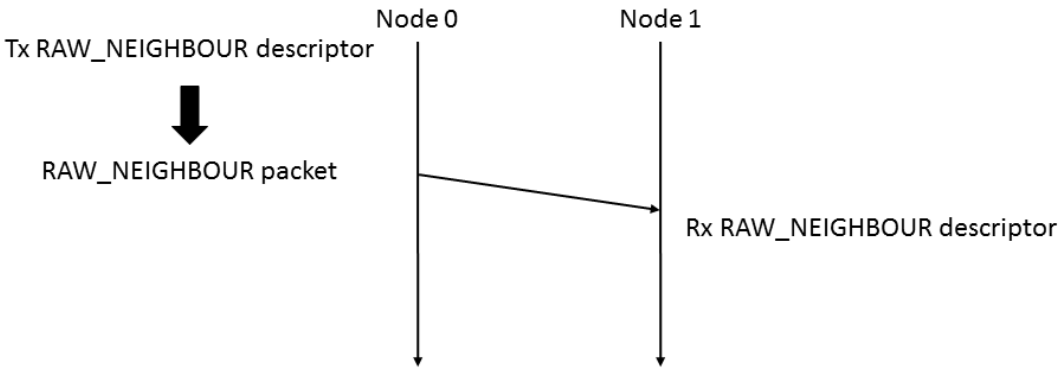



Fig. 3.3 RAW_NEIGHBOUR transfer

## RAW_DATA

The RAW_DATA transfer is used to send data directly from one PS to another. RAW_DATA transfers have a maximum payload of 255 Bytes, and are of Medium priority.

The RAW_DATA transfer is similar to the RAW_NEIGHBOUR transfer. Figure 3.4 is an example of a RAW_DATA transfer. The PS writes the descriptor in the Tx_RAW FIFO, the Tx_RAW Controller reads the descriptor and creates the packet. The packet is forwarded to the Router, through which it is sent to the next node. The receiving Router then checks if it needs to forward the packet to the NI or the next node of the network. When the packet reaches its destination, it is sent to the Rx Controller, which creates the Rx descriptor and writes it to the Rx_RAW FIFO. The receiving PS is informed with an interrupt that the Rx_RAW FIFO is not empty and can at any time read the descriptor.

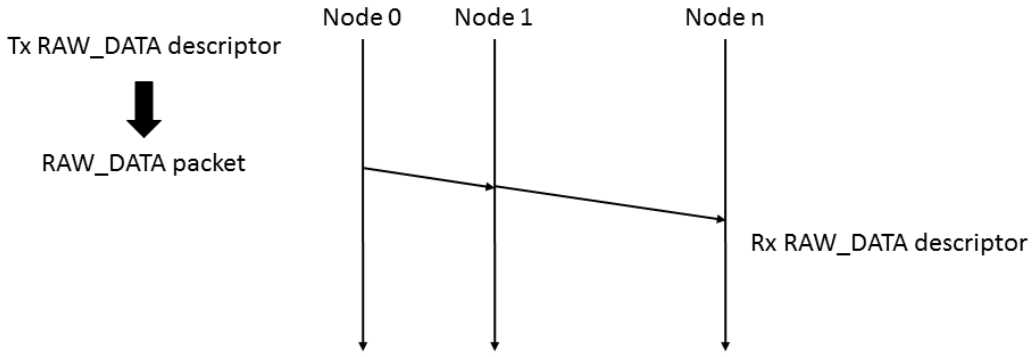



Fig. 3.4 RAW_DATA transfer

## LONG_DATA

The LONG_DATA transfer is used to send data from one node's memory to another's, and as such it belongs in the RDMA category. LONG_DATA transfers have a maximum payload of 64 KBytes and are of Medium priority. They are comprised of three different types of packets, the INIT packet to start the transfer, the LONG_DATA packets to transfer the payload 248 Bytes at a time and the ACK/NACK packet to acknowledge or reject the transfer. LONG_DATA descriptors only include the payload size and the source address of the memory to read the payload but not the destination address to write it. That is because, for this type of transfer, a table in the form of BRAM memory exists that is written by the PS and informs the Rx Controller for where to write incoming LONG_DATA transfers according to payload size. If an empty space for the transfer is not found, the transfer is rejected by the receiver.

As with the previous transfers, first the PS writes the descriptor in the Tx_DMA FIFO, the Tx_DMA Controller reads the descriptor and creates the INIT packet. If node 1 is the sender and node n the receiver, after node's n Rx Controller receives the INIT packet, it tries to allocate the necessary resources, such as the counter for the payload, the destination address and so on. If it is unable to do so the transfer is rejected.

In the case of a rejected transfer, as shown in figure 3.5, it generates a NACK packet and sends it back. Meanwhile, node's 1 Tx_DMA Controller has already started a local DMA read, generating and sending LONG_DATA packets. When the NACK with the current transfer's Message ID is received by node's 1 Rx Controller, a descriptor with S = 1 and E = 1 is written in the Rx_DMA FIFO and the Tx_DMA Controller is informed about the NACK. It then completes the latest packet, empties the DMA read FIFO and terminates the transfer. Any packets already sent are skipped by node's n Rx Controller, as their id does not belong to an active transfer.

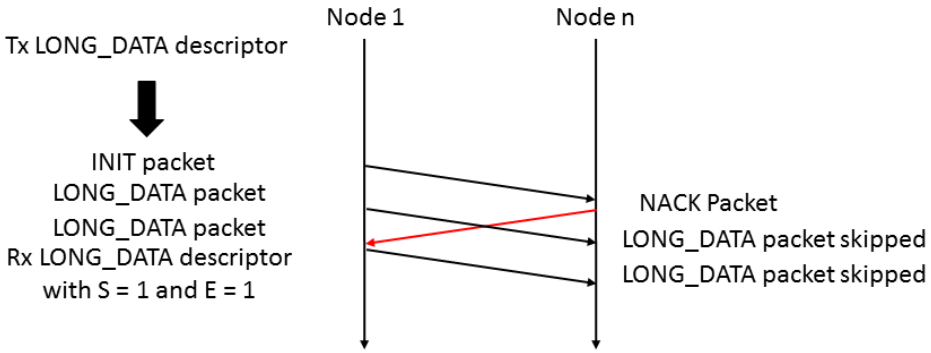



Fig. 3.5 LONG_DATA rejected transfer

Otherwise, if node's n Rx Controller allocates the resources it needs successfully, as with figure 3.6, a NACK is not sent and node's 1 Tx_DMA Controller generates packets until the DMA read FIFO empties. Every time node's n Rx Controller receives a LONG_DATA packet with this transfer's Message ID, it starts a local DMA write for the payload, updates the counters and checks if the transfer counter reached zero. If so, all the payload is received, and it generates an ACK packet, sends it back to node 1 and writes a descriptor in the Rx_DMA FIFO. When node's 1 Rx Controller receives the ACK, it also writes a descriptor in the Rx_DMA FIFO with S = 1 and the transfer is completed successfully.

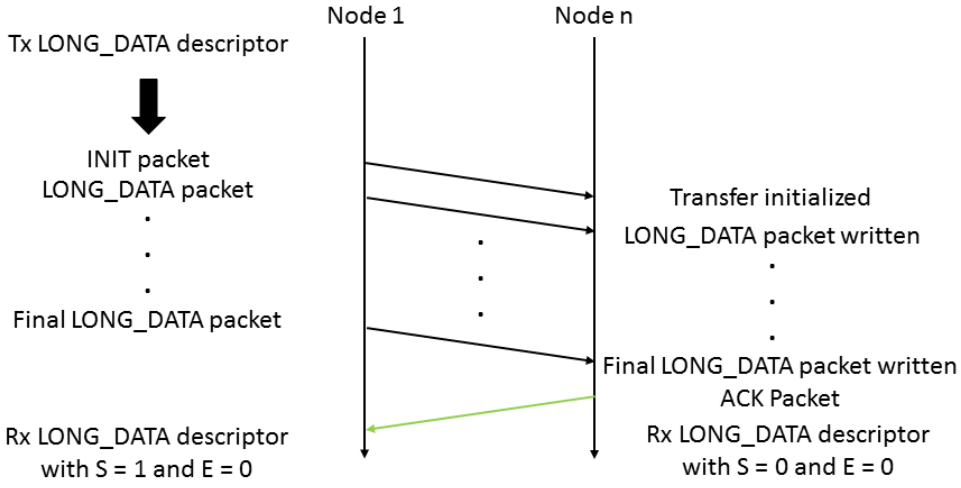



Fig. 3.6 LONG_DATA successful transfer

## RDMA_WRITE

The RDMA_WRITE transfer is very similar to the LONG_DATA transfer. RDMA_WRITE transfers also have a maximum payload of 64 KBytes and are of Medium priority. They too are comprised of three different types of packets, the INIT packet to start the transfer, the RDMA_WRITE packets to transfer the payload 252 Bytes at a time and the ACK/NACK packet to acknowledge or reject the transfer. The key difference is that RDMA_WRITE descriptors include the payload size, the source address of the memory to read the payload from at the sender's side and the destination address to write it at the receiver's side. RDMA_WRITE transfers can also be rejected since only 32 RDMA transfers can be active at a time. This means that if the network has more than 33 nodes, and more than 32 of them begin RDMA transfers on the same node at the same time, any more than the first 32 will be rejected.

As with the previous transfers, first the PS writes the descriptor in the Tx_DMA FIFO, the Tx_DMA Controller reads the descriptor and creates the INIT packet. If node 1 is the sender and node n the receiver, after node's n Rx Controller receives the INIT packet, it tries to allocate the necessary resources, such as the counter for the payload, the destination address and so on. If it is unable to do so the transfer is rejected.

In the case of a rejected transfer, as shown in figure 3.7, it generates a NACK packet and sends it back. Meanwhile, node's 1 Tx_DMA Controller has already started a local DMA read, generating and sending RDMA_WRITE packets. When the NACK with the current transfer's Message ID is received by node's 1 Rx Controller, a descriptor with S = 1 and E = 1 is written in the Rx_DMA FIFO and the Tx_DMA Controller is informed about the NACK. It then completes the latest packet, empties the DMA read FIFO and terminates the transfer. Any packets already sent are skipped by node's n Rx Controller, as their id does not belong to an active transfer.

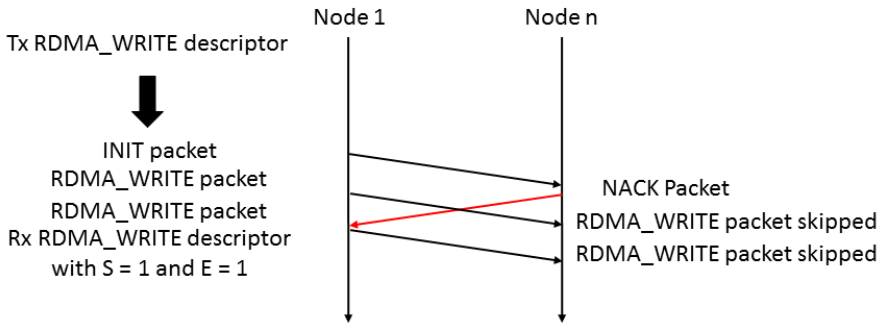



Fig. 3.7 RDMA_WRITE rejected transfer

Otherwise, if node's n Rx Controller allocates the resources it needs successfully, as with figure 3.8, a NACK is not sent and node's 1 Tx_DMA Controller generates packets until the DMA read FIFO empties. In every packet's destination address the offset for the payload already sent is added. Every time node's n Rx Controller receives an RDMA_WRITE packet with this transfer's Message ID, it starts a local DMA write for the payload using the destination included in the packet, updates the counters and checks if the transfer counter reached zero. If so, it generates an ACK packet, sends it back to node 1 and writes a descriptor in the Rx_DMA FIFO. When node's 1 Rx Controller receives the ACK, it also writes a descriptor in the Rx_DMA FIFO with S = 1 and the transfer is completed successfully.

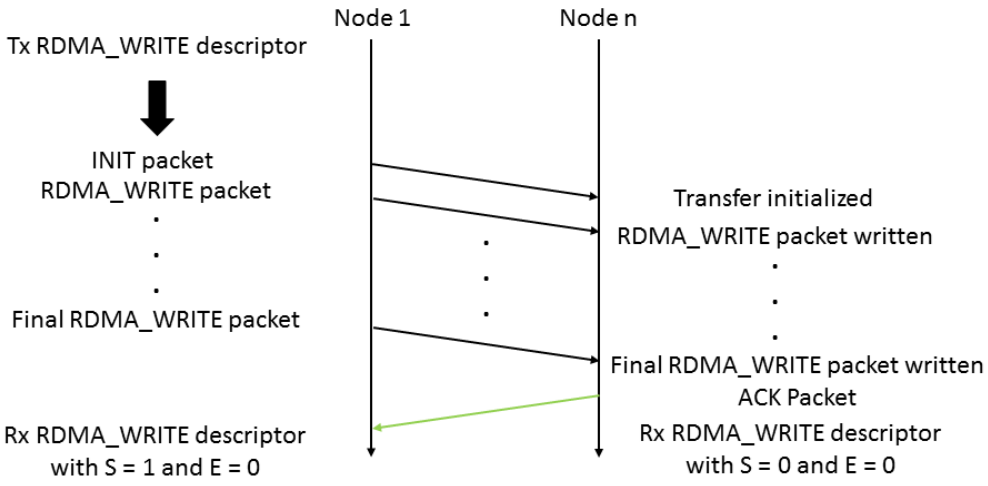



Fig. 3.8 RDMA_WRITE successful transfer

## RDMA_READ

The RDMA_READ transfer is used to receive data from a remote node instead of sending. RDMA_READ transfers also have a maximum payload of 64 KBytes, and are of Medium priority. They are comprised of four different types of packets, the RDMA_READ packet to request a transfer from the remote node, the INIT packet to start the transfer, the RDMA_RESPONSE packets to transfer the payload 252 Bytes at a time and the ACK/NACK packet to acknowledge or reject the transfer. RDMA_READ descriptors specify both the source address and the destination address, as do RDMA_WRITE descriptors, only this time the source address is to read data from the remote node and the destination address is to write the data in the local node. RDMA_READ transfers can be rejected.

As with the previous transfers, first the PS writes the descriptor in the Tx_DMA FIFO, the Tx_DMA Controller reads the descriptor and creates the RDMA_READ packet. If node 1 is the sender and node n the receiver, after node's n Rx Controller receives the RDMA_READ packet, it creates an RDMA_RESPONSE descriptor and writes it to the Tx_DMA FIFO. Node's n Tx_DMA controller then creates an INIT packet and sends it to node 1.

In the case of a rejected transfer, as shown in figure 3.9, node 1 generates a NACK packet and sends it back. Since node 1 initiated the RDMA_READ transfer, a descriptor of type RDMA_READ with S = 1 and E = 1 is written in the Rx_DMA FIFO. Meanwhile, node's n Tx_DMA Controller has already started a local DMA read, generating and sending RDMA_RESPONSE packets. When the NACK with the current transfer's Message ID is received by node's n Rx Controller the Tx_DMA Controller is informed about the NACK. It then completes the latest packet, empties the DMA read FIFO and terminates the transfer. Any packets already sent are skipped by node's 1 Rx Controller, as their id does not belong to an active transfer. Node's n PS is not informed with a descriptor about the failed transfer.

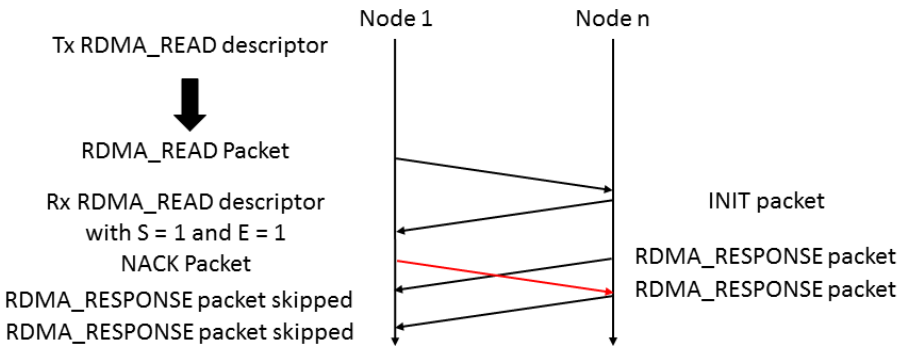



Fig. 3.9 RDMA_READ rejected transfer

Otherwise, if node's 1 Rx Controller allocates the resources it needs successfully, as with figure 3.10, a NACK is not sent and node's n Tx_DMA Controller generates packets until the DMA read FIFO empties. In every packet's destination address the offset for the payload already sent is added. Every time node's 1 Rx Controller receives an RDMA_RESPONSE packet with this transfer's Message ID, it starts a local DMA write for the payload using the destination included in the packet, updates the counters and checks if the transfer counter reached zero. If so, it generates an ACK packet, sends it back to node n and writes a descriptor of type RDMA_READ with S = 1 in the Rx_DMA FIFO. When node's n Rx Controller receives the ACK the transfer is completed successfully. Node's n PS is not informed with a descriptor about the successful transfer.

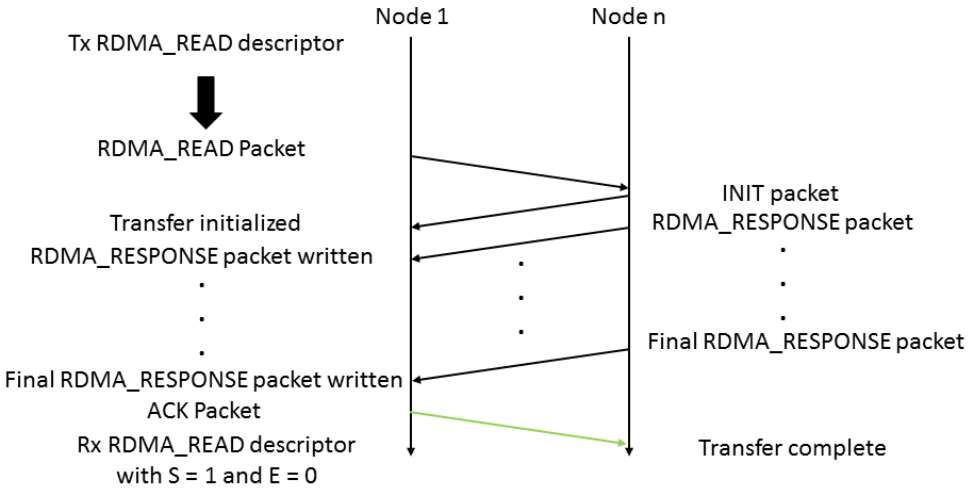
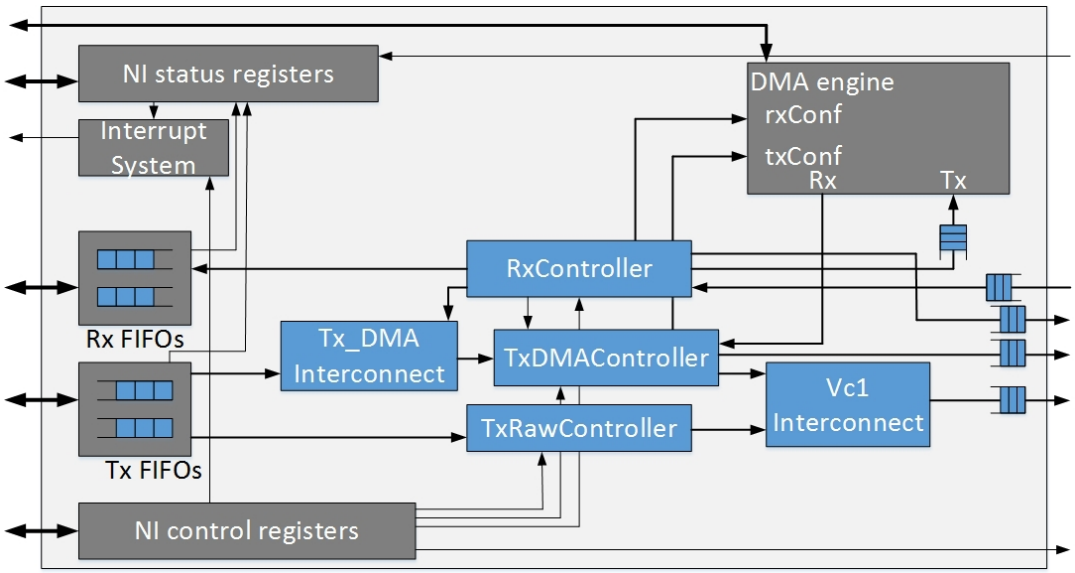



Fig. 3.10 RDMA_READ successful transfer

## 3.2 Controllers

The Controllers are the most important part of the NI, as they handle the transfers. Each controller is optimized to perform a certain task with minimum latency and resources. IN earlier iterations the two Tx Controllers were implemented together as one Controller. But as RDMA transfers generate many packets per transfer, while RAW transfers generate a single packet, having a Controller for each transfer allows for RAW packets to be sent in-between RDMA packets instead of waiting. That is very useful since the destination can be different and that way RAW transfers are not delayed by RDMA transfers.




### Tx_RAW Controller

The Tx_RAW Controller handles the RAW_NEIGHBOUR and RAW_DATA transfers. It reads descriptors from the Tx_RAW FIFO and generates RAW_NEIGHBOUR or RAW_DATA packets. To generate the RAW_DATA packets it also reads the Node id from the corresponding Status Register. RAW transfers are of medium priority, so the Tx_RAW Controller's output is sent to the Vc1 FIFO through an interconnect. For every transfer the Tx_RAW Controller also generates the packet's destination in the form of the physical interface it will be sent from and sends it along with the packet to the Router. For RAW_NEIGHBOUR packets the destination is the descriptor's DESTINATION field, already in the form of a physical interface, while for RAW_DATA packets the descriptor's DESTINATION field is sent to a Routing Table and the returned value is

the physical interface.

Figure 3.11 is the FSM of the Tx_Raw Controller. When the Tx_Raw Controller starts receiving a RAW_DATA descriptor, it reads the Routing Table to get the destination, and after one clock starts generating the packet. The descriptor is read and the packet is generated 8 Bytes per clock.

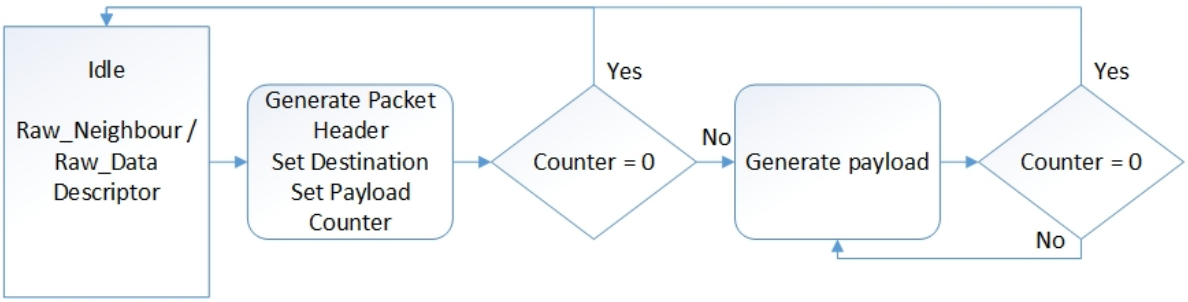



Fig. 3.11 Tx_RAW Controller Packet Generation

## Tx_DMA Controller

The Tx_DMA Controller handles the LONG_DATA, RDMA_WRITE and RDMA_READ transfers. Figure 3.12 is the FSM of the Tx_DMA Controller. It reads descriptors from the Tx_DMA FIFO, or the Rx Controller and generates INIT, LONG_DATA, RDMA_WRITE, RDMA_READ and RDMA_RESPONSE packets. It also reads the Node id from the corresponding Control Register as the Source id. Then it reads the DMA_START and DMA_END Control registers to get the starting address and the maximum ending address, to setup the DMA transfer. RDMA_READ packets are of low priority and they are output to the Vc0 FIFO. All the other packets are of medium priority, so they are output to the Vc1 FIFO through an interconnect. The Tx_DMA Controller is also connected to the DMA engine. To start a transfer it sends a command to the DMA engine, and when data is received, it adds them to packets as payload. Similarly with the Tx_RAW Controller, for every transfer the Tx_DMA Controller also generates the packet's destination in the form of the physical interface it will be sent from and sends it along with the packet to the Router. To generate the destination, the descriptor's DESTINATION field is sent to a Routing Table and the returned value is the physical interface. Finally, the Tx_DMA Controller is connected to the Rx Controller, in order to get informed whenever a NACK packet is received.

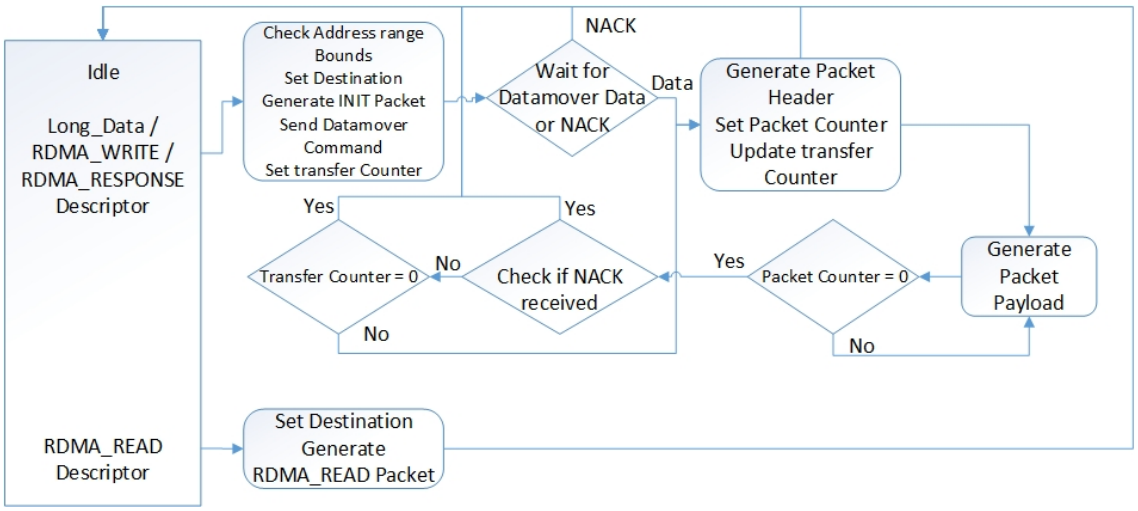



Fig. 3.12 Tx_DMA Controller Packet Generation

When the Tx_DMA Controller starts receiving an RDMA_WRITE descriptor, it checks the bounds of the address offsets given. In case the transfer is out of bounds, which means that the starting address plus offset is higher than the maximum ending address, it is rejected. If everything is ok, an INIT packet and a single command for the DMA engine are generated. Then the controller waits for data from the DMA engine. When the DMA engine starts sending data, the Controller starts creating packets, until all data is processed.

### Rx Controller

The Rx Controller handles all the received packets. It reads descriptors from the All_Vcs_In FIFO and writes descriptors in the corresponding Rx FIFO. It also generates ACK/NACK packets, which are of high priority and are output to the Vc2 FIFO. As with the other Controllers, it generates the ACK/NACK packet's destination with the use of a Routing Table and sends it along with the packet to the Router. The Rx Controller keeps internally information about the ongoing RDMA transfers and is connected to the Long Data Table Control Register, which is needed for LONG_DATA transfers. It also needs to read the DMA_START Control register to setup the DMA transfer. The Rx Controller is connected to the DMA engine. To start a transfer it sends a command first and the data afterward. Finally, the Rx Controller is connected to the Tx_DMA Controller, in order to inform it whenever a NACK packet is received.

When the Rx Controller starts receiving a RAW_DATA packet, it starts generating the descriptor, as shown in Figure 3.13, the FSM of the Rx Controller. The packet is read and the descriptor is generated 8 Bytes per clock. When the Rx Controller receives an INIT packet, it tries to allocate the necessary resources to accept the transfer. If it succeeds, every time an RDMA_WRITE packet is received, a command is sent to the DMA engine, followed by the payload, and the controller awaits for another packet, not necessarily of the same transfer. This is repeated for every RDMA_WRITE packet, which means that many small transfers are sent to the DMA engine, instead of a larger one as with the Tx_DMA Controller. Only when the internal transfer counter reaches zero, the Rx Controller waits for the DMA engine to complete the transfer, before generating the descriptor and the ACK packet.

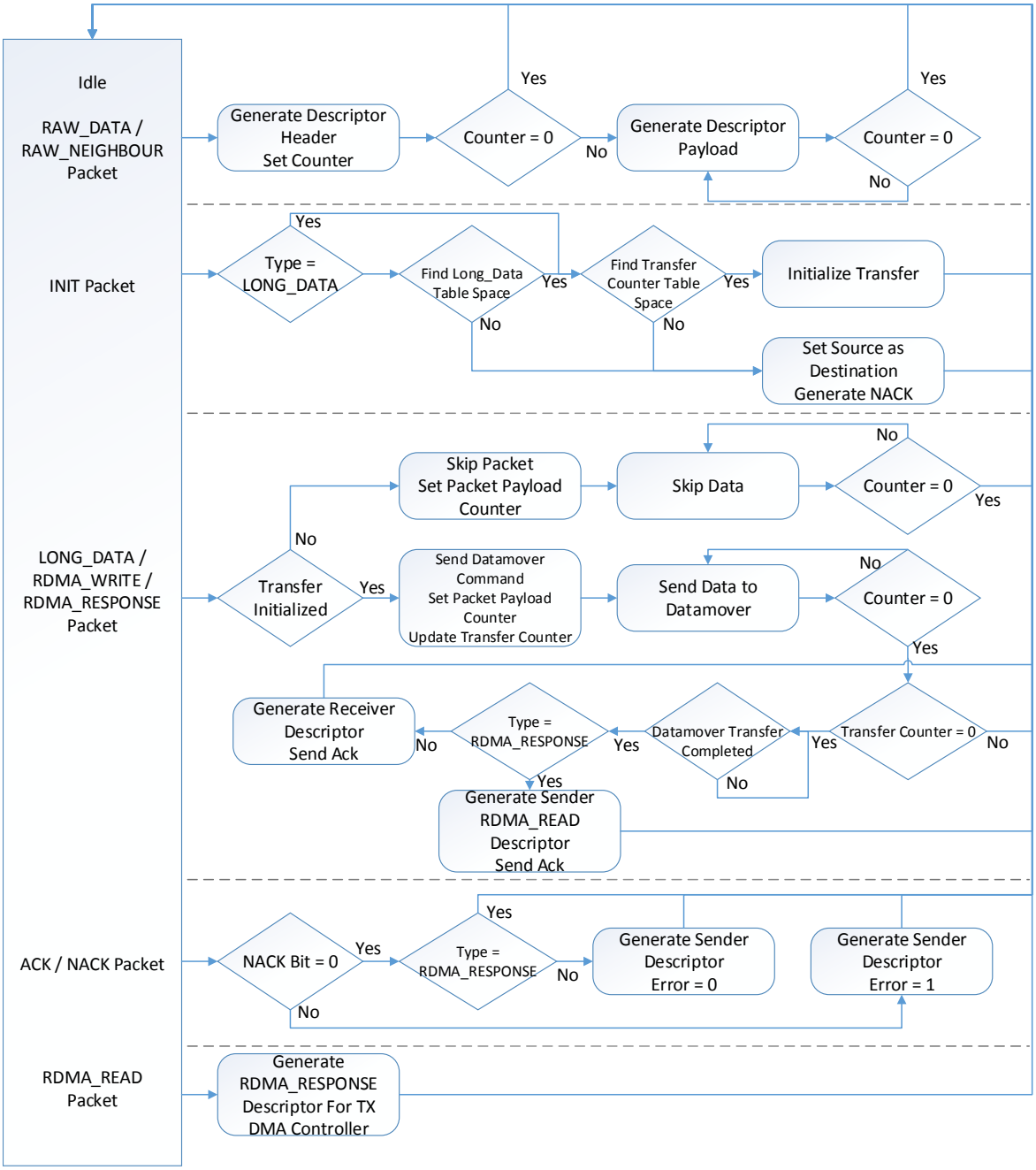



Fig. 3.13 Rx Controller Packet Generation

### Interconnect

The Interconnect is used to receive packets for the Tx_RAW and Tx_DMA Controllers and forward them to the Vc1 FIFO. It uses the Round-robin algorithm for packet priority and if one of the two lanes is empty, the other is given priority in the same clock. To implement the Round-robin algorithm a token variable is used that changes the priority every time the input that has the token is read and has valid data. If that stream is empty, the other stream is read in the same clock. If the stream that does not have the token is read because the other stream is empty, the token remains to the other stream.

The Interconnect is an FSM, as shown in Figure 3.14. The first state reads the input that has the token, and if it is empty the other input is read. If the first input is read with valid data, the FSM goes to the second state, where it reads the rest of the packet from that input and forwards it until the TLAST signal is asserted and then returns to the first state. If the second input is read with valid data, the FSM goes to the third state, where again it forwards the packet until the TLAST signal is asserted and then returns to the first state. If the input that was read had the token, then the token changes input.

The same interconnect is also used to send descriptors to the Tx_DMA controller either from the Tx_DMA FIFO or from the Rx Controller, in the case of RDMA_RESPONSE descriptors.

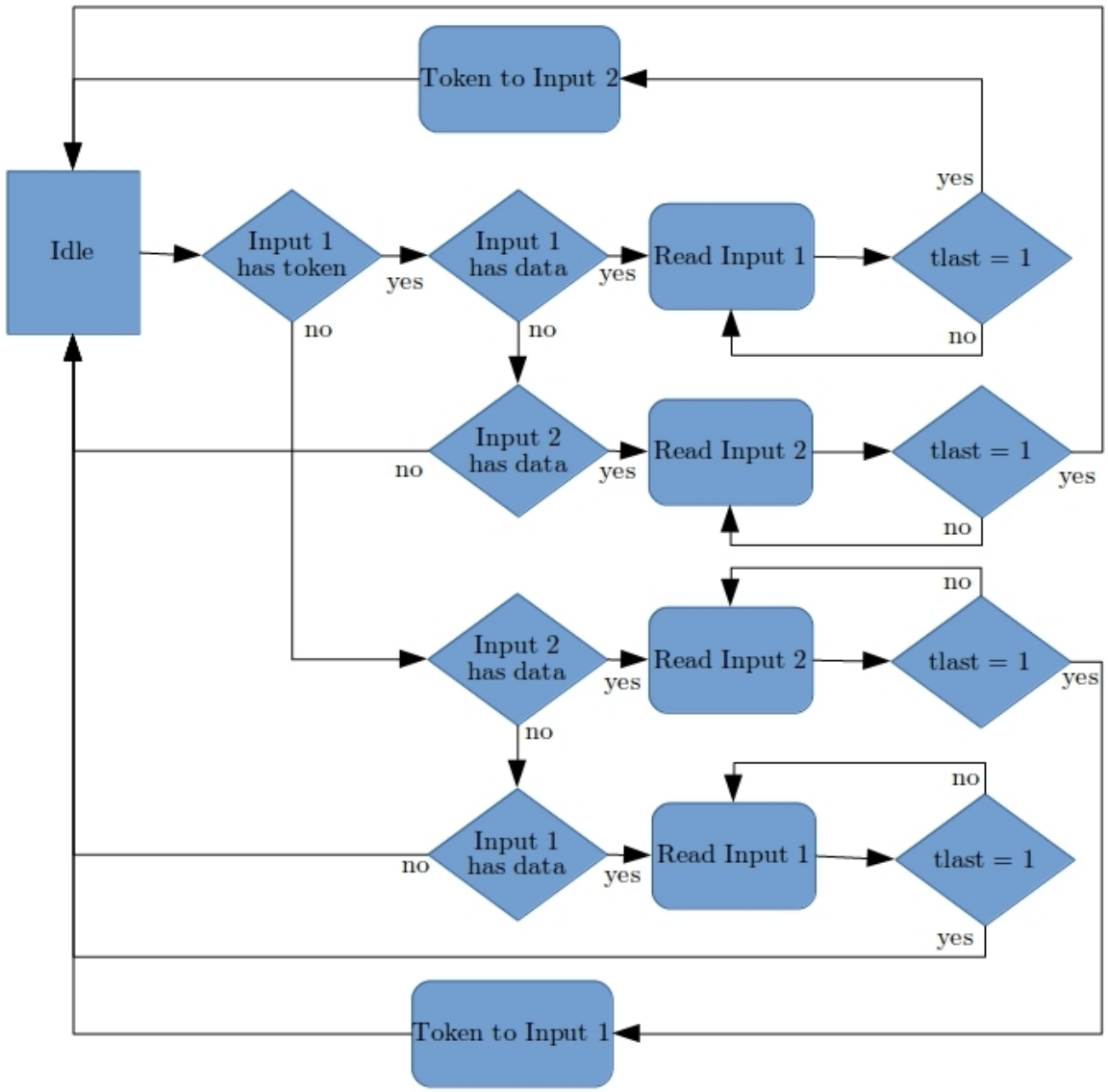
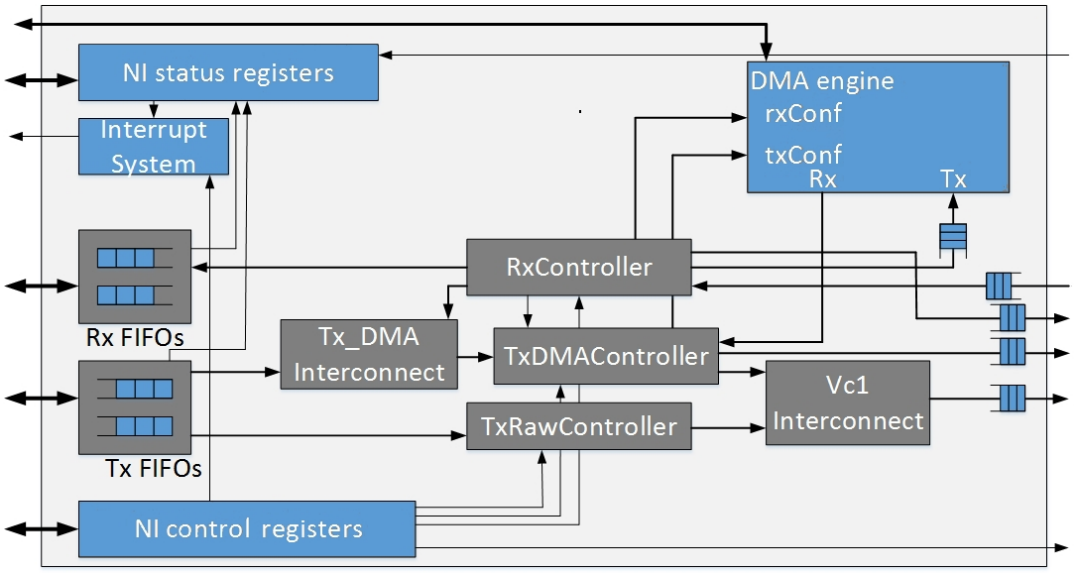



Fig. 3.14 Interconnect FSM

## 3.3 DMA engine, Interrupt System, Control and Status Registers




### DMA engine

The DMA engine is used to transfer data directly to and from the memory without interrupting the PS. It is connected to the PS's slave HP port, and has two channels, one for reading from the memory and one for writing to the memory. The reading channel is connected to the Tx_DMA Controller, and when the Controller reads a descriptor, it sends the corresponding command to start a read from the memory. The DMA engine reads the command and starts a read transfer, and the received data are forwarded to the Tx_DMA Controller. Similarly, the writing channel is connected to the Rx Controller, and whenever the Rx controller receives and RDMA or LONG_DATA packet, it sends the corresponding command to start a write to the memory, and then the data to be written. The DMA engine reads the command and starts a write transfer, and as the Rx Controller sends data to the DMA engine, they are forwarded to the memory.

### Interrupt Handler

The Interrupt Handler receives the status of the four FIFOs connected to the PS in the form of a "Not Full" signal from the Tx FIFOs and a "Not Empty" from the Rx FIFOs, masked by the MSKIRQ register, and a four bit wide reset signal, each reset corresponding to one input. It also sends a four bit output to the PS through a register, each bit indicating that an interrupt has been for the corresponding FIFO. Finally, it generates an interrupt that is connected to the PS through the GIC.

Whenever one of the input signals goes from zero to one, an interrupt is generated and the corresponding output bit is set to one. Every output bit that is set to one has to be reset by the PS, and while set to one, the changes of the corresponding input are not monitored, and interrupts are not generated for that bit. That is because until the PS acknowledges the already generated interrupts and clears them, no more interrupts for the same FIFO should be generated. The process that the interrupts work is:

1. A FIFO changes state.
2. The corresponding output bit of the Interrupt Handler is set to one and an interrupt is generated.
3. The PS receives the interrupt.
4. The PS reads the GPIO that is connected to the Interrupt Handlers output, to find out which FIFO is empty or full.
5. The PS resets the output bit by setting the reset bit to one, acknowledging the interrupt. Then it proceeds to interact with the FIFO.

### Control and Status Registers

The NI has several control and status registers to allow for communication between it and the PS. The expected functionality is described in table 3.3

Table 3.3 Registers

| Name | Width | Access | Description |
|---|---|---|---|
| VERSION | 16 bits | ro | Version register |
| IFNUMBER | 3 bits | ro | Number of interfaces used by the current version |
| IFINFO_BASE_1 | 3 bits | ro | Interface 1 (Rx_1) status table, shows direction (Rx/Tx) and status (connected or not) |
| IFINFO_BASE_2 | 3 bits | ro | Interface 2 (Tx_1) status table, shows direction (Rx/Tx) and status (connected or not) |
| RAW_TX_STATUS | 32 bits | ro | The Vacancy of the TX_RAW FIFO. When greater than 0, in the Tx_RAW FIFO there is space available for new descriptors from the PS. |
| RAW_RX_STATUS | 32 bits | ro | The Occupancy of the Rx_RAW FIFO. When greater than 0, in the Rx_RAW FIFO there are descriptors available to be read from the PS. |
| RDMA_TX_STATUS | 32 bits | ro | The Vacancy of the TX_DMA FIFO. When greater than 0, in the TX_DMA FIFO there is space available for new descriptors from the PS. |
| RDMA_RX_STATUS | 32 bits | ro | The Occupancy of the RX_DMA FIFO. When greater than 0, in the RX_DMA FIFO there are descriptors available to be read from the PS. |
| NODEID | 8 bits | rw | Node ID register, set by the PS after the discovery protocol |
| DMA_START | 64 bits | wo | DMA start Address register |
| DMA_END | 64 bits | wo | DMA end Address register |
| MSKIRQ | 4 bits | rw | Mask Interrupt Register. Used to enable/disable interrupts |
| PNDIRQ | 4 bits | rw | Pending Interrupt Register. Used to report which FIFO was the cause of the interrupt. |

Several BRAMs are also needed, mostly routing tables to read the destination interface of the packets, as show below (table 3.4). All routing tables are mirrored, meaning that they have the same values. We need more than one routing table because many modules need access to it.

Table 3.4 BRAMs

| Name | Width | Depth |
|---|---|---|
| PHY1_ROUTING_BASE | 8 bits | 256 lines |
| **Description** | Routing table BRAM. The array index represents the destination node id. The value represents the transfer interface, or 0xff for unused ids. Setup by the PS after the discovery protocol. Connected to the Router's Link Controller for Tx Interface 1. | |
| RX_ROUTING_BASE | 8 bits | 256 lines |
| **Description** | Routing table BRAM. Connected to the Rx Controller. | |
| TX_DMA_ROUTING_BASE | 8 bits | 256 lines |
| **Description** | Routing table BRAM. Connected to the Tx DMA Controller. | |
| TX_RAW_ROUTING_BASE | 8 bits | 256 lines |
| **Description** | Routing table BRAM. Connected to the Tx Raw Controller. | |
| LONG_BUF_BASE | 64 bits | 32 lines |
| **Description** | This BRAM represents the addresses and sizes of buffers available for receiving the LONG messages. It is setup by the PS and written to by the NI to indicate a used buffer. | |

# Chapter 4

# System Implementation

## 4.1 Tools used

The NI was implemented using the Xilinx Vivado Design Suite - HL System Edition 2016.3. The tools used are the Vivado HLS, Vivado IDE and Xilinx SDK.

### Vivado HLS

The Xilinx Vivado HLS tool provides a higher level of abstraction for the user by synthesizing functions written in C into IP blocks, by generating the appropriate Vhdl and Verilog code. Those blocks can then be integrated into a hardware system. Vivado HLS is tightly integrated with the rest of the Xilinx design tools and provides comprehensive language support and features for lower level optimizations, making it possible to optimize the C code for hardware systems.

The Vivado HLS tool allows for C functions written in C, C++, SystemC, or an OpenCL API C kernel. We decided to use C++, as some of the libraries we needed were for C++. To debug the code Vivado HLS uses a C test bench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL Cosimulation. The tool also adds some constraints to the exported IP block, like the clock period, clock uncertainty, and FPGA target. The clock uncertainty defaults to 12.5% of the clock period if not specified. The clock period we used was 6 ns to allow for frequencies up to 166,6 MHz. Finally, the tool allows for directives to be added to the code to direct the synthesis process to implement a specific behavior or optimization. Directives are optional and do not change

the behavior of the c code in the simulations, only the synthesized IP block. The Vivado HLS design flow follows the normal C design flow, but extends it to hardware design:

1. First we had to compile the code and create a C test bench with set inputs, then we executed our function using the test bench to simulate the IP block's functionality and debug the C algorithm.

2. After making sure that our function works as intended, we synthesized the C algorithm into an RTL implementation and verified the output with the use of the C/RTL Cosimulation. At this point we could start adding directives to optimize the design, but we thought it was best to try it first and confirm its functionality. To do that we exported the IP to be used in Vivado IDE.

3. After making sure the basic function was working as intended, which we will explain in the Vivado section, we returned to HLS and started adding directives to optimize the running time and resources. The use of some directives notably the PIPELINE directive changed the form and functionality of the IP block quite a bit, and we had to retest it both in Vivado HLS and Vivado IDE.

This process was followed for all the Vivado HLS IP blocks we designed individually:

- Rx Controller
- Tx Raw Controller
- Tx Dma Controller
- Interconnect
- Interrupt Handler
- LongDataBramController
- PosFinder

After each module functioned as expected, we starting building a project with all of them by adding them and the other needed IPs a few at a time and debugging the design using Vivado IDE and Vivado SDK.
The directives we used are the following:
INTERFACE
Specifies how RTL ports are created from the function description. Several choices are

available, and the ones we used are:
ap_ctrl_none : No block-level I/O protocol.
ap_none : No protocol. The interface is a data port.
ap_vld : Implements the data port with an associated valid port to indicate when the data is valid for reading or writing.
bram: Implements array arguments as a standard RAM interface.
axis: Implements all ports as an AXI4-Stream interface. Only the optional signals included in the C structure of the Stream are implemented.
e.g. #pragma HLS INTERFACE ap_vld port=longDataTable_out

PIPELINE
Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
e.g. #pragma HLS PIPELINE II=1

RESOURCE
Specifies that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL. The core we used is RAM_1P_BRAM to specify that this IP has only one port of the BRAM available to use. The second port of the BRAM is connected to a BRAM Controller XIlinx ip that allows the PS to edit the BRAM.
e.g. #pragma HLS RESOURCE variable=LongDataTable core=RAM_1P_BRAM

ARRAY_PARTITION
Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
e.g. #pragma HLS ARRAY_PARTITION variable=RDMApayload complete dim=1

## Vivado IDE

The Vivado IDE is the GUI for the Vivado Design Suite. All of the Vivado design Suite tools are written with a native Tcl interface, and all of those commands are available through the IDE either through the GUI or through the Tcl console. Tcl commands can be entered in the Tcl Console in the Vivado IDE or using the Vivado Design Suite Tcl shell. You can run analysis and assign constraints throughout the design process. Timing

and power estimations are provided after synthesis, placement, and routing. Design checkpoints are created after every part of the process and it is possible to interact with the design at each design stage. The Vivado IDE is an evolution of the PlanAhead tool that has shipped with the ISE Design Suite. While the Vivado Design Suite allows for many options and changes to a design, we followed a fairly straightforward design flow:

1. First, we added to the IP repository the folders of our HLS and VHDL IPs. This allows for the addition of instances of these modules to our block design. The repositories were updated automatically whenever we exported new versions of our IPs from HLS.

2. Secondly, we created a block design and added the Zynq Ultrascale+ IP that is the PS. Then we edited the settings to our needs, by enabling the ports and modules we needed, disabling those we did not need, and setting the correct parameters for the DDR memory and the clock frequency. Also, when we started using the AXIOM board, we imported a constraints file provided by SECO for the board. The settings were edited several times, along with changes to the PL.

3. Many smaller block designs were created and synthesized, alongside their respective testbenches to debug each IP individually. If an IP did not have the desired functionality, we would find the problem and return to HLS to fix it.

4. Then, we started adding our working IPs to the main block design and connecting them with the PS and the IPs provided by Xilinx that we would use, marked the signals we wanted to monitor as "debug signals", and synthesized the main design. Marking a signal as "debug signal" connects it to an ILA.

5. While the design was still fairly small and manageable, we used a testbench to debug it first, but as the project grew, we would set up the debug signals and go through implementation, place and route, and bitstream generation directly, since it was more efficient to check it by downloading the bitstream to the FPGA.

6. To debug the downloaded Bitstream we used the Xilinx SDK.

The above process was repeated many times, ranging from major functionality changes to minor speed optimizations, until we were within our goals.

### Xilinx SDK

The Xilinx SDK is an IDE for development of embedded software applications targeted towards Xilinx embedded processors. The SDK works with hardware designs created with Vivado Design Suite. The SDK is based on the Eclipse open source standard. SDK features include a feature-rich C/C++ code editor and a compilation environment with easy project management, application build configuration and automatic Makefile generation and a well-integrated environment for seamless debugging and profiling of embedded targets as well as system level performance analysis. The SDK also provides focussed special tools to configure FPGAs and create bootable Images.

To use the Xilinx SDK, from the Vivado IDE, after exporting the project, including the bitstream, we open the SDK using the preconfigured directories. That way, the SDK automatically imports the hardware wrapper and generates the files needed for the project. Then we create a new Hello World project, which generates not only the project files, but also the needed BSP, which includes the needed drivers for the included modules that the PS has access to.

The SDK is then used to create a basic program to be run by the PS to test and debug the PL functionality. To be able to program the FPGA, the JTAG port has to be connected to the pc, and to monitor and debug it we use the UART port as well. Another very useful tool that is part of the Vivado ISE is the Hardware Manager, that connects to the ILAs that have been added to the Vivado project and allows us to monitor in real-time the values of the signals between our modules, while the program is running.

The debug program we wrote first initializes all the individual modules using the libraries provided by Xilinx, then initializes the NI by writing to the registers and BRAMs and waits for connection to another board, or to itself for loopback mode. After the connection is up, the user can send raw and RDMA packets to check the data integrity and the throughput of the design. In the case of an error the program informs the user, and in combination with the Hardware Manager it is easy to identify the malfunctioning module.

Another debug tool that should be noted is the XSCT console, which gives us direct access to all the memory-mapped modules of the FPGA, and allows us to read any register or DDR address we use. To use the XSCT console we have to be running the

program in debug mode and pause it before requesting a memory read or any other command.

## 4.2 Schematic of implemented NI - interface with router

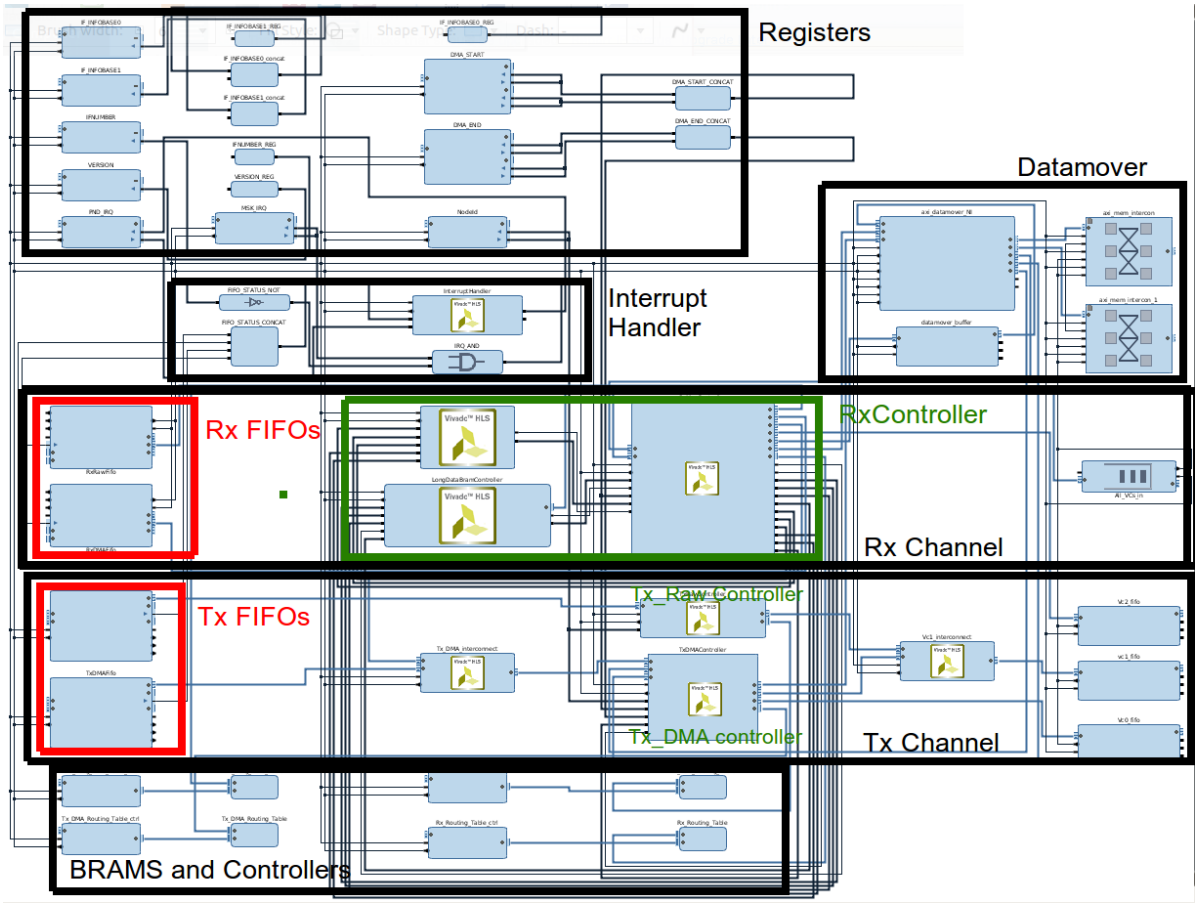



Fig. 4.1 NI implementation

The implemented design, as shown in figure 4.1, is comprised of six main groups. The Registers group includes all the registers needed to initialize the NI and interact with the PS. The Interrupt Handler monitors the Tx and Rx FIFOs and generates interrupts when needed. The Datamover, which is a DMA engine module, is responsible for the DMA transfers. The Rx channel consists of the Rx_RAW and Rx_DMA FIFOs that connect the NI to the PS, the Rx Controller, and the All_Vcs_In FIFO to receive packets from the Router. The Tx channel consists of the Tx_RAW and Tx_DMA FIFOs that

connect the PS to the NI, the Tx_RAW and Tx_DMA Controllers and the Vc2, Vc1 and Vc0 FIFOs to send packets to the Router. Because the Vc1 channel is used by both controllers, an interconnect is needed between the Controllers and the Vc1 FIFO. Another interconnect is used between the Tx_DMA FIFO and Tx_DMA Controller, as the Rx Controller also sends Descriptors to the Tx_DMA Controller. Finally, several BRAMs are used as routing tables and connected to each Controller, as well as a BRAM needed for the LONG_DATA transfers.

To transfer data to and from the PS, we used one of the available High Performance Master (HPM) ports. The interface of that port uses the AMBA AXI4 protocol which is targeted at high performance, high clock frequency system designs and includes features that make it suitable for high speed interconnect:

- Separate address/control and data phases.
- Support for unaligned data transfers using byte strobes.
- Burst based transactions with only start address issued.
- Issuing of multiple outstanding addresses with out of order responses.
- Easy addition of register stages to provide timing closure.

We also utilized the slave High Performance Coherent port in connection to the DMA engine for transfer of data to and from the memory, that uses the AXI4 interface too.

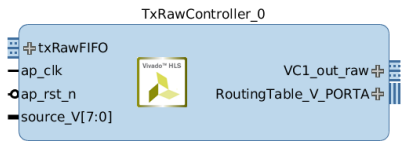
The NI communicates with the PS through the AXI4-Stream FIFOs, which are connected to the HPM port of the PS through an interconnect and provide AXI4-Stream ports for our IPs. The AXI4-Stream protocol (table 4.1) is used to transfer data efficiently from point to point and without the use of addresses. Each AXI4-Stream acts as a single unidirectional channel for a handshake data flow.

Table 4.1 AXI4-Stream signals we Used

| Name | Direction | Size | Optional | Description |
|---|---|---|---|---|
| TVALID | Out | 1 Bit | No | **Indicates when TDATA contains valid data.** |
| TREADY | In | 1 Bit | No | **Indicates if receiver is ready to read data.** |
| TDATA | Out | N Bits | No | **Data to be transfered.** |
| TLAST | Out | 1 Bit | Yes | **Indicates the end of the packet.** |
| TKEEP | Out | N/8 Bits | Yes | **Indicates valid Bits. Used only when TLAST is asserted.** |
| TDEST | Out | M Bits | Yes | **Normally used with Xilinx IPs. In our implementation, this signal contains the interface id from which the packet will be sent and is read by the Router.** |

Not all optional signals were used for all our streams, only those needed per stream. The controllers were designed as Finite State Machines using certain directives to produce the intended functionality. The most important directive used is the PIPELINE directive, which reduces the initiation interval by allowing the concurrent execution of operations within our function.

## Tx_RAW Controller




The Tx Raw Controller is an FSM of two states, as described in Chapter 3.2. It uses the PIPELINE directive and has a latency of 1cc. It also uses the "INTERFACE ap_ctrl_none" directive, as the control signals HLS generates, notably "ap_start", "ap_ready" and other, are not needed. The "INTERFACE axis" directive is used for the txRawFifo input stream, with the optional signal TLAST enabled, and the VC1_out_raw output stream, with the optional TLAST and TDEST enabled, and the "INTERFACE ap_none" directive is used for the source_V data input. Finally, the

RoutingTable interface that communicates with the Routing Table Bram is implemented using the "INTERFACE bram" directive and the "RESOURCE core=RAM_1P_BRAM" and the Bram is read and written to as if it was an array.

e.g. dest = RoutingTable[pos];

Whenever there is a descriptor in the Tx_Raw FIFO, the controller starts receiving it, and with a latency of 1cc generating the packet, as shown in 4.2. The transfer shown below has a payload of 124 Bytes. As the Controller is pipelined, there is no latency between packets. The latency is the same for RAW_NEIGHBOUR transfers.

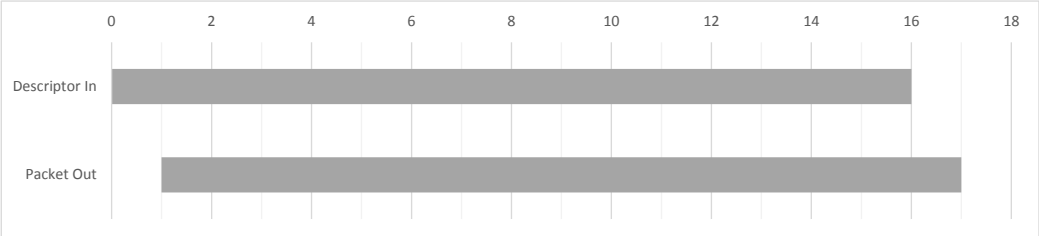
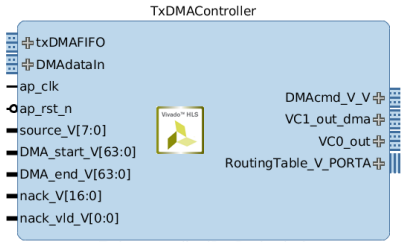



Fig. 4.2 Tx_RAW Controller latency

## Tx_DMA Controller




The Tx DMA Controller is an FSM of 14 states, as described in Chapter 3.2. It uses the PIPELINE directive and has a latency of 1cc. It also uses the "INTERFACE ap_ctrl_none" directive, as does the Tx Raw Controller. Another directive, the "INTERFACE axis" is used for all the stream interfaces.

- txDMAFIFO, from which the RDMA and LONG_DATA descriptors are received. It has the optional signal TLAST enabled.
- DMAstsin, from which the Datamover transfer status is received. It has the optional signal TLAST enabled.
- DMAcmd, from which the Datamover command is sent to start dma transfers. It has no optional signals.
- DMAdataIn, from which the data from the Datamover is received and added to packets. It has the optional signal TLAST enabled.
- VC1_out_dma, from which all the RDMA and LONG_DATA packets except RDMA_READ are sent. It has the optional signals TLAST and TDEST enabled.
- VC0_out, from which the RDMA_READ packets are sent. It has the optional signals TLAST and TDEST enabled.

For access to the Routing Table Bram, the RoutingTable interface is implemented using the "INTERFACE bram" directive and the "RESOURCE core=RAM_1P_BRAM" similarly to the Tx Raw Controller. Finally, the source, DMA_START, DMA_END, nack and nack_vld data signals are implemented using the "INTERFACE ap_none" directive.

The latency of an RDMA_WRITE transfer is shown in Figure 4.3. First, the Controller reads a Descriptor and generates an INIT packet for the Rx_Controller on the receiving side, and a command for the local Datamover. Then the controller waits for data from the Datamover, which has a latency of about 50 cc. When the Datamover starts sending

data, the Controller starts creating packets of 252 Bytes max payload, until all data is processed.
The transfer shown below has a payload of 320 Bytes, which means two packets are generated. The latency is very similar for LONG_DATA transfers and RDMA_RESPONSE transfers.

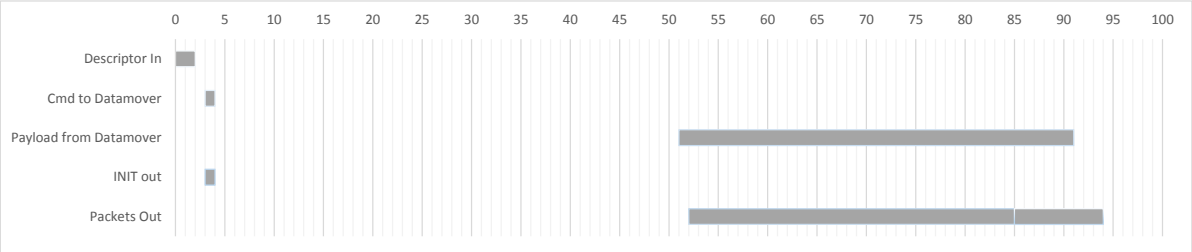
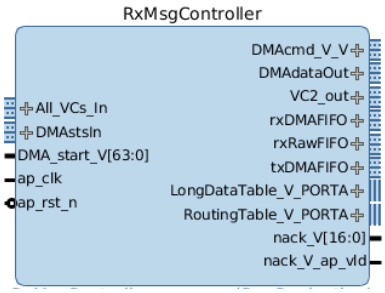



Fig. 4.3 Tx_DMA Controller latency

## Rx Controller




Using the PIPELINE directive, HLS schedules all input reads at the start of every clock, and determines later the state to be executed. That means that some BRAMs that were read in multiple states, had to be read more than once in the same clock, but we had only one available port, which created a dependence and added latency to the controller. To avoid that, another IP, the Long Data Bram Controller, was created, to manage the BRAM read and write, and send the correct data back to the Rx Msg Controller, adding 1 clock cycle of latency to the transfers. Also, because of the PIPELINE directive, all loops are unrolled, and for every iteration a new state tree is created. That means that for a 32 iterations loop at the start of the FSM, the loop will be unrolled and the FSM will be created for every unrolled instance of the loop. That was very inefficient, so another IP, the Pos Finder, was created to perform that loop and send the outcome back. The Pos Finder IP stores the data for LONG_DATA and RDMA transfers, if there is space, every time an INIT packet is received, and searches for that data every time an RDMA or LONG_DATA packet is received. This IP is responsible for rejecting transfers and skipping the packets of rejected transfers. These

two IPs plus the Rx Msg Controller are connected as shown in figure 4.4 and a wrapper is generated, that is the Rx Controller IP.

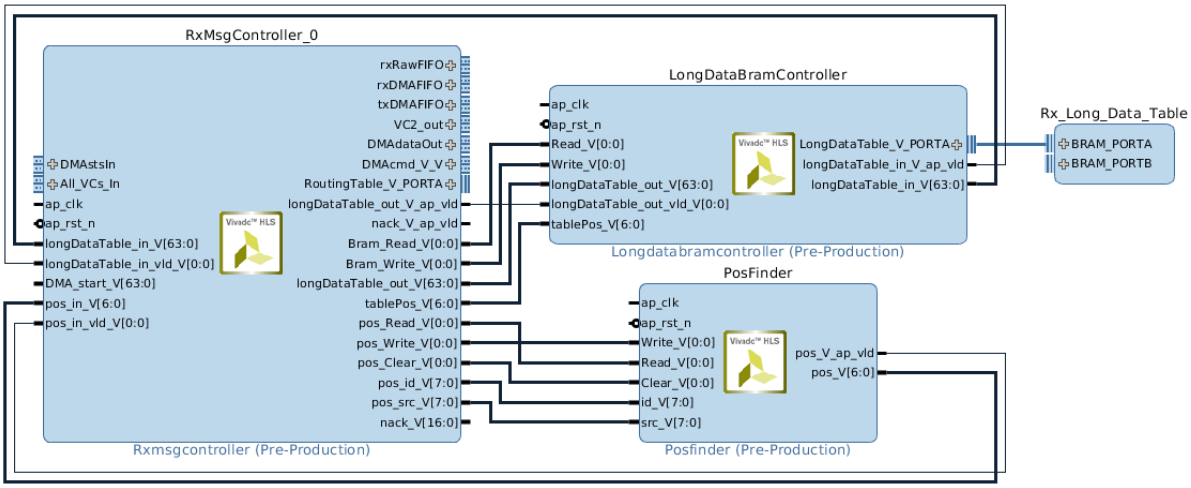



Fig. 4.4 Rx Msg Controller with Long Data Bram Controller and Pos Finder

The Rx Msg Controller is an FSM of 22 states, as described in Chapter 3.2. Other than the PIPELINE directive, it uses several other directives. One is the "INTERFACE ap_ctrl_none" directive, as with Tx Raw and Tx DMA Controllers. Also, the "INTERFACE axis" is used for all the stream interfaces.

- All_VCs_in, from which the packets are received. It has the optional signals TLAST and TDEST enabled.
- DMAstsin, from which the Datamover transfer status is received. It has the optional signal TLAST enabled.
- DMAcmd, from which the Datamover command is sent to start dma transfers. It has no optional signals.
- DMAdataOut, from which the data received from the packets is sent to the Datamover. It has the optional signals TLAST and TKEEP enabled.
- VC2_out, from which the ACK/NACK packets are sent. It has the optional signals TLAST and TDEST enabled.
- rxDMAFIFO, from which the RDMA and LONG_DATA generated descriptors are sent to the DMA FIFO. It has the optional signal TLAST enabled.

- rxRawFIFO, from which the RAW_DATA and RAW_NEIGHBOUR generated descriptors are sent to the RAW FIFO. It has the optional signal TLAST enabled.
- txDMAFIFO, from which the RDMA_RESPONSE generated descriptor is sent to the Tx DMA Controller. It has the optional signal TLAST enabled.

The RoutingTable interface is implemented using the "INTERFACE bram" directive and the "RESOURCE core=RAM_1P_BRAM" similarly to the other Controllers. Finally, all the data signals used to connect the Rx Msg Controller to the Long Data Bram Controller and the Pos Finder use the "INTERFACE ap_none" directive.

### Long Data Bram Controller

The Long Data Bram Controller manages the Long Data BRAM reads and writes without having dependences between signals. To perform a Write, it receives a write request and the position from the Rx Msg Controller, and it waits for valid data. When it receives the data it performs a write transfer to the Bram. To perform a Read, it receives a read request and the position from the Rx Msg Controller, and then returns the data back to the Rx Msg Controller.
The Long Data Bram Controller uses the "PIPELINE" directive with only two states, one to receive the request from the Rx Msg Controller, send the position to the bram and in the case of a write send the data too, and a second to send the data to the Rx Msg Controller in the case of a read. As with the other controllers, the Long Data Bram Controller also uses the "INTERFACE ap_ctrl_none" directive. The interface that communicates with the Long Data Bram is implemented using the "INTERFACE bram" directive and the "RESOURCE core=RAM_1P_BRAM" similar to the RoutingTable interface of the Rx Msg Controller.

### Pos Finder

The Pos Finder works similar to a fully associative CACHE. It has three internal tables of 32 positions, id, src and used. Three types of transfers are supported. First is the WRITE transfer, where it searches for an empty position and if it finds one, it stores the id and src, sets used to 1, and returns the position, else it returns an out of bounds position to indicate that it is full. Second is the READ transfer, where it searches for a position with the id and src given, and if found it returns the position, else it returns an out of bounds position to indicate that no transfer with that id and src combination is

active. Finally, there is the CLEAR transfer, where it finds the position with the id and src given, and sets used to 0 to indicate that the position is empty.
The Pos Finder uses the "PIPELINE" directive too, and the searches are performed by loops that are unrolled automatically. The "INTERFACE ap_ctrl_none" directive is used too. To allow for a latency of 1cc, the use of the "ARRAY_PARTITION complete" directive was necessary for all three tables.

Whenever RAW_DATA or RAW_NEIGHBOUR packet is received, the controller starts receiving it, and with a latency of 1cc generating the packet, as shown in Figure 4.5. The transfer shown below is a RAW_DATA transfer has a payload of 124 Bytes. As the Controller is pipelined, there is no latency between packets. The latency is the same for RAW_NEIGHBOUR transfers.

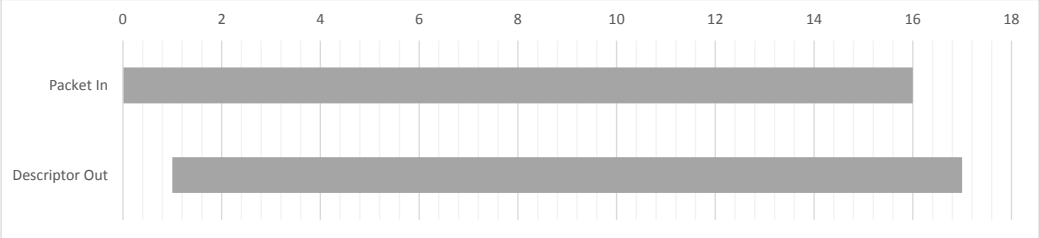



Fig. 4.5 Rx Controller RAW_Data latency

The latency of an RDMA_WRITE transfer is shown in Figure 4.6. First, the Controller receives the INIT packet and allocates the necessary resources. Then the controller waits for packets with a message id matching the INIT packet's. The first packet arrives about 50cc later, as that is the latency of the Tx_DMA Controller. When the Rx Controller starts receiving packets, it sends a command per packet to the Datamover, followed by the payload of the packet. The status of every completed transfer is received about 50cc after the data is written, but the next transfer has already started. only for the last packet the Controller waits for completion before generating the Descriptor, to make sure all the data is written before the PS accesses them. The latency is very similar for LONG_DATA and RDMA_RESPONSE transfers.

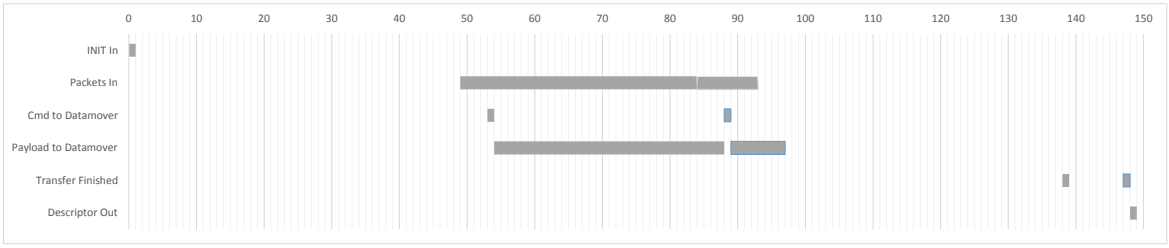
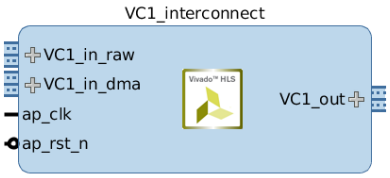



Fig. 4.6 Rx Controller RDMA_WRITE latency

## Interconnect




The Interconnect has a latency of 1cc.
The directives used are the "PIPELINE" directive for the FSM, "INTERFACE axis" directive for the two inputs and the output with the optional signals TLAST and TDEST enabled, and the "INTERFACE ap_ctrl_none" directive.

## Interrupt Handler

The Interrupt Handler is the only IP that does not use the "PIPELINE" directive. It receives a four bits wide signal named pndirq_in, which is the pending interrupts for the two Rx and the two Tx FIFOs connected to the PS, and a four bits wide irqclear signal that is sent from the PS with the use of an AXI GPIO. It's outputs are the four bits wide signal named pndirq_out, which is stored in registers, and the 1 bit wide interrupt signal that is connected to the PS through the internal Interrupt Controller. Whenever one of the pndirq_in bits goes from low to high and the same pndirq_out bit is low, the interrupt signal is asserted and the same pndirq_out bit is also asserted, so that the PS can read or write the corresponding FIFO. After the PS has received and processed the interrupt, it then asserts the relevant irqclear bit to deassert the pndirq_out bit. Until the PS clears the pndirq_out bits that are high, interrupts are not generated for these bits. The Interrupt Handler uses the "INTERFACE ap_ctrl_none" directive and both the inputs and outputs use the "INTERFACE ap_none" directive.

## Datamover

The AXI DataMover IP provides the basic AXI4 Read to AXI4-Stream and AXI4-Stream to AXI4 Write data transport and protocol conversion. It is used to access the DDR memory without interrupting the PS. The IP has two channels, the MM2S channel which reads data from the DDR and sends them to the Tx_DMA Controller, and the S2MM channel which receives data from the Rx Controller and writes them to the DDR, each with its own dedicated command stream, status stream, AXI4 Data port and AXI4-Stream Data port. The MM2s channel handles transactions from the AXI4 to the AXI4-Stream domain and the S2MM channel handles transactions from the AXI4-Stream to AXI4 domain.

To start a transfer, a command is sent to the corresponding command port with the command format explained in table 4.3. After that, for MM2S transfers the controller waits for about 50cc until it starts receiving data and at the end the status word, and for S2MM transfers the Rx controller starts sending data and after about 50cc after the last data word it receives the status word. For both types of transfers the status word should have bit 7 set to one and bits 6 - 4 set to zero, as described in table 4.2.

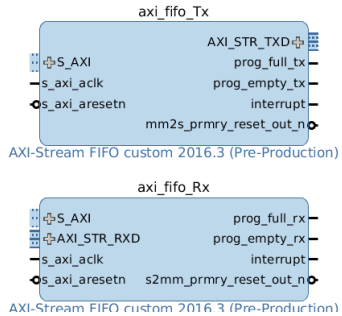
Table 4.2 Datamover Status Format

| Bits | Description |
|---|---|
| 7 | **Transfer OKAY** |
| 6 | **Slave Error** |
| 5 | **Decode Error** |
| 4 | **Internal Error** |
| 3 - 0 | **TAG** |

Table 4.3 Datamover Command Format

| Bits | Description |
|---|---|
| 103 – 100 | **Reserved** |
| 99 – 96 | **TAG** An arbitrary user assigned value that flows through the DataMover execution pipe and gets inserted into the corresponding status word. Used by the Rx Controller to assert if the transfer is complete. |
| 95 - 32 | **Start Address** This field indicates the starting address to use for the memory-mapped side of the transfer requested by the command. |
| 31 | **DRE Re-alignment Request** This bit is only used if the optional DRE is included by parameterization. The bit indicates that the DRE alignment needs to be re-established prior to the execution of the associated command. The NI does not use the DRE. |
| 30 | **End of Frame** This bit indicates that the command is an End of Frame command. This generally affects the MM2S element (Read Master) because it causes the stream output logic to assert the TLAST output on the last data beat of the last transfer needed to complete the command. This bit should be set for S2MM commands when TLAST is expected to arrive with the number of bytes programmed in the command. All the NI transfers are set as EOF. |
| 29 – 24 | **DRE Stream Alignment** This field is only used by the MM2S and if the optional MM2S DRE is included by parameterization. In this design, the DRE is not included. |
| 23 | **Type** This field determines the type of AXI4 access. Setting this to 1 enables INCR. A value of 0 enables FIXED address AXI4 transaction. All the NI transfers are set as INCR. |
| 22 to 0 | **Bytes to Transfer** This 23-bit field indicates the total number of bytes to transfer for the command. In this design, only 16 of the 23 bits are used and the transfers can be from 1 up to 64 KBytes. A value of 0 is not allowed because it causes an internal error from the DataMover, so it is checked by the controllers before starting a transfer. In the case of 0 Bytes to Transfer, the descriptor is discarded. |

## FIFOs




Two kinds of FIFO were used in this design, both intellectual property of Xilinx. The first is the AXI4-Stream FIFO v4.1, that allows memory mapped access to AXI4-Stream interfaces and is used to connect the PS to the controllers. It can be configured as Tx, Rx or both. Both the AXI4 interface and the AXI4-Stream interface are configured to have 64 bits width and the FIFOs were set to have the minimum depth of 512 locations. It should be noted that the FIFO was edited slightly to allow access to the Programmable Full and Programmable Empty signals for the Interrupt System, that were normally accessible only from the AXI4 interface. Four of these FIFOs were used, the Tx_RAW, Tx_DMA, Rx_RAW and Rx_DMA FIFOs. The second FIFO used is the AXI4-Stream Data Fifo, that is an AXI4-Stream to AXI4-Stream FIFO and was used as a buffer between modules. The Vc0, Vc1 and Vc2 FIFOs, as well the All_Vcs_In FIFO are of this type.

## Registers and BRAMs

The registers were implemented using the AXI Gpio IP. The AXI GPIO design provides a general purpose input/output interface to an AXI4-Lite interface. The AXI GPIO can be configured as either a single or a dual-channel device. The width of each channel is independently configurable with a maximum value of 32 bits. 32 bit or smaller registers use only one channel, setup as input for WO, output for RO, or both for RW. For RW registers the output of the Gpio has to be connected to its input. 64 bit registers are implemented using the second channel of the Gpio and a Concat Xilinx IP that concatenates the two 32bit wide values to a 64bit wide output.

BRAMs are implemented using the Xilinx Block Memory Generator IP and setup as true dual-port. The first port of each BRAM is connected to the respective Controller, and the second to an AXI BRAM Controller IP that connects the BRAM to the PS.

## 4.3 Resources on the AXIOM board

The Resources our project occupied were measured after implementation, placement and routing. To get the utilization data, we opened the implemented design and executed the TCL command "report_utilization -hierarchical -file <location_of_the_report_file>", which generates a hierarchical utilization report with data for all the included IPs. Table 4.4 shows the utilization of the HLS modules we designed for the NI, and table 4.5 shows the utilization of the NI, including the Datamover, the FIFOs, the registers and everything else, the utilization of the Router and the utilization of the PS. It should be noted that the PS utilization is very high because the interconnect connected to the HPM port has 28 slaves, not all of which are mandatory.

Table 4.4 HLS modules Utilization

| Module | LUTs | FFs | BLOCKRAMs |
|---|---|---|---|
| Tx_Raw_Controller | 119 | 170 | 0 |
| Tx_DMA_Controller | 1131 | 1118 | 0 |
| Rx_Controller | 4540 | 3930 | 0 |
| Interrupt Handler | 8 | 8 | 0 |
| Tx_DMA_Interconnect | 157 | 269 | 0 |
| Vc1_Interconnect | 165 | 283 | 0 |
| Pos Finder | 636 | 597 | 0 |
| LongDataBramController | 3 | 5 | 0 |
| Total | 6759 | 6380 | 0 |
| Available | 274080 | 548160 | 912 |
| Utilization | 2.47% | 1.16% | 0% |

Table 4.5 Total Utilization

| Module | LUTs | | FFs | | BLOCKRAMs | |
|---|---|---|---|---|---|---|
| NI | 12438 | 4.54% | 13841 | 2.52% | 24 | 2.63% |
| Router | 2038 | 0.74% | 3632 | 0.66% | 7 | 0.77% |
| PS | 30451 | 11.11% | 36195 | 6.61% | 0 | 0% |
| Total | 44927 | 16.39% | 53668 | 9.79% | 31 | 3.4% |
| Available | 274080 | | 548160 | | 912 | |

# Chapter 5

# Evaluation

## 5.1 Board description

Three different boards were used throughout the duration of this thesis. The Digilent Zedboard was the first FPGA to be used. The Zedboard is an evaluation and development board based on the Xilinx Zynq-7000 AP SoC, combining a dual Cortex A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells. The Zedboard did not have dedicated transceivers to use in our design, and we did not succeed in transferring data from board to board with it, but it was used in the first steps of the design to develop and debug the first versions of the controllers. Due to limitations of the board's 32bit Processor and our very unoptimized code, the first controllers used 32bit wide words and were very inefficient, with limited functionality and were never included in a complete project, but they were the first steps, getting us familiarized with the tools and the design process.

Later, the Xilinx ZC706 was used. The ZC706 evaluation board for the XC7Z045 AP SoC provides a hardware environment for developing and evaluating designs. Because its features include a GTX transceiver with SMA connection and a transceiver with SFP connection, and the PS is the same as the Zedboard, this was the first board that was programmed with a working demo of the project. That first demo was still very rough and inefficient.

Finally, we ported our design to the AXIOM board. The AXIOM Board is designed to be the perfect combination of High-Performance Computing, Embedded Computing and Cyber-Physical Systems. It is meant to be an ideal platform for real-time data analysis of

a huge amount of data in a short time frame, machine learning, neural networks, server farms and so on. The Axiom board has four USB Type C ports, and also 2 USB Type A – not to mention the mini-DP connector. The board also features an RJ-45 Ethernet connector with a Gigabit Ethernet transceiver.
Its features include:

- Wide boot capabalities: eMMC, Micro SD, JTAG
- Heterogeneus 64-bit ARM FPGA Processor: Xilinx Zynq Ultrascale+ ZU9EG
  - 64-bit Quad-Core ARM Cortex™-A53 MPCore @ 1.2GHz with L1 Cache 32KB I/D per core, L2 Cache 1MB, on-chip Memory 256KB
  - 32-bit Dual-core ARM Cortex-R5 MPCore R5 @ 500MHz with L1 Cache 32KB I/D per core, Tightly Coupled Memory 128KB per core
  - Mali-400 MP2 GPU with L2 Cache 64KB
  - swappable DDR4 SO-DIMM RAM @ 2400MT/s (up to 32GB) for the Processing System
  - 600K System Logic Cells
  - 548K CLB Flip-Flops
  - 274K CLB LUTs
  - 2,520 DSP Slices
  - 12 GTH transceivers (8 on USB Type C connectors + 4 on HS connector)
- Easy rapid prototyping, because of the Arduino UNO Pinout

The processor is 64-bit, which allowed the use of 64bit words for our design, making it much more efficient.

## 5.2 Experimental results

To test the throughput of our design, we wrote a test code in the SDK, that sets the NI up to send the packets through physical interface 2 (Tx) and receive them through physical interface 1 (Rx), basically a hardware loopback mode. To get timestamps we used the internal PS timer, and we took timestamps at several points during a transfer, giving us values for the FIFO write latency, the PL packet generation and transfer latency, the IRQ read latency and finally the FIFO read latency. For all RDMA and LONG transfers the FIFO write latency, 270cc ,the IRQ read latency, 140cc, and the FIFO read latency, 320cc, are exactly the same, as an average of 1000 runs for every transfer is shown and the descriptors written and read are the same size for all transfers. The PL packet generation latency and the throughput are as shown in table 5.1.

Table 5.1 RDMA and LONG PL latency and throughput

| Type | Payload | Latency | Throughput |
|---|---|---|---|
| LONG_DATA | 15 Bytes | 220cc | 0.028 Gbps |
| | 255 Bytes | 285cc | 0.455 Gbps |
| | 4095 Bytes | 1010cc | 3.9 Gbps |
| | 16383 Bytes | 3420cc | 6.13 Gbps |
| | 65535 Bytes | 12750cc | 7.3 Gbps |
| RDMA_WRITE | 15 Bytes | 225cc | 0.028 Gbps |
| | 255 Bytes | 300cc | 0.455 Gbps |
| | 4095 Bytes | 1050cc | 3.86 Gbps |
| | 16383 Bytes | 3460cc | 6.08 Gbps |
| | 65535 Bytes | 12900cc | 7.27 Gbps |
| RDMA_READ | 15 Bytes | 285cc | 0.022 Gbps |
| | 255 Bytes | 350cc | 0.422 Gbps |
| | 4095 Bytes | 1100cc | 3.7 Gbps |
| | 16383 Bytes | 3520cc | 5.98 Gbps |
| | 65535 Bytes | 12960cc | 7.23 Gbps |

For RAW_NEIGHBOUR and RAW_DATA transfers, the FIFO write and FIFO read latencies do not stay the same, only the IRQ read latency is consistent, as shown in table 5.2.

Table 5.2 Raw latency and throughput

| | 15 Bytes | 127 Bytes | 255 Bytes |
|---|---|---|---|
| FIFO write | 800cc | 1430cc | 2730cc |
| PL transfer | 100cc | 114cc | 135cc |
| IRQ read | 110cc | 110cc | 110cc |
| FIFO read | 550cc | 870cc | 1520cc |
| Throughput | 0.06 Gbps | 0.076 Gbps | 0.085 Gbps |

As shown below (figure 5.1), the Raw transfers have very low throughput, with the bottleneck being the AXI4 write and read transfers to and from the FIFOs. The RDMA and LONG transfers (5.2) for small payloads are also fairly inefficient, with $\tilde{1}$.4 Gbps throughput for 1 KByte payload, but scale up quite fast, with 3.8 Gbps throughput for 4 KBytes payload, 5 Gbps throughput for 8 KBytes payload and finally 7.27 Gbps throughput for the maximum payload of 64 KBytes. Also it should be noted that the transfers are synchronous, meaning that before starting a new transfer, the PS waits for the previous one to complete. Our partners at Evidence have a working PetaLinux version that uses the NI and with asynchronous transfer report throughput of up to 9.3 Gbps for RDMA_WRITE transfers of max payload. That is because by sending many transfers together the FIFO write, IRQ read and FIFO read latency is pipelined. Unfortunately, because in our basic SDK test project we have not implemented multitasking to handle the interrupts, and receive packets while at the same time sending packets, we do not have asynchronous test data.

Fig. 5.1 Raw Transfer Throughput

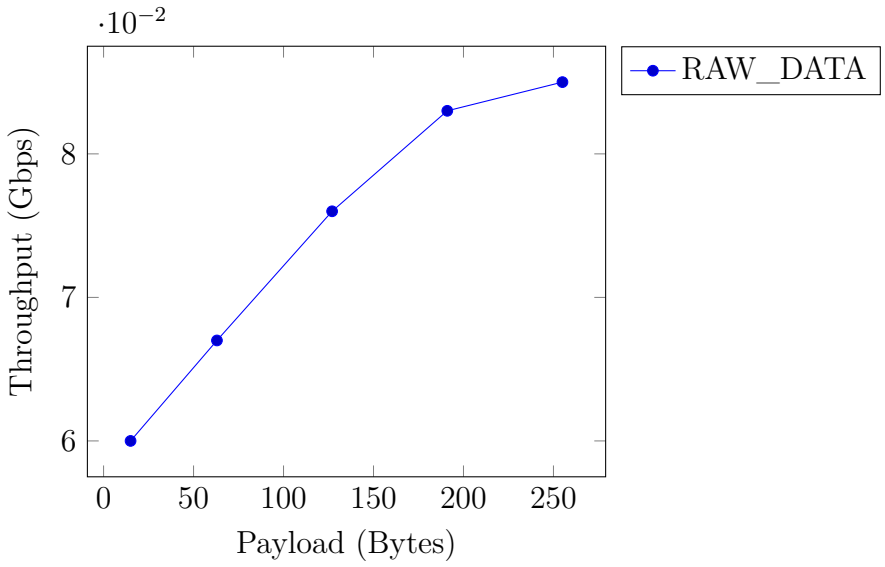
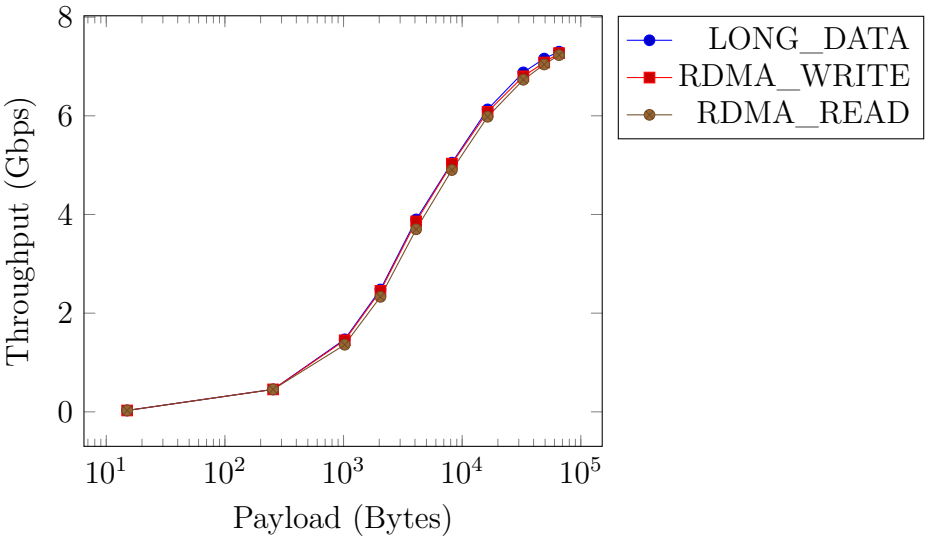



Fig. 5.2 DMA Transfer Throughput

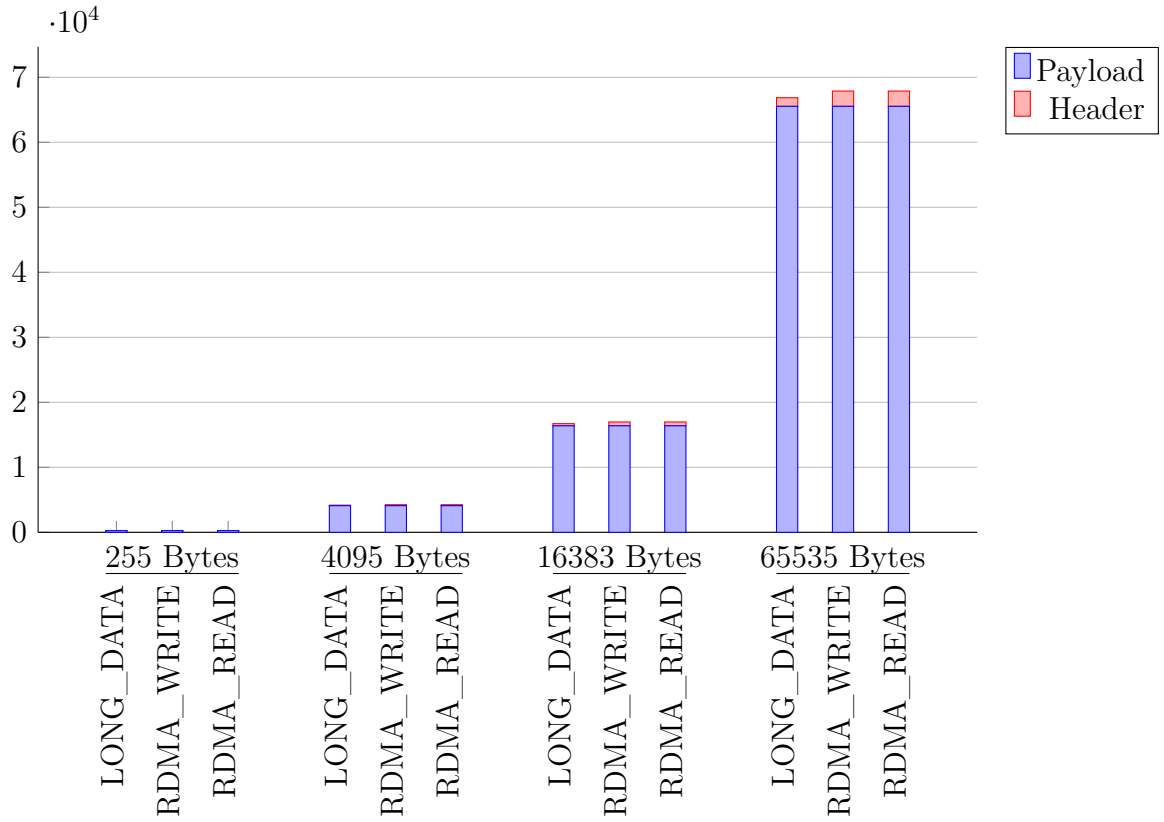



Finally, we calculated based on the payload size, how many packets were generated per transfer, and from that we estimated the sum of the headers, as shown in table 5.3 and figure 5.3 below.

Table 5.3 DMA Transfers with header size per payload size

| | LONG_DATA | | RDMA_WRITE | | RDMA_READ | |
|---|---|---|---|---|---|---|
| | Header Size | % | Header Size | % | Header Size | % |
| 15 Bytes | 5 Bytes | 33% | 9 Bytes | 60% | 9 Bytes | 60% |
| 255 Bytes | 10 Bytes | 3.92% | 18 Bytes | 7.06% | 18 Bytes | 7.06% |
| 4095 Bytes | 85 Bytes | 2.08% | 153 Bytes | 3.74% | 153 Bytes | 3.74% |
| 16383 Bytes | 335 Bytes | 2.04% | 594 Bytes | 3.63% | 594 Bytes | 3.63% |
| 65535 Bytes | 1325 Bytes | 2.02% | 2349 Bytes | 3.58% | 2349 Bytes | 3.58% |

Fig. 5.3 DMA Transfers with header size per payload size




That means that as the payload size gets bigger the transfer becomes more efficient, with the overhead reaching a minimum of about 2% for LONG_DATA transfers and about 3.6% for RDMA transfers. That explains why the LONG_DATA transfers have slightly better throughput than the RDMA transfers. The reason the RDMA_WRITE transfers are slightly faster then the RDMA_READ transfers is that the latter have

the extra latency of the RDMA_READ packet reaching the remote node to start the RDMA_RESPONSE transfer, as explained in Chapter 3.

## 5.3 Comparison to related work

As many-core and multi-core architectures become more relevant, the research on this subject will expand even more. Our design is comparable to the existing research, as it supports RDMA transfers, with the data written in a specific part of the memory, used similarly to a scratch-pad memory, has event notification using interrupts and a 16 core network of four AXIOM boards has been successful.

# Chapter 6

# Conclusions

With the expansion of research upon Cyber-Physical Systems, the Axiom project is very relevant, as the outcome of this project will not only be a functional and very capable board, but also a step forward in this research field.

## 6.1 Brief conclusions

This thesis' outcome is a completely functional Network Interface, that along with the Router comprise the Interconnect of the Axiom board. The Network Interface we designed follows all the guidelines needed for it to be adapted into the Axiom project. It is cost effective, with about 16% usage of the FPGA's available resources. It is efficient, with utilization of up to 7.3 Gbps of the available 10 Gbps USB type-c transceivers with our tests, and with reports from the project's partners of throughput up to 9.3 Gbps. The NI we designed and implemented has already been integrated to the PetaLinux processing system that will be provided along with the Axiom board, and the OmpSs programming model is being tested in networks of Axiom boards with very promising results.

In Chapter 3 we analyze the Design Architecture, first by reviewing the needed functionality, the supported transfers, their usage and their limitations, and then by analyzing the architecture of each module and it's job as part of the whole. Then in Chapter 4 we explain how through this thesis we became adept at the usage of the Xilinx Vivado Suite, by presenting the tools and their basic functionality, along with their design flow and our developing and debugging process. The second part of Chapter 4 is the implemented design, with information about the creation of each module, the problems that occurred and how we overcame them, and the detailed usage for the IPs provided by Xilinx. The

final part of Chapter 4 is the resource utilization on the Axiom board. In Chapter 5 we reference the boards we used for this project, and their basic functionality and proceed to analyze the throughput we achieved for each transfer and extract several information about the NI.

## 6.2 Next steps for improving the NI architecture / implementation

While the outcome is functional and with good results, several improvements could be made to the performance. Some examples would be:

- **Enabling Cache Coherence.** While the NI throughput is good, a great way to speed up performance, if not apparent, is enabling Cache Coherence for the LONG and RDMA transfers. Since the maximum payload of 64 KBytes is a fairly big payload to read from the DDR, being able to use the cache would speed up both the reads from the Datamover to send the data, and the reads from the PS after a transfer is complete, a speedup that would not show in the NI, but would be significant.
- **Support for even larger transfers.** As figure 5.2 shows, while for larger payloads, the throughput has started to stabilize, it has not reached peak value yet. The idea is that doubling the max payload size would provide even more efficient transfers with higher throughput.
- **Utilize more GP lanes than one per interface.** The USB type-c cable and interface provide two transceivers per interface. that means that it is possible to utilize both of them for 128 bit words instead of the 64 bit words we use. in such a case the speedup would be apparent, as while the whole project would have to be reimplemented, the latency of each transfer would be cut almost in half, effectively doubling our throughput.

We hope to revisit this project and tackle some of the suggested improvements, hopefully attaining even greater result.

# References


[1] D. Theodoropoulos et al., “The AXIOM project (Agile, eXtensible, fast I/O Module),” in Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on, July 2015 2015, pp. 262–269.

[2] Carlos Alvarez et al., “The AXIOM Software Layers,” ”ELSEVIER Microprocessors and Microsystems”, July 2016.

[3] Giorgi, R., Bettin, N., Gai, P., Martorell, X., & Rizzo, A. (2017). AXIOM: A Flexible Platform for the Smart Home. In Components and Services for IoT Platforms (pp. 57-74). Springer International Publishing.

[4] Katevenis, Manolis GH, Vassilis Papaefstathiou, Stamatis Kavadias, Dionisios Pnevmatikatos, Dimitrios S. Nikolopoulos, and Federico Silla. "Explicit communication and synchronization in SARC." IEEE micro 30, no. 5 (2010): 30-41.

[5] S. G. Kavadias, M. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, “On-chip communication and synchronization mechanisms with cache-integrated network interfaces,” in Conf. Computing Frontiers. ACM, 2010, pp. 217–226.

[6] Kavadias, S., Katevenis, M., Zampetakis, M., & Nikolopoulos, D. S. (2012). Cache-integrated network interfaces: Flexible on-chip communication and synchronization for large-scale CMPs. International Journal of Parallel Programming, 1-22.

[7] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, D. Nikolopoulos, Formic: cost-efficient and scalable prototyping of manycore architectures, in 2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (IEEE, New York, 2012), pp. 61–64

[8] Papaefstathiou, V., Pnevmatikatos, D., Marazakis, M., Kalokairinos, G., Ioannou, A., Papamichael, M., ... & Katevenis, M. (2007, July). Prototyping efficient interpro-

cessor communication mechanisms. In Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on (pp. 26-33). IEEE.

[9] R. Bolla et al., "Energy efficiency in the future internet: a survey of existing approaches and trends in energy-aware fixed network infrastructures," in IEEE Communications Surveys & Tutorials, vol. 31, 2011, pp. 223–244.

[10] M. Valera, S.A. Velastin, "Intelligent distributed surveillance systems: A review," in IEE Vision, Image and Signal Processing, April 2005, pp. 192–204.

[11] Pentland, Alex P., "Smart rooms," in Scientific American 274.4, 1996, pp. 54–62.

[12] L. C. De Silva, C. Morikawa, I. M. Petra, "State of the art of smart homes," in Scientific American 274.4, vol. 25, October 2012, pp. 1313–1321.