Technical University of Crete

School of Electrical and Computer Engineering

# Design and HLS Implementation of the AXIOM Network Router

Ioannis Tampakakis

Thesis Committee

Professor Dionisios Pnevmatikatos (ECE)

Associate Professor Polyxronis Koutsakis (ECE)

Dr. Dimitris Theodoropoulos (ECE)

Chania, November 2017

Πολυτεχνειο Κρητης

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

# Σχεδίαση και υλοποίηση σε **HLS** του **Router** για το δίκτυο του **AXIOM**

Ιωάννης Ταμπακάκης

Εξεταστική Επιτροπή

Καθ. Διονύσιος Πνευματικάτος (ΗΜΜΥ)

Αναπ. Καθ. Πολυχρόνης Κουτσάκης (ΗΜΜΥ)

Δρ. Δημήτρης Θεοδωρόπουλος (ΗΜΜΥ)

Χανιά, Νοέμβριος 2017

# Abstract

The AXIOM project (Agile, eXtensible, fast I/O Module) aims at researching new architectures for Cyber-Physical Systems (CPSs) of the future. One of the project's goals is to design a small and affordable board that could be used as a LEGO-style CPS node and provide flexibility, energy efficiency as well as modularity through high speed interconnection of many nodes which in turn will pave the way for a fast and reliable communication. However, since the communication layer is the backbone of any distributed system, if not carefully designed and implemented it can lead to network congestion, low throughput, high latency and generally performance degradation. In this work we present a router design and implementation that enables a seamless interconnection of multiple AXIOM boards spanning inside a high throughput and low latency network. Based on a cost-efficient network infrastructure, the router follows the paradigm of keeping resource utilization to a minimum while using the well-known and trusted router primitives. Virtual channels are implemented in order to facilitate a prioritized packet arbitration which combined with Virtual Cut-Through switching and a deterministic routing algorithm provides efficient and reliable packet forwarding. Packet flow control is implemented through the transmission of interrupt-like signals between adjacent nodes leading to lossless packet transfers. With a fully pipelined architecture and an effective streaming communication protocol the router offers a low-cost, high-speed interconnection for the network's nodes.

# Περίληψη

Το AXIOM project (Agile, eXtensible, fast I/O Module) στοχεύει στην έρευνα νέων αρχιτεκτονικών για τα Cyber-Physical συστήματα CPSs του μέλλοντος. Ένας από τους σκοπούς του project είναι ο σχεδιασμός μίας μικρής και οικονομικά προσιτής πλατφόρμας η οποία θα μπορεί να χρησιμοποιηθεί σαν κόμβος τύπου LEGO σε ένα CPS, και ταυτόχρονα θα παρέχει προσαρμοστικότητα, ενεργειακή οικονομία αλλά και δομοστοιχειώτητα μέσω υψηλής ταχύτητας διασύνδεση πολλών κόμβων, το οποίο με τη σειρά του θα ανοίξει τον δρόμο για ταχεία και αξιόπιστη επικοινωνία. Δεδομένου ότι το επίπεδο της επικοινωνίας είναι η ραχοκοκαλιά κάθε κατανεμημένου συστήματος, αν δεν σχεδιαστεί και υλοποιηθεί προσεκτικά μπορεί να οδηγήσει σε συμφόρηση του δικτύου, χαμηλό throughput, μεγάλες καθυστερήσεις και γενικά να υποβαθμίσει την απόδοση. Σε αυτή την εργασία παρουσιάζεται ο σχεδιασμός και η υλοποίηση ενός router το οποίο επιτρέπει την αδιάλειπτη διασύνδεση πολλών AXIOM boards τα οποία εκτείνονται μέσα σε ένα δίκτυο υψηλού throughput και χαμηλού latency. Βασισμένο σε μια δομή δικτύου χαμηλού κόστους, το router ακολουθεί την αρχή της ελάχιστης δυνατής χρήσης των πόρων της πλατφόρμας ενώ αξιοποιεί τα γνωστά και δοκιμασμένα δομικά στοιχεία των routers. Χρησιμοποιεί ιδεατά κανάλια ώστε να επιτρέπει την διαιτησία βάσει προτεραιότητας τύπου πακέτου το οποίο σε συνδυασμό με μία τεχνική Virtual Cut-Through μεταγωγής και ένα ντετερμινιστικό αλγόριθμο δρομολόγησης, προσφέρει αποδοτική και αξιόπιστη προώθηση πακέτων. Ο έλεγχος της ροής των πακέτων υλοποιείται μέσα από σήματα που προσομοιάζουν interrupts, τα οποία στέλνονται μεταξύ γειτονικών κόμβων με αποτέλεσμα την μεταφορά πακέτων δίχως απώλειες. Με μία πλήρως pipelined αρχιτεκτονική και ένα αποδοτικό streaming πρωτόκολλο επικοινωνίας, το router προσφέρει μία χαμηλού κόστους και υψηλής ταχύτητας διασύνδεση των κόμβων του δικτύου.

# Acknowledgements

First and foremost, I would like to thank my supervisor and mentor, Dr. Dionysios Pnevmatikatos for his inspiration, guidance and the trust he showed in me. The opportunity he offered me undoubtedly helped me evolve both on a professional and personal level. I would also like to express my gratitude to Dr. Polychronis Koutsakis and Dr. Dimitris Theodoropoulos for their interest in my work and for contributing to its evaluation as members of the thesis committee.

I would also like to thank everyone from MHL for their meaningful help and suggestions during the implementation of my work. Our communication and weekly meetings proved to be invaluable during the course of my thesis.

Next, I would like to thank my childhood friends as well as my ECE friends for all these great years and moments we had.

Last, but most important, I would like to thank my parents and my sister for their continuous support and patience during these years. I would have probably never reach this point without them. I hope I will be able to repay you somewhere down the line.

# Contents

# List of Figures

# List of Tables

## LIST OF TABLES

# Nomenclature

### Roman Symbols

| | |
|---|---|
| $APU$ | Accelerated Processing Unit |
| $AXIOM$ | Agile, eXtensible, fast I/O Module |
| $BRAM$ | Block RAM |
| $cc$ | Clock Cycle(s) |
| $CLB$ | Configurable Logic Block(s) |
| $CPS$ | Cyber-Physical System |
| $FF$ | Flip Flop(s) |
| $IBUFF$ | Input Buffering |
| $II$ | Iteration Interval |
| $LC$ | Link Controller |
| $LT$ | Link Traversal |
| $MPI$ | Message Passing Interface |
| $NI$ | Network Interface |
| $NoC$ | Network on Chip |
| $RC$ | Route Computation |

## NOMENCLATURE

| | |
|---|---|
| $RDMA$ | Remote Direct Memory Access |
| $RT$ | Routing Table |
| $SA$ | Switch Allocation |
| $SAST$ | Switch Allocation and Switch Traversal |
| $SATA$ | Serial Advanced Technology Attachment |
| $ST$ | Switch Traversal |
| $UFC$ | User Flow Control |
| $VC$ | Virtual Channel |
| $VCA$ | Virtual Channel Allocation |
| $VCT$ | Virtual Cut Through |
| $XBAR$ | Crossbar |
| $XonXoff\_RX$ | Downstream XonXoff Controller |
| $XonXoff\_TX$ | Local XonXoff Controller |

# Chapter 1

# Introduction

Entering the Cyber-Physical era, both objects and people are treated as nodes making up the same digital network of exchanging information. To that effect, we expect that "things" or systems will eventually become somewhat smart as people, something that will establish a rapid and close system-system as well as system-human and human-system interaction as a common thing. There has been a tendency for CPS exploitation by many applications that require interactions between humans and the physical environment.
A CPS integrates a set of hardware-software components to interact with the physical world. What is expected from these systems is the ability to react in real-time, provide sufficient computational power for the assigned tasks, be as energy efficient as possible for such tasks, scale up through modularity, allow for easy programmability across performance scaling, and efficiently utilize the best existing standards at minimal costs. However, there is a noticeable performance limit in mainstream CPSs, which presents a challenge for the increasingly heavy load tasks, namely video surveillance and smart home applications.
The AXIOM project (Agile, eXtensible, fast I/O Module) [1] aims at researching new software/hardware architectures for the Cyber-Physical Systems (CPSs) of the future in which the above expectations are possibly realized. Ultimately, one of the goals of the project is to design a small affordable board that could be used as a LEGO-style CPS node which will feature general purpose capability coupled with reconfigurable resources. This paves the way for building larger systems with more performance while keeping the programming task simple by using a familiar shared memory programming model. The

scalability/extensibility that AXIOM brings, is achieved by providing easy programmability, not limited only to the single module but also to the larger systems consisting of several modules tightly interconnected.

## 1.1 Motivation

Current systems are typically isolated and the only way to build larger systems is to re-design a new system from scratch by adding more resources (e.g., more cores, more memory, etc.). As pointed out above, AXIOM aims at realizing a small custom board that provides flexibility, energy efficiency and modularity, based on the Xilinx Zynq devices that feature a dual- or quad-core ARM processor, tightly coupled with FPGA fabric. In the manycore era, AXIOM aims to enable modularity through high speed interconnection of many AXIOM nodes as seen in Figure 1.1. That interconnection plays

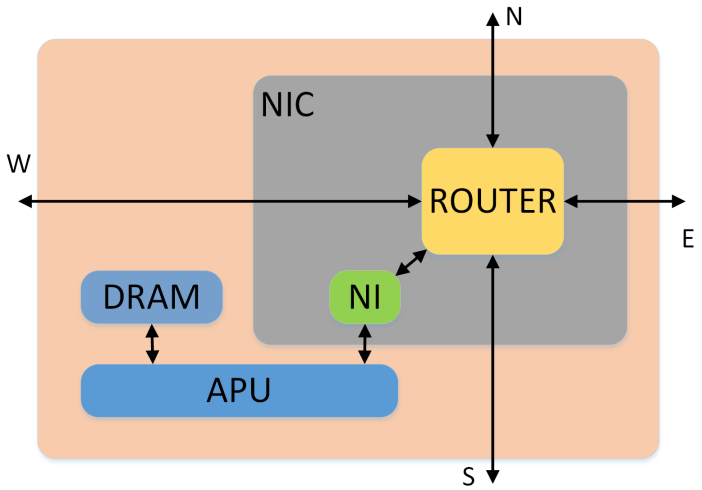



Figure 1.1: The AXIOM node, featuring the proposed router.

a crucial role for the need of providing both fast and reliable communication (including lossless control flow as, e.g., Infiniband, but with a simplified scope and cost). However, the communication layer being the backbone of any distributed system, if not carefully designed and implemented can lead to network congestion, low throughput, high latency, and poor overall system performance. Keeping in mind that the AXIOM board can be considered as the building block of future multi-node CPSs, what is of utmost importance

is utilizing a cost-efficient and high-throughput network, by optimizing and customizing the networking logic, so as to maintain fast inter-board communication leading to more effective systems and faster applications. In this work, we will present a router design that creates the possibility of a seamless interconnection of systems spanning in multiple boards as shown in Figure 1.2.

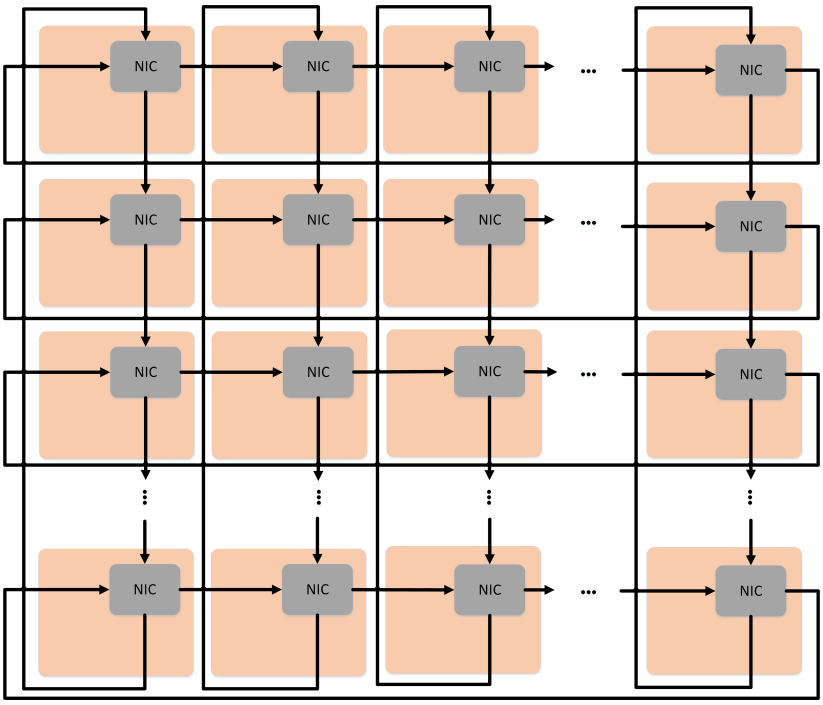



Figure 1.2: An network of a 2-D torus topology, based on the AXIOM node (Figure 1.1). Apart from a 2-D torus, the AXIOM node could form ring and 2-D mesh topologies.

## 1.2 Thesis Contribution

This thesis, contributes the design and development of a router architecture that answers the demand of efficient and low cost interconnection among boards which are nodes in a network. The router achieves the highest performance possible so as to meet application demands, while limiting the resource utilization to a minimum in order to make its implementation possible on small-ranged reconfigurable SoCs. The key points to highlight are the following:

- Multiple topology support (section 3.1)
- Virtual Cut-Through packet switching (section 3.3)
- Fully pipelined design (section 4.1)
- Packet-priority based arbitration (section 4.3)
- Fast inter-board communication (section 5.3)
- Low cost implementation (section 5.4)

## 1.3 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we review related work regarding router design and implementation as well as comparisons made to them. In Chapter 3 we present the network infrastructure upon which we designed our router. This includes a detailed analysis of supported topologies along with the packet forwarding method and the different packet types inside the network. Chapter 4 describes the proposed router architecture, and discusses key points of the design such as pipeline stages, routing table configuration, scheduling as well as packet flow control. Chapter 5 provides a more technical description of the specific steps followed during the implementation, which entails describing the tools used, giving detailed schematics of the router and of course the final resource utilization on the boards. We then, in Chapter 6, present the experimental results of the proposed router design implemented on the actual AXIOM boards. Finally, Chapter 7, we conclude our work, and suggest directions towards future work.

# Chapter 2

# Related Work

Scalable interconnect architectures constitute the basis on which heterogeneous computing platforms and their unifying programming environments are being developed. Parallelism in its core, is all about the efficient co-operation of connected nodes which is enabled by a meticulously designed interconnect. Topics such as NoC topologies, input versus output queueing and scheduling have been thoroughly researched and can be considered textbook material ([2], [3], [4]).
Below, we present some of the works that are concerned with FPGA interconnection and data transferring based on NoC routers.

### High Performance Network-on-Chip Architectures

Psarras in his PhD thesis [5] proposes three different alternative architectures, which can significantly improve the performance of NoCs, or lead to an overall lower power consumption. One of those, is a pipelined router architecture, called ShortPath [6], that parallelizes - for the first time - the allocation steps involved in the operation of a VC-based router without resorting to speculation. Most importantly, ShortPath is augmented with an always productive pipeline bypassing mechanism, which skips all stages without contention, and "fast-forwards" the flits to the first encountered point of contention.
The other two, based on appropriate wire engineering, take advantage of fast link traversal to rapidly send and receive flits between neighbouring nodes in half a clock cycle. Under this clocking principle, two design alternatives are explored, which enable half-cycle and Double-Data-Rate (DDR) link traversal. The proposed approaches can significantly

improve network performance, or decrease the area and/or power cost of the NoC. Although not obvious at first glance, half-cycle link traversal opens up new possibilities for reducing wire capacitance. By harnessing these opportunities, RapidLink [7] renders the half-cycle-delay requirement easier to achieve and potentially extends half-cycle traversal capabilities to longer links.

### Low-Latency Virtual-Channel Routers for On-Chip Networks

This paper [8] by the University of Cambridge presents a low latency router for Networks on Chips that shows how the removal of control overheads - such as routing and arbitration logic - from the critical path can reduce the clock cycles and latency to a minimum. Simulations demonstrated important cycle time improvements whilst allowing single cycle flit routing.

### A Comprehensive Comparison between Virtual Cut-Through and Wormhole Routers for Cache Coherent Network On-Chips

In [9], the point of interest is how the scaling of semiconductor technology can change the concept of Virtual Cut-Through (VCT) switching method being inefficient due to larger buffers compared to Wormhole switching. Wang et al. through detailed hardware implementations evaluate the hardware costs of Wormhole and VCT switching methods with both deterministic and adaptive routing. In hindsight the paper proves that VCT routers are actually efficient NoC routers as the critical path can be reduced up to 27% while area and power overheads are trimmed down.

### Merged Switch Allocation and Traversal in Network-on-Chip Switches

In this work, G. Dimitrakopoulos et al. took a different approach regarding innovative switch designs. Differentiating from the architecture-level point of view solutions that took for granted most of the switch primitives and the way they are organized - albeit efficient, they aimed for a design of new macros from scratch, equipping them with the ability of handling concurrent arbitration and multiplexing, while maintaining their scalable qualities (number of ports and data width). The proposed optimization concerns

a new structure called Merged ARbiter and multipleXer (MARX) [10] that merges the functionality of the arbiter and the crossbar's multiplexers, in a new circuit that performs the two distinct steps of switch allocation and switch traversal in parallel. As far as the arbitration policy is concerned both simple round robin as well as more complex weight-based selection policies that offer much better throughput can be implemented following the proposed design methodology. However, even with more complicated selection policies the router's timing overhead when implemented with MARX units was negligible when compared to a simple round-robin policy router with separate arbiter and multiplexer, which is considered one of the most significant achievements of this work.

### A High-Throughput Distributed Shared-Buffer NoC Router

In this letter [11], a new router design that aims to emulate an output-buffered router (OBR) practically based on a distributed shared-buffer (DSB) router architecture is proposed. Along with efficient pipelining and a novel flow control, the suggested architecture uses two crossbar stages with buffering sandwiched in between. The packets are stored there with two constraints. The first is that packets arriving at the same time can not be stored to the same buffer and secondly a packet can not be stored to a buffer that already holds a packet with the same departure timestamp. Additionally flow control is implemented on a flit-by-flit basis instead of a per packet basis. The proposed DSB architecture outperforms IBR configurations when the amount of buffering is increased, whereas in terms of power consumption there is a noticeable power cost increase which can be justified for NoC applications which demand high bandwidth.

### A Gracefully Degrading and Energy-Efficient Modular Router Architecture for On-Chip Networks

The study in [12] proposes a novel fine-grained modular router architecture. Aside from using decoupled parallel arbiters, compared to existing designs it utilizes smaller crossbars for row and column connections targeting less output port contention all while implementing a new switch allocation method called Mirroring Effect. What stands out is the feature of graceful degradation of the network in case of permanent faults as well as the reduction of the dynamic power consumption. The achieved packet latency ends

up being 4-40% lower than that of two existing architectures while power consumption drops 6-20% lower.

### A 4.6 Tbits/s 3.6 GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS

Princeton University, in collaboration with Intel, presented a NoC router design, targeted at a 36-core shared memory chip multiprocessor system which targeted a clock frequency of 3.6 GHz, which in itself poses a challenge considering that any NoC router should maintain power and area efficiency. To that effect, Kumar et al. in [13] made several unique circuit and micro-architectural innovations and design choices, the most important being a novel high throughput and low latency switch allocation mechanism (SPAROFLO), a non-speculative single-cycle pipeline, a low-complexity virtual channel allocator and finally a dynamically managed shared buffer design. The results yielded by this design showed a significant latency reduction both at no-load and saturation situations.

### The FORMIC project

In 2012, FORTH-ICS introduced the Formic board [14], a cost-efficient building block designed for scalable multi-board prototypes. Formic filled an important gap of both academic and commercial platforms, which either were too expensive or did not feature adequate SRAM and board-to-board connections. What was designed was a scalable hardware architecture based on which a 64-board, 512-core proof-of-concept prototype was implemented. Formic 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 first hardware design project which uses the Formic board is a prototype of a non cache-coherent manycore architecture, based on ideas of the SARC project [15]. Each board fits in its FPGA eight CPUs, their private L1 and L2 caches, eight GTP links at 3 Gbp/s and a full network-on-chip centered around a 22-port crossbar. A variable number of boards can be interconnected in a 3D-mesh using the GTP links, growing the system as required. In the 64 board architecture, a minimum sized packet of 36 bytes needed 3-4 cycles to traverse the internal network while board-to-board traversals add 5-6 cycles per hop. These delays were by then, in line with state-of-the art 2D-mesh multi-core architectures.

### Efficient Network Interface design for low cost distributed systems

This thesis [16] developed a high-throughput and cost-efficient Network Interface (NI) specifically tailored to the requirements of the AXIOM project [1]. The NI design was inspired by the FORMIC project and extended to facilitate the needs of the OmpSs programming model, including multiple data transfer types. To implement the proposed design three different message controllers were created, an interrupt system as well as the necessary interconnects. The end result was an efficient, fully pipelined network interface utilizing a small percentage of the available resources of the AXIOM board while achieving high throughput.

## 2.1 Comparisons

The AXIOM router, presented in the following chapters, focused mainly on a cost-efficient and a functionally reliable design. Additionally as most of the above works deal with SoC interconnects in large scale networks, our comparison will be more qualitative than quantitative.

From an architectural standpoint the AXIOM router managed to reduce the pipeline stages (section 5.2) just like MARX [10] did by merging stages, and ShortPath [6] by skipping stages with no contentions. However some of our stages were not single-cycle something that leaves us behind when compared to these two works. Regarding scheduling, when compared to MARX [10] we judge that both router designs provide intricate algorithms which can handle packet forwarding in a fair and efficient way.

From an implementing point of view, the AXIOM router does a fairly good job since it surpasses the maximum provided bandwidth of FORMIC [14] while being way more cost-efficient in terms of FPGA resources but induces a bigger latency.

# Chapter 3

# Network Infrastructure

Prior to presenting the router's architecture and implemented design, we need to define the broader context in which the router exists, and that is the network. In our case as described in section 1.1 the network's building blocks will eventually be the AXIOM boards, acting as nodes interconnected in a fast and cost-efficient way. Consequently, the way these nodes will be organised, as well as the messages that will be exchanged between them, set the router requirements such as the number of input and output ports, the number of virtual channels, as well as the scheduling and routing algorithm. This organization and these messages exchanged are described in the following paragraphs that deal with the supported network topologies and the way the packets are forwarded inside the network along with a brief description of the packets.
Before diving into details, let us name the basic features of the proposed AXIOM router:

- Multiple ad-hoc topology support
- Supports different message type transmissions
- Utilizes Virtual Channels to prioritize packets
- Uses Virtual Cut-Through switching

## 3.1 Supported Topologies

A network's topology is one of the main factors that affect performance. It defines the physical organization of the network composed by the nodes, and consequently the

available paths between all the nodes (routing algorithm) [17]. Simply because any other topology aside from ring or 2-D mesh and torus would add a significant overhead to either resources or logic, we decided to set as our goal a fully functional 2-D torus topology, with a ring topology acting as a milestone at the midpoint of this work. For any other, more complicated topology, we can expand the existing design and logic to achieve a more efficient network. Figure 3.1 presents the implemented topologies, depicting an example network of 8 nodes connected on a ring topology, a network of 9 nodes in 2-D mesh as well as an example of a 2-D torus topology.

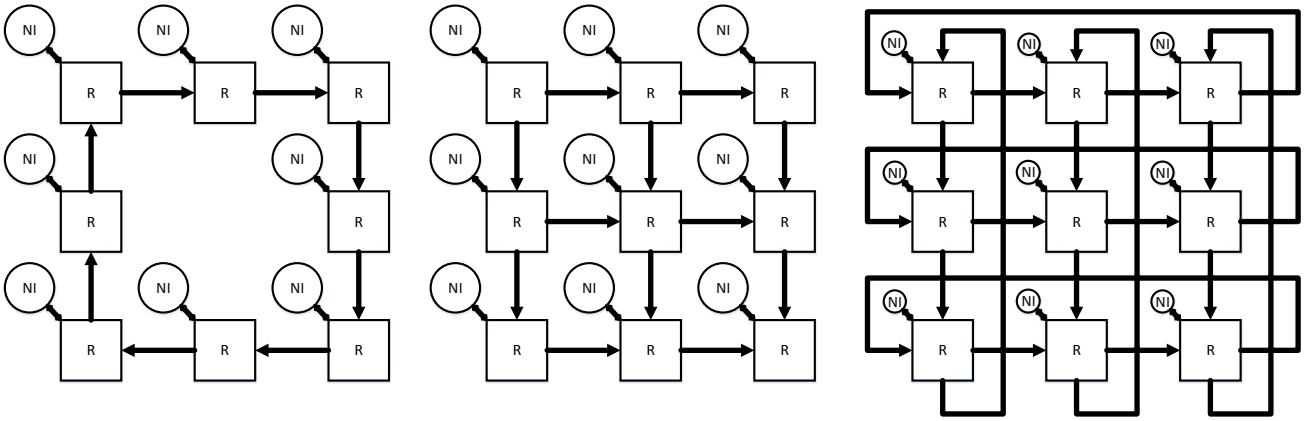



Figure 3.1: Ring (left), 2-D Mesh (center) and 2-D Torus (right) network topologies

## 3.2 Supported Packet Types

Although the router handles packets as chunks of data without really interfering with their contents, we judge that it is necessary to give a general description for each packet type and additionally present their respective headers, before moving on to the general architecture and implementation.

Two basic categories of data transfers are supported, those that transfer data from and to the system's APU, called RAW transfers, and those that transfer data directly from and to the memory of the APU with the use of a DMA engine, called RDMA transfers. Each of these include a number of packet types to facilitate the successful transmission of data from one node to another. These packets types are listed below along with a brief description of their purpose.

- **RAW_NEIGHBOUR** packets are 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.
- **RAW_DATA** packets are used to send data directly from one APU to another.
- **INIT** packets are sent from one node to another, when the first wants to inform the second that he is about to send an RDMA packet. It includes information regarding the message size about to be transmitted. It precedes LONG_DATA, RDMA_WRITE and RDMA_READ packets.
- **LONG_DATA** packets are used to send data from one node's memory to another's. It includes the source address of the sender's memory to read the payload as well as the payload size.
- **RDMA_WRITE** packets are almost identical to the LONG_DATA packet with the only difference being that the RDMA_WRITE also includes the destination address of the receivers memory where the payload will be written.
- **RDMA_READ** packets are used to request data from a remote node's memory. Includes the source address of the remote memory from which the data will be read, the payload size and the destination address that the data will be written on the local APU's memory.
- **ACK/NACK** are used at the end of any RDMA transfer to signify a successful or unsuccessful message transmission.

Overall, a packets header consists of 5 bytes with identical information for all packets and some extra bytes of information in the case of RDMA packets. The information included in a header contains the following:

1. The SOURCE field is 8 bits wide and represents the source node, where the packet initially got generated.
2. The DESTINATION field is 8 bits wide and its value is the packet's destination node. This value is used as the address of the routing table BRAM, which contains which router output the packet should request.

3. The MESSAGE ID field is 8 bits wide and its value is an id given by the PS and along with the SOURCE and DESTINATION fields create a unique id for each transfer.

4. The TYPE field is 3 bits wide and its value is the packet's type.

5. The PORT field is 3 bits wide and is used by the PS, for example to encapsulate Ethernet packets.

6. The VC field is 2 bits and dictates the packets priority

7. The PAYLOAD SIZE field is 8 or 16 bits, depending on the packet type, and holds the number of useful data bytes in the packet

Table 3.1: Packet types

RAW packet

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

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 1 | | |
| ... | | |

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 1 | | |
| ... | | |

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 | | |

## 3.3 Packet Switching

Each message (whether an RDMA transfer, RAW messages or a simple ACK) is broken down to packets, and each packet to flits as shown in Figure 3.2. Packets are transferred to

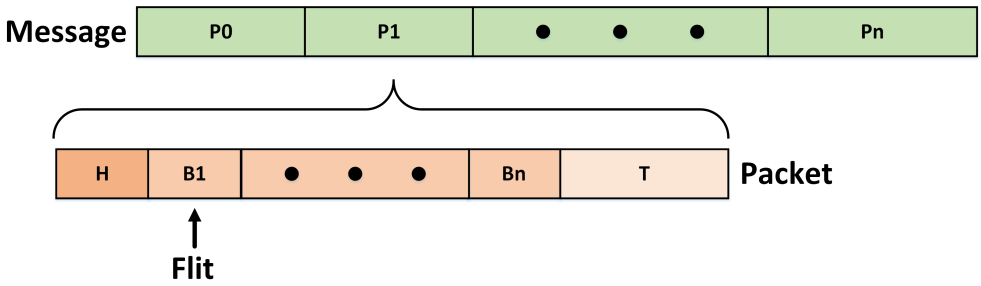



Figure 3.2: Data granularity of the network.

their destination through multiple routers along the routing path in a hop-by-hop manner. Each node's router keeps forwarding an incoming packet to the next node's router until the packet reaches its final destination. Switching techniques dictate when the router will forward the incoming packet to the neighbouring node, consequently affecting the network performance and buffer size required for each router. When starting off this thesis, there was a variety of switching methods discussed namely Store & Forward, Wormhole and Virtual Cut-Through (VCT), each with each own benefits and drawbacks [9].
The evaluation criteria for the above methods were exclusively cost and speed. Store & Forward switching came in last, both in cost and speed comparisons, since it utilizes packet-size buffers in contrast to Wormhole which uses flit-size buffers, and transmits data in a framing manner instead of streaming as Wormhole and VCT do. Wormhole, although being the most cost-efficient method with a minimum buffer size, and forwarding flits in streaming fashion, it presented the risk of Head-Of-Line (HOL) blocking with a packet's flits, being dispersed in multiple nodes inside the network which has zero additional buffer space to accommodate flits from other packets. Hence, Virtual Cut-Through became the only viable option, accepting the trade-off of buffers larger than the minimum size combined with an efficient and safe packet forwarding style.
However we should mention, that the VCT version we adopted, comes with a small tweak. Although - like in traditional VCT - the buffers are reserved at the granularity of packets, the size is not that of a single packet, but a variable multiple of the worst-case size packet

depending on the VC priority.
In Figure 3.3 we give a brief example of VCT switching. Each packet has a header flit which is decoded into information important to the packet's routing. The header flit, does not have to wait for the rest of the packet to move downstream, so as long as there is no crossbar conflicts or downstream buffer space issues, the header is immediately moved to the next node. The router holds the switch's configuration until the end of the packet and so the rest of the flits can simply follow the same route that was given to the header, without having to be routed from the beginning.

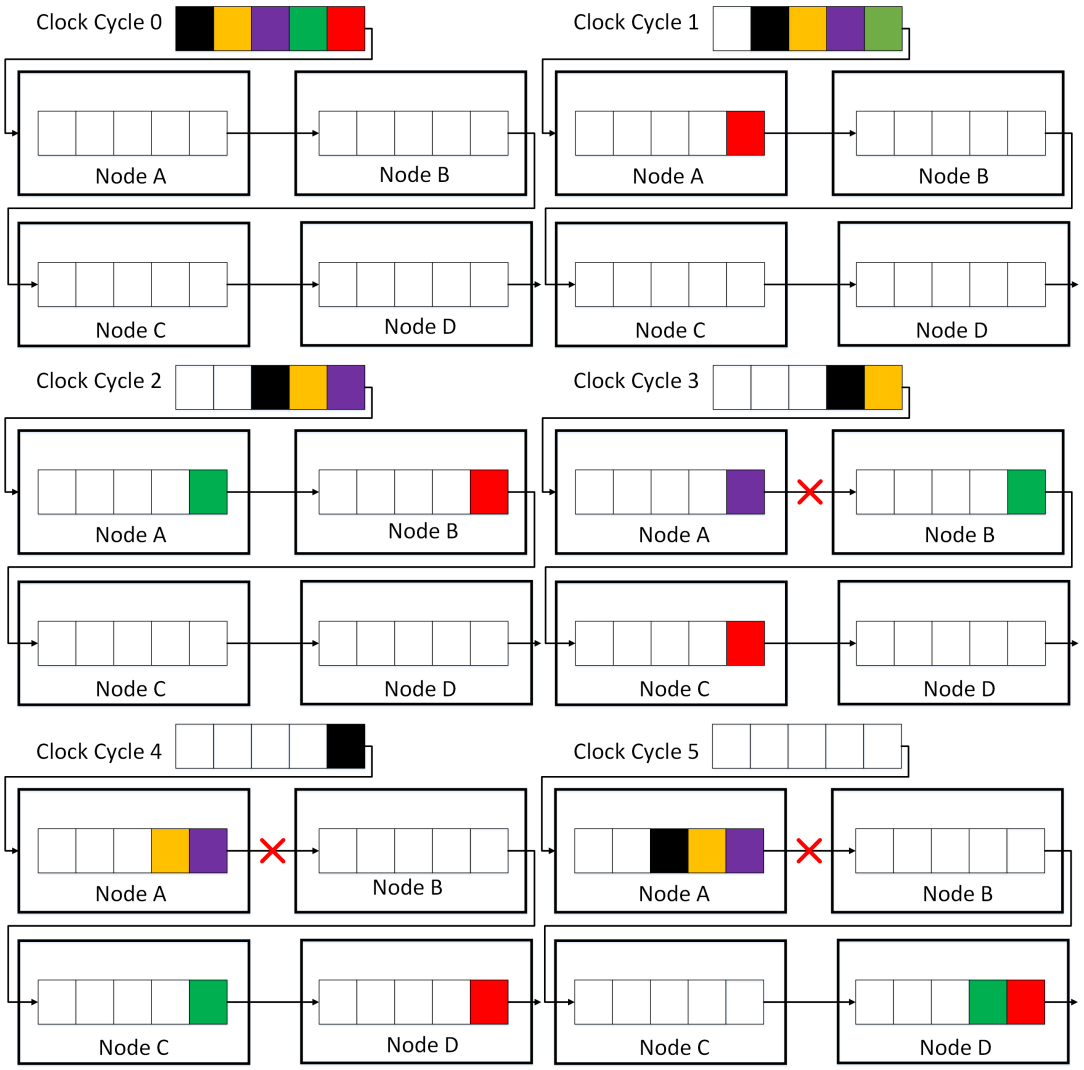



Figure 3.3: An example illustrating Virtual Cut-Through switching. A packet is being injected into the network (at Node A), traversing several nodes, before reaching its destination (Node D). Each node hop happens on a per-flit basis. When a link traversal is unavailable, flits that cannot make it through, accumulate to the buffer. The rest of the flits that made the hop before the link becoming unavailable, keep on moving downstream towards their destination.

# Chapter 4

# Router Architecture

In this chapter we will describe the proposed router architecture and highlight integral parts of the design such as the pipeline and which tasks each stage has to complete, the routing table and how it is configured to successfully help with the routing decisions, the scheduling that takes place among packets and finally the way the router handles traffic inside the network. Before moving on, in Figure 4.1 we present the architecture's block diagram so as to have a general idea of the design while diving into the description of individual blocks and tasks.

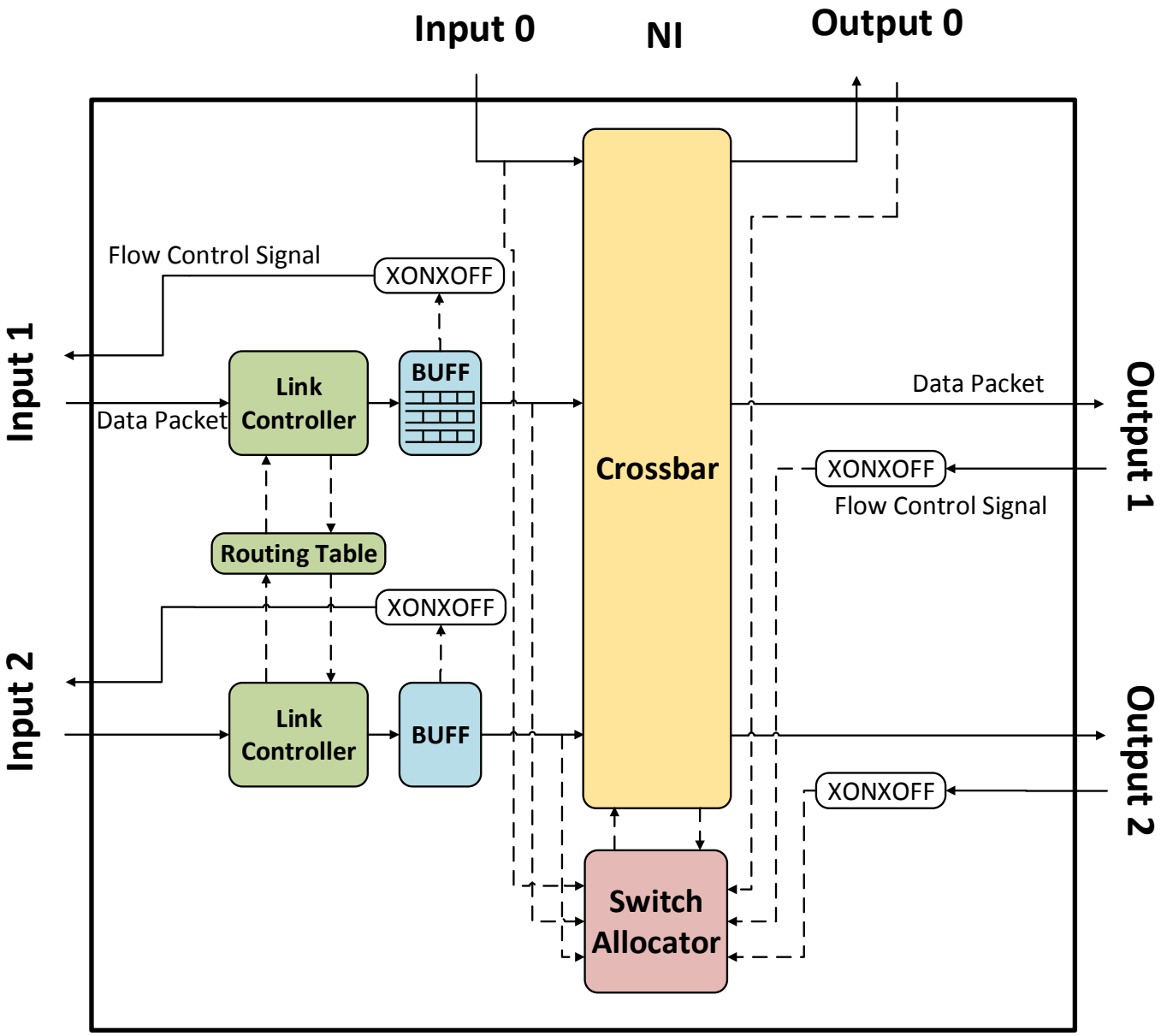



Figure 4.1: The proposed 3x3 router architecture.

## 4.1 Pipeline Stages

One of the key points of this thesis was creating a fully pipelined design. Although in theory that does not seem to create an issue, there where numerous problems that

we had to tackle, mainly related to the tools used. Our design consists of five pipeline stages: Route Computation (RC), Input Buffering (IBUFF), Switch Allocation (SA), Switch Traversal (ST), and finally Link Traversal (LT), each with its own distinct task. The task based view is presented in Figure 4.2.


Figure 4.2: The pipelined tasks. From left to right: Route Computation, Input Buffering, Switch Allocation, Switch Traversal, Link Traversal

## Route Computation

This is the higher-level decision making process that handles the task of directing packets towards their destination [18]. This decision is made by the routing algorithm implemented either by the NI or the router, which depending on its complexity uses a number of factors (network traffic, downstream buffer status, etc.) to determine what the next hop will be. Something most routing algorithms have in common is the access to the routing table. The routing table can be viewed as a map that dictates the path that should be followed for each possible destination. Again, depending on the complexity of the routing algorithm, the path can be predetermined and unique, or multiple and variable. In our case, where the routing algorithm is deterministic, non-minimal, and regular, RC will consist mainly of the routing table look up in order to forward packets to the next node.

However, there is a case where the routing table access is skipped and the router accepts as the packet's destination whichever output was dictated by the NI. This takes place only during the set-up of the network when the discovery algorithm is run. We should also mention that the routing table access is needed only for the header flit of the packet. The rest of the packet (body and tail flits) do not need any further routing, as they simply follow the header flit.

## Input Buffering

After deciding through which output the packet needs to exit the router, comes the process of temporarily storing the packet internally, before the router is ready to forward it. Although buffering is one of the main sources of added overhead to a router design, it's imperative that the packet is safely stored somewhere inside the router, in case the latter is not able to handle it yet, either due to increased network traffic or due to shortage of downstream buffer space or even in case of some packets taking priority over others. The size of these buffers can differ from one design to another, mostly based on the adopted switching technique. For instance, in Store & Forward switching, the buffer has to be big enough to store a whole packet (including header flit and body/tail flits), while in Wormhole switching there is no need for buffers bigger than a flit's size. In this work, we take a similar approach to Virtual Cut-Through as discussed in section 3.3, which led to using buffers larger than a single packet size.

Another requirement we had upon beginning the architecture design, was to take into consideration that we would not treat every packet the same way. More specifically, certain packets should be handled as soon as possible upon reaching a node inside the network, while for others we could afford postponing their forwarding depending on the importance of the information they held.

With that being said, we decided to discern the packet types into three different priorities; High, Medium and Low. Virtual Channel 2 holds the packets of the highest priority (ACKs, NACKs), Virtual Channel 1 is the buffer for medium priority packets (INIT, RAW_NEIGHBOUR, RAW_DATA, LONG_DATA, RDMA_WRITE) and lastly Virtual Channel 0 keeps the packets of the lowest priority (RDMA_READ). It should be noted that on the proposed architecture each VC priority uses a different size buffer, because although the agreed upon maximum capacity was two times the worst-case packet size, that size is variable between the different priorities.

Finally, a secondary but equally important task undertaken by this stage is informing the adjacent nodes about the buffers' status. In case the available space in a queue drops below or raises above a certain threshold, then any node directly connected to the node that the input buffer status has changed, is informed about that change through flow control messages.

## Switch Allocation

This is the main algorithmic stage of the pipeline. Here, all packets, from every lane and every virtual circuit, request one of the crossbar's output links and are halted until that output link is granted to them. For an output to be granted the router needs to assess which packets are of higher priority than others and attempt to pair their respective input streams to the requested output streams in the most efficient way possible. This means that any free crossbar output link should be allocated to a single input queue, but this however is not always the case. An output link might stay idle if the input buffers of the node it leads to are full, or simply because no input stream requests that crossbar output. As far as the request-grant handshake procedure is concerned, it takes place between the allocator and only the header flit of each packet. After the decision is made and the output has been granted, the remaining flits of that packet will be forwarded through the same output as the header, making it obvious that the allocator's task switches from granting an output, to maintaining the crossbar's input-output connection until the whole packet has left the node.

## Switch Traversal

In this stage, the input queues whose packets were granted a crossbar output, are paired to that output in order to reach the link that will inject them back into the network (or to the Network Interface, in case this node is the packets destination). With a unidirectional 2-D torus topology in mind, as seen in section 3.1, in each node there are two inputs from the network, and one from the network interface that connects the router with the PS. The same is true for the output links. This leads to a three-to-three ($3x3$) crossbar design. The way these inputs connect with the outputs is through a number of multiplexers on the input side, and a number of demultiplexers on the output side all connected to the output of every multiplexer.

## Link Traversal

Link Traversal stage is the final stage of our pipeline. After moving through the crossbar, packets head either to the NI through the internal link or to the next node through

the physical link. This is the stage with the most straightforward task since it only involves wire connections and no logic. Each lane or link includes two different channels, one incoming an one outgoing. The ultimate goal for each link is to send and receive both data packets and flow control packets. For reasons that are explained later, at the moment each link of each channel is reserved exclusively either for data or flow control packets, depending whether the link is treated as an input or an output.

## 4.2 Routing Table and Configuration

In most router designs, we find a single routing table which the router uses to find the correct output port that each packet should be forwarded through, towards its destination. We quickly realized however, that a single routing table approach could possibly lead to races regarding more than a single read on the same address, which eventually will occur whenever two different packet headers arrive on a node simultaneously (but through different inputs) and have the same destination node ID leading to two memory reads at the same time. This would result to a stalled RC stage for one of the packets which in turn would complicate the pipeline as there was no guarantee that the latency for accessing the routing table would always be the same.
Instead we went for a time efficient but more resource costly approach, by using multiple copies of the same routing table. In fact we need one mirrored routing table for each physical input link. Although resource requirements are certainly higher, this way we avoid dependencies whenever two different inputs attempt to find the output port for the same destination node. These mirrored routing tables have a depth of the number of nodes in the network. This way, the table is indexed in essence by node IDs and the data in each address space correspond to the router's output port through which the next hop will take place towards the packets destination node. The way that the routing table is initialized is during the set-up of the network, when a discovery algorithm is run by the APU, in order to identify nodes at the end of every physical link, of every router. On subsection 7.3.1 we will discuss in what ways the current routing table set-up can be expanded and improved upon.

## 4.3 Scheduling

This section focuses on the scheduling task of the router. Assuming a router of $x$ input lanes (internal or physical) and $y$ virtual channels per lane, this leads to $xy$ streams that the router has to arbitrate between. When only one input requests a specific output, the router should connect the corresponding input with the designated output. When two or more packets from different inputs compete for gaining access to the same output port in the same cycle the router is responsible for resolving the contention. This means that only one packet - and consequently one input - will gain access to the output port. The flits of the packet that the decision did not favor, stay in the input buffer and make a request anew, in the next cycle. To achieve an efficient but still fair arbitration between different inputs, we suggest a two level scheduling algorithm. On the higher level, VCs should be accessed according to a fixed priority while on the lower level, the sequence in which it processes input links depends on a Round-Robin algorithm, which uses a token to discern the prioritized lane from the rest. A high-level flowchart representation of the scheduling algorithm is presented in Figure 4.3. Even though the per-VC and per-lane arbitration seem to operate independently, their eventual outcomes in the SA stage are very much dependent, each one affecting the aggregate matching quality of the crossbar.

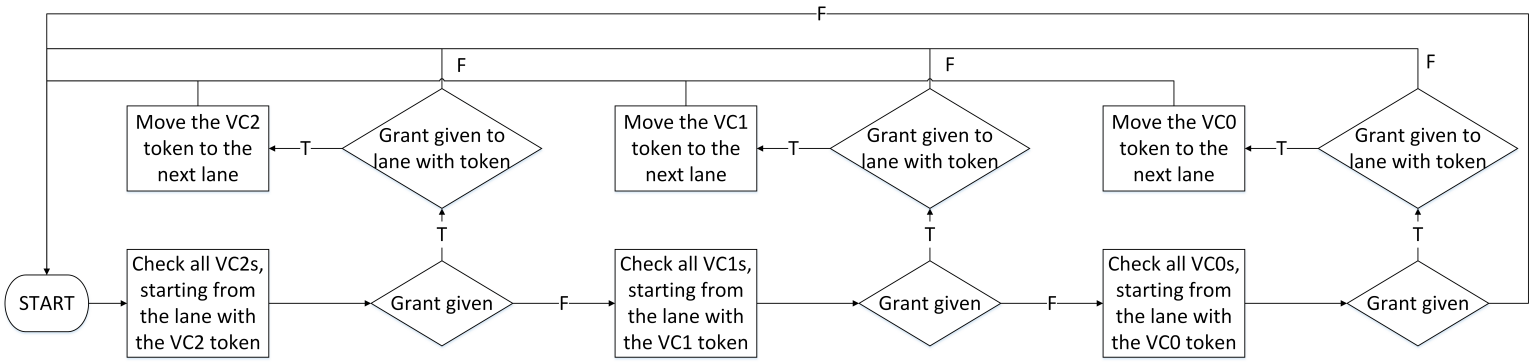



Figure 4.3: The scheduling process between VCs and input lanes.

### 4.3.1 Virtual Channel Scheduling

As presented on section 4.1's Input Buffering stage, there are three distinct VC priorities, $VC_2$ is the high priority virtual channel, $VC_1$ the medium priority VC and $VC_0$ the lowest. Focusing on a single input lane, the switch allocator must always attempt to process packets coming from the highest priority VC available before moving on to a lower one. As long as there are data available on the highest priority VC, the route controller will always forward these packets instead of packets of lower VCs, unless the requested output is already in use or the downstream VC has no space available. After checking all of the $VC_x$ from all the lanes, the controller either starts over from the highest priority VC if a new packet was granted an output, or moves one VC priority down if no new packet was forwarded. The same logic propagates to all VC priorities. For example when checking $VC(x-1)$, if a packet manages to acquire the requested output then the route controller begins checking once again from the highest priority VC whereas if all $VC(x-1)$ packets fail to reach the requested output, the scheduler moves on to checking packets from $VC(x-2)$. If, at the worst case, the route controller fails to route a new packet from all available VCs then it simply forwards the flits from the already routed packets and waits for any kind of change in the network that will allow new packets to be routed on the next cycle.

### 4.3.2 Lane Scheduling

On the lower level of scheduling the controller attempts to forward packets from any lane but from a specific VC priority according to the procedure mentioned on the previous paragraph. The question under discussion on this paragraph is how we could access the input lanes, based on a "fair" approach so that no lane ends up being prioritized when it should not be. With that in mind we proposed a Round Robin algorithm that uses a token to signify the first lane to be accessed. The basic premise behind this approach is that when an input lane has the token, the route controller starts checking for new packets from that lane, and moving on to the next one in numerical order until all the lanes are checked. The token however, will stay on the same lane, until the controller successfully routes a packet from that lane. When it does, the token is moved to the next lane and will stay there until a packet from that lane is granted an output.
We should point out that eventually there are as many tokens as there are VCs. This

can be easily explained, considering that it would be wrong to give Lane B the priority for $VC_1$ packets, just because Lane A just forwarded a $VC_2$ packet and the token moved from Lane A to Lane B for all VCs.
In Figure 4.4 we see a simple scenario of three packets that are all on the SA stage. As thoroughly described above, initially the scheduler checks the highest VC priority on all lanes, for packets that can be granted output. After it successfully routes them in this case its a VC2 packet from Lane #1 it "resets" and checks again from the highest priority for new packets. Only when on that priority there are no more packets present, it moves to lower priorities and once again checks all lanes for said priority. On this example, both input lanes have available packets from VC1 requesting the same output. The arbiter follows the Round-Robin logic, and grants the output request of the lane that has the VC token for that VC priority. Finally on the last clock cycle, the packet that lost all previous contentions, either due to VC priority or due to the Round-Robin algorithm, is now granted the output that it has requested.

## 4.4 Flow Control

Flow control is the mechanism that describes how a packet or flits of packet are transferred between two endpoints, either across NIs (end-to-end flow control) or across routers (link-level flow control) providing lossless operation and high communication throughput. A NoC's flow control is usually implemented through the use of credits or Stop-Go flags [19].
In this work, after taking into account the added overhead of keeping track of credits we decided to go with the latter. Our Stop-Go signals, which are named Xon-Xoff, are sent to every adjacent node whenever a buffer status is changed. Depending mainly on the agreed upon design and a packet's time-of-flight, the router has to notify adjacent nodes about buffer "fullness" whenever a certain threshold is reached on the buffers available slots. This requires to calculate the worst case latency (including router latency and physical link speed) in order to set the buffers empty-to-full threshold correctly. When that threshold is reached a flag is toggled which triggers the creation of a flow control packet (in this case Xoff) to send to the adjacent node. Similarly, when the queue's utilization drops below said threshold, an Xon packet is transmitted informing the adjacent node that it can once again successfully send a new data packet.

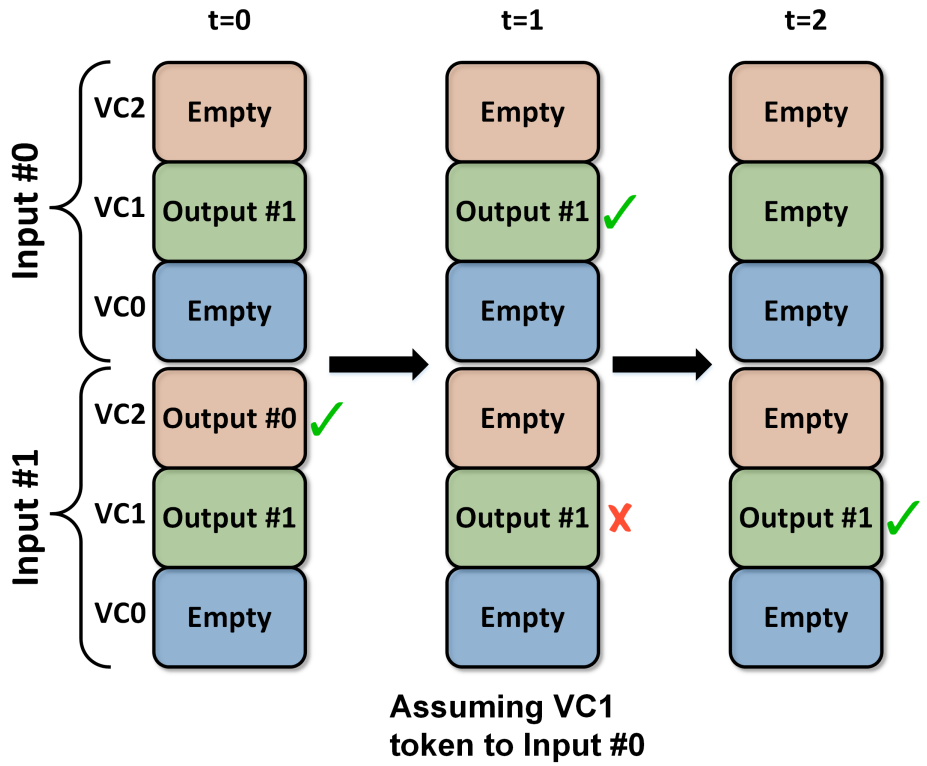



Figure 4.4: A scheduling example of one VC1 packet in one input, and two packets in the second input, a VC2 and a VC1, all requesting outputs on the same clock cycle. For simplicity, all packets are supposed to be single-flit.

# Chapter 5

# System Implementation

This chapter provides all the necessary information of how the suggested architecture was implemented. First off, we present the tools that were used for the technical aspect of our work. Next, we proceed to the detailed description of the modules which includes the tasks performed and how they match their respective pipeline stages, their ports and how they interface with the rest of the design along with their timing performance. Towards the end of the chapter, we incorporate the aspect of the physical connections between the nodes and finally list the resource utilization of our design on the AXIOM board.

## 5.1 Tools Used

The two main tools that were used throughout this work were both Xilinx tools, namely the Vivado High Level Synthesis (HLS) [20] tool and the Vivado Design Suite [21].
High-Level Synthesis (HLS) tools are hailed as one of the most promising ways to bridge the design productivity gap, especially for reconfigurable systems [22]. These tools increase designer productivity at a possible performance and/or silicon cost, although the designer can play a significant role in the minimisation of both. Currently, one of the key challenges for the designer is to efficiently use the vendor-defined methodology and the design guidelines of the HLS tool.
The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that the designer can synthesize into a Xilinx or a Xilinx compatible field programmable gate array (FPGA). C specifications can be written in C, C++, SystemC, or as an Open Computing Language (OpenCL) API C

kernel, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors. Using the Vivado HLS tool we are allowed to develop and verify algorithms at the C-level, control the C synthesis process through optimization directives, create multiple implementations from the C source code using optimizations directives, all while keeping an easy to read and portable C source code ready to be re-targeted into different devices. The way that HLS synthesizes the C code is that it turns the top-level C function as the top RTL module while synthesizing its arguments into RTL I/O ports. The rest of the C functions are synthesized into blocks in the RTL hierarchy mimicking the same hierarchy existent in the C source code. With regard to the optimization directives mentioned above, the tool can follow the designer's instructions on how it should transform the code area the directive refers to. The usual code areas that can be improved upon through the use of directives are

- The interfaces that will be implemented on the arguments of the top-level function (BRAM, AXI4, AXI4-Stream, bus, etc)
- The arrays that the designer can pick how these will be implemented into block rams and registers
- The loops that can be either fully or partially unrolled, pipelined etc. eventually enhancing parallelism

The Vivado Design Suite is designed to improve productivity. This tool suite aims to increase the overall productivity for designing, integrating, and implementing systems using the Xilinx or Xilinx compatible devices. With the Vivado Design Suite, the user can accelerate design implementation with place and route tools that analytically optimize for multiple and concurrent design metrics, such as timing, congestion, total wire length, utilization and power. The Vivado Design Suite provides the designer with design analysis capabilities at each design stage. This allows for design and tool setting modifications earlier in the design processes where they have less overall schedule impact, thus reducing design iterations and accelerating productivity. All of the Vivado Design Suite tools are written with a native tool command language (Tcl) interface. All of the commands and options available in the Vivado Integrated Design Environment (IDE), which is the graphical user interface (GUI) for the Vivado Design Suite, are accessible through Tcl. Our first approach towards these tools was made under the general concept that every

custom IP that was needed would be created with HLS since that provided a number of benefits, with writing in a C-based language being the most significant, and then connect our custom IPs along with the Xilinx provided IPs in the Vivado Design Suite IDE and let the tool map our design on the FPGA. However, on the first steps of the implementation and while encountering numerous difficulties with HLS mainly because of lack of familiarity with the tool, we decided to take a lower level approach and attempted to use VHDL to create the modules. Unfortunately, we once again reached a dead end when we could not tackle problems related with integral parts of the design, with one of the most representative being the stream interface. That brought us back to HLS where in the end we managed to deal with every one of the problems presented by Vivado's HLS tool and successfully synthesized, verified and exported the modules that met resource and latency expectations to the Vivado Design Suite tool to merge with the rest of the IPs provided by Xilinx.

## 5.2 Implementation

The next paragraphs encapsulate the steps we took towards implementing the proposed architecture, using the tools mentioned above. Here the reader can find a detailed description of all the router modules that were created and their functionality, alongside with schematics to help visualize each parts connection with the rest of the architecture.

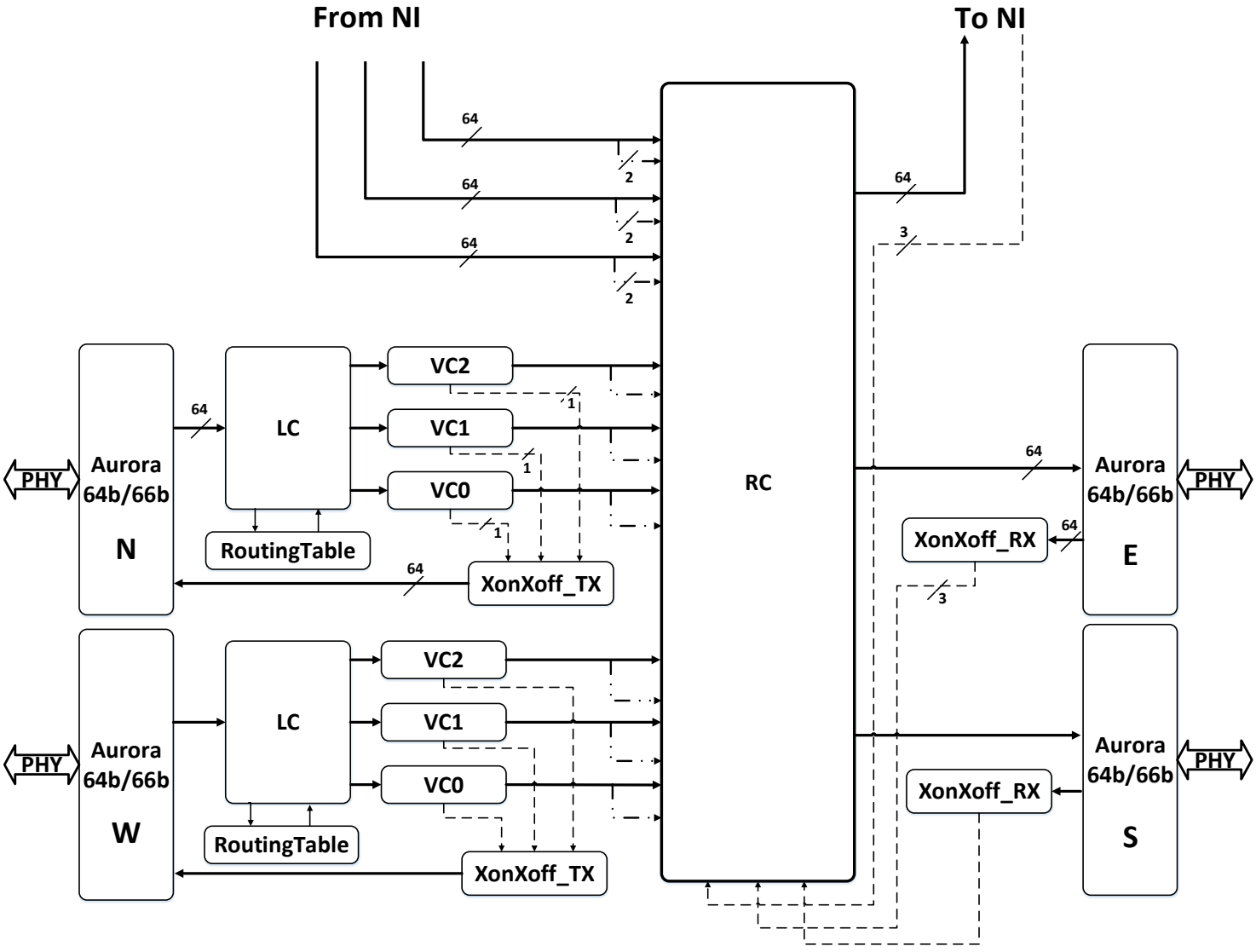



Figure 5.1: Schematic of implemented architecture.

What we managed to achieve is a better organization of the tasks, mainly by taking advantage some of the optimizations the tools used allow, ending up with a different ordering of the pipeline stages which resulted in one stage less than expected, specifically by merging SA and ST [10] and embedding it in the end of IBUFF stage. In Figure 5.2 we can see the final organization of the tasks and in Figure 5.3 a timing perspective of the proposed pipeline architecture. The next paragraphs go into further detail describing our architecture's design.

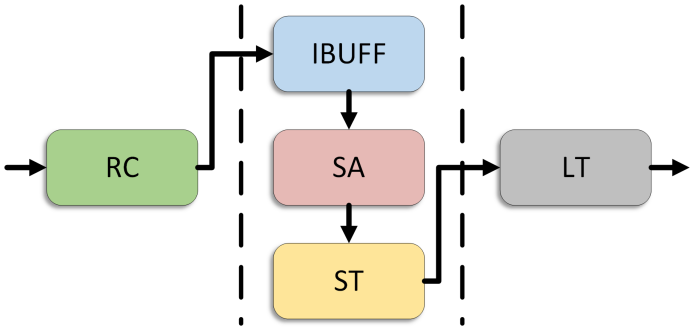



Figure 5.2: The pipelined tasks. From left to right, top to bottom: Route Computation, Input Buffering, Switch Allocation, Switch Traversal, Link Traversal

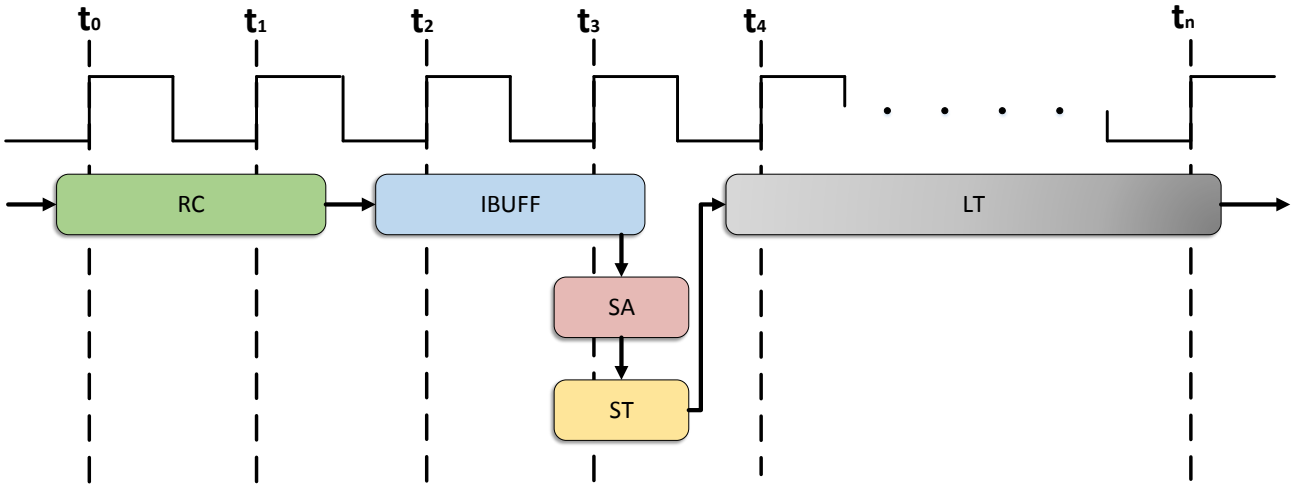



Figure 5.3: A clock cycle perspective of the pipelined tasks.

## Routing Table

The Routing Table, along with the Link Controller are the two modules that carry the task of route computation (section 4.1). During the first steps of our implementation we decided that a BRAM of depth as big as the number of nodes in the network would suffice as a routing table. This would be done in a way that each address space would correspond to the same node ID. The value inside every slot of the routing table would be the router's output port through which the packet will make the next hop towards it's destination. Specifically the value '0' would signify that this node is this packet's destination, so the packet would leave the network and head to the PS through the NI. Every

other value would match a physical output's ID which through the Aurora 64b/66b IP (section 5.2) would lead to a downstream node. Furthermore the BRAM's initialization was already agreed upon to take place through the discovery protocol and before data packets started being injected in the network. The routing tables's task of finding the suitable output port towards each destination node consists of receiving the header's destination field from Link Controller as the BRAM's address of which the data are being read and after 1 clock cycle to respond to Link Controller with the data read, which is the output port found.
The routing table is implemented using a Xilinx provided BRAM IP, which is connected on one side with an AXI BRAM Controller module which facilitates the initialization of the BRAM by the PS and on the other with the Link Controller. The BRAM to BRAM Controller connection happens through an AXI bus interface while the one with the Link Controller uses a BRAM interface, which includes all the necessary signals such as *data_in*, *data_out*, *addr_wr*, *addr_rd*, *wr_en*, *rd_en*, etc..

## Link Controller

This is the module responsible for handling the input streams from the Aurora 64b/66b IP which in essence means finding the output port for each packet received and store it to the appropriate VC afterwards. The way LC (Figure 5.4) is connected with Aurora is through an AXI4-Stream interface with no side signals, which include just the TDATA[(n-1):0], TVALID and TREADY. This means that there is no additional information present regarding whether this is the a packet's header, body or tail flit or where this flit is headed. However this does not constitutes a problem since LC is the IP that is responsible for the route computation as well as the input buffering. The algorithm on which we based these tasks upon is fairly simple. LC starts by receiving a header flit from Aurora 64b/66b. From this first chunk of data, we already have all the information needed about the incoming packet. Depending on the packets TYPE field, it either reads the routing table to find the appropriate router output that the packet must be forwarded through, or simply bypasses the routing table (in the case of RAW_NEIGHBOUR packets, sent during the discovery mode of the networks set up procedure). After finding the appropriate output for the next hop, we now have to find a way to know when we

will treat a flit as a header again. The solution to that is depended on both the current packet's type and its payload. For packet types that have a fixed amount of flits (such as ACK, NACK, INIT, RDMA_READ) we know exactly which flit will be the tail and so we treat every next flit as header. On the other hand, for packet types with variable flit number, we use the packets payload as a counter and decide when we are done processing said packet. To this point, we have added some useful information to each flit which translate as added side signals to the output streams. These are TDEST, containing an output port ID and TLAST, used as a flag to notify the route controller that it is handling a packet's tail flit. The last task the LC has to accomplish is to buffer the packet. As previously described there are three different packet priorities, which translate to three different FIFOs used as virtual channels. To which virtual channel the packet should be buffered is dictated by the header's VC field.
With regard to LC's connections, there are four AXI4-Stream interfaces present, one with the Aurora IP and three with the FIFOs acting as the VCs, and a BRAM interface with the Routing Table. The LC's latency is 2 cc and that is attributed to the fact that it inherits the BRAM's latency when requests the output port from the Routing Table hence making the IBUFF task a 2 clock cycle stage. However, by adding the PIPELINE directive we managed to keep the II to 1, thus pipelining the module completely.

Directives used:

- **#pragma HLS PIPELINE II=1** to set the pipeline's initiation interval of this function to 1.
- **#pragma HLS INTERFACE axis off port=***argument* to implement the AXI-Stream interface. Generates TVALID and TREADY signals automatically, while additional sideband signals such as TLAST and TDEST ought to be explicitly specified by the user, with a struct type. Parameter 'off' specifies a non registered mode.
- **#pragma HLS STREAM variable=***input* **depth=***8* to configure the implementation of FIFOs used internally
- **#pragma HLS RESOURCE variable=***routing_table* **core=***RAM_1P_BRAM* to specify the resource to be used to implement the array in the RTL.

- **#pragma HLS INTERFACE bram depth**=256 **port**=*routing_table* to implement the array arguments as accesses to an external BRAM.

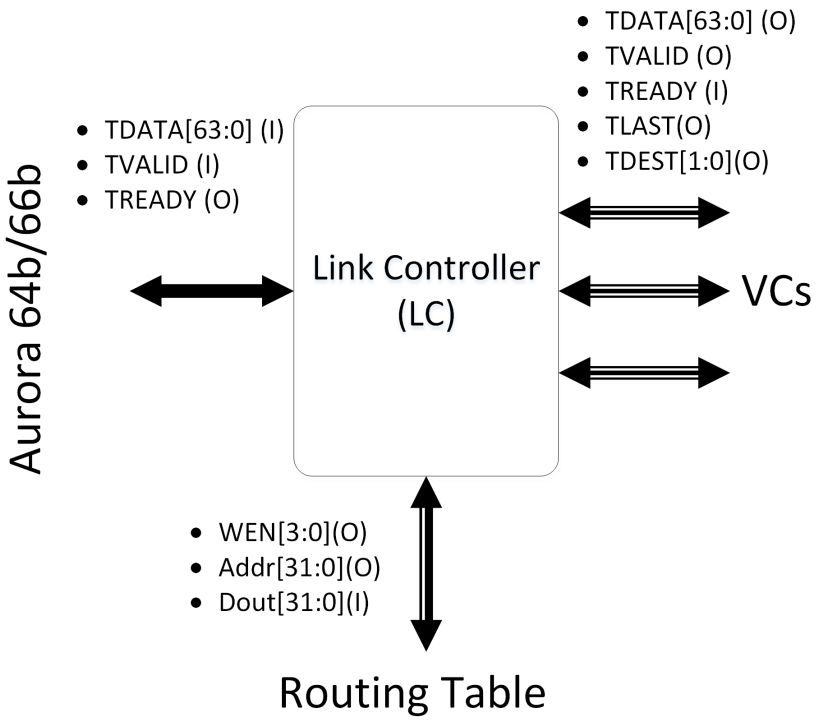



Figure 5.4: Schematic of link controller.

## Virtual Channels

As mentioned above, each input's buffering stage is composed of three FIFOs each treated as a separate virtual channel. These FIFOs were implemented with Vivado's FIFO Generator, which through the IPs configuration menu, allows for a programmable full signal that - as it's name suggests - is asserted when the number of words in the FIFO is greater than or equal to the programmable threshold set by the user. It is deasserted when the number of words in the FIFO is less than the programmable threshold. This signal connects directly to the XonXoff_TX module and its assertion triggers the creation of a flow control packet, described on the next paragraph. Since each VC holds different kinds of packets we tried to trim down the FIFOs' size, but still make them large enough to be

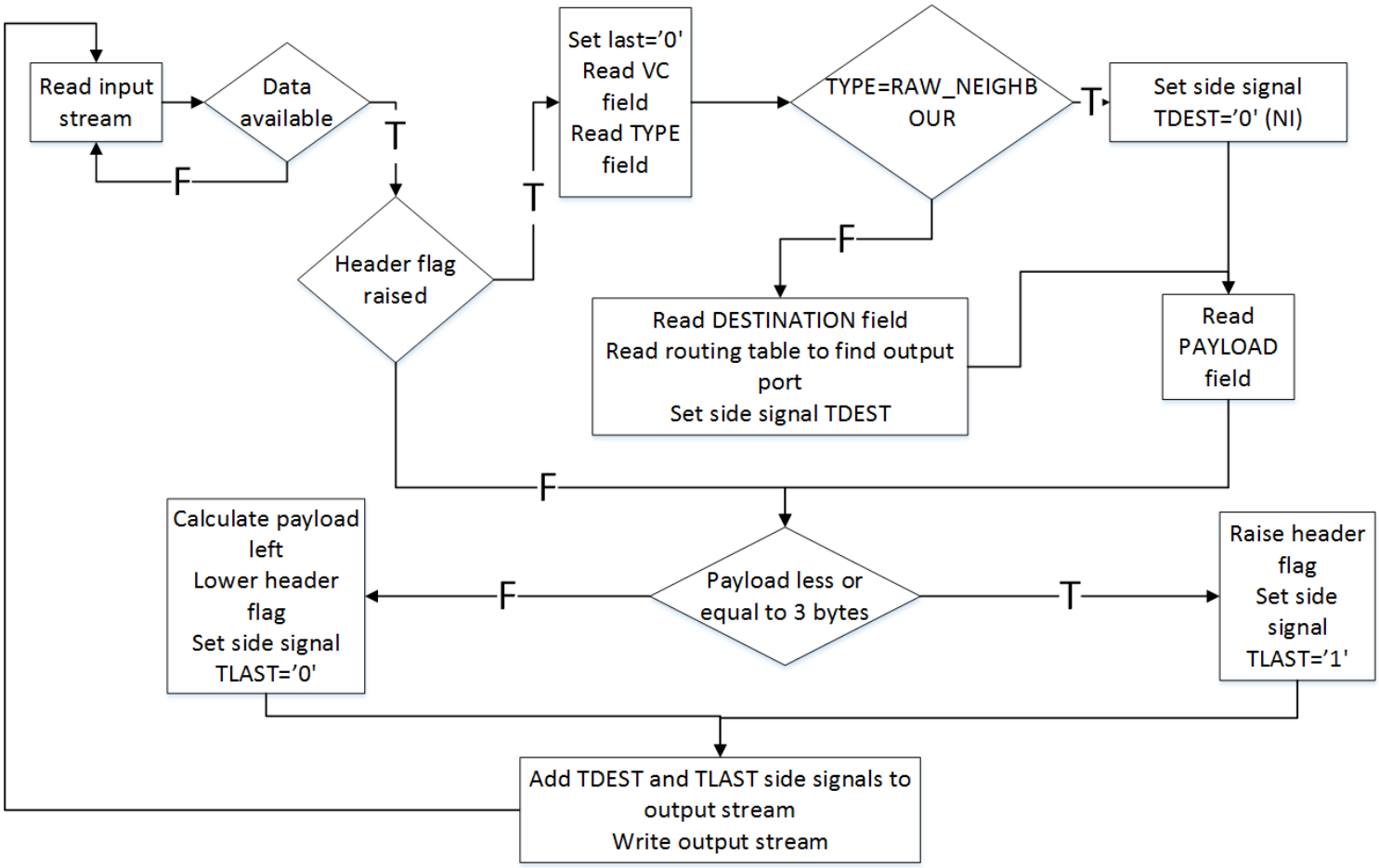



Figure 5.5: Schematic of link controller's FSM.

able to hold two of its worst-case size packets. As a result the final VC setup should be as follows: every FIFO must be an AXI Stream FIFO of 8 byte data width and their respective depth would be 16 slots for VC2, 128 for VC1 and 16 for VC0. For VC2, 16 slots might seem excessive since according to the two worst-case size packets rule, that would mean just 2 slots since in case of ACKs and NACKs, packet equals flit (8 bytes). The problem with that was that even if there was an option for a 2 slot FIFO, Vivado would always implement it as a FIFO with depth of 16. As for VC1 we can see that our rule is in effect. With the LONG_DATA as the worst-case size packet, we are required to have a total of 65 slots for two packets, which forces us to move up to 128 slots. Finally for VC0 which only holds RDMA_READ packets, hence a maximum payload (including header) of 14 bytes per packet which translates to 4 slots for our FIFO we are once again

led to a 16 slot buffer.
During the implementation stage, by using Xilinx's FIFO Generator IP to create the VCs, we took advantage of the built-in FIFOs option. The built-in FIFOs [23] is a memory resource provided by Ultrascale that allows for efficient implementation of FIFOs with the dedicated logic in the block RAM. This way, any need for additional CLB logic for counters, comparators and status flags is eliminated, something that would be necessary with a generic block RAM implementation. However, the built-in FIFO option, automatically restricts us to using a minimum depth of 512, which we accepted as a trade-off, for smaller area utilization. We later on discovered another Xilinx FIFO IP, the AXI4-Stream Data FIFO which allows a minimum depth of 16 slots, however instead of a programmable full signal, it provides a 32-bit counter which holds information about the FIFO's occupancy but by that time we had already created our design based on the single bit full flags, so we stayed with the IPs mentioned above. Though this might not be the most cost-efficient way, we still uphold the rule of storing at worst two of the largest packets of every VC priority in the implemented design.
Concerning the interfaces, since we "propagate" the AXI4-Stream use throughout the architecture, these modules could not be different. This means that both the data input and data output of the FIFO, use the AXI4-Stream interface. The only ap_none interface present is the one that drives the full-flag information from the FIFO's programmable full signal to the XonXoff_TX module's input. The latency of a read operation for the selected FIFO cores implementing the VCs, is 2 clock cycles.

## XonXoff_TX & XonXoff_RX

These are the modules solely responsible for the networks flow control. Each node needs at all times to be able to send information upstream about the local VCs, as well as have access to up-to-date statuses of the downstream VCs.
Updating upstream neighbouring nodes is something that takes place whenever any of the programmable full signals of the local VCs changes. When that happens, XonXoff_TX immediately sends a control packet through the input Aurora 64b/66b's back channel. The useful data are only the lower 3 bits, and each one corresponds to a certain VC for that lane, specifically bit 0 holds VC0's status, bit 1 VC1's and bit 2 VC2's. A value of

'0' means OFF or stop, signifying that the buffer is full, or about to be full, when considering currently mid-flight packets. A value of '1' translates as ON or go, which means that based on the threshold already set, and accounting for in-flight packets, a worst-case packet can be sent from an upstream node and still be successfully buffered. On the other hand, XonXoff_RX is the one receiving the stop-go packets from the adjacent downstream node. Its only input is the receiving channel of the output Aurora 64b/66b from which it keeps only the useful 3 bits and through its connection to the RC, helps the crossbar allocation procedure by informing about downstream buffer availability.
Both of the above modules interface with the Auroras with an AXI4-Stream connection, while in XonXoff_TX the fullness flags are 3 single bit inputs with no interface protocol (ap_none) and in XonXoff_RX the signal that informs the Route Controller about the FIFOs on the next node is a single 3-bit no interface protocol output. The XonXoff_TX module uses two clock cycles to receive and process the data from the VCs and write the XonXoff signal to an "input" Aurora module, while the XonXoff_RX has a single clock cycle latency to receive the XonXoff packet from an "output" Aurora and inform the RC about any status changes.

Directives used:

- **#pragma HLS PIPELINE II=1** to set the pipeline's initiation interval of this function to 1.
- **#pragma HLS INTERFACE axis off port=***argument* to implement the AXI-Stream interface.
- **#pragma HLS STREAM variable=***argument* **depth=***8* to configure the implementation of FIFOs used internally
- **#pragma HLS INTERFACE ap_none** to implement a simple data port with no protocol.

## Route Controller

The main algorithmic part of the router's functionality takes place inside the RC module (Figure 5.7). On each clock cycle, RC performs two distinct tasks, based on information

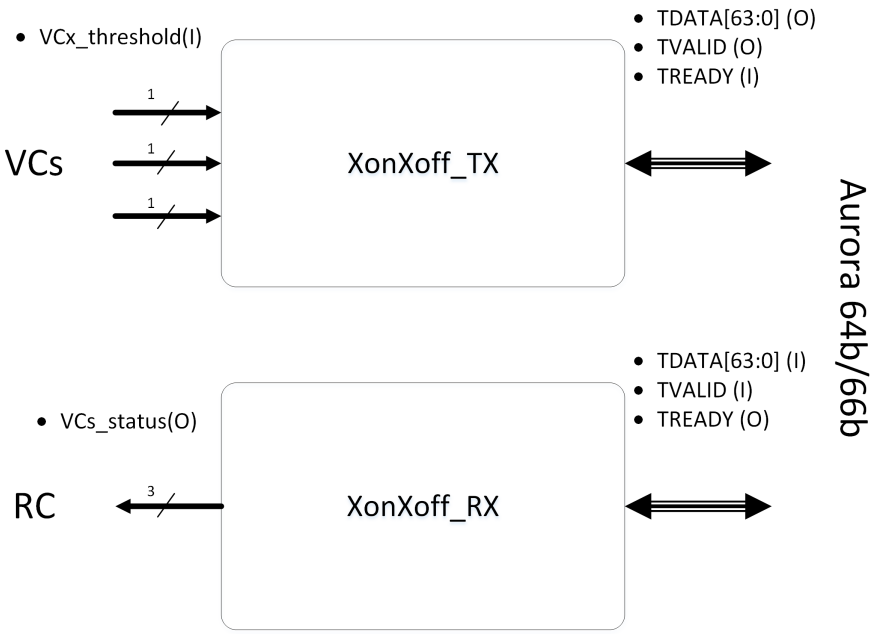



Figure 5.6: Schematic of XonXoff_TX and XonXoff_RX.

held on local registers concerning the matches made between outputs and inputs that are currently active and the downstream buffer space. The first task is that of checking all input streams to find headers of packets that have not yet been routed and are waiting for an output grant. The sequence of checking the streams depends on the VC priority on a first level, and then on the Round-Robin algorithm. Both these ways of scheduling are presented and thoroughly described in section 4.3. Eventually RC matches inputs with unrouted packets to free outputs on an internal memory and moves on to the second part of the algorithm. On the second stage, the controller reads from all input streams that have been granted an output, either on this cycle or on a previous one, and writes the data on the output stream that each input is paired to, according to the matching table. Whenever the TLAST signal is asserted on a stream, RC clears the respective pairing from the table, leaving the output free for the next packet.

The RC connects to the rest of the design with a number of single and multi-bit connections. There are 9 AXI-Stream interface inputs that represent the data streams from each VC (3 lanes each of 3 VCs hence 9 input streams) and 3 AXI-Stream outputs heading either to the NI or to the Aurora IPs representing the router's data outputs. Apart from those, there 9 no protocol inputs, branching off the input streams which hold the

destination information for each packet that has been attached on the stream since the route computation stage, as well as 3 3-bit no protocol inputs, signifying the buffer status of the NI and each of the two adjacent downstream nodes. One of the most important milestones was managing to create the fastest route controller module possible. Our first attempt was not what we expected as we ended up needing more that a hundred clock cycles to check all the inputs for either new packet routing or for forwarding already routed packets and write all these flits at the appropriate outputs. However, after code restructuring and directive application we successfully created a module of zero clock cycles latency (and obviously fully pipelined). This means that the RC acts as an asynchronous IP simply demultiplexing and multiplexing inputs to outputs according to the route decisions that are made on the fly.

Directives used:

- **#pragma HLS PIPELINE II=1** to set the pipeline's initiation interval of this function to 1.
- **#pragma HLS INTERFACE axis off port=***argument* to implement the AXI-Stream interface.
- **#pragma HLS STREAM variable=***input* **depth=***8* to configure the implementation of FIFOs used internally
- **#pragma HLS INTERFACE ap_none** to implement a simple data port with no protocol.
- **#pragma HLS ARRAY_PARTITION variable=***argument* **complete dim=***1* to partition an array into smaller arrays or individual elements. This will result in RTL with multiple small memories or multiple registers instead of one large memory. This effectively increases the amount of read and write ports for the storage resulting to throughput improvement, while requiring more memory instances or registers.

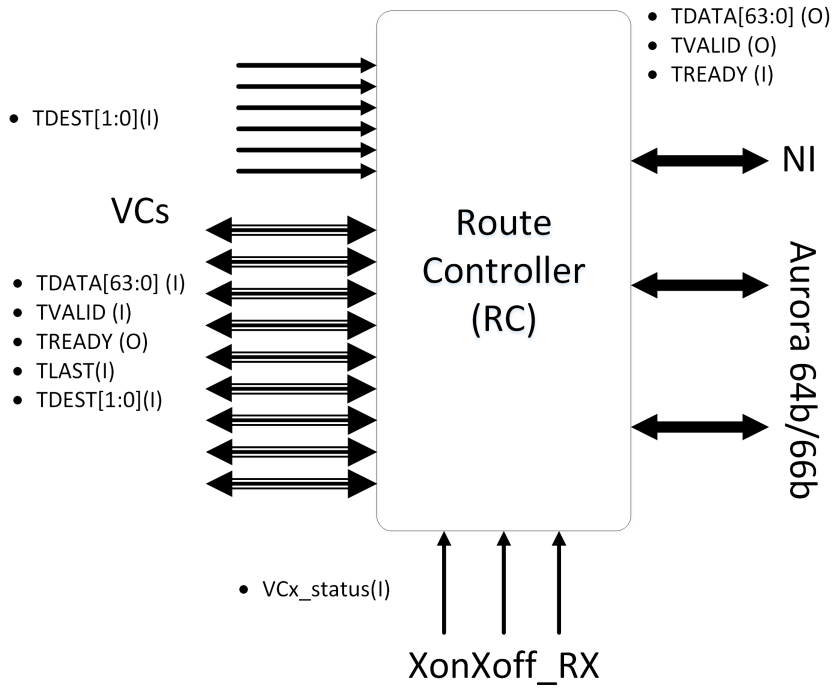



Figure 5.7: Schematic of route controller.

## Interfacing with the physical layer

Towards a seamless connection to the physical links, we decided to incorporate into our design the Xilinx provided Aurora 64B/66B [24] IP cores. Aurora 64B/66B is a lightweight, serial communications protocol for multi-gigabit links, used to transfer data between devices using one or many GTX or GTH [25] transceivers, while supporting the AXI4-Stream user interface, making it an ideal choice for merging our design with the physical level. The core implements the Aurora 64B/66B protocol using the high-speed serial GTX, GTH or GTY transceivers in applicable UltraScale, Zynq-7000, Virtex-7, and Kintex-7 devices. It can use up to 16 consecutive device GTX, GTH or GTY transceivers running at any supported line rate to provide a low-cost, general-purpose, data channel with throughput from 500 Mb/s to over 400 Gb/s. Aurora 64B/66B cores automatically initialize a channel when they are connected to an Aurora 64B/66B channel partner, as demonstrated in the Figure 5.9. In this work, we utilize a single lane on each channel that we set up. After initialization, applications can pass data across the channel as frames or streams of data. Streams are implemented in Aurora 64B/66B as a single, unending

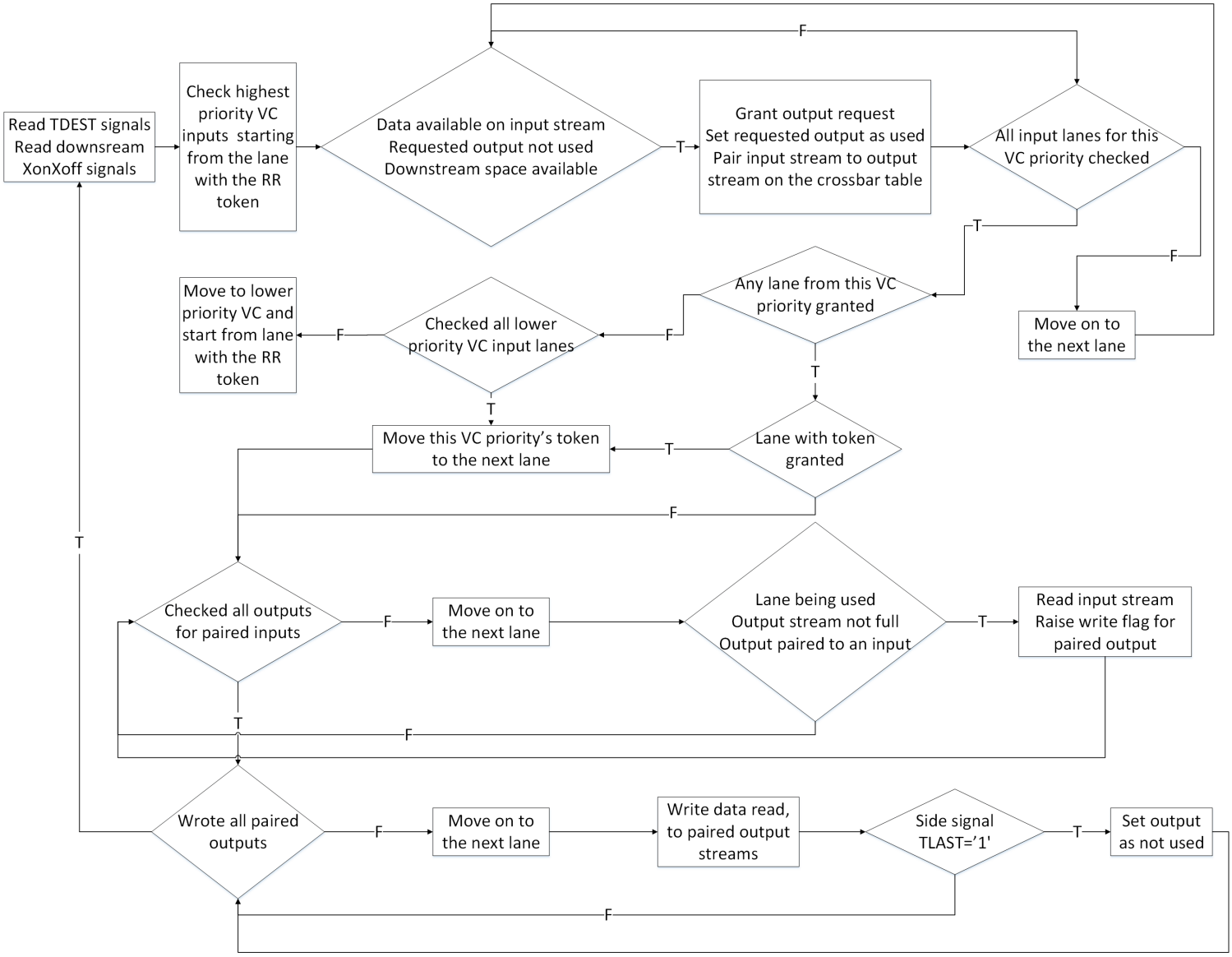



Figure 5.8: Schematic of route controller's FSM.

frame. Whenever data is not being transmitted, idles are transmitted to keep the link alive.

For a default single lane configuration, latency through an Aurora 64B/66B core is caused by pipeline delays through the protocol engine (PE) and through the GTX and GTH transceivers. The PE pipeline delay increases as the AXI4-Stream interface width increases. The transceiver delays are determined by the transceiver features [25]. Table 5.1

shows the maximum latency and the individual latency values of the contributing pipeline components for the default core configuration on UltraScale GTH transceiver based devices.

Table 5.1: Latency for the Default Aurora 64B/66B Core Configuration

| Latency Component | Clock Cycles |
|---|---|
| Logic | 46 |
| Gearbox | 1-2 |
| Clock Compensation | 7 |
| **Maximul Total** | **54-55** |

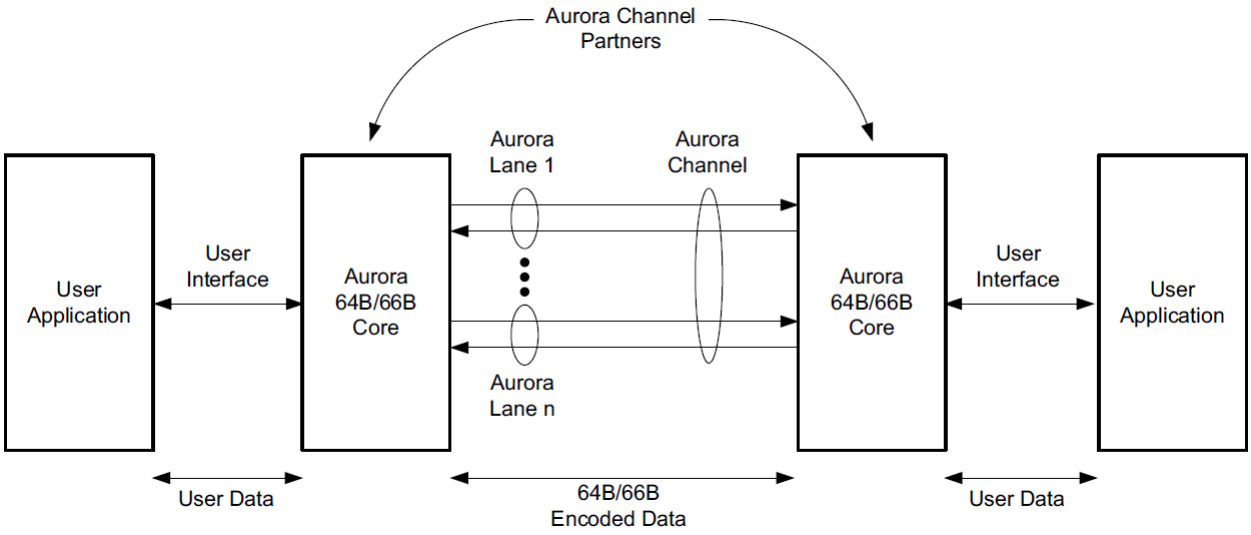



Figure 5.9: Aurora 64B/66B overview.

## 5.3 Board Interconnection

Targeting the highest throughput possible with any of the available connectors on the AXIOM board, we took advantage of the USB type-C [26] connectors using the GTH transceivers. GTH transceivers are power-efficient transceivers, supporting line rates from 500 Mb/s to 16.375 Gb/s. The GTH transceiver is highly configurable and tightly

integrated with the programmable logic resources of the UltraScale architecture used on the AXIOM boards. However, the maximum bandwidth supported by the transceivers does not match that of the USB cables utilizing the GTH transceiver. The USB type-C connectors provide four differential pairs [27] ($TXp_1$, $TXn_1$, $RXp_1$, $RXn_1$, $TXp_2$, $TXn_2$, $RXp_2$, $RXn_2$) of SuperSpeed data bus which in couples ($TX_1$ - $RX_1$, $TX_2$ - $RX_2$) can handle up to 10 Gb/s on full duplex. That can be translated to two channels, each comprised of single bidirectional lane (using a single GTH transceiver) with 10 Gb/s bandwidth for each, which as a result is the maximum throughput we aim for.
On each AXIOM board acting as network node, there are four type - C connectors, allowing for a total of eight high-speed serial communication channels. For our implementation we decided to use just a single channel from each connection thus using one GTH transceiver from each connector. Furthermore, as we mentioned before, we currently use just one lane of the channel for data transfers as the other one is reserved for flow control packets. Hence, any of the connectors can be treated either as an input or as an output for the router but not both. This leaves us with enough connectors to place the board in a network with a ring topology and use just two out of the four connectors, or in one with a 2-D torus topology and use all four USB type-C connectors. As seen in Figure 5.10, two boards are connected in a ring topology, utilizing two physical links on each, one as an input and one as an output.

In Figure 5.11 we can see a set-up of two boards that utilize all four connectors. Unfortunately, due to lack of enough USB type-C cables (and initially lack of AXIOM boards) we decided to set-up two independent rings while using the full 3x3 architecture, which although can not be described as a Torus network, at least we could test that all available lanes can successfully work in parallel.

## 5.4 Implementation Cost

Our goal was to create and implement a design which would be as low-cost as possible, in order to leave most of the available PL space of the board free for application specific accelerators. Our final design accomplishes that by occupying only 1% of the overall resources for logic. Table 5.2 provides the resource utilization of the router implemented on the zczu9eg chip.

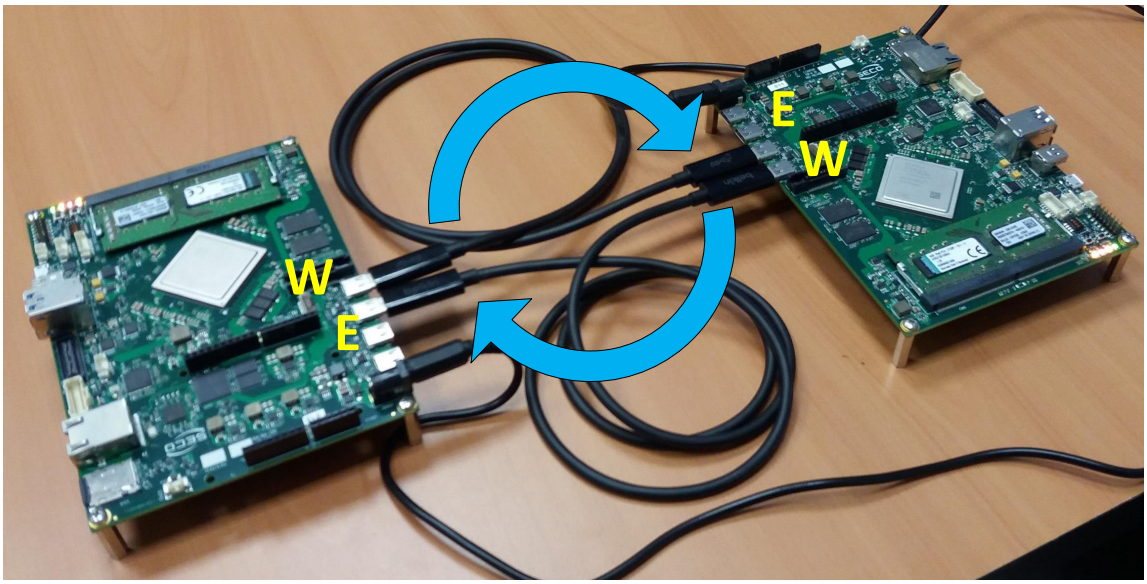



Figure 5.10: Two AXIOM boards connected in a ring topology, using the USB type-C cables to implement the physical links.

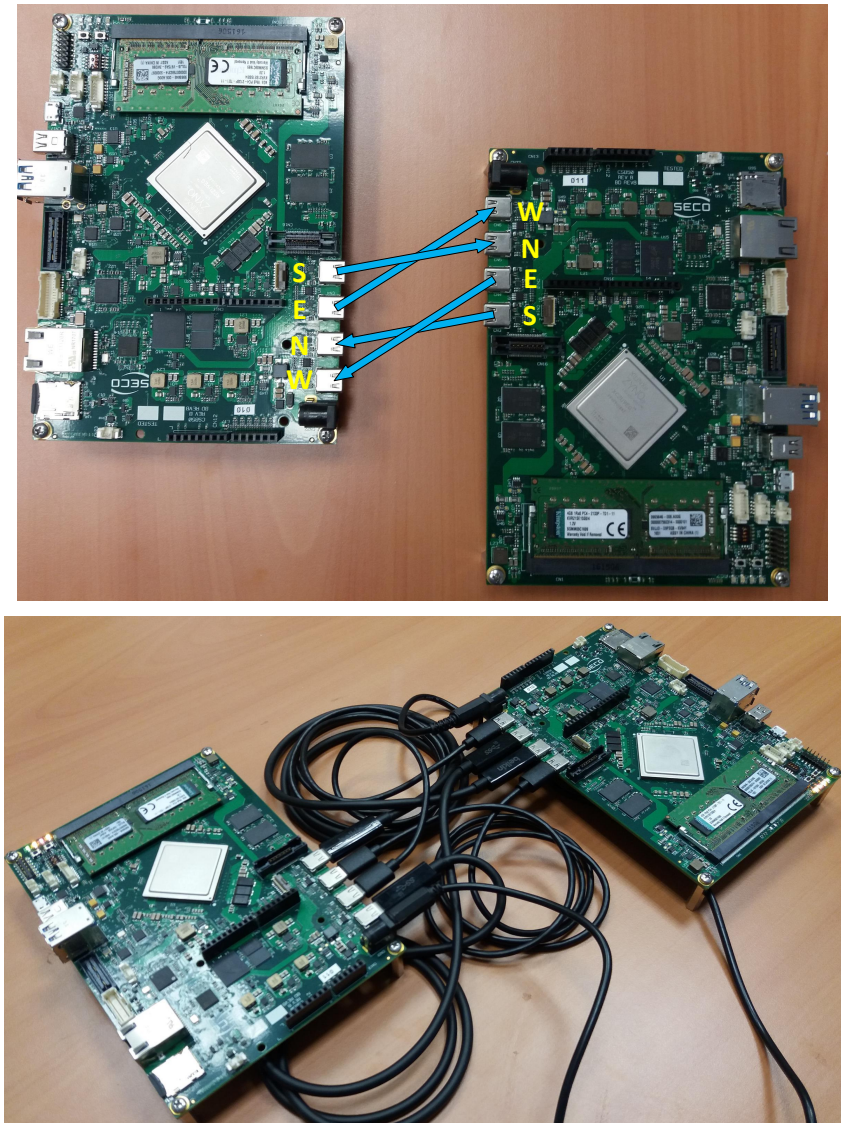



Figure 5.11: Two AXIOM boards programmed using the 3x3 router (for 2-D torus topologies) in 2 independent rings, to utilize all 4 connectors.

Table 5.2: Router resource utilization on the zczu9eg chip. For modules with more than one instance inside the design (x2 etc) the values give the sum of their aggregate resources. To calculate resources for a single instance, divide by the number of instances.

| | CLB | | | BRAM | | GT |
|---|---|---|---|---|---|---|
| | LUT | | FF | BRAM36K/FIFO | | |
| | Logic | LUTRAM | | BRAM36K | FIFO | |
| **Link Controller (x2)** | 196 | | 446 | | | |
| **Routing Table (x2)** | | | | 4 | | |
| **VCs (x6)** | 12 | | 12 | | 6 | |
| **Route Controller** | 1502 | | 38 | | | |
| **XonXoff_TX (x2)** | 36 | | 44 | | | |
| **XonXoff_RX (x2)** | 2 | | 6 | | | |
| **Aurora 64B/66B 0 (x2)** | 794 | 70 | 2824 | 2 | | 2 |
| **Aurora 64B/66B 1 (x2)** | 778 | 68 | 2658 | 2 | | 2 |
| Total | **2569** | **138** | **6009** | **8** | **6** | **4** |
| Available | 274080 | | 548160 | 912 | | 12 |
| Percentage | **0.99%** | | **1.1%** | **1.54%** | | **33.33%** |

# Chapter 6

# Results

After thoroughly describing the architecture and implementation of the router, in this chapter we will present the results of our work and discuss how well it could be integrated on the AXIOM platform (Figure 6.1). Specifically, we will go through the details of the AXIOM board's specifications in order to give an overview of it's available resources, and then move on to a more technical evaluation of the router's capabilities by discussing it's performance.

## 6.1 AXIOM Board Description

During the the course of this thesis, there was a number of boards that were available to us, including the SECO produced AXIOM board. However, before we were able to get a hands on experience with AXIOM, we had to compromise by using other boards to debug and test our designs. These initial boards were the Digilent Zedboard at first and then the Xilinx ZC706. We will not go into more detail about those two, mainly because neither one (and especially the Zedboard) met the requirements for both available transceivers and connectors, not to mention the restricted resources.
AXIOM is the first board that combines three worlds in one: Arduino, ARM computing and FPGA. The board in fact presents the same pinout of Arduino Uno, so to let you attach an Arduino Uno-compatible shield to the board. Also, the presence of Arduino UNO pinout enables fast prototyping and exposes the FPGA I/O with a user friendly interface. The ARM computer on board consists of a 6-core heterogeneous processor - a 64-bit Quad core A53 running at 1.2GHz and a 32-bit Dual core R5 at 500MHz. The FPGA

Figure 6.1: The AXIOM board.

processor is a new generation Zynq Ultrascale Plus (ZU9EG) [28]. The combination of a powerful heterogeneous 6-core ARM processor with a flexible and fast connection unleashes the true potential of the FPGA. For what concerns the RAM, the board features as Processing System a swappable SO-DIMM that goes up to 32GB, and also a soldered 1 GB Programmable Logic. Speaking of storage option, the board has 8 GB of eMMC and a micro SD card reader. However, the board could support up to 32 GB of eMMC. With regard to the board's connectivity, there are four USB Type C ports onboard, and also two USB Type A - not to mention the miniDP connector. Additionally a single channel 24-bit LVDS interface is available, suitable to attach an LCD with touch panel, is provided as an option. Finally the board also features an RJ-45 ethernet connector with a Gigabit ethernet transceiver.

The AXIOM Board is in other words 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 datasets in a short time frame like machine learning, neural networks, server farms, bitcoin miners, and so on. Below we present a list summing up the most significant features of the board.

- 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

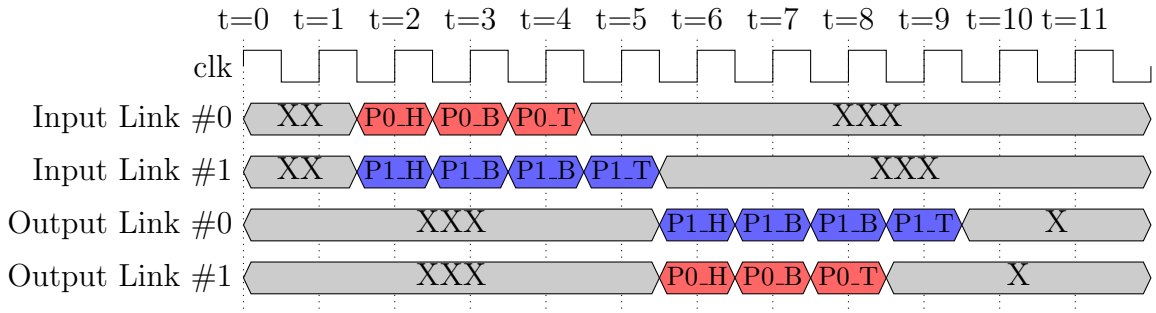
## 6.2 Experimental results on the AXIOM board

In our network interconnect controller we combine the network interface [16], and the router. Although the efficiency of our NIC is evaluated based on both its latency and throughput, if we focus only on the router, there is really no way to judge its efficiency according to throughput. We can treat the router, as the middle man between the NI and the physical links which are the USB type-C cables described in section 5.3. These

two are mainly responsible for the bandwidth achieved and as long as the slowest router pipeline stage still meets the required frequency of 161.13 MHz we can safely state that there is no degradation of the network's throughput that the router is responsible for. Any drop-downs in throughput can be attributed to the DMA engines and the communication between the PS and the various interconnects with the PL. As a result, in this section we chose to provide a number of examples to demonstrate the routers functionality, focusing mainly on the pipeline's efficiency and the routing arbitration.

## Scenario #1: Packets, arriving simultaneously, requesting different output

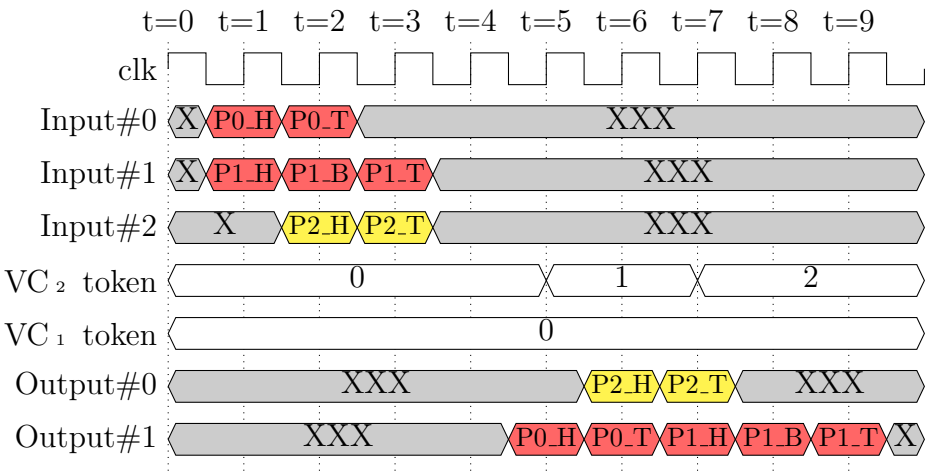
In this scenario, we test the case of two different type packets (but of the same VC priority), arriving on the exact same time, on different input lanes, but requesting different outputs. The point of this test, is to present the ability of the scheduler to handle more than one packets from different inputs, in the same cycle, under the premise that there are no contentions for the output ports that are being requested.




## Scenario #2: Packets, arriving simultaneously, requesting same output

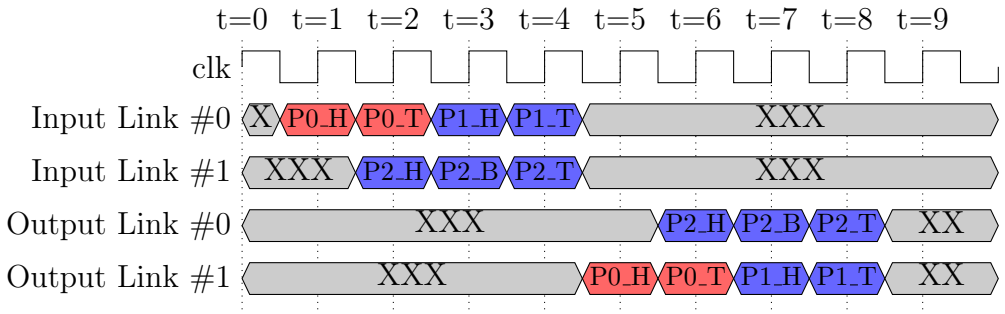
In the following waveform simulation, we tested the case of two packets arriving simultaneously requesting the same output port. As we can see below, one packet is received on the internal link connected to the NI (Input Link #0), while the second is received

through a physical link (Input Link #1). For the shake of simplicity, we assume that both packets are of the same VC priority. Packet #0, consists of two flits, the header (P0H) and the tail flit (P0T). Packet #1 has three flits. The requested output is Output Link #1 for both packets. One cycle after the arrival of the high priority (red-VC2) packets, a lower priority (yellow-VC1) arrives on the other physical link. As we can see and based on the lane scheduling description given in subsection 4.3.2, since the token for the high priority VC is on Lane #0, Packet #0 takes priority over Packet #1, and as soon as Lane #0's packet is granted the requested output, the token moves over to the next lane. On the next cycle, since the scheduler cannot forward P1, drops a priority level down, finding the low priority packet being ready to be routed since it's output request is not conflicting with another. It is successfully forwarded but the lower priority VC token does not change hands since the input processed, was not the input holding the token. One cycle later, Output Link #1 is free again, and P1's output request is granted. After forwarding Packet #1, the token for that priority moves again to the next lane.




## Scenario #3: Packets, arriving back-to-back on same input port, requesting same output port

With this example we wish to present the uninterrupted flow of packets through the router, even when the scheduler is working with a full load of packets arriving back-to-back on some input lanes and random arrivals on others.




## Throughput results based on packet type and payload size

In this section we will present the results of the throughput tests we run, based on a C code we wrote and downloaded to the ARM processor of the Ultrascale+ chip. Before evaluating the results, two things should be noted. Firstly the code written supports only synchronous transfers, which means that the APU has to wait for a transfer to complete before initializing a new one. Secondly, since the router's latency was 4 cc plus the 54-55 cc added from the Aurora 64b/66b cores for the link traversal stage we judged that there would be no actual throughput fluctuations if we calculated the throughput based only on the router's latency.

With that in mind we decided to determine the achieved throughput, accounting for the latency added by the APU, the Interrupt System, the NI and the router. Hence, we refer to the result tables of Amourgianos-Lorentzos' thesis [16] which depict the performance of the Network Interface Controller as a whole.

Table 6.1: RDMA and RAW data transfer throughput

| Packet Type | Payload | Throughput |
|---|---|---|
| LONG_DATA | 15 Bytes | 0.028 Gbps |
| | 255 Bytes | 0.455 Gbps |
| | 4095 Bytes | 3.9 Gbps |
| | 16383 Bytes | 6.13 Gbps |
| | 65535 Bytes | 7.3 Gbps |
| RDMA_WRITE | 15 Bytes | 0.028 Gbps |
| | 255 Bytes | 0.455 Gbps |
| | 4095 Bytes | 3.86 Gbps |
| | 16383 Bytes | 6.08 Gbps |
| | 65535 Bytes | 7.27 Gbps |
| RDMA_READ | 15 Bytes | 0.022 Gbps |
| | 255 Bytes | 0.422 Gbps |
| | 4095 Bytes | 3.7 Gbps |
| | 16383 Bytes | 5.98 Gbps |
| | 65535 Bytes | 7.23 Gbps |
| RAW_DATA | 15 Bytes | 0.06 Gbps |
| | 127 Bytes | 0.076 Gbps |
| | 255 Bytes | 0.085 Gbps |

There is a noticeable discrepancy in the throughput achieved between RAW data transfers and RDMA transfers. This can be easily explained if we keep in mind that with RAW data transfers, the payload needs to be written from the processor to the NI's FIFOs with the AXI Memory Mapped protocol, which is one of the two main reasons of the added timing overhead of the network. Although the RDMA messages can scale up throughput by carrying more payload per transfer, have to deal with the second reason of additional latency which is the latency of the DMA Engine, utilized to fetch the payload from each APU's memory. However, as we can see in the table above, the DMA latency is slowly absorbed as the payload of the transfer increases.
In the following figures (Figure 6.2, Figure 6.3), we have plotted the results from Table 6.1 to get a better visual representation of the throughput's scaling in correlation with the payload.

Figure 6.2: Raw Transfer Throughput

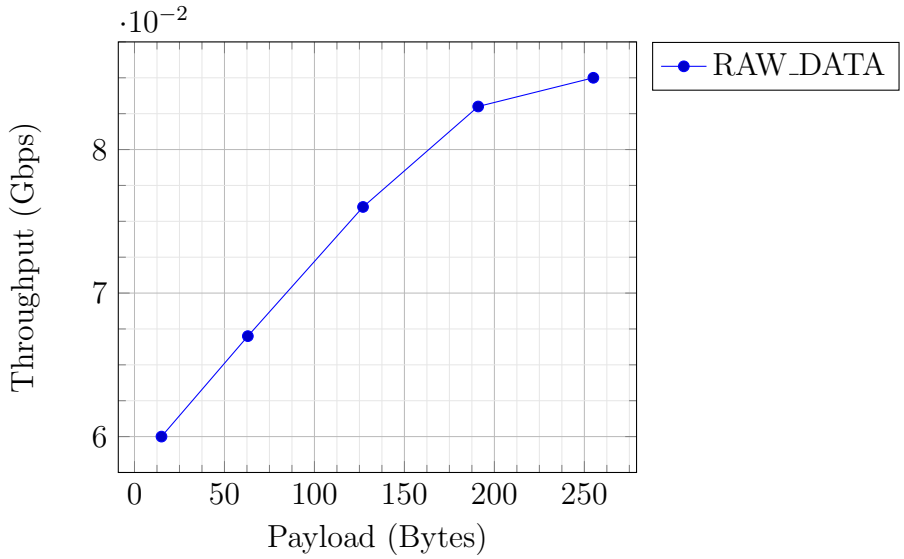
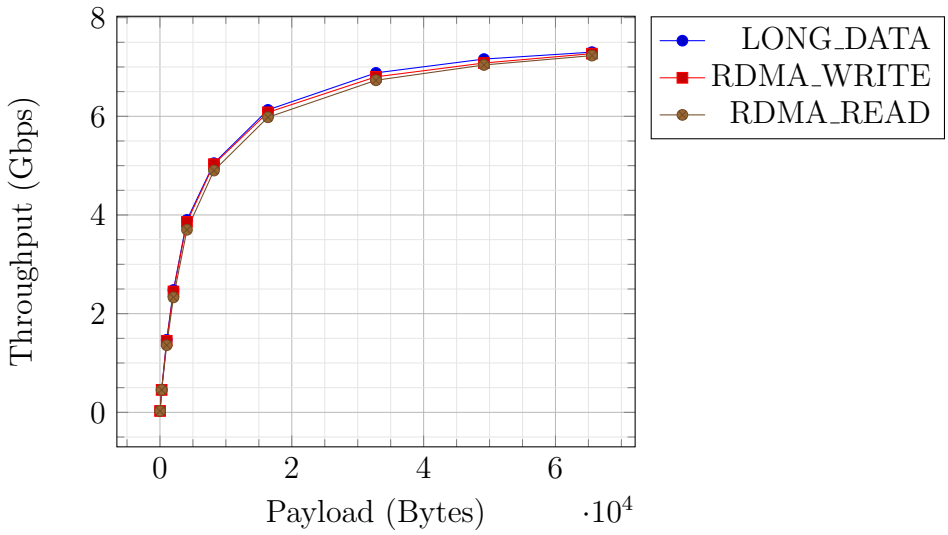



Figure 6.3: RDMA Transfer Throughput



# Chapter 7

# Conclusion and On-going/Future Work

## 7.1 Conclusions

The main goal of this work was to create a router that could interconnect multiple FPGAs into a small cluster and make their communication as fast and as cost efficient as possible. This means that there were certain limitations throughout the course of this thesis that guided the design and the implementation process, mainly with regard to throughput. The target bandwidth was already dictated by the USB type-C cables specifications and set to 10 Gb/s, however the router is not the only component that would affect the efficiency of the final design. In retrospect we realise that evaluating the router just by the network's throughput would may be unfair since there are many variables in play, such as the added latency from the caches and DDR memories on the boards that contain the useful data of the packets. Of course, it is obvious that implementing a more complicated design that would handle traffic in a more intricate way, such as different routing paths depending on network congestion or transmitting different packets interleaved with each other on the same link, would adequately improve the router's performance. Due to this being the first attempt to facilitate the AXIOM interconnect, we decided to follow a simpler approach, and just focus on pipelining the task of buffering and routing the packets with the lowest resource utilization possible for which we consider this work successful.

## 7.2 On-going Work

This section presents certain optimization attempts that kicked off during the closing point of this thesis and have already yielded positive results without being completed yet.

### 7.2.1 Bidirectional Data

Maybe the most important improvement is the one that will allow data packets to flow both ways. This means that instead of a maximum of two input and two output links, there are now four input and four output links or similarly every physical link is both an input and an output resulting in a much more fast and efficient network (Figure 7.1). To do this, we utilize the User Flow Control (UFC) interface on the Aurora IPs. This

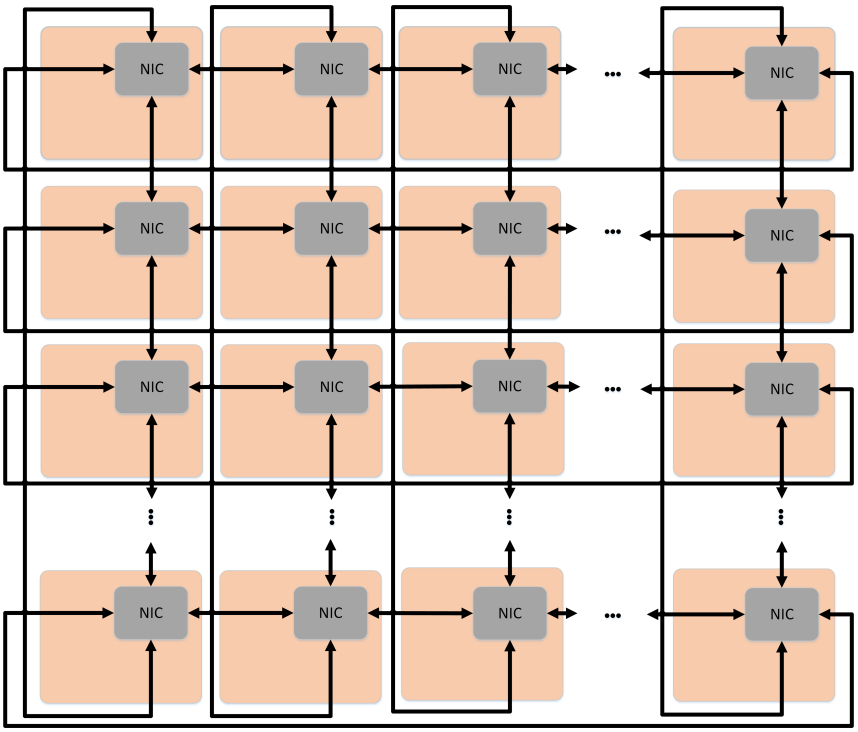



Figure 7.1: Improved AXIOM network 2D-Torus topology.

opens the way for control packet transmission, without the router having to internally arbitrate data and control packets, change priorities, add new packet types or new VC priorities on the design that is already implemented. These control packets are sent as

UFC messages through a separate in-band channel and the Aurora 64b/66b IP treats them by default as higher priority messages than the data messages. How this translates in effect, is that UFC messages do not have to wait for the frame in progress to end, on the contrary, they can interrupt the data message transmission (but without cancelling it altogether) inject the control message inside the stream and then relinquish the channel back to the rest of the data message. It becomes apparent, that now both data channels of the Aurora 64b/66b IP can be used for data packet transmissions, since control packets can be restricted to their own channel. One final obstacle that we need to tackle is that now timing is not being met on the Route Controller module section 5.2 and that is due to the added input and output ports that the scheduler implemented by RC has to arbitrate between.

### 7.2.2 Adding leniency to Scheduling

Currently only one VC priority is allowed to participate from each input in each switch allocation stage, and it is the same priority for all inputs. An improvement to that would be to allow for a looser VC scheduling. What we suggest is that if grants were given to packets from a certain priority but there are still output ports that will stay idle as a result of the first-stage arbitration, the scheduler attempts to grant packets from lower VC priorities, knowing that any higher VC priority packets that could be forwarded, have already been given an output. This is already coded inside the new Route Controller module that implements the bidirectional data flow described above, however due to timing mismatches it has not yet been put to practice.
Below, in Figure 7.2 we give an example of a more lenient towards VC priority scheduler that allows for packets of different VC priorities to be forwarded, given that for lower VC packets to be granted an output, there are no higher VC packets that could have been successfully given an output port.

### 7.2.3 Two GTH transceivers per USB connector

Vasilis Amourgianos-Lorentzos in an attempt to increase the interconnect's available throughput, suggested we took advantage of the second GTH transceiver available on each USB type-C connector. This way, instead of a single TX-RX pair with a bandwidth of 10 Gbps, we could now take advantage the second pin pair and aim for a target

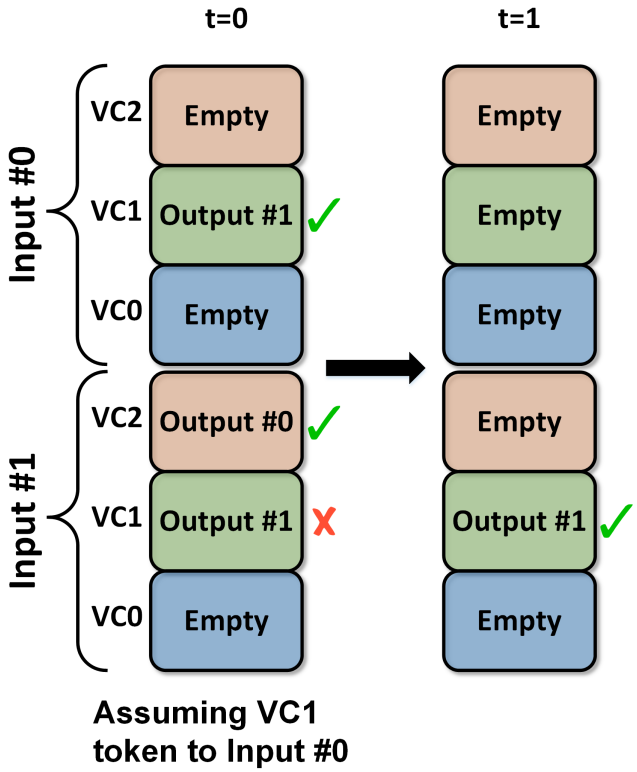



Figure 7.2: An example of a more lenient scheduling. Again, for simplicity, all packets are supposed to be single-flit. Can be compared to Figure 4.4.

throughput of 20 Gbps. This called for some necessary alterations to both the NI's and the router's design, specifically widening the data stream from 64 bits to 128 bits. This was successfully implemented since the Aurora 64b/66b modules are ideal for multi-lane channels and allowed us to easily scale up the data width.

## 7.3 Future Work

Finally, we discuss some ideas that have the potential of improving the existing router design but have not been tested yet.

### 7.3.1 Routing Algorithm

From a more macroscopic view, there are a few upgrades that we could implement to increase the network's throughput. As mentioned in section 3.1 one of the biggest factors affecting a network's performance is its topology. However for different topologies, different routing algorithms can be applied. In this case we suggest a more advanced routing computation method instead of the deterministic one applied. Routing algorithms are categorized to minimal or non-minimal path, adaptive or non-adaptive as well as regular or different paths. By introducing either a minimal path algorithm, or an adaptive one, or even better both, we can achieve a much more efficient way for forwarding packets inside the network. Instead of following a predetermined path at all times when on the other hand the networks traffic is variable, we can now adjust to the networks congestion levels and use a smarter route towards each packets destination node.

### 7.3.2 Adding Time-out Policy

Inside a network, when a sender transmits its data, usually awaits for a form of approval from the receivers side that the transmission went through successfully or in some cases that the transmission is still in process and no errors have occurred. This approval is usually sent from the receiver to the transmitter through the form of an acknowledgement packet (or negative acknowledgement, when informing about errors). For this work, this "handshaking" process, happens at the level of the APU and the network interface. The NI however is not aware about network traffic or the local router's current buffer capacity. This creates the potential for deadlocks, for example in a case where a packet injected in the network from the NI, gets stalled, and eventually blocks another lower priority packet, indefinitely. One way to tackle this problem is if a time-out policy was designed that would give the router authority over packets that are congesting the network and monopolize links. Obviously the router should update the root NI about a cancelled packet so as to keep the lossless attribute of the network, and generally work hand-in-hand with any form of control and arbitration with any layer above it.

### 7.3.3 Implementation Improvements

As already discussed in section 5.2 the FIFOs used to buffer incoming flits should be large enough to hold two of the worst-case size packets depending on which VC priority the packet belongs. FIFOs in Xilinx FPGAs can be generated either as BRAMS or Logic cells. Generally, and unless the FIFOs are very small and narrow, the most efficient way is to use BRAMs. However, BRAM size sets a larger minimum size for the FIFOs than the one we need. Specifically for a data width of 8 bytes, the total size of the BRAMs assuming a minimum depth of 512 slots, would be 12288 bytes for all three VCs, which corresponds to 96 packets of the biggest size.
This can be considered an overkill, since according to our design only a total of six packets can exist inside these three FIFOs at runtime. These packets can fit within a single BRAM of 512 slots depth and 8 bytes width. Noting that we provide sufficient read/write ports and that we arrange head and tail pointers accordingly, we can use just one BRAM sectioned to three parts to implement all the VCs for a single lane, resulting to a 66% reduction in the BRAMs used.

# References


[1] Theodoropoulos, D., Pnevmatikatos, D., Alvarez, C., Ayguade, E., Bueno, J., Filgueras, A., Jimenez-Gonzalez, D., Martorell, X., Navarro, N., Segura, C., et al.: The axiom project (agile, extensible, fast i/o module). In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on, IEEE (2015) 262–269 1, 9

[2] Duato, J., Yalamanchili, S., Ni, L.M.: Interconnection networks: an engineering approach. Morgan Kaufmann (2003) 5

[3] Dally, W.J., Towles, B.P.: Principles and practices of interconnection networks. Elsevier (2004) 5

[4] Benini, L., De Micheli, G.: Networks on chips: A new soc paradigm. computer **35**(1) (2002) 70–78 5

[5] Psarras, A.: High-performance networks-on-chip. PhD thesis 5

[6] Psarras, A., Seitanidis, I., Nicopoulos, C., Dimitrakopoulos, G.: Shortpath: A network-on-chip router with fine-grained pipeline bypassing. IEEE Transactions on Computers **65**(10) (2016) 3136–3147 5, 9

[7] Psarras, A., Moisidis, S., Nicopoulos, C., Dimitrakopoulos, G.: Rapidlink: A network-on-chip architecture with double-data-rate links. In: Electronics, Circuits and Systems (ICECS), 2016 IEEE International Conference on, IEEE (2016) 93–96 6

[8] Mullins, R., West, A., Moore, S.: Low-latency virtual-channel routers for on-chip networks. In: ACM SIGARCH Computer Architecture News. Volume 32., IEEE Computer Society (2004) 188 6

[9] Wang, P., Ma, S., Lu, H., Wang, Z.: A comprehensive comparison between virtual cut-through and wormhole routers for cache coherent network on-chips. IEICE Electronics Express **11**(14) (2014) 20140496–20140496 6, 16

[10] Dimitrakopoulos, G., Kalligeros, E., Galanopoulos, K.: Merged switch allocation and traversal in network-on-chip switches. IEEE Transactions on Computers **62**(10) (2013) 2001–2012 7, 9, 32

[11] Soteriou, V., Ramanujam, R.S., Lin, B., Peh, L.S.: A high-throughput distributed shared-buffer noc router. IEEE Computer Architecture Letters **8**(1) (2009) 21–24 7

[12] Kim, J., Nicopoulos, C., Park, D., Narayanan, V., Yousif, M.S., Das, C.R.: A gracefully degrading and energy-efficient modular router architecture for on-chip networks. ACM SIGARCH Computer Architecture News **34**(2) (2006) 4–15 7

[13] Kumary, A., Kunduz, P., Singhx, A., Pehy, L.S., Jhay, N.: A 4.6 tbits/s 3.6 ghz single-cycle noc router with a novel switch allocator in 65nm cmos. In: Computer Design, 2007. ICCD 2007. 25th International Conference on, IEEE (2007) 63–70 8

[14] Lyberis, S., Kalokerinos, G., Lygerakis, M., Papaefstathiou, V., Tsaliagkos, D., Katevenis, M., Pnevmatikatos, D., Nikolopoulos, D.: Formic: Cost-efficient and scalable prototyping of manycore architectures. In: Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, IEEE (2012) 61–64 8, 9

[15] Katevenis, M.G., Papaefstathiou, V., Kavadias, S., Pnevmatikatos, D., Nikolopoulos, D.S., Silla, F.: Explicit communication and synchronization in sarc. IEEE micro **30**(5) (2010) 30–41 8

[16] Amourgianos-Lorentzos, V.: Efficient network interface design for low cost distributed systems. Diploma thesis, Technical University of Crete, Greece (2017) http://purl.tuc.gr/dl/dias/9F0A3578-6759-426E-A687-0B126EA3F684. 9, 51, 54

[17] Chen, J., Li, C., Gillard, P.: Network-on-chip (noc) topologies and performance: a review. In: Proceedings of the 2011 Newfoundland Electrical and Computer Engineering Conference (NECEC). (2011) 1–6 12

[18] Gebali, F., Elmiligi, H., El-Kharashi, M.W.: Networks-on-chips: theory and practice. CRC press (2011) 21

[19] Flich, J., Bertozzi, D.: Designing network on-chip architectures in the nanoscale era. CRC Press (2010) 27

[20] Xilinx: Vivado Design Suite User Guide: High-Level Synthesis. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug902-vivado-high-level-synthesis.pdf. 29

[21] Xilinx: Vivado Design Suite User Guide: Getting Started. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug910-vivado-getting-started.pdf. 29

[22] Georgopoulos, K., Chrysos, G., Malakonakis, P., Nikitakis, A., Tampouratzis, N., Dollas, A., Pnevmatikatos, D., Papaefstathiou, Y.: An evaluation of vivado hls for efficient system design. In: ELMAR, 2016 International Symposium, IEEE (2016) 195–199 29

[23] Xilinx: UltraScale Architecture Memory Resources. Xilinx. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf. 38

[24] Xilinx: Aurora 64B/66B LogiCORE IP Product Guide. Xilinx. https://www.xilinx.com/support/documentation/ip_documentation/aurora_64b66b/v10_0/pg074-aurora-64b66b.pdf. 42

[25] Xilinx: UltraScale Architecture GTH Transceivers. Xilinx. https://www.xilinx.com/support/documentation/user_guides/ug576-ultrascale-gth-transceivers.pdf. 42, 43

[26] Wikipedia: USB Type-C (2017) https://en.wikipedia.org/wiki/USB-C#Specifications. 44

[27] Wikipedia: Differential signaling (2017) https://en.wikipedia.org/wiki/Differential_signaling. 45

[28] Xilinx: UltraScale Architecture and Product Data Sheet: Overview. Xilinx. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. 50