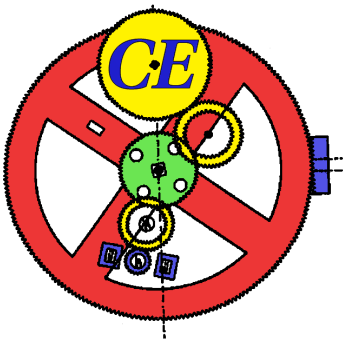


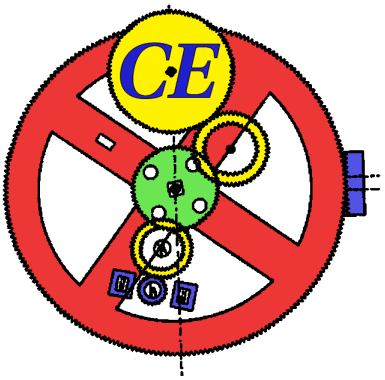
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

An Off-Chip Bridge for On-Chip Network-Based Systems Supporting Traffic Quality of Service

Matías Escudero Martínez



CE-MS-2010-36

Abstract

Prototyping Systems on Chip (SoC) on FPGA technology improves the time that the designer needs to spend in the verification stage when developing new systems or upgrading the existing ones. However, FPGA prototyping is very challenging due to the few resources available in this chips, and often the large designs do not fit into one single FPGA. Since Networks on Chip (NoC) are used as SoC interconnects, extending them for offering communication services between two different chips will solve the problem. This work explores how a bridge can extend the NoC while providing Quality of Service (QoS) to the applications. After a detailed investigation of the NoC protocol stack, we found that the best layer for placing the bridge was at the transport layer. Here, the bridge multiplexes several NoC connections on an off-chip link, with data granularity of words. The bridge offers QoS by means of TDM scheduling and a credit-based mechanism preserves the end-to-end flow-control offered by the NoC. Software and hardware implementations of this bridge are tested and verified showing that they are fully functional. Moreover, two different NoC configuration procedures are developed, so the SoC resources can be monitored from an external host. Finally, this work concludes that it is possible to extend a NoC over more than one single chip without loosing the QoS and hiding the complexity of the off-chip link to the applications.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

An Off-Chip Bridge for On-Chip Network-Based Systems Supporting Traffic Quality of Service

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Matías Escudero Martínez
born in Vigo, Spain

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

An Off-Chip Bridge for On-Chip Network-Based Systems Supporting Traffic Quality of Service

by Matías Escudero Martínez

Abstract

Prototyping Systems on Chip (SoC) on FPGA technology improves the time that the designer needs to spend in the verification stage when developing new systems or upgrading the existing ones. However, FPGA prototyping is very challenging due to the few resources available in this chips, and often the large designs do not fit into one single FPGA. Since Networks on Chip (NoC) are used as SoC interconnects, extending them for offering communication services between two different chips will solve the problem. This work explores how a bridge can extend the NoC while providing Quality of Service (QoS) to the applications. After a detailed investigation of the NoC protocol stack, we found that the best layer for placing the bridge was at the transport layer. Here, the bridge multiplexes several NoC connections on an off-chip link, with data granularity of words. The bridge offers QoS by means of TDM scheduling and a credit-based mechanism preserves the end-to-end flow-control offered by the NoC. Software and hardware implementations of this bridge are tested and verified showing that they are fully functional. Moreover, two different NoC configuration procedures are developed, so the SoC resources can be monitored from an external host. Finally, this work concludes that it is possible to extend a NoC over more than one single chip without loosing the QoS and hiding the complexity of the off-chip link to the applications.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-36

Committee Members :

Advisor:	Kees Goossens, ES, TU Eindhoven
Chairperson:	Sorin Cotofana, CE, TU Delft
Member:	Arjan van Gemund, ST, TU Delft
Member:	Georgi Kuzmanov, CE, TU Delft

To my beloved parents

Contents

List of Figures

List of Tables

Acknowledgments

This thesis concludes my research on the field of Networks on Chip. Before ending this chapter of my life, I would like to thank the following people, who helped me in this throughout this project.

At first, I would like to thank Kees Gossens for allowing me to do this project and giving me advise and directions during the research. Thanks to Ashkan Beyranvand Nejad for his daily assistance and for being always willing to discuss all the problems we had to face during this days. Many thanks to Radu Stefan and Andrew Nelson, who gave me some useful tips and ideas for the design part of the project.

I would also like to thank Arjan van Gemund, who not only offered himself for being in the committee, but also gave me the chance to work with him and, more important, to learn from him.

Special thanks to all my friends, the ones I met last years in Delft and the ones who are in Spain, for all the fun during the hours I was not working in the lab or studying.

Last but not least, I would like to thank my parents for they love and support through all my life. I could not be where I am today without them.

Matías Escudero Martínez
Delft, The Netherlands
December 15, 2010

1 Introduction

Embedded systems are all around us. They are the brains of all the intelligent devices that help us in our every day life. Televisions, set-top boxes, cell phones, digital cameras, car navigation systems, electronic driving assistance [30]. Consumer applications demand more and more functionalities and processing power. For example, smartphones offer nowadays a wide range of very features, such as Wi-Fi, Internet, touch-screens, high definition video playing, recording and encoding, gaming, GPS, USB connections and flash card readers. Embedded systems offer all these features on a single Systems on Chip (SoC), consuming less power, important for the portable devices, at a unit price of a few US dollars.

The complexity of these SoCs grows due to the increasing number of heterogeneous and independent *applications* integrated on a single chip. Applications run over one or more processors, such as General Purpose Processors (GPPs), Digital Signal Processors (DSPs), or Application Specific Processors (ASPs) [13]. Applications might share also resources, such as memories or other peripherals. Generically we will call these processors and peripherals *Intellectual Properties* (IP). Following Moore's Law, the number transistors doubles roughly each two years and due to the market pressure SoCs integrate an ever-higher number of IPs. However, hardware design is difficult and the design productivity is highly based on IP reuse [19]. Therefore, integrating an always-growing number of many applications and IPs in one SoC requires an scalable interconnect [9].

In the last years there have been large efforts in *Networks on Chip* (NoC) field. NoC paradigm provides the scalable and modular interconnect that can easily provide a communication infrastructure for large-scale designs [6][2]. By implementing *Quality of Service* (QoS) mechanisms the NoC is able to give reliable communication services with guaranteed throughput and bounded latency. Such NoCs suitable to be generated automatically are helping the system designer to create interconnects that meet the specific application requirements [11].

Field Programmable Gate Arrays (FPGA) prototyping is a technique that allows faster and cheaper validation of SoCs designs [24]. The RTL description of the hardware is synthetised and mapped onto the specific logic gates of the FPGA. The result is a cycle-accurate system that can be used for SoC validation. However, NoCs occupy a large portion of the FPGA resources. The reason for this is that signal routing and floorplanning is very complex in FPGAs, and also responsible of the main area cost.

By using more than one FPGA it is possible overcome the lack of logic resources for emulating large SoCs [37]. Hence, a system must be partitioned into a number of sub-systems, each of them implemented on one FPGA. Off-chip communication between these FPGAs is thus required.

Prototype boards typically include only one FPGA chip. Although multi-FPGA boards exist, to interconnect sub-systems implemented on each FPGA an interconnec-

tion scheme is required. Furthermore, they can not be upgraded with more chips if a future SoC design requires more logic resources. A multi-board approach can be more challenging, but adding more logic resources is easier and thus the resulting system more flexible. Almost all prototype boards include several communication standards, such as Ethernet, USB or PCI-Express, that can be used for FPGAs interconnection. Finally, multi-board solution is more generic since it can be also used with multi-FPGA boards.

Companion chips allows to optionally extend the functionality of a SoC [29]. This might be interesting for chip makers, who can offer a SoC offering basic functionalities for the low-end market, and the same SoC plus the companion chip offering extra features for the high-end devices. This increases the sales of the basic SoC, lowering its price per unit, and reduces the design time of the high-end systems, because only the new functionalities need to be designed. If the SoC uses a NoC, extending the interconnect to the companion chip might reduce the design and verification time by hiding the off-chip link complexity from the IPs and applications.

In this thesis we explore the potential of extending a NoC with off-chip communication capabilities. Our aim is to give an unified view of the NoC to the applications, independently of the FPGA on which they execute. The resulting system, two NoCs connected by a bridge, should provide the same functionality as a single NoC, while guaranteeing a minimum throughput and bounded latency. Applications show the same functional behavior when running on a cluster of sub-systems compared with the original system. Such an off-chip bridge must keep the QoS of the application in terms of throughput and latency requirements.

Furthermore, run-time configuration of the on-chip interconnect for different usecases is also essential [15]. The configuration is done by a *host*, typically a microcontroller or processor. The host is responsible of the carrying out all the configuration procedures and also might work as system monitor. An off-chip bridge will allow to move the host to a Personal Computer (PC) or any other off-chip processor. Thus, configuration can be performed remotely, and the system be accessed from the outside of the chip, simplifying the system debug and verification.

Therefore, the problem we want to solve is:

> *Design, implement and test an off-chip bridge that enables the extension of a NoC among two or more chips, while providing the applications QoS, and allowing the access from an external host such as a PC to the SoC or sub-SoCs.*

This thesis is organized as follows: In Chapter 2 some other research efforts involving the interconnection of separate chips are reviewed. Chapter 3 describes the Æthereal NoC in depth and introduces the concepts needed for understand the bridging requirements and schemes discussed in Chapter 4. Chapter 5 presents the design of the bridge, describing the mechanisms and techniques used to achieve the requirements. The implementation in hardware and software is shown in Chapter 6, together with the configuration process of the NoCs. The bridge is test and verified in Chapter 7. Finally, in Chapter 8 the conclusions of this thesis are given, including recommendations for future research.

2 Related Work

Significant research efforts have been done in different fields that needs to partition or extend a NoC among several chips. Therefore off-chips links are needed in order to implement a connection between NoCs or with other devices.

A design for off-chip NoC interfaces is proposed in [1]. This work is adapted to Globally Asynchronous Locally Synchronous (GALS) architecture. The interface is placed at the physical layer in order to offer a unified view of the NoC protocol to different NoC-based subsystems. Such interface uses one parallel connection of 78 signals between chips per each bidirectional link.

The concept of sub-NoCs and the way of interconnecting them is discussed in [8]. Such interconnections are placed at the network level but maintaining guaranteed service, which introduce some additional issues. They give two possible solution for the flow control: global end-to- end flow control and local end-to-end flow control. In the first approach flow control is at the level of the whole NoC and local flow control is at the sub-NoC level. Furthermore this paper also shows the architecture of the TDMA synchronizer, that is in charge of adapt the TDMA slot tables between two sub-NoCs keeping the guaranteed service.

Some other work has been carried out in the field of emulating one ASIC with several FPGA's with developing purposes. The work presented in [23] shows how multi processor SoC (MPSOC) with 48 cores can be fitted in 4 FPGA's by extending the NoC (Arteris) with off chip synchronous links. A multi purpose emulation platform which can be used with different NoC topologies is presented in [22]. Network links between routers placed in different chips are emulated by using high speed serial links as inter chip or inter board connections. Serial links are chosen due to their great degree of scalability and the simpler configuration and setup. The drawback of this configuration is that the off chip link introduces 40 cycles latency. Therefore according with results of this paper the latency between two IP is two or three times bigger if there is an off chip link in NoC.

NoCs with off chip links may extend the functionality of the chip. In [29] the NoC has one off chip link for increasing the number of tasks that the SoC can perform by connecting with a companion chip. Therefore the resulting system SoC is more flexible and easily upgradeable. This technique also allows to sell the common IPs to other customers and encapsulating the differentiating functionality in the companion chip. The new chip might be a FPGA instead of an ASIC since the production may be lower. The interconnection link of [29] (called HSEL) is PCI Express or a proprietary link interconnect technology. HSEL is directly connected to two master and two slave NI ports, at the transport level. As a result the system is more flexible since new functionality can be spliced in at any step of the existing process. However, there is an increasing of the power consumption, area and latency.

Bridges can be added to the NoC not only for interconnecting NoCs, but also for

connecting other different devices. In [21] a bridge is developed to interface a multi-core SoC, built using NoC paradigm, to the Internet. The bridge operates above the network layer in the NoC side and at the application layer at the Internet side. Otherwise, if the bridge is operating below the application layer at the Internet side all the IPs must implement the TCP/IP stacks, what may waste more area. The delay that such bridge introduces is in the range of hundreds of microseconds. This might seem very high values for the latency in a chip, but taking into account that this bridge will send data through the Internet it is very likely that the latency introduced by Internet may be even higher.

Multi-processor systems can also take advantage of the intrinsic multi hop nature of the NoC, that can allow inter-chip and inter-board connection in a transparent way for the IPs [38][25][4]. In [38] the design and implementation of a inter-chip interconnect for 4 DSP's per chip is presented. Inter-chip interconnect module is responsible for the conversion and transmission of data between multiple SoCs using PCI Express as off chip connection. The inter-chip module is made up of two components: QPB, the asynchronous bridge module and the PCI-Express interconnect module. QPB implements the conversion between NoC packets and the PCIExpress packets. Therefore the bridge is located at the data link level. Since there are different clock frequencies in the intra-chip connections than in the inter-chip connections, the QPB is also in charge of accommodating the data rates using an asynchronous FIFO. The results show that this solution has a big throughput (around 1.5 Gbps) and the latency is around 22 PCI-Express clock cycles.

An architecture for scalable computing machine built using FPGA nodes is proposed in [25]. Such architecture allows to implement large scale computing applications using a heterogeneous combination of hardware accelerators and embedded microprocessors spread across many FPGAs. All the computing units are interconnected by a flexible communication network. This architecture does not use a NoC for intra-FPGA communication but direct links. Serial multi-gigabit transreceivers implement inter-FPGA communication. This links can eventually emulate 10 Gigabit Ethernet or Infiband protocols in order to create a inter-cluster communication. This architecture also implements an off chip communication controller (OCCC) [5] for providing reliability between tasks. This high-speed communication architecture is implemented over SERDES and it achieves a throughput between 0.3 and 2 Gbps and a latency between 1 and 12 microseconds, depending on the packet size.

3 Æthereal

This chapter gives an overview of the NoC to which the bridging scheme is applied. First, Section 3.1 introduces Æthereal NoC [12] and presents its main features. Section 3.2 describes some of the interesting modules of the Æthereal that are important to understand the bridging scheme selection and bridge operation. In Section 3.3 the protocol stack is discussed together with the functionalities provided by each layer. Finally, Section 3.4 shows the procedure for configuring the NoC, and enabling application communication.

3.1 Æthereal Overview

Æthereal NoC is an interconnection network that allows to combine a large number of Intellectual Properties (IP) into a working SoC improving the scalability and the resource sharing. This chapter will introduce Æthereal architecture and operation as well as its protocol stack [14].

Due to the realtime communication requirements that many IPs might need, Æthereal provides guaranteed services (GS) like uncorrupted, lossless communication, ordered data delivery, guaranteed minimum throughput and bounded latency. Best Effort (BE) traffic is also possible. BE traffic still provides lossless communication and in order delivery, but using the available throughput and latency left by the GS. These features are essential for the construction of robust SoCs. In addition, Æthereal also decouples the different behaviors of the IPs even while sharing communication resources. Hence, the SoC developers can test their IPs independently of each other.

In order to provide such decoupling between IPs, Æthereal chooses contention-free routing, or pipelined Time-Division-Multiplexed (TDM) circuit switching. Thus, GS communication requires resource reservation. A connection must be opened before the IP can transmit data, and it must be closed to release the resources. This configuration process is done by the host, typically a microcontroller placed on the SoC.

3.2 Architecture

Æthereal blocks are shown in Figure 3.1. When IP_1 wants to fetch some data from the main memory a bus transaction i.e. a read transaction for instance is initiated in the IP data port. In this case IP_1 is using distributed memory, so it is connected to a target *bus* that decodes the address and forwards the message to the appropriate initiator port. At this point, the read request message is serialized by the target *shell*. The elements of the request, such as address, command or burst size will be placed into individual *words* of streaming data. This words are 37 bits wide.

The streaming data is fed into the input queue of the Network Interface (NI). There the streaming words are packetised and sent to the router as flow control digits (*flits*).

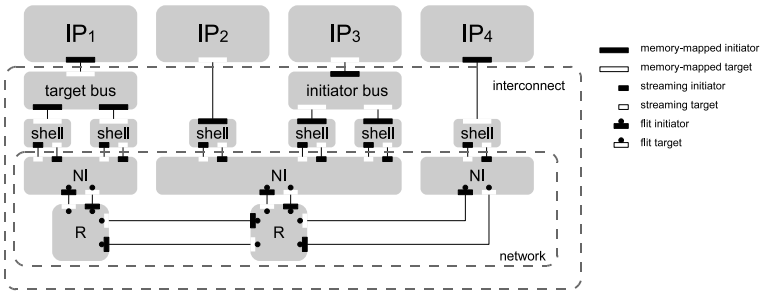


Figure 3.1: An example of a SoC that uses Æthereal Network on Chip as interconnect [16].

Thus, each packet may have one or more *flits*. *Flits* in turn are divided in physical digits (*phits*), the words that are sent over the wires between two modules. The first *phit* of each *packet* carries the header, which has the path of the packet and credits.

Based on the input queue and the configuration of the NI, the NI arbiter schedules the *flits* of the packet and sets its path in the header. According to such path, the routers of the network will forward the flits until they reach their destination NI. Due to the contention-free routing, no arbitration is needed in the routers.

The destination NI takes the streaming data from the packet and places the phits in the output queue. The initiator shell deserializes the request and issues a transaction in its initiator port. The shell could be directly connected to the target IP_2 if the latest is not shared with more initiators, otherwise an initiator bus would be needed, like for IP_3, which is shared by IP_1 and IP_4.

Finally, the response is generated by the target, serialized by the shell and fed into the NI as streaming data. Once this data is scheduled, it crosses the router network all the way back until it gets stored in the output queue of the first NI. The target shell recomposes the response and sends it back to the bus and this to IP_1.

Æthereal gives interconnection services to the *applications* of the SoC. Thus, many times applications need to share the network and some memory mapped targets, with some latency and throughput requirements. However, in Æthereal arbitration only takes place in the NIs and in the initiator buses. However, due to contention-free routing, there is no need of any additional arbitration inside the network [27]. The NI arbiter has a programmable TDM table that regulates the injection of GS flits in the network in such a way that two flits can never arrive at the same link at the same time. Thus, the network can not experience congestion. This implies that each connection behaves like an two independent FIFOs (one for each direction), enabling predictability and allowing to guarantee bounded latency and minimum throughput per channel.

3.3 Æthereal Protocol Stack

Æthereal is designed to give a suitable communication service to a wide variety of applications with different requirements. Thus, it implements three types of communication:

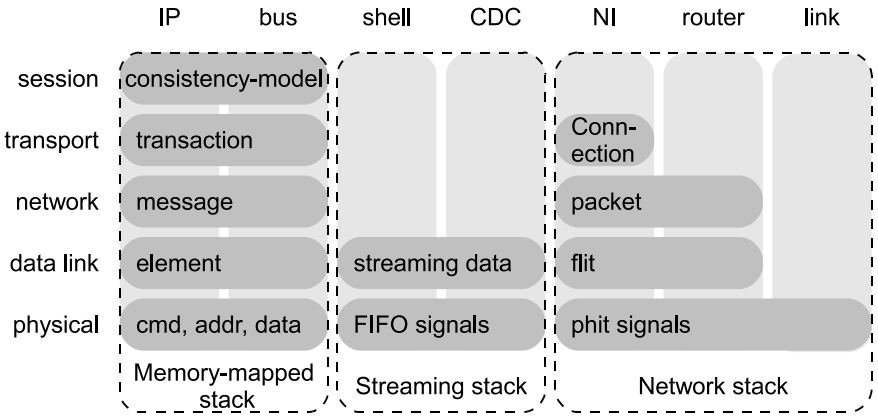


Figure 3.2: Interconnect protocol stacks [16].

memory-mapped, streaming, and network communication [16], each with its own stack. Figure 3.2 shows how these stacks could be divided into five layers according to the seven-layer Open Systems Interconnection (OSI) reference model [7].

3.3.1 Network Stack

The network stack is based on *connections* [28] that provide bidirectional communication channels between two NIs guaranteeing in order delivery thanks to end to end flow control. Each connection identifies the communication of an application and its properties, such as guaranteed throughput, bounded latency and jitter. As shown in Figure 3.2 connections correspond to the transport layer of the stack. Thus, from NI to NI a connection behaves like two FIFOs (one per direction). One IP feeds data in the input streaming port that later is delivered to another receiver IP at the other end of the network. Æthereal connections must be created with the desired properties before being used, reserving resources inside the network such as buffer space or percentage of the link usage. Connections must be closed after they are used, in order to release the resources for new connections.

Connections data is carried from one NI to other by the routers. They are responsible of routing packets according to the content of the header. Æthereal routers have no routing table because the path that each packet must follow at each router is in the header. GS traffic only needs one-word buffer per input port at routers. There is no arbitration done at routers because the network is designed to be contention-free. Arbitration and scheduling is done by the NIs at the level of flits, and each flit is made of three phits, that are the signals that cross from one router to the next one.

Routers have buffers for BE traffic. BE flits are scheduled when there is no GS traffic and the next router in the path has enough buffer space. Thus, BE flits have local-link flow control. Contention between two BE packets is solved with Round-Robin arbitration.

3.3.2 Streaming Stack

The streaming stack only includes the data link and the physical layer. This stack is used by the NIs, Clock Domain Crossing (CDC) modules and shells. IPs might also take advantage of this stack if they have streaming ports. Such streaming ports make use of a simple FIFO interface with a valid and accept handshake. With this mechanism the data link level is able to handle the flow control of each individual word of streaming data. When reading from a streaming port that has no data available (e.g. due to an empty FIFO) or writing to one that can not accept more data (e.g. due to a full FIFO) causes a process to stall. Thus, the data link layer provides flow control in the streaming stack.

Figure 3.2 also shows how NIs bridge between the network stack and the streaming stack and connections offer bi-directional point-to-point streaming communication without any assumption on the time or value of the individual words. This is the most basic type of communication that an IP can use.

NIs are also in charge of buffering, arbitration and flow control. They store each packet until it can be scheduled. The scheduling is done at the granularity of flits. GS flits are send to the router according with the TDM table and the credits available at that moment. The flits of the BE packets are sent only when there are no GS flits waiting and when the router has enough buffer space. Each port of the NI can handle data for one connection. NIs works as follows: A shell or an IP places data words on the NI input FIFO. When there are enough credits, this data words are packetised and sent inside the time slot assigned for that connection in the TDM table, which is programmable. This guarantees that inside the network there is no contention, and thus buffering is not needed in the routers. The routers forward the packet according to the path that is in the packet header. When a packet arrives to the remote NI, its data is stored in the output FIFO of the corresponding port. The credits are sent back when data is consumed in the output FIFO.

The interconnect (buses and shells) may have different clock frequencies than the network thanks to the CDCs. These modules allow to implement heterogeneous SoCs in terms of clock frequency and phase, which is specially important in large designs.

3.3.3 Memory-mapped Stack

Memory-mapped protocols are based on a request-response transaction model and have interfaces with dedicated groups of wires for command, address, write data, read data, flags, masks, etc [33][26]. Shells bridge between the memory mapped stack and the streaming stack by serializing requests and responses and feeding them into the NIs. Hence, shells allow point-to-point memory mapped communication by translating the transactions into streaming words and vice versa. Therefore the complexity of the network and its streaming nature remains hidden for the IPs that communicate by addressing the location that they want to access. There are two kinds of shells depending on the functionality. Target shells are connected to the initiator ports of the master IPs and serialize the request and deserialize the response. Initiator shells deserialize the request and forward it to the target ports of the slave IPs.

Some initiator memory-mapped IPs often need to communicate with multiple targets or use distributed memories. Additionally, memory-mapped targets may also be shared by multiple initiators. Thus shared targets must be arbitrated, and initiators transactions multiplexed accordingly to the protocol of the target port. Such arbitration and multiplexing can not be done by the network since it only offers point-to-point connections, without ordering neither synchronization between them. Such problems are addressed outside of the network, and are solved by placing very simple buses between the shells and the IPs. Thus ordering and synchronization of the transactions takes place at the session layer and is carried out by the initiator buses while target buses forward the request to the right target based on the address of the requests.

3.4 Æthereal Configuration

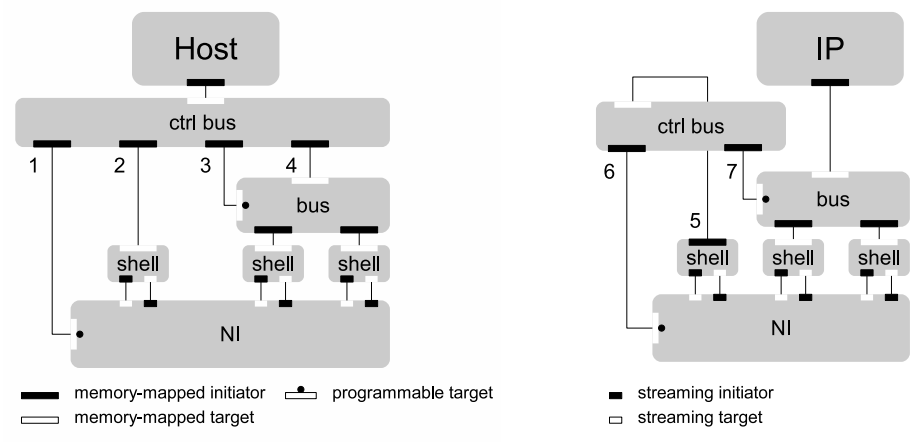
Æthereal provides configuration and control infrastructures, allowing the SoC to adapt to different applications that may run at different moments. This reconfigurability must be performed by some IP (typically a microcontroller) that we call the host. The host is able to open and close connections by changing the values of the TDM slot table in the NI through its dedicated control port. The address layout is configured at the programmable buses, that allow to change the way they decode the transaction addresses. In other words, by programming the buses in the right way the host can choose over which connection is carried a transaction according with the address. Additionally, NIs must also be able to set the proper path in the header of each packet and hence they need to be programmed accordingly.

This configuration is done in a distributed fashion. The host is able to reach the control ports of all the NoC modules using interconnect itself, without any additional infrastructure at the network level. Figure 3.3 shows an example of configuration architecture possible in Æthereal. The local control bus in Figure 3.3(a) is a fixed address decoder. This bus allows the host to perform transactions over different connections. Thanks to the first port of the control bus (called local port), the host is able to open and close control connections to other remote control buses. The second port (called remote port) enables the host to use the previous opened connections and configure the remote modules. The third port (programming port) is used for configure the programmable target bus. This bus is used for the application in which the host is the initiator. The host accesses the remote control buses (Figure 3.3(b)) through the network using the control connection opened before. The remote control bus is used to configure the connections at the remote NI and to configure the remote programmable buses. IPs could also be configured in a similar way.

3.4.1 Initializing the NoC and Opening a Connection

The configuration of the NoC is divided in two steps: Initialization and connection set up. During the initialization, the host sets the configuration connections. After that, the connections required for the applications are opened.

The configuration connections have the request channel, that goes from the remote port of the local NI (port 2 in Figure 3.3(a)) to the control bus of the remote NI (port 5



(a) (b)

Figure 3.3: Local (a) and remote (b) control buses [14].

in Figure 3.3(b)) and the response channel, that goes backwards from the remote NI to the remote port of the local NI. For the initialization stage the host writes the path of the remote NI in the local NI through the local port 1. Now the request channel is thus open. The host uses this channel to configure in the remote NI (port 5 and 6) the path to the local configuration port and to open the response channel. Before configuring the next remote NI, the request channel is closed. This procedure is done for each NI of the Network. At the end of the initialization stage, all the remote NIs have one response channel going to the local NI for configuration purposes, but there is only one request channel for configuration open at a given time.

To open a connection between two NIs, both NIs must be configured. First, the host establishes the request channel with the target NI as described before. Second, it writes in the configuration registers of the remote NI (port 6 in Figure 3.3(b)) the path of the new connection and allocates the TDM slots if the connection has GS. Third, the host programs the remote bus if needed. Fourth, the host closes the request channel. The procedure is repeated for configuring the request channel of the connection.

Closing a connection is more challenging than opening one. The reason is that ensuring that no information is lost requires significant effort. Before disable the ports of a connection, the host must wait until:

- No new request are going to be initiated.
- All ongoing transactions between initiators and targets must be allowed to finish.
- No data is crossing the network.

The network can not know about any potential outstanding transaction, thus IPs themselves must implement some sort of mechanism to ensure that no reconfiguration is perform before finishing doing their work.

To make sure there is no streaming data crossing the network, the host must wait until data leaves the input queue, crosses the network, and is consumed from the output

queue. The credits must be delivered to the destination as well. The host can check this conditions in the configuration ports of the NIs. When this conditions are satisfied, the host can proceed to close the connection guaranteeing that no data will be lost. Closing a connection in the NI only requires to unset the enable flag of the corresponding port and unset the slots.

BE traffic does not employ end-to-end flow control to ensure that no data is stored in the routers, an special tagged message as an end-of-stream marker is needed, thus requiring cooperation of the IPs.

4 Bridging Schemes

The goal of this chapter is to provide an overview of the possible locations for placing a bridge that will allow extending the NoC with off-chip communication capabilities. The well-defined Æthereal stack, discussed in the previous chapter, and its layered division will allow to develop a systematic study of the different options available for implementing the off-chip bridge. Figure 4.1 shows the whole protocol stack and the three different communications provided by Æthereal: distributed shared memory, point-to-point shared memory, and point-to-point streaming. All of this communications may have guaranteed service or best-effort service. Figure 4.1 also shows the places and the layers were the bridge could be placed, according to the stack used and the layers involved.

Section 4.1 discuss the requirements that will characterize the bridging schemes discussed in Section 4.2. Finally, Section 4.3 will show the conclusions.

4.1 Bridging Requirements

To come into possible bridging schemes in Æthereal, some requirements will be defined in order to characterize the a bridging scheme. We can characterize a bridging scheme based on the following parameters:

- *Transparency:* The bridge must be hidden from the IPs. In order to send data or send a request, IPs would have to do exactly the same operations no matter

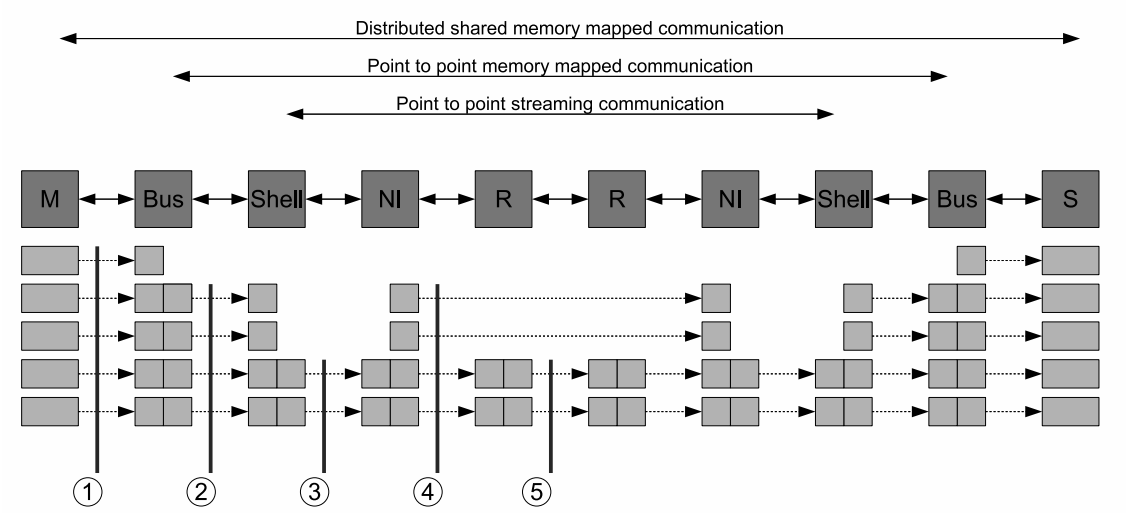


Figure 4.1: Possible locations for the bridge.

whether the target is in the same side of the bridge or in the other side. This means that the addressing scheme of the whole system must remain the same and can not be affected by the bridge. The simplest way to achieve transparency is by designing a bridge that operates at least one layer below the one that the IP is using. With a transparent bridge, the IPs will not need special configurations nor modifications in their interface with the NoC to be placed in a bridged system.

- *Decoupling:* Each part of the bridged must be able to run in different chips or even in different boards. Therefore two subNoCs can have different clock frequencies and or phases. Given the previous requirement of transparency, the bridge must support the full decoupling among the two sub-systems. Thus, the bridge must take care of the physical and temporal dependencies such as voltage levels, distance between boards, clock frequencies and phases differences, etc. The off-chip link will solve the board-to-board differences, while the bridge must have to adapt with some physical differences between the off-chip link and the sub-systems. Since GS in Æthereal is based on contention-free routing, the bridge must guarantee that GS packets must arrive to the routers synchronized with the TDM table of each sub-NoC. Thus, if a given bridging scheme does not need coherence among the TDM tables of both sub-NoC, we will say that such scheme is decoupled. Similarly this affects the clocks of both sub-NoCs. If the clocks do not need to have the same frequency and/or phase, the system is thus decoupled.

- *Quality of Service:* The bridge must preserve the Quality of Service (QoS) that is provided by the network. Inevitably the bridge and the off-chip communication will introduce more latency and perhaps a maximum throughput constraint. This would have to be bounded in order to the support for realtime traffic (traffic that needs to arrive its destination before a given deadline) that the NoC gives. Since the goal is to provide application QoS, both sub-NoCs and the bridge must have QoS.

- *Area Cost:* Since the bridge is meant to be implemented in silicon or in an FPGA low area cost would be desirable. Buffers and memory in general tend to use large portions of silicon so a good bridge design would not need to store too much data. The concept of area cost includes the number of pins (that are normally an scarce resource) used for the off-chip link, and the amount of board resources that it would need. Thus, parallel bridging will be much more expensive than serial bridging.

- *Performance:* The time cost in terms of latency introduced by the bridge must also be kept as low as possible. High latencies may also damage the throughput of the applications that use the memory-mapped stack. When an initiator issues a request, it must wait for the response. Thus, the round-trip delay may damage the number of transactions per unit of time that an IP can execute. To compare the latency introduced by the proposed schemes, the following sections will assume that the delay and bandwidth of the link is always constant and equal for both directions.

4.2 Bridging Schemes

Figure 4.2 illustrates seven different bridging schemes, showing the layer they are and the data granularity they use. It also shows when arbitration is required and the amount of off-chip links needed. The following sections discuss each scheme in detail, and characterize them according in terms of transparency, decoupling, QoS and latency.

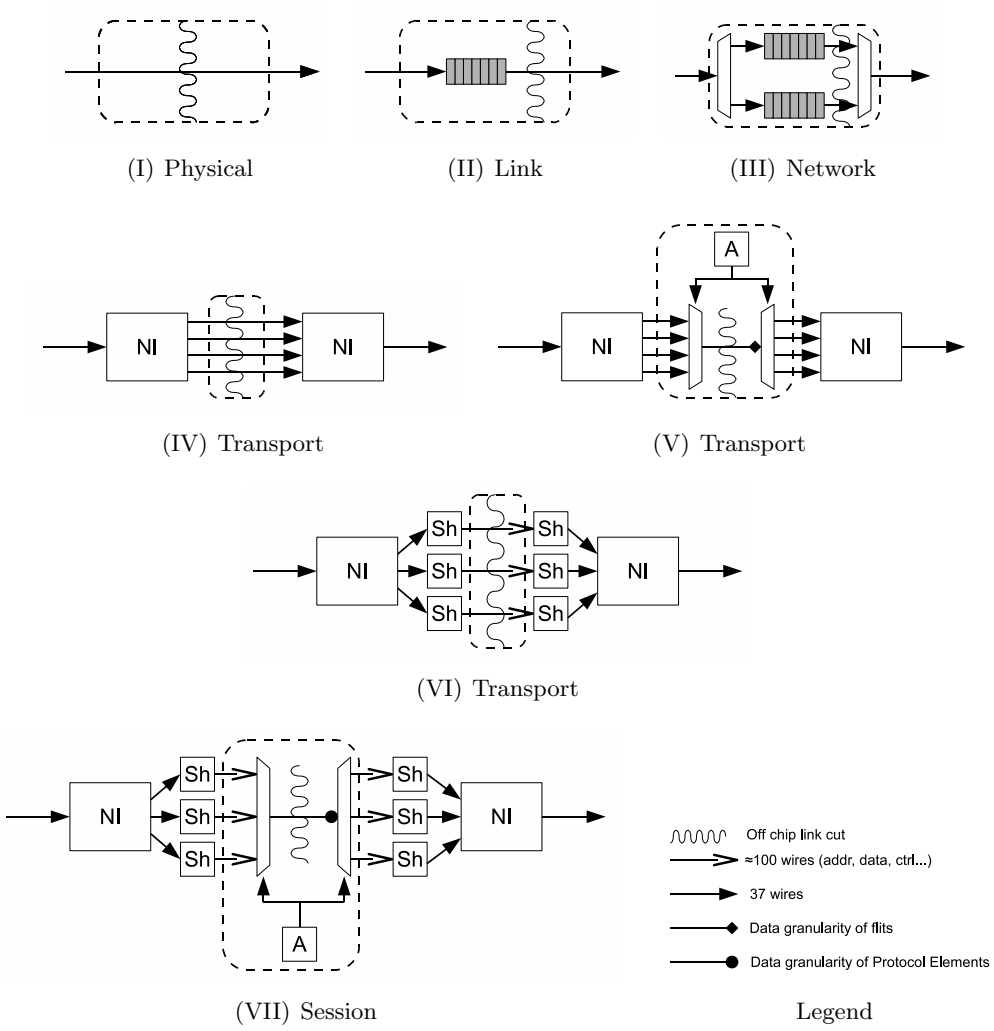


Figure 4.2: Bridging schemes.

4.2.1 Physical Layer

The physical layer is the lowest layer of the stack. The proposed solution for bridging at this layer is scheme I of Figure 4.2. Such bridge would be placed at points 4 or 5 of Figure 4.1. The bridge acts as the NoC link wires, requiring to have a parallel link and a complete clock synchronization, both in frequency and phase, between the two

sub-systems. Such clock dependency and parallel links are expensive if sub-systems are far away each other, i.e. in two different boards. Thus, this scheme does not meet neither the decoupling requisite, nor the cost requisite. On the other hand, the transparency requirement is fulfilled because scheme I is placed at the lowest layer.

BE traffic flow is controlled at the link level [12]. When one flit is received, one credit is sent back. This bridge introduces a significant round-trip delay due to the off-chip latency, thus increasing the overall latency. The throughput of the system would also be affected because a flit has to wait for the previous flit to arrive to the other side and for the credit to come back. Moreover, the QoS can not be preserved because it may happen that the second flit carries GS traffic, and is waiting for a BE flit.

4.2.2 Link Layer

By adding a buffer to the previous design we obtain Scheme II (Figure 4.2). The bridge behaves like a NoC link when is placed at this layer and therefore it will only have one input and one output at each side. Similar to Scheme I there is still a tight time dependency between both sides of the bridge and it requires an expensive parallel link to connect both subsystems. Mesochronous links, shown in [17], can relax the time dependency only to a frequency dependency, so both subsystems could have different phases.

At this level the bridge is similar to a pipelined NoC link, which is transparent to the packets and it must offer reliability and in order delivery service. Hence, the bridge must guarantee that its buffers will not overflow, so implementing some kind of local flow control mechanism is required. Furthermore, the buffer is shared among all the traffic, both BE and GS. When a BE packet enters in the queue, GT packets that might come after will have to wait to be transfer, damaging the QoS of the whole system. Scheme II has even worse round-trip delay as scheme I, so the latency of the system is still bad.

4.2.3 Network Layer

In Figure 4.2, scheme III goes one level up in the protocol stack, reaching the network layer. The bridge is still similar to a pipelined link, but the key difference is that it introduces another buffer. This scheme splits the traffic in two specific virtual channels per QoS class: one for BE and another for GS. QoS is now available, solving the problem of the bridging at the link layer. The bridge is now taking over the role of a router. Thus, the bridge is managing data at the form of packets. In this scheme the local flow control is only needed for BE packets, since the GS ones are scheduled by the TDM tables at the NIs. GS packets arrive when the router at the other side can accept them. Meanwhile, BE packets might need to wait until there is space in the receiver buffer. The receiver buffer must be signaled back when has enough space for a BE packet.

However, this scheme does not solve the time dependency between the two sub-systems. TDM tables at each side still need to be synchronized and coherent. Such synchronization it is very hard and expensive to achieve. Furthermore, the delay or skew introduced by the off chip link must be taken into account and be compensated somehow. Due to the contention-free routing, routers do not have buffers, and therefore no packet can arrive to a router outside of its TDM slot scheduled for that packet in the

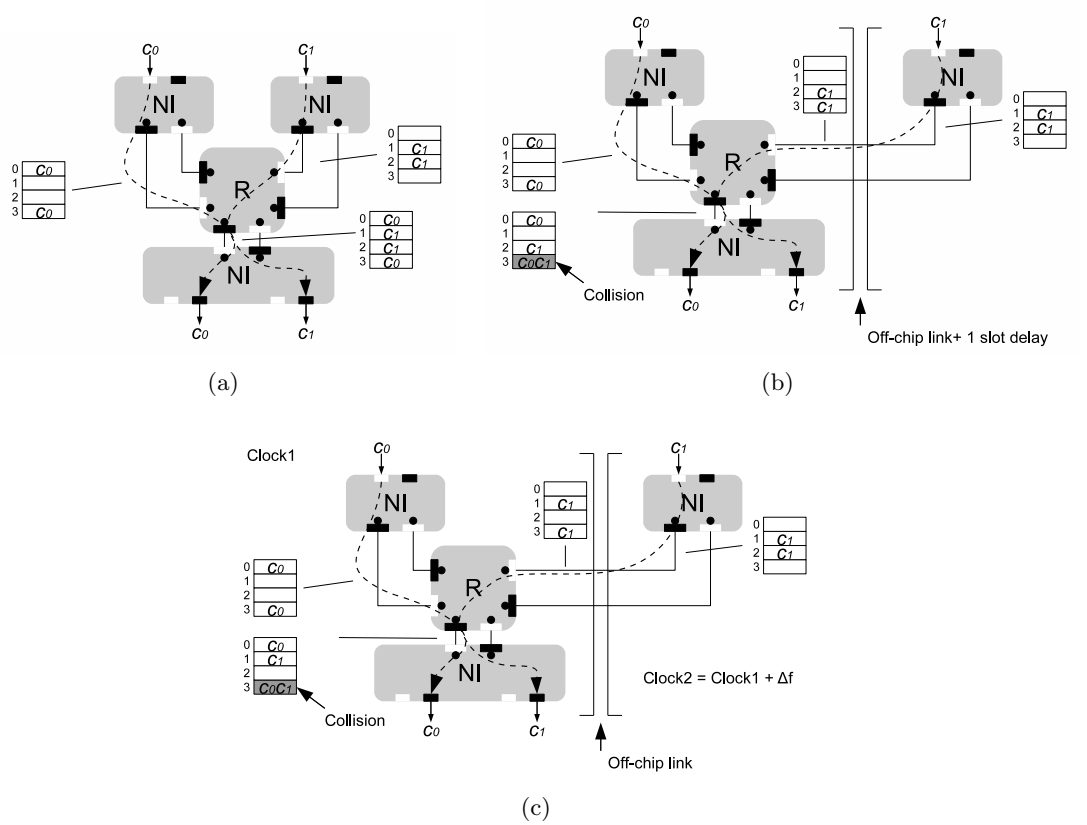


Figure 4.3: Correct contention-free routing (a), and the same NoC with slot unalignment problems (b) or frequency problems (c).

receiver router. Figure 4.3 shows the possible problems due to clock skew or frequency differences. System 4.3(a) shows that channel c_0 has slots $0, 3$ assigned while channel c_1 has slots $1, 2$. Thus, words traveling on c_0 never arrive at the same time to the router than words of channel c_1.

In Figure 4.3(b) the off-chip link adds TDM phase difference to the data transmission. Such phase could be introduced by delay of the off-chip link or by a phase difference of multiple cycles between clock1 and clock2. The result is that one subsystem sees that the other subsystem is one (or more) TDM slot behind or ahead. This slot shifting leads to data colliding at the router output. Figure 4.3(c) illustrates a frequency difference between each system clock. Now, as one system runs faster, some words are received in the wrong slot, and leading to possible contention on the router, what is not allowed by Æthereal.

One possible architecture for a bridge that operates at this layer is shown in Figure 4.4. This bridge has the advantage that it sends data through the off-chip link in the network format. Hence, at the other side the router may be fed directly from the bridge, without queuing the flits. This may lead to reduce the latency and save area resources in one of the chips.

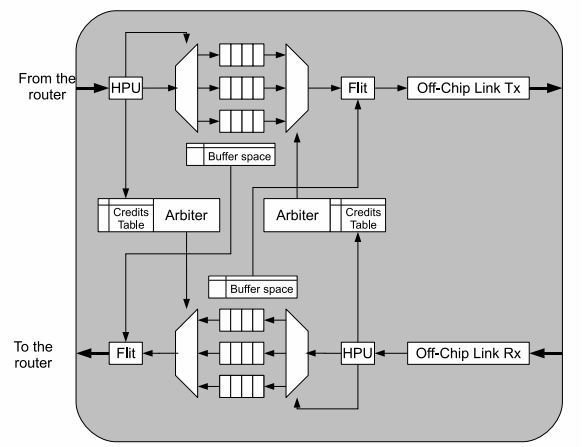


Figure 4.4: Bridge architecture for the transport layer.

This scheme keeps the QoS over the system, since BE traffic and GS traffic are decoupled. However, the cost increases compared to the previous scheme, since now we need two buffers and two off-chip links. The latency is still not good due to the long round-trip delay, same as it happens with scheme I and II. The transparency requirement is met because there is no need of packet header modification inside the bridge, which can transport any packet that arrives to its input.

The clock phase dependency can be solved at the link layer with a mesochronous link like in Scheme II, but the system is not completely decoupled since it requires to have same frequency at both sides. Furthermore, both NoCs need same number of slots and the same slot length. Due to this dependencies and the kind of clock synchronization required for implementing this bridge, the off-chip link would be strongly limited both in distance and thus connectivity. Another approach could be to introduce arbitration in the bridge, which leads us to Scheme IV, one layer up in the protocol stack.

4.2.4 Transport layer

By placing the bridge at point 3 in Figure 4.1, between the shell and the NI, we obtain Scheme IV of Figure 4.2. Such bridge leaves the task of dealing with the TDM tables to the NIs because connections end at the first sub-NoC and restart again in the second sub-NoC. NIs also take care of end-to-end flow control within each sub-NoC. Thus, the bridge is in charge of sending streaming data (messages) through the off-chip link but it also implements flow-control over the off-chip link because now it is possible that the receiver NI would not accept the word that has been sent over the bridge. There is also link level flow-control between the bridge and the NI, with FIFO protocol.

This scheme is transparent for the other modules of the NoC and supports both streaming and memory-mapped communication. Since bridging is now out of the network stack, data words do not need to arrive at a determined moment because the NIs will scheduled them according with the TDM table configuration. Therefore each subsystem may have different clocks and the TDM table of one sub-NoC does not need to be

synchronized with the other. In fact, one subsystem may have different table/slot sizes, or different number of slots assigned for the same connection than the other subsystem. Moreover, by using CDCs it is possible to place the sub-NoCs in different clock domains than the bridge and hence leading to a highly decoupled system.

The main drawback of this scheme is that the connections cross the bridge in parallel. Such a solution might be very expensive in terms of area cost because it needs to replicate the logic for transferring the streaming words to the off-chip packets and it also spends a lot of chip pins and on-board resources. However, dedicated resources for each connection guarantee the QoS of GS is preserved in the whole system.

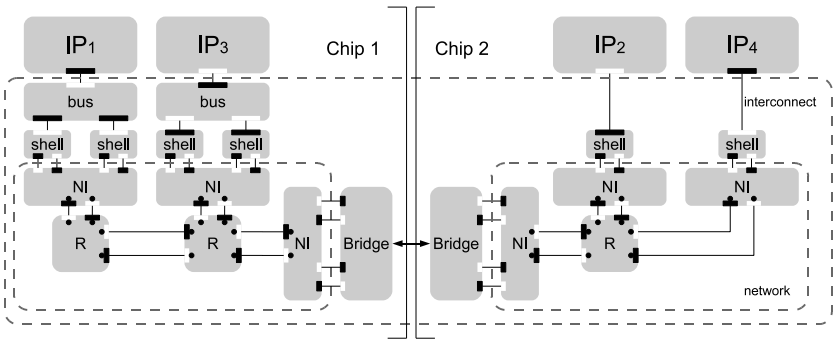
Another drawback that may appear is underutilization of the off-chip links when the connections carry transaction based messages. In this scheme the connections do not share the off-chip link, so the bridge channel will be empty all the time between the request is sent and the response is received.

Scheme V solves the problem by multiplexing several connections over same off-chip link. The bridge allocates a percentage of the transmission time to each connection, accordingly to their needs. The arbitration could be done in a TDM basis or with any other scheduling algorithm that gives QoS, maintaining bounded latency and a minimum throughput for each application. Since both sub-NoCs have QoS within the network stack, we can say that this scheme fulfills the QoS requirements.

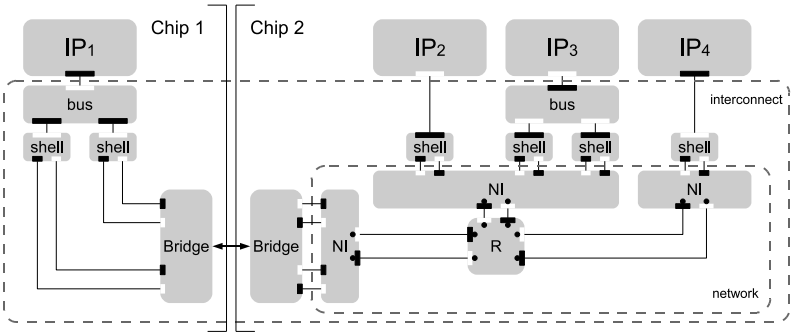
This scheme still needs to establish flow-control across the off-chip link for each connection. The bridge communicates with the NIs with the FIFO valid/accept protocol, this is, link level flow-control. The off-chip link must guarantee minimum throughput and maximum latency in order to support GS connections. If various connections send messages over the bridge, some (de)multiplexing architecture is needed which might increase the area cost.

Regarding the time dependency, it is not longer required to have synchronization between both sub-NoCs. Having same clock frequency and fixed skew is not needed. Of course, the off-chip link must be reliable and offer in order delivery. In order to add guaranteed service over the bridge the off-chip link must provide also bounded latency and minimum throughput. Therefore we can say that the decoupling requirement both in time and among connections is fulfilled.

Additionally, schemes IV and V give some other options for structuring the NoC among two different chips. Figure 4.5 shows two different ways of split the system showed in Figure 3.1 with a bridge that follows scheme V (For scheme IV would be the same, but with two off-chip links, one per connection). In Figure 4.5(a) the bridge splits two sub-NoCs. The first one has IP_1 and IP_3. There are three connections: $IP_1 \rightarrow IP_3$, $IP_1 \rightarrow IP_2$ and $IP_4 \rightarrow IP_2$. The last two need to jump from one chip to the other so the bridge must have two connections. However there is no need of having one sub-NoC in each sub-system. Figure 4.5(b) only has IP_1 in chip 1. Thus, the shells might be connected directly to the bridge, and no network is needed in this chip. This configuration might be interesting in case of large IPs, or temporal IPs that are only needed for testing and debugging stage, i.e. a microcontroller doing the task of a monitor.



(a)



(b)

Figure 4.5: Bridge placed at the transport layer, between networks (a) and between the network and the interconnect (b)

4.2.5 Transport layer, Memory-mapped stack

At the transport layer we can move the bridge to point 2 of Figure 4.1. This implies that the bridge is now working in the memory-mapped stack as shown before in Figure 3.2. Consequently it will not handle neither connections nor streaming data, but it must deal with transactions. Thus, it has initiator and target ports in order to communicate with IPs, shells and/or buses. According to this scheme we would need one link per each initiator-target pair that crosses the bridge. Since transactions have a parallel nature, the bridge can have to use parallel off-chip links, as show in Scheme VI, or serialize the transactions.

So far in schemes I-V the data on a link has been a sequence of words (phits). In scheme VI the bridge sends transaction elements, requests and responses, that might be serialize in order to use the less expensive serial off-chip links. Serializing parallel transactions from the bus format to serial data messages is shown in Figure 4.6(a). At the other chip, the bridge gets the messages from the off-chip link and it converts them to the bus protocol. Finally the shell converts the transaction to serial streaming data, that is fed into the NI. This is very inefficient because the shell and the bridge are performing the same task. In fact, the shell is a bridge between the memory-mapped stack (parallel)

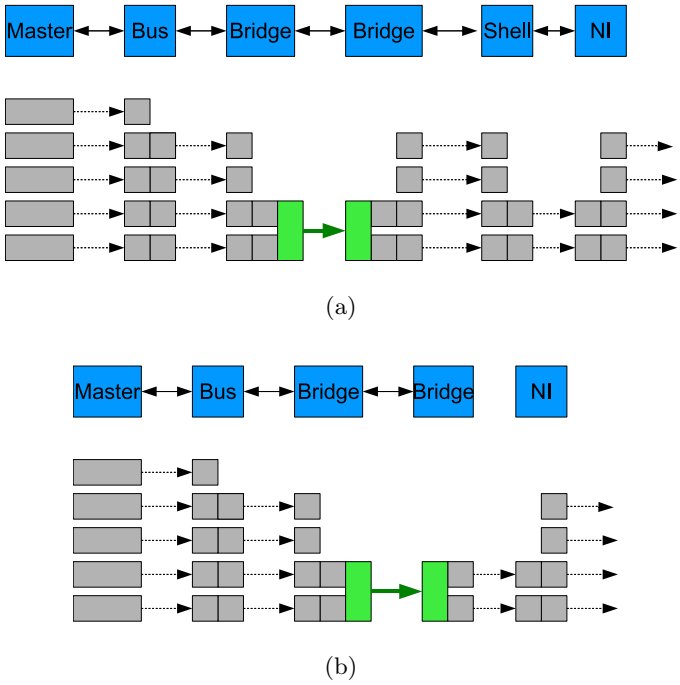


Figure 4.6: Bridge at the session layer (a) and the simplified version (b).

and the streaming stack (serial).

A simplified architecture, shown in Figure 4.6(b), has less overhead while provides the same functionality. This solution removes the redundant (de)serializers and leads to a design that is very similar to the one proposed in scheme V.

Any of these two solutions have a big impact in terms of area cost, both on the silicon and on the board. As shown in scheme VI each initiator-target pair requires to have its own bridging resources. Even worse, due to the request-response nature of the transactions, each of the off-chip links is likely to be underutilized.

Another disadvantage of scheme VI is that it does not support streaming communication. An IP that has an streaming interface can not be used in the SoC unless major changes in the IP itself. Only connections that carry memory-mapped communication would be allowed to cross the bridge. Hence the transparency requirement is not fulfilled.

4.2.6 Session layer

Moving one layer up, the bridge reaches the session layer. Such bridge is illustrated in scheme VII of Figure 4.2. This bridge can be placed at point 1 or 2 of Figure 4.1. The main difference with scheme VI is that now the bridge is in charge of multiplexing transactions, thus allowing to have only one off-chip link. In order to provide QoS and support GS arbitration between transactions is required. Thus the off-chip link, that might be parallel or serial, can be shared among several initiator-target pairs improving its utilization compared to scheme VI. A serial link would require to add a serializer, increasing the complexity of the implementation.

Similarly to the previous scheme, the streaming communication is not supported since we are out of the streaming stack, thus degrading the transparency capabilities of the bridge. However, point-to-point memory mapped communication is still available since it is a simple case of shared memory mapped communication.

Since converting transactions to packets is more complex than converting messages to packets, the latency might be bigger than in scheme V. Another drawback is that interleaving transactions might lead to loose the order between them, to deadlock, or to underused throughput or bursty traffic in the off-chip link due to bridge stalls while waiting for responses. Stalls damage the latency, and the lost of order among different transactions lead us to say that the decoupling requirement is not fulfilled. Furthermore it is much more simple interleaving messages than transactions and this may have negative impact on the area cost on the silicon, compared to scheme V.

Another problem of this scheme is that the bridge may have to implement interfaces for all the bus protocols supported by Æthereal, increasing the design time and damaging the modularity of the whole system because a different bridge must be implemented for each bus protocol. Depending on the specific details of the bus protocols, interleaving transactions of different bus protocols might not be even possible.

4.3 Summary

Table 4.1: Comparison of the different bridging schemes.

Stack	Scheme	Transp.	Decoupl.	QoS	Cost	Latency
Network	I Physical	Good	Very bad	Bad	Regular	Bad
	II Link	Good	Very bad	Bad	Regular	Bad
	III Network	Good	Very bad	Good	Bad	Bad
Streaming	IV Transport	Good	Very Good	Good	Very Bad	Good
	V Transport	Good	Very Good	Very Good	Very Good	Good
Memory mapped	VI Transport	Bad	Good	Good	Very Bad	Bad
	VII Session	Bad	Regular	Bad	Bad	Bad

Table 4.1 summarizes the discussion of the previous section. With all this information, we can conclude that the best option for bridging between two sub-NoC placed in different boards is scheme V. It fulfills the requirements of transparency and decoupling, while providing QoS at a low cost in terms of silicon and board resources. Furthermore, and thanks the transparency, it also allows to keep Æthereal configuration architecture. In the following chapters the design and implementation of a bridge that follows scheme V will be discussed.

5 Bridge Design

The purpose of the bridge is to provide a connection between two NoC in such way that two IPs can send data between each other in a transparent fashion.

1Gigabit Ethernet link is chosen due to its high throughput and its availability in almost all the commercial FPGA boards in the market. Some FPGAs, moreover, are equipped with hardwired Ethernet Mac modules saving logic resources. The drawback of using this modules is that the bridge will be dependent on the hardware available at the FPGA and at the board.

The bridge is divided in two different parts according to the protocol is dealing with. On one side it has ports that accepts the streaming data words (phits) coming from the NoC, and on the other side Ethernet frames goes to the Ethernet link. Thus, the bridge must deal with the physical layer of the NoC and with MAC and physical layers of the Ethernet stack, always trying to reduce latency and maximize the throughput.

5.1 Architecture

The bridge is set at the NoC transport level. From the NoC point of view, the bridge creates connections between two different NIs located in different chips. Thus, data that comes out from one port of the NI on one chip will be fed to the NI on the other chip. Such data will be sent through an Ethernet link. In order to increase the utilization of the link and to allow more flexibility, it is designed to allow multiple connections over the same bridge. Thus, the bridge has multiple input and output ports, one for each connection.

In Figure 5.1(a) it is possible to see the multiple bridge connections from the NoC point of view. The other parts of the NoC will not realize that two different NoC are involved in the communication. The IPs or shells will use a point to point streaming communication either for on chip communication or for off chip communication. Figure 5.1(b) shows that multiple connections may share the same off chip link. Such connections may also have different source or destination NIs.

The NoC provides flow control at the transport level, that is, between two NIs. Flow control also exists in the data links from the IP to shell and from the shell to the NI. Since both NoCs may have different data rates, the buffers in the receiver side of the bridge may overflow if the transmitter side sends data too fast. In order to avoid buffer overflows in the NIs or in the bridge we need to introduce flow control in the bridge connection, between the NIs of each chip. Thus, the bridge module will have both transmission and receiving buffers per connection and it will use a credit based technique to prevent those buffers from overflowing. Such buffers will also adapt the different data rates of the NoCs and the Ethernet link. In Figure 5.2 we can now see that the flow control covers all the path between the source and the destination.

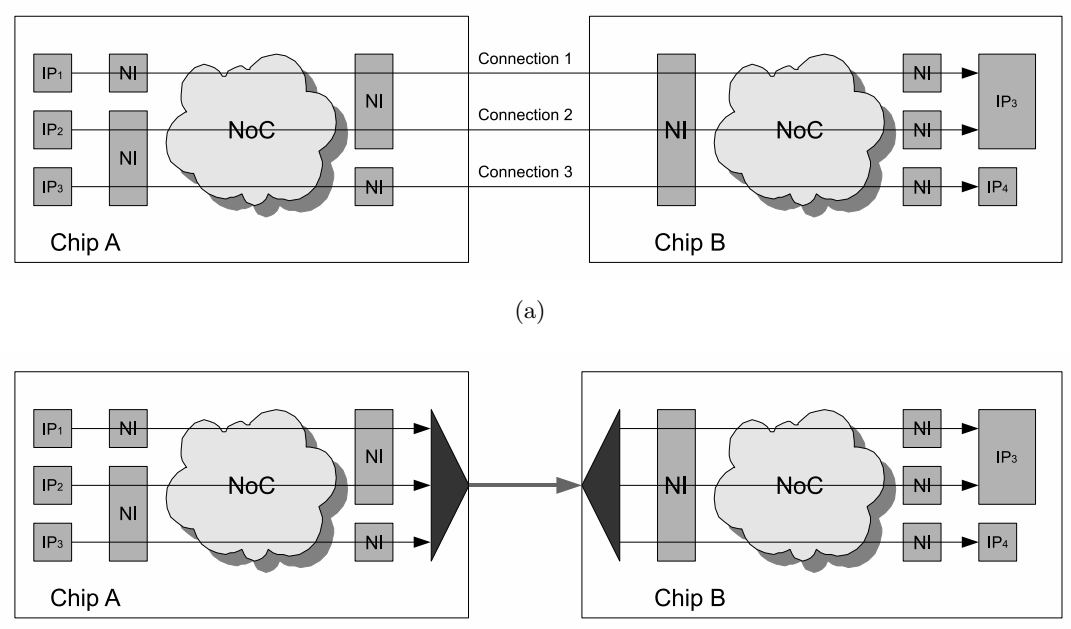


Figure 5.1: Logical view (a) and link view (b) of the connections across the bridge. The connections share the same off chip link.

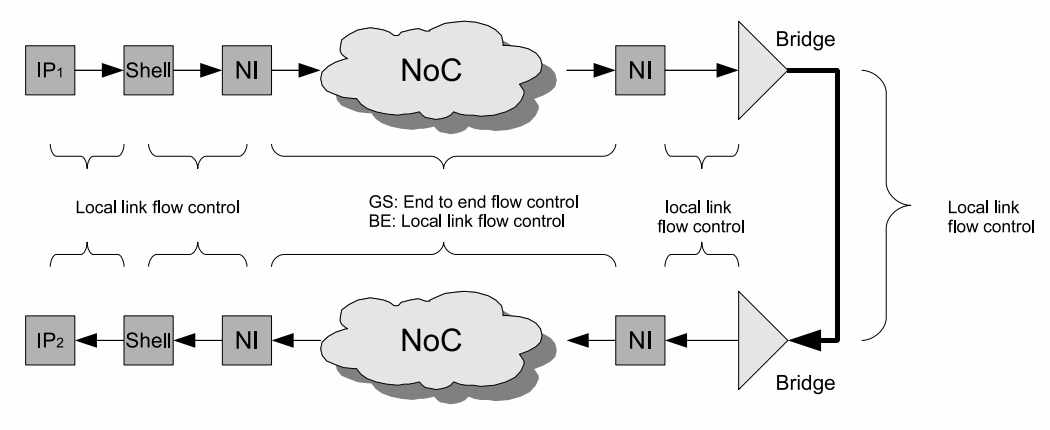


Figure 5.2: Flow control between IPs

5.2 Connection Multiplexing and Scheduling

Since multiple connections can be sent over the Ethernet link, a multiplexing mechanism is required. The chosen approach involves chopping the payload of the Ethernet frame in sections, that we will call slots. The data carried by each slot is uniquely associated with one connection by the connection identifier. In the receiver side such identifier is used to select the output buffer where the data must be placed and it does not need to know

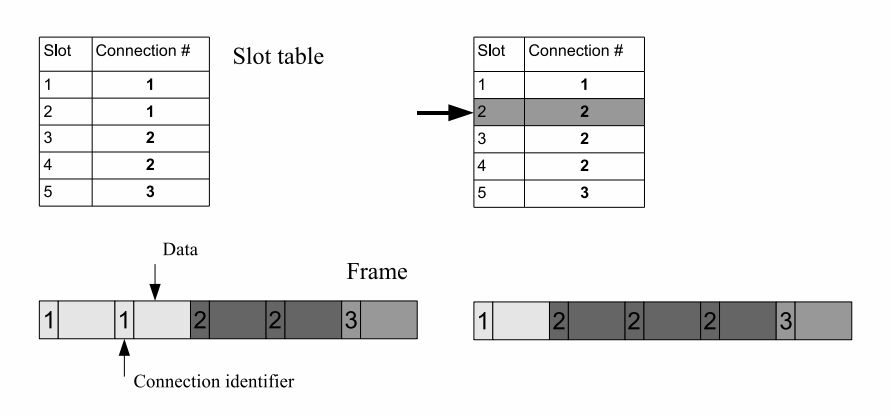


Figure 5.3: TDM slot allocation.

the scheduling policy that was used on the transmitter side. Thus, the bridge can use dynamic scheduling policies like Round Robin. Other advantage of this scheme is that when using Time Division Multiplexing it will be possible to change the slot assignments at run time.

In order to decide which slots belong to a connection, the bridge has three possible scheduling policies: Time Division Multiplexing, Round Robin (RR) and priority based scheduling. With priority scheduling the bridge assigns the next slot to the connection that has the biggest priority. If its buffer is empty, the slot is assigned to the next connection in the priority scale. With Round Robin policy, each connection can only use one slot consecutively. The next slot is for the next connection that has data in the buffer.

When the bridge is using TDM, the slot table is used. This slot table has as many entries as slots are in the ethernet frame. The table assigns each slot to one connection, and a connection may have zero, one or more slots assigned. Since the slot table is a RAM memory, it is possible to reprogram it at runtime. The slot table is connected to a memory-mapped port of the bridge. Thus, each entry has a unique address in the NoC and the user is able to effectively change the throughput assigned to a connection by writing the connection number in the table entry, as it is shown in Figure 5.3. The more slots a connection has, the more throughput it can use.

A connection identifier is sent along with the data in each slot. The advantage of this technique is that the receiver side of the bridge does not need to know the configuration of the slot table of the transmitter, making easier the configuration of the bridge.

5.3 Flow Control

In order to keep end to end flow control and avoid data lost it was decided to add a credit based flow control mechanism on the off chip link. The bridge implements flow control at the transport level so there are two FIFOs per connection, one for data coming from the NOC (Tx FIFO) and other for data coming from the Ethernet (Rx FIFO). FIFOs store phits, the data unit of the data link level.

The Tx FIFO has a *credit counter* associated to it. Each time a phit goes out of the Tx FIFO, the counter is decremented. If the counter is zero, no more phits can be sent. When credits are received from the other side of the bridge, they are added to the credit counter, so the connection can start sending phits again. The value of this credit counter will never be bigger than the number of available positions in the Rx FIFO. Thus, it is not possible for the transmitter to send data if there is no free space in the receiver.

The Rx FIFO has another counter associated that keeps track of how many positions have become available, called *phit counter*. Each time a phit goes out of the FIFO to the NoC the phit counter increases and when it reaches a preset trigger value, its value (credits) is sent to the other side of the bridge as soon as possible, and will be added to the credit counter of the other side. Then the counter associated to the RX FIFO is reset to zero. Even if the preset value is not reached credits may be sent back to the Tx FIFO when there is no other data to be sent. To avoid deadlock stalls, the trigger value must be lower than the size of the Rx FIFO. Since connections are bidirectional credits are sent back in a slot of the reverse channel. Therefore a connection must have allocated slots in both sides of the bridge. Credits have a dedicated place in the packet format, as we will show later, but they can be sent at any other place inside the slot if the trigger value is reached. This mechanism is necessary to avoid starvation and support very asymmetric connections, i.e. a connection that has many slots assigned in one direction and only one for the credits in the other.

In Figure 5.4 we can see an example of how the credit based flow control works. In Figure 5.4(a) phits arrive to the transmitting buffer and it has enough credits to send them to the other chip. When those phits are sent the credit counter is decreased in an equal amount (see Figure 5.4(b)). When the receiver consumes a phit it increases its own counter. When this counter reaches the trigger value in Figure 5.4(c), its value is sent back to the transmitter side in order to update the credit counter as it is shown in Figure 5.4(d). Figure 5.4(e) shows the case where the credits are sent back even when they did not reach the trigger value, because the link was free at that time.

5.4 Ethernet Packet Scheduling

The Ethernet (IEEE 802.3 standard [20]) frame format has a very big overhead, as is shown in Figure 5.5. It needs 6 bytes for each address, 2 bytes for length, and 4 bytes for CRC the preamble, and a minimum payload of 46 bytes. Thus the minimum MAC frame size is 64 bytes large. In the physical layer of Ethernet a preamble of 8 bytes and an interframe gap are added, so the minimum frame length is 84 bytes.

On the other hand NoC phits are 37 bits. Since the MAC layer of Ethernet deals with complete bytes instead of bits, phits will be encapsulated in 5 bytes. The shells are going to be used in the implementation convert a write transaction of one word into 3 phits. Each extra word in the transaction requires one extra phit. Read transactions only need 2 phits for the request, and as many phits as requested words for the response. The maximum amount of words that can be requested in one transaction is 32.

Using one frame for sending each transaction means that we will be using 84 bytes for sending 15 bytes (simple write request) of useful data. By doing some simple calculus we can see that such solution would not be efficient and the performance would be extremely

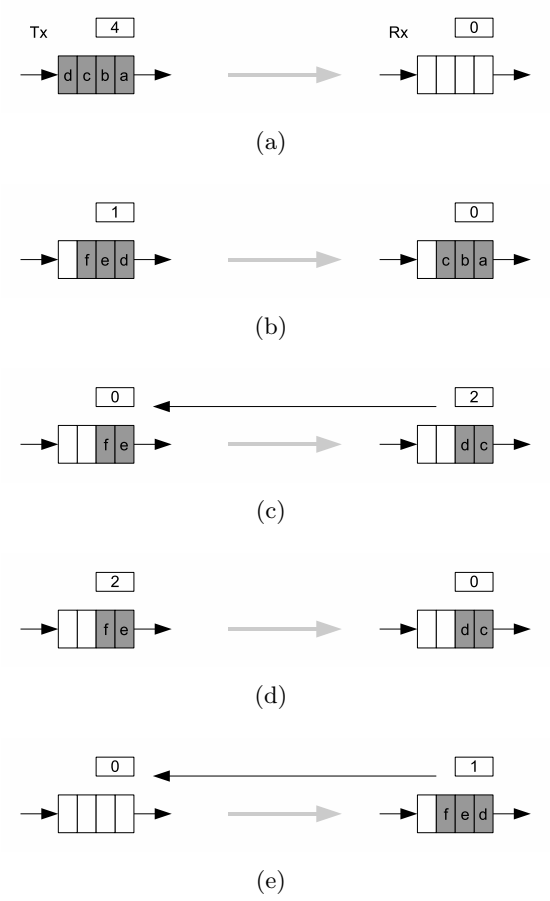


Figure 5.4: Example of the flow control mechanism.

Preamble (8 bytes)	MAC dest (6 bytes)	MAC src (6 bytes)	Type (2)	Payload (64-1522)	CRC (4 bytes)	Interframe gap (12 bytes)

Figure 5.5: 802.3 MAC Frame.

low.

$$\eta = \frac{Data}{Frame\ size} = \frac{sreq \times s_{phit}\ (bits)}{L_{eth}\ (bits)} \tag{5.1}$$

Where s_{req} is the number of words of the request or the response, $s_{phit} = 37bits$ and L_{eth} is the length of the Ethernet frame. Figure 5.6 shows the efficiency when an Ethernet packet carries only one transaction. The graph illustrates this scheme is highly inefficient for small transactions. For example a 10 word write transaction would barely reach 50% of the link capacity. Due to the minimum size of the payload of the Ethernet frame, during the first part of the graph the frame size remains constant and therefore the efficiency grows linearly. When the transaction fills the payload the frame size starts

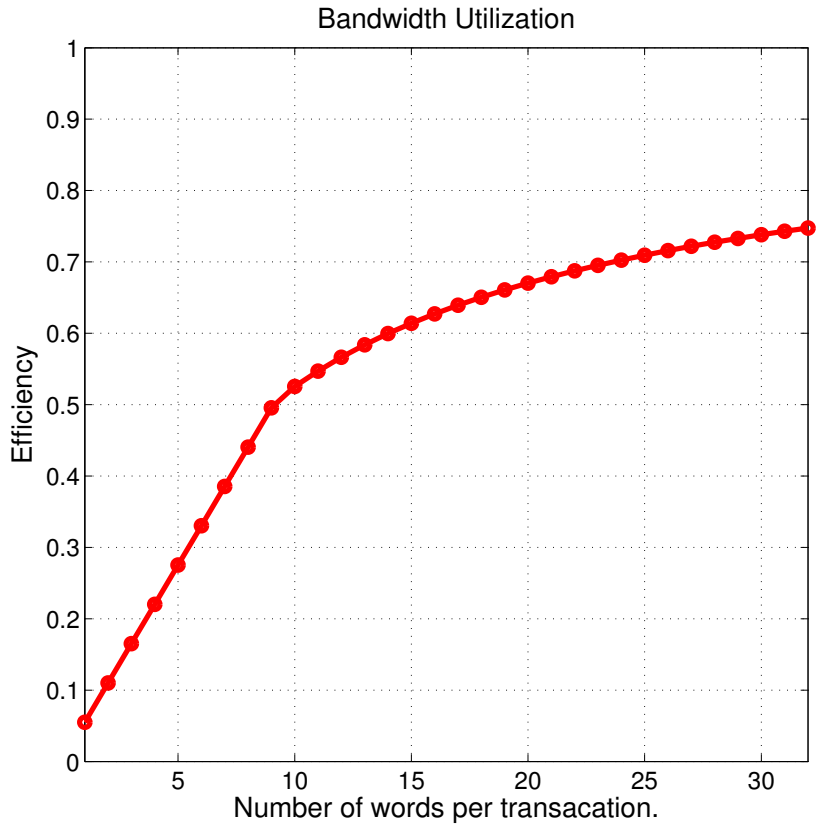


Figure 5.6: Bandwidth utilization of the Ethernet link.

increasing as well.

The other issue with this solution is that a frame can not be sent as soon as a phit arrives to the bridge because the MAC layer has to send first the preamble, adding more latency.

Other option could be waiting for a minimum number of phits, but this is not feasible due to some reasons: if a reading command arrives to the bridge it would wait for more phits but they are not going to arrive since the module is waiting for the respond to the completion of the reading command, leading to a deadlock situation. A timeout mechanism would solve this issue, but then we will find that the bridge had to wait for sending Ethernet frames with a few phits (because of the timeout), so the latency is bigger and the efficiency is still poor. Furthermore, frames can not be sent immediately one after the other due to the preamble and the interframe gap.

With the aim of increasing the efficiency without damaging the latency, the bridge uses a solution based on sending larger frames than it is needed and with fixed length. Such frame is divided in slots, that carries the phits of one connection. If any phit is available in the queue or arrives at that the very moment the slot is been sent, it will be placed on the payload. The larger the frame and the slots are, the bigger chances

that when a phits arrives can be placed on the Ethernet frame without waiting too much time. Thus the efficiency would increase when big amounts of data arrive to the bridge while the latency would be kept at its minimum.

Then main drawback of the described system is that it sends frames even if there is no data to be sent. This does not have major issues when operating only with Ethernet links but in a future it will be possible to allow the bridge sending Ethernet frames through an Ethernet switches. Sending frames in a fast and regular basis even if there is no data to be send may not be feasible since it could overload the switches. In this case another packet scheduling mechanism might be necessary, including a good throughput planning and quality of service.

5.5 Packet Format

The payload of the frame is divided in slots plus one identifier byte at the beginning for debugging purposes. The first byte of the slot carries the slot identifier. The two most significant bits of the slot identifier will always be "01" and the other six bits of the byte are the connection number, meaning that all the data carried on the slot belongs to that connections.

The second byte is used to send back the credits of the connection. The most significant bits of the credit bytes will be "10". The following six bits are the amount of free slots in the receiving FIFO that is in the transmitting side. If more credits are needed the transmitter can send more credit bytes, but always in a slot of the corresponding connection.

Phits will be placed after the credit byte(s). In order to know whether one byte of the slot is part of a phit or not, the bridge provides a little header to each phit. The first three bits must be set at "1" and the thirty seven bits of the phit data after, which makes exactly five bytes. Thus, when the receiver finds a byte that starts with the bits "11", it means that the following 37 bits transport a phit. From this 37 bits, the first 5 bits are in the first byte, and the other 32 bits are in the following four bytes.

If the first two bits of a byte are set to "0" means that that byte does not contain any useful data (garbage data). Thus, phits can start at any byte in the slot, and two might be placed one just after the other or with any amount of garbage bytes in between, as is shown in Figure 5.7. This allow to place a byte in the slot at any time the phit arrives to the bridge, emptying the buffer and leaving space for more data.

Assuming a bridge that uses the maximum frame length (P_{eth} =1500 bytes of payload) and $N_s = 10$ slots per frame, it is possible to find out the efficiency of this protocol. So each slot has $s_{slot} = 150$ bytes. Thus, the number of phits per slot is:

$$phits_{slot} = \frac{150 - 2}{5} = 29 \ phits \tag{5.2}$$

The minus two is due to the connection byte and the credit byte. Finally the efficiency for this system is:

$$\eta = \frac{N_s \times phits_{slot} \times s_{phit} \ (bits)}{L_{eth\_max} \times 8 \ (bits)} = \frac{10 \times 29 \times 37}{1542 \times 8} = 86.98\% \tag{5.3}$$

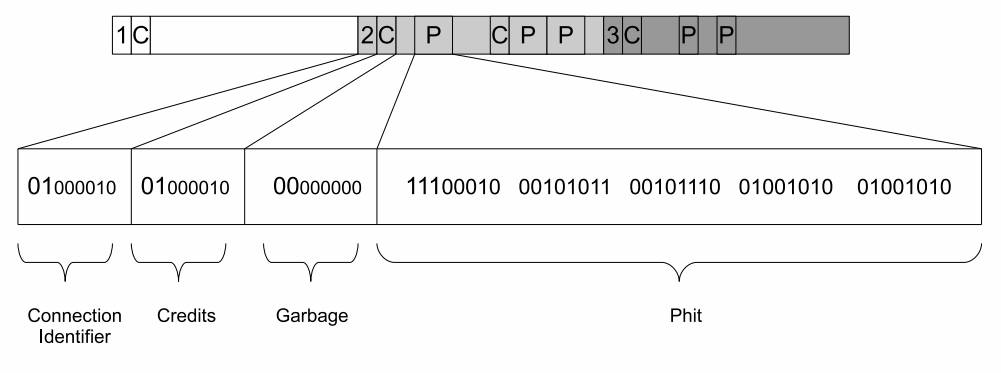


Figure 5.7: Protocol format

Thus, the throughput will be 870Mbps if the bridge is using the full bandwidth of 1Gbit Ethernet.

5.6 Formal Model

Æthereal NoC offers performance analysis in such a way that the designer can verify that the requirements of each application are met. The characterization of the applications and the NoC behavior is done by constructing models, given as variable-rate dataflow graphs [14][18]. To model a system with two sub-NoCs we propose in this thesis to model each sub-NoC individually, and insert a model of the bridge in between.

In order to construct a model the bridge we propose to use the same approach as Æthereal: a cyclo static data flow graph [3][31]. The connections that cross the bridge are scheduled by using a TDM scheme. Thus, according with [18] each one of those connection that crosses the bridge can be seen as a latency rate server.

The data flow graph for modeling the bridge is the same that is described in [18] and is shown in Figure 5.8. In this model data and credits have different channels and each channel has different actors for modeling the latency and rate. Actors $v_{d,\theta}$ and $v_{d,\rho}$ model the scheduling latency and the rate regulation they suffer in the sender side of the bridge, and actor $v_{d,\phi}$ model the latency introduced by the MAC and the physical layer of the Ethernet link, both in the sender and in the receiver. The three actors on the credit path have the same purpose.

The numbers in the tip of the arrow represents the number of tokens that must be at each actor in order to send tokens to the next actor. The number of tokens that are sent is in the tail of the arrow.

The response time $\tau_{d,\theta}$ of actor $v_{d,\theta}$ is $\tau_{d,\theta} = \theta_d + d_d(t_d)$ where θ_d is the time that it takes to the bridge to accept one word from the FIFO and $d_d(t_d)$ is the worst case latency for data. The latency in this actor might be experienced for more than one data word at a time. Actor $v_{d,\rho}$ bounds the rate at which data and credits can be sent. The response time of this actor is $\tau_{d,\rho} = p_n/\rho_d$ where p_n is the period of the TDM table in cycles, and ρ_d is the maximum number of words that can be sent in one TDM period assuming that there are always credits available. The black dot in the arrow means of

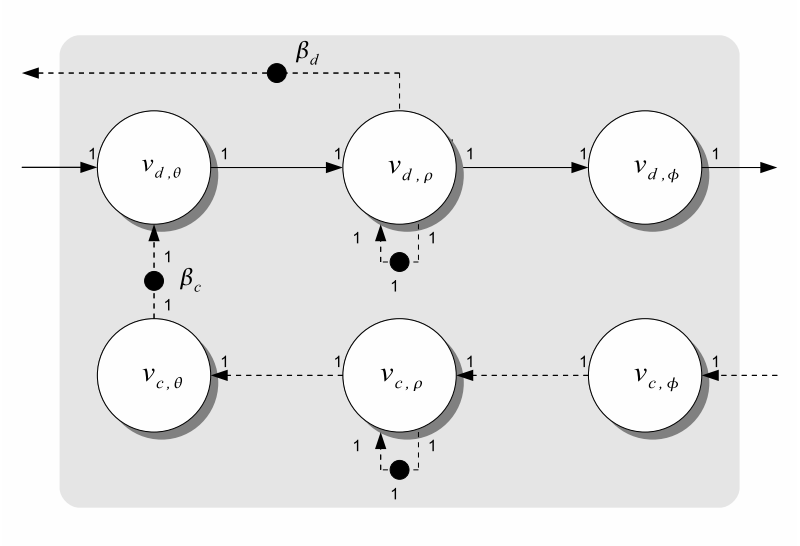


Figure 5.8: Bridge dataflow model

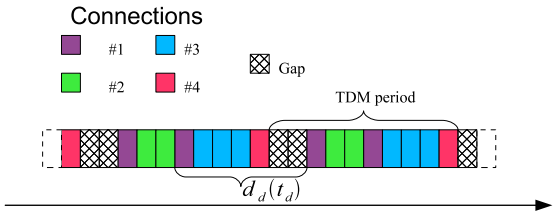


Figure 5.9: Ethernet frames from the TDM slots point of view.

this actor means that token (data phit in this case) only can be sent one by one. $\tau_{d,\phi}$ is the delay that introduces the Ethernet link.

Figure 5.9 is an example of the Ethernet frames that the bridge sends. Each one of the squares represents one slot, assigned to one connection. The *gap* slots model the time between frames and their header. Let be $n_s = 2$ the maximum number of words (5 bytes) in each slot. Thus, for this example the TDM period would be $p_n = 10 * 2 = 20$. For the connection 1 graph this leads to $\tau_{d,\rho} = 20/(2*2) = 5$. The example in Figure 5.9 that in the worst case data will have to wait during seven slots, thus $d_d(t_d) = 7 * 2$. Assuming that $\theta_d = 1$, we obtain $\tau_{d,\theta} = 15$.

With all this values it is possible to model any TDM table configuration of the bridge. This model would allow the designer to know whether the application requirements will be met or not.

6 Implementation

This chapter shows how a bridge with the features described in Chapter 5 is implemented. In Section 6.1 the hardware bridge is presented. Section 6.2 shows the details of a software bridge implementation and. Section 6.3 and Section 6.4 discuss the configuration of a NoC with the host located outside the bridge and the configuration of two sub-NoCs.

6.1 Hardware Bridge Implementation

This section shows how the bridge is implemented in RTL. The bridging logic is meant to be independent of the lower technology, thus being possible to implement it in silicon or in any FPGA. However, the Ethernet hardware is very hardware dependent due to the high time requirements that it needs. Furthermore, Ethernet requires to use a specific chip for implementing the physical layer (PHY chip). There are multiple protocol choices for interfacing with this family of chips (MII, GMII, RGMII, SGMII), all of them standardized. The test platform in this work is the Xilinx ML510 board [32] which has a Virtex-5 FPGA [36]. This board has two physical Ethernet chips with their corresponding connectors, increasing the testing opportunities for the future. One of this PHY chips has both GMII and SGMII interfaces, while the other only is accessible through SGMII. For simplicity and because this standard consumes less pins of the FPGA we will use SGMII for interfacing with both PHY chips.

6.1.1 Hardware Module Description

The top level module description is depicted in Figure 6.1. Note that this figure only shows one bridge with one Ethernet interface. The bridging logic implements all the mechanisms shown in the previous chapter. It multiplexes the NoC connections that come out of the NI, implements flow control within the Ethernet link and converts the NoC words to bytes. The bytes are fed into the Ethernet MAC module provided by Xilinx and called Tri-mode Ethernet MAC (TEMAC) [34]. This module adds the preamble, padding and the Frame Check Sequence (FCS) to the frames. The TEMAC also deals with the negotiation required for setting the link speed, 10, 100, or 1000 Mbps, selecting whether the link is duplex or half-duplex, etc. The TEMAC module is already embedded in the Virtex 5 FPGAs, saving logic resources for other purposes.

TEMAC uses the SGMII interface to connect with the PHY chip. SGMII interface sends and receives serial data at 1Gbps over four wires (two differential signals for sending and another two for receiving). For that reason, frames must be serialized at a General Purpose Transreceiver (GPT) [35] using 8b/10b encoding. The GPT is an special module from Xilinx that takes care of very low level this like ensuring that all the signals and clocks are synchronized and have the same delay. Finally, frames arrive to the PHY chip,

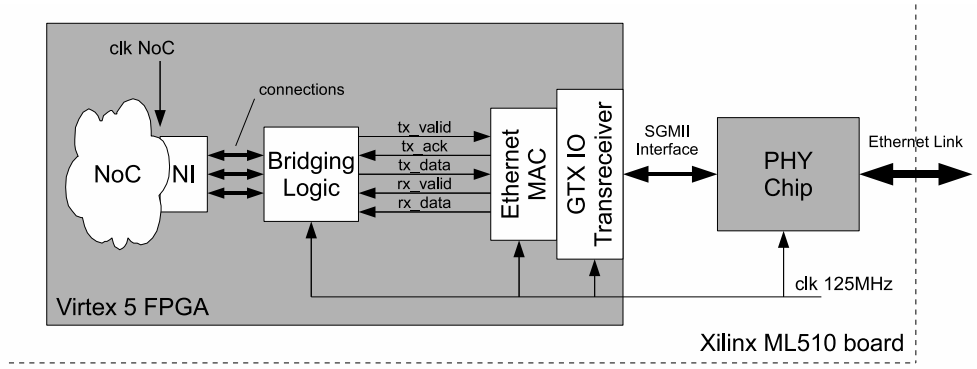


Figure 6.1: Top view of the bridging system.

that sends them to the other side of the link, where they will go through the reverse decoding process until arriving to the other NoC.ding process until arriving to the other NoC.

The bridge has one input FIFO and one output FIFO per port. This FIFOs store the data words of each connection when they can not be sent immediately. Moreover they act like a sort of Clock Domain Crossing because they are dual clock FIFOs and hence they decouple the bridge from the clock domain of the NoC. This is specially important because the Ethernet logic must work with a clock frequency of 125 MHz , while the NoC will typically work with a different clock of 50-200 MHz in the FPGAs, or at even higher frequencies in silicon, around 500MHz [10].

The module description of the transmitter (from the NoC to the Ethernet module) side of the bridge, is shown in Figure 6.2. The flow control logic generates a *phit_valid* signal whenever there is one or more phits in the queue and when there one or more credits for transmitting. Signal *phit_accept* is asserted when the serializer gets the phit and thus, the phit is removed from the queue and the credit counter is decremented.

The phit counter keeps track of how many empty positions are available in the output FIFO. This FIFO stores the phits that come from the Ethernet link and delivers them to the NoC. When the NoC accepts one phit, the phit counter is incremented by one. When its value is over a predefined trigger value t, the signal *credit_tx_request* is asserted to indicate that it is needed to send the value of the phit counter for reloading the credit counter at the other side of the bridge. Once this value is sent, the phit counter is reset to zero.

The serializer gets the phits from the FIFO and split them into bytes and adds the header to both the phits and the credits. If the *credit_tx_request* signal is set to one, the serializer will send the credit value. Otherwise, it will send only the credit value at the beginning of the slot. Finally, the frame sender finite state machine (FSM) deals with the handshake signals of the Ethernet module, selects between sending the header (source address, destination address, frame length and frame number) or the payload, that is the output of the serializer. The frame sender FSM also signals the serializer and the scheduler when a new frame starts.

The scheduler has a TDM selector counter and a TDM table. The counter, change

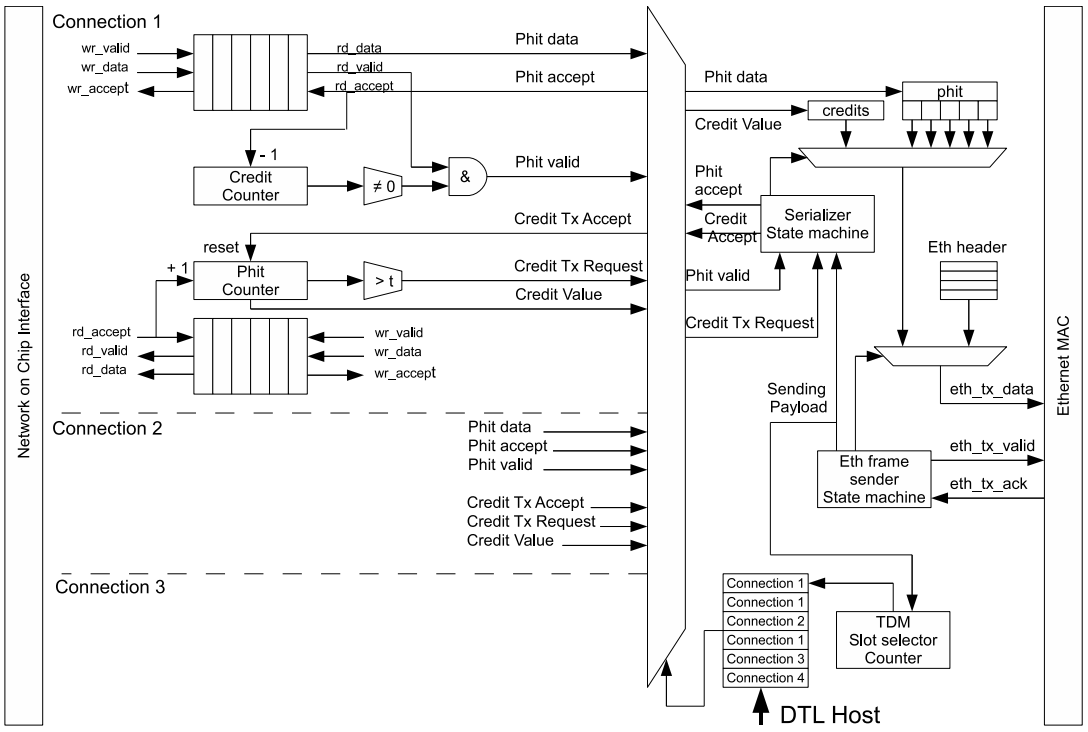


Figure 6.2: Bridge module description (TX)

the address that is fed into the TDM RAM each N cycles. Therefore N is the size of each slot in frame bytes. The resulting address determines which is the slot been sent, and the output of the TDM RAM the connection that can send data over this slot. The connection number is send to the multiplexer, and this selects which connection signals from the flow-control stage will be fed into the serializer. The TDM table of the scheduler can have a default configuration, but it can also be changed through the memory-mapped port of the bridge.

In Figure 6.3 that the frame is received from the Ethernet MAC module. The frame receiver gets the payload data from the frame, and feeds it into the deserializer. This module reconstructs the phits and recognizes the connection labels, and the credits. The connection number is used in the demultiplexer, selecting the appropriate connection at each given time.

After the phit passes through the demultiplexor it is pushed into the FIFO. Due to the flow control system, described in the previous chapter, the output FIFO must always have at least one empty memory position for the phit. Thus, no phits are dropped. When the phit comes out of the output FIFO, it goes to the NoC, and the phit counter is incremented.

When credits value is received, the *credit_valid* signal is asserted and they are added to the credit counter of the input FIFO. In this way the transmitter will never run out of credits if the receiver has enough space in the FIFO to accommodate phits.

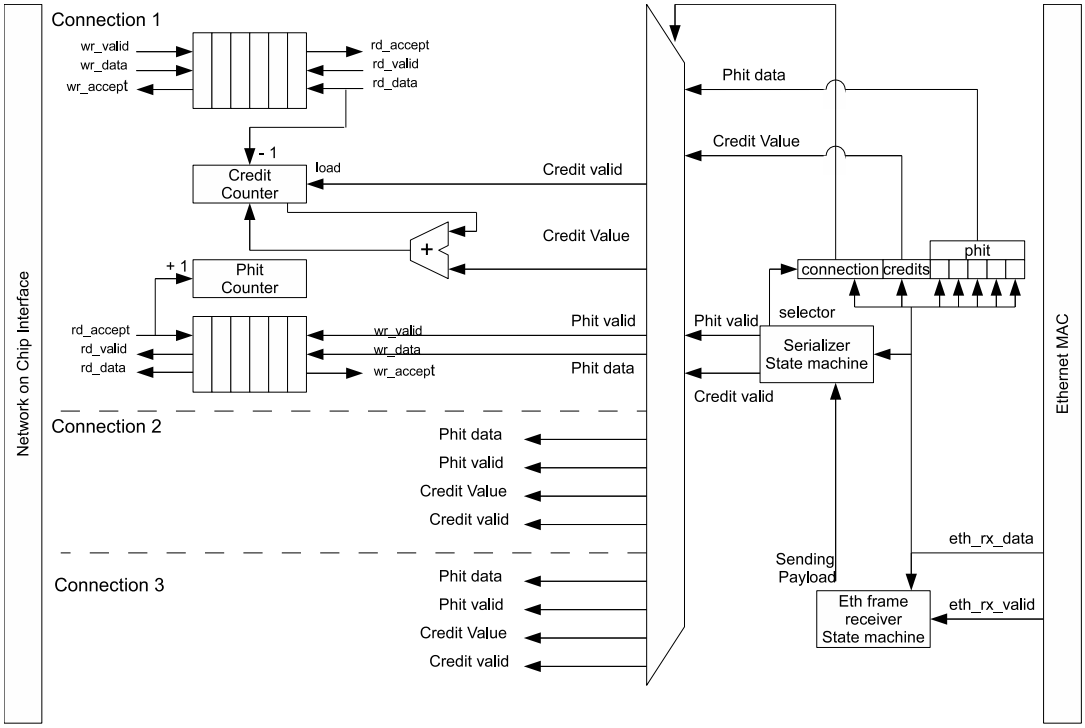


Figure 6.3: Bridge module description (RX)

6.1.2 Hardware Implementation Results

In the previous section we saw how the bridge was built. Now the area cost of the bridge will be explained. Since the implementation is done on a FPGA, Figure 6.4 shows the number of resources consumed in terms of Slices. Those are the results obtained with Xilinx ISE 11.4 for bridges with different amount of bidirectional ports. As expected, the amount of registers and memory resources grow linearly with the number of ports due to the input and output FIFOs of each port. In this test FIFOs are 64 phits depth, therefore 64 is also the maximum number of credits.

Finally, a 12-port bridge implementation occupies the 6% of the Virtex-5 logic resources.

6.2 Software Bridge Implementation

The hardware bridge connects the system not only with other prototype boards, but also with a PC. The PC has an Ethernet card that allows to capture Ethernet frames, but in order to send or receive useful data to the NoC, this data must be encoded in such a way that the hardware bridge and the NoC understands it. Therefore the PC needs a software implementation of the bridge and the NoC protocol stack. This software must provide streaming and memory-mapped communication with the IPs of the bridge, as well as NoC configuration resources.

This section will describe how a NoC Application Programming Interface (API) that

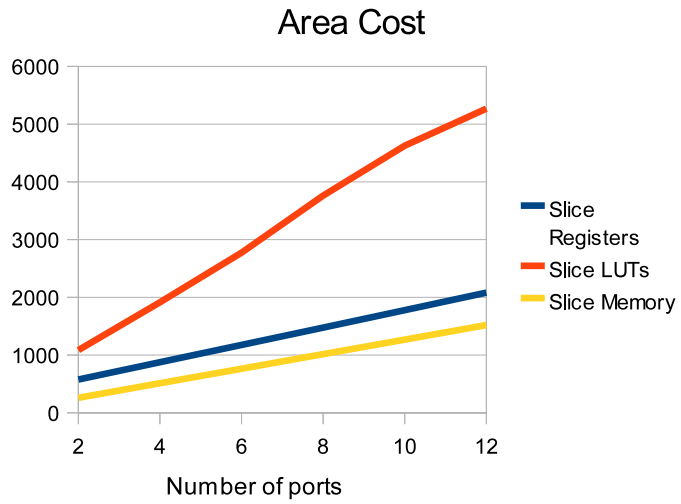


Figure 6.4: Bridge area cost.

offers the required kind of communication is implemented. As proof-of-concept application, a command line tool that uses the NoC API is also implemented, providing configuration and basic debugging commands.

6.2.1 Description of the NoC Software Library

The following description of the NoC API will follow a bottom-up approach, starting from the low level access to the Ethernet hardware and finishing with the transaction level functions.

The Ethernet hardware is accessed via the Linux *sockets*, as shown in Figure 6.5. Opening a socket to the low level network facilities of the Linux kernel gives the NoC API the possibility of sending its own tailored frames. The very low level functions of the NoC API are also in charge of building the frames, writing the correct headers and Frame Correction Sequence. Through some *ioctl* functions the network card of the PC is set in *promiscuous mode*. In this mode the Ethernet card turns off the address filter and accepts each and every incoming frame, without taking care of the destination MAC address. Otherwise, the hardware system implemented in the board would have to be synthetised for every PC it is connected to because the destination MAC address of the hardware bridge is coded in the firmware. This could be quite annoying for large systems which takes hours to be synthesized.

During the NoC API initialization, two threads are spawned, one for receiving and another for sending Ethernet frames from and to the hardware bridge. The receiving thread decodes incoming frames and places the streaming words (phits of 37bits) in the queue of the corresponding connection. When credits are received, the receiving thread adds the new credits to the credit counter associated with the transmitting queue. The transmitting thread takes the phits from the transmitting FIFO and places them into an frame. This frame will be sent when it is full of data, there is no more data in any

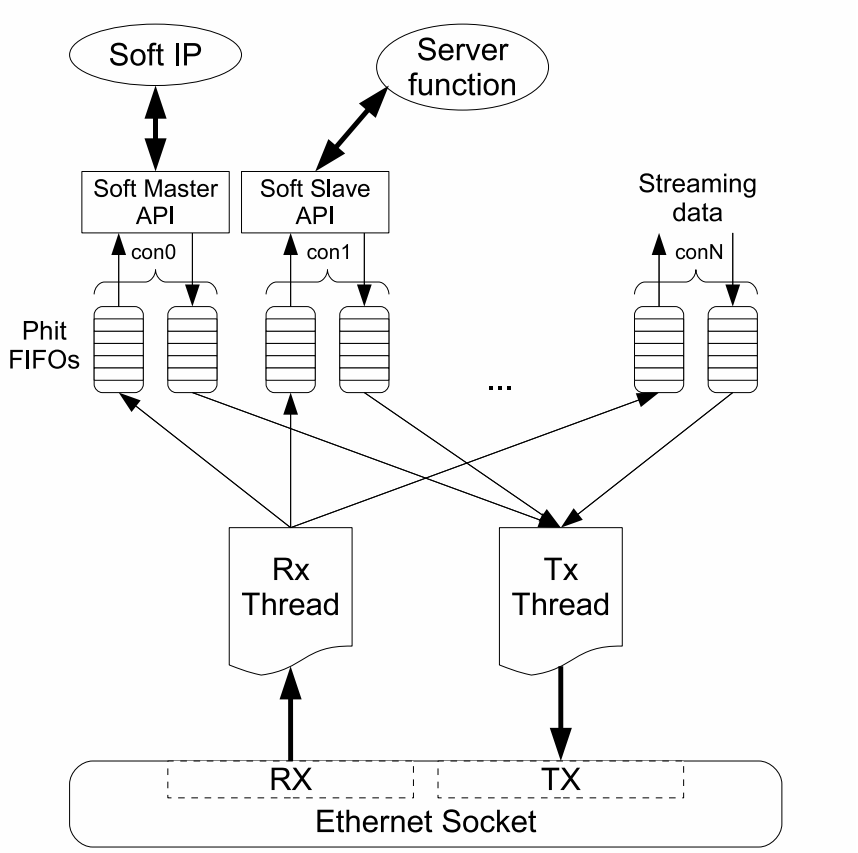


Figure 6.5: NoC API software description.

transmitting queue, or some software IP executes a *flush()*, forcing the Tx thread to send a frame. Tx thread also takes the phit counter and puts its value in the correct slot. Whenever a phit is consumed from the Rx FIFO by the memory-mapped API, the phit counter is increased, thus keeping the credit-based flow-control implemented by the hardware bridge.

Streaming communication API can be used by calling *write_phit()* and *read_phit()* functions. This functions places a phit in the transmitting queue and takes a phit out of the receiving queue, respectively. *read_phit()* can be blocking when there is no data in the Rx FIFO. *write_phit()* can also be blocking, stalling the execution of the PC program when there is not enough space in the Tx FIFO. The blocking/non-blocking behavior is controlled through a flag passed to the functions.

The streaming stack of the NoC API is used by the memory-mapped stack. Soft Master API relays in *transaction_master()* function. This function performs a transaction on the hardware system. The request message is formed by encoding the command, the size of the transaction and the address. In case of a write request, the data and mask are also placed into phits. This phits are placed into the Tx FIFO. This function also waits for the response when performing reading requests.

NoC API does not provide address decoding. Therefore, the user of the API must

specify over which connection the transaction is going to be sent or received. We choose not to implement address decoding because we believe that configuring such decoder would be more complex for the user than simply choose the right connection identifier.

Slave software IPs can be implemented thanks to *slave_serve()* function. For implementing an slave in the PC, the designer will need to call this function, giving to it the identifier of the connection. This function will block until it receives a request. The request is passed to *process_transaction()*. The designer must implement the desired functionality in this function. There is an example developed, which prints in the screen the address and value of write requests. The full NoC API is in Appendix A.

6.3 NoC Configuration from PC

One of the goals of developing an off-chip bridge was configuring the NoC from the PC. As discussed in chapter 3, the host only needs to have access to one streaming port (from now on, the remote port), placed in the local NI, and the configuration memory-mapped port of the local NI. This was previously shown in Figure 3.3(a). In order to move the host to the PC, we propose a bridge with four configuration connections (Figure 6.6) plus some other connections for user applications:

- *Connection 0:* This connection will be used to configure the bridge itself. It will be connected to an initiator shell, further connected to the configuration port of the bridge.
- *Connection 1:* This connection will be used to configure the local NI. Since the configuration port is a memory-mapped target, an initiator shell is needed.
- *Connection 2:* This connection will be forward to the remote port of the local NI. The destination of this port is the control buses of the remote NIs. By configuring the local NI, is possible to select the remote control bus to be configured.
- *Connection 3:* This connection is optional and only needed when there is a programmable bus with ports connected to the local NI. This bus will exists as long as the old host IP remains there. That is a decision of the NoC designer.
- More connections can be used for accessing another IPs, such as shared memories.

The main advantage of this scheme is that the configuration procedure of the NoC does not need to be changed. From the point of view of the local NI and the remote NI and control buses the system remains the same.

Æthereal toolchain automatically generates the *C* code that configures the NoC. This code access the NoC local and remote ports performing read and write transactions on the address space of the local or remote connections, respectively. We reuse this code by changing the low level IO functions, called *art_write()* and *art_read()*. Figure 6.7 shows the code of both options. When the host is in the FPGA, the address is masked accordingly depending if the write/read is local or remote. When the host is on a PC, the code perform a transaction to the local or the remote connection.

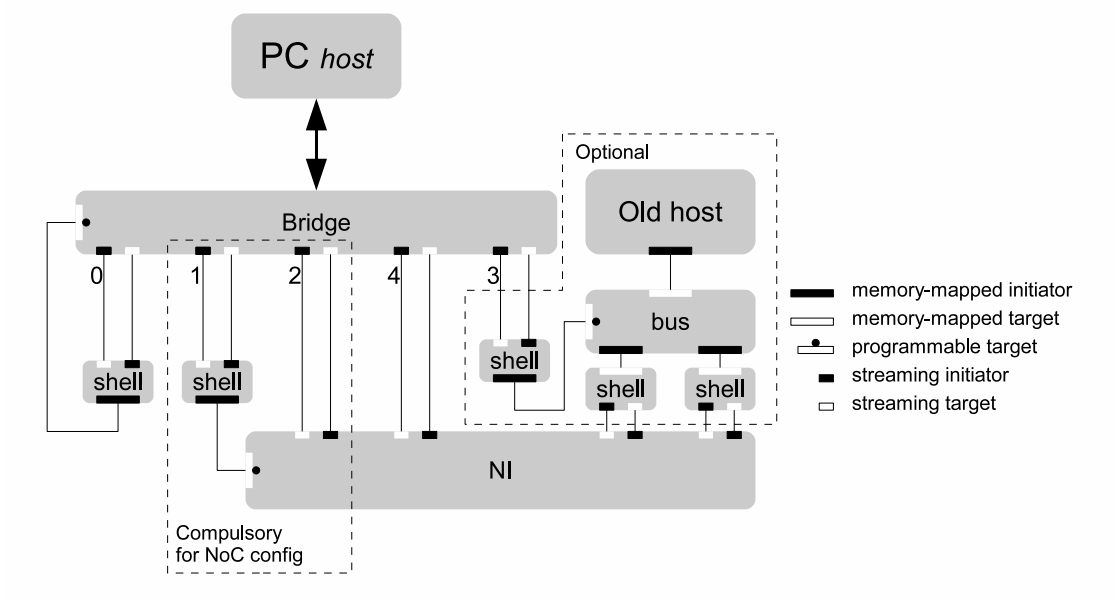


Figure 6.6: Network configuration from a PC using the bridge.

The first thing the host must do is configuring the bridge through connection 0. It should allocate at least one slot for connection 1 and one slot for connection 2. Then the host must start configuring the NoC as it was described in Section 3.4. In order to open the response channels with the remote NIs, the host has first to configure the request channel in the local NI through connection 1 and the configure the remote NI through connection 2. The API makes sure that the local NI is configured before the data is sent to the remote port.

Opening a connection between two NIs is done in the same way as in Section 3.4. The host opens the request channel that goes from the local NI to the remote NI, sets configures the response channel of the connection to be open and closes the request channel. Finally, the host opens a request channel with the other remote NI, configures a request channel between the two remote NIs, and closes the configuration request channel.

If the host wants to open a connection between a remote NI and itself, it needs to configure the request channel on connection 4 using connection 1. Finally, it must also configure the bridge to allocate the desired resources in the off-chip link. Closing a connection between the host and a remote NI requires two additional steps. The host must check that there is no data traversing the off-chip link or stored in the bridge queues (both in the software and in the hardware parts), and remove the resources allocated for that connection at the bridge.

6.4 Configuring Two Sub-NoCs

This section will describe how the configuration of two sub-NoCs is done. For simplicity we place the host in one of the sub-NoCs, but it could be a PC, however. In Figure 6.8(a)

the host is connected to the local NI of sub-NoC 1, in a similar way as in Figure 3.3(a). Therefore the configuration procedure of this sub-NoC follows the same steps as described in Section 3.4. The host must open four connections with remote NI that is connected to the bridge. Until this point, the bridge is treated like any other IP.

The bridge that is in the second sub-NoC (Figure 6.8(b)) follows the same connection pattern as the one described in Section 6.3. Port one is used for configuring the bridge, port two is for configuring the *remote_local* NI (the NI that is connected to the bridge in the second sub-NoC), and port three for the configuration of the other NIs of the second sub-NoC (*remote_remote*).

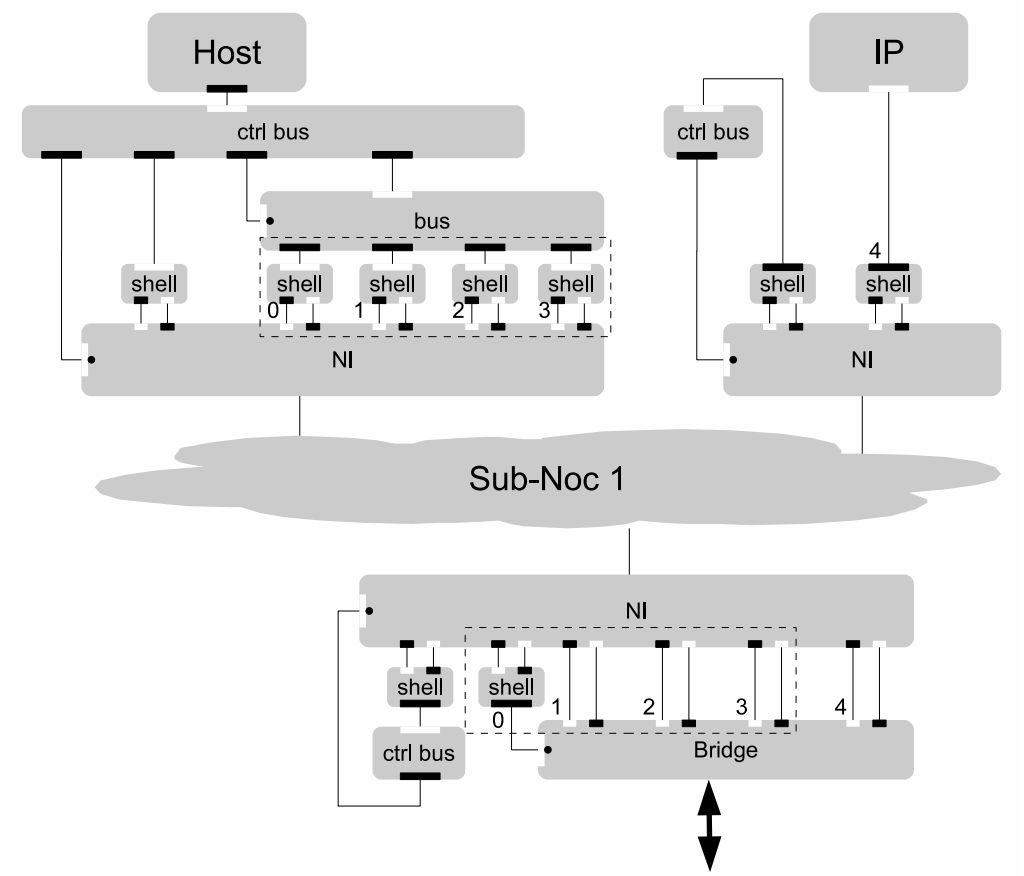
In order to configure the whole NoC (both sub-NoCs), the host must follow the next steps:

- First NoC is configured as usual. Additionally, the host sets up four connections to communicate with the bridge (connections 0, 1, 2 and 3). Connections between the other NIs and the bridge are also set (connection 4).
- With connection zero, the host can modify the slot table of the local bridge, allocating bandwidth for the request channel of connections 1, 2 and 3.
- With connection one, the host modifies the slot table of the remote bridge and allocates bandwidth for the response channel of connection 1, 2 and 3. Connection 1 needs a response channel because the credits need to come back.
- Now the host have full access to connections 2 and 3. This connections are the local port and the remote port, respectively, of the second sub-NoC. Thus, the host only has to run the configuration procedure of the second sub-NoC. The only difference is that instead of doing the write/read transactions over the local and remote ports, it does them over the *remote_local* and the *remote_remote* ports. This can be easily done by changing two masking addresses in the code.
- Finally, the host can configure the TDM table of the bridge in such a way that it opens the application connections, like connection 4. This is done at the end of the configuration process, and hence IPs of a sub-NoC can not send data to the other sub-NoC before both are well configured. This acts like a simple booting system avoiding data lost. If a connection sends data to the bridge before the configuration is finish, this data will be stalled at the bridge until it is safe to cross the off-chip link.

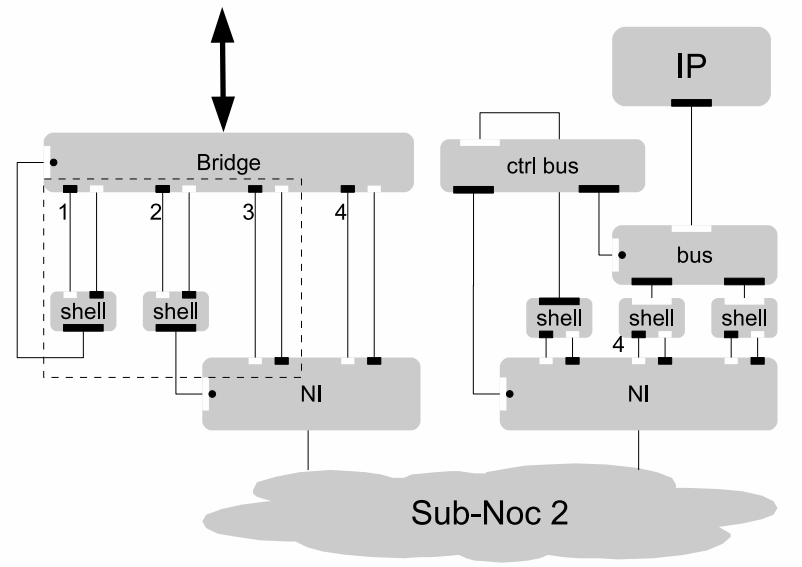
At the end of the configuration process of the system illustrated in Figure 6.8 we achieve communication between the initiator IP in sub-NoC 2 and the target IP in sub-NoC 1. This communication is carried over by connection 4. Connection 4 goes from the remote NI of sub- NoC 1 to the NI of the bridge, crosses the bridge, and the goes from the NI of the bridge in sub-NoC 2 to the remote NI of the sub-NoC 2.

```
 #define LOCAL_CONN   1
#define REMOTE_CONN 2

// "where" can be either REMOTE or LOCAL
void art_write(unsigned address, unsigned data, place_t where) {
#ifndef PC // Address based configuration
       if (where == REMOTE) {       // Add REMOTE mask
               address = (CFG_EQUAL | RMT_EQUAL | address);
       }
       else {                                       // Add LOCAL mask
               address = (CFG_EQUAL | LCL_EQUAL | address);
       }
       unsigned volatile* const addr = (unsigned * const) address;
       *addr=data;
#else // Connection based configuration
       if (where == REMOTE){
               conn->c_number = REMOTE_CONN;
       }
       else {
               conn->c_number = LOCAL_CONN;
       }
       transaction_master(conn, WR, address, (int *)&data, 0);
#endif
}

unsigned art_read(unsigned address, place_t where) {
       unsigned data;
#ifndef PC // Address based configuration
       if (where == REMOTE) {       // Add REMOTE mask
               address = (CFG_EQUAL | RMT_EQUAL | address);
       }
       else {                                       // Add LOCAL mask
               address = (CFG_EQUAL | LCL_EQUAL | address);
       }
       unsigned const volatile * const addr = (unsigned * const) address;
       data=*addr;
#else // Connection based configuration
       if (where == REMOTE){
               conn->c_number = REMOTE_CONN;
       }
       else {
               conn->c_number = LOCAL_CONN;
       }
       transaction_master(conn, RD, address, (int *)&data, 0);
#endif
   return data;
}
```

Figure 6.7: IO code for NoC configuration when the host is in an embedded processor on the FPGA or in the PC



(a)



(b)

Figure 6.8: System configuration with two sub-NoCs.

7 Test Cases

This chapter shows the results of the experiments done. In Section 7.1 the bridge is tested for streaming data traffic, and some results about the latency and performance are shown. Section 7.2 test the bridge in a simple environment were the performance in terms of transaction latency is measured. Two test cases are presented in Section 7.3 and 7.4. The first one is a SoC with a host outside of the FPGA. The second one shows a SoC divided in two different FPGAs.

7.1 Standalone Bridge Performance (Streaming)

For testing the performance of the bridge with streaming communication, the test-bench illustrated in Figure 7.1 places two bridges in the same FPGA. Traffic Generators (TG) are attached to the ports of one bridge, and Measurement Units (MU) are attached to the other. TGs injects phits in the bridge ports according with a preset rate. The type of the traffic is uniform. TGs also write a timestamp, the value of *timer*, in each phit when these are accepted by the bridge. Therefore, when a MU receives one phit, compares the timestamp with the timer value, and computes the latency for that phit.

Figure 7.2 shows the latency results for different injection rates, and different number

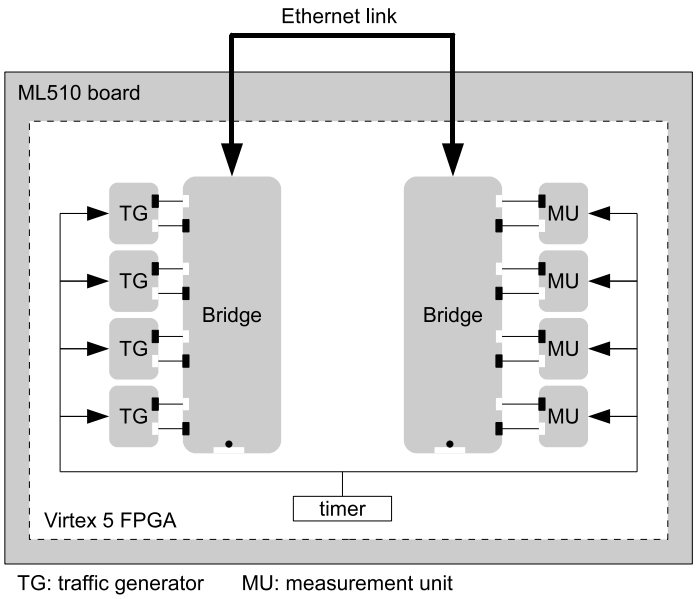


Figure 7.1: Standalone bridge for testing streaming communication.

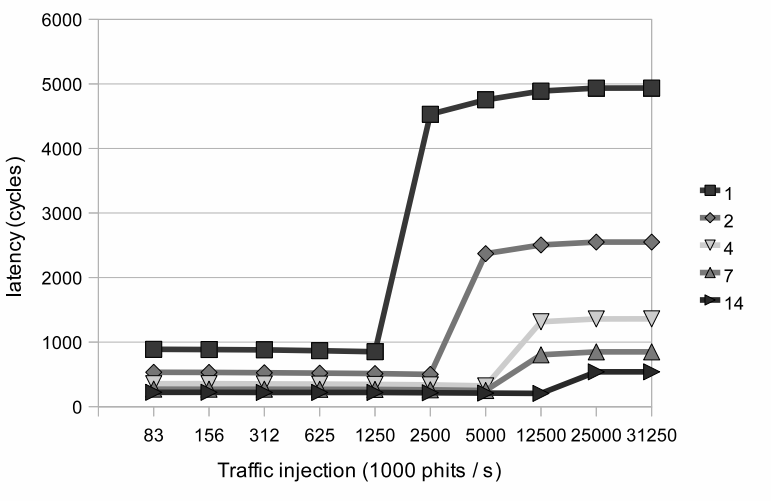


Figure 7.2: Latency of the streaming traffic.

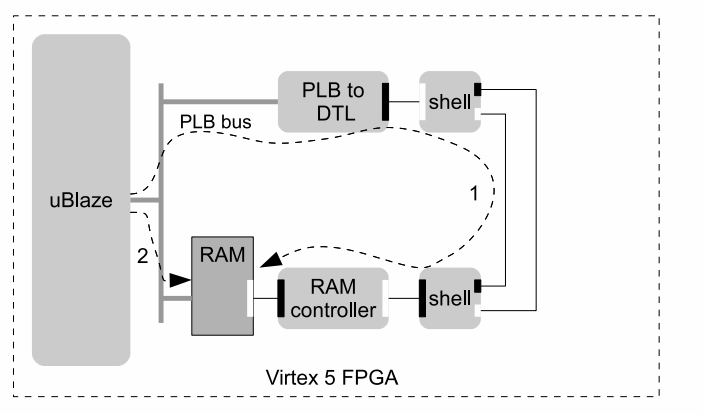
of slots assigned to the connection. The setup used Ethernet frames with a payload of 1400 bytes and slots of 100 bytes. FIFOs were 64 phits deep.

The flat segment of the graph corresponds to the throughput where the latency is guaranteed. When the number of slows assigned to one connection grows the latency starts dropping because they have to wait less time for their TDM slots. For example if a connection has only one slot assigned it is probable that when it tries to send one phit it will need to wait for its slot. On the other hand, the phits of a connections that has all the slots reserved will not need to wait for been sent. The graph also shows that when the TGs generate more traffic than the bridge can send, the latency grows dramatically. This is because there are not enough slots for carrying all the phits, and these have to wait in the FIFOs, increasing the latency of the system, and loosing the QoS.

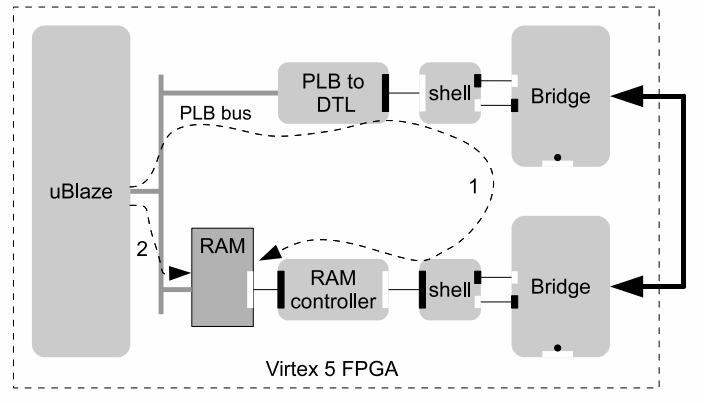
7.2 Standalone Bridge Performance (Memory mapped)

For testing how the bridge affects the performance when carrying memory-mapped communication we have designed a simplified system that only has one initiator-target pair and two IPs, along with shells, but not with the network core modules, such NIs or routers.

With this test we aim to isolate and show the latency introduced by the bridge when is placed in the interconnect. For this we have developed the systems described in Figure 7.3. A MicroBlaze (μB) processor accesses an external RAM in two different ways: directly through the PLB bus or through a simple NoC interconnect. This NoC interconnect is only has the memory-mapped and streaming stacks. System 7.3(a) is used to measure the latency introduced by other modules different from the bridge. Following path 1, the μB issues a request that is converted from the PLB protocol to the NoC bus protocol by the PLB to DTL converter. The target shell serializes the transaction and the streaming that is fed into the initiator shell directly (7.3(a)) or through the



(a)



(b)

Figure 7.3: Standalone memory-mapped control (a) and full (b) tests.

bridge (7.3(b)). Then the request is deserialized and passed to the RAM controller. If the request is a write operation the μB can know when it is done by polling the RAM directly through the PLB bus (path 2). All the blocks run with a 125 *MHz* clock.

Subtracting the latency of system (a) to the latency of system (a) we can know how much latency the bridge introduces in the transactions.

We have obtained that the latency introduced by system (a) is 50 cycles. For system (b) we have obtained the graph shown in Figure 7.4. The latency decreases when the number of slots assigned increases, as expected because the time the transaction waits for its assigned slot is shorter. However, the latency can not be lower than 237 cycles (in average) due to the gap slots in the TDM table. This gap slots can not be used because they model the time that the headers of the Ethernet packet that need to be sent (Section 5.6), and the fixed latency of the link (220 cycles).

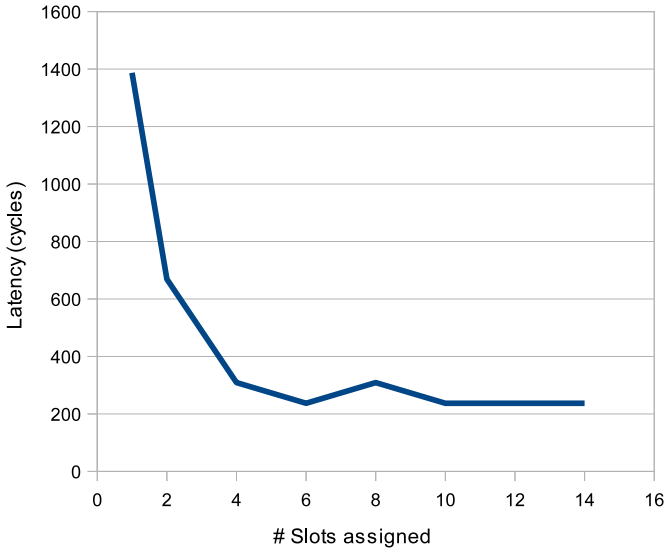


Figure 7.4: Latency of the memory-mapped traffic.

7.3 Test Case: NoC with external host

The system used for testing the bridge when connected with a PC is illustrated in Figure 7.5. This system interconnects two *tiles*, a shared memory and a microcontroller running as system monitor. Each tile has a microprocessor inside, together with instruction and data memories, three Direct Memory Access (DMA) controllers, three input external memories (*cmemin*) and three output external memories (*cmemout*). DMA modules fetch data from their *cmemouts* and writes it in the *cmemins* of the other tile. They also can read from the *cmemins* of the other tiles or the shared memory and store the data in its one *cmemout*. Therefore, DMA modules move the data between the *cmemout* of its own tile and the *cmemin* of other tile or the shared memory. Each tile has three groups of DMA controllers, *cmemout* and *cmemin*. Hence, each tile has three initiator ports and three target ports. The main memory of the system is shared by four initiators: the monitor, two tiles, and the host. All the transactions with the shared memory are arbitrated by the initiator bus.

The configuration of the network is done from an external PC. First, the host configures the bridge, allocating slots for the local connection and the remote connection. The host the opens a request channel with NI_2 by configuring NI_1. With this request channel the host has access to the configuration port of NI_2 and can configure the response channel for that connection. The host waits until the configuration data is sent by polling the credit counter of the local NI. The same process is done with NI_3 and NI_4, creating response channels for each remote *NI*. Following there is an example of this procedure:

```
// Configuring the Bridge
>setcon 0
>write 0x00 0
>write 0x01 1
```

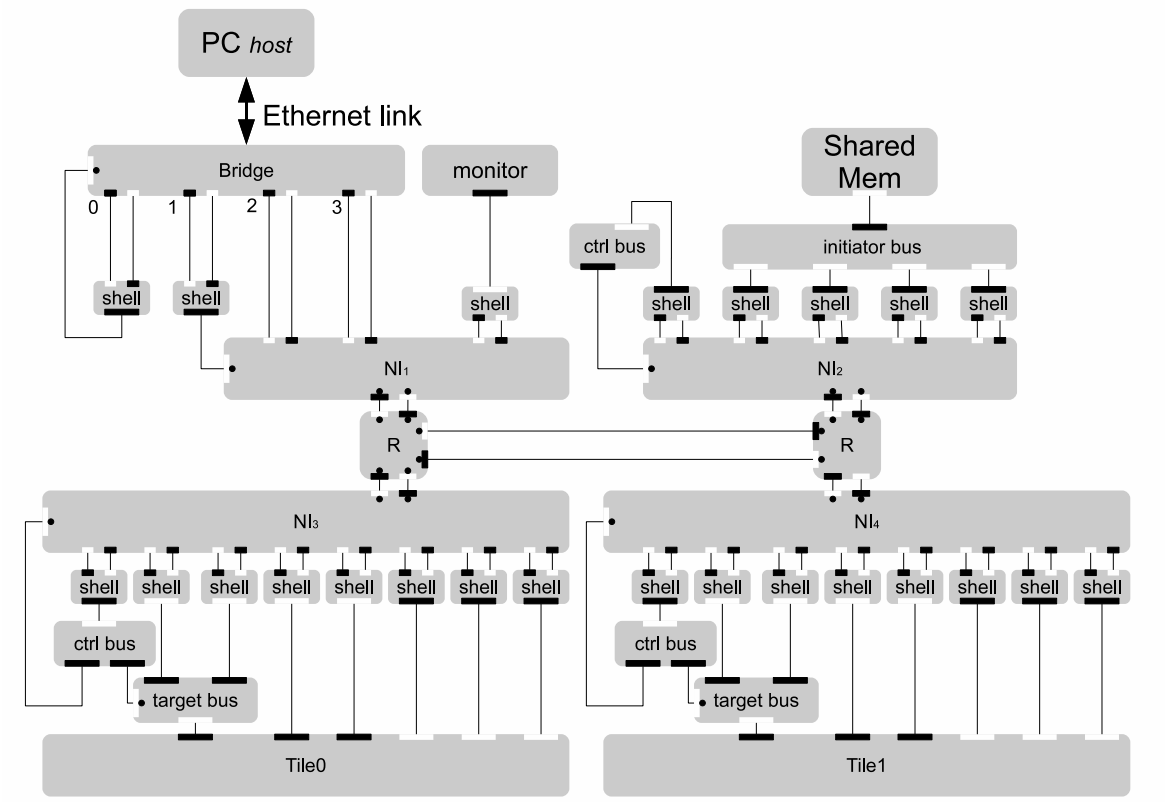


Figure 7.5: NoC with a PC as host.

```
>write 0x02 2
>write 0x03 3

// Configuring the NoC
>nocconf
// Configuring the response channels of the NIs
- Set Credits on LOCAL
Writing Address 100, connection 1 :  <- 30000009
- Set Slots on LOCAL
Writing Address 200, connection 1 :  <- 0
 --- NI 0
-Open config request channel
Writing Address 0, connection 1 :  <- c0000002
- Set Credits on REMOTE
Writing Address 100, connection 2 :  <- 30000009
- Set Slots on REMOTE
Writing Address 200, connection 2 :  <- 0
- Configuring the path
Writing Address 0, connection 2 :  <- c0000001
 --- NI 1
-Open config request channel
```

```
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000004
- Set Credits on REMOTE
Writing Address 100, connection 2 :  <- 30000009
- Set Slots on REMOTE
Writing Address 248, connection 2 :  <- 0
- Configuring the path
Writing Address 0, connection 2 :  <- c0000004
 --- NI 2
-Open config request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits on REMOTE
Writing Address 100, connection 2 :  <- 30000009
- Set Slots on REMOTE
Writing Address 248, connection 2 :  <- 0
- Configuring the path
Writing Address 0, connection 2 :  <- c0000004
```

Once all the response channels are open, the host starts configuring the connections that will provide communication services for the applications. First, the host opens the configuration request channel (from NI_1 to NI_2) and sets the credits, the slots and the path of the request channel of the application connection (from NI_2 to NI_3). Then it changes the configuration request channel to NI_3 and opens the response channel of the application connection (from NI_3 to NI_2). If needed the configuration of the bus is also possible. The host only needs to open the configuration request channel and access them with the right address.

This is an example of how a connection is opened. The full log of the configuration is in Appendix B.1.

```
// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 104, connection 2 :  <- 3000000d
- Set Slots
Writing Address 220, connection 2 :  <- 1
Writing Address 224, connection 2 :  <- 1
- Set Path
Writing Address 4, connection 2 :  <- c0000028
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 108, connection 2 :  <- 30000013
- Set Slots

Writing Address 22c, connection 2 :  <- 2
Writing Address 230, connection 2 :  <- 2
- Set Path
Writing Address 8, connection 2 :  <- c0000018
```

```
// Configuring target bus
- Open configuration request channel
- Set addr decoder
Writing Address 504, connection 2 :  <- 7ff00000
Writing Address 404, connection 2 :  <- 2000001
```

Through the fourth connection of the bridge, the host can access the shared memory. Using the command line it is possible to see and change the contents of the shared memory:

```
>setcon 3
>read 0 1
38
>write 0x00 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
>dump 0x00 20
 0: 1   4: 2   8: 3   c: 4
10: 5  14: 6  18: 7  1c: 8
20: 9  24: a  28: b  2c: c
30: d  34: e  38: f  3c: 10
40: 11 44: 12 48: 0  4c: 0
>
```

Therefore we can conclude that the configuration procedure was correct.

7.4 Test Case: NoC with two sub NoCs

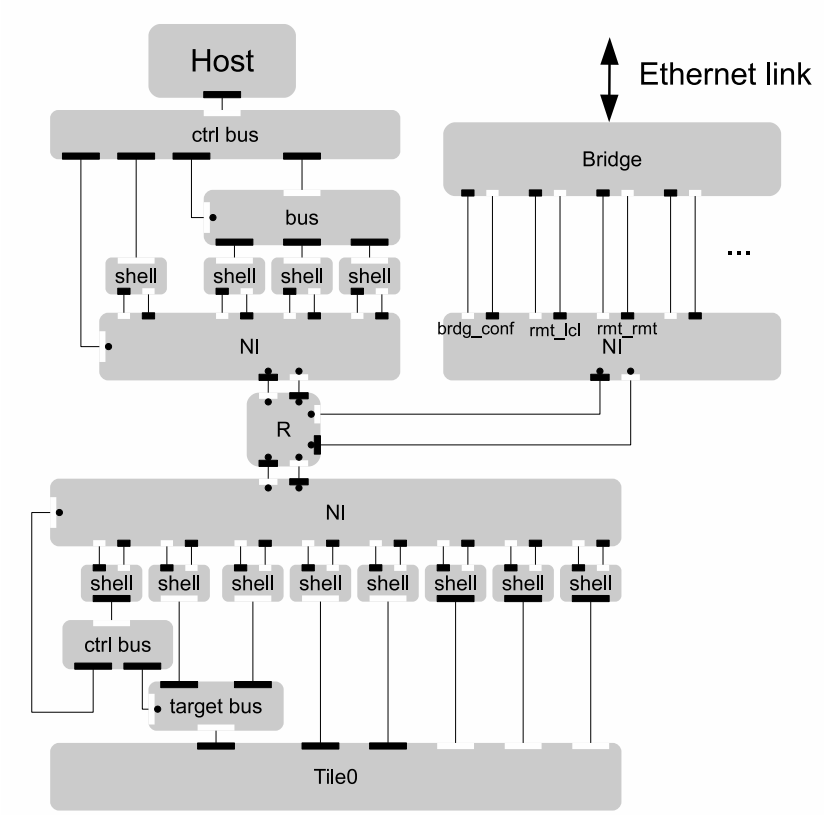
The second test case, shown in Figure 7.6, consist of two sub-NoCs connected by the bridge. Each sub-NoC has one of the tiles described in the previous section. The first sub-NoC (NoC_1) has also a microprocessor working as host. NoC_2 contains the shared memory. There are 9 connections crossing the bridge:

- Connection *brdg_conf*: Through this connection the host can configure the remote bridge, allocating the needed resources for the other connections.
- Connection *rmt_lcl*: The host uses this connection for configuring NI_4 in NoC_2. In this way the host is able to open request channels for configuring NoC_2.
- Connection *rmt_rmt*: The host uses this connection for configuring NI_5 and NI_6. In this NIs is where the request and response channels of the application connections will be configured.
- Other connections: Used for application communication.

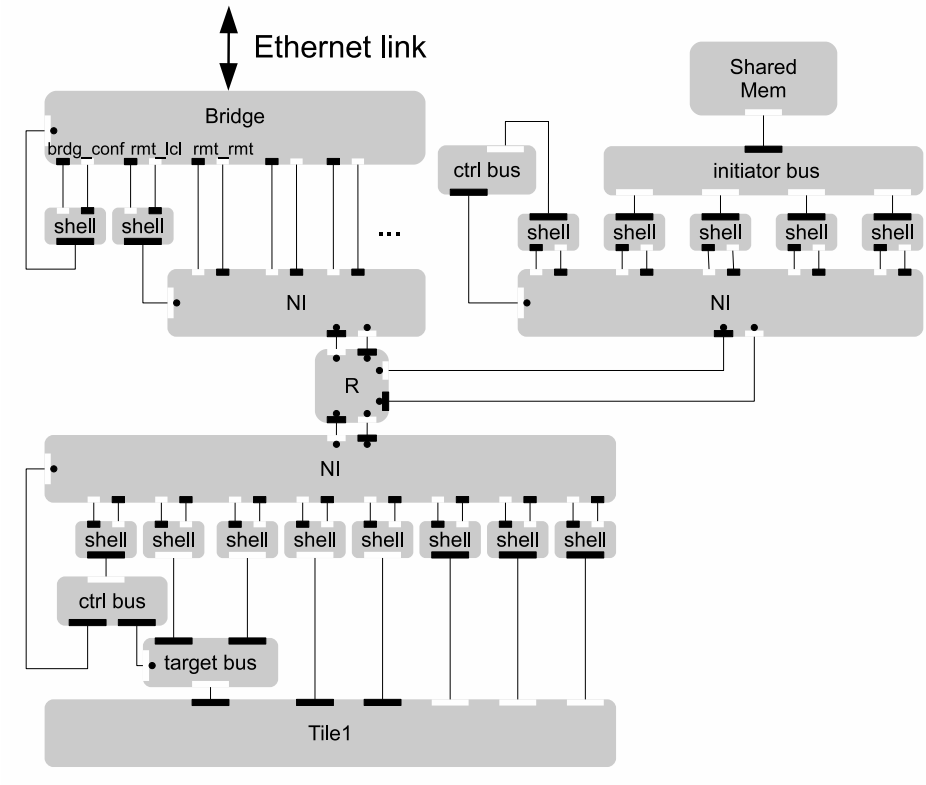
The configuration procedure of NoC_1 is the same as any single chip NoC. In this test case the bridge does not need to be configured because its TDM table was preloaded at synthesis with the correct values and therefore all its connections have the correct slots allocated. For configuring NoC_2, the host uses the same program that it would use if they were in the same chip, without NoC_1 in the middle. The only change is masking the remote and local addresses so the host uses now *rmt_lcl* instead of the local port, and *rmt_rmt* instead of the remote port.

For example, this is the opening of a request channel on NoC_1 and on NoC_2

```
Configuring NoC1...           Configuring NoC2...
- Set Credits on LOCAL
LOCAL WRITE addr 0x80000100   LOCAL  WRITE addr 0x76000100
- Set Slots on LOCAL
```



(a)



(b)

Figure 7.6: System with two sub-NoCs.

```
LOCAL WRITE addr 0x80000200   LOCAL  WRITE addr 0x76000200
 --- NI 0
-Open config request channel
LOCAL WRITE addr 0x80000000   LOCAL  WRITE addr 0x76000000
- Set Credits on REMOTE
REMOTE WRITE addr 0x80004100  REMOTE WRITE addr 0x77004100
- Set Slots on REMOTE
REMOTE WRITE addr 0x80004200  REMOTE WRITE addr 0x77004200
- Configuring the path
REMOTE WRITE addr 0x80004000  REMOTE WRITE addr 0x77004000
```

The full log of the configuration can be seen in Appendix B.2.

Once both NoCs are configured, the host can access the shared memory, and read and modify its contents, proving that the configuration procedure was correct.

8 Conclusions

In this thesis, an off-chip bridge for a Network on Chip (NoC) is proposed. This bridge allows to successfully extend the NoC over to different chips, while preserving the Quality of Service (QoS) of the applications and hiding the complexity of the off-chip interconnection from the IPs.

Several bridging schemes are described in detail characterizing them according with a well defined requirements in terms of transparency, decoupling and QoS. This systematic approach shows that the best layer for placing a bridge is the transport layer, multiplexing several connections over one off-chip link. The bridge can provide QoS thanks to the TDM scheduling at the granularity of streaming data words. The bridge also preserves end-to-end flow control by implementing local-link flow control with a credit based mechanism.

This design is implemented in RTL and tested in an FPGA. There is also a software implementation that allows to configure the NoC from a remote PC with an Ethernet card. Moreover, this thesis also implements a procedure for configuring one NoC from the outside of the chip using the bridge, and another procedure for configuring systems with a Noc divided into two subNoCs by a bridge.

Finally the bridge implementation is tested on an FPGA in standalone environment. Two other test cases are also tested: one with a NoC that is configured and debugged from the outside, and another SoC with two sub-NoCs. No one of the tiles used in those SoCs needed to be changed, only the NoC and the configuration code was different.

This work proves that is possible to implement a bridge that allows to extend a NoC over two FPGAs, while providing end-to-end QoS, transparency, and decoupling. Moreover, one possible NoC configuration method is described. This method reuses the configuration process provided by Æthereal, and only two steps more are needed to open one connection. Therefore the NoC designer can have much more logic resources at his disposal. This bridge also allows to design SoCs with companion chips expending less time in developing and verifying the interconnect. The main disadvantage is that the bridge introduces more delay significantly compared with the delay introduced by the NoC, mainly due to the delay introduced by the Ethernet link.

This work has also resulted in one research paper that will be published and presented at the *Design, Automation, and Test in Europe* DATE) conference of 2011.

8.1 Future Work

Currently, the bridge is usable and has a reasonable performance. Continuing the same vein of research, we propose some other uses or extensions for the bridge that might be interesting:

- Allow the bridge to connect with more than one board by using an Ethernet switch, increasing the amount of resources available for the designer. This must be done carefully, because if the buffers of the switch get full, ethernet frames will be dropped.
- Adding some mechanism for recovering from frames lost. For example, adding the TCP/IP layers between the bridge and the Ethernet. This would also increase the connectivity of the SoC, because the IPs of the NoC could have access to resources located on Internet. This adds great challenges because TCP protocol introduces a high latency, and can not provide QoS.

- Develop an integrated simulation and verification toolchain. Starting with a SystemC description of the SoC, it would be interesting to study the possibility of moving the system to the FPGA step by step. When the systemC simulation is working, the designer can implement the NoC and one of the IPs in the FPGA, while the other IPs run as processes in a PC. The communication between the hardware IPs and the software IPs is provided by the NoC, by the hardware bridge, and by the software bridge. This may allow faster developing and upgrading of the SoCs.
- Finding the best way of partitioning large SoCs can be challenging. Developing a tool that automatically finds the optimal place of the bridge in the NoC based in a cost model of the applications and the SoC could be interesting.
- Integrate the formal model of the bridge with the formal model of the NoC to analyze the unified systems at the design time.

Bibliography

[1] E. Beigne and P. Vivet, *Design of on-chip and off-chip interfaces for a GALS NoC architecture*, Asynchronous Circuits and Systems, 2006. 12th IEEE International Symposium on, March 2006, pp. 10 pp.–183.

[2] Luca Benini and Giovanni De Micheli, *Networks on chips: A new SoC paradigm*, IEEE Computer **35** (2002), no. 1, 70–80.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. PeperStaete, *Cyclo-static data flow*, ICASSP'95, International Conference on Acoustics, Speech, and Signal Processing, vol. 5, 1995, pp. 3255–3258.

[4] C. Chang, J. Wawrzynek, and R.W. Brodersen, *BEE2: A high-end reconfigurable computing system*, Design & Test of Computers, IEEE **22** (2005), no. 2, 114–125.

[5] C. J. Comis, *A high-speed inter-process communication architecture for FPGA-based hardware acceleration of molecular dynamics*, Master's thesis, University of Toronto, 2005.

[6] William J. Dally and Brian Towles, *Route packets, not wires: on-chip interconnection networks*, Proc. Design Automation Conference (DAC), 2001, pp. 684–689.

[7] J.D. Day and H. Zimmermann, *The OSI reference model*, Proceedings of the IEEE **71** (1983), no. 12, 1334 – 1340.

[8] Samuel Evain, Jean-Philippe Diguet, and Dominique Houzet, *NoC design flow for TDMA and QoS management in a GALS context*, EURASIP J. Embedded Syst. **2006** (2006), no. 1, 4–4.

[9] S. Furber, *Future trends in SoC interconnect*, VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on, April 2005, pp. 295 – 298.

[10] K. Goossens and A. Hansson, *The aethereal network on chip after ten years: Goals, evolution, lessons, and future*, Design Automation Conference (DAC), 2010 47th ACM/IEEE, June 2010, pp. 306 –311.

[11] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago González Pestana, Andrei Rădulescu, and Edwin Rijpkema, *A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification*, Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE) (Washington, DC, USA), IEEE Computer Society, March 2005, pp. 1182–1187.

[12] Kees Goossens, John Dielissen, and Andrei Rădulescu, *The Æthereal network on chip: Concepts, architectures, and implementations*, IEEE Design and Test of Computers **22** (2005), no. 5, 414–421.

[13] Kees Goossens, Om Prakash Gangwal, Jens Röver, and A. P. Niranjan, *Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends*, Interconnect-Centric Design for Advanced SoC and NoC (Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch, eds.), Kluwer, 2004, pp. 399–423.

[14] Andreas Hansson, *A composable and predictable on-chip interconnect*, Ph.D. thesis, Eindhoven University of Technology, June 2009.

[15] Andreas Hansson and Kees Goossens, *Trade-offs in the configuration of a network on chip for multiple use-cases*, Proc. Int'l Symposium on Networks on Chip (NOCS) (Washington, DC, USA), IEEE Computer Society, May 2007, pp. 233–242.

[16] ———, *An on-chip interconnect and protocol stack for multiple communication paradigms and programming models*, Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS), October 2009.

[17] Andreas Hansson, Mahesh Subbaraman, and Kees Goossens, *aelite: A flit-synchronous network on chip with composable and predictable services*, Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), April 2009.

[18] Andreas Hansson, Maarten Wiggers, Arno Moonen, Kees Goossens, and Marco Bekooij, *Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis*, IET Computers & Digital Techniques (2009).

[19] J. Henkel, *Closing the SoC design gap*, Computer **36** (2003), no. 9, 119 – 121.

[20] IEEE, *IEEE Std 802.3 - 2005: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, Tech. report, IEEE, 2005.

[21] Bhavani Prasad Kommineni, Rajkumar Srinivasan, Rickard Holsmark, Alf Johansson, and Shashi Kumar, *Modeling and evaluation of a NoC-Internet interface*, Swedish System on Chip Conference, Bstad, April 13-14, 2004, 2004.

[22] A.-M. Kouadri-Mostefaoui, B. Senouci, and F. Petrot, *Scalable multi-FPGA platform for networks-on-chip emulation*, Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on, July 2007, pp. 54–60.

[23] Xinyu Li and O. Hammami, *Multi-FPGA emulation of a 48-cores multiprocessor with NOC*, Design and Test Workshop, 2008. IDT 2008. 3rd International, Dec. 2008, pp. 205–208.

[24] Nobuyuki Ohba and Kohji Takano, *A SoC design methodology using FPGAs and embedded microprocessors*, Proceedings of the 41st annual Design Automation Conference (New York, NY, USA), DAC '04, ACM, 2004, pp. 747–752.

[25] Arun Patel, Christopher A. Madill, Manuel Saldana, Christopher Comis, Regis Pomes, and Paul Chow, *A scalable FPGA-based multiprocessor*, FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 2006, pp. 111–120.

[26] Philips Semiconductors, *Device Transaction Level (DTL) protocol specification. version 2.2*, July 2002.

[27] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, *Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip*, Computers and Digital Techniques, IEE Proceedings - **150** (2003), no. 5, 294–302.

[28] Andrei Rădulescu and Kees Goossens, *Communication services for networks on chip*, Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation (Shuvra S. Bhattacharyya, Ed F. Deprettere, and Jürgen Teich, eds.), Marcel Dekker, 2004, pp. 193–213.

[29] Frits Steenhof, Harry Duque, Björn Nilsson, Kees Goossens, and Rafael Peset Llopis, *Networks on chips for high-end consumer-electronics tv system architectures*, DATE '06: Proceedings of the conference on Design, automation and test in Europe (3001 Leuven, Belgium, Belgium), European Design and Automation Association, 2006, pp. 148–153.

[30] Frank Vahid and Tony D. Givargis, *Embedded system design: A unified hardware/software introduction.*

[31] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit, *Modelling run-time arbitration by latency-rate servers in dataflow graphs*, Proceedingsof the 10th international workshop on Software & compilers for embedded systems (New York, NY, USA), SCOPES '07, ACM, 2007, pp. 11–22.

[32] Xilinx Inc., *ML510 Embedded Development Platform User Guide, UG356 (v1.1)*, `http://www.xilinx.com/support/documentation/boards_and_kits/ug356.pdf`, December 2008.

[33] ———, *Processor Local Bus (PLB) v3.4*, April 2009.

[34] ———, *Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC User Guide, UG194 (v1.9)*, `http://www.xilinx.com/support/documentation/user_guides/ug194.pdf`, October 2009.

[35] ———, *Virtex-5 FPGA RocketIO GTP Transceiver User Guide, UG196 (v2.1)*, `http://www.xilinx.com/support/documentation/user_guides/ug196.pdf`, December 2009.

[36] ———, *Virtex-5 FPGA User Guide, UG190 (v5.3)*, `http://www.xilinx.com/support/documentation/user_guides/ug190.pdf`, may 2010.

[37] S. Xu and H. Pollitt-Smith, *A multi-microblaze based soc system: From systemc modeling to fpga prototyping*, Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on, June 2008, pp. 121 –127.

[38] Yaming Yin and Shuming Chen, *Design and implementation of a inter-chip bridge in a multi-core SoC*, Design & Technology of Integrated Systems in Nanoscal Era, 2009. DTIS '09. 4th International Conference on, April 2009, pp. 102–106.

Software API

This appendix shows the NoC API. This API provides communication with from a PC to a NoC that has a bridge. It implements the functionality of a bridge for sending and receiving phits, but in software instead of hardware. Since most of the modules of a SoC are memory-mapped, it also implements the conversion from DTL transactions to phits doing the same serialization as the shells. This allow the user to read and write from memories in an easy way.

```
/*
 * DATALINK API.
 * With this API is possible to send individual phits through any connection of the
 * bridge.
 */

/*Initializes the communication with the NoC
 * dl returns the data of the   datalink socket.
 * socket needs to contain an initialized ethernet socket.
 */
int init_noc_comm(t_datalink *dl, t_eth_socket *socket);

/*Closes the communication with the NoC */
void close_noc_comm(t_datalink *dl);

/*
 * Sends a phit to the noc.
 * dl is the datalink socket.
 * connection is the connection number where the phit will be sent.
 * data is a pointer to an array with phit.  It has a fixed lenght of 5 bytes.
 * first 3 bits of data[0] are not sent to the NoC
 * flags.  If is set to one, this function will stall until the phit is sent.
 */
int write_phit(t_datalink *dl, int connection, char *data, int flags);

/*
 * Receives a phit to the noc.
 * dl is the datalink socket.
 * connection is the connection number where the phit will be received.
 * data is a pointer to an array with phit.  It has a fixed lenght of 5 bytes.
 * first 3 bits of data[0] are not valid.
 * flags.  If is set to one, this function will stall until a phit is received.
 */
int read_phit(t_datalink *dl, int connection, char *data, int flags);

void print_phit(unsigned char *phit);

/*
```

```
 * BUS API.
 * With this API is possible to send or read data words to the target ports of the NoC.
 * bridge.
 */
enum command_type {WR=0, RD=1};

// DTL socket
typedef struct {
               t_datalink *dl; // Datalink socket
               int c_number; // connection over which transactions will be sent
} t_dtlbus;

extern t_dtlbus *conn;

/*
 * Returns a pointer to DTL socket.
 */
t_dtlbus *create_connection();

/*
 * Closes the DTL socket.
 */
void destroy_connection(t_dtlbus *conn);

/*
 * Initiates a transaction
 * conn is a DTL socket.
 * command can be WR or RD, for a write or read transaction.
 * address is the address of the transaction
 * data is an array that contains the words to be sent to the target, or
 * the buffer where the result of a read will be stored
 * size is the size of the transaction, in words.
 */
int transaction_master(t_dtlbus *conn, int command, unsigned address, int *data, int size);

/*
 * This function waits for a transaction.  It implements the functionallity of a target port.
 * When a transaction is received it calls the function "process_transaction".
 * After process_transaction returns this function keeps waiting for more transactions.
*/
void slave_serve(t_dtlbus *conn);

/*
   * This function is called by slave_serve whenever a request is received.
   * It is intended to be overwriten by   the user of the API. Otherwise it prints the content of the
   *   Command indicates whether the request is a read or a write.
   * data is an array   that contents the words of a write transaction, or where the words must be ret
     * case of a read transaction
     * size is the number of requested words.
   */
void process_transaction(int command, unsigned address, int *data, int size);
```

NoC Configuration

B

This appendix contains the configuration logs of the test cases.

B.1 NoC Configured from PC

```
AEthereal Command Interpreter

// Configuring the Bridge
>setcon 0
>write 0x00 0
>write 0x01 1
>write 0x02 2
>write 0x03 3

// Configuring the NoC

>nocconf

// Configuring the response channels of the NIs

- Set Credits on LOCAL
Writing Address 100, connection 1 :  <- 30000009
- Set Slots on LOCAL
Writing Address 200, connection 1 :  <- 0
 --- NI 0
-Open config request channel
Writing Address 0, connection 1 :  <- c0000002
- Set Credits on REMOTE
Writing Address 100, connection 2 :  <- 30000009
- Set Slots on REMOTE
Writing Address 200, connection 2 :  <- 0
- Configuring the path
Writing Address 0, connection 2 :  <- c0000001
 --- NI 1
-Open config request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000004
- Set Credits on REMOTE
Writing Address 100, connection 2 :  <- 30000009
- Set Slots on REMOTE
Writing Address 248, connection 2 :  <- 0
- Configuring the path
Writing Address 0, connection 2 :  <- c0000004
```

```
 --- NI 2
-Open config request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits on REMOTE
Writing Address 100, connection 2 :  <- 30000009
- Set Slots on REMOTE
Writing Address 248, connection 2 :  <- 0
- Configuring the path
Writing Address 0, connection 2 :  <- c0000004

//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000004
- Set Credits
Writing Address 104, connection 2 :  <- 30000006
- Set Slots
Writing Address 200, connection 2 :  <- 1
Writing Address 214, connection 2 :  <- 1
- Set Path
Writing Address 4, connection 2 :  <- c0000014
// From master to slave NI
- Open configuration request channel
- Set Credits
Writing Address 104, connection 1 :  <- 3000001a
- Set Slots

Writing Address 204, connection 1 :  <- 1
Writing Address 208, connection 1 :  <- 1
Writing Address 20c, connection 1 :  <- 1
Writing Address 210, connection 1 :  <- 1
Writing Address 214, connection 1 :  <- 1
- Set Path
Writing Address 4, connection 1 :  <- c0000014

art_dbg_write = 3


//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
Writing Address 108, connection 2 :  <- 30000008
- Set Path
Writing Address 8, connection 2 :  <- 40000018
// From master to slave NI
- Open configuration request channel
```

```
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 104, connection 2 :  <- 30000008
- Set Path
Writing Address 4, connection 2 :  <- 40000024
// Configuring target bus
- Open configuration request channel
- Set addr decoder
Writing Address 500, connection 2 :  <- 7ff00000
Writing Address 400, connection 2 :  <- 78000001

art_dbg_write = 35


//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 104, connection 2 :  <- 3000000d
- Set Slots
Writing Address 220, connection 2 :  <- 1
Writing Address 224, connection 2 :  <- 1
- Set Path
Writing Address 4, connection 2 :  <- c0000028
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 108, connection 2 :  <- 30000013
- Set Slots

Writing Address 22c, connection 2 :  <- 2
Writing Address 230, connection 2 :  <- 2
- Set Path
Writing Address 8, connection 2 :  <- c0000018
// Configuring target bus
- Open configuration request channel
- Set addr decoder
Writing Address 504, connection 2 :  <- 7ff00000
Writing Address 404, connection 2 :  <- 2000001

art_dbg_write = 67


//Configuring Application
```

```
// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 108, connection 2 :  <- 3000000d
- Set Slots
Writing Address 228, connection 2 :  <- 2
Writing Address 22c, connection 2 :  <- 2
- Set Path
Writing Address 8, connection 2 :  <- c0000038
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 10c, connection 2 :  <- 30000013
- Set Slots

Writing Address 234, connection 2 :  <- 3
Writing Address 238, connection 2 :  <- 3
- Set Path
Writing Address c, connection 2 :  <- c0000028

art_dbg_write = 99


//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 10c, connection 2 :  <- 30000008
- Set Path
Writing Address c, connection 2 :  <- 40000048
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 110, connection 2 :  <- 30000008
- Set Path
Writing Address 10, connection 2 :  <- 40000038

art_dbg_write = 131


//Configuring Application
```

```
// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000004
- Set Credits
Writing Address 10c, connection 2 :  <- 3000000c
- Set Slots
Writing Address 208, connection 2 :  <- 3
Writing Address 20c, connection 2 :  <- 3
- Set Path
Writing Address c, connection 2 :  <- c0000012
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 110, connection 2 :  <- 30000012
- Set Slots

Writing Address 200, connection 2 :  <- 4
Writing Address 230, connection 2 :  <- 4
- Set Path
Writing Address 10, connection 2 :  <- c000000d
// Configuring target bus
- Open configuration request channel
- Set addr decoder
Writing Address 500, connection 2 :  <- 7ff00000
Writing Address 400, connection 2 :  <- 78000001

art_dbg_write = 163


//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 114, connection 2 :  <- 3000000d
- Set Slots
Writing Address 23c, connection 2 :  <- 5
Writing Address 240, connection 2 :  <- 5
- Set Path
Writing Address 14, connection 2 :  <- c0000058
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 114, connection 2 :  <- 30000013
```

```
- Set Slots

Writing Address 204, connection 2 :  <- 5
Writing Address 208, connection 2 :  <- 5
- Set Path
Writing Address 14, connection 2 :  <- c0000058
// Configuring target bus
- Open configuration request channel
- Set addr decoder
Writing Address 504, connection 2 :  <- 7ff00000
Writing Address 404, connection 2 :  <- 1000001

art_dbg_write = 195


//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
Writing Address 118, connection 2 :  <- 3000000d
- Set Slots
Writing Address 244, connection 2 :  <- 6
Writing Address 248, connection 2 :  <- 6
- Set Path
Writing Address 18, connection 2 :  <- c0000068
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 118, connection 2 :  <- 30000013
- Set Slots

Writing Address 20c, connection 2 :  <- 6
Writing Address 210, connection 2 :  <- 6
- Set Path
Writing Address 18, connection 2 :  <- c0000068

art_dbg_write = 227


//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000002
- Set Credits
```

```
Writing Address 11c, connection 2 :  <- 30000008
- Set Path
Writing Address 1c, connection 2 :  <- 40000078
// From master to slave NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000008
- Set Credits
Writing Address 11c, connection 2 :  <- 30000008
- Set Path
Writing Address 1c, connection 2 :  <- 40000078

art_dbg_write = 259


//Configuring Application

// From slave to master NI
- Open configuration request channel
Reading Address 100, connection 1
Writing Address 0, connection 1 :  <- c0000004
- Set Credits
Writing Address 110, connection 2 :  <- 30000006
- Set Slots
Writing Address 218, connection 2 :  <- 4
Writing Address 21c, connection 2 :  <- 4
- Set Path
Writing Address 10, connection 2 :  <- c0000024
// From master to slave NI
- Open configuration request channel
- Set Credits
Writing Address 108, connection 1 :  <- 3000001e
- Set Slots

Writing Address 218, connection 1 :  <- 2
Writing Address 21c, connection 1 :  <- 2
Writing Address 220, connection 1 :  <- 2
Writing Address 224, connection 1 :  <- 2
Writing Address 228, connection 1 :  <- 2
- Set Path
Writing Address 8, connection 1 :  <- c0000044

art_dbg_write = 291

>setcon 3
>read 0 1

38
>write 0x00 0x1000
>write 0x04 0xa
>write 0x08 0x445
```

```
>write 0x10 0x3e8

>dump 0x00 6
0: 1000
4: a
8: 445
c: 0
10: 3e8
14: 0

>write 0x00 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
>dump 0x00 20
0: 1
4: 2
8: 3
c: 4
10: 5
14: 6
18: 7
1c: 8
20: 9
24: a
28: b
2c: c
30: d
34: e
38: f
3c: 10
40: 11
44: 12
48: 0
4c: 0
>
```

B.2 NoC with two sub NoCs

```
Configuring NoC1...
init aethereal
- Set Credits on LOCAL
LOCAL WRITE addr 0x80000100
- Set Slots on LOCAL
LOCAL WRITE addr 0x80000200
 --- NI 0
-Open config request channel
LOCAL WRITE addr 0x80000000
- Set Credits on REMOTE
REMOTE WRITE addr 0x80004100
- Set Slots on REMOTE
REMOTE WRITE addr 0x80004200
- Configuring the path
```

```
REMOTE WRITE addr 0x80004000
 --- NI 1
-Open config request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits on REMOTE
REMOTE WRITE addr 0x80004100
- Set Slots on REMOTE
REMOTE WRITE addr 0x80004200
- Configuring the path
REMOTE WRITE addr 0x80004000
CFG INIT COMPLETE

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x80004104
- Set Slots
REMOTE WRITE addr 0x8000420C
REMOTE WRITE addr 0x80004210
- Set Path
REMOTE WRITE addr 0x80004004
// From master to slave NI
- Open configuration request channel
- Set Credits
LOCAL WRITE addr 0x80000104
- Set Slots

LOCAL WRITE addr 0x80000204
LOCAL WRITE addr 0x80000208
LOCAL WRITE addr 0x8000020C
LOCAL WRITE addr 0x80000210
LOCAL WRITE addr 0x80000214
LOCAL WRITE addr 0x80000218
- Set Path
LOCAL WRITE addr 0x80000004
// Configuring target bus
- Open configuration request channel
- Set addr decoder
LOCAL WRITE addr 0x80000D00
LOCAL WRITE addr 0x80000C00

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x80004108
```

```
- Set Path
REMOTE WRITE addr 0x80004008
// From master to slave NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x80004104
- Set Path
REMOTE WRITE addr 0x80004004
// Configuring target bus
- Open configuration request channel
- Set addr decoder
REMOTE WRITE addr 0x80004500
REMOTE WRITE addr 0x80004400

//Configuring Application

// From slave to master NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x8000410C
- Set Slots
REMOTE WRITE addr 0x80004224
REMOTE WRITE addr 0x80004228
- Set Path
REMOTE WRITE addr 0x8000400C
// From master to slave NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x80004108
- Set Slots

REMOTE WRITE addr 0x8000424C
REMOTE WRITE addr 0x80004250
- Set Path
REMOTE WRITE addr 0x80004008
// Configuring target bus
- Open configuration request channel
- Set addr decoder
REMOTE WRITE addr 0x80004504
REMOTE WRITE addr 0x80004404

//Configuring Application
```

```
// From slave to master NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x80004110
- Set Path
REMOTE WRITE addr 0x80004010
// From master to slave NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x8000410C
- Set Path
REMOTE WRITE addr 0x8000400C

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x80004110
- Set Slots
REMOTE WRITE addr 0x80004254
REMOTE WRITE addr 0x80004258
- Set Path
REMOTE WRITE addr 0x80004010
// From master to slave NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x80004114
- Set Slots

REMOTE WRITE addr 0x80004204
REMOTE WRITE addr 0x80004208
- Set Path
REMOTE WRITE addr 0x80004014

//Configuring Application

// From slave to master NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
```

```
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x80004114
- Set Path
REMOTE WRITE addr 0x80004014
// From master to slave NI
- Open configuration request channel
LOCAL READ addr 0x80000100
data = 0
LOCAL WRITE addr 0x80000000
- Set Credits
REMOTE WRITE addr 0x80004118
- Set Path
REMOTE WRITE addr 0x80004018

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x8000411C
- Set Slots
REMOTE WRITE addr 0x80004214
REMOTE WRITE addr 0x80004218
- Set Path
REMOTE WRITE addr 0x8000401C
// From master to slave NI
- Open configuration request channel
- Set Credits
LOCAL WRITE addr 0x80000108
- Set Slots

LOCAL WRITE addr 0x8000021C
LOCAL WRITE addr 0x80000220
LOCAL WRITE addr 0x80000224
LOCAL WRITE addr 0x80000228
LOCAL WRITE addr 0x8000022C
LOCAL WRITE addr 0x80000230
- Set Path
LOCAL WRITE addr 0x80000008
// Configuring target bus
- Open configuration request channel
- Set addr decoder
LOCAL WRITE addr 0x80000D04
LOCAL WRITE addr 0x80000C04

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
```

```
REMOTE WRITE addr 0x80004120
- Set Slots
REMOTE WRITE addr 0x8000421C
REMOTE WRITE addr 0x80004220
- Set Path
REMOTE WRITE addr 0x80004020
// From master to slave NI
- Open configuration request channel
- Set Credits
LOCAL WRITE addr 0x8000010C
- Set Slots

LOCAL WRITE addr 0x80000234
LOCAL WRITE addr 0x80000238
LOCAL WRITE addr 0x8000023C
LOCAL WRITE addr 0x80000240
LOCAL WRITE addr 0x80000244
LOCAL WRITE addr 0x80000248
- Set Path
LOCAL WRITE addr 0x8000000C
// Configuring target bus
- Open configuration request channel
- Set addr decoder
LOCAL WRITE addr 0x80000D08
LOCAL WRITE addr 0x80000C08
confdone
0 40404040 0 0
Configuring NoC2...
init aethereal noc 2
- Set Credits on LOCAL
LOCAL  WRITE addr 0x76000100
- Set Slots on LOCAL
LOCAL  WRITE addr 0x76000200
 --- NI 0
-Open config request channel
LOCAL  WRITE addr 0x76000000
- Set Credits on REMOTE
REMOTE WRITE addr 0x77004100
- Set Slots on REMOTE
REMOTE WRITE addr 0x77004200
- Configuring the path
REMOTE WRITE addr 0x77004000
 --- NI 1
-Open config request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits on REMOTE
REMOTE WRITE addr 0x77004100
- Set Slots on REMOTE
REMOTE WRITE addr 0x77004200
```

```
- Configuring the path
REMOTE WRITE addr 0x77004000
CFG INIT 2 COMPLETE

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
LOCAL  WRITE addr 0x76000104
- Set Slots
LOCAL  WRITE addr 0x76000208
- Set Path
LOCAL  WRITE addr 0x76000004
// From master to slave NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x77004104
- Set Slots

REMOTE WRITE addr 0x77004204
REMOTE WRITE addr 0x77004208
- Set Path
REMOTE WRITE addr 0x77004004

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
LOCAL  WRITE addr 0x76000108
- Set Path
LOCAL  WRITE addr 0x76000008
// From master to slave NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x77004108
- Set Path
REMOTE WRITE addr 0x77004008

//Configuring Application

// From slave to master NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits
REMOTE WRITE addr 0x77004104
- Set Slots
REMOTE WRITE addr 0x7700420C
```

```
- Set Path
REMOTE WRITE addr 0x77004004
// From master to slave NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits
REMOTE WRITE addr 0x7700410C
- Set Slots

REMOTE WRITE addr 0x7700420C
- Set Path
REMOTE WRITE addr 0x7700400C

//Configuring Application

// From slave to master NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits
REMOTE WRITE addr 0x77004108
- Set Path
REMOTE WRITE addr 0x77004008
// From master to slave NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits
REMOTE WRITE addr 0x77004110
- Set Path
REMOTE WRITE addr 0x77004010

//Configuring Application

// From slave to master NI
- Open configuration request channel
- Set Credits
REMOTE WRITE addr 0x77004114
- Set Slots
REMOTE WRITE addr 0x77004210
- Set Path
REMOTE WRITE addr 0x77004014
// From master to slave NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
```

```
- Set Credits
REMOTE WRITE addr 0x7700410C
- Set Slots

REMOTE WRITE addr 0x77004204
- Set Path
REMOTE WRITE addr 0x7700400C

//Configuring Application

// From slave to master NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits
REMOTE WRITE addr 0x77004118
- Set Path
REMOTE WRITE addr 0x77004018
// From master to slave NI
- Open configuration request channel
LOCAL  READ addr 0xF6000100
data = 0
LOCAL  WRITE addr 0x76000000
- Set Credits
REMOTE WRITE addr 0x77004110
- Set Path
REMOTE WRITE addr 0x77004010
confdone
```