CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Design and Implementation of a Network-on-Chip based Embedded System-on-Chip

Master's Thesis in Embedded Electronic Systems Design

Panagiotis Strikos

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Design and Implementation of a Network-on-Chip based Embedded System-on-Chip

Panagiotis Strikos

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Design and Implementation of a Network-on-Chip based Embedded System-on-Chip

Panagiotis Strikos

Supervisor: Ioannis Sourdis, Department of Computer Science and Engineering
Co-Supervisor: Martin Rönnbäck, Cobham Gaisler
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2021

Design and Implementation of a Network-on-Chip based Embedded System-on-Chip

PANAGIOTIS STRIKOS
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Today, the demand for more computing power has led embedded computing systems to become more complex than ever. As a result, a wide range of multi and many-core System-on-Chip (SoC) architectures has been proposed. Traditionally, the bus has been used as an interconnection mechanism in many embedded systems, including the space domain. As the need for extensive processing rises though, and while multi and many-core architectures become a necessity, the bus often fails to accommodate the communication needs of such systems. By lacking the ability to scale well, buses introduce a bottleneck in the communication needs of the system's throughput. On the other hand, Networks-on-Chip (NoC) have emerged to become a paradigm for complex architectures, since they offer a scalable communication solution, serving as a replacement to the traditional bus-based interconnections. This thesis studies the upgrade of a bus-based embedded System-on-Chip by replacing its AMBA 2.0 AHB bus with an existing Network-on-Chip. To achieve that, a network interface is designed, a unit responsible for communicating with both the AHB components and the NoC, while leaving the original functionality of the systems intact. An in-depth analysis of a network interface is performed, and at the same time, a modified NoC-based version of the systems is presented featuring FastTrackNoC routers. Our evaluation shows that compared to the baseline bus-based System-on-Chip, the NoC-based one, improves communication latency from 44% and up to 97%, while resulting in a $1.68\times - 37.5\times$ higher throughput. At the same time, the proposed system increases the area overhead by a factor of $7\times - 72\times$. Although the system was only analyzed in simulation, it also has the potential to be implemented in hardware, as RTL descriptions for both the NoC and the SoC have been developed.

Keywords: System-on-Chip, Network-on-Chip, Network Interface, AMBA, AHB, FastTrackNoC, On-Chip Interconnect

Acknowledgements

First of all, I would like to thank my supervisor at Chalmers, Ioannis Sourdis, for suggesting the topic, and for his guidance and invaluable advice over the past few months. My sincere thanks also go to my supervisor at Cobham Gaisler, Martin Rönnbäck, for his help and feedback, and to Jan Andresson, the experience of whom proved to be pivotal in the technical decisions of this thesis. Last but not least, I would like to thank my family and friends. There are no proper words to convey my deep gratitude for their endless support and belief in me.

Panagiotis Strikos, Gothenburg, October 2021

Contents

List of Figures

List of Tables

Abbreviations

AHB Advanced High-performance Bus
AMBA Advanced Microcontroller Bus Architecture
AXI Advanced eXtensible Interface
DDR Dual Data Rate
FIFO First In, First Out
FSM Finite State Machine
HOL Head-of-line
IP Intellectual Property
NI Network Interface
NoC Network-on-Chip
NRC Next-Route Calculation
SDR Single Data Rate
SoC System-on-Chip
VC Virtual Channels
ZLL Zero-load latency

1

Introduction

Interconnects are critical for the performance and efficiency of every System-on-Chip (SoC). Many embedded SoCs today still use a bus for communication, since the simplicity of the traditional SoC on-chip interconnects offers the advantage of simplifying analyses (e.g. for real time properties).

Buses, however, do not scale well to the number of components used in the system [3]. Buses like the Advanced Microcontroller Bus Architecture (AMBA) 2.0 Advanced High-performance Bus (AHB) [2] limit the number of masters that can be connected to a single bus, effectively setting a ceiling to the size of a SoC. In addition, as the number of masters grows, so does the interference between them when competing for the shared bus resources.

To mitigate these effects, the bus system can be split up into multiple buses, connected through bridges [2], which allows scaling the number of masters in the system, but at a cost of complexity and reduced predictability. Furthermore, the bridges themselves might also become bottlenecks in the system, due to several masters trying to access them to gain access to another bus, thus reducing the overall performance [3]. Therefore, using a Network-on-Chip (NoC) instead of a bus, has the potential to improve the SoC communication, and enhance the system's performance, efficiency and scalability.

1.1 Goals and Challenges

This thesis investigates the upgrade of a bus-based embedded System-on-Chip by replacing its AMBA 2.0 AHB bus with an existing Network-on-Chip. Within the thesis, a Network Interface (NI) is designed and implemented to allow the integration of the NoC to the SoC.

The proposed NI along with the NoC, compared to the baseline system, aims to offer the same functionality with better performance and scalability, within reasonable area costs. To achieve that, we identify different operations, that need to be treated separately by the proposed interface, in an attempt to reduce communication latency and improve throughput.

The greatest challenge while designing the NI is for the system to maintain all of the bus' features and capabilities, without requiring to alter the connected Intellectual Property (IP) cores.

Furthermore, it may be challenging for the suggested NI to display a high level of flexibility and adaptability in altering its parameters to be able to keep up with any possible changes to either the NoC's or the bus' parameters.

1.2 Approach

An embedded bus-based SoC will be used as a baseline and its bus will be replaced with a given NoC. Our starting point will be an implementation of the NoC described in [4]. From that point, we maintained the core features that were suitable for the system at hand and tailored its various parameters.

The system is suitable for featuring Gaisler's 32-bit LEON3 [5], a processor that implements the SPARC V8 architecture, and is part of Gaisler's open-source GRLIB IP library [1]. Other than the processor, the baseline design also includes an AMBA AHB 2.0 bus, which is replaced with the NoC, a memory controller, and serial communication interfaces. After simulating the system, the NoC's parameters will be tuned in order to suit the needs of the particular SoC at hand and its components.

Following completion, measurements will be taken to evaluate the designed system. Using benchmarking, the performance of the new NoC-based SoC will be evaluated using CAD, and activity reports based on simulations. Moreover, the resource utilization and timing impact of the NoC-based system will be analyzed and compared to those of the original bus-based SoC. Finally, the performance of the developed NI will be analyzed in terms of latency and throughput.

1.3 Related Work

Several studies have explored the NI implementation for a Network-on-Chip [6, 7, 8, 9]. Those designs were optimized for performance, but often introduced an overhead too high to be applicable to real world systems [10]. More recent research on NIs aims at integrating the necessary networking functionalities while restricting the NI area, power and latency [11].

A number of interfaces have been proposed in the literature with features such as handling out-of-order transaction [12, 10, 13, 14], fault tolerance [15, 16], and Quality-of-Service (QoS) [11].

For the communication between the NI and the IP cores, various protocols have been used with the Open Core Protocol (OCP) [17, 18] and the STBus [19] being among them. The majority of the published work though, uses the AMBA Advanced eXtensible Interface (AXI) [20, 14], a protocol used in many Systems-on-Chip, which offers backward compatibility to the NoC [12].

Saponara et al. present an NI that supports the AMBA AXI bus protocol, as well as STBus TYPE 3 [11]. Their NI also supports error and power management, ordering handling, security, QoS management, programmability, end-to-end protocol interoperability, and remapping.

Radulescu et al. present an efficient, network independent, on-chip NI that supports shared memory abstraction, and flexible network configuration [10]. Their NI is compatible with existing bus protocols (AXI, OCP, DTL).

An NI that improves resource utilization and increases memory parallelism is presented by Ebrahimi et al [21]. The interface provides dynamic buffer allocation, and is compatible with the AMBA AXI protocol which allows backward compatibility with existing IP cores.

Another NI was designed to use Ping Pong buffers, and managed to increase the throughput between the router and the processing core [22].

Finally, a low power NI was designed by switching the entire asynchronous First In, First Out (FIFO) the system used, based on the traffic conditions between the IP core and the NI [23].

Our contribution is an NI which, potentially, can be suitable for space applications. The design improves greatly the latency and throughput, while keeping the area overhead in acceptable levels. At the same time, the implementation of the proposed system, brings together for the first time, Gaisler's AMBA bus with the high-end HighwayNoC.

1.4 Outline

Chapter 2 introduces the basic concepts behind the AMBA bus architecture, and particularly the AHB buses. Furthermore, concepts of Networks-on-Chip are explained. That chapter, provides the reader with the background needed for the rest of this report.

Chapter 3 presents the NI design. The decisions that were made during the design process, are discussed and analyzed in detail.

Chapter 4 presents the simulation-based evaluation of the implemented system with the proposed NI. Simulation results regarding the system's latency, throughput and area are gathered and compared to the baseline's values.

Chapter 5 completes the thesis with the conclusions drawn from the results, and a discussion about possible future work.

2

Background

This chapter is an introduction to the basic concepts presented in this work. This way, the decisions that were made during the design of the network interface (NI), which is the main topic of this work, can be later discussed in detail.

In Section 2.1, the essentials of the AMBA buses are showed, and in particular the AHB 2.0 bus protocol, that was part of the baseline system. Section 2.2 includes information about GRLIB, Gaisler's open source library that was used for every aspect of the bus-based system. Finally, in Section 2.3, Networks-on-Chip are discussed along with their most important attributes.

2.1 AMBA Bus Architecture

AMBA (Advanced Microcontroller Bus Architecture) [2], is a standard for on-chip interconnect specifications, provided freely by ARM. AMBA is designed to support high-performance communication between various blocks in a System-on-Chip design. AHB (Advanced High-Performance Bus), part of AMBA 2.0, targets high-performance, and high clock frequency system modules.

A typical AHB-based SoC can be seen in Figure 2.1. The number of masters and slaves can vary on different systems. Still, the maximum number of components is limited, and in fact, no more than 16 masters and 16 slaves can be connected to a single bus, a fact that often leads to a bottleneck for the system's scalability.

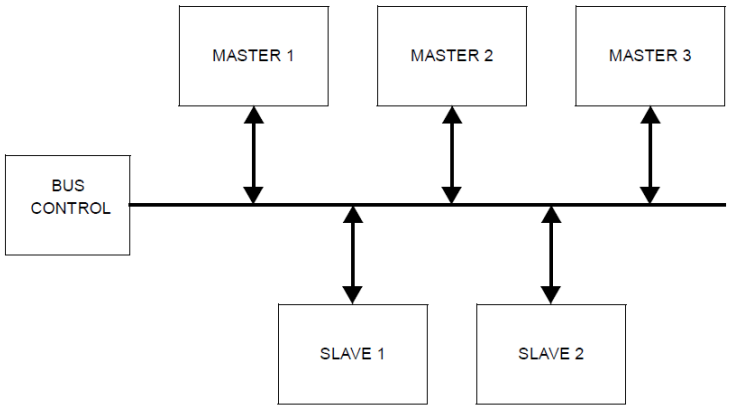


Figure 2.1: An AHB-based SoC with 3 masters and 2 slaves [1, p. 66].

2.1.1 AMBA AHB Components and Interconnection

A typical AMBA AHB design contains the following components:

AHB Master

A master is a unit that has the ability to initiate transactions, by providing an address and control signals. Due to the nature of AMBA, only one master can access the bus at any time. The interface of an AHB master is presented in Figure 2.2.

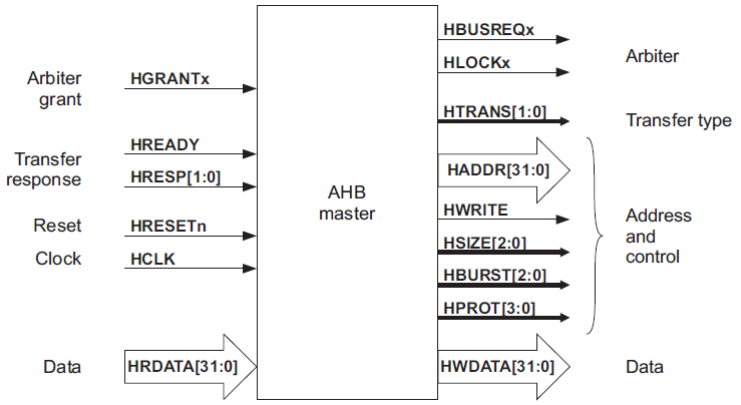


Figure 2.2: AHB master interface [2, p. 3-49].

AHB Slave

A slave responds, when needed, to requests generated from an active master. The slave might return requested data from a single, or a range of given addresses. It also informs the master whether the operation was completed successfully or not. Figure 2.3 presents the interface of an AHB slave.

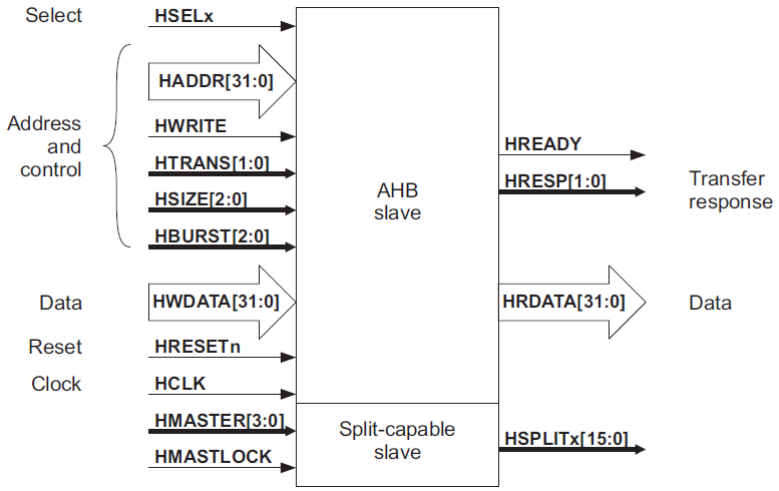


Figure 2.3: AHB slave interface [2, p. 3-45].

AHB Arbiter

The arbiter has control over which unit accesses the bus at any given time. The arbiter monitors different requests and decides which one should be prioritized. The arbitration protocol is fixed in any system, but the designer can select between a number of arbitration algorithms, such as the highest priority algorithm or the Round Robin.

AHB Decoder

This unit is decoding data on the data bus, identifying the master that the transaction is intended for.

Figure 2.4 illustrates the interconnection of the previously mentioned system with 3 masters and 2 slaves.

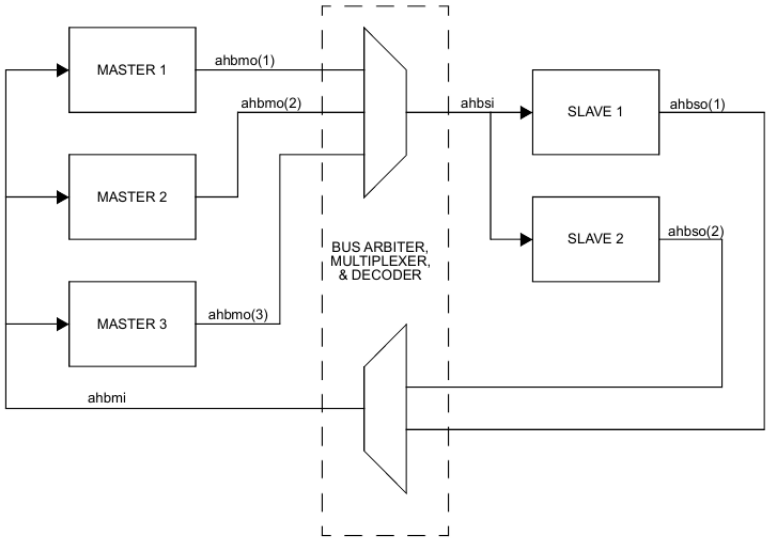


Figure 2.4: AHB Interconnection view for a system with 3 masters and 2 slaves [1, p. 67].

Two multiplexers are used, one for the addresses, and one for the data. All master units drive the address, the data, and a set of control signals, all grouped together in a VHDL record. The Arbiter selects which one will reach the slaves' inputs. In a similar manner, all slaves drive the data read signal to another record. The output of the active slave is selected by the decoder, and is driven to the master that initiated the operation.

2.1.2 AHB Operation

For any master to be able to access the bus, it must first send a request, and be granted access to it by the arbiter. Once the master has access, it can initiate an operation, which can be broken down to the following two phases:

- Address Phase: in this first part of an operation, the master drives the address bus with the address that is required for the operation, along with the necessary control signals that the slave will require.
- Data Phase: here, any necessary data are provided through the write data bus. This phase can last from just a cycle, to many more. If it is required from the slave to respond, it will do so by driving the read data bus. This phase can be extended from the slave by driving the HREADY signal low. This way the slave adds a waiting state, postpones the operation and obtains extra time to prepare and respond.

Figure 2.5 demonstrates a simple bus transfer without any waiting states. The master drives the HADDR signal with the address, and the control signals, after the clock's first rising edge, for the slave to sample them in the second rising edge. When the slave is ready, it provides a response with the requested data, through the HRDATA signal which is sampled in the third rising edge.

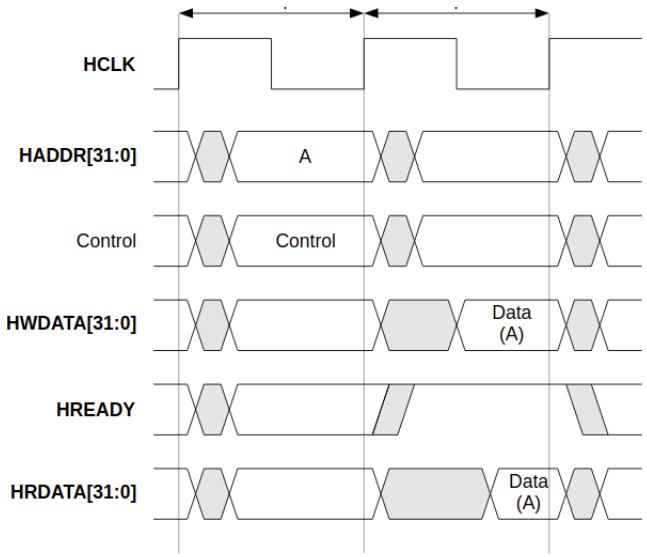


Figure 2.5: Bus transfer without waiting states [2, p. 3-6].

Another example can be seen in Figure 2.6, where the slave uses the HREADY signal in the second and third cycle to postpone the operation.

2.1.3 AHB Signals

The AHB signals are presented in Table A.1 [2] in the Appendix. The common H prefix of the signals hints that they are AHB related, and it differentiates them with similar signals of various AMBA protocols.

2.2 The GRLIB Library

The GRLIB IP Library [1, 5] is an integrated set of reusable IP cores, designed for System-on-Chip development. The library is developed by Cobham Gaisler and

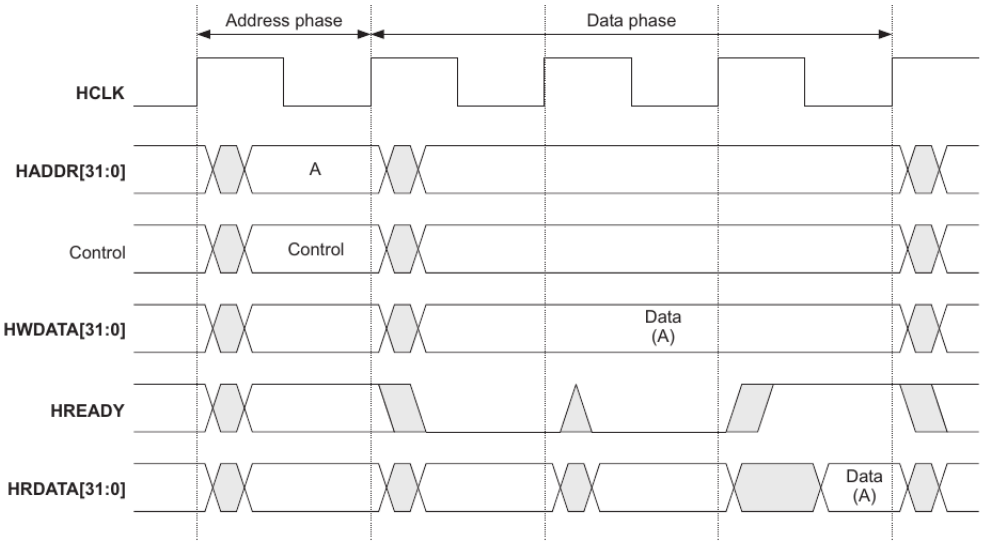


Figure 2.6: Bus transfer with waiting. By lowering the HREADY signal, the slave prolongs the duration of the Data Phase [2, p. 3-7].

provided under the GNU GPL license. The IP cores are centered around a common on-chip bus, and use a coherent method for simulation and synthesis. The library is vendor independent, with support for different CAD tools and target technologies. A unique plug&play method is used to configure and connect the IP cores without the need to modify any global resources.

The library offers a number of different cores, among which is the AHB controller, SRAM controller, 16/32/64-bit DDR/DDR2 controllers and the NOEL-V RISC-V processor.

For this thesis, a baseline system was built by Gaisler which included the GRLIB AHB controller as well as a number of AHB masters and AHB slaves.

2.3 Networks-on-Chip

This section introduces Network-on-Chips (NoCs), and provides the reader with the necessary theoretical background. The concepts and terminology presented here, will be later used when describing our proposed system.

Although buses are traditionally used in SoCs, they are often source of performance bottlenecks in systems with numerous cores. NoCs offer a promising on-chip communication infrastructure that can result in a faster, less power consuming system. The network's architecture, is based on a number of nodes, interconnected with each other with physical links. The most important parameters of a NoC, are packet format, topology, routing, and flow control [4] as described below.

2.3.1 Packet Format

A message is the highest level unit of resource allocation in a network. One message is a group of bits of arbitrary length, delivered from a source to a destination node.

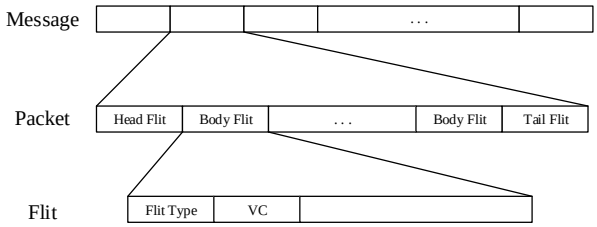


Figure 2.7: Different units of network resource allocation. A message is divided into packets and packets are divided into flits.

As seen in Figure 2.7, a message is divided into packets, which comprise of routing information, such as the addresses of the sending and receiving terminals, a payload etc. Although the packets are often of a fixed length, some networks use packets of different sizes, depending on the operation performed. Despite the length of each packet, there is a limit to a packet's length for a given system.

Packets are divided further into flits, which are of a fixed length for every NoC. Besides any possible payload, a transmitted flit carries an extra set of control bits which are relative to the destination of a packet, as well as to the resources the NoC needs to allocate. There are three types of flits available, namely header, body and tail.

A header is always the first flit of a packet, while the tail is the last flit. Everything in between is a body flit. A packet that contains only a head is called a single flit packet. Otherwise, a packet can be formed by a head, a tail, and zero or more body flits between them.

2.3.2 Topology

A NoC's topology describes its structure and organization, and determines the arrangement and connection between the network routers and the channels. Figure 2.8 presents a few of the possible network topologies: 2D mesh, ring, crossbar and torus, all with different capabilities and features.

The 2D mesh, one of the most popular topologies, consists of an $M \times N$ grid of switches, with each one connected to every adjacent switch. Every router has five ports: four for each possible neighboring router, and one to which a local core can be connected.

The ring, or what can be described as a 1-D torus, is a number of switches with each one connected to its neighboring switch. Although the ring is preferred due to its simplistic design, it can limit the networks scalability and performance when the number of nodes is increased.

An $M \times N$ crossbar directly connects M inputs to all N outputs using multiplexers or tri-state gates. Although crossbars offer simplicity and high speed, they do not scale well for bigger networks, where their cost can render them inefficient [4].

Lastly, the torus is a topology similar to the mesh, where nodes at the edges are

connected to switches at the opposite edge. Tori are known to exhibit low latency and high speed, but also complexity in their wiring and a relatively high cost.

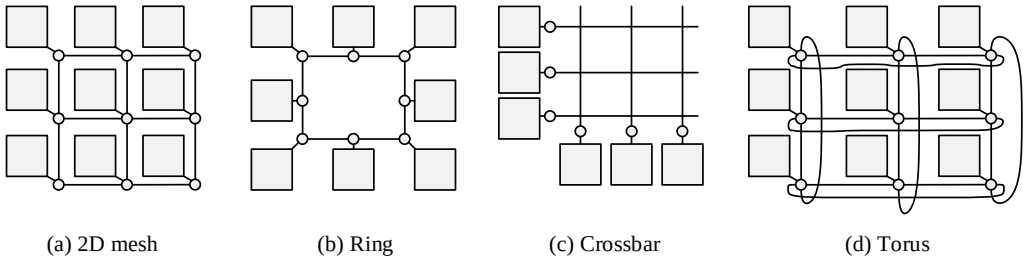


Figure 2.8: Example of different types of topologies.

An additional topology, similar to the 2D mesh, and the one we used for this thesis, is the Concentrated Mesh (Figure 2.9). In the 2D mesh, although every router has the potential to be connected to five components, not all of them exploit that capability. Specifically, the routers that are placed in the periphery of the mesh, are left with open ports.

That is not the case with the concentrated mesh, in which there is a component connected to each and every available port of all routers. Examples of such networks can be seen in Figure 2.9.

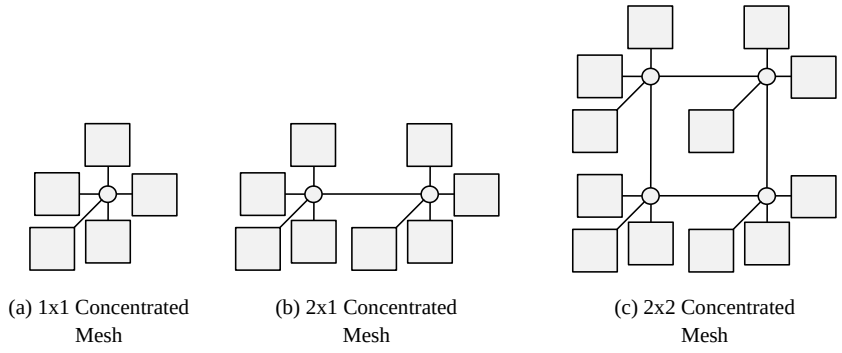


Figure 2.9: Concentrated Mesh Topology.

Compared to the 2D mesh, it is clear the concentrated mesh possess the ability to utilize more cores and minimize the number of routers, which also positively affects the implementation of the system.

Specifically, in a 2D mesh, there is one core for each router, therefore in an $M \times N$ network, there can be $M \cdot N$ cores connected. By using an $M \times N$ concentrated mesh, on the other hand, we can connect a total number of $M \cdot N + 2(M + N)$ cores.

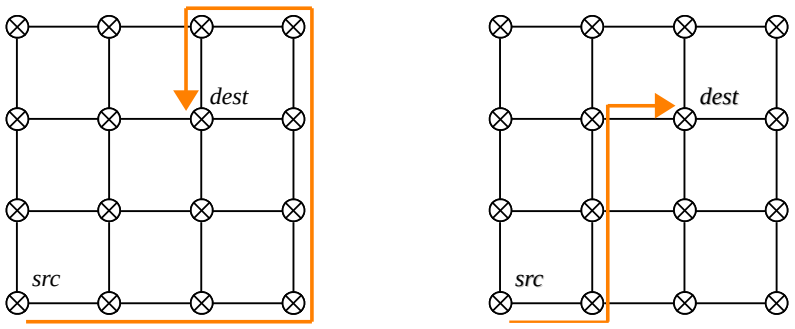
For example, a 2×2 concentrated mesh consists of only 4 routers and can utilize 12 cores, while a 2D would need 12 routers for the same purpose.

Topologies can also be classified as either regular or irregular [24], based on their structure and whether or not they use a homogeneous distribution of routers. Regu-

lar topologies provide re-usability and allow for the designer to redesign the system for different applications with little effort. Irregular topologies on the contrary, offer a decreased delay and power consumption.

2.3.3 Routing

The routing algorithm is employed by the network to calculate the traverse paths for packets, from a source to a destination node. Depending on the network's topology, the number of routing paths may vary.



(a) non-minimal (b) minimal

Figure 2.10: Example of (a) non-minimal and (b) minimal routing path for a 2D 4x4 mesh network.

Numerous criteria can be used to classify routing algorithms, one of which is the routing path's length. Depending on whether it reaches its destination with the minimum number of hops or not, routing can be classified as non-minimal (Figure 2.10(a)) or minimal (Figure 2.10(b)). Although minimal paths can achieve lower power consumption, non-minimal routing can also produce less load imbalance and can provide better fault tolerance [4, 25].

A second classification is between deterministic and non-deterministic routing algorithms (Figure 2.11). Deterministic algorithms always follow the same predefined path, from a specific source to a destination, even if there are multiple paths available. Non-deterministic algorithms on the other hand, may select a different path for the same pair of nodes.

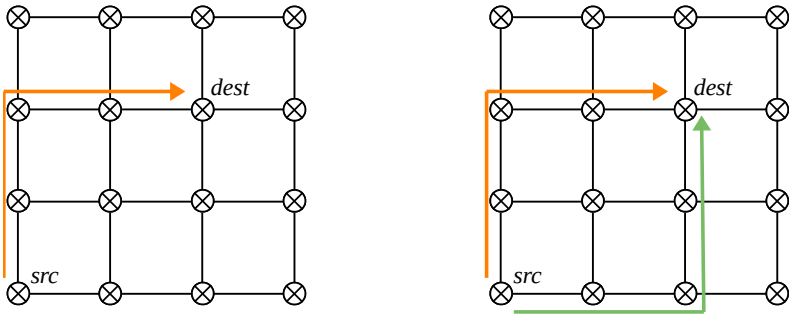
2.3.4 Flow Control

Flow control manages allocation resources for packets while they traverse across the network. The most essential resources are the channels between the network's nodes, and the storage implemented within theses nodes.

A fair flow control strategy avoids deadlock [4], a situation where a number of packets is waiting indefinitely on each other to release a resource in order to make progress.

2.3.4.1 Bufferless Switching

In bufferless switching, when a packet's header is unable to allocate a resource, it waits until that resource becomes available again. Meanwhile, the network drops or



(a) deterministic (b) non-deterministic

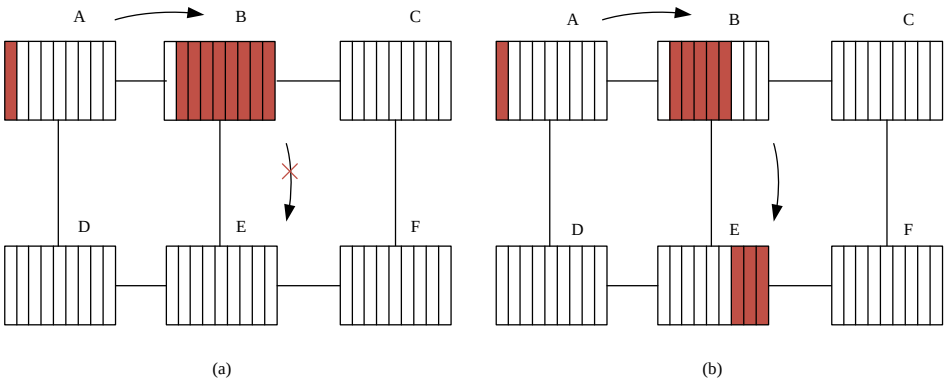
Figure 2.11: Example of (a) deterministic and (b) non-deterministic routing path for a 2D 4x4 mesh network.

misroutes the packet, instead of storing it.

For example, in circuit switching which is a form of bufferless flow control, a packet is transmitted only when the whole path has been reserved by the network. This type of switching is generally inefficient since it wastes valuable bandwidth, but preferable in specific cases where messages of considerable length are transmitted.

2.3.4.2 Buffered Switching

Opposite to bufferless, buffered switching offers a more efficient flow control, by buffering flits while they wait for a specific resource to become available. In packet switching, the message is broken down to packets and packets to flits which are then transmitted through the network. These methods exhibit a more efficient usage of the network resources and are frequently used in modern networks.



(a) (b)

Figure 2.12: Examples of buffered switching flow control. In store-and-forward (a), the packet stored in buffer A has to be transferred to B before it can start transferring to E. In cut-through (b), the packet's flits can move to C without the whole packet having to be buffered in B.

A number of flow control mechanisms are available. Methods such as the store-and-forward store the packet as a whole in a node's buffer (Figure 2.12a), and once it is complete, the packet moves flit by flit to the next buffer and so on. This process

might be trustworthy, but is also slow since it leaves a big part of the network available, but idle for long periods of time.

Cut-through on the other hand, is a different flow control mechanism that does not require a packet to be full before it moves on (Figure 2.12b). Even though this is an improvement in terms of resource usage, it might lead to Head-of-line (HOL) blocking, a situation where a number of packets are held up by the first packet in line, as demonstrated in Figure 2.13.

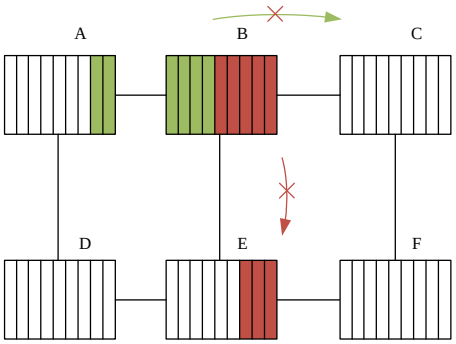


Figure 2.13: An example of HOL blocking. If the red packet is blocked, the green packet is also blocked, only because it is second in line, although its flits need to be transferred into a different buffer.

An alternative comes with the use of Virtual Channels (VC) control flow (Figure 2.14), where several virtual channels are associated with every single physical channel. This method allows multiple packets to be stored in the same channel, and at the same time it solves the HOL blocking by allowing flits to bypass blocked packets (Figure 2.14).

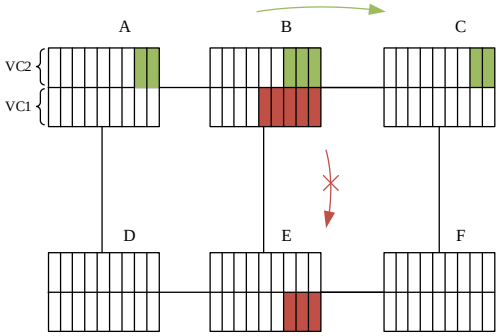


Figure 2.14: A network with virtual channels. Each physical channel (A-F) includes 2 VCs. Every VC contains a number of registers to store flits. This time, the blocked red packet, does not block the green from continuing.

2.3.4.3 Credit-Based Flow Control

In credit-based flow control, a receiving node hands out credits to a transmitter. Those credits correspond to the number of empty, and thus available, buffer slots. The transmitter on its side, uses a counter for these credits. When the transmitting

node forwards a flit, it essentially consumes one of its credits. On the other hand, the receiver stores that flit in a buffer until it is ready to use it, and when that time comes, it increases its buffer space and hands out one new credit. All in all, every transaction between two nodes can happen only when the transmitting unit has the necessary credits to spend.

2.3.5 Router Architecture

The block diagram of a typical NoC router that uses virtual channels can be seen in Figure 2.15.

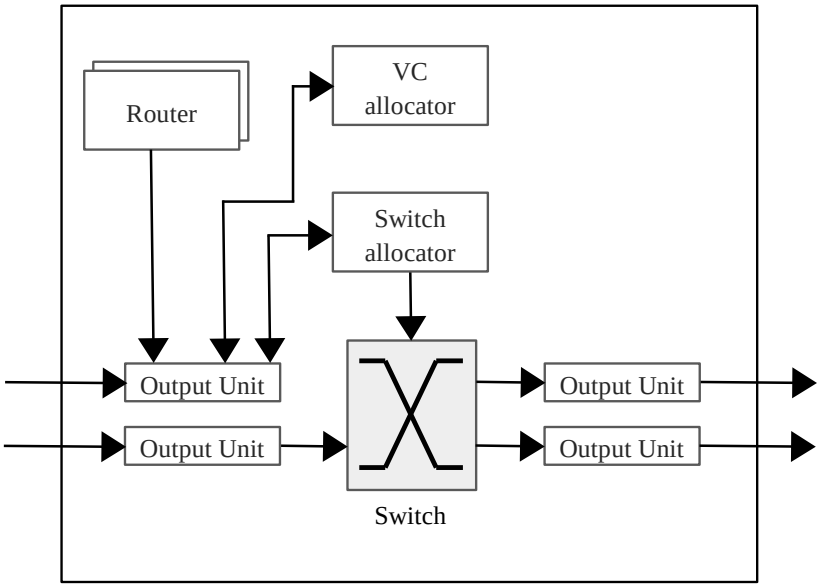


Figure 2.15: NoC router with virtual channels. Its main components are the input/output units, the switch (crossbar), the VC allocator (VA), switch allocator (SW) and routing unit.

The input buffers, output buffers and switch, form a group known as **datapath**, which is responsible for storing and moving flits. The router, VC allocator (VA) and switch allocator (SA) on the other hand are part of the **control plane** that controls the datapath.

During a typical operation, incoming flits are received, and possibly stored in the inputs buffers. For the flit to traverse through the router, and reach its output, a route must be specified, and a virtual channel must be allocated from the VA unit. When these conditions are met, the SA arranges for the flit to traverse through the switch, to the router's output, and from there, to either the next router's input, or to the local NI's input port.

The number of input and output units can vary, based on the NoC's topology. Figure 2.16 illustrates an example of the structure of a NoC router in 2D mesh network.

The router includes five inputs and outputs. Four of them span across all four directions and connect to neighboring routers. The fifth port is connected to the local NI.

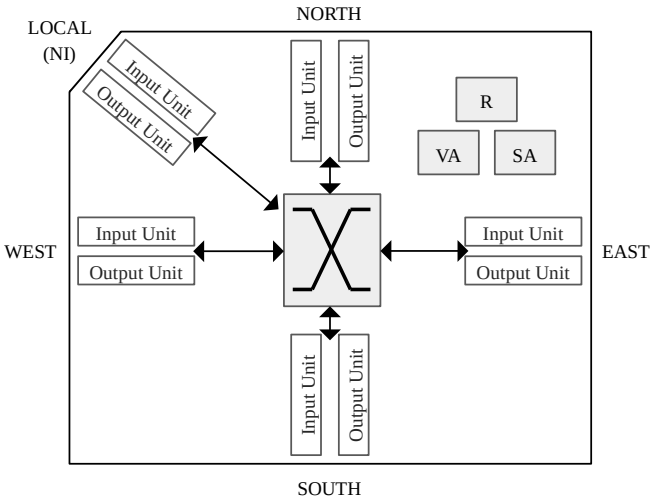


Figure 2.16: Five-port NoC router architecture. The router is connected to other routers in North, East, South and West ports, and to the local unit through the NI in the local port.

The aforementioned routing steps can be pipelined in a number of different ways. According to the selected pipeline, different technologies of router architecture have been designed. Nowadays, the trend is shifting towards Dual Data Rate (DDR) networks. Cutting edge research has recently demonstrated DDR NoCs [26, 27, 28] that increase the network's throughput. By using a slightly slower clock, the router's control is removed from the critical path, as was often the case, and the router has the ability to be traversed twice in each clock period.

A few notable router architectures are, among others, the Scorpio [29], the Shortpath [30] NoC routers, and Intel's full custom NoC router [31]. Scorpio and Shortpath, manage to demonstrate high frequency by pipelining the router's datapath in only two stages and putting the control logic in the system's critical path. Intel's router on the other hand, achieves an increased throughput, but also a lower latency, by deeply pipelining the router's datapath.

3

Design

This chapter introduces the designed network interface (NI). Here, an in-depth analysis of the proposed system takes place, as well as the ideas behind the choices that led to this result.

Section 3.1, describes the main ideas that served as basis in designing the NI. In section 3.2, we present the features of every unit that was part of the proposed system. Section 3.3 presents the NI in a higher lever.

Getting into the system's details, section 3.4 describes the specific packet format that was selected based on the range of available AHB operations. To conclude, sections 3.5 and 3.6 discuss analytically the design of the NI.

3.1 Design Flow

Starting from a bus-based system, our intention was to replace its AHB controller with a Network-on-Chip. The design process was divided into three main steps.

First, the definition of the NI's specifications took place, keeping in mind its two main responsibilities: communicating with the AHB units as well as the NoC. The supported operations were defined, and based on that, a specific packet format was selected. At this stage, the core of the NI's architecture was shaped to support the communication with the AMBA units, along with the packet generation and packet reading.

The next step was the development of a crossbar that would interconnect the NIs instead of the NoC. The crossbar, selected to reduce the system's complexity, was gradually introduced in the baseline system, parallel to the AMBA bus. Furthermore, it allowed for the NIs to be verified with the provided benchmark.

Last step towards the system's upgrade was the complete removal of the AMBA controller, but most importantly, the replacement of the crossbar with a NoC. For the NIs to be able to communicate with the NoC, virtual channels and credit based flow control were included.

3.2 System Parameters

3.2.1 Baseline System-on-Chip

Originally, a baseline system was provided by Cobham Gaisler, featuring the following:

- an AMBA 2.0 AHB bus controller [5, p. 62]: The controller combines an AHB arbiter, a bus multiplexer, and a slave decoder as per the AMBA 2.0 standard. The unit can support up to 16 masters and 16 slaves connected.
- AHB masters: those units simulate the behavior of actual IPs, as for example CPUs.
- AHB slaves: memory models that can be configured to allow access to up to four memory banks.

Although the cores support the full range of AHB operations, the ones included in the testbench and thus those that our carried out from our NI as well, are the following: single operations, bursts of unspecified length, back to back reads and writes, interrupts, and splits.

The baseline system included only 3 master and 2 slave cores, but its high level of abstraction allowed us to modify it according to our needs.

3.2.2 Network-on-Chip

As a model, we initially adopted the Single Data Rate (SDR) version of the FastTrackNoC [32], one with a regular 2D mesh topology that uses the minimal and deterministic routing path. The NoC's flow control is cut-through with 4 virtual channels of 5 registers each, with a width of 32 bits. Finally, the network uses credit based flow control.

In the given system, the AHB controller was replaced with a 2D mesh network, which ultimately lead to a system similar to the one in Figure 3.1.

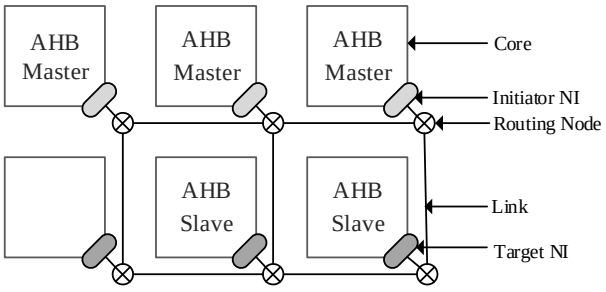


Figure 3.1: A 2D 2x3 mesh network with AHB masters and slaves as nodes.

Shortly, the parameters of the original NoC were modified to reflect a concentrated mesh topology, as the one seen in Figure 3.2. An advantage of this topology, as well as the main reason for selecting it, was that, for a given number of components,

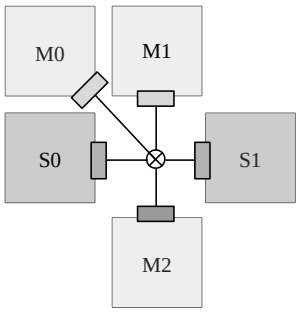


Figure 3.2: A single router concentrated mesh network with 3 master and 2 slave cores connected.

is utilizes a smaller number of routers compared to a 2D mesh and therefore it minimizes the number of hops a flit has to make to reach its destination.

On the other hand, it can lead to increased congestion as the decreased number routers results in less VC registers, and thus limited storage in their registers for incoming flits.

3.2.3 Crossbar

The first step towards a NoC-based system is to replace the bus controller with a crossbar and an arbiter that connects each master's NI, with every slave's NI and vice versa. The block diagram of the design is presented in Figure 3.3.

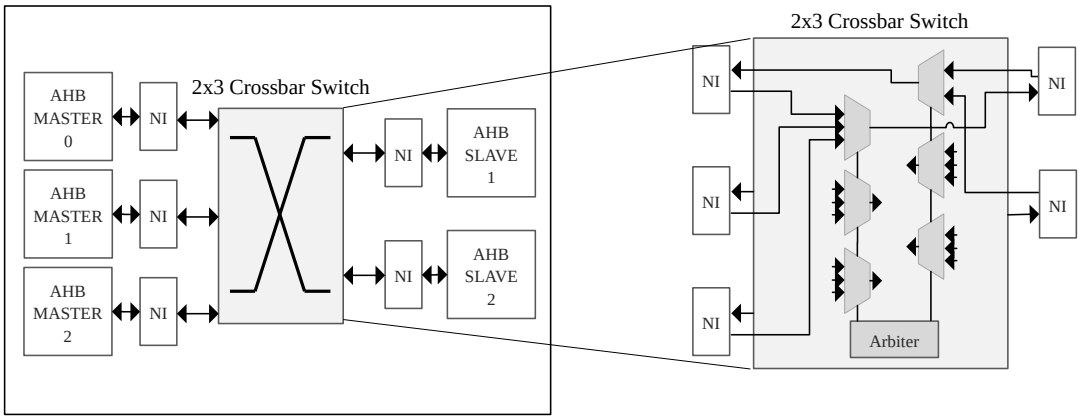


Figure 3.3: A crossbar connecting the NIs in the system. The connection of only one of the multiplexers is shown to reduce the figure's complexity.

Using the crossbar, the (at the time) newly designed, and constantly changing NIs, were tested with the provided benchmark. At the same time, this simpler design removed some of the complexity that the NoC later added to the system, as it does not include virtual channel allocation or credit based flow control.

By adding the necessary features in the NI, the crossbar was gradually replaced with the actual Network-on-Chip. At this stage, the NoC can be seen simply as a black

box, and the whole system is shown in Figure 3.4.

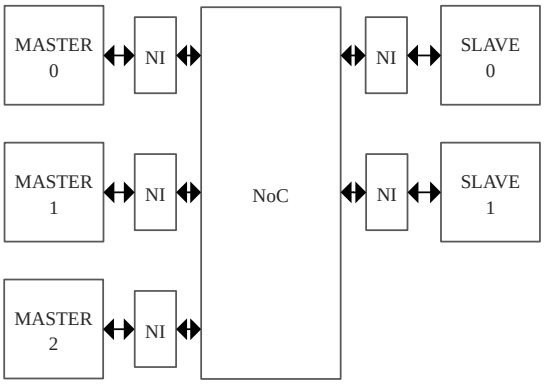


Figure 3.4: Block diagram of the final system configuration with the NoC and the NIs inserted and having replaced the AHB controller.

3.3 The Network Interface

The network interface that was designed, has the ability to use two separate protocols: the AHB protocol for communicating with the AMBA units, and the NoC protocol for exchanging packets with the network.

The main objective was to replace the bus, without affecting the original system's functionality or changing it in any other way. This means that from the perspective of the AMBA units, they should be unaware of any change in the system.

An example of a read request and response in the new system is presented in Figure 3.5, to demonstrate the interaction of the NI with the AMBA cores and the NoC.

After the NI performs the necessary handshake with the AMBA master, it receives the address and the control bits. The interface is responsible for creating NoC-acceptable packets with that information, and relaying it to the network's input port. Here, it is assumed that the NI possesses the necessary credits to complete the operation.

The packets traverse the network, following a specific path through one or more routers, and finally arrive to the Target interface. There, the AMBA relevant information is gathered, and is driven to the input of the slave.

When the AMBA slave replies, the opposite operation takes places, and the data pass through the two interfaces, and the network, to reach the master.

If the described, or any other operation, is observed from the AMBA units' perspective, then their functionality should not be affected. Similarly, the NoC simply transfers packets from a source to a destination NI, no matter what those packet contain.

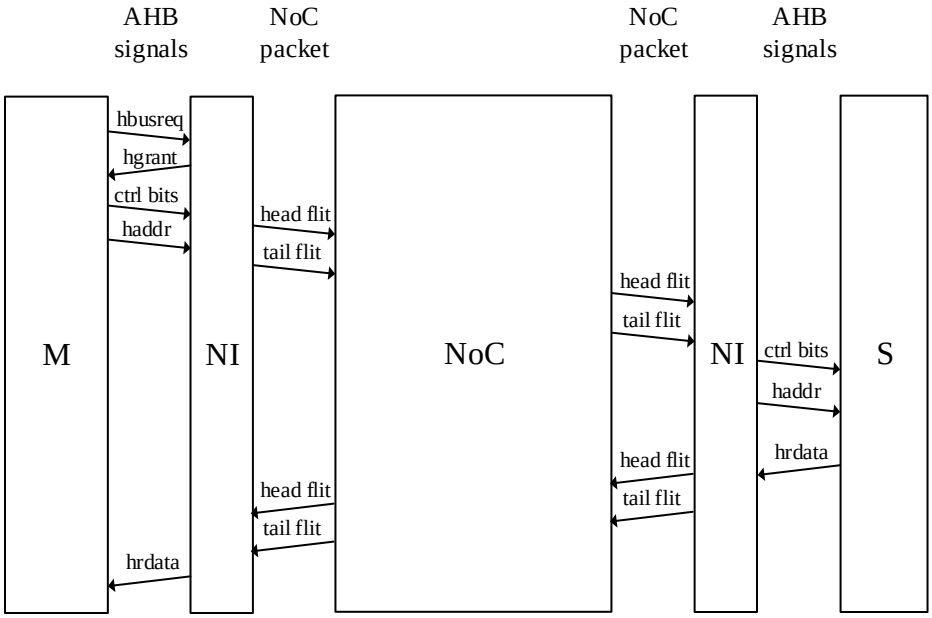


Figure 3.5: A reading request executed in the new system with a NoC instead of a bus. Events happen chronologically from top to bottom.

3.4 Packet Format

For the packet format to be analyzed, all the possible operations must be first identified and categorized, since different operations might require distinct formats. The possible operations can be divided into *requests*, *responses* and *interrupts*.

Requests, when initiated from an AMBA master, can either be reads or writes. A slave can also trigger split requests, but as will be explained later, such requests will be handled from the slave's NI in our system.

A **response** packet comes as a reply to a master's request. When an NI receives a read request, it replies with a read response, providing the master with the data retrieved from the slave. In the case of a write request or an interrupt, a response is not required.

Interrupts are a special kind of request packet, in a sense that they contain some of the information carried from other packets, but they are always single flit packets and they never expect a response from the Target NI.

The length of each packet is not constant, since it depends on the operation. Each packet is broken down into 32-bits flits, with an extra five control bits, resulting in a total of 37-bits. The control bits include the flit type which can be head, body, flit, single or idle. The second field included is the vc id, which is the id of the virtual channel used for the packet.

3.4.1 Request Packet Format

Every request packet consists of three parts, namely *header*, *address* and *payload*. The general form of such a packet, and the way it is divided into flits, can be seen

in Figure 3.6.

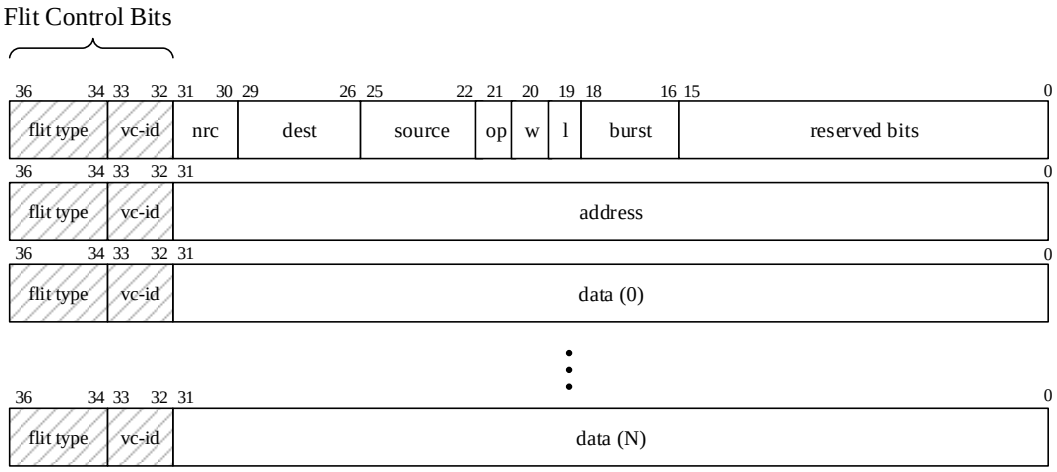


Figure 3.6: Request packet, divided into flits with added flit control bits.

The first flit which is the header part of the packet stores the following information:

NRC: When the NI is ready to send a flit, it performs the Next-Route Calculation (NRC), to instruct the receiving router on selecting a specific output port for the incoming flit. The range of the NRC values is the same as the five available ports each router has: North, West, South, East or Local. The NI only calculates the NRC before transmitting the head. For the rest of the packet's flit, the same NRC value is assumed. After a router receives the head flit containing the NRC, it calculates it again for the next router in the flits path, until the flit arrives to its destination.

dest: destination x and y coordinates of the target node in the network. This address will be read from the NoC to guide each flit to the correct destination. The destination address is calculated through a look-up table which translates a part of AMBA's HADDR signal sent from the master, from an AHB address to a network address.

source: The sender's unique coordinates in the network. Although the source address is not needed for a packet to reach its destination, it will be later used from the Target NI to know the node to which it might have to reply to.

op: Operation flag, which is high for a write request and low for a read request. This bit will help the Target NI to better interpret the contents of its receiving packet.

w, l, burst: AMBA AHB HWRITE, HLOCK and HBURST signals respectively. These signals, are crucial for the Target NI to better understand the content and number of incoming flits, so it can communicate with the AHB slave successfully.

reserved bits: Currently, the 16 least significant bits of the header flit are not used. Some of them may be utilized in case the network's parameters are modified. For example, changing the number of VCs or the number of source/destination nodes might necessitate the usage of more/less bits in the respective fields. Furthermore, in the future, additional features might be introduced to the header, as for example, error checking or a valid flit indicator bit.

The second flit of the packet carries the information stored in the AMBA HADDR. Independent of whether it is a read or a write request, the master always transfers an address to the slave, and that address must reach the Target NI.

When the master requests a single operation, one address is enough, but that is not the case in bursts of known or unknown length. In such an event, the master sends only the first address of a burst. Based on the value of the HBURST signal, the slave can then calculate the rest of the address values.

In the occasion of an unknown burst, the slave will assume a length of 8 and reply to the master (prefetchable reads).

Lastly, the third flit, and every other flit after that, can carry data. The number of data flits varies, depending on the operation.

3.4.2 Response Packet Format

Figure 3.7 represents the general format of a response packet format. The packet is divided into two main parts: the *header*, which contains the address information, and the *data* which holds the slave's response on a specific request.

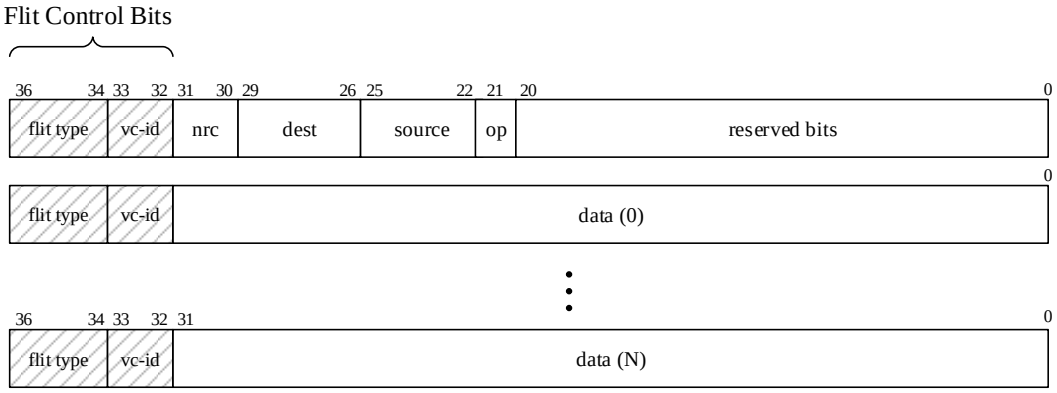


Figure 3.7: Response packet, divided into flits with added flit control bits.

It should be noted that there is no address field in a response packet. Due to the AMBA protocol's nature, responses are needed only in read requests in form of data sent from the slave to the master, and in write requests as an acknowledgment that the operation was performed successfully.

In the proposed system, all writes are assumed to be posted, and no acknowledgments are needed. Therefore, the only possible response will be to a read request, and it only contains the data from the HRDATA signal.

Once again, the length of the packet depends on the type of the read operation. When the master initiates a single read request, the response packet will contain only a flit of data, while if the request is a burst, the number of data flits will be enough to accommodate the master's needs.

3.4.3 Interrupt Packet Format

The format of an interrupt packet is presented in Figure 3.8. Although it bears resemblance with the response packet, the interrupt is always a single flit packet.

For calculating the destination coordinates fields, the 32 bits interrupt from the core are translated to a network address.

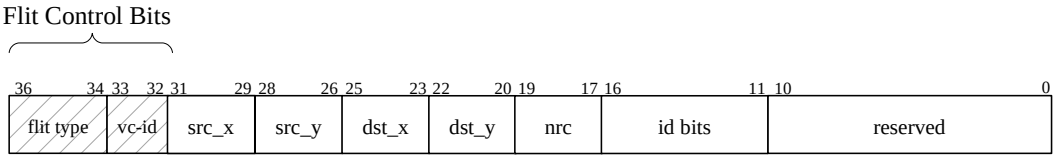


Figure 3.8: Format of the single flit interrupt packet.

3.5 Initiator Network Interface

An Initiator is a NI that is connected with an AMBA AHB master. Its purpose is to imitate the slave's behavior when communicating with a master, and convert requests, such as memory fetches, into traffic, according to the NoC specifications. Moreover, the Initiator receives packets from the NoC, translates them, and forwards the information to the master. The Initiator's block diagram can be seen in Figure 3.9.

The Initiator's architecture can be divided into two parts, each one of which is controlled from a Finite State Machine (FSM). The upstream is responsible for communicating with the master, and creating packets from the information it receives, while the downstream is in charge of incoming packets as well as relaying the necessary information back to the master. The two FSMs operate as independently as possible.

A number of components are shared by the two FSMs. The arbiter is responsible for selecting the VC that the FSM will write to or read from. The arbiter is a sequential circuit that is implementing a bit priority encoder. A possible upgrade to our system would be a more sophisticated and fair arbiter.

The arbiter bases its decision on information that receives from the credit controller regarding available VCs. The credit controller, keeps count of the number of credits received for each VC of the router that the NI is connected to. Furthermore, it

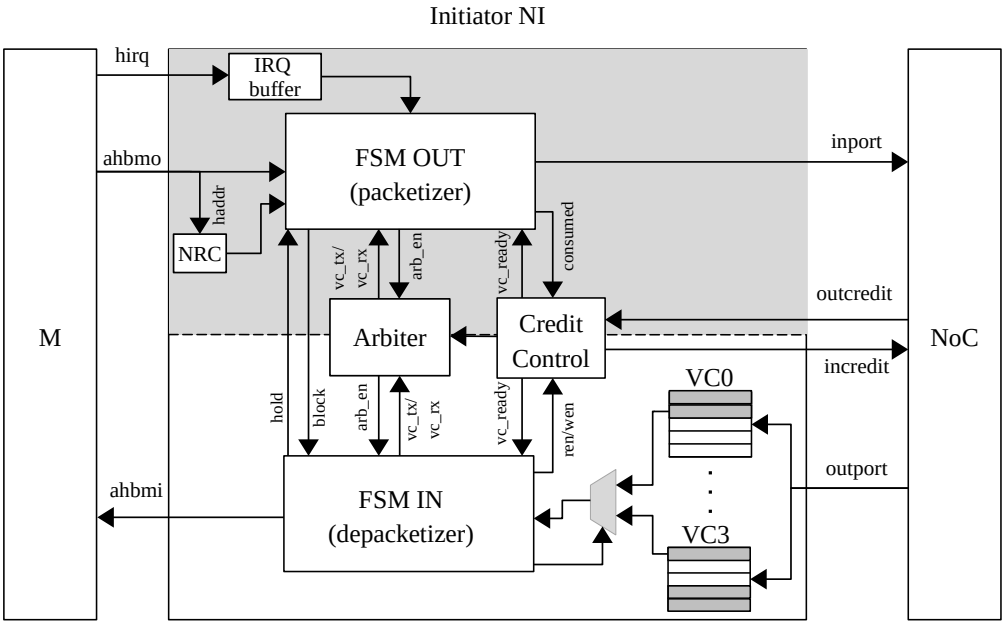


Figure 3.9: Block diagram of an Initiator NI.

observes the NI's internal registers for each VC. When a stored flit is read from the NI and a free spot is created, the credit control unit sends a new credit for that VC, back to the router that the NI is connected to.

3.5.1 Initiator NI Upstream

The master's output defines every outgoing packet. The FSM is responsible for creating every flit of each packet and driving it to the NoC input port. The FSM collaborates with the arbiter for selecting the VC id for the head flit, and with the credit controller to see if there is any available credits for the selected VC.

When the packetizer actually manages to send a flit through the network, it raises the `credit_consumed` flag and lets the credit controller know that it should update its counters.

The packetizer also gets the output of the NRC calculator, a unit that determines the value of the NRC field based on the value of the 12 most significant bits of HADDR.

The only signal not directly connected to the packetizer is the HIRQ. An interrupt occurs at any point in time, and it only stays active for a few clock cycles. We need therefore to ensure that the system will catch it, even if the packetizer is occupied in a different activity at the time the interrupt occurs.

That being said, the AMBA HIRQ signal's value is latched in a dedicated buffer. The FSM, when idle, first examines whether there is an interrupt waiting or not. If there is one waiting, then a single flit packet is created for the interrupt. While this operation is happening, the packetizer is blocked from interacting with the master's input.

In the presented system, a single buffer is used for interrupt storing. However, many more can be latched if a FIFO is used. Its size highly depends on the time the FSM needs for every one interrupt to be packetized and send to the NoC. In the future, if needed, specific resources may be used in the NoC only for the interrupts.

3.5.2 Initiator NI Downstream

When a packet arrives in the NI from the NoCs output port, regardless if it is a read response or an interrupt, its flits should be stored to the registers of one VC.

Every flit is driven to the input of all VCs. The id of the VC that the incoming flit should be stored to, is part of the head flit. Therefore, the packetizer's only responsibility there, is to read that field and raise the write enable of the corresponding VC.

One of the available register outputs has to be selected using a multiplexer. The FSM, with the help of the arbiter, makes that selection when the packet's head arrives, and keeps reading from that same VC for however long it takes for the packet to end, at which point the FSM will see the packet's tail.

When the FSM decides it is time to read a flit from a register, it raises the VCs read enable signal. When the depacketizer changes either the write or the read enable signal of any of the registers, it also passes that information to the credit controller for it to update its counters and give away new credits.

The depacketizer, also has the ability to force the output FSM to ignore the master's output for sometime, using the *hold* signal. This becomes relevant when the system is dealing with read bursts or back to back read requests.

Given that reads are not posted, after a packet for such a request is sent, the depacketizer comes in a state where it simply waits for an incoming packet containing the slave's response. During this time, since the two FSMs were built to be independent, the packetizer has finished its job and is waiting idle for the next master's output.

When a burst or back to back operations occurs, while the master is waiting for the response of its first request, it keeps the address of its next read request to its output. If the packetizer were to operate freely, then it would keep recognizing the output as a new read request and continue to create new packets, despite the fact that it would essentially be a single request from the master.

This is where *hold* is needed. While the depacketizer is waiting for the slave to respond, it blocks the input FSM. It should be noted that the packetizer is only limited from creating new read request packets. If an interrupt occurs during that time, it is served the way it should be.

3.5.3 NI - Master communication

According to AHB specifications, one of the system's masters, is assigned to be the *default master*. When that specific core requests the bus without having to compete with any other unit, it finds the bus already assigned to it. The master does not have to raise the BUSREQ signal, as it normally should according to AMBA, and

as a result, one clock cycle is saved from the total operation. In the NoC-based system though, each master is now connected to a single NI, so it can be thought as it is always the default master. Therefore, the HGRANT signal is always set to indicate that this specific master has priority. Consequently, this single cycle is saved in every operation.

In every aspect, the handshake between the master and the NI imitates the one that the master makes with the AHB controller. In the first round, the master drives the address bus and the control signals which are read and stored in two flits by the NI. In the second round, the master drives again the control signals and if needed starts sending data through the data bus. For each data transfer, a new flit is created by the NI.

The interface monitors the HTRANS signal to understand when a request operation starts. From that point on, the NI also observes the rest of the control signals to keep track of the master's status. Moreover, it uses the HREADY signal to postpone the operation when needed. The way the NI interprets every AHB signal of the master-NI interface is described in Table 3.1.

AHB Signal	**Usage in the NI**
HGRANT	The master is only connected to the NI, so this signal is constant, always leading the master to believe that it has priority.
HREADY	Normally, this signal originates from the slave as a mean for it to declare it is ready for the next transaction. In the proposed system, the NI uses HREADY to stall the master from changing its output.
HRESP	Always driven with value OKAY. Even when a split occurs in the slave, the Target NI will take care of it and it will never reach the master.
HRDATA	Its functionality is the same as in the AHB protocol. When the Initiator receives a response packet, it drives the response data in the HRDATA and raises HREADY.
HBUSREQ	Although this signal exists in the master, it is not used anymore since it always has priority according to the HGRANT signal and does not need to request access to the bus anymore.
HLOCK	A lock functionality has not yet implemented in the proposed NI. For now, the HLOCK value is passed to the Target NI for future use. It is important to note that this signal would not have the same meaning for the two systems. Although in some cases it is useful to lock the bus and prioritize a specific operation, the same idea is not desirable in a NoC.
HTRANS	Its value is monitored from the NI so it can understand when a transmission starts (NONSEQ), when it ends (IDLE) and when there are back to back operations (SEQ).
HADDR	Monitored from the NI and saved on the beginning of the master's first phase of any operation. The address becomes a separate flit that is sent to the slave.
HWRITE	This is read from the NI in the beginning of the operation and it is one of the signals that guides the FSM to build the NoC packet. As analyzed before, read and write request

	packets differ in their format.
HSIZE	This signal is for now ignored from the NI. All transfers are assumed to be of 32-bits width.
HBURST	The type of burst determines the number of flits that will be created and sent. Aside from that, it is also part of the control flit that the slave receives in order to know the number of flits it should expect.
HPROT	This signal is ignored for now, as the NI concentrates on carrying the packets from and to the AHB units, not considering the type of data.
HWDATA	Monitored from the NI and saved on the second stage of any operation. In case of a burst, multiple flits are created based on the value of HWDATA.

Table 3.1: Usage of the AHB signals from the Initiator NI.

For the purpose of building a faster system, every write operation is assumed to be posted. When dealing with an AMBA write, the master makes available the data, and in the best case, one clock cycle later the slave has finished storing it, resulting in a low latency operation. That is not the case in a NoC based system though, since the network adds latency to every single packet transfer. If we were to use non-posted writes, the master would have to wait for the Initiator interface to create the packet, send it through the network, and the wait to get back the acknowledgment signal from the slave. During this time, the master would be available (high HREADY signal), but idle, thus wasting valuable resources. Instead, with posted writes, the NI confirms the correct result of any write request as soon as it receives the data, deceiving in a way the master into believing that the operation is completed, even before the packet's flits reach the slave.

This technique is not applicable in read requests. Due to the nature of the AHB protocol, after the master outputs the address, it waits for the AHB controller, or the NI in our case, to provide the read data in HRDATA and raise the HREADY signal. Therefore, the NI is forced to lower HREADY until the data arrive, leaving the master in a waiting state.

3.5.4 Initiator NI FSM State Diagrams

Below, the state diagrams of the two FSM of the Initiator NI are presented.

3.5.4.1 Initiator's Packetizer State Diagram

Figure 3.10 shows the state diagram of the Initiator's packetizer, the module responsible for coordinating with the AHB master and the rest of the NI's unit to create packets and forward them to the NoC.

All the supported operations can be seen in the diagram, together with the different phases required to communicate with the master. For single write operations, the FSM creates a three-flit packet containing the AHB signals. Single and burst reads create two-flit packets with the control signals and the requested address.

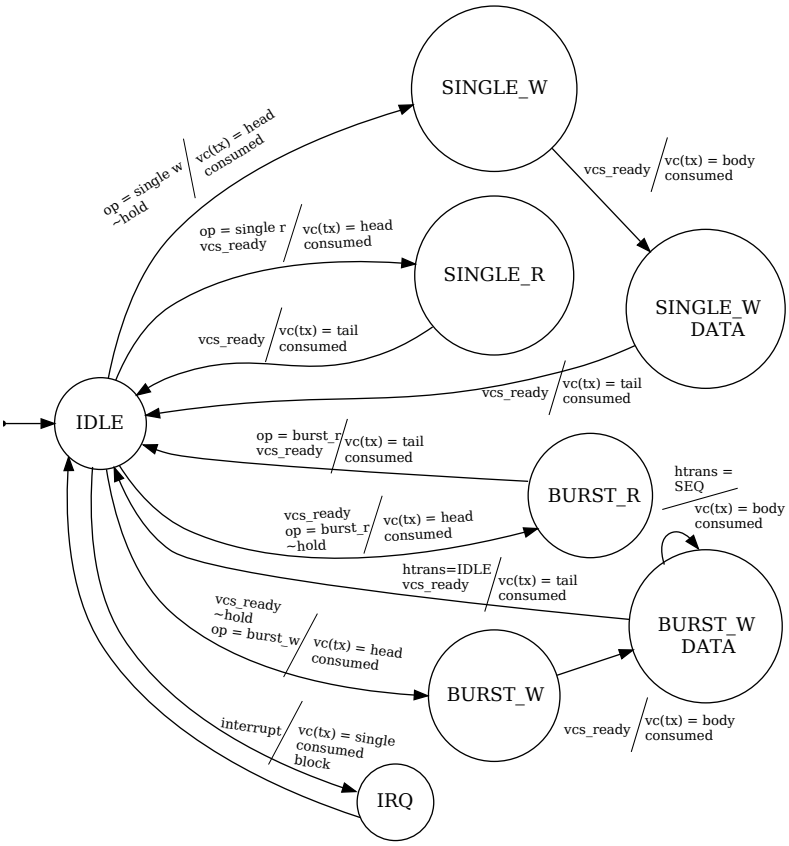


Figure 3.10: State diagram of the Initiator's Packetizer.

For write bursts, the NI keeps creating flits, until the AMBA HTRANS signal gets the value idle, at which point the operation ends.

Finally, interrupts are handled from the FSM, which creates a single flit packet when it detects a valid interrupt waiting, being stored in the interrupt buffer.

3.5.4.2 Initiator's Depacketizer State Diagram

The depacketizer, illustrated in Figure 3.11, is the second FSM operating in the Initiator, a unit with more complex logic compared to its counterpart. The depacketizer has two main responsibilities: handle the system's input, which is the slave's response to requests previously made by the master, and also handle HREADY, the AMBA signal used to coordinate with the master, either by allowing it to continue its operation, or by putting it on hold.

The latter establishes a need for the depacketizer to follow the packetizer's steps when creating packets, so it can manage the HREADY signal and allow the master to move to the next stage along with the packetizer.

For example, when the master initiates a single read request, the packetizer first creates a flit with the control signals and at the same time, the depacketizer raises HREADY and allows the master to move to Phase-2, as this has been described earlier. In the next clock cycle, the packetizer creates the tail flit, which contains

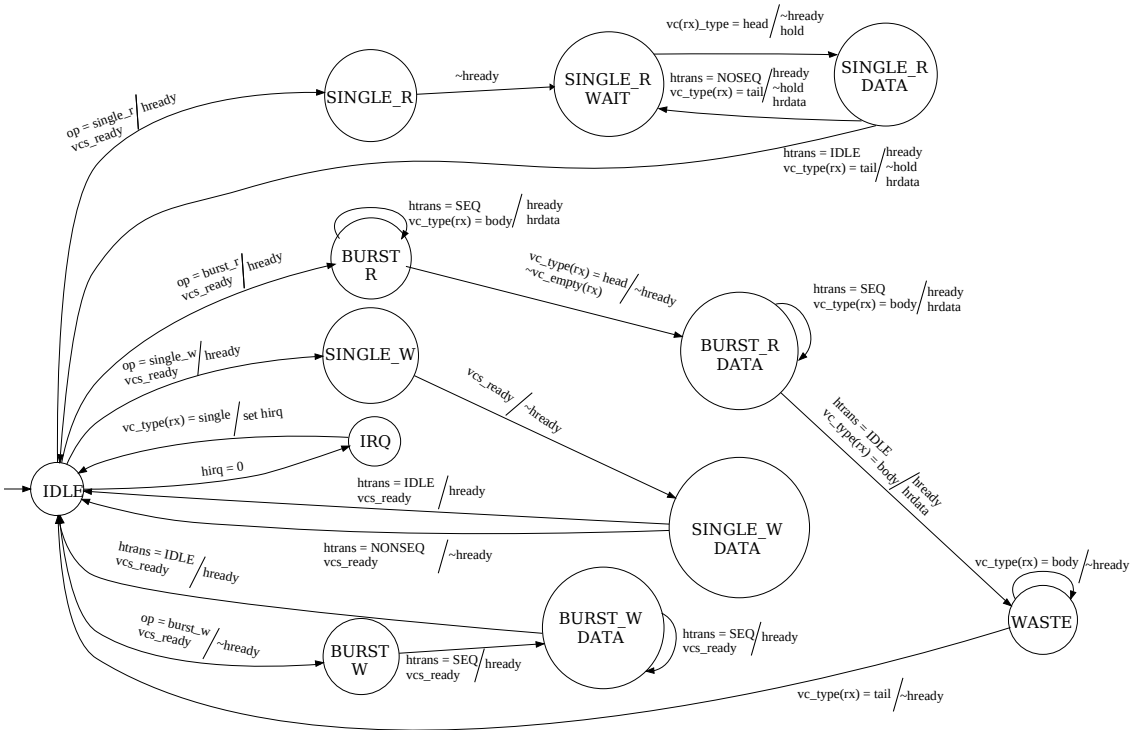


Figure 3.11: State diagram of the Initiator's Depacketizer.

the requested address, while the depacketizer lowers HREADY, preventing this way the master from finishing its operation.

At this point, the packetizer has completed its duty and is idle again, and the master waits for a response. The depacketizer gets into a state where it observes the VCs for possible incoming packets, and when the time comes, in cooperation with the system's arbiter, it forwards the response to the master. After finishing, and while being in the SINGLE_R_DATA stage, the FSM examines the possibility of a new head flit waiting on top of the VC buffers. If there is one indeed, it goes back one state and performs the same steps, saving this way a cycle in the reading operation. Otherwise, it returns to the idle state.

Every other operation performed in the NI, follows the same principle as the one described above. It is also worth mentioning the read burst operation. As explained, the master always receives the data of 8 addresses, even if it actually needs less. In that case, the FSM goes into the WASTE state, where it keeps reading from the VC's buffer until every extra received flit is wasted.

3.5.5 Initiator NI timing example

Figure 3.12 presents an example of a single read operation from the NI's point of view, to better demonstrate its function and capabilities.

For a single read operation, the Initiator NI needs to create 2 flits, according to the request packet format in Figure 3.7. The header flit with the control information, and the tail flit which contains the address.

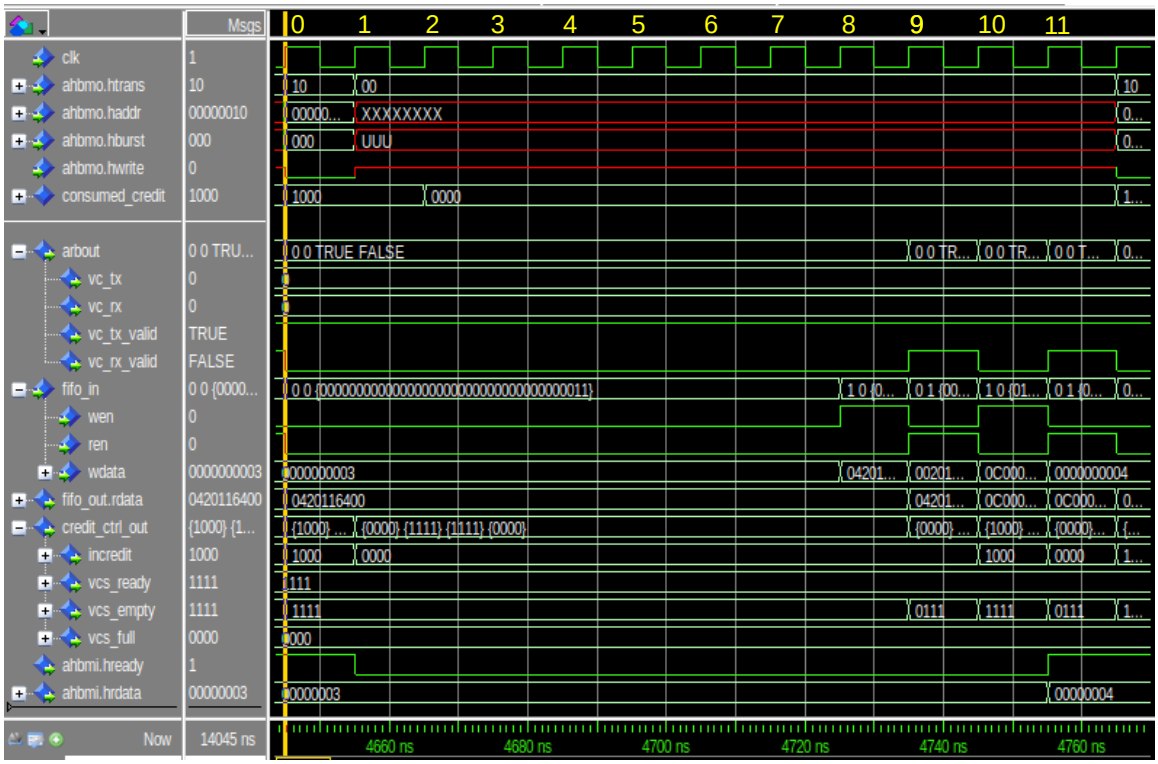


Figure 3.12: Initiator NI simulation for a single read operation, where a master requests to receive the data from address 0x00000004. Clock cycles are numbered, starting from the moment the operation is initiated.

During clock cycle 0, the master outputs the control bits, and the address 0x00000010 in `ahbmo.haddr`. The most important signals at this point are: the HTRANS, which has a binary value of "10", indicating that this is the first phase of the AHB operation, HBURST which suggests a single operation, and finally HWRITE, which by being 0 means that this is a reading request from the master.

During cycle 0, the NI builds the head flit, and reads the arbiter's output to decide on the VC that will be used. In the figure, the arbiter outputs the signal `vc_tx`, which is the VC selected for the transmission, and has the value 0.

Before forwarding a flit into the NoC, the NI consults the credit control unit's output on whether there are any VC buffers available in the next router or not (`vcs_ready`). Given that everything is ready, the NI forwards the head flit in the NoC and raises the `consumed` signal to inform the credit control unit that it just spent one of the available credits. Later, the NI will receive back that credit from the router connected to its port.

After sending the head flit, the NI creates the read operation tail flit during cycle 1. To accomplish that, it uses the address given by the master during its first phase. Meanwhile, the NI lowers `ahbmi.hready` signal in the master's input, to prevent it from moving into the second phase.

After sending the packet, the NI waits to receive a response with the requested data. In cycle 8, the NI receives indeed a head flit, which is stored in a buffer in

VC0 (Virtual Channel 0). One cycle later, at the beginning of cycle 9, the head flit is available in the buffer's output `fifo_out.rdata`. At the same time the NI activates again the arbiter and gets the value of `vc_tx`, which corresponds to the VC in which the flit is stored. Knowing the VC, the NI gets the head flit, keeps the necessary information and waits again for the next flit.

In the beginning of cycle 10 the tail flit arrives and is stored in the VC's buffer. In cycle 11, following a procedure similar to the one that took place with the head flit, the the NI strips the tail flit from everything but the slave's data response.

Having all the information needed, the NI drives the master's `ahbmi.hrdata` with the value 0x00000004 and at the same time raises again the `ahbmi.hready` signal, as per the AHB protocol.

This last action concludes the read operation which in total needs a minimum of 11 clock cycles.

3.6 Target Network Interface

A Target is an NI connected to an AMBA AHB slave and its functionality is similar to that of the Initiator. Regarding its interface with the slave, the Target NI is responsible for imitating the behavior of a master. Moreover, this NI's purpose is also to exchange packets with the NoC, while negotiating with it for available virtual channels. The Target's block diagram can be seen in Figure 3.13.

Once again, the interface can be divided into upstream and downstream. Similar to the Initiator NI. Each part has a separate FSM operating, along with a number of units. The arbiter and credit controller are the same as the one used in the Initiator NI, in terms of both their design and their functionality.

3.6.1 Target NIs Downstream

A part of the NIs downstream comprises of the same set of FIFOs, implementing a number of VCs, that were in the Initiator NI as well. Once again, the input FSM reads the VC id from the incoming flits and stores them to a specific register. At the same time, one of the available outputs is read and driven to the FSM.

One extra element of the Target NI, is the presence of the BUFFER unit. The Target NI needs to handle splits generated from the slave. AMBA RETRY/SPLIT can be produced during the address phase of a transaction and provide the slave with a way to indicate that it is not ready to complete the given operation. When a RETRY/SPLIT happens, originally the AMBA master, or the Target NI in our system, should wait and retry the last operation when allowed again.

Since every operation is stored as packets in the VC registers, the flits have been already read from the register once the RETRY/SPLIT occurs, and therefore cannot be read again. So, the use of buffers that store the last two flits that came out the FIFO is crucial to the normal operation of the Target NI.

Moreover, the input FSM has the ability to signal its counterpart FSM that a read request header just arrived. This helps in saving a cycle from the total read

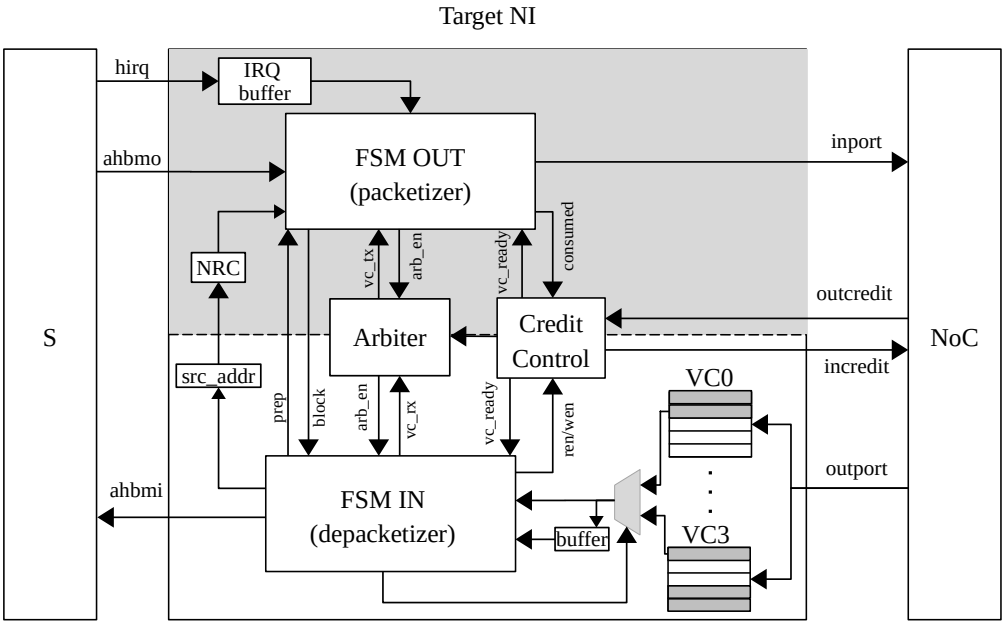


Figure 3.13: Block diagram of a Target NI.

operation. When the head flit of a read request reaches the depacketizer, the result of the operation, i.e the HRDATA, is not yet available from the slave. What is known though, is the information that is going to be used from the packetizer to build the head flit of the response packet. This way a full clock cycle is saved by sending early the head flit back to the Initiator NI.

3.6.2 Target NIs Upstream

The Target NI's upstream design is almost identical to that of the Initiator NI. The packetizer, responsible for building outgoing packets, follows the slave's output for read responses.

The selection of proper VCs is once again made using information provided by the arbiter and credit control unit. An interrupt buffer exists to locate and store interrupts generated from the slave.

The only difference here is that in case of a read response, the head flit containing the receiver's address is created with information received from the output FSM. Therefore, the NRC unit collects the destination address from a register in which the destination address of the previously arrived head flit has been stored.

3.6.3 NI - Slave Communication

In the Table 3.2 we describe the way the NI interprets every AHB signal relative to the slave.

AHB Signal	**Usage in the NI**
HSEL	Since each slave is connected only to one NI, the HSEL is a

	constant indicating that every transfer is intended for this slave
HADDR	The address that is arriving from the master in the packet or calculated in the target in case of a burst is fed in the slave's HADDR signal
HWRITE	It is interpreted the same as in the AHB protocol. This signal indicates the type of operation (read/write) and it is part of the request packet's control bits
HTRANS	The Target NI tries to imitate the master's behavior towards the slave. The HTRANS values depend on the input given to the slave, and follow precisely the AMBA protocol
HSIZE	Until now, a size of 32-bit width transfer has been assumed
HBURST	The value of HBURST, arrives with the request packet from the master and is fed to the slave as is.
HWDATA	In case of a write request, at least one packet containing data arrives and is driven to the slave's HWDATA
HPROT	This signal is being ignored from the NI for now, so currently is driven with the value 0000 from the Target NI
HMASTER	Practically each slave is only connected to one master, so the HMASTER signal is constant.
HIRQ	Interrupts are received from the FSM, translated into a network address and transmitted into the network.
HREADY	The Target's FSM observes this signal at all times. HREADY is the way for the slave to let the NI know that it is ready for the next step of an operation.
HSPLIT	In case of a SPLIT/RETRY, the NI waits for the HSPLIT to retry the last operation
HRESP	In case of an SPLIT/RETRY, the NI stops the ongoing operation. By changing the HSEL to all zeros, it is implied that the slave is not selected anymore. When the slave, using the HSPLIT indicates it is available to retry, the NI repeats the last operation stored in its buffers
HRDATA	When a slave outputs the requested data, the NI creates a new flit as part of an outgoing packet. According to AMBA, in a burst, the slave will keep sending data and for each one a new flit will be created from the NI

Table 3.2: Usage of the AHB signals from the Target NI.

3.6.4 Target NI FSM State Diagram

In this section, we discuss in more detail, the two FSM included in the Target FSM by presenting their state diagrams.

3.6.4.1 Target's Depacketizer State Diagram

The FSM's state diagram can be seen in Figure 3.14. Similar to the Initiator, the Target's depacketizer monitors the NI's input for incoming requests from a master.

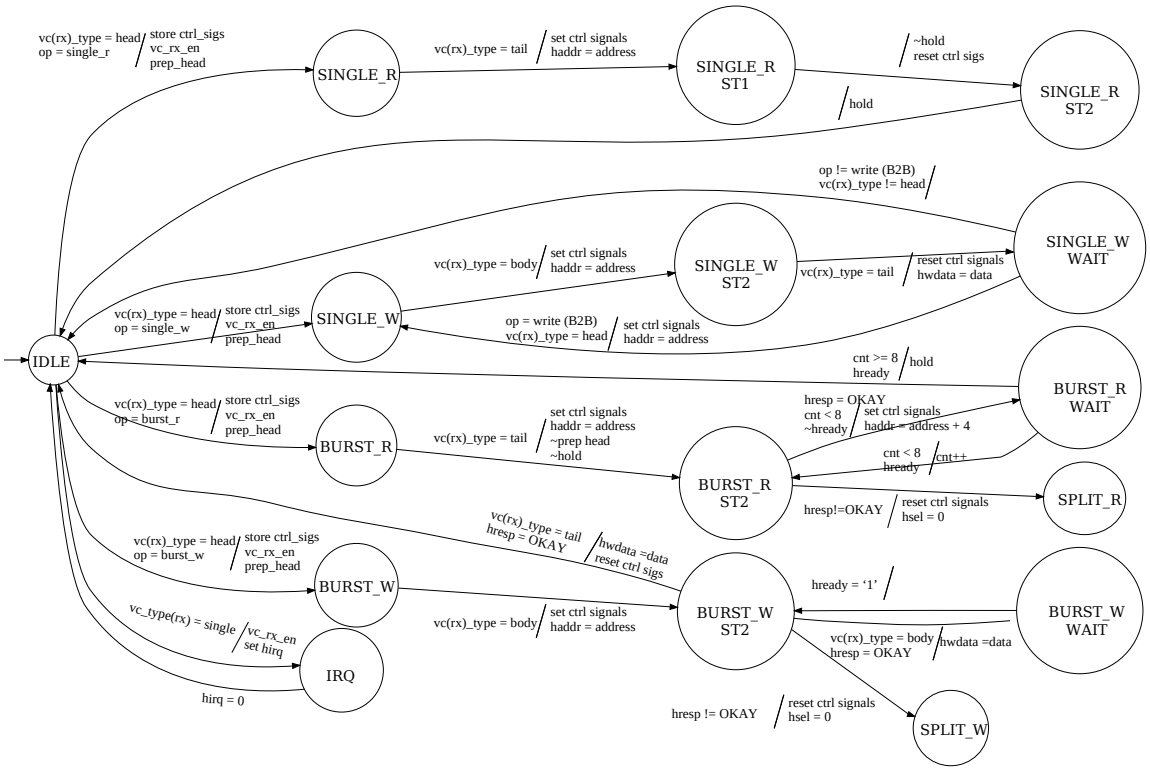


Figure 3.14: State diagram of the Target's Depacketizer.

When a head flit is detected in the buffer of the VC that the arbiter pinpoints, the FSM gets in a specific state, according to the operation detected from the control signals which are part of the head flit.

From there, the depacketizer's functionality lies in its ability to strip the flit from any unnecessary information, store the control signals and read the next flit, given that is available in the VC. In the next clock cycle, those stored signals are combined with the information contained in the next flit to form the first phase of the AHB slave's normal operation.

Two special states are the splits that may occur while being in the middle of reading or writing bursts. In such a case, the FSM handles the splits according to the AMBA protocol. The important difference between this system and the baseline, is that splits are handled in the Target NI, leaving the master free of any possible delays. Although the slave is still occupied, for the same time it would normally be in a bus-based system, precious time might be saved from the master.

3.6.4.2 Target's Packetizer State Diagram

The packetizer's state diagram, as can be seen in Figure 3.15, is the simplest one so far, as it is utilized in the three occasions where the slave is required to create a packet: when responding to single read request or burst read requests and when generating an interrupt.

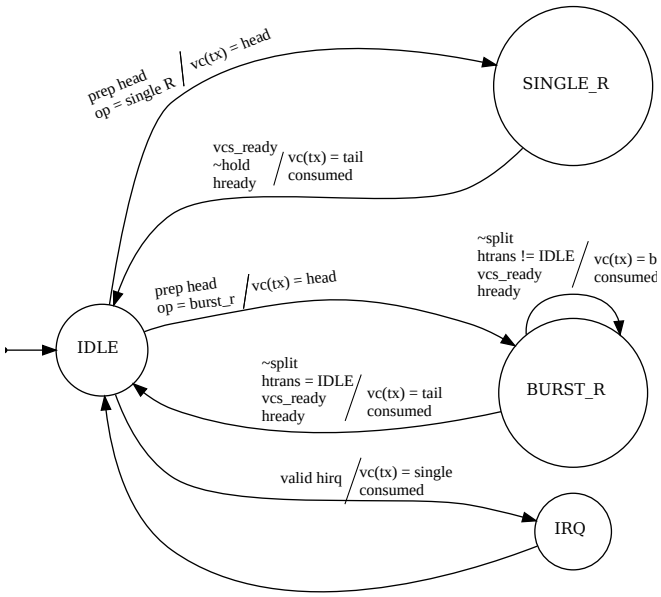


Figure 3.15: State diagram of the Target's Packetizer.

In all three cases, the FSM's way of operating is really straightforward. When a single read occurs, the FSM first sends a head flit with the control signals, and then a tail flit with the slave's respond. In the event of a burst read, after sending the head flit, the FSM creates 8 more flits with data. Finally, if the FSM realizes that a valid interrupt has been generated, it creates and forwards a single flit containing the address of the core that is to be interrupted. It is reminded, that when responding to a read request, the head flit is created early, after a request from the depacketizer.

3.6.5 Target NI timing example

Figure 3.16 demonstrates a single read AHB operation. The simulation is the same as the one presented in Figure 3.12. This time though, we study closer the events taking place in the Target NI.

Before cycle 0, the master that is connected to the Initiator NI, has already requested the data of the address 0x00000010, and accordingly, the NI has created a packet.

In the events shown in the above figure, cycle 0 is the moment that the head flit of that packet arrives in the Target NI, as can be seen from the change in the signal `wdata`, which is the input of one of the VC buffers.

In the next cycle, that flit reaches the buffer's output (`fifo_out.rdata`), and the NI, based on the `vc_rx` output of the arbiter, raises its read enable signal

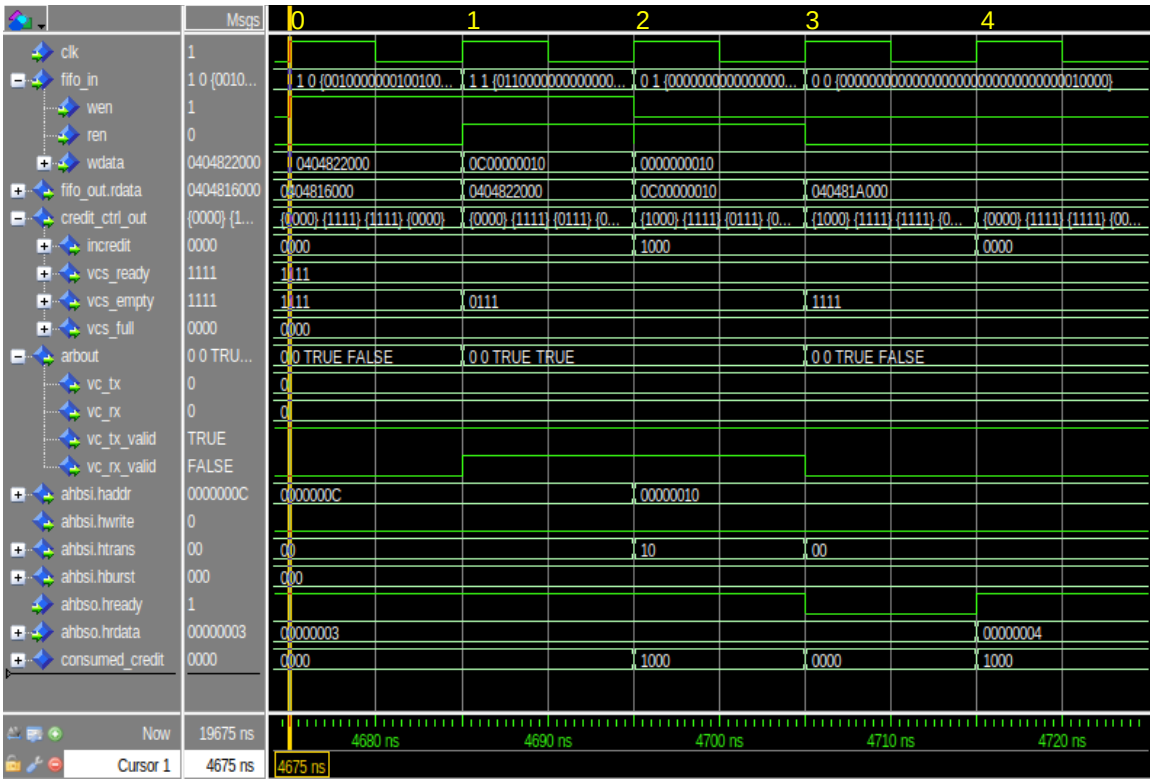


Figure 3.16: Target NI simulation for a single read operation, where a master requests to receive the data from address 0x00000004. Clock cycles are numbered, starting from the moment the NI receives the head flit.

`fifo_out.ren`). At this point, the NI has realized this is going to be a single read operation. The response of that operation will be a two flit packet, the first of which will have information that the NI already possesses from the incoming flit, like the Initiator's and Target's address, and the NRC.

Therefore, at that time, the head flit is created and sent through the network, even though the slave has not even replied yet. This way, the operation's latency is reduced by one clock cycle, since the alternative would be to wait for the slave to respond and only then spend a cycle to send the head flit.

Right after the head, the tail flit also arrives, and after being stored to a VC's buffer, it is available to the NI in cycle 2. At that moment, the NI drives the slave's input with all the control signals and the request address which initiates the slave's first phase.

In the beginning of cycle 3, the control signals are modified according to the AHB protocol. The operation is concluded in cycle 4 where the slave finally responds using the `ahbso.hrdata` signal, while raising the `ahbso.hready` signal at the same time.

At that very moment, the NI creates the tail flit using the slave's response as a payload, and sends it through the NoC, using one of it's remaining credits.

4

Evaluation

To emphasize the advantages of the NoC-based system with the designed NI over the baseline, the Zero-load latency (ZLL) of each system is discussed. ZLL refers to the latency of an operation in absence of contention.

Moreover, we present simulation results from different configurations covering all the supported operations. In the simulations the number of connected components as well as the number of operations varies. We also analyze a case study of a burst write for different injection rates to study the highest rate each system can handle before it saturates. Finally, we discuss the resources needed for each one of the two systems.

Based on these metrics, we conclude on whether the NoC-based system is preferred, and under which circumstances that happens.

The tools used were Intel Modelsim Starter Edition 2020.1, and Xilinx Vivado 2021.1.

4.1 Zero-Load Latency Analysis

In the NoC-based system, the ZLL of an operation depends on the latency added by the two NIs, L_{NI_i} for the Initiator and L_{NI_t} for the Target, the latency added by the NoC itself L_{NoC}, and finally, the latency added by the two components involved: L_{AHB_M} for the master, and L_{AHB_S} for the slave.

$$ZLL = L_{NI_i} + L_{NI_t} + L_{NoC} + L_{AHB_M} + L_{AHB_S} \tag{4.1}$$

For the SDR version of FastTrackNoC, the ZLL_{NoC}, has been calculated [32]:

$$ZZL_{NoC} = 1 + hops + hops_{turn} + N \text{ clock cycles} \tag{4.2}$$

where *hops* refers to the number of routers a packet traverses straight through, $hops_{turns}$ is the number of routers in which the packet's flits turns, and N refers to the number of flits.

If a single router system is assumed, in the best case scenario, the two cores will be on opposite ports. Therefore, every flit traversing the router, will execute a single straight jump. Moreover, we omit the N factor of the equation. In the paper, the

authors calculate the latency of the NoC and therefore, each packet's latency is equal to the summed up latency of each one of its flits. In our case though, only the traversal of the first flit adds latency to the operation, since the arrival of the rest of the flits is pipelined with the operations that take place in the NI.

Therefore, the equation now becomes:

$$ZLL_{NoC} = 1 + 1 + 0 + 1 = 3 \text{ clock cycles} \tag{4.3}$$

Next, we calculate the system's ZZL for the two basic operations: the single read and the single write.

The ZLL for a single write operation is as follows: The Initiator NI adds zero latency for each one of the packet's 3 flits. The first flit requires 3 clock cycles (one hop) to enter and exit the NoC. The Target interface needs one clock cycle to store the head flit in a VC buffer, and one more until the body flit arrives with the bus address. Finally, the AHB slave adds two extra cycles of latency until it raises the HREADY signal, meaning that the operation is over. In total:

$$ZLL_{NoC\_W} = 0 + 2 + 3 + 0 + 2 = 7 \text{ clock cycles} \tag{4.4}$$

When dealing with a bus, in the best case, the master initiating a write will also be the system's default master, and therefore, will immediately get access to the bus, without spending time on arbitration. The bus' latency for that kind of operation is:

$$ZLL_{bus\_W} = 2 \text{ clock cycles} \tag{4.5}$$

Regarding the second operation, the single read, the ZZL is as follows: The Initiator does not add any latency when it outputs the flits to the NoC. The packet's head traverses the NoC in 3 clock cycles. The Target NI spends again 2 cycles until the address is driven to the AMBA slave, which requires two more cycles until it outputs the results in HRDATA. From there, and since the response's head flit has been sent already, the NI creates the tail of the response immediately. That flit needs three extra cycles to traverse the NoC. Finally, back in the Initiator NI, that flit will be stored in a VC buffer, and arrive at the master after in 1 clock cycle. In total:

$$ZLL_{NoC\_R} = 0 + 3 + 2 + 2 + 3 + 1 = 11 \text{ clock cycles} \tag{4.6}$$

The corresponding number for a single read in an AHB bus is similar to that of the single write:

$$ZLL_{bus\_R} = 2 \text{ clock cycles} \tag{4.7}$$

Table 4.1 summarizes the results.

In a real world setup, assuming an implementation in a 28 nm process, with square tiles with a side of 2.1 mm, we can estimate a delay of about 100 ps/1 mm of wire [32].

Interconnect	Operation	ZLL (clk cycles)
Bus	Single Write	2
NoC		7
Bus	Single Read	2
NoC		11

Table 4.1: Summarized Zero-Load Latency results of the baseline and NoC-based system for the two single operations.

In addition, a 4x4 concentrated mesh NoC with at most 32 cores, can operate with a clock period of about 400 ps [32]. Consequently, the NoC's ZLL is now calculated as:

$$ZLL_{NoC\_W} = 7 \cdot 400 = 2800\,\text{ps} \tag{4.8}$$
$$ZLL_{NoC\_R} = 11 \cdot 400 = 4400\,\text{ps}$$

On the other hand, for different bus implementations the system's period would vary. For example, assuming a bus-based system where the critical path consists of $log(N)$ wires, each one being 10 mm, where N refers to the number of connected cores:

$$ZLL_{bus} = log(N) \times 200\,\text{ps} \tag{4.9}$$

Therefore, for implementations with 4, 8, 16 and 32 cores:

$$ZLL_{bus\_4} = log(4) \times 200 = 400\,\text{ps} \tag{4.10}$$
$$ZLL_{bus\_8} = log(8) \times 200 = 600\,\text{ps}$$
$$ZLL_{bus\_16} = log(16) \times 200 = 800\,\text{ps}$$
$$ZLL_{bus\_32} = log(32) \times 200 = 1000\,\text{ps}$$

Based on those, Tables 4.2 and 4.3 summarize the results of the newly calculated latency.

SINGLE WRITE		
Connected Cores	ZLL NoC (ns)	ZLL Bus (ns)
4	2.8	0.8
8	2.8	1.2
16	2.8	1.6
32	2.8	2

Table 4.2: Single write operations ZLL calculated in ns for various system configurations for both systems.

Although there is a point where a NoC would be faster even for single operations, this is not our concern, since we aim in lower latency and higher throughput for

SINGLE READ		
Connected Cores	ZLL NoC (ns)	ZLL Bus (ns)
4	4.4	0.8
8	4.4	1.2
16	4.4	1.6
32	4.4	2

Table 4.3: Single read operations ZLL calculated in ns for various system configurations for both systems.

more complex operations. Through the example above though, we demonstrated a weakness of the bus. One, that is capitalized in order to build a faster interconnection system.

4.2 Latency Calculation from Simulation

In this section, we present the results regarding the system's latency for the supported operations: burst reads and writes of unknown length, and back to back reads and writes. Afterwards, we compare these results to those of the baseline system.

Using VHDL code, we described and simulated a set of different systems with 4, 8, 16 and 32 components, half of which are AHB masters, with the rest being AHB slaves. Moreover, we simulate operations of various lengths. For back to back operations, we run simulations for 32, 64, 128, 256, 512 and 1024 concurrent master requests. Regarding the unknown length bursts, the baseline system's core requested a set of 8 addresses, so we run simulations for 1, 2, 4, 8, 16, 32 and 64 requests in a row.

In each case, the latency in clock cycles has been extracted from simulation, and from there, the total latency in a time unit is calculated based on the system's size, as explained earlier.

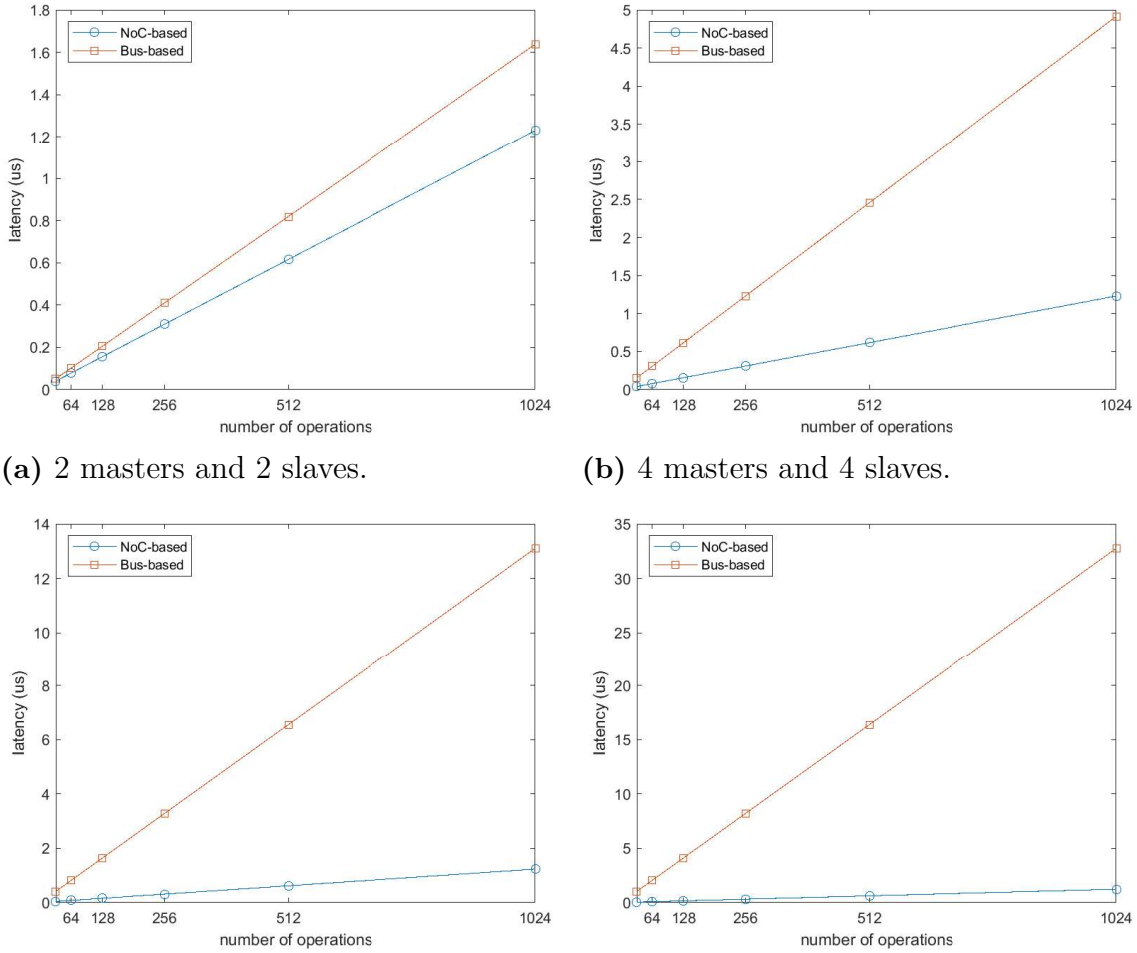
In the case of the NoC-based system, once again a concentrated mesh network has been used. As a traffic pattern, the *closest neighbor* was used, meaning that each NI is communicating with a NI that is connected to the same router if possible.

When simulating a bus, the baseline system has been used, with all the cores connected to the same AHB bus controller, which means that there are no bridges in the system.

4.2.1 Back to Back Operations

Figure 4.1 illustrates the latency for back to back writes in the two systems for a number of different implementations.

We observe that in all cases, the NoC-based system performs better. Furthermore, although in both cases the latency growth seems to be linear, the bus' latency exhibits a bigger slope, meaning that with more operations performed, the NoC becomes even better compared to the bus. Moreover, using a greater amount of



(a) 2 masters and 2 slaves.

(b) 4 masters and 4 slaves.

(c) 8 masters and 8 slaves.

(d) 16 masters and 16 slaves.

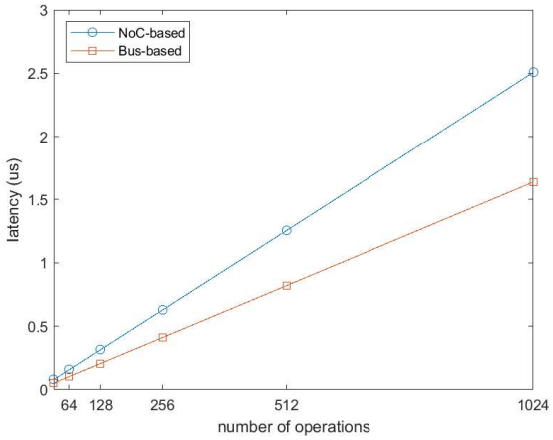
Figure 4.1: Back to back write operation latency for varying number of operations and number of AHB cores connected.

cores, leads to an even larger difference in latency. It is clear that the latency lines are relatively close when only 4 cores are used (Figure 4.1(a)), but this is not the case as the number of connected cores increases.

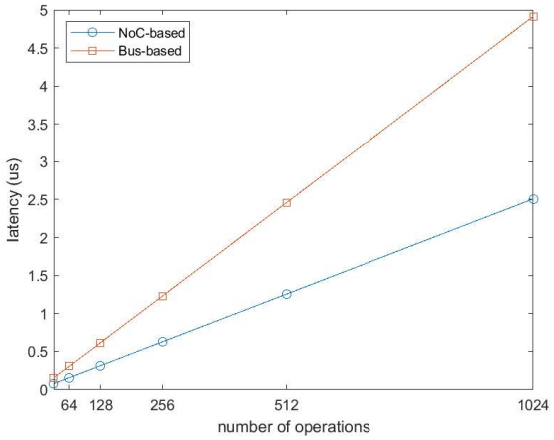
Figure 4.2 demonstrates the latency for read operations. The result of the first simulation, with only 2 masters, and 2 slaves, show that the proposed system exhibits lower latency compared to the bus-based system. This is not the case in the rest of the results, where once again, the NoC-based system manages to perform better.

Comparing the two systems, the one featuring the NoC, performs 8 times faster for back to back writing operations in smaller, but realistic systems, with only four cores, and up to 26 times faster in larger systems with 32 cores. In total, the NoC shows a latency improvement of $87\% - 96\%$ compared to the bus.

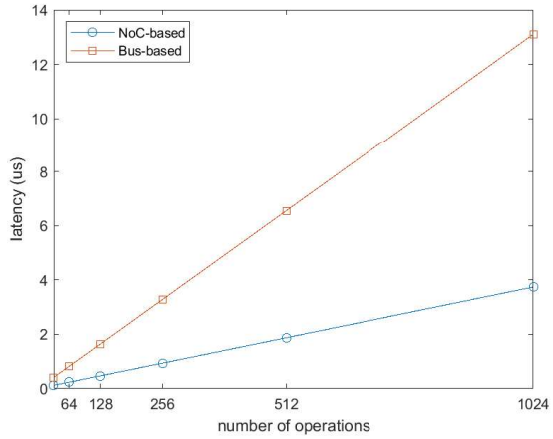
When it comes to back to back reading operations, the NoC has up to 4 times smaller latency in 4-core systems and up to 17 times in large 16-core implementations, which translates to $75\% - 94\%$ lower latency for the proposed design.



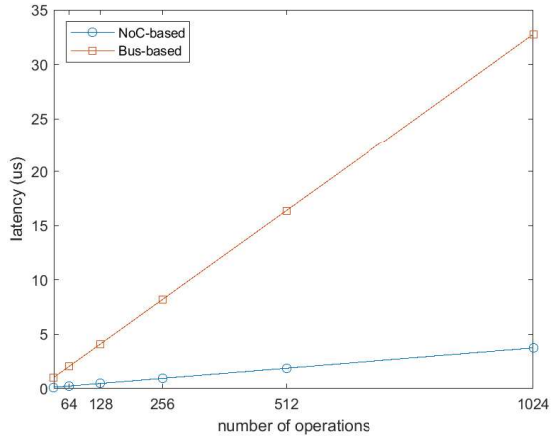
(a) 2 masters and 2 slaves.



(b) 4 masters and 4 slaves.



(c) 8 masters and 8 slaves.



(d) 16 masters and 16 slaves.

Figure 4.2: Back to back read operation latency for varying number of operations and number of AHB cores connected.

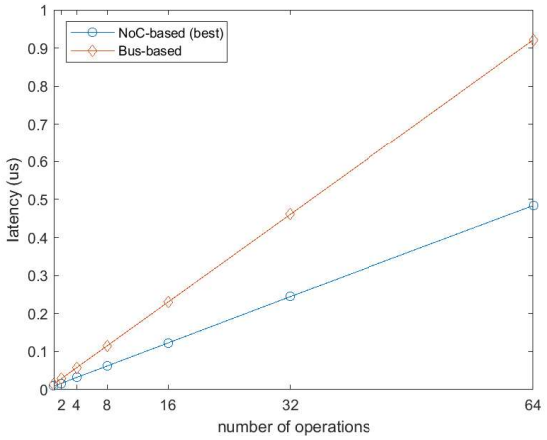
4.2.2 Burst of Unknown Length Operations

Simulation results for write bursts are presented in Figure 4.3. Bursts of length 8 have been used. For stressing the system, we selected to repeat the bursts 1, 2, 4, 8, 16, 32 and 64 times in different simulations.

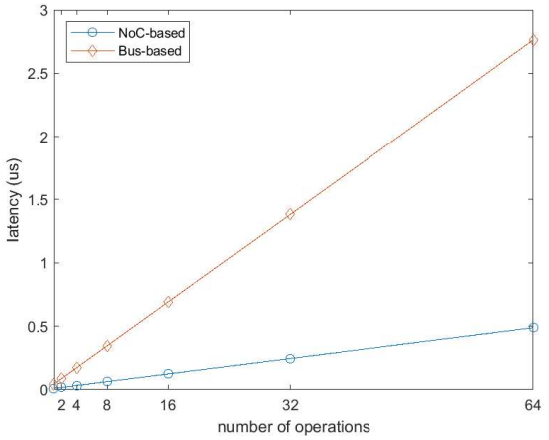
As can be observed from the figure, the NoC-based system performs better in all occasions. Once again, when more cores are connected, the NoC-based system scales better.

Lastly, Figure 4.4 shows the results for reading bursts. Since we have selected to use a prefetchable read of 8 addresses from the AHB slave, there are two corner cases regarding the number of data sent that are actually useful.

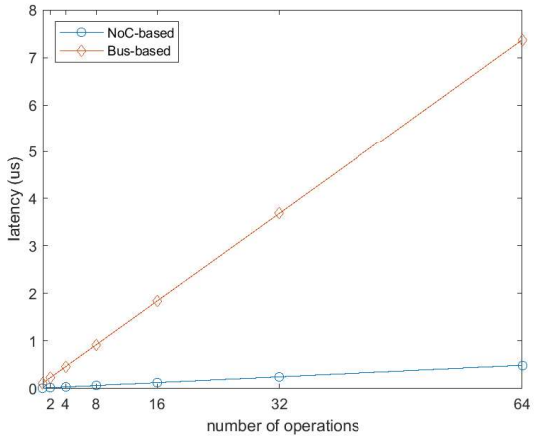
In the best case scenario, some multiple of 8 is requested and there are no wasted flits. In the worst case though, the master requests 1 flit more than a multiple of 8, in our case 9, so the Initiator NI, when the slave responds, has to spend 7 clock cycles idle, wasting valuable resources. Those two case are included in the figure.



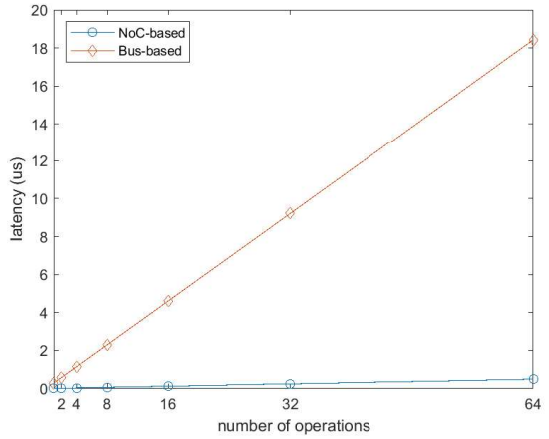
(a) 2 masters and 2 slaves.



(b) 4 masters and 4 slaves.



(c) 8 masters and 8 slaves.



(d) 16 masters and 16 slaves.

Figure 4.3: Burst write operation latency for varying number of operations and number of AHB cores connected.

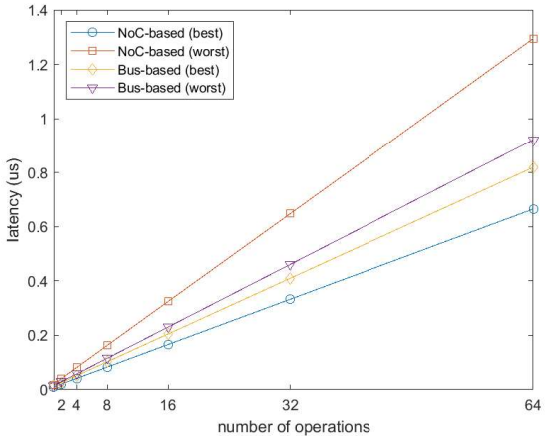
The first case (Figure 4.4(a)) seems very interesting, as the NoC's latency can either be lower or larger compared to that of the bus, depending on the burst length.

In every other case, for 8, 16 or 32 cores, the NoC-based system exhibits lower latency. In fact, the proposed system, when dealing with writing bursts, perform with up to 1.9 times lower latency in small 4-cores systems and 38 times in large 32-core systems, which translates to $47\% - 97\%$ lower latency.

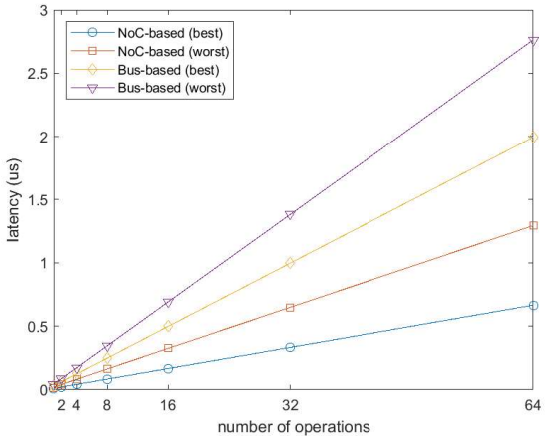
On the other hand, handling a reading burst, it manages to operate with 1.8 and 9 lower latency for 4-core and 16-core systems respectively, or an improvement of $44\% - 88\%$.

4.2.3 Sensitivity Analysis for varying injection rates

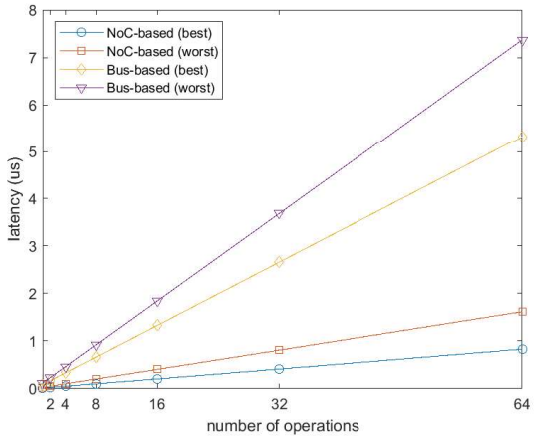
In this section we compare the latency of the two systems using different injection rates. The injection rate refers to the number of information that enters the interconnection system in a specific amount of time.



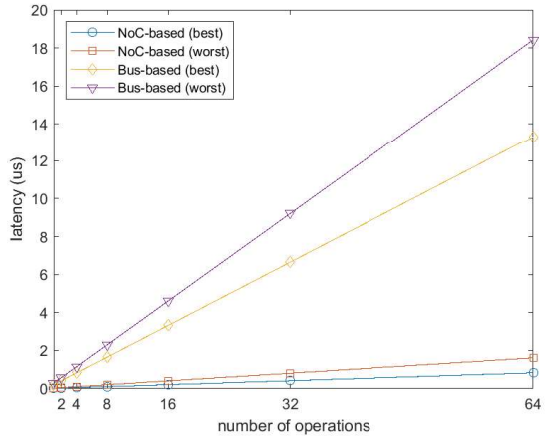
(a) 2 masters and 2 slaves.



(b) 4 masters and 4 slaves.



(c) 8 masters and 8 slaves.



(d) 16 masters and 16 slaves.

Figure 4.4: Burst read operation latency for varying number of operations and number of AHB cores connected.

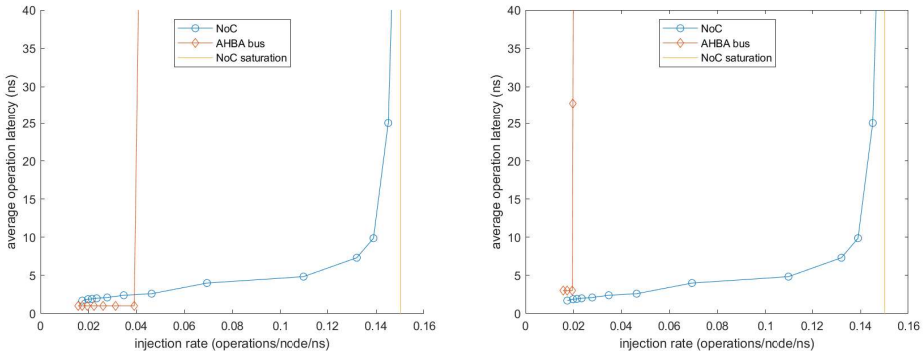
The goal of using different injection rates is to stress the system to a point where the bus or the NoC becomes congested. This way, the maximum throughput of each system can be defined.

For simulating a NoC-based system where the master has the ability to operate on different frequency relative to the interconnection, a large FIFO was inserted after each master core and before the NoC port. In this configuration, the master was always able to output data and store them in the FIFO, disregarding whether the credit control unit possess credits for the NoC. At the same time, the NoC, reads from the FIFO, instead of the master's output, at its own pace. The latency measure here, is the difference between the moment the Initiator creates and stores the head flit in the FIFO, and the time it reaches the Target's VC buffer's input.

When the master operates slower than the NoC, there is a low injection rate, and a situation arises where each newly produced flit is almost immediately consumed from the NoC. In this case, the average latency, approaches the head's ZZL of the burst write.

On the opposite side, when raising the injection rate, the master sends requests with a higher rate and the NI produces enough flits to create congestion in the NoC, until the network saturates. As a rule of thumb, the saturation limit is considered to be the point were the average latency is equal to 4 times the ZLL.

Different configurations with 1, 2, 4, 8 and 16 masters were studied for their average latency. Two of which are illustrated in Figure 4.5: the 4-core system with 2 masters and 2 slaves, and the 8-core system, including 4 masters and 4 slaves.



(a) 2 masters and 2 slaves. **(b)** 4 masters and 4 slaves.

Figure 4.5: Average operation latency over injection rate for different configurations.

For the experiment, we simulated 512 sequential burst operations. For the NoC, we assumed once more a concentrated 2D 4x4 mesh with 4 VCs of 5 registers each. For the clock period, the results from Table 4.3 were used. Lastly, the traffic pattern selected, was the closest neighbor, meaning that each master communicates with only one slave which is connected to the same router or the one next to it, minimizing this way the hops each flit has to perform to reach its destination.

As a measuring unit for the injection, we selected the operations/unit/ns. An operation refers to different amount of data depending on the system. When using a bus, a write burst of length 8, as the ones used before, comprises of 32 bytes of address that are transferred through the address bus and 32 bytes of data that are transferred through the data bus, a total or 64 bytes of 512 bits.

On the other hand, according to the NI's specifications, a burst write packet of length 8 consists of 10 flits of 37 bits each: a head flit carrying the control signals, one more for the first address, and 8 data flits. Therefore, a total of 370 bits are traversing the NoC during the operation.

Figure 4.5(a) includes the average operation latency of the two systems for a configuration with 2 masters and 2 slaves. The saturation point of the NoC is also illustrated. For low injection rates, up to 0.04 operations/unit/ns, the bus performs better than the NoC. There, both systems are really close to their ZLL latency, and the bus' ZLL is lower than the NoC's when only two masters are connected.

After that, the bus' latency increases very fast with the bus being saturated instantly. The NoC, saturates at the rate 0.15 operations/unit/ns. The NoC-based system exhibits 3.5× higher throughput compared to that of the bus-based design.

Figure 4.5(b) shows the second configuration with 4 masters and 4 slaves. In this case, although the NoC exhibits the same latency, the bus seems to saturate earlier than before. As a result, the discrepancy in terms of throughput is even larger, as the NoC-based systems now performs with 7.5× the throughput of the bus-based system.

The two Figures, highlight the merits of the NoC over the bus. In very low injection rates, and with only a few masters, the bus has enough time to attend to all of their requests before they create the next, and the system's latency is low. The moment the components start producing requests fast enough to create the slightest traffic, congestion is built and the system saturates.

The NoC has the ability to communicate with many masters simultaneously, and there lies the explanation of its better performance. Master units do not block the NoC when communicating, and as a result, multiple, and higher-frequency operating units can perform without saturating the network.

Table 4.4 summarizes the results of all the configurations that were investigated.

Number of Masters	Bus max IR (op/node/ns)	NoC max IR (op/node/ns)
1	0.089	0.15
2	0.040	0.15
4	0.019	0.15
8	0.011	0.15
16	0.004	0.15

Table 4.4: Highest pre-saturation injection rates possible for each systems and 5 different configurations.

4.3 Required Resources

As a unit for the required resources for implementation, we selected to present the number of slice registers utilized in an FPGA for the purposes of each of the two systems.

For measuring the resources, we analyzed the area report from Xilinx Vivado, with the Nexys 4 DDR as the selceted board, which features the XC7A100T-1CSG324C FPGA.

Besides the bus' area, which is constant, and independent of the connected components, the NoC's area depends on a number of parameters. The size of the mesh, which dictates the number of routers, the number of virtual channels, and the buffers of each virtual channel, all affect the area of the NoC. Each VC, and its buffers, are implemented from registers, so decreasing those parameters, greatly affects the system's area overhead.

Moreover, the total number of connected components, also influence the system's slice registers, since one NI is needed for each master or slave.

For those reasons, several syntheses were run. We synthesised systems with 32 components (4x4 mesh), 16 (3x2 mesh), 8 (2x1 mesh), and finally 4 components (single router or 1x1 mesh). In each configuration, we run variations with 2 or 4 virtual channels, and in each on of these two cases, measurements were taken after altering the number of buffers from 2 to 5.

Among all these configurations, two of them stand out, the two corner cases. First, the one that has a 4x4 2D concentrated mesh, with 4 VCs and 5 buffers, and it is the one that should offer the lower latency, and highest throughput, disregarding the consumed area. The results of that configuration compared to the bus' required slice registers, are presented in Table 4.5.

Interconnect	Connected Components	Slice Registers
AHB bus	<= 32	237
NoC (4 VCs, 5 buffers)	32 (4x4 Mesh)	17182
	16 (3x2 Mesh)	7907
	8 (2x1 Mesh)	3620
	4 (1x1 Mesh)	1738

Table 4.5: Slice registers utilization for a bus and NoCs with different sizes of a Concentrated Mesh topology, with 4 VCs and 5 buffers.

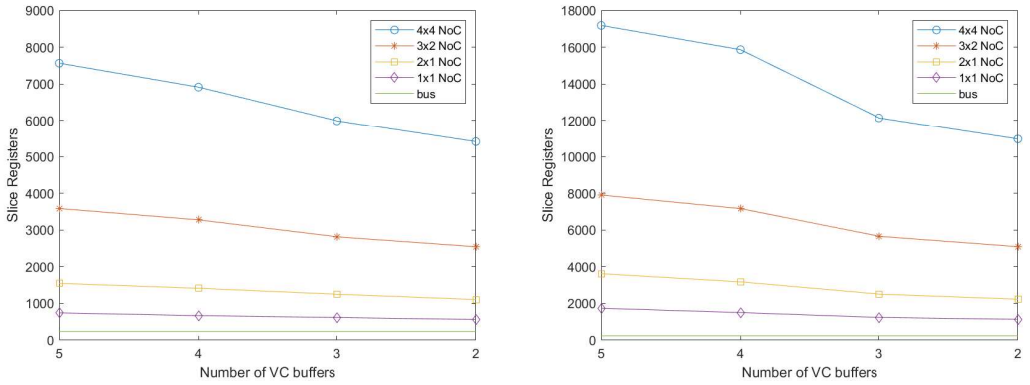
It is clear that the NoC utilizes a much larger area in terms of slice registers. Although we have already shown that it trades those resources with lower latency, the system exhibits an increased number in slice registers of $7 - 72\times$, compared to that of the bus, depending on the number of connected components.

The second case is the one with the lowest number of VCs and buffers, 2 and 2 respectively. The results are presented in Table 4.6.

Interconnect	Connected Components	Slice Registers
AHB bus	<= 32	237
NoC (2 VCs, 2 buffers)	32 (4x4 Mesh)	5415
	16 (3x2 Mesh)	2544
	8 (2x1 Mesh)	1106
	4 (1x1 Mesh)	562

Table 4.6: Slice registers utilization for a bus and NoCs with different sizes of a Concentrated Mesh topology, with 2 VCs and 2 buffers.

This time, the system requires less slice registers for its implementation. Specifically the NoC and the required NIs would consume $2.4 - 23\times$ the resources of the bus. Moreover, we expect from these systems to exhibit higher latency, since they have lower capabilities. Regarding the throughput, it is expected to still be higher than the bus', since a decrease in the VCs from 4 to 1, results in the system lowering its throughput by around 30% [26].



(a) Comparing the bus with systems with 4 virtual channels.

(b) Comparing the bus with systems with 2 virtual channels.

Figure 4.6: Comparing area utilization of the bus, with a number of systems with 2-D concentrated mesh topology, and various number of VCs, nodes and VC buffers.

Figure 4.6 presents the results for all the different configurations.

Comparing the results for 4 VCs in Figure 4.6(a), with those for 2 VCs in Figure 4.6(b) we immediately see a decreased area overhead. Moreover, each time the number of buffers, or connected component decreases, so does the area.

4.4 Summary

Based on the presented simulation results, we can conclude that in most cases, the NoC-based system exhibits lower latency in complex operations compared to that of the bus-based system. Furthermore, when dealing with larger scale systems, 8 cores and above, the NoC largely outperforms the bus in terms of latency and throughput.

In particular, the NoC outperforms the bus $87\% - 96\%$ in concurrent write operations, $75\% - 94\%$ in concurrent reads, $47\% - 97\%$ in write bursts, and finally $44\% - 88\%$ in read burst operations.

Regarding the throughout of each system, the NoC-based system performs with an improved throughput of $1.68 - 37.5\times$ for configurations with 1 to 16 connected masters.

Lastly, the NoC together with the designed NI requires from $2.4-23\times$ more resources for small implementations that have 2 VCs with 2 buffers each, and up to $7 - 72\times$ more resources for faster systems that have 4 VCs with 2 buffers each.

As a result we can conclude that the most promising combinations are the ones with 4 and 8 connected components, where the NoC-based implementation largely outperforms the bus-based system in terms of latency and throughput. At the same time, using 2 VCs and 2 buffers keeps the required resources within acceptable limits.

5

Conclusions and Future Work

In this work, a discussion on Networks-on-Chip and buses has been provided to highlight the main elements of each one of these technologies. Using that knowledge, two versions of the designed NI were described in detail, both theoretically and by providing simulation results. One of the NIs, the Initiator is specifically designed to communicate with AHB masters, while the other, the Target, specializes in the AHB slave communication. The architecture of both NIs was discussed, including the decisions and requirements that led us to the current design. Every part of those interfaces was analyzed in depth. Finally, simulation results that demonstrate the system's capabilities were presented and compared with those of the baseline system.

Our main goal was to capitalize on the NoC's better scalability, to improve the baseline system in terms of latency and throughput. In order for the NoC's advantages to surface, we increased the system's complexity by connecting more cores to it, and escalated the traffic in the system by increasing the injection rate. The system's main disadvantage, that of the increased resources (area) requirements, was also addressed. Various implementations were explored to calculate the hardware resources needed for different systems. With the exception of one case, the NoC-based system exhibits higher area overhead compared to the bus-based design. However, the proposed system compensates for that disadvantage by improving the latency and throughput of the system. All in all, the NoC-based system offers a trade-off between area overhead, and therefore power, and performance. On applications where area overhead and power consumption are limited, the more complex configurations may not be preferred or even afforded. Still, for small systems, a slightly increased area overhead, could offer great improvement in performance.

This thesis' contributions are the following:

- We designed and presented two Network Interfaces that are capable of communicating with both the network and the AMBA cores connected to the system.
- The proposed Network Interfaces allow a normal integration of the NoC in the provided system, without affecting in any way the AHB cores.
- The newly designed system is able to perform the same AHB operations as the baseline, while exhibiting a lower latency and higher throughput for increased complexity and traffic. The system deals with one of the bus' greatest disadvantages, since it has the ability to scale better and handle the increased traffic flow. Using logic simulation, we were able to verify that claim.

- The system is designed in such a way, that its parameters can be altered with only little effort.
- The NoC-based system withstands a throughput of 1.68× for single master systems and up to 37.5× for 16-master systems, compared to that of the bus.
- The proposed system, in terms of latency, outperforms the bus 87%-96% in concurrent write operations, 75%-94% in concurrent reads, 47%-97% in write bursts, and finally 44%-88% in read burst operations.
- The NoC, together with the NIs, requires an area overhead which is $2.4 - 23\times$ higher compared to that of the bus, for systems that have 2 VCs with 2 buffers each, and up to $7 - 72\times$ more area for faster systems that have 4 VCs with 2 buffers each.

5.1 Future Work

The work presented in this thesis has room for expansion. Due to the depth and complexity of interconnection networks, the designed Network Interface can be advanced in multiple ways in the future, some of which are the following:

- It would be really interesting for the NI to be implemented in one of the FPGAs used by Cobham Gaisler. This way, aside from verifying the design, a real-world application could be executed, and the system's latency could be compared to that of the baseline system's implementation.
- The NIs set of operations are limited, as was explained earlier. In the future, the interface could support the full range of AHB operations, as described in the AHB protocol.
- Although currently the NI supports the AHB protocol, it would be advantageous for an interface to be able to communicate in other protocols too, like the AMBA AXI [20]. At the moment the functionality can be added only by using AHB-to-AXI adapters [1].
- The baseline systems consisted of a single bus with a capability of no more than 32 cores connected. Naturally, that number was also used when designing the NI. In reality, the norm would be to use bridges in an attempt to lower the system's latency. Therefore, a possible expansion of this work, would be to replace a more complex set of buses and bridge and connect more than 32 cores.
- Despite the fact that our baseline system uses a width of 32-bits for it address and data bus, this is not always the case. A possible feature that could be added to the NI, is the capability of handling different channel widths.
- One of the AHB features that is missing from our NI, is supporting the AHB's lock signal. Although locking a bus can be helpful for a specific operation, it contradicts the full purpose of the Network-on-Chip. The idea behind the network is to distribute flit simultaneously in different destinations, and thus locking the whole structure was not an option. Later, a different approach for prioritizing certain packets can be developed and implemented.

- The NI features an arbiter that is responsible for selecting the next VC that the interface FSM will read from or write to. The arbitration method selected is "bit priority", which decreases the unit's complexity, but can lead to unfair decisions. Researching different arbitration methods could lead towards improving the NI, always keeping in mind the affect it would have to the system's area overhead.
- The packet format was selected in a way that serves the NoC's and AMBA's needs, while being versatile enough to cover possible future needs using some of the unused bits. A possible improvement of the NI could be the addition of some error checking bit, as for example a Cyclic Redundancy Check (CRC).

Bibliography

[1] Cobham Gaisler, "GRLIB IP Library User's Manual," 2020.

[2] ARM, "AMBA Specification (Rev 2.0)," 1999.

[3] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, pp. 70–78, Jan. 2002.

[4] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[5] Cobham Gaisler, "GRLIB IP Core User's Manual," 2020.

[6] R. Bhoedjang, T. Ruhl, and H. Bal, "User-level network interface protocols," *Computer*, vol. 31, pp. 53–60, Nov. 1998. Conference Name: Computer.

[7] P. Steenkiste, "A high-speed network interface for distributed-memory systems: architecture and applications," *ACM Transactions on Computer Systems*, vol. 15, pp. 75–109, Feb. 1997.

[8] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes, "Hamlyn: a high-performance network interface with sender-based memory management," *Proc. Hot Interconnects*, 1995.

[9] G. Blair, A. Campbell, G. Coulson, F. Garcia, D. Hutchison, A. Scott, and D. Shepherd, "A network interface unit to support continuous media," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 264–275, Feb. 1993. Conference Name: IEEE Journal on Selected Areas in Communications.

[10] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, pp. 4–17, Jan. 2005. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[11] S. Saponara, T. Bacchillone, E. Petri, L. Fanucci, R. Locatelli, and M. Coppola, "Design of an NoC Interface Macrocell with Hardware Support of Advanced Networking Functionalities," *IEEE Transactions on Computers*, vol. 63, pp. 609–621, Mar. 2014. Conference Name: IEEE Transactions on Computers.

[12] X. Yang, Z. Qing-li, F. Fang-fa, Y. Ming-yan, and L. Cheng, "NISAR: An AXI compliant on-chip NI architecture offering transaction reordering processing," in *2007 7th International Conference on ASIC*, pp. 890–893, Oct. 2007. ISSN: 2162-755X.

[13] M. Ebrahimi, M. Daneshtalab, P. Liljeberg, J. Plosila, and H. Tenhunen, "A High-Performance Network Interface Architecture for NoCs Using Reorder Buffer Sharing," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, (Pisa, Italy), pp. 546–550, IEEE, Feb. 2010.

[14] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen, "Memory-Efficient On-Chip Network With Adaptive Interfaces," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 146–159, Jan. 2012. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[15] L. Fiorin, L. Micconi, and M. Sami, "Design of Fault Tolerant Network Interfaces for NoCs," in *2011 14th Euromicro Conference on Digital System Design*, pp. 393–400, Aug. 2011.

[16] H. Kariniemi and J. Nurmi, "NoC Interface for fault-tolerant Message-Passing communication on Multiprocessor SoC platform," in *2009 NORCHIP*, pp. 1–6, Nov. 2009.

[17] "Open Core Protocol (OCP) Files."

[18] B. a. abdelkrim zitouni and r. tourki, "Design and implementation of network interface compatible OCP For packet based NOC," in *5th International Conference on Design Technology of Integrated Systems in Nanoscale Era*, pp. 1–8, Mar. 2010.

[19] T. Tayachi and P.-Y. Martinez, "Integration of an STBus Type 3 protocol custom component into a HLS tool," in *2008 3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, pp. 1–4, Mar. 2008.

[20] "AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite," p. 306, 2003.

[21] M. Ebrahimi, M. Daneshtalab, N. P. Sreejesh, P. Liljeberg, and H. Tenhunen, "Efficient network interface architecture for network-on-chips," in *2009 NORCHIP*, pp. 1–4, Nov. 2009.

[22] K. Swaminathan, G. Lakshminarayanan, and S.-B. Ko, "High Speed Generic Network Interface for Network on Chip Using Ping Pong Buffers," in *2012 International Symposium on Electronic System Design (ISED)*, Dec. 2012.

[23] K. Swaminathan, G. Lakshminarayanan, F. Lang, M. Fahmi, and S.-B. Ko, "Design of a low power network interface for Network on chip," in *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–4, May 2013. ISSN: 0840-7789.

[24] K. Tatas, K. Siozios, D. Soudris, and A. Jantsch, *Designing 2D and 3D Network-on-Chip Architectures*. New York, NY: Springer New York, 2014.

[25] S. Ma, *Networks-on-chip: from implementations to programming paradigms*. Boston, MA: Elsevier, 2014.

[26] A. Ejaz, V. Papaefstathiou, and I. Sourdis, "HighwayNoC: Approaching Ideal NoC Performance With Dual Data Rate Routers," *IEEE/ACM Transactions on Networking*, pp. 1–14, 2020.

[27] A. Ejaz, V. Papaefstathiou, and I. Sourdis, "FreewayNoC: A DDR NoC with Pipeline Bypassing," in *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, (Turin), pp. 1–8, IEEE, Oct. 2018.

[28] A. Ejaz, V. Papaefstathiou, and I. Sourdis, "DDRNoC: Dual Data-Rate Network-on-Chip," *ACM Transactions on Architecture and Code Optimization*, vol. 15, pp. 1–24, June 2018.

[29] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, "SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 25–36, June 2014. ISSN: 1063-6897.

[30] A. Psarras, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos, "ShortPath: A Network-on-Chip Router with Fine-Grained Pipeline Bypassing," *IEEE Transactions on Computers*, vol. 65, pp. 3136–3147, Oct. 2016.

[31] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, pp. 51–61, Sept. 2007. Conference Name: IEEE Micro.

[32] A. Ejaz and I. Sourdis, "FastTrackNoC: A DDR NoC with FastTrack Router Datapaths," *Technical Report*.

A

Appendix 1

Name	Source	Description
HCLK Bus clock	Clock source	This clock times all bus transfers. All signal timings are related to the rising edge of HCLK.
HRESETn	Reset controller	The bus reset signal is active LOW and is used to reset the system and the bus. This is the only active LOW
HADDR[31:0] Address bus	Master	The 32-bit system address bus.
HTRANS[1:0] Transfer Type	Master	Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE Transfer Direction	Master	When HIGH this signal indicates a write transfer and when LOW a read transfer.
HSIZE[2:0] Transfer size	Master	Indicates the size of the transfer, which is typically byte (8-bit), half-word (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HBURST[2:0] Burst type	Master	Indicates if the transfer forms part of a burst. Four, eight and sixteen beat bursts are supported and the burst may be either incrementing or wrapping.
HPROT[3:0] Protection control	Master	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection. The signals indicate if the transfer is an opcode fetch or data access, as well as if the transfer is a privileged mode access or user mode access. For bus masters with a memory management unit these signals also indicate whether the current access is cacheable or bufferable.
HWDATA[31:0] Write data bus	Master	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HSELx Slave select	Decoder	Each AHB slave has its own slave select signal and this signal indicates that the current transfer is intended for the selected slave. This signal is simply S combinatorial decode of the address bus.
HRDATA[31:0] Read data bus	Slave	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended.

		However, this may easily be extended to allow for higher bandwidth operation.
HREADY Transfer done	Slave	When HIGH the HREADY signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer. Note: Slaves on the bus require HREADY as both an input and an output signal.
HRESP[1:0] Transfer response	Slave	The transfer response provides additional information on the status of a transfer. Four different responses are provided, OKAY, ERROR, RETRY and SPLIT.
HBUSREQx	Master	A signal from bus master x to the bus arbiter to indicate that the bus master requires the bus. There is an HBUSREQ signal for each bus master in the system, up to a maximum of 16 bus masters.
HLOCKx Locked transfer	Master	When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is LOW
HGRANTx Bus grant	Master	This signal indicates that the bus master is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when HREADY is HIGH, so the master gets access to the bus when both HREADY and HGRANT are HIGH.
HMASTER Master Number	Arbiter	These signals from the arbiter indicate the bus master that is currently performing a transfer and is used by the slaves that support SPLIT transfers to determine the master that is attempting an access. The timing of HMASTER is aligned with the timing of the address and control signals.
HMASTLOCK Locked sequence	Arbiter	Indicates that the current master is performing a locked sequence of transfers. This signal has the same timing as the HMASTER signals.
HSPLITx[15:0] Split completion Request	Slave (SPLIT capable)	A split-capable slave uses the 16-bit split bus to indicate to the arbiter the bus masters that can reattempt a split transaction. Each bit of this split bus corresponds to a single bus master.

Table A.1: The AMBA AHB signals [2]