Virtual Circuits in Network-on-Chip

Master of Science thesis nr.: 87

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Engineering

Lyngby, 18th of September 2006

Supervised by Jens Sparsø
Co-Supervised by Mikkel Stensgaard

s011493 - Christian Place Pedersen

Abstract

This project investigates virtual circuits circuits in Network-on-Chip, specially how virtual circuits can be used to give guarantees on bandwidth and latency requirements. The project is in two parts: A study, which investigates guaranteed service in NoC implementations and an extension of the virtual circuit support in MANGO.

The study conducted in this project examines seven different NoC implementations - both academic and industrious. The main focus of the study is to investigate the different approaches to guaranteed service.

The second part of this project describes the implementation of reconfigurable virtual circuits in MANGO, an asynchronous NoC developed in the PhD thesis by Tobias Bjerregaard, at IMM, DTU. The implementation presented here offers flexible programming of virtual circuits as well as reuse of virtual channels.

Acknowledgements

This Master of Science thesis has been carried out at the Computer Science and Engineering division of Informatics and Mathematical Modelling department at the Technical University of Denmark, during 2006.

I would like to thank my supervisor Jens Sparsø for his guidance, discussions and support during the course of this project. I would also like to thank co-supervisor Mikkel Stensgaard for his most valuable inputs, ideas and support. Thanks to Tobias Bjerregaard for providing insight in MANGO and to Matthias Stuart for the discussions on MANGO and NoC implementations.

Contents

List of Figures

CHAPTER 1

Introduction

A System-on-Chip (SoC) is as the name suggests, a system made up of smaller subsystems on the same chip. These subsystems can be anything from general processing units over application specific units to memory modules and interfaces. Because the subsystems can be tailored to a specific application or be very general and broad, SoCs can cover a great range of applications and products, from small, low power embedded systems that have very thight constrains on area and power to large general systems that can handle a broad range of applications. By introducing subsystems into the design process, individual components can be designed and verified which makes the process of designing and verifying the entire system easier. In addition, the SoC approach allows for reuse of subsystems, which enables the creation of intellectual property cores (IP cores) that can be designed and used in several SoCs. This allows for a large amount of reuse which greatly decreases the time it takes to construct even large and complex SoCs.

The SoC approach to system design allows the system designer to plug IP cores toghether, through a common interface via an interconnect. In many cases, this interconnect is a bus structure or point-to-point links. As CMOS technology has scaled down, the cost of wires - i.e. communication - can no longer be neglected. To cope with the increased capacitance on a bus as more IP cores are connected, segmented busses are being used, but at the cost of increased latency and lower bandwidth for each IP core. To maintain and even increase the performance of the communcation infrastructure in very large SoCs, alternative communication infrastructure are being investigated.

In recent years Network-on-Chip (NoC) has received increasing interest as an approach to handle the on-chip communication of large scale SoC. A network is characterised by having segmented links connected through intermediate points that route information incoming on one link to another of the connected links. NoC addresses many of the challenges that rise from scaling busses and point-to-point links to facilitate the increasing number of IP cores. Many of the solution can be adopted from research already done within the field of large scale networks, such as the Internet.

New challenges and possibilities arises from NoC, such as guaranteeing bandwidth and latency between IP cores.

As all IP cores are connected to the network, NoC enables logical point-to-point links between all IP cores. This increases the flexibilty of the SoC, which can be used by introducing use cases. A use case is a way to describe the use of the network. The SoC can thus handle different applications with the same hardware, which gives the chip a larger market.

1.1 Previous Work

Due to the rise in NoC research, several implementations of the NoC concept have been proposed and described both in the academic world and in the industry. The implementations investigate different aspects of the design spectrum within NoC.

The following networks are developed and described in the academic world. $\times$pipes [4] is a library of network components that can be put togheter using a designated compiler, [19], that allows different network architectures. CHAIN [1] and SPIN investigate different tree-topologies, but differ greatly as CHAIN is an asynchronous implementation where SPIN is synchronous. QNoC [10] is another synchronous network that uses virtual channels to give statistical guarantees on connections in the network. SoCBUS establishes an end-to-end connection for all data transfers across the network. MANGO [5] is a feature rich asynchronous NoC that offers among other things hard guarantees on bandwidth and latency between the connected IP cores.

In the industry, the NoC adoption has been sparse, but academic implementations have resulted in spin offs as fx. CHAIN [2] which is now developed by Silistix [18]. Philips Research has also implemented and described a synchronous NoC called Æthereal [16] that offers both best effort traffic and guaranteed service connections. The guaranteed service connections divides the time to multiplex the links in the network. In addition, Æthereal can be reconfigured to support different application requirements at runtime. Lastly IBM, Sony and Toshiba's Cell processor [20] uses a so called Element Interconnect Bus which bears great resemblance to the NoC concept.

1.2 Project Description

This project will build upon MANGO inorder to investigate how more flexibility can be added to the guaranteed service connections used by MANGO to provide hard guarantees on bandwidth and latency. Allowing for more dynamic uses of the MANGO NoC.

The MANGO NoC has been designed at IMM DTU, by Tobias Bjerregaard in his PhD thesis [5]. MANGO is a novel implementation of the NoC concept and is characterised by being asynchronous and providing hard guarantees on both bandwidth and latency. This is accomplished by introducing guaranteed service connections based on virtual channels on the intermediate links inside the network, creating end-to-end virtual circuits. MANGO's virtual circuits can be setup - at runtime - between any two IP cores, i.e. the virtual circuits can be tailored to meet the requirements of a wide range of applications on the same hardware.

Furthermore this project will take a look on the state of current NoC implementations, with focus on the guaranteed services approaches taken.

The main challenge in this project is to determine and in turn implement the extra feature that is needed to allow for dynamic reconfiguration of connections in the MANGO NoC. The solution will aim to provide a flexible approach to how the virtual circuits can be used and created.

1.3 Report Structure

The report is in two major parts. Part one, consisting of chapter 2, 3 and 4, is an introduction to the NoC domain. Chapter 2 describes general NoC concepts and introduces Quality-of-Service as it is used in this project, furtheremore use cases as a general way of describing traffic patterns in SoC design is described. Lastly, the chapter introduces asynchronous circuits. In chapter 3, a study of different NoCs and their take on Quality-of-Service is described. Chapter 4 discusses virtual channel specific topics.

Chapters 5, 6 and 7 make up part two about MANGO and the modifications and extensions that are done in this project. Chapter 5 thoroughly describes the MANGO NoC, with focus on the parts that must be taken into consideration when guaranteeing service. The extensions and modifications conducted in this project are described and discussed in chapter 6. Chapter 7 describes the test, which have been conducted as part of this project. Finally chapter 8 and 9 holds the discussion and conclusion respectively.

CHAPTER 2

Domain Introduction

This chapter introduces the basic concepts built upon in this project. First, an introduction to Network-on-Chip is given, followed by a description of which services can be offered by a communication structure. The term use case is then introduced and described in the context of NoCs. Lastly, a short introduction to asynchronous circuits is given.

2.1 Network-on-Chip

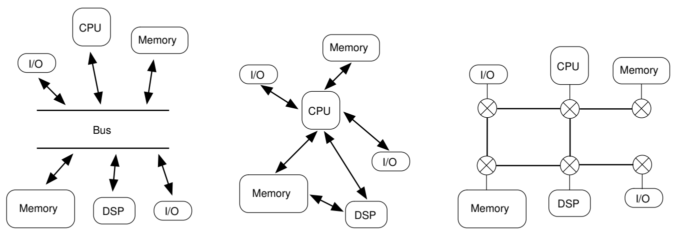
The NoC concept shares many concepts and features with large scale network, such as those created between computers in fx. the Internet. One of the great differences is that a NoC is static in contrast to most large scale networks that have to cope with ever-changing routes and number of connected users. This affects the design choices such as the transmission protocol and flow control. In addition, because a NoC is a closely coupled environment, data loss and corruption can be neglected in most cases.

In this section, basic concepts, distinctive features and terms of Network-on-Chip (NoC) will be described and discussed. Then more advanced topics within NoC are described. These topics include interfaces, topology, protocols and flow control. Lastly, different abstraction levels of the NoC and their effect on the overall SoC design process will be discussed.

This section is based on the terms and concepts introduced in [9], where a survey of the current state of NoC research and practises can be found.

2.1.1 Basic Concepts

As the CMOS technology develops, transistors shrink in size, causing the wire resources to make up a larger part of the overall system. Therefore, the on-chip communication structure also takes up more of the overall ressources in a SoC. In addition, the most commonly used types of communication structures in SoC - bus (figure 2.1(a)) and point-to-point (figure 2.1(b)) links, scale poorly when more subsystems or IP cores are added to the system. NoC-based systems as the one seen in figure 2.1(c) try to cope with the problems that arise when scaling a bus and point-to-point links.



(a) A bus. (b) Point-to-point links. (c) An on-chip network.

Figure 2.1: Examples of communication structures: 2.1(a): a bus, commonly used in SoC; 2.1(b): point-to-point links, used for high performance links and 2.1(c): network, segmented links connected by routers.

Figure 2.2 shows the basic elements of a NoC in SoC. These basic elements can also be thought of as abstraction levels which will be detailed in section 2.1.7. The basic elements of a NoC are:

IP Cores are not part of the NoC, but are shortly described here. They are connected to the network interfaces and make use of the services provided by the network to gain access to another resource in the system. A core can typically either initiate requests, be the target of these requests or have both capabilities. The cores have different roles ranging from general processing units over signal processors and I/O controllers to storage such as RAM.

Network Adapters (NA) provide the *interface* to the IP core through which the network *services* can be accessed. The NA encapsulates the data provided on the core interface into *packets*, which are again spilt into flow control units (flits) before they are sent into the network through the NAs network interface. Flits are used in order to minimise the need for buffers and wires in the network.

Routers or switches routes data on the links in accordance with a chosen routing protocol. The router *routes* the data flits from one point in the network to another. The routers are arranged in a *topology* and they handle *forwarding* and *flow control* of data between the network adapters. The routers generally connect a number of links and one or more network adapters.

Links are a group of bundled parallel wires that run from one point to another point on the chip. The data width of the links depends on the number of wires bundled together, and how many control signals that are used. On the links, data is transmitted one flit at the time. Links connect routers to other routers and to network adapters.

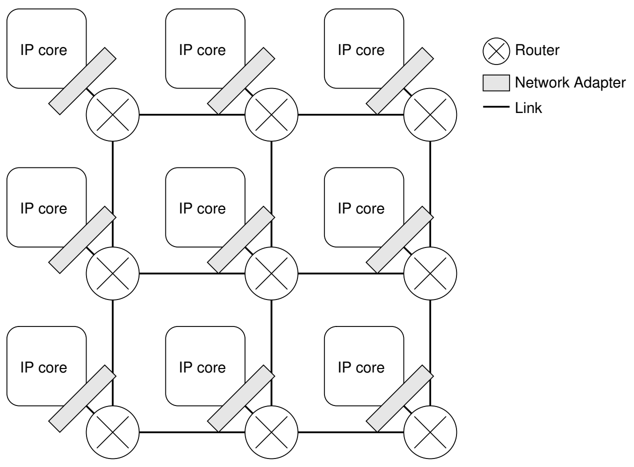


Figure 2.2: A 3x3 2 dimensional NoC. The basic elements of a NoC include: Network adapters, routers and links. Connected to the network adapters are IP cores, such as computational units, memory and I/O.

2.1.2 Interface

Because a NoC is just another implementation of a communication structure suited for SoC, each communication structure should present the same interface to the core. By using a common interface or socket, the system designers can choose different communication structures and other IP cores that suite their specific needs. An example of such a common interface is the Open Core Protocol (OCP) [30] interface which provides a master and a slave interface for initiating and receiving request respectively. To make use of some special services that are provided specifically by a communication structure, the system must be aware of these services. In the case of NoC an IP core is said to be network aware and can therefore make use of the extra features and services. A NoC can for example offer guaranteed bandwidth between any two of its interfaces in the network, an IP core must be aware of this feature in order for it to setup connections.

2.1.3 Network Adapter

The network adapter (NA) communicates with the IP cores through what is called the core interface (CI) in this report. The NA transforms the request from the core into a request that can be transmitted by the network, through the network interface (NI). Figure 2.3 shows the interfaces and how they are used to make the network transparent for the IP cores. This allows for the network to be clocked by another clock

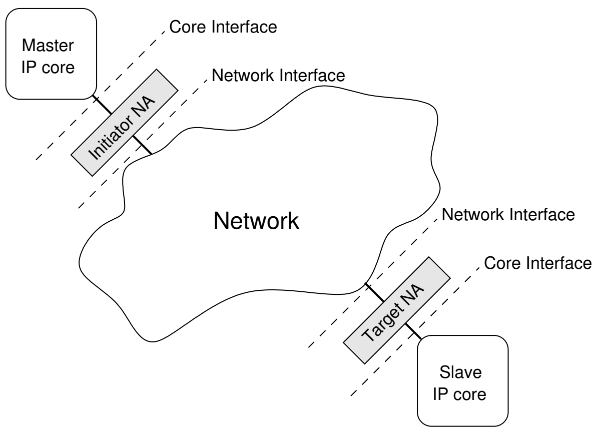


Figure 2.3: The NA translates between a core interface presented to the cores and a network interface presented to the network.

or no clock at all, allowing Globally Asynchronous Locally Synchronous (GALS) systems. The NA then handles the synchronisation between the different domains, further allowing each IP core to be clocked individually.

The figure also shows how the terms regarding master-slave interface are used in this report. A master IP core initiates request to a target slave IP core. Therefore, the master IP core is connected to a slave port on the network and a slave core is connected to a master port on the network. To avoid confusion, the term initiator NA is used in this report when describing the NA connected to a master IP core and the slave IP core is connected to the target NA. This means that requests are transmitted from the initiator to the target, while responses are transmitted from the target to the initiator. It should be noted that a target NA is always the target of requests and is thus the NA that receives a request and generates a response if needed.

2.1.4 Topology

The network topology defines how the routers in the network are connected. A collection of network topologies that are used in NoCs are shown in figure 2.4. Topologies can be grouped in several ways: direct networks and indirect networks; homogeneous and inhomogeneous.

In direct networks, each router has at least one IP core directly connected, whereas in the indirect networks some routers are only connected to other routers and only designated routers are connected to IP cores. A mesh network, figure 2.4(a), is a typical example of a direct network and tree structured networks, figure 2.4(d) and 2.4(d), are examples of indirect networks where only the leafs of the tree are IP cores. However, an indirect mesh network can be created by only connecting IP cores at the edges of mesh and the trees can be made direct by adding IP cores to all nodes in a tree.

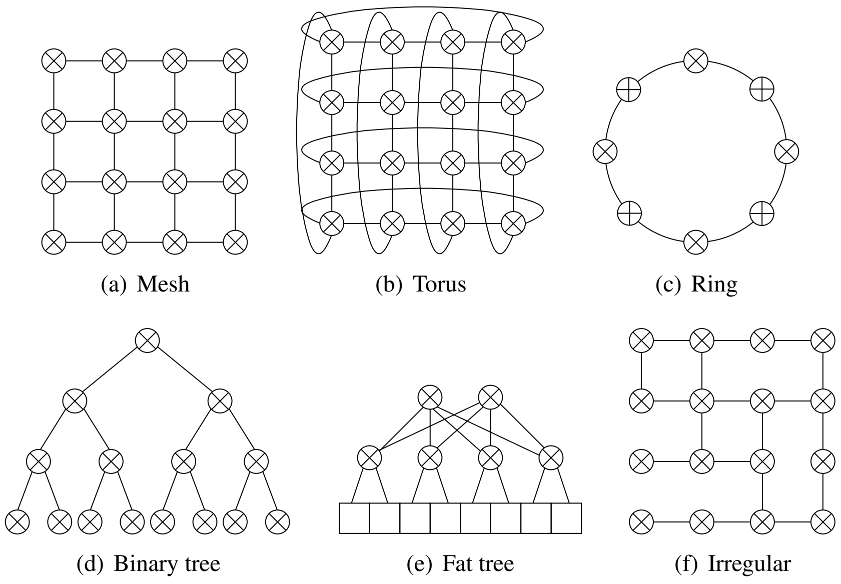


Figure 2.4: Different NoC topologies.

A homogeneous network is a network that scales in a predictable way, such as mesh, ring (figure 2.4(c)) and balanced trees do. Inhomogeneous networks such as the irregular network shown in figure 2.4(f) are arranged to suit the needs of a specific application in order to optimise the overall system, i.e. area, power or speed. Optimising the network by clustering IP cores that have a high amount of communication with each other decreases latency and switching activity resulting in better performance and a lower power consumption. Clustering can be done by creating irregular topologies or by mixing topologies to form a more optimised topology for that specific application. The cost however is that the flexibility of the system is lowered because while the system might perform very good when running that particular application, it may not be a viable solution for other applications.

Figure 2.4(b) shows a torus network that differentiates itself from the mesh network by connecting opposite edges and by usually having unidirectional links, where most other topologies favour bidirectional links. The torus network performs poorly on applications with high amounts of local traffic, but comes at a relative low cost in area due to having unidirectional links. Ring networks 2.4(c) are not common in NoC, but as shall be seen in 3.3 they can make for an interesting topology.

2.1.5 Protocol

A protocol describes how data is transferred from one point in the network to another point. In line with [9] switching is used here as transport of data, while routing determines the path the data takes.

Most NoCs today employ *packet switching* where packets are forwarded through the network in flits. The most commonly used forwarding strategies are store-and-forward and wormhole. In a *store-and-forward* routing based network the router receives and stores the entire packets, i.e. it buffers multiple flits, and then forwards the packet based on the routing information held within the packet. Store-and-forward is the strategy most commonly used in macro networks, but can also be found in some NoC implementations. *Wormhole* routing is chosen in most NoC implementation, because it tries to minimise packet latency. The routing decisions in the routers are made on basis of the first flit in the packet and all the subsequent flits are forwarded as they arrive, thus also minimising the need for deep buffers in the routers. The downside of wormhole routing is that a large part of the network can be stalled if a worm of data is stalled itself, thus spanning several routers and blocking for yet other worms of data. There are however techniques to avoid this, one of which is called virtual channels, which will be described in greater detail later in this report.

Opposite packet switching is *circuit switching*, where a circuit is set up from source to destination and data is then transported on the circuit. Circuits may be dynamically setup and torn down as they are used or they may be statically created in the network prior to execution.

Connection-oriented mechanisms are dedicated or logical connection paths setup from sender to receiver before data is transmitted, which are torn down after use. Hence a circuit switched network is always connection-oriented. *Connection-less* networks do not setup connections, rather the sender transmits the data onto the network which then forwards it to the receiver. Packet switched networks can be either connection-oriented or connection-less, and as in the case of Æthereal both can be available in packet switched networks. Æthereal is examined in more detail in section 3.5. A very interesting example of a connection-oriented packet switched NoC is SoCBUS which will be described further in section 3.6.

When using a *deterministic routing* algorithm the path followed in the network is only based on the source of the packet and its destination. Source routing is a deterministic routing algorithm where the path through the network is decided by the source alone, whereas in XY-routing (mesh and torus networks) which is also a deterministic algorithm, the packet is routed fully along one axis and then fully along the other axis to arrive at its destination. In *adaptive routing* schemes the routing decisions are taken dynamically on a per-hop basis, and the decision can be based on congestion in the surroundings. NOSTRUM uses an extra wire between routers to signal congestion and then routes packets around the congested links. A more thorough investigation of NOSTRUM is conducted in section 3.4.

Minimal routing always follows the shortest path between sender and receiver. If the shortest path is not taken it is said to be *non-minimal*. The HERMES NoC employs a minimal routing algorithm and will be looked further into in section 3.1.

2.1.6 Flow Control

Flow control is the mechanisms that controls the flow of packets through the network, on both a global and a local scale. It ensures that the operation of the network is correct, such as avoiding deadlocks. Flow control can be extended to control the utilisation of the network resources, which can be used to improve the performance and give guarantees on data transfers. Here, flow control issues and the concept of virtual channels is briefly described.

As mentioned above, flow control is about ensuring the correct operation of the network, which means that it must avoid deadlocks. A deadlock is when network resources are waiting for each other to be released. A classical example of a deadlock is shown in figure 2.5. Deadlocks can be avoided by breaking cyclic dependencies in the resource dependency graph. A closely related problem in networks is livelock. Livelock is less common than deadlocks and occurs when resources constantly change state without getting anywhere.

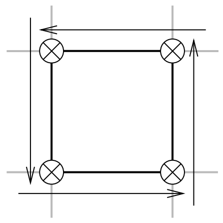


Figure 2.5: Four streams of data wait for each other to release the occupied network resources, resulting in a deadlock.

In every network, macro network as well as NoCs, buffers play an important role. They provide decoupling on the segmented links in the network. In NoC design, buffers account for a very large part of the overall router area, and it is therefore a concern to minimise the size of the buffers. Two things contribute to the buffer size, the data width and the depth of the buffers. It has been shown that deeper buffers do not solve congesting problems but rather delay it. However, deeper buffers do smooth out burst traffic. Buffer placement is another topic within NoC design. Fx. placing buffers at the input results in packets being blocked behind a packet that is unable to leave through its designated output port.

Several independent buffers can be used to share a physical link. This is called *virtual channel*s (VC). A physical link is typically shared between 2 to 16 VCs which involves a matching number of buffers and an arbitration mechanism. Figure 2.6 shows an example of four VCs across a link. VC implementations therefore introduce a power and area overhead, because of the extra buffers in the routers. The advantage of VCs is that they can be used to break dependencies within the network, thus avoiding deadlocks. Wire utilisation can be optimised, performance can be improved due to fewer stalls in the network and VCs can be used to offer differentiated quality of service by reserving a certain priority in the VCs along a path to form vir-

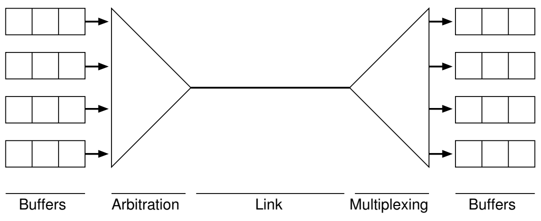


Figure 2.6: A link with four buffers and the arbitration mechanism.

tual circuits. A more in depth discussion of VC and virtual circuits is presented in chapter 4.

2.1.7 Network Abstraction

When designing a SoC, the communication structure can be abstracted at different levels. The most common model used to abstract networks is the OSI model which consist of 7 layers. In NoC some of the OSI layers are redundant due to the inherent differences between macro networks and NoCs. Due to these differences, a collapsed model is introduced in [9] that matches NoC usage. The collapsed model which is shown in figure 2.7 will be used in this project. The figure shows the collapsed model and the corresponding OSI layers, as well as communication directives between the different layers.

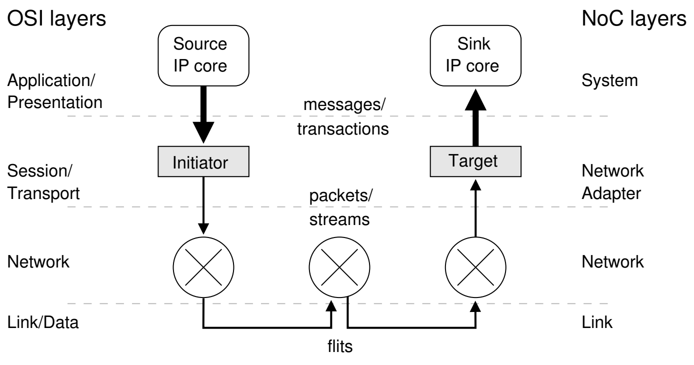


Figure 2.7: OSI layers and the adapted NoC abstraction layers.

The four different layers shown in the above figure also represents different abstractions of the NoC. A high level abstraction model plays an important role in the SoC design process, as it allows greater complexity to be captured. However, the lower levels are also an important part of the design process, as the awareness in the system of the lower levels can lead to better performance. In the following, the four layers depicted in figure 2.7 will be described.

System
The highest layer in the NoC model consists of processes and system architecture. The network details are mostly hidden at this level. At this level, the system only considers a generic communication structure and most SoC research applies at this level. The different processes of the system are communicating by passing messages back and forth between each other. In the process, the communication can be thought of as high level send and receive statements. In a CSP like language the sender would have a send statement, *send(message, receiver)*, and the receiving process would then wait for the message with a receiving statement, *receive(message, sender).*

Network Adapter
The job of the network adapter is as mentioned earlier to decouple the IP cores from the network, as well as handle end-to-end flow control. As an abstraction layer this is the first layer that adds network awareness, in such a way that packets and streams are distinguished. Messages from the system level above are broken into either packets or streams. Packets contain information on their destinations placement in the network, whereas the streams follow pre-configured routes to their destination that has been set up prior to the transmission. A packet would be send in a more specific way than the messages above, code for transmitting a packet would look like *send_packet(packet, destination_address)* and a stream would be transmitted with *stream_data(data, circuit).*

Network
At the network layer, the network is described in terms of routers and links, thus the topology of the network can be described at this level. Routing and transport protocols are considered as well as the flow control in between the routers.

Link
This is the lowest layer in the NoC abstraction model and at this level the atomic units of a transmission, the flits are considered. Data encoding and synchronisation is considered at this level.

2.2 Quality of Service

A network can provide Quality of Service (QoS) to the cores connected to it. Different levels of service can be offered to the cores, thus QoS is defined as a quantification of the service level.

Services such as low latency, high through-put and controlled jitter, are interesting for real time applications and other applications that require guarantees on the transmission services. Services of increasing quality are offered in the different layers of the network abstraction model described in section 2.1.7. Services which can be characterised as high level services are offered in the high abstractions layers and build upon services from the lower layers. Examples of services that build upon each other from the bottom and up are given in the following.

Lossless service guarantees that data transmitted arrives and is received as it was sent, i.e. no error is introduced in the data delivered by the service. Together with *in order* delivery, that ensures that a data packet B, transmitted after a data packet A, will never be delivered before A, this provides the basis for guaranteed service (GS).

GS provides guarantees on transmission specific parameters such as through-put and latency. *Guaranteed through-put* enables streamed data to arrive at a constant rate and *guaranteed latency* ensures that the latency on data transmissions such as read and write operations have a maximum latency.

The guarantees are offered differently in different NoC implementations. The asynchronous MANGO NoC, described in more detail in section 3.7 and in chapter 5, offers hard guarantees whereas QNoC offers statistical guarantees. More on QNoC and the concept of statistical guarantees in section 3.2.

2.3 Use Cases

A use case is a way to describe the communication requirements of a system. In [26] use cases are used as input to the mapping tool used to generate a NoC - more specifically the Æthereal NoC. This section will give an introduction to use cases, their use in NoC design and how multiple use cases can be exploited to optimise NoCs.

A use case describes the traffic pattern i.e. which cores need to communicate, the bandwidth and latency requirements and the traffic types. The use case can be used to plan the traffic. By planning the traffic, power consumption can be lowered as well as optimising overall performance in the network. By distributing the traffic equally over the network, congestion points in the network can be avoided to a large extent. General network layouts can be tailored to meet specific requirements of an application, thus allowing the network and in turn the entire system to handle a wide range of applications. In NOSTRUM, circuits are setup at run time, but the allocation of bandwidth to each circuit can be configured during run time allowing the same fabricated chip to meet bandwidth requirements of several applications.

The application domain of SoCs ranges from digital hearing aids over multi-media phones to set-top boxes, all of which are applications that handle several tasks simultaneously. An example of a use scenario involving a multi-media phone is shown in figure 2.8. If the roaming and GSM specific traffic is discarded the figure shows four different typical phone tasks; voice call, general application usage, music playback and video call. Each of the four tasks presents different requirements to the system. In addition some of the tasks are executing in parallel creating yet more constraints and requirements for the system.

Playing back or streaming audio requires a low jitter and a constant bandwidth connection between a signal processing unit and the audio output. In addition the audio stream can either be stored locally requiring access to off-chip storage, or the audio stream can be acquired across the GSM net requiring access to the RF device. Video calling requires a low and constant latency between IP cores in the system along with some high bandwidth connections to support the video de- and encoding. General application usage such as text messaging, playing games or browsing the Internet generally has lower requirements and can in many cases be handled by best-effort type services. Voice calls requires, as the video call low latency and jitter free connections between RF devices, signal processors and audio input and output devices.

The figure can be segmented into several parts in different ways. Each of the four tasks has requirements that could be described by a use case. However from a system point of view the segmentation shown along the time axis, A to E, into use cases makes more sense. Such a view can be obtained by creating use cases that encapsulate all the tasks that run in parallel. Such a use case completely describes the use of the system in that time interval. As mentioned earlier, the general applications of SoC involves multiple tasks, some of which can execute in parallel. Therefore the use of a system can rarely be described by one use case which leads to having multiple use cases describing the requirements at different times, even more use cases than there are tasks. Use cases can be used to make an application run on a specific SoC, by specifying circuits, routes and services to setup. Use cases can also be used as input to a mapping tool laying out the system to meet the requirements of a specific application.

2.3.1 Multiple Use Cases and Mapping

When considering multiple use cases, the requirements to the communication structure become more complex, as the requirements are now distributed between multiple use cases. In the following, only NoC is considered, but some of the discussion here

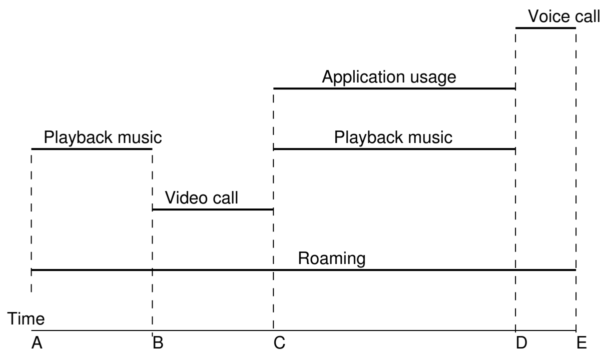


Figure 2.8: An example of how a mobile multi-media phone could be utilised. From A to B the user of the phone listens to some music, from B to C a video call is received, followed by the user interacting with some build in applications while playing more music from C to D. Lastly the user is making to different times. a voice call D to E, during the entire use the phone is handling roaming and other GSM network specific operations.

also applies to other communication structures such as configurable crossbars. One method used in NoC research to handle the problem of how to find the requirement of each connection, when the information is segmented between several use case is to construct a worst case use case. This approach is taken in [26], which uses the worst case use case to map the Æthereal NoC. The NoC mapped from the worst case use case is however most often over engineered, which leads to a bigger design and a higher power consumption. The same group of authors has since described another method to solve this problem that leads to less overhead, by keeping track of requirements during the mapping process. They also introduce the concept of smooth switching, and that approach will be described later in this section.

The Viper [14] is a multiprocessor SoC (MPSoC) developed by Philips Research and is designed for use in set-top boxes. Viper features two general processors (a MIPS CPU and a TriMedia CPU), along with several dedicated processors and I/O interfaces. A set-top box such as the Viper is a prime example of a MPSoC and the need for multiple use cases. Figure 2.9 illustrates a possible use case scenario for a set-top box such as the Viper. The figure shows examples of four use cases, displaying a broadcast, recording a broadcast while viewing another, playing a recorded video while recording a broadcast and only playing a recorded video. When creating use cases, all possible interleavings of system tasks must be considered and placed into several different use cases, creating a large amount of use cases. The important thing is that not all interleavings are possible, fx. a set-top box will not at any time playback a recorded video, while at the same time viewing a broadcast. The exception being picture-in-picture, where several video streams or broadcasts are shown on the screen in smaller partitioned areas, which again could be described by a use case.

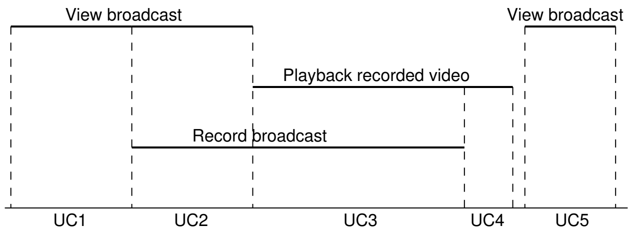


Figure 2.9: A use case example with a set-top box.

The authors of the above mentioned worst case use case has described another approach of mapping multiple use cases to a NoC in [27]. In addition, they introduce the concept of smooth switching. Smooth switching is an important feature as switching between use cases involves reconfiguration of the NoC, which might cause interruptions or delays in ongoing communication. Such a delay or interrupt would not be acceptable in many multi-media applications, and smooth switching addresses just that problem. In addition to specifying use cases, the authors also specify which use cases that should be able to smoothly switch to another use case without interrupting tasks that survives the switch. The communication of these tasks can then be mapped in the network to avoid interruption when other tasks are stopping and starting. This is done by creating use cases that can handle all the tasks of the use cases that can smooth switch in between each other. It should also be noted that not all switches has be smooth, one example being the switch between UC4 and UC5 in figure 2.9.

To avoid creating the very large amount of use cases by hand, [27] presents the approach of creating use cases for all the tasks of the system and then grouping use cases that run in parallel together in compounds. In addition, they specify which use cases should be able to handle smooth switching in between them. All this information is then fed into a mapping tool that creates all the different use cases and maps the requirements to a NoC. They avoid the overhead of their previous approach where they used worst case to construct the network, by considering the overall constraints when mapping each individual use case.

2.4 Asynchronous Circuits

Asynchronous circuits are clockless circuits, which as opposed to synchronous cicuits do not have a global synchronisation. Instead, asynchronous circuits make use of local handshakes. This chapter gives an introduction to asynchronous circuits and especially the concepts used in this report. A comprehensive introduction to asynchronous circuits can be found in [32].

Asynchronous circuits are interesting to explore, due to some of the inherent properties, such as *lower power consumption*; *high operating speed*; *less emission of electro-magnetic noise*; *robustness towards enviroment variations*; *better composability*; and *no clock distribution problems*. Of course, these advantages come at a cost and just like synchronous implementations there are trade-offs to be made. One of the disadvantages with asynchronous circuits is the lack of design and test tools, which makes implementation and verification a tedious job. In NoC, asynchronous circuits are especially interesting due to their composability and the fact that no clock has to be distributed over the entire chip. Where in synchronous circuits the entire system is limited by the slowest path in the system, asynchronous circuits do not have this global dependency and therefore does not suffer from this dependency. The behaviour of asynchronous circuits depends on the data, which makes them hard to predict and depending on the implementation style this can affect the execution time and power consumption.

Asynchronous circuits inherently support the Globally Asynchronous Locally Synchronous (GALS) scheme, which makes them very suitable for NoC implementations. The main themes when designing asynchronous circuits are handshake protocols, data encoding, handshake blocks and combinatorial logic blocks. This section is divided into a section on handshake protocols, followed by a description of basic building blocks and how timing is matched through the circuit.

2.4.1 Handshake Protocols

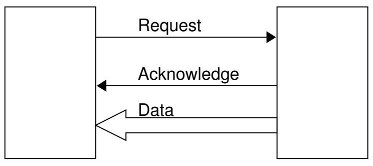
A handshake protocol is a series of events that occur in a specific order. A general handshake occurs as follows: A request is initiated, as data is valid; an acknowledge is raised by the target, as the data is accepted; the data no longer has to be valid. In return-to-zero protocols the request is lowered followed by a lowered acknowledge. This examples is a push channel handshake as data is pushed forward along the request, as opposed to a pull channel where data is requested by the reciever and delivered as part the acknowledge.

The request and acknowledge can be signaled on seperate wires to the data which is called bundled data (figure 2.10(a)), where the dual-rail embeds the request in the data wires (figure 2.10(b)). Dual-rail uses 2 wires per bit for a total of $2 \times bit\_width + 1$, where in the bundled data the overhead only consists of 2 wires (request and acknowledge). As mentioned, protocols also differ in whether they return to the same state after a handshake (4 phase, figure 2.10(c)) or keep the state, thus using the shift on the wire as part of the signalling (2 phase, figure 2.10(d)).

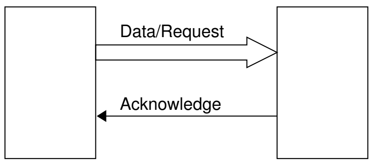
These 3 different aspects of a handshake protocol can be combined to create the handshake protocol of choice, MANGO employs 2 phase push dual-rail and 4 phase push bundled data.

2.4.2 Data Flow

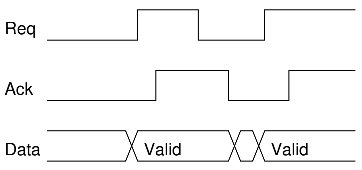
Two types of building blocks create the fundament for asynchronous circuit design: handshake blocks and functional blocks. Handshake blocks handles the flow control such as, splitting, selecting or joining data channels or control. The function block manipulate the data, without taking part in the handshakes.



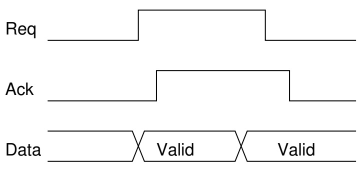
(a) Bundled data (pull)



(b) Dual-rail (push)



(c) 4 phase



(d) 2 phase

Figure 2.10: Asynchronous circuits can be implemented with different style of handshake protocols.

C-element

The basic element in asynchronous design is the Muller C-element, which is shown in figure 2.11. The C-element only changes output value when all inputs are the same, it is therefore suitable for the indication style circuit that is employed in asynchronous design.

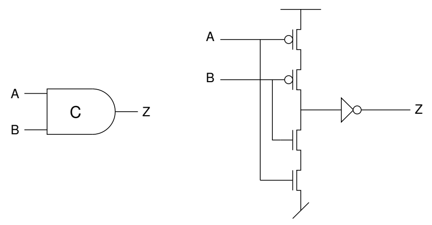


Figure 2.11: The basic element of asynchronous design is the Muller C-element, the output only changes when both inputs are the same.

Latch

The basic handshake latch is controlled by a C-element and in figure 2.12 a 4 phase data latch is shown. The latch is operated by the EN signal that is generated by the output acknowledge and the input request.

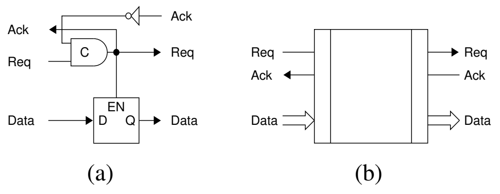


Figure 2.12: A hanshake latch is controlled by C-element (figure 2.12(a)). The symbol used for handshake latches (figure 2.12(b)).

Functional Blocks

Several latches can be placed after each other to form a pipeline of latches. In between the latches, functional blocks can be placed. Because the functional blocks do not affect the handshake, the request and acknowledge is exchanged between the latches surrounding the functional block, as is depicted in figure 2.13. The figure

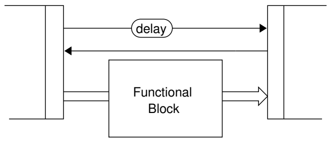


Figure 2.13: The functional block does not affect the actual handshake, but the arrival of the request in the next latch must be matched with the data.

also indicates a delay on the forward flowing request, which is needed to match the arrival of the request with the arrival of data. The delay of data in the functional block depends on the actual data and the request delay will therefore either be a function of the data or a fixed worst case delay.

CHAPTER 3

A Network-on-Chip Study

In recent years, the academic interest in NoC [12] has increased and different research groups have made publications on NoC concepts and implementations. Among academic implementations of NoC are the CHAIN network [1], SPIN [17] and ×PIPES [4].

Industrial implementation of the NoC concept, has also picked up over the last few years. Examples of these are the Element Interconnection Bus in the Cell processor [20], most known for its use in the upcoming Play Station 3. Philips are also researching in the NoC area with their Æthereal network [16], for use in the company's multi-media products such as televisions [33] and set-top boxes [14].

This chapter is a study of NoC implementations with focus on how Quality-of-Service features are implemented. The seven different NoCs in this study are chosen because they represent different approaches to and different levels of QoS. The first NoC described only provides BE routing and gives no guarantees on latency and throughput. The study then looks at NoCs that provides increasing levels of guarantees on latency and throughput.

Each of the NoCs is described in relation to what basic concepts the implement, such as the interface they implement and the unique features that makes them stand out. Their routing scheme are described, along with the components that they are constructed with and what topologies they support. The different communication structures are investigated with focus on what kind of service the NoCs offer. Especially how setup and configuration of service, such as guaranteed latency and throughput are handled.

Most of the studied NoCs are developed in academic environments, but Æthereal and Octagon are developed by Philips and STMicroelectronics respectively.

3.1 Hermes

The HERMES NoC [25] is developed at Faculdade de Informática PUCRS, Brazil. HERMES is designed to implement very small switches and be of immediate practical use. The routing scheme employed is XY routing between the switches arranged in a 2d-mesh structure.

A HERMES switch consists of control logic, five bidirectional ports and buffers. Four ports connect to neighbouring switches and the fifth port connects to the local IP core. The buffers are placed at the input ports. HERMES implements wormhole routing, for low memory and latency. The flit size is 8 bit and the two first flits in a packet contains the header, which consists of the target address and the number of flits in the packet.

The routing is a typical XY routing scheme where the packet is routed fully in the X direction and when aligned properly on the X axis, it is routed in the Y direction until it reaches its target, thus employing a minimal routing alghorithm. When a packet has been granted access to an output port through its header flit the connection between input and output port is reserved for that packet. The number of flits is counted and compared to the flit number in the header until the entire packet has propagated through the switch, at which point the connection is closed. The connected ports inside the switch are stored in a switching table. Five simultaneously connections can be operated at any one time. The arbitration scheme prioritises the ports, based on which port last had a successful connection established to ensure that no port gets starved.

HERMES employs input queues as mentioned earlier. The purpose of these buffers is to reduce the number of switches affected by a blocked packet. The size of the buffer is parametrisable, with a default size of eight flits.

HERMES is loss less and inorder, but does not provide any form of latency or bandwidth guarantees. Because HERMES is connection-less no such guarantees can be given.

3.2 QNoC

At the Electrical Engineering Department, Israel Institute of Technology, Quality-of-Service NoC (QNoC) [10] has been developed and described. QNoC is a network of multi-port switches connected to each other by links. A link is composed of parallel point-to-point lines. The topology of QNoC is a mesh and it employs wormhole packet forwarding with hop-by-hop credit-based back-pressure flow-control. Routing paths are static, shortest path and X-Y coordinate based. Traffic is divided into four different classes of service: signaling, real-time, read/write and block-transfer, with signaling having the highest priority and block-transfers the lowest priority. A distinct feature of QNoC is that, unlike other wormhole based routing systems, packet forwarding is interleaved according to the QoS rules. This means that a high priority packet, such as a real-time packet, will preempt a low priority packet such as a read-

/write packet. QNoC does not provide hard guarantees on the service, but instead QNoC provides a statistical guarantee.

QNoC is loss less and it assumes that retransmission is never required. Packets always travel the shortest route, which minimises power dissipation and maximises the utilisation of the network. QNoC is unable to give hard guarantees because it has been chosen not to implement circuit switching due to the high cost of managing and establishing circuits. QNoC applies an irregular mesh topology to accommodate the irregular structure of SoC floor plans.

An IP core or as they are called in QNoC, a system module is connected to a router via a standard interface that is adapted to the communication needs of that module. The inter-router links are similarly adjusted to meet the expected bandwidth needs of that link. The expected communication needs and bandwidth requirements are developed from an extensive simulator that has been built alongside QNoC. Routing is done over fixed shortest paths, using X-Y directions in a coordinate fashion, thus packets are routed first all along X and then perpendicular or first along Y and then along the perpendicular X axis.

In QNoC, four different types of traffic are considered. They are as mentioned earlier, signaling, real-time, read/write and block-transfer. *Signaling* has the highest priority and is used for urgent and very short messages, such as interrupts and control signals. The introduction of a signaling service removes the need for dedicated single-use wires. The *Real-Time* service is used for streaming real-time applications, such as streamed audio and video. As described earlier, QNoC does not use circuit switching, thus the real-time service level is packet based like the remaining service levels. A real-time link may be allocated a maximum bandwidth that should not be exceeded. *Read/Write* is designed to provide bus-like semantics and supports short memory and register access. *Block-Transfer* service level is used for transfers of long messages and large blocks of data. These four service levels are prioritised, with signaling given the highest priority and block-transfers lowest. More service levels could easily be defined and included in this priority scheme. It would however, as described later, require more buffers.

A packet in QNoC consists of three parts: target routing address, command and payload. The routing address is required for routing, the command field identifies payload format. The payload can be of arbitrary length and contains, along with the payload, operation specific control information such as sender identification. A packet is transmitted as multiple flits, and the flit transfer over the link is controlled by handshaking. Three types of flits exist: full packet; end of packet; body. The full packet is a packet that is only one flit long. The body flit is a flit that is not the last flit of a packet. Flit type and service level is indicated on separate out-of-band wires.

A router in QNoC is connected to up to five links: four neighbouring routers and one IP core. The router forwards packets between input and output ports, flit by flit. Upon entering the router the flit gets stored in an input buffer. When the first flit of a packet arrives at the router, the output port for that packet is determined. Every flit arriving on that port and service level is then forwarded to the same output port. Because QNoC uses direct buffer mapping, every service level has its own buffer at each input. The buffer is small and only capable of storing a few flits, but the buffer size can be altered at design time. The buffers at each service level and input port are managed by credits. When a flit passes from input to output, a credit is passed back to the previous router on a separate wire. The actual transfer of data can be done by different types of handshake interface, an asynchronous interface could be used. Inside the router, each output port keeps tracks of available buffer space in the connected router in a table for each service level. The output link is allocated according to the available slots in the next routers buffers, the service level and a round-robin scheme between the input ports.

3.3 Octagon

Octagon is an on-chip communication architecture designed to meet the performance requirements of network processor SoCs, such as Internet router implementations. The communication architecture is described in [21]. The authors present the following desirable properties of Octagon: two-hop communication between any pair of nodes (depending on the number of nodes and configuration); higher aggregate throughput than a shared bus or crossbar under certain implementation conditions; simple, shortest-path routing algorithm; and less wiring than a crossbar interconnect. Octagon can operate in two modes; packet- and circuit-switched. A node in Octagon is associated with a processing unit and a memory module, this means that only non-local memory request generates Octagon communication requests. According to the authors of Octagon this is a very desirable feature when working with network functions such as routing table look-up and Internet Protocol classification.

An example of a basic Octagon configuration that consists of eight nodes and twelve bidirectional links is shown in figure 3.1. As can be seen from the figure, every node can reach any other node with a maximum of two hops. When scaling Octagon up this maximum length changes, but this will be discussed later. The complexity of Octagons implementation increases linearly with the number of nodes that must be connected through the network.

In circuit switched mode, a network arbiter allocates the path between the two nodes that are to communicate. The path is reserved for a number of clock cycles for that communication. The arbiter permits several pairs of nodes to communicate with to each other at the same time, as long as the paths do not overlap, i.e. the arbiter permits spatial reuse of resources in circuit switched mode. To optimize the throughput when working in circuit switched mode, the authors of Octagon have developed the best-fit algorithm to schedule connections in the Octagon network. When working in packet switched mode the, nodes, routes the packets according to a destination field in each packet. The node calculates a relative address based on the destination address and the node's own address, and the packet is then either routed left, right, across or in to the node.

One of Octagon's strong features is its ability to scale. In [21] two scaling strategies are presented, low wiring complexity or high performance. As the naming of

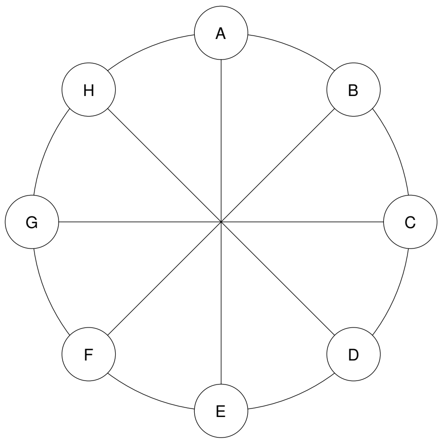


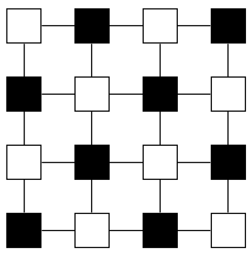
Figure 3.1: A basic Octagon configuration, with 8 nodes and 12 bidirectional links.

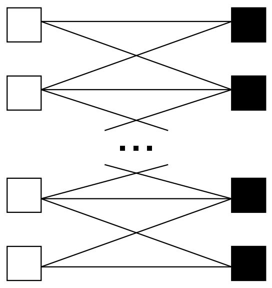
the strategies suggests Octagon can be optimised for both low area and high performance. When scaling for low wiring the strategy is to connect several basic Octagon configurations as the one shown in figure 3.1. Nodes that are members of two basic Octagon loops function as bridges, and performs hierarchical routing. This allows the standard nodes to be used without modifications, because only the bridge nodes need to know about the other circles. An Octagon network with 22 nodes scaled for low wiring complexity has a maximum distance of 6 hops while maintaining very few wires when compared to a fully connected crossbar. The other scaling strategy proposed has higher performance than the above mentioned low wiring scaling strategy at the cost of more complex wiring. The high performance scaling strategy allows 512 nodes to be connected with a maximum distance of 6 hops and 2,304 links compared to a crossbar that would require $512 + \cdots + 1$ links. The reader is referred to the above mentioned article for more on the interesting scaling strategies.

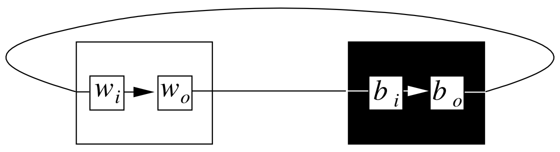
3.4 NOSTRUM

At the Laboratory of Electronic & Computer Systems, Royal Institute of Technology, Stockholm, the Nostrum Network on Chip has been described in several articles, fx. the protocol stack is described in [23] and the guaranteed bandwidth approach is described in [24]. Nostrum takes the same approach as macro networks, ie. computer networks, and emphasises the layered communication by using layered protocols. Nostrum implements among other things switch load distribution, guaranteed bandwidth and multicasting. Furthermore, a simulation environment is provided.

IP cores or resources as they are called in Nostrum, are organised in a two-dimensional mesh, where each switch is connected to its switch neighbours and to its resource. Other topologies are possible in Nostrum, but the outset of Nostrum is a 2d-mesh. The routing scheme chosen is hot-potato or deflective routing [15], which eliminates the need for buffers in the switch compared to traditional store-and-forward routing schemes. Furthermore [28] describes how load distribution is implemented in Nostrum. Each switch indicates its current load by sending a stress value to its neighbours. This means that each switch has a picture of the load in its surroundings. Incoming packets are sorted by the number of switch cycles the packet has been travelling (in its current implementation). The packet with highest priority chooses its output port first and then the remaining packets choose in descending priority. Because of the hot-potato scheme packets which find their desired output port to be taken by a higher priority packet must choose another output port. Because the routing decisions are made locally and on individual flits there is no guarantee that flits let alone packets will arrive in order. Only best-effort traffic is handled by the above described routing scheme. The guaranteed service traffic is described in the following.


(a) Topology


(b) Bipartite graph



(c) Buffers

Figure 3.2: Temporally Disjoint Networks due to topology (3.2(a) and 3.2(b)) and buffers (3.2(c)).

As mentioned earlier, Nostrum provides Guaranteed Bandwidth as described in [24]. The guaranteed bandwidth is accomplished by introducing looped containers. A container, once created, is looping around on a predetermined path, a virtual circuit, and can either carry data or be marked as empty. As mentioned earlier Nostrum employs hot-potato routing, but to avoid packets being deflected into the local port when it was not meant for that resource, the insertion of new packets is only granted when there is a free output port. To guarantee bandwidth, insertion of the data must be ensured and because the container is already in the network the insertion of data is guaranteed. The header of a container contains two bits, the first bit marks the packet as a container the second bit marks it as empty or full, in addition to the normal header of a Nostrum packet. A container is a special type of packet hence the extra marking. When a container arrives at an input port at the source of the virtual circuit it will, because of the deflective routing scheme and lack of buffers in the switch, be forced to leave on the next clock cycle. The virtual circuit will however not be deflected away on another path, but will always follow the path of the virtual circuit. The separation of the virtual circuits in each switch is done by Temporally Disjoint Networks and will be explained later in this section. The source of the virtual circuit will thus have one clock cycle to place data in the container, while the container resides at the input port of the switch. Once the container has left the input port of the switch it will follow the path with precedence over the normal BE traffic to the destination. Upon arriving at the destination the container will be unloaded and possibly be filled with new data. It is obvious to introduce multicasting by simply having the container loop around on a virtual circuit consisting of all the recipients of such a multicast. This is described in [22], and only involves changes to the switch. The granularity of the bandwidth allocation is determined by the maximum bandwidth available divided by the round-trip delay on a virtual circuit and the amount of bandwidth acquired by a virtual circuit depends on the amount of containers inserted on the virtual circuit. There is an upper bound on the amount of containers that can be inserted on the virtual circuit, which comes from the amount of buffer space along the virtual circuit. Virtual circuits are semi-static, in the meaning that the virtual circuits and their paths can only be created at design time. The amount of containers can however be adjusted at run time, by inserting and extracting containers.

As mentioned earlier, Nostrum uses the concept of Temporally Disjoint Networks (TDN), to explain the time-division multiplexing in the network. TDNs arise from the topology and the amount of buffers in the network. By colouring the nodes in the network in a chess-board like fashion, figure 3.2(a), it can be realised that two flits residing in different coloured nodes can never contest for the same resource, under the assumption that a flit does not get stored in a node, as it is the case with hot-potato routing employed by Nostrum. These two flits can then be tought of as being in different networks. Further a 2d-mesh network can be displayed as a bipartite graph, figure 3.2(b), where the two networks are placed opposite to each other. From this, it can be realised that a flit will always change network on each move, which leads to figure 3.2(c) where the biparte graph is collapsed into one node for each network. This is refered to as the topology factor. Introducing input and output buffers in each node, w_i, w_o, b_i, b_o, gives more distinct networks. This can be realised, by looking again at figure 3.2(c), it can be seen that a flit has to travel through at least four buffers, and the four flits in each of the four buffers will never contest for resources. These distinct networks is what Nostrum calls TDN. A virtual circuit can then be assigned to one or more specific TDNs and several virtual circuits will then be able to use the same switch. This however puts an upper bound of the number of virtual circuits that can pass through a single switch. The upper bound scales with the amount of buffer stages in each nodes as $TDN = Topology\ factor \times Buffer\ stages$.

3.5 Æthereal

The Æthereal NoC is developed at Philips Research Laboratories, Eindhoven, The Netherlands and is one of the few industry implementations. Æthereal has been described in several articles such as [16], [31] and [13]. In addition, mapping approaches and tools have further been described in [26] and [27] amongst other. Æthereal is a synchronous NoC, which offers among other things both GS and BE traffic, interfaces for standard sockets, monitoring and reconfiguration. In [3] and [33] an Æthereal implementation of a digital video reciever and a high-end consumer TV-system is compared to traditional interconnect solutions.

The Æthereal NoC offers a range of standard sockets (including OCP [30] and AXI - The Advandced eXtensible Interface) through network interfaces. The term network interface is used of the NA in Æthereal. The network interfaces handles the conversion from messages to packets, thus being a NA as described in section 2.1. The network interface is composed of a kernel and a shell, which will be described later in this section. Connected to the network interface is a router, which are used to provide contention-free source routing based on time-division multiplexing (TDM).

Æthereal uses a logical notion of synchronicity to create time slots for flits. A circuit is created by reserving consecutive slots in consecutive routers. Source routing is used because it allows for topology independence, and the path is made from a list of output ports on the route. As mentioned, two types of traffic service exist, namely BE and GS. GS traffic gives guarantees on both throughput and latency. Due to the nature of TDM the latency and throughput guarantees depend on each other. BE traffic is moved through the network using spare slots not used by GS connections and contesting BE packets are arbitrated using a round robin scheme. Interleaving of BE packets are not possible. GS traffic can however interrupt in the middle of BE traffic. Interleaving of GS connections are avoided by scheduling. A credit based flow control scheme is used between the BE buffers to avoid loss of flits by overflowing the buffers. The neighbouring routers keep track off each others buffer status by exchanging credits as flits are forwarded.

GS traffic is scheduled by the network interfaces, and the flits are then forwarded from their allocated slot to the next in the routers. Depending on the operating scheme the slot reservations are either done in network interfaces and routers or solely in the network interfaces. Æthereal operates with two ways of reserving slots: Distributed and central. The distributed approach uses slot tables in the routers and network interfaces, where the central approach only sets up the network interfaces. A path is then applied like in the BE case. A slot table matches input and output ports for each time slot. In both cases the network interface handles the insertion of data by use of a similar slot table. Æthereal thus requires an extensive scheduling of the traffic patterns in order to avoid congestion on links.

The authors of Æthereal argue that the distributed scheme is scalable, due to the route being distributed in the routers where in the central scheme the length of the routing path affects the flit size. The central scheme however has significantly smaller routers due to not using slot tables. The central scheme therefore works very well in smaller NoCs, which in the near future seems to be most likely.

The Æthereal network interfaces are split into two major parts; a network kernel that recieves and provides message to the network; and a network shell which converts standard sockets into the message type handled by the kernel. The actual message structure is irrelevant to the kernel, as it just handles the message as payload in packets that must be transported across the NoC. The kernel and shell communicate through ports and each port allows for a peer-to-peer connection with a shell and thus an IP core connected to another kernel. A network kernel can have several shells attached, both master and slave shells thus connecting several IP cores to one network kernel. Multiple connections are possible over each port, by using a connection ID to signal different connections. Each port has a number of buffer pairs, one pair for each connection that can be handled simultaneously, the pair is split in a reciever and sender buffer part. The use of buffers allows IP cores and the NoC to operate at different clock frequencies, and end to end flow control ensures that packets are only transmitted if the recieving end has buffer space to accept the packet. To support protocol features such as narrowcast and multicast special modules can be inserted between the shell and the kernel to enable these features. These modules are optional and can be inserted at design time if required by the application.

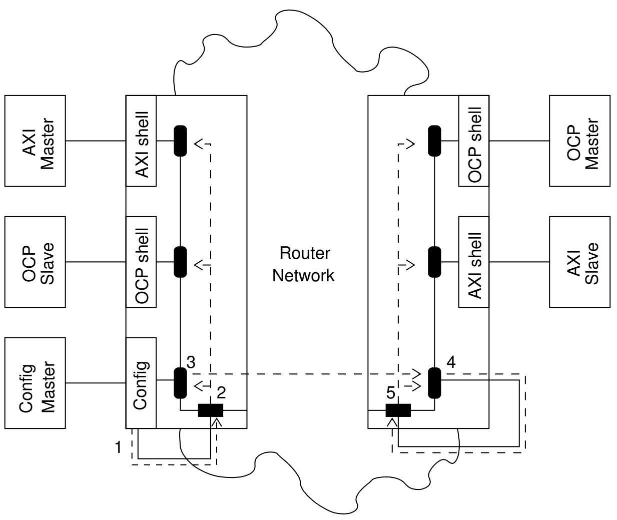


Figure 3.3: The configuration master first sets up its own configuration port (step 1), secondly it sends a configuration request to the remote network interface using the network (step 2), which sets up a return path to the configuration port. In step 3 the remote network interface is configured. Step 4 sets up the channels from the right to the left network interface, step 5 sets up the paths the other way.

One major asset of Æthereal is the support for configuration during run time. A special configuration shell is used to connect a configuration master to a network kernel. This configuration shell is connected to a standard port on the network kernel as well as to a special configuration port. The standard port is used to forward configuration requests to other parts of the network and the configuration port is used to configure the other standard ports of the network kernel. The remote network kernel connects a port to its own configuration port, allowing the configuration master to access its ports. This means that the configuration is done using the network i.e. no seperate configuration channels or network is needed. Figure 3.3 shows two network kernels, one with a configuration shell; and one with a configuration loop. It also shows the use of different shells for different sockets.

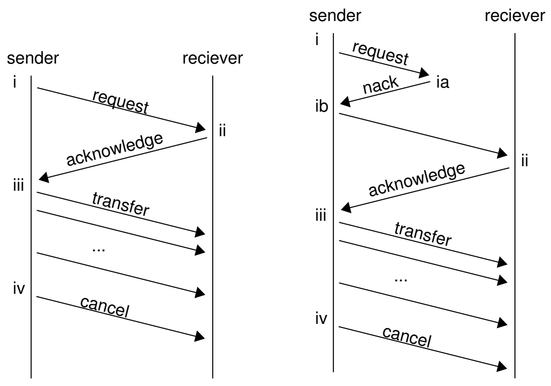
A major part of the (re)configuration of Æthereal comes from the associated use case tools that were described in section 2.3. Because the use cases define the usage of the network ressources, mapping of connections in the network is used to allow use case switching. Scheduling is a very important aspect of Æthereal because it is used to avoid congestion on GS traffic. The scheduling allows for very small buffers in the GS part of the routers because a GS flit is never stalled in the network. This requires a very extensive scheduling to ensure.

3.6 SoCBUS

In [34] the Linköping SocBUS is proposed and described as a bus replacement network. It is developed at the Department of Electrical Engineering at Linköping University as part of Daniel Wiklunds PhD thesis. SoCBUS implements circuit switching only in a form they call packet connected circuit (PCC), which will be described later in this section. By only making circuit switching available, avoiding deadlock and guaranteeing in-order delivery becomes much simpler than in packet switched networks.

IP cores are arranged in a 2 dimensional mesh, because the authors of SoCBUS found the 2D mesh topology the most suitable for NoCs, due to an acceptable low wire cost and reasonable high bandwidth. In addition, cores with a high amount of communication between each other can be placed closer together to avoid large amounts of long distance traffic. The switches are 5 port switches with four of the ports connecting to neighbouring switches and one port connected to an IP core through a wrapper. These wrappers handle the differences between the IP cores interface and the interface used by SoCBUS, such as transaction style, port width, endianness and clock frequency. As mentioned, SoCBUS only employs circuit switching, but during setup of the circuits the switch has to route a request from the source of the circuit to the end destination. The routing scheme used for this is a minimum path length algorithm, where the routing decision is based on the destination address of the request. This is possible and feasible due to the fact that a NoC is static, as described in section 2.1.7.

Because SoCBUS only uses circuit switched connections between cores the concept of packet connected circuit is introduced. Circuits in the network are set up by performing the following transaction, see figure 3.4: (i) The source sends a request that is routed towards its destination, as described above. When it passes a switch the route through that switch is locked. (ia) If a switch is unable to route the request closer to its destination it returns a negative acknowledge to the source, which releases all locks as it travels back along the route. (ib) The source retries by making a new request. (ii) When the request reaches the destination of the circuit, the destination returns an acknowledgement which permanently locks the route. (iii) When the source node receives the acknowledgement it begins the actual data transfer, (iv) which is ended by a cancel request that tears down the route by releasing all the locks along the route. The switches can now allocate the route to another circuit when it receives a new request. SoCBUS promotes the use of PCC because it is deadlock free, uses simple routing hardware, buffering is only needed in the request phase, connections have low latency as each switch only needs a single latch and there is no constraints on the routing algorithms that can be used.



(a) No retry (b) Retry needed

Figure 3.4: There are six different transaction types, two are only used if the source needs to retry the setup of the circuit. (a) shows a succesfull try, and in (b) a retry is needed to complete the transmission.

As mentioned, a switch has five ports which all use the same physical interface. The internal interface between switches in SoCBUS has eight wires for forward going data as well as the request packets used to set up circuits. The transmissions on the data link are controlled by one forward going wire, carrying framing and timing information and two backward going, that handle the acknowledgements. The port that is connected to the local core interfaces thorugh a wrapper to the core. The wrapper handles everything that differs between the IP cores interfaces and the internal interface used in SoCBUS.

To allow for wire delay and skew on the links between switches, SoCBUS uses mesochronous clocking with signal retiming. A mesochronous system has the same frequency but allows for an unknown phase. In addition, SoCBUS proposes to use optimized drivers and transmission-style wires on the links in the network.

Even though SoCBUS offers guaranteed service connections, it is hard hard to give guarantees with the approach taken by SoCBUS. SoCBUS has to establish a connection every time data is to be transmitted and the setup of connections can take a very long time. It is impossible to know how long the setup is going to take, if the connection is forced into trying to establish over and over again. When a connection is established a high throughput and a low latency is offered by SoCBUS.

3.7 MANGO

MANGO is the an abbreviation of *Message-passing Asynchronous Network providing Guaranteed services over OCP-interfaces* and is described and implemented the PhD thesis [5] by Tobias Bjerregaard at IMM, DTU. As the name indicates MANGO is an asynchronous NoC, offering guaranteed service connections between IP cores with OCP sockets.

MANGOs routers have four ports that can be connected to other routers and one port which is used to connect to IP cores through an NA. MANGO offers in addition to the GS indicitated by the name, also a full fledged BE network. The BE network is source routed and the forward and return path can be different. It allows for great flexibilty when routing packets, but limits the length of a BE path to 5 hops because of the width of the flits. The interfaces matched the synchronous OCP standard, the NAs handle the conversion to the asynchronous domain.

Connection-oriented end-to-end circuits can be setup which allows for GS communication across the network. Connections are established as virtual circuits that are formed from VCs between the routers. The guarantees offered is hard guarantees on bandwidth and latency, due to the novel approach taken with the scheduling on the shared links, bandwidth and latency guarantees are decoupled. This is in contrast to most guaranteed service implementations, which are synchronous an which relies on time-division multiplexing. The scheduling of the shared links are done using and arbitration alghorithm called ALG. ALG or Asynchronous Latency Guarantee is presented as part of the MANGO PhD. Figure 3.5 shows an ALG link, which consists of VC control part that surrounds an admission control, a static scheduler and the shared link, with a merge and split.

The static scheduler schedules the channels in priority, thus a higher priority channel will always be granted access over lower priorities. The admission controller

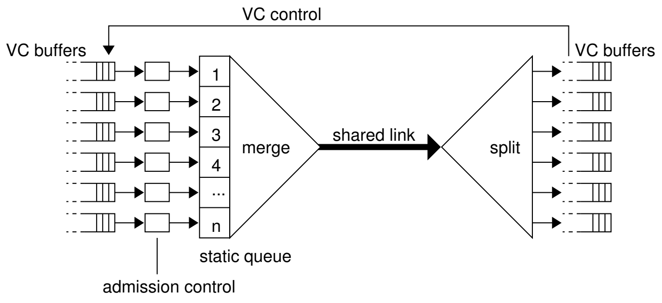


Figure 3.5: A complete ALG link.

placed in front the scheduler ensure that no VC is starved. The admission controller monitors the flits waiting at each of the schedulers channels and only admits higher priority channels when none of the lower priority channels has been stalled due to this particular channel. The admission controller takes a snapshot of the schedulers queues every time a flit leaves the scheduler. If a new flit arives on that particular VC it is stalled until all the flits that where in the schedulers queue at the time of the snapshot has been granted access to the link. This ensures that no VC waits for more than one flit at each higher priority channel. Each VC has a VC control wire associated with it that handles flow control on the VCs. As can be seen from figure 3.6 each VC handshakes to ensure the flow, while the link utilisation is maximised using a pipelined link. The long handshake for each VC is slow because of a long wire, but several VCs can be performing such a handshakes because the link is pipelined.

The ALG has eight channels, where the lowest priority channel is used by the BE router. The BE router uses credit based buffers to handle flow control. Credits are exchanged as flits move between the routers ensuring that no buffer overflows. BE packets are split into flits, where the first flits holds the routing path, which is rotated as the flit moves from hop to hop. The following flits follow the header flit, which reserves the path trough the router, until the last flit of the packet releases the path by indicating end of packet.

As mentioned earlier the conversion between the synchronous domain of the IP cores and the asynchronous network is handled by the NA. MANGO uses two different network adapters, namely a target and an initator NA. These differ in the CI which they offer, the target has a master interface which connects to a slave IP core and the initiator is connected to a master through the NAs slave interface. The synchronisation between the clocked and the clock-less interfaces is handled by a synchroniser. The OCP transactions on the CI is transformed into messages which is passed in the

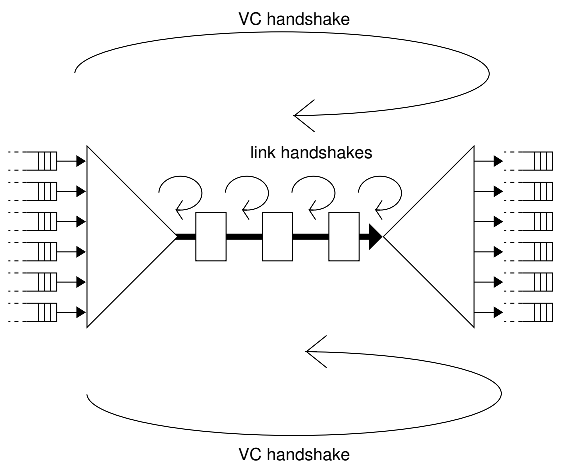


Figure 3.6: Overlapping handshakes controls the flow while maximising performance on the shared link.

network. The NA converts the message into flits by serialising the message.

A table in each router dictates how VCs are connected to form a virtual circuit. These tables are configured by writing to memory mapped addresses in the network. All configuration is carried out using OCP writes, which is translated into BE traffic if the configuration has to travel across the network to access the ressource. At the local interface of the router three VCs can be accessed by the IP cores, in addition to the BE network. The connection ID signal of the OCP interface is used to access the circuits. The first flit in a transaction, both BE and GS indicates the return connection port and type, which is used by the target NA to return the response if needed.

CHAPTER **4**

Guaranteed Service with Virtual Circuits

From the study of todays NoC, especially their approaches to QoS, it is obvious that guaranteeing service can be done in a number of different ways. As described earlier, this project investigates how MANGO can be extended to allow guaranteed service connectons to be reconfigured at runtime. Therefore, this chapter discusses virtual circuits, which is the way MANGO implements guaranteed service.

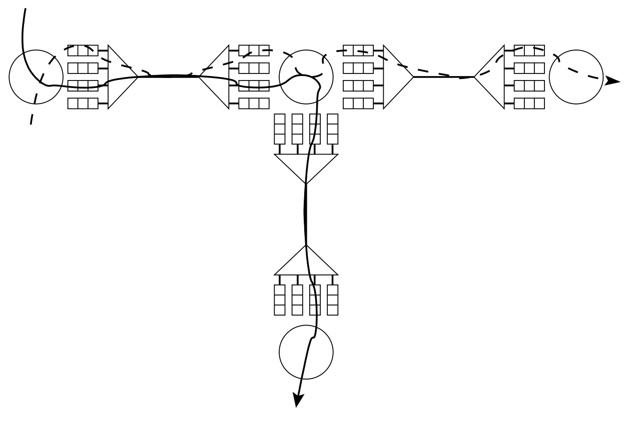


Figure 4.1: Two virtual circuits being used by two different streams. On the joint link the dashed stream is using a higher priority VC than the solid stream.

As mentioned in section 2.1.6 virtual circuits are made up of a number of VCs. In figure 4.1 two streams are shown following two seperate virtual circuits. The figure has four VCs on each link and only one link is shown, in a bidirectional implementation a seperate set of links and buffers would be used for the reverse direction. Another thing to note on the figure is the use of both output and input buffers. Using only output or input buffers is a more common approach to minimise the buffer overhead introduced.

An alternative to using VCs is time division multiplexing, which Æthereal employs and the technique was presented in section 3.5.

A great deal of the ideas and concepts discussed in this chapter can be applied more generally to NoC connections and not just virtual circuits.

The following sections will describe different concepts regarding virtual circuits and their use in managing QoS. In section 4.1 virtual circuits and abstraction layers are discussed, following is section 4.2 that describes programming models that can be used in conjunction with virtual circuits. Section 4.3 describes how use cases, as described in section 2.3, can be used to store and setup virtual ciruits, followed by a discussion of ways to distribute these use cases to the relevant parts of the NoC. Lastly, section 4.4 discusses how the system can be given knowledge about the state of the virtual circuits in the network.

4.1 Abstraction of Virtual Circuits

This section will describe how virtual circuits can be abstracted, using the model introduced in section 2.1.7. However as described there, the system level offers little to no insight in the connections created in the underlying communication structure and is therefore omitted from the discussion in this section. This section is therefore divided into three subsections that describe and discusses the impact on and of virtual circuits at each abstraction level.

4.1.1 Network Adapter

Because the NA handles the transfer of information from the system level above to the network level below, it recieves messages and creates packets and streams. Streams are inherently connection oriented and all connections and thus also virtual circuits originate from a NA and ends at another NA. The NA can be said to insert information into the endpoint of a circuit, which then transports the information to the circuit's other endpoint where another NA unwraps the data and passes it as a message to the system level above.

4.1.2 Network

At the network level the virtual circuit is partitioned into severel chained VCs. The streams are split into flits that the routers transmit from one VC to the next. In circuit switched implementation the routers need to be informed or configured in advance of the stream arriving in order to connect incoming VCs to their respective outgoing VCs. Thus when considering the network level, the route of the virtual circuit must be considered, such as the topology and mapping of cores to network ports.

4.1.3 Link

At the link level, flow control of the flits is considered, i.e. movement of the flits from buffer to buffer. The other logical channels sharing the physical link can be negleted to be an extra delay on the flow control signals.

4.2 Programming Model

Three ways of controlling the setup will be described and discussed here. (i) Distributed programming: a network master interface request the setup of a virtual channel. (ii) Central programming: the network controller sends setup packets to each router that needs a new setup. (iii) Central request, distributed setup or hybrid: the network controller sends request for virtual channels to the master network interfaces, who then sends setup packets along the virtual circuit that is to be setup. In the following, the three models will be discussed against each other followed by individual discussions.

The central and the hybrid programming models confine the knowledge of the system state to one part of the application by ensuring that one unit in the system has all the knowledge of the current state of the virtual circuits at the network level. This information is not available as such in the distributed programming model. It is however implicitly available globally from the state of the application and thus available at the system level. The central knowledge of the state offers more control over the setup process, because only one master sets up the network. Having the information distributed introduces synchronisation issues, fx. there is no inherent way of synchronising the setup between multiple masters.

Between the hybrid model and the central model the main difference is in the distribution of the setup information to the routers and NAs. The central model is producing a large amount of packets from the same point in the network which spreads to most of the network, where the hybrid model produces less traffic around the controller. Depending on different aspects, such as the topology each model may have an advantage.

The two models based on a central controller both needs to be able to program every aspect of the network through network ports. This means that every router aswell as NA must be able to accept setup requests of its internal data structures.

4.2.1 Distributed

A distributed programming model works by letting the master network interfaces at the IP cores request the set up of virtual circuits. A master NI sends a setup packet using BE towards the destination of the virtual circuit. Whenever the packet arrives at a router it is setup to facilitate the virtual circuit and the setup packet moves on towards the destination. If the setup packet at any point fails to setup the virtual channel it reverses its path and tears down reservation that was done in previously visited routers. Upon arrival at the destination, the setup packet returns to the master and the virtual circuit is setup. In a model as just described the global knowledge of the state of the network is lost from a network point of view. Because of this, it can be hard to determine the state of the network and guaranteeing a reliable setup and a fixed time frame for the setup phase. Because the virtual circuits are requested from the IP cores connected to the master NI the system must have a knowledge of which virtual circuits are present at any one point.

4.2.2 Central

The central programming model has a central controller that handles the setup of the network. When the controller sets up virtual channels it fetches a configuration file from a memory which can be off chip. This configuration file is then split into a configuration file for each router and distributed to the appropriate routers in the network. When a router receives the appropriate configuration file it adopts the configuration and sends an acknowledgement back to the network controller. In this way, there is always a central controller that knows the exact state of the virtual circuits across the entire chip.

4.2.3 Hybrid

The last programming model discussed here is a hybrid of the two that has already been discussed here. The central controller from the central model is kept, but the setting up of the routers is distributed as it is in the distributed model. The central controller fetches a configuration file and sends a packet to the master NI of a virtual circuit that needs to be set up. When the NI receives this packet it, much like the distributed model, sends a packet towards the end point of the virtual circuit and sets up all the virtual channels on the route. The NI sends an acknowledgement to the central controller when the set up packet returns to the NI after having set up the virtual circuit. Again, this as in the central model the controller has the knowledge of all the virtual circuits.

4.3 Distribution of the Use Cases

In all three programming models discussed in section 4.2 there is a need for distributing configuration packets to routers and NAs. This section will discuss how this configuration information can be distributed in the network as use cases. When considering that the setup packets have to propagate through a large network, in some cases all parts of the network has to recieve configuration information, the distribution can impact the performance of the NoC. The following will describe three different types of NoCs, where the distribution of configuration information impacts the NoCs differently.

NoCs that only setup connections during start-up, have no other traffic to contest for the network ressources. Therefore these network *can* rely to a large extend on waiting *long enough* for the configuration to finish. The setup time becomes low, due to the low amount of traffic being distributed in the net, but the exact time it takes to setup the network must be established through simulation.

The above NoCs can not change use case during operation, but for NoCs that have the ability to reconfigure connections setup traffic will have to contest for the network ressources. The setup time will therefore be less predictable and the setup time is hard to establish through simulation. The setup can be done under the assumption that not only does the network change state or use case, but so does the application, which gives the NoC time in the order of miliseconds [26] to switch use case. Because the application is also changing state, it may be possible to drop packets in order to propagate the setup packets faster in the network.

As described in section 2.3 there are NoCs that allow for applications to maintain connections through use case switching. In those types of NoCs dropping packets or temporally interrupting streams are not an option and the exact state of the network becomes important in determining when the network is done being reconfigured. The state of the network will be discussed in the next section, here follows a discussion of how the distribution is handled.

4.3.1 Distribution

There are several ways that use cases can be distributed: (i) using BE packets; (ii) BE packets with priority; (iii) a virtual signaling net; (iv) introducing a physical net. The following will describe each of these in more detail.

Ordinary Best Effort Traffic

Distributing setup packets as BE traffic is a relatively simple implementation. The BE router examines all incoming BE packets to determine where they have to go, and thus checking whether it is a setup packet is a trivial expansion. Because the packets are distributed on BE the setup time might become very large.

Prioritised Best Effort Traffic

This gives rise to the idea of being able to flag the setup packets priority. This would give the benefit of having the easy implementation of solution one, but introduces an upper limit on the setup time.

Guaranteed Service Traffic

When introducing priority on the BE packages, an obvious extension of that concept is to implement a virtual signalling net. A virtual signalling net could be implemented as one of the virtual channels on each link. The virtual signalling net could also be used for other types of signalling, ie. interrupts. This approach gives good guarantees on the setup of the virtual circuits, and would be able to work in very highly utilised networked. However, if the overall number of virtual channels in the network is low, a virtual signaling net would be expensive in terms of resources used. Even network with many virtual circuits would see an impact of a signalling net because the level of guaranteed service that a network could offer would be lower than if the virtual signalling net was not there.

A Physical Network
Because of the impact on resources and performance a virtual signalling net has, introducing a physical network for the setup distribution becomes an option. This setup network could be implemented in several ways: direct wires from the controller to every router and NA in the network, a network with same topology as the NoC or a network with a different topology than the NoC. Using a physical network to distribute the setup packages to the routers will give the benefit of guaranteeing the setup time, without impacting the performance of the NoC. The price of another physical net is an increase in area, depending on the actual implementation of the physical net. The second net could arrange the routers in a ring topology and thereby reducing the amount of extra wires used.

4.4 Network State

As discussed in the above section, the current state of connections becomes increasingly important as use case switches become more complex. The state of the network includes information on what virtual circuits are setup or being setup, what connections are in use and whether or not BE paths are setup if needed.

As described in section 4.2, the choice of programming model greatly affects the traffic patterns of configuration traffic. It also affects how the progress of the configuration process can be monitored, as the state information in a distributed scheme is spread in the entire network. The distributed scheme will most likely solve this problem at the system level, but knowing exactly when a circuit has been setup would be very useful information. In the two central based programming models knowing when the entire use case has been setup could be used to allow the application to use the connections earlier thus minimising the delay caused by use case switches.

Acknowledgement on configuration packets can be used to keep track of the configuration process, as every configuration step is acknowledged. Waiting for a response from each configuration step however means that the configuration process will stall and wait for the acknowledgement. One way to circumvent this is by threading the configuration process and allow for several configuration packets to originate from the same controller. This however requires that the NA and IP core can handle accounting transactions.

Reading configuration tables in the network ressources can be used to determine the current state of the network. In all three programming models the current state of the network is available in the network ressource, i.e. routers and NAs, because the virtual circuits are setup in the actual router and NA. Reading the configuration tables is simply an addition that is nice to have in a NoC, as it allows for any IP core using the NoC to determine its state.

CHAPTER 5

The MANGO Network-on-Chip

Because the goal of this project is to make MANGO capable of reconfiguring virtual circuits during execution.

MANGO has been built in a very modular fashion, which makes it very customisable, i.e. the arbitration algorithm can be exchanged by another, a new routing algorithm can be used, without changing the general flow control of the BE router. Or another flow control implementation can be used, while maintaining the current routing implementation. This fact makes MANGO a very desirable platform for exploring different aspects of NoC research on an asynchronous implementation. Because MANGO has been used as a proof-of-concept for an asynchronous NoC that offers guaranteed service, it is implemented in a standard cell library. This limits the implementations and features that can be explored easily due to the fact that standard cell implementations take a long time to understand.

This chapter will describe how MANGO operates, specially concerning guaranteed service. During the presentation of features, challenges and problems in the current implementation will be highlighted. The following chapter 6 summarises these challanges in a specification and then presents an in-depth description and discussion of the challenges and solutions hereof.

5.1 Setting Up Virtual Circuits in MANGO

Guaranteed service connections in MANGO are as mentioned based on virtual circuits. The virtual circuits are made up of VCs between the network ressources along the corcuit. VCs are supported in MANGO between the routers, and the lowest priority channel is used for BE traffic. Between a router and its connected NA the number of VCs are four, 3 GS and 1 BE. The VCs are linked together in the routers by pointers that are initiated during the setup phase, thus these pointers must be distributed to the routers. Two types of pointers exist in MANGO, *steer* and *select* pointers, which handle the forward and backward flow control respectively.

The distribution of these pointers along with GS information to the target NA is done using the BE network, presented by MANGOs BE router. The pointers are written into memory mapped registers in the routers, which allows OCP transactions to be translated to setup packages in the network. The BE network is source routed and therefore the NAs must be able to translate global addresses into BE paths. MANGO takes a very flexible approach that allows each NA to present its own view of the address space, meaning that one location in the global memory can be adressed by different addresses depending on the NAs view. Even more flexibility is added to the topology as the BE paths are a list of output ports in the routers, and the forward and reverse paths are uniquely defined in the flit header. The translations from global adresses to BE paths are written by the master core to its NA. The top 8 bits of the global adress is used to identify the BE paths. A maximum of 16 BE paths can be in the NA [29], but the master core can replace specific entries if needed.

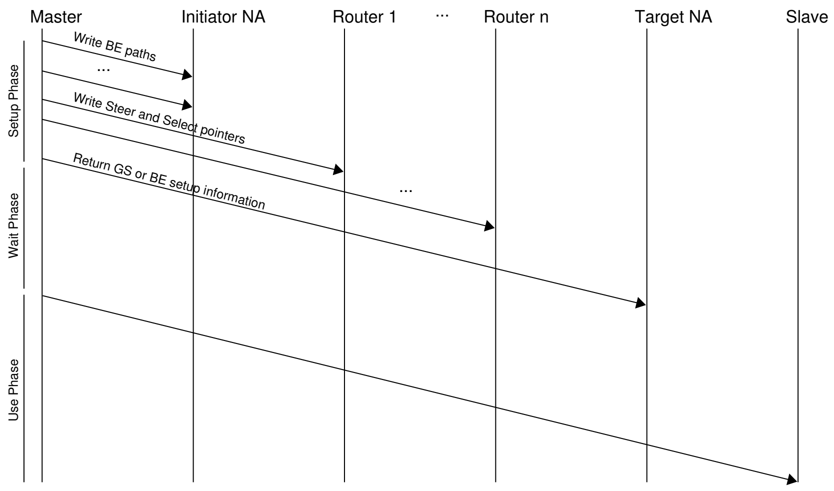


Figure 5.1: Using MANGO GS connections can be split up into a setup phase and a use phase. In the setup phase setup information is distributed to the network ressources, and in the use phase the GS connections are used to transport data between master and slave. In between is a wait phase which ensures that the virtual circuits are fully setup before they are used.

As mentioned, the GS connections are setup by distributing the pointers to the routers and target NAs. Figure 5.1 shows a typical setup phase in a MANGO. As can be seen from the figure no confirmation is sent back to the master and the order in which the routers are set up does not matter because the setup requests have no affect on each other. One could choose to interleave the different parts of the setup phase, write BE path to router 2; setup router 2; write BE to router 1; setup router 1; etc. It should also be noted that nothing dictates that new circuits cannot be setup during run time, but as will be explained in section 6.2 a VC can be left in a wrong state if the pointers are reconfigured during run time.

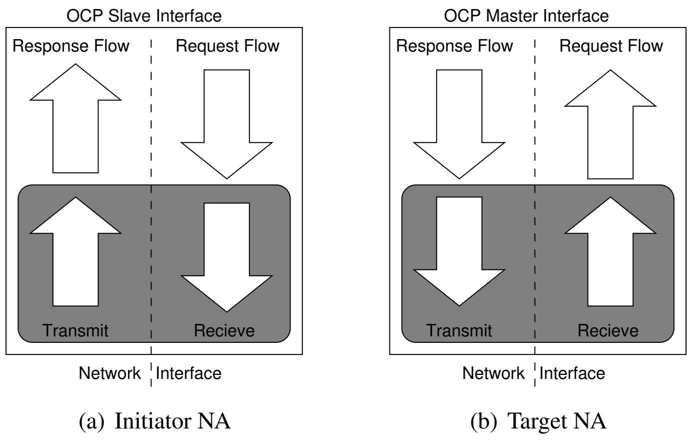
As can be seen from the figure it is hard to predict when the setup of the ressources has finished. The current approach is to wait *long enough* in the wait phase before using the virtual circuts. This has a few complications, first off reconfiguring the network is not feasible due to the unknown time frame for the setup packages, thus the delay between stopping one use case and starting another could become very long. Secondly, the uncertainty also means that the setup could have failed, but the system will never know. This can be handled by implementing acknowledgements on setup request. This should of course be in both NAs and routers.

When virtual circuits have been setup the only knowledge about them being available is at the master who set up the circuit. Other masters have no knowledge of this, which could in some cases be very interesting. One could imagine an adaptive creation of virtual circuits, i.e. virtual circuits are created as needed by a master. The master could then probe the network ressources along a path to the slave core and either set up a virtual circuit if there is room for another circuit in the routers. In the case where the circuit can not be setup or the required guarantees can not be met this could be reported to the application. To determine the state of the routers and NAs in terms of available virtual channels an OCP read of configuration registers should be available to the master cores.

5.2 The MANGO Network Adapter

The MANGO NA is actually two different NAs, an initiator and a target NA. The initiator sends requests to the target which in turn sends a response if needed. Both NAs are split in two ways, asynchronous vs. synchronous; and request flow vs. response flow. MANGO is an asynchronous network with synchronous OCP interface. There is thus an obvious need for converting between the clocked domain of the OCP cores and the clockless domain in MANGO. The request flow is the forward going flow of data from an OCP master to an OCP slave, which means the request flow in the initiator sends data and the request flow in the target recieves data. The response flow is invoked when a request requires a response, the target NA sends the response through the response flow and the initiator recieves through it. Figure 5.2 shows the clock and clockless domains and the direction of flows in the two NAs. The reader is reminded of the defininitions concerning the NA in section 2.1.3, especially figure 2.3 on page 8.

The OCP master core writes the BE paths and GS information to the initiator NA through its CI. This means that each master core must know that it is connected to a particular network. A central controller of the network handles all the setup in the network is therefore not possible. If an initiator NA could handle configuration request through its NI, MANGO would be able to support a central programming model in addition to the distributed model currently used.



(a) Initiator NA (b) Target NA

Figure 5.2: The MANGO NA handles the conversion between the synchronous OCP interface and asynchronous (grey) network. The end-to-end flow between IP cores is handled by the request and response flow.

5.3 Best-Effort Routing

BE routing plays a major role in the configuration of MANGO and an implementation of a reconfigurable MANGO would also rely on the BE network. MANGO uses an XY-routing alghorithm to avoid deadlocks, in line with the proof in [11], which guarantees that if all packets are routed fully in X and then Y deadlocks can be avoided. MANGO is source routed so any routing path could in principle be choosen. The routing alghorithm is implemented by merging the five input ports, latching the flit and then demultiplexing the flits in the right direction. The merge and the demultiplexer are sticky, meaning that once the first flit of a packet is detected the rest of the packet is handled before all other flits. Once the *end of packet* (eop) is detected, the merge and multiplexer is released. Because of the single storage element as can be

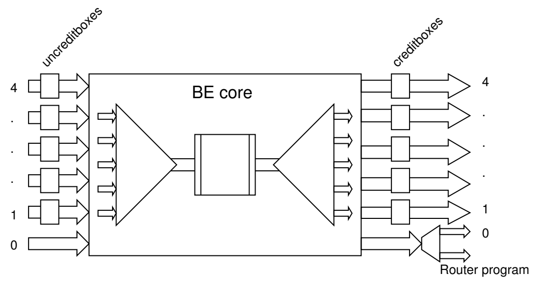


Figure 5.3: The BE part of the MANGO router, the core is embedded in between the credit handling. The BE core consists of a sticky merge, a latch and a sticky multiplexer.

seen in figure 5.3 unresolved deadlocks exist in the current implementation.

The performance of the BE network has a large impact on the setup time, especially if ordinary traffic is allowed during a reconfiguring phase. Therefore, the discussion in section 4.3 is very relevant for a reconfigurable MANGO. In addition, a new BE router and routing scheme is needed to guarantee that deadlocks are avoided. The current router can be seen in figure 5.3 and in line with the before mentioned modular approach of MANGO the router core handles the routing alghorithm and the credit boxes surrounding the core handle the flow control. The split on the output of port 0 splits flits to either the local port or to a programming interface on the GS router.

CHAPTER 6

Reconfigurable Virtual Ciruits in MANGO

Based on the discussion of virtual circuits and guaranteed service in chapter 4 it is obvious that creating reconfigurable circuits can be done in many ways and the overall network and system must be taken into consideration. The current state of MANGO allows for virtual circuits as it is, but as described in chapter 5, there are several areas in which MANGO's support for virtual circuits needs to be improved to allow reconfiguring. The following section holds a specification for a modified version of MANGO that can be reconfigured. The sections following the specification discusses implementation and design ideas and considerations.

6.1 Specification

The following specification is based on the discussion in chapter 5.

- Unused virtual channels are **reusable** for new connections.
- Any initiator network adapter can **program** any other network adapter.
- Enable the configuration **state** in the network ressources to be determined.
- Use **prioritised** BE traffic to distribute configuration of virtual circuits and other important BE traffic.
- New configurations are **acknowledged** by receiving unit.
- A new **routing** alghorithms for BE routing, is needed.

The following sections describe solutions to each of the six items in the above specification. The only thing that really stops MANGO from reconfiguring its VCs to form new virtual circuits is the fact that VCs can be left in different states and can therefore not be connected to form new circuits. Solutions to this problem are discussed in section 6.2.

The introduction of a central controller of the network is a major feature and the current initiator NAs do not allow for that. This is the topic of section 6.3.

A nice to have feature, which could lead to adaptive creation of virtual circuits, is the ability to determine which VCs and BE paths that are currently setup in the network. Solutions to this are described in section 6.4.

Section 4.3 highlighted the trouble of distributing setup packets while the network is being used. This can lead to a large, maybe too large, setup time, and section 6.5 discusses this within the scope of MANGO.

As mentioned earlier, MANGO just distributes the configuration and then waits long enough. This is not a viable solution and the introduction of acknowledgement is described in section 6.6.

The current routing scheme used in MANGO makes the BE router *very* small, but it can not be guaranteed that deadlocks will not occur. Section 6.7 therefore introduces alternative routing solutions.

6.2 Reusable Virtual Channels

A virtual circuit in MANGO consists, as described earlier, of one or more virtual channels connecting an initiator NA to a target NA. A virtual channel is implemented across a shared media, from one router's output port to the neighbouring router's input port. This section will first provide insight in the problems with the current implementation of connecting VCs in regard to reconfiguration. Then a solution to the problem will be described and discussed.

Virtual Channel implementation in MANGO

In MANGO a virtual channel consists of, in addition to the shared link, a lock, a key, and an unlock wire, figure 6.1 shows how these elements interact. The actual implementation of the virtual channel separates itself from the figure in where the key box and buffer is placed inside the router. This implementation is done due to the fact that MANGO is output buffered, so for the key to have information regarding actual buffer space on a link it must be placed near the buffer.

The lock and key boxes are both placed at the output port of the router as illustrated in figure 6.2. The unlock wire is routed through a switch controlled by the steer and select pointers mentioned earlier. The switch is a fully connected crossbar composed of multiplexers. Looking again at figure 6.1 the unlock wire can thus be connected to all the outputs of all the neighbouring routers. The key toggles the unlock wire when a falling edge on the acknowledge is detected. An even amount of toggles indicates that an even number of flits have passed through the VC, leaving the unlock wire in its initial state. If an uneven number of flits has passed the VC

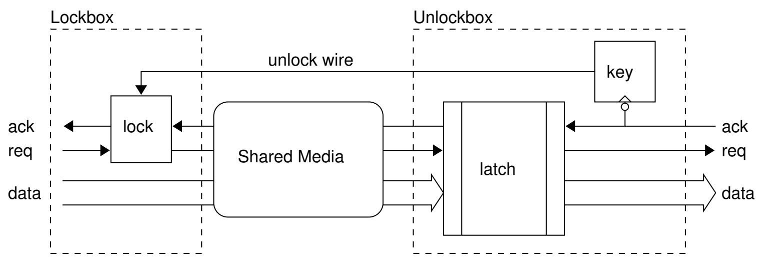


Figure 6.1: The virtual channel and the lock unlock boxes which MANGO uses as flow control on inter router links. [7].

the unlock wire will not be in the inital state, and if switched to another lock it could toggle the input of the lock causing a false signal in the lock. This would create a false handshake which has dire consequences for an asynchronous system.

The key and lock converts between the 2 phase protocol on the shared link and the 4 phase protocol used within the routers. Because the lock and unlock are placed inside the output port the unlock wire is 2 phase inside the unlock switch. This lowers the switching activity, but it also causes the problem with switching the unlock wires to other ports.

In the following, three solutions to reconfiguring the NoC is presented, (i) flushing of the VCs to ensure initial conditions; (ii) moving the handshake conversion in order to isolate the unlock wire to one VC and; (iii) implementing a merge like component that enables event-based unlocking.

The assumption used in the following is that the decision to reconfigure is done at application level, which means that no conflicting configurations will be initiated

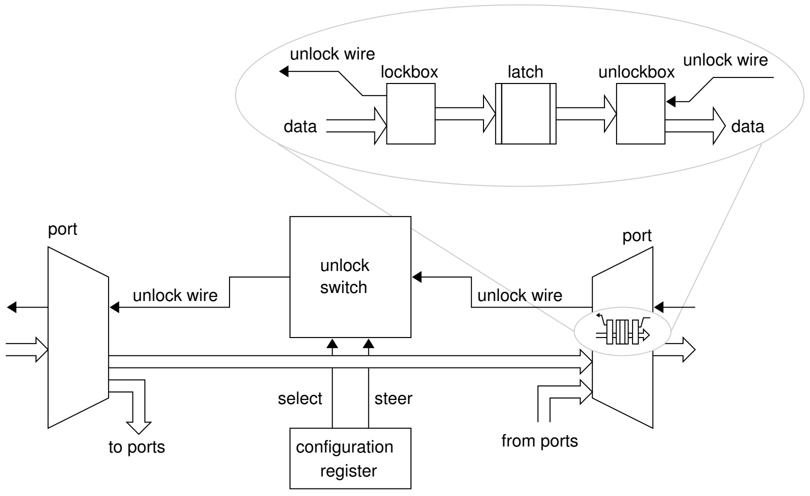


Figure 6.2: The lock and key box that controls the virtual channels are placed very close in the output part of the router.

at the same time and that tearing down of connections only happens on connections that are not being used.

Flush to an Even Number of Flits

As proposed in [8] the unlock wire can be brought back to its initial state by ensuring that an even number of flits have passed on the VC. To ensure that a virtual circuit has seen an even amount of flits when being torn down, the master NA would count the number of flits transmitted on each virtual circuit. When a virtual circuit is no longer in use the master NA would then transmit a tear down packet to close down the virtual circuit. The number of flits in the tear down packet is dependant on the amount flits already transmitted on the virtual circuit.

The NA only needs to know if the number of flits transmitted on a given virtual circuit is even or uneven. The counter can therefore be implemented as a simple toggle flip-flop. As a master NA, in the current implementation of MANGO, offers three GS connection and one BE connection to the IP core, only three of these toggle flip-flops are needed. The tear down packet must be transmitted along the virtual circuit when the application indicates that it is done using the virtual circuit. If the packet were returned to the initiator NA, when the tear down of a virtual circuit is completed, the state of the virtual circuits could be tracked in this way. Virtual circuits consist of a forward path, from master to slave, and a backward path, from slave to master. A mechanism for flushing the virtual channels must therefore be available in the target NA. In the MANGO NA the synchronous part of the NA handles packets and conversion to flits [6]. Therefore the flit count should obviously be done in the synchronous part.

The NA will increase in size due to the extra counters and the delay on setting up new virtual circuits might increase as old ones will have to be flushed. This delay could be hidden if some or all virtual circuits can be flushed prior to use case switches.

Move the Handshake Conversion

This solution extends on the phase conversion discussed briefly above, by moving the handshake conversion to the edge of the router. By moving the key to the physical link it unlocks, the 4 phase to 2 phase conversion is moved till after the unlock wire switch. In the current implementation, one key can be connected to every single VC on the other 4 ports of the router. Moving the key effectively ties the key, unlock wire and lock together, thus avoiding to switch different keys and locks toghether. The unlock wire switch will now experience full 4-phase handshake and the unlock wire will still be 2-phase. The input to the key always returns to the same state after each handshake, and key and lock pairs will never be in different state.

By moving the handshake conversion, the router will be all 4-phase and the link will be 2-phase. The unlock wire switch will see more switching activity since the signals it routes are now 4-phase instead of 2-phase, thus increasing power consumption. The area overhead is minimal if at all.

This is the solutions choosen and the modifications to MANGO can be seen in appendix A.1. Figure 6.3 shows the new placement of the key.

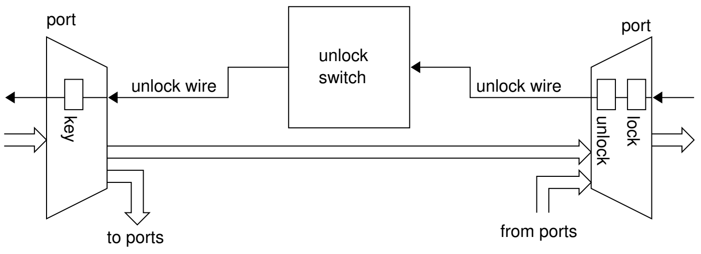


Figure 6.3: New location of the key.

A Gated Merge

The unlock wire switch must enable any unlockbox to be connected to any lockbox in the neighbouring routers, and is therefore implemented as a full crossbar. The crossbar is controlled by the steer and select pointer, and the crossbar itself is composed of large multiplexer trees, which connects a VC input to all the VCs at the output, including the local port. This results in a 24 port crossbar, three from the local port and seven from each of the other three ports on the router. Only three ports because it is no possible to route GS connections back to the port they came from.

When reconfiguring the crossbar, false handshakes can be generated as described above. Therefore the solution presented here is to decouple the input and output of the unlock wire switch. In this way, the unlock wire switch will only propagate events from its selected input ports to the output ports. If mutual exclusion on the inputs of each multiplexer tree can be guaranteed, this can be accomplished by replacing the multiplexers with a 2-phase merge. Because only one signal is routed through the unlock wire switch and no handshaking is done, the merge can be implemented as XOR gates. However mutual exclusion is in no way guaranteed, as several VCs in one router will be operational at the same time and every tree sees the events of the other VCs. A controlled arbitration mechanism is therefore used to avoid that the merge sees the events from the other VCs. This will be called a gated merge and its principle is illustrated in figure 6.4. The decoupling of in- and output is done in the gates. Only when the gate is open does an event on the input of the gate trigger an event on the output of the gate. Mutual exclusion is guaranteed by an address decoder that at any one time at most has one gate open. The gates can be operated by the steer and select pointer if properly decoded. The XOR shown in the figure is a tree of XOR gates like the current multiplexer tree.

Any extra latency that might be introduced through the unlock wire switch is in the reverse latency path and should therefore not have an impact on the overall

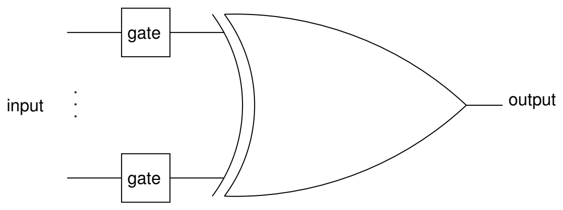


Figure 6.4: The gate propagates events through when the gate is open and thereby ensure mutual exclusion on the XOR gate. The XOR gate is implemented as a tree of 2 input XOR gates.

performance of the link. It would change the requirements for the VCs, due to the way links are pipelined in MANGO, as described in the study of MANGO in section 3.7.

6.3 Programming the Network Adapters

To provide the flexibility needed for the different programming models discussed in 4.2, any NA should be programmable from any master core. The following will describe solutions that allow for this. Only changes are needed in the initiator, because the target is inherently capable of supporting configuration through the network.

Two solutions are proposed here, enable both the NI and CI to write to the configuration register and write configuration through the NI only.

Write Through Both Core and Network Interface

As mentioned earlier, the initiator's configuration register is placed inside the request flow and the network port can not access the register. The register only has one write port, but this can be easily modified either by adding another write port to the register or inserting a multiplexing mechanism in front of the register to emulate two write ports. Multiplexing between writes from the CI or NI requires that they do not try to write at the same time. Handling two writes at the same time would require a more complex multiplexing mechanism or two write ports on the register. There might be constraints on the addresses that can be written at the same time. However two writes at the same might not be needed. If a central unit controls the routing paths, they will only be written through the NI and in a distribute scheme only the CI will write to the register. In the hybrid scheme also described in section 4.2 the central controller will notify the master core, which in turn will setup its own routing paths as it is done in the distributed model. Figure 6.5 depicts the layout of the initiator NA in which the register can be accessed from both the CI and NI.

The response flow of an initiator NA can handle two types of packet formats in the current implementation, namely responses and interrupts. If it is assumed that the only requests an initiator will see from the network are configuration requests, then

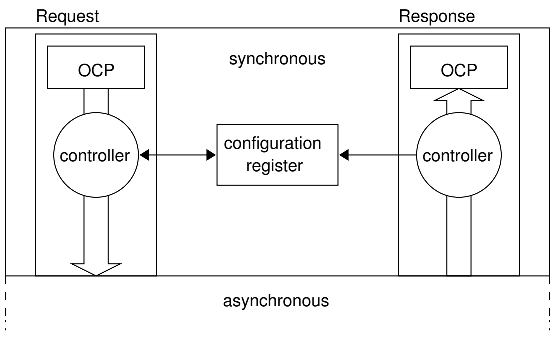


Figure 6.5: The configuration register is moved so it can be accessed by both flows in the initiator NA.

single writes can be used to program the NA. Requests can be distinguished from responses and interrupts in that they have a different package format.

Setup Through the Network Interface

As observed above, only one of the interfaces will write to the configuration register at any one time, thus one write port should be sufficient. To allow any master core to write, the NI is an obvious choice as the only interface that can write. This implementation is similar to the implementation in the target NA. For a master core to write to its own initiator NA, it must thus send the request throug the locally connected router as shown in figure 6.6. Special care must be taken to ensure that a NA can forward a configuration request to it self through the local router without any routing information set up.

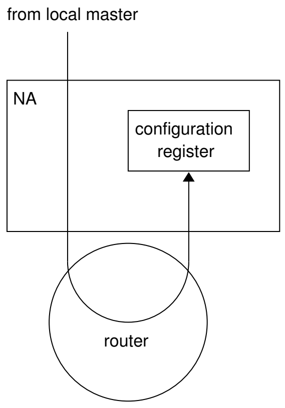


Figure 6.6: Only the NI can write to the configuration register, therefore the local master core must write through the locally connected router.

The implementation choosen, is to move the register so that is can be accessed by both the request and the response flow. No changes to the functionality of the request flow has been done. The response flow however has been changed to allow requests to be handled as setup packets. The changes and additions to implement this solution are indicated in appendix A.2.

6.4 Determine the State of the NoC

The state of the the NoC is composed by information present in the network ressources. In a central scheme, this information is available both in the network ressources as well as in the central controller. For the central controller to have a complete picture of the NoC, it must be the only one accessing the configuration ressources in the network. For processes running in the system to gain a view of the state, either the entire system state or part of it, the process must contact the central controller to gain this information.

If the configuration is handled by several processes, the knowledge of the system state is spread and several ressources must be inquired to gain a view of the system. Furthermore, the system state can change during the inquiry leaving the requesting process with an imprecise picture of the state.

As mentioned, the system state is available both in the processes that controls the setup, be it one or more, and in the network ressources. Again, inquiring all the network ressources may result in an imprecise picture of the state. Network ressources in MANGO do not support reading the state of network ressources.

Because a precise picture of the system is hard to give, one must be aware of what such a picture of the state is used for. It has been mentioned earlier in this report that determining which ressources are available could be used to create an adaptive configuration of virtual circuits. It would be possible to create a system where the NoC is hidden from the application in such a way that connections are created as needed. Thus the application would just request a stream or data from an address and the NoC would then ensure that connections are set up to accomedate this.

It has been chosen not to implement any means of determining the state from the configuration in the network ressources. The only way to determine the state is to know which circuits have been setup by one or more master cores.

6.5 Controlling the Setup Time

In section 2.3 use cases and the desire to switch smoothly between use cases is highlighted. Section 4.3 discusses ways to distribute the setup packets in the network, one thing to notice is that switching use cases usually takes time at the system level, given the NoC time in order of milliseconds to switch its use case.

MANGO uses the BE network to distribute the setup packets, this approach works well in the current MANGO implementation. Configuration is only done once during initialisation where no other process uses the network. It is therefore safe to assume that the setup time will be low and waiting for the setup to finish is a feasible approach. When allowing use case switches at run time and smooth switching, the assumption that BE traffic will arrive fast must be reconsidered. Packets transmitted as BE is not provided with any form of guaranteed minimum arrival time, however in MANGO the BE router accesses the shared links as a VC, but with lowest priority. This means that the BE router will get a packet across the link fairly often due to the way ALG operates, as described in section 3.7. This does not affect the overall guarantee on the BE as individual BE packets must still contest for the link with other BE packets. The following will discuss the distribution schemes of section 4.3 in the context of MANGO.

Ordinary Best Effort Traffic is the approach used now and has the advantage of being relative simple, but having the downside of being very unreliable. Introducing a new BE router in which output ports does not block each other, will decrease the average time it takes to distribute the configuration packets. If the amount of traffic in the network is high, it might not be possible to setup the network within miliseconds.

Prioritised Best Effort Traffic allows BE packets with priority to overtake other packets. Depending on the implementation a packet with priority can overtake packets incoming on the same or overtake packets leaving on the same port. The ideas from QNoC (section 3.2) can be used to create a priority scheme, the cost is area and more complex arbitration in the BE router.

Guaranteed Service Traffic can give hard guarantees on the configuration phase. It would however require extensive changes in the way MANGO routers are implemented, as GS is not examined at all, just routed according to the steer and select pointers. It is not a feasible solution for this project.

A Physical Network also offers great control over the distribution of configuraton information as in the GS above. The area and complexity overhead of such an implementation is probably not going to be feasible in NoCs.

It has been choosen to keep the configuration traffic in the BE network and not priotise it. This togheter with the changes to the routing alghorithm discussed in section 6.7 has lead to the BE router implemented in this project. Appendix A.4 describes the implementation of the new BE router.

6.6 Acknowledgment Configuration

The approach of MANGO for ensuring that the configuration is done is to wait *long enough*, which is an unknown period of time. As mentioned earlier this is not a viable approach if the configuration of the virtual circuits is to be changed. Let alone is it a good approach if setup is only allowed during initialisation, it introduces uncertainty in the system that should be avoided. This section describes ways of implementeting acknowledgement from the network ressources. The router, target and initiator NA implementations of acknowledge is discussed after a general discussion of how acknowledgements will the affect the overall configuration phase.

MANGO uses OCP writes to configure the network, therefore the obvious solutions is to ensure that the OCP master waits for acknowledge on writes. By expanding the subset of OCP commands to include the Write Non-Posted [30] command, the master will recieve the added information that the configuration has been completed. It is possible to add any erros that might have occured in the response acknowledgement. In this implementation it has been choosen to do just this and expanding the OCP command set, an alternative solution is to handle the acknowledgement at NA level. This removes the information the acknowledgement was suppose to add to the system namely a status on the configuration state and is therefore not a viable solution.

In extension of the discussion regarding setup times in section 6.5, it should be noted that the impact of high traffic is even larger when adding acknowledgement. This is due to the acknowledgement having to return to the sender using the BE net. Furthermore the master core can not initiate a new OCP commands in the current implementation of the NA, if the NA could handle several pending request, different threads could be exploited to minimise the delay.

Router

When setting up virtual circuits the routers recieve a configuration to setup the steer and select pointer this needs to be acknowledged. The current router implementation is shown in figure 5.3 on page 44. A configuration request can be distinguished from ordinary request because they are destined to the local port of the router and has a router programming bit set. The split seen on output port 0 of the router forwards the configuration request to the programming interface. For the acknowledgement to indicate succes in addition to arrival of the configuration, the acknowledgement must be generated near the programming interface. The acknowledgment should not be generated inside the core of the router for this reason aswell as making the acknowledgement independent of the routing scheme.

Another important aspect is how the acknowledgement is inserted in the network, as it can affect the operation of the network. The obvious way is to add an extra input port to the router where the acknowledgement can be inserted. The implementation choosen is to add a merge in front of input port 0 (the local port), as can be seen in figure 6.7. It effectively creates six input port in addition to the six output ports that is created by the programming split, this is important as it affects the routing scheme discussion in the following section 6.7.

The latches added on the acknowledge path is added to ensure that the flits can move towards the merge without stalling the local port. Two are needed when a master configures its locally connected router. In the implementation described in appendix A.3 the acknowledgement is generated when the programming interface is

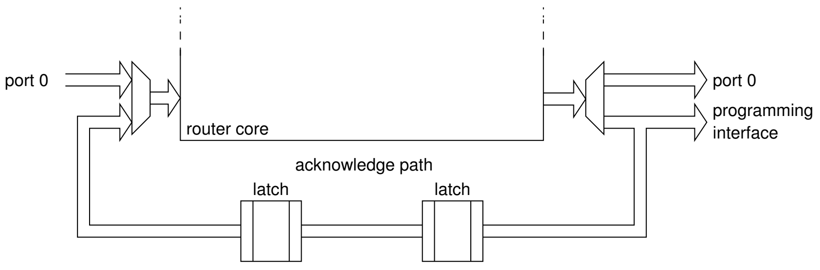


Figure 6.7: An acknowledge path is added to the local port of the router, in addtion to a acknowledgement generation block and a merge.

requested and then construct the acknowledgement. It can therefore only assume that the acknowledgement succeeded, when issuing the acknowledge.

Target Network Adapter

When a configuration requests arrives at the target data is setup on the configuration register and a write is executed. This is all handled in the synchronous part of the request flow. Two existing features of the NA is used to create the acknowledgement, namely the storing of return paths and the interrupt. The NA stores the return path in a register which is inserted in the response packet. The interrupt is triggered via an input port in the response controller which then generates the interrupts and transmits it across the network to its destination.

A target NA is only capable of handling one request at the time, the return path register can therefore be used to create the acknowledgement. The request controller sets up an acknowledge signal to the response flow which treats it like a interrupt, but uses the stored return path to create the response. It should be noted that the acknowledge is not generated until the configuration register signals that the write is done. Appendix A.3 describes the changes done to implement this.

Initiator Network Adapter

Because of the modifications done in section 6.3 the inititator can be programmed through both interfaces. For configuration arriving at the NI, the acknowledgement should be send to the the master core that requested the configuration. In the case of configuration from the local master the acknowledgements should be returned to the master through the CI. For configuration requests on the CI to the local NA the write to the registers is done at once. Therefore implementing acknowledgement on local configuration is just increasing the overhead. In the case of a central controller it is very likely that the BE paths in the local NA is going to be exhanged quite frequently, as it must address a large number of ressources in the network. Therefore adding extra overhead and maybe extra delay as the acknowledge is completed is not the right choice, thus no extra acknowledgement between the CI and the local master has been implemented.

Turning to the NI, the desired behaviour is very similar to the one in the target NA described above. The same approach can however not be used because the initiator NA does not store the incoming BE paths for use as the return path. The initiator NA have no used for a return path in other cases and the first flit holding the path is therefore discarded when recieved. To circumvent this problem an alternative solution has been explored, the return path could be added to the list of BE paths in the configuration register. If the return path was added with a fixed address this could be used to select it when returning the acknowledge. To access a specific BE path a 8 bit address is given to the configuration register, but in the implementation of the initiator NA this 8 bit address is taken from the CI and thus not accessible from within the router.

Another difference to the target is when the acknowledge can be handled, interrupting the request flow to transmit the acknowledge can have consequences for the given guarantees and should therefore be avoided. An alternative is to buffer the acknowledge and transmit it during idle stages, this could however cause for very long delays on the acknowledgement and slowing down the configuration phase considerablythe .

Several problems with implementing acknowledgement on NI configuration requests have been highlighted above. Some challenges can be overcomed with reasonable modification, but all in all the best solution would be to implement more interaction between the two control flows which would require extensive modifications of the initiator. This is well beyond the scope of this project and no implementation of acknowledgement in the initiator has been done. The discussion in chapter 8 further elaborates on this.

6.7 Routing Schemes

In the previous sections topics regarding the number of in- and output ports and throughput in the BE router has been highlighted. In this section routing schemes will be analysed and an implementation of the new BE router in MANGO will be described. The BE network of MANGO is currently implemented by a BE router where the flow control is handled by credit exchanged and the routing alghorithm is handled by a very small BE router as described in section 5.3. The added port and the acknowledgement generation in the router increases the risk of deadlocking the current router. The following will analyse new routing implementations for the core part of the BE router.

It is choosen to only investigate deterministic source routed alghorithms, because the main goal of implementing a new BE router is to avoid the possible deadlock situations in the old router. Source routing is how it is currently done in MANGO and it is choosen not to change this, distributed routing schemes is more complex and the routers must know the topology of the network.

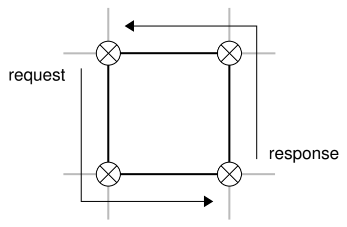
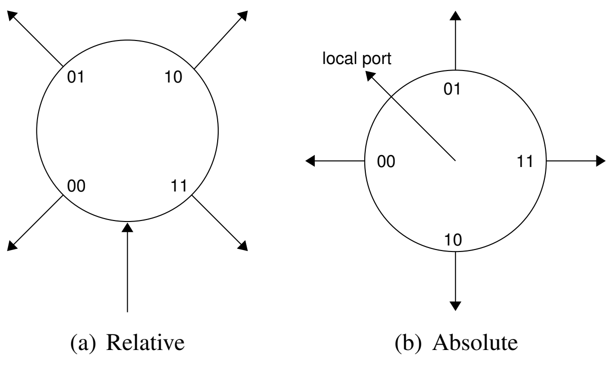


Figure 6.8: The response and the request follows to different paths to comply with the XY routing alghorithm.

Two of the most used deterministic routing alghorithms are XY-routing and BE with seperate channels for request and response.

An XY-routing alghorithm routes fully along the X-axis and then along the Y axis to reach its destination. This means that the request and the response might follow different paths, depending on the topology and the location of the communicating nodes. In a mesh net this can easily be obtained as figure 6.8 illustrates, in irregular topologies extra care must be taken to ensure proper operation.

Responses and request can be seperated by using two sets of buffers, one for requests and one for responses. This effectively creates two networks and cyclic dependencies can be avoided, if the requests and responses can be decoupled at the end point. This is generally the case, but it is important to ensure. The path of the request and response need not be the same, but in many cases it is which simplifies the routing decisions and avoids topology dependencies.



(a) Relative (b) Absolute

Figure 6.9: In an relative scheme the local port can not be distinguished from the generic ports, where in the absolute scheme routing backwards indicate the local port should be selected.

A path is a list of either directions or ports the packet must follow from its source to the destination. Directions are used in relative routing schemes where the output ports are numbered relative to the input ports. The alternative is to give the output ports absolute values. With five ports the hop can be indicated with two bits, because it rarely makes sense to send the incoming packets out the same way it entered. As note erlier the acknowledgement and the programming port on the MANGO routers effectively creates six ports, which means that MANGO must use at least three bits for each hop. Figure 6.9 shows how ports can be numbered in a relative and in an absolute scheme. The figure depicts two routers that use two bits to address the output ports.

The forward path is used for requests and the reverse path is used for responses, in its current form MANGO uses a two part BE path, which holds both the forward and the reverse path. This makes it very flexible as the two paths can be completly independently created, it is also needed because of the way the current router is constructed. It further limits the amount of hops on a path, by constructing the reverse path from the forward path more hops can be reach, the cost is more complex logic in the routers and NAs.

6.7.1 The New Best-Effort Router

It has been choosen to implement a fully connected crossbar as BE router to increase the input ports that can be served at once. This has been choosen to avoid dependencies between the input ports that could result in a deadlock. Figure 6.10 shows the new router which is build of five splitters, five merges and five latches, one for each of the ports one the BE core. Interleaving of packets is avoided at the merge which only selects a new input port after it has seen the end of packet. The organisation of the splits followed by the merge ensures that the split will always operate on complete packets which means that only the merge has to avoid interleaving of packets. The split selects and output based on the BE path, which is then rotated and inverted to create a reverse path.

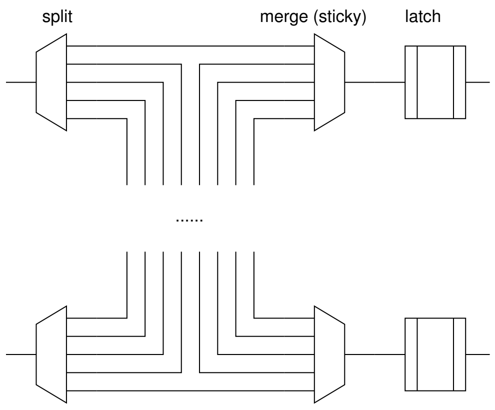


Figure 6.10: The new BE router is a fully connected crossbar with latches on the outputs.

The router implements an absolute XY-routing mechanism. This is choosen over the alternatives because it is relative simple to implement and allows for constructing a reverse path from the forward path. The principle of figure 6.9(b) is used here, where opposite port numbers are each others complements. This choice implies the topology, which has to be a regular mesh, which is the downside of this implementation. The implementation is documented in appendix A.4 and is a behavioural model. It has been implemented as a behavioural model because it lifts the abstraction level away from a standard cell implementation and shows that asynchronous timing can easily be modeled and simulated toghether with components implemented in standard cells.

CHAPTER 7

Testing

The implementations have been tested to ensure that the system works as specified. This chapter describes the test enviroment create for this project, which is depicted in figure 7.1. A description of how tests have been carried out and how the test traces should be intepreted is also described in the following.

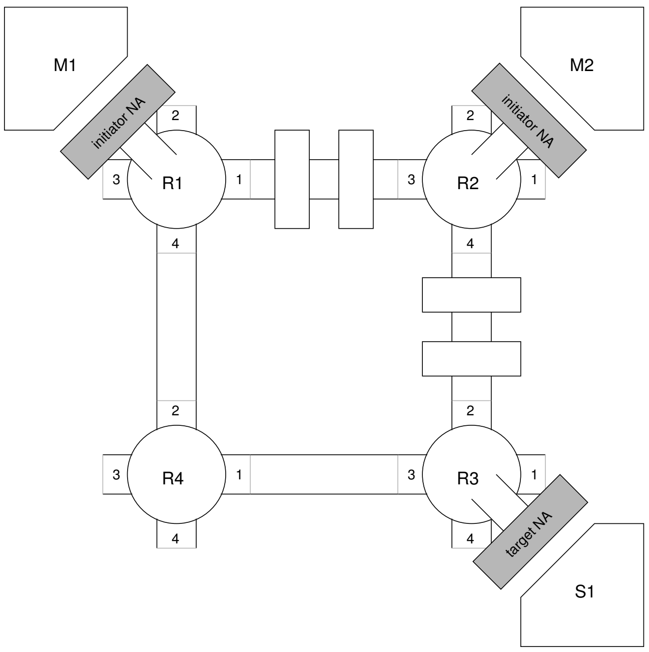


Figure 7.1: The test setup consists of four routers connected in a 2×2 mesh net, two initiator and one target NA, and three IP cores 2 master and one slave.

The test enviroment consists of three IP cores, two masters and one slave and the source code for the network and the system can be found in appendix C.5 and

C.6 respectively. The IP cores are connected through the appropriate NA to three network ports, the source code for the cores can be found in appendix C.7 and C.8. The network consist of four routers with pipelined links in between them, it should be noted that router 4 is connected directly to the neighbouring routers. It is possible to have any number of pipeline pairs, where a pipeline pair consists of two pipeline stages which are have opposite inverted inputs and outputs.

The test vectors are applied from through the master IP cores, which reads in a test file, which consists of OCP commands [30] in the following format:

```
MCmd␣MConnID␣MThreadID␣MAddr␣MData
```

The test file is generated by *mangofy*, which is a script for MANGO that creates OCP commands from a specification of the configuration. The format of the specification is described in [8]. The format has been extended to support remote configuration and acknowledgements as it has been implemented in this project, the new *mangofy* commands is summarised in B.1. BE paths, virtual circuits incl. end-points and general OCP commands can be specified with *mangofy*.

Below here is two tests described, the first shows one master programming BE routes in another master's NA. The second test shows a master that configures a GS connection for another master, reconfigures the network whereby a specific VC is reused. The wave traces from these two tests can be found in appendix B The following section 7.1 examplifies how the wave traces should be read. In addition to these tests a test bench have been created for the BE router which can be found in the description of the BE router implementation in appendix A.4.

Configuring BE Paths Remotely

- Master 2 writes BE paths to master 1's NA.
- Master 1 issues read and write commands to the slave.

The wave trace can be found in appendix B.3.

Reuse of a VC and remote circuit setup

- Master 2 configures a GS connection from master 1 to the slave on a path over router 2.
- Master 1 issues reads and writes on the GS connection, which leaves the VC ctrl wire at different state than after it has been initialised.
- Master 2 reconfigures the netowrk by creating a connection from itself to the slave reusing the link from router 2 to router 3.
- Master 2 sets up a connection from master 1 to the slave over router 4.

- Master 2 writes on the GS connection.
- Master 1 writes on the GS connection

Wave traces resulting from the test can be found in appendix B.4.

7.1 Test Example

Wave traces of signals can be extracted from simulation tools such as Modelsim from Mentor Graphics, figure 7.2 shows such a wave trace. The figure has has some points in time marked which are of interest. Following is a description of these points.

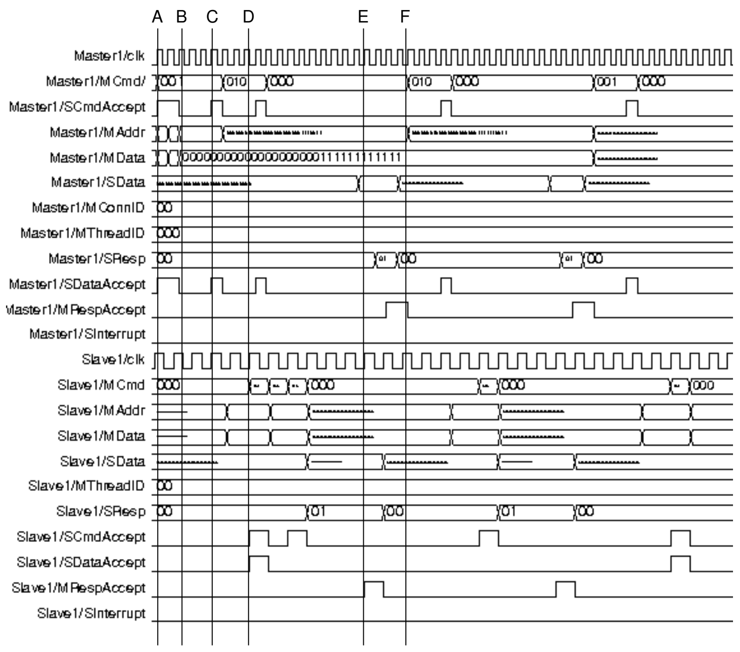


Figure 7.2: A demonstration of the test output.

A After the initiale reset a BE path is written to the NA. A BE path is written using two writes, one for the actual path and one for the lookup address. It is followed by a write to the slave and the NA lowers SCmdAccept while it transmits the flits.

B SCmdAccept is raised as the previous write has been transmitted. The master then initialises a read targetted for the slave. The master idles as it waits for the response.

C The read arrives at the slave core and the response is generated as SResp is set to 01 (Data Valid). The slave waits for the flits to be transmitted.

D The MRespAccept is set high by the target NA as the flits are transmitted.

E A new read is initiated as the SResp and SData indicates that the response cycle is completed.

CHAPTER 8

Discussion

In this chapter there is a discussion of the general status in NoC design, especially regarding connection-oriented solutions. Followed by a discussion of the current status of MANGO and in which directions MANGO and this project could be taken.

8.1 NoC Status

NoC design and research have seen a increasing interest, well illustrated by the NoCs that are included in the study and the many more that has been omitted. Few NoC articles concern the use of NoCs in real world applications. Æthereal has been documented in apllications [3] [33], in both articles the cost of the system is higher in terms of power and area. It has proven diffucult to find suitable applications where the NoC approach is a better aproach than the old implementation. Replacing existing interconnects with a NoC does not make use of the added features and flexibility offer by NoC, therefore suitable platforms and applications where the added flexibilty can be used are needed.

One application area where the added flexibilty can be taken advantage of might be general DSP implementations. A MPSoC with multiple programmable DSPs connected by a NoC would offer a very large degree of flexibility that could be used in DSP applications such as hearing aids and mobile phones.

Future guaranteed service connections in NoCs should focuse on one of two directions: Apply use cases to schedule connections to maximise performance and minimise network ressources or adaptive creation of connections based on request from the application. The two approaches differs greatly, where the scheduling approach is fully deterministic the adaptive approach creates connections as needed.

In systems where the applications traffic patterns can be determined at design time, using use cases to map the connections in the network is a very good solution. Because everything can be scheduled synchronous NoCs that employ time division multiplexing can minimise buffers to keep the router area low. In a NoC like

MANGO it is harder to control the exact flow in all the routers, because the global synchronisation is not available. For small systems where the input is regular - sampled - traffic patterns can be easily determined and multiple use cases can be used to capture different system modes.

In systems where the input can not be determined fully the system must have more intelligence to cope with a non-deterministic behaviour. The contrast to the above mentioned scheduling approach is an adaptive approach where - in the extreme case - the NoC is completly hidden from the application which then just requests a connection to a ressource. To elaborate, the application does not know what kind of intercconect the system employs but simply request a connection and the NoC then creates the connections as needed and where there are available ressources. It is obvious that - as with SoCBUS - giving hard guarantees can be diffucult because it can not be ensured that the connection can be established.

The two approaches differ greatly especially from an application point-of-view, where the use case approach requires complex scheduling, the adaptive approach raises the abstraction level that the application designer can work on. The NoC design will also vary as the adaptive approach requires complex network ressources that can create connections ad-hoc, where the use case approach can be implemented with simple routers that just route. It is obvious that two such different approaces will be suitable for very different systems, which shows how flexible the NoC approach is in solving the interconnect challenge.

8.2 The Status of MANGO

As stated earlier MANGO offers a very flexible platform for testing implementations of NoC research. The modular approach taken with MANGO makes it possible to change approaches to core functionality such as it has been illustrated with the BE router. MANGO could therefore prove to be platform for testing NoC research topics in an asynchronou enviroment. To make full use of the possibilities in MANGO the implementation details should abstracted allowing for a faster exploration of different solutions.

The standard cell implementation plays an important role as it proves that a core concept, such as given hard guarantees in an asynchronous enviroment is possible. Because of this it is possible to build upon MANGO and reliably explore advanced NoC research topics.

The implementation done in this project consists of several parts: Reusable VCs, remote NA programming, acknowledge on configuration, a new BE router and generation of configuration files.

Moving the key component in the VC control is the best way to solve the problem of reusing the VCs. It takes away the need for tearing down connections, however as it will be discussed in the following section on future work, more signaling including tearing down should be implemented.

Allowing any initiator to program any NA - target and initiator alike - adds great flexibility to the overall solution and enables different configuration approaches to be taken, such as central and distributed configurations.

Adding acknowledgements to the configuration reguest increases the information available, but it also increases the setup time. The implementation of acknowledgement in the target NA is done so that it is not returned until the configuration has been done, where in the router the acknowledge is generated just prior to the configuration being committed.

The BE router implements an XY-routing scheme where the return path is generated from the forward path, this makes the router dependent on a mesh topology.

8.3 Future Work

This project has moved MANGO further, but many areas can still be investigated on, both areas touched in this project aswell as in the original MANGO implementation. Tobias Bjerregaard has suggested improvements in [8], and the reader is directed there for the specific topics. This section will only concern future work based on this work.

Signaling, both during setup and tear down is needed to enhance the information about the state of the NoC. A signal - maybe in form of an interrupt - is needed to notify processes in the system of conclusion of a configuration phase or a succesfull configuration of a specific connection. Tearing down connection is needed to have a view of available connections in the network.

The **Network Adapters**, must be rethinked to add more flexibility to the system. It should be considered to create a generic router design, which would allow similar features to be added in the initiator and target NA. Currently cores that needs both a master and a slave interface must connect to two different NAs which again connects to two different routers. Creating an NA where both master and slave features could be implemented would enable a more flexible approach.

The new **BE router** is too limiting because it adds an unnecessary constraint on the system, namely requiring a mesh topology. A different implementation of the BE router core should make use of a different routing scheme, creating two virtual networks for response and request could be an option. It has not been implemented in this project as a matter of choice.

CHAPTER 9

Conclusion

This work has presented an implementation of reconfigurable guaranteed service connections within the MANGO NoC, furthermore a study of current NoC designs has been documented.

The virtual circuits used by MANGO to implement guaranteed services has been investigated and a solution that allows for reconfiguration of end-to-end circuits and reuse of shared ressources has been developed. The implementation consists of a number of parts which together applies an increased flexibility to the virtual circuit implementation adopted by MANGO. The main parts of the solution is shortly presented here, allowing reuse of the VCs, programming NAs, acknowledgement on setup and a crossbar based BE router. The generation of setup files have been upgraded to support all the added features, which makes using and testing MANGO much easier. The solution presented succesfully solves the goal of this project namely to provide the MANGO NOC with features to allow reconfiguration and reuse of virtual circuits.

It has been the aim to provide a flexible solution that allows for a multitude of mechanisms to setup and reconfigure the virtual circuits. It is therefore possible to use diffenerent programming approaches, such as a distributed configuration and a centrally controlled configuration scheme. The flexibility could be increased further if signalling was implemented to notify the processes of a succesfull completion of the configuration phase.

The study that has been conducted in this project has shown a great variety in the approaches to provide guaranteed service in SoC interconnects. Increasing the service level tends to increase the complexity which affects the cost of the interconnect, it is therefore important to choose the right level of service for the application to maximise performance.

Bibliography

[1] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, 2002.

[2] W. J. Bainbridge, A. Bardsley, and R. W. McGuffin. System-on-chip design using self-time networks-on-chip.

[3] Chris Bartels, Jos Huisken, Kees Goossens, Patrick Groeneveld, and Jef van Meerbergen. Comparison of an Æthereal network on chip and a traditional interconnect for a multi-processor DVB-T system on chip. In *Proc. IFIP Int'l Conference on Very Large Scale Integration (VLSI-SoC)*, October 2006.

[4] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[5] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. Supervised by Assoc. Prof. Jens Sparsø, IMM.

[6] T. Bjerregaard, S. Mahadevan, R. G. Olsen, and J. Sparsø. An OCP compliant network adapter for GALS-based soc design using the MANGO network-on-chip. In *Proceedings of the International Symposium on System-on-Chip (SoC'05)*, pages 171–174. IEEE, nov 2005.

[7] T. Bjerregaard and J. Sparsø. Implementation of guaranteed services in the MANGO clockless network-on-chip. *IEE Proceedings: Computing and Digital Techniques*, 2006. Accepted for publication.

[8] Tobias Bjerregaard. *Do you MANGO?*, 2006. Internal technical document.

[9] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1, 2006.

[10] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. Qnoc: Qos architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2-3):105–128, 2004.

[11] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture, A Hardware/Software Approach.* Morgan Kaufmann, 1999.

[12] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. *Design Automation Conference, 2001. Proceedings*, pages 684–689, 2001.

[13] John Dielissen, Andrei Rădulescu, Kees Goossens, and Edwin Rijpkema. Concepts and implementation of the Philips network-on-chip. In *IP-Based SOC Design*, November 2003.

[14] S. Dutta, R. Jensen, and A. Rieckmann. Viper: a multiprocessor soc for advanced set-top box and digital tv systems. *IEEE Design & Test of Computers*, 18(5):21–31, 2001.

[15] U. Feige and P. Raghavan. Exact analysis of hot-potato routing. *Foundations of Computer Science, 1992. Proceedings., 33rd Annual Symposium on*, pages 553 –562, 1992.

[16] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22(5), Sept-Oct 2005.

[17] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 250 –256, 2000.

[18] Silistix http://www.silistix.com/, 2006.

[19] Antoine Jalabert, Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Xpipescompiler: A tool for instantiating application specific networks on chip. *Proceedings - Design, Automation and Test in Europe Conference and Exhibition, DATE 04 and Proceedings - Design, Automation and Test in Europe Conference and Exhibition*, 2:884–889, 2004.

[20] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, 2005.

[21] F. Karim, Anh Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22(5):36–45, 2002.

[22] Zhonghai Lu, Bei Yin, and A. Jantsch. Connection-oriented multicasting in wormhole-switched networks on chip. *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, 00:205–2110, 2006.

[23] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. The nostrum backbone - a communication protocol stack for networks on chip.

[24] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. *Proceedings - Design, Automation and Test in Europe Conference and Exhibition, DATE 04 and Proceedings - Design, Automation and Test in Europe Conference and Exhibition*, 2:890–895, 2004.

[25] F. Moraes, N. Calazans, A. Mello, L. Moller, and L. Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, 2004.

[26] Srinivasan Murali, Martijn Coenen, Andrei Rădulescu, Kees Goossens, and Giovanni De Micheli. Mapping and configuration methods for multi-use-case networks on chips. In *Proc. Design Automation Conference. Asia and South Pacific (ASP-DAC)*, January 2006.

[27] Srinivasan Murali, Martijn Coenen, Andrei Rădulescu, Kees Goossens, and Giovanni De Micheli. A methodology for mapping multiple use-cases on to networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2006.

[28] E. Nilsson, M. Millberg, J. Oberg, and A. Jantsch. Load distribution with the proximity congestion awareness in a network on chip. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 1126–1127, 2003.

[29] R. G. Olsen. OCP based adapter for network-on-chip. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. Supervised by assoc. prof. Jens Sparsø, IMM.

[30] Release 2.0 Open Core Protocol (OCP) Specification, 2003.

[31] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1):4–17, January 2005.

[32] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Kluwer Academic Publishers, dec 2001.

[33] Frits Steenhof, Harry Duque, Björn Nilsson, Kees Goossens, and Rafael Peset Llopis. Networks on chips for high-end consumer-electronics tv system architectures. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 148–153, March 2006.

[34] D. Wiklund and Dake Liu. Socbus: switched network on chip for hard real time embedded systems. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., 2003.

APPENDIX A

Implementations

The following describes where modifications has been done to implement the solutions described in chapter 6.

A.1 Handshake Conversion

The handshake conversion between the 4 phase unlock signal inside the router to the 4 phase across the link is handled by a an edge triggered flip-flop with a feed back link. A gate diagram of the conversion block called *key* here is shown in figure A.1. The Verilog source code is shown in appendix C.1. The flip-flop changes state every time a handshake is completed on the virtual channel.

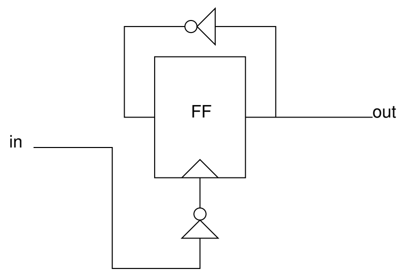


Figure A.1: Gate level diagram of the key.

The key has been moved from the unlock to the output of the ports both the local port and the port to neighbouring routers.

The following files have been modified in connection with the moving the key:

io_port_39bits_8vcs_sidechain.v - The key has been inserted before the output of each VC control wire.

local_io_port_39bits_4chs.v - The key has been inserted before the lock boxes in the local port.

unshace_ctrl_input_decoupled.v - The funcionality of the key has been removed and implemented in the key.

A.2 NA Programming

Allowing the response flow to write into the configuration register which is implemented as a Look-Up table (LUT), has required the implementation af an arbiter that allows either the response or the request flow to write into the LUT. The arbiter code is written in VHDL and the code can be found in C.2. In addition the following files have had modifactions done as part of the implementation.

initiator_req.vhd - The LUT has been moved out of the response flow, thus ports and signals have been adjusted accordingly with the move.

initiator_sync.vhd - The LUT and its associated arbiter has been moved to this file.

initiator_resp.vhd - Ports to allow the response controller to write to the LUT has been added

initiator_resp_ctrl.vhd - A write function similar to the one in the request flow has been implemented.

A.3 Acknowledgement

Acknowldgement on the configuration requests have been implemented. The OCP command used is the Write Non Posted (WRNP), which resembles a read. The initiator NA has therefore had WRNP added to the commands that it can handle.

The following describes the implementation of the acknowledgement in the target NA and router respectively.

A.3.1 Target NA

Acknowledgement from the target are generated in the response path (target_resp_control.vhd), when signaled from request flow. The signal is asserted when the LUT signals that the write configuration has been completed. It is the request flow controller (target_req_control.vhd) that handles the write and therefore asserts the acknowledge signal to the response flow.

The acknowledge is generated in a similar fashion to the interrupt handling. Because the target only handles one request at the time the configuration acknowledge does not affect the overall guarantees. It is further ensured that the overall flow is not affected, by ensuring that the request flow still handles requests in one clock cycle.

A.3.2 Router

As shown in the figure 6.7 on page 57 a acknowledge generator, two latches and a merge has been added. The acknowledge generator is implemented behavioural and the source code is shown in appendix C.3. It has two inputs, an input which is the request signal and the data from the program split to the programming blocks. When a request is sampled the BE path is captured from the data stream and the first flit is emitted on the acknowledge path, on the second request the response flit is generated with a data valid flag set. The last flit of the configuration request is sampled to capture the eop field.

The latches and the merge is the same as is used in the BE router and the source code for those can be found in appendix C.4. All the modules and extra signals have been added to the BE router (BE_XYrouter_39bits.v).

A.4 Best-Effort Router

The BE router has been implemented as a behvioural model and the source code can be found in appendix C.4. As described in section 6.7 the BE rouer is a fully connected crossbar, with latches on the output port. The router is build from 3 different modules, split, merge and the latch, which has been instanced and connected in the BE_5x5_39_behav module.

The split (be_behav_demux) is a demultiplexer that selects one of its output ports depending on the path in the first flit. It further rotates and and inverts the BE path.

The merge is build from 2 input merges that keeps their input choice until an eop field has been seen.

APPENDIX **B**

Tests

B.1 The Mangofy Script

The *mangofy* script has been updated, the following two commands has been changed:

BE_PATH{ *name, BE_lut_tag, global_addr, routing_path, [remote_NA]*}

The remote_NA tag is used if the BE path is to be programmed in a remote NA. The script will then ensure that the configuration requests are sent to that NA. It is required that a BE path to the NA has been defined prior.

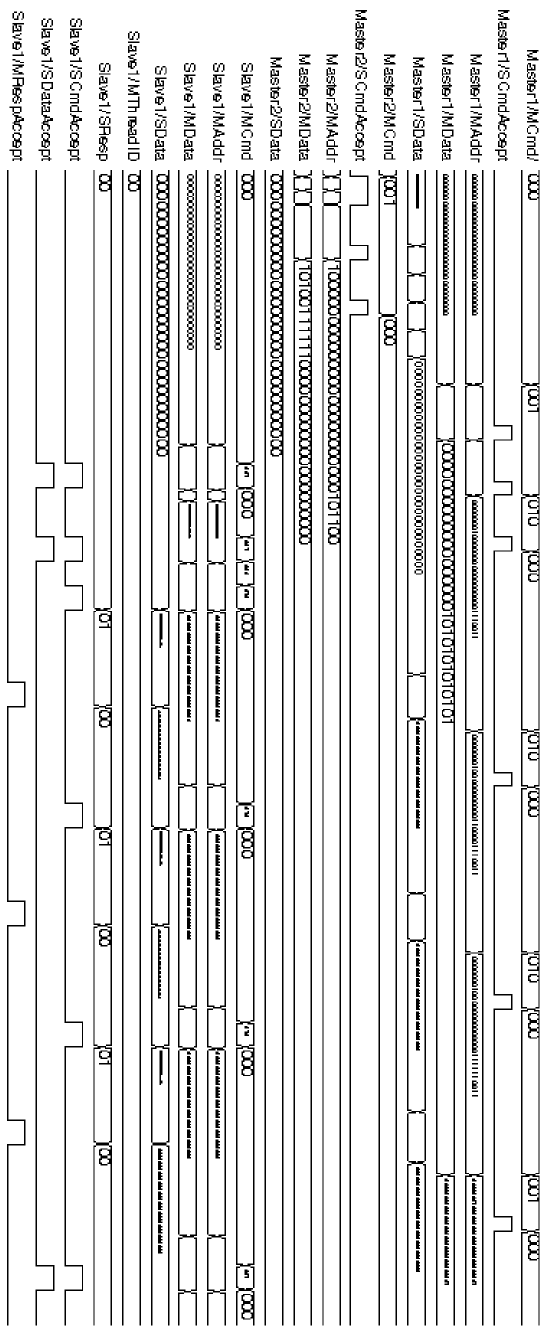
SETUP_SERVICE{ *type, connection_ID, response_connection_ID, response_port/path, master_na, target_na*}

The master_na tag has been added so that remote endpoints can be added. *local* should be used if the master in the connection is the present core otherwise the BE path name should be used.

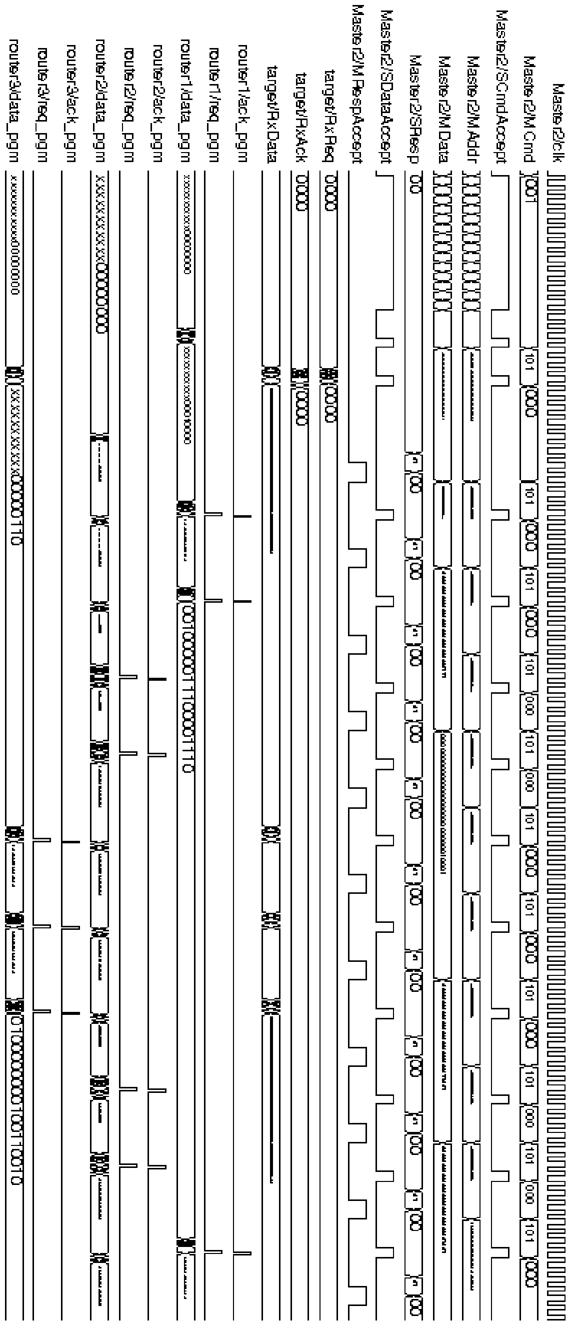
In addition some checks have been removed to allow OCP commands to use BE paths that has not been declared in the scope of the core. It is therefore required that the user is aware that everything has been declared properly.

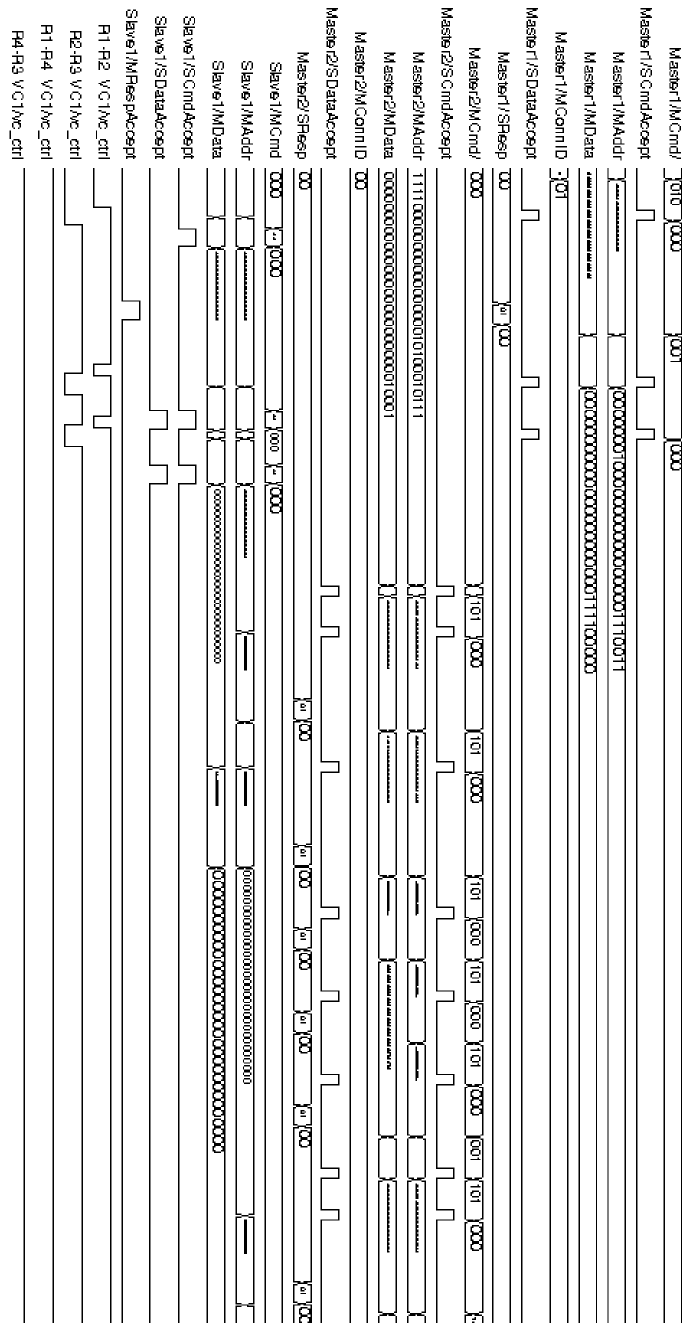
B.2 Wave traces

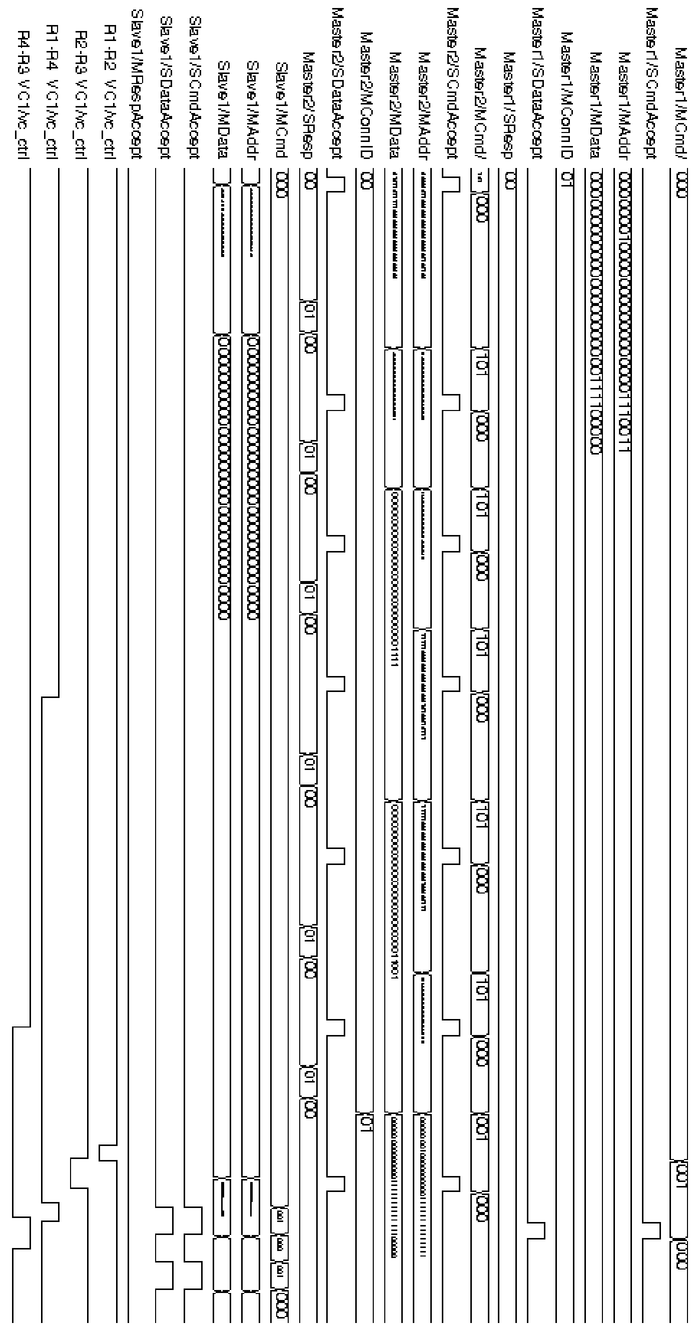
B.3 Configuring BE Paths Remotely



B.4 Reuse of a VC and remote circuit setup




Master1/MCmd/
Master1/SCmdAccept
Master1/MAddr
Master1/MData
Master1/MConnID
Master1/SDataAccept
Master1/SResp
Master2/MCmd/
Master2/SCmdAccept
Master2/MAddr
Master2/MData
Master2/MConnID
Master2/SDataAccept
Master2/SResp
Slave1/MCmd
Slave1/MAddr
Slave1/MData
Slave1/SCmdAccept
Slave1/SDataAccept
Slave1/MRespAccept
R1-R2 VC1/vc_ctrl
R2-R3 VC1/vc_ctrl
R1-R4 VC1/vc_ctrl
R4-R3 VC1/vc_ctrl


Master1/MCmd/
Master1/SCmdAccept
Master1/MAddr
Master1/MData
Master1/MConnID
Master1/SDataAccept
Master1/SResp
Master2/MCmd/
Master2/SCmdAccept
Master2/MAddr
Master2/MData
Master2/MConnID
Master2/SDataAccept
Master2/SResp
Slave1/MCmd
Slave1/MAddr
Slave1/MData
Slave1/SCmdAccept
Slave1/SDataAccept
Slave1/MRespAccept
R1-R2 VC1/vc_ctrl
R2-R3 VC1/vc_ctrl
R1-R4 VC1/vc_ctrl
R4-R3 VC1/vc_ctrl

APPENDIX C

Source Code

C.1 key.v

```
//——————————————————————————————————
// Verilog netlist for the key.

// The key is actually a 4ph to 2ph converter, consisting of a
// flip-flop with an inverter on the clock, so that we register the
// downgoing transistions in the 4-phase communication. And there's a feedback
// loop on the flip-flop.

// Christian Place Pedersen 2006
//——————————————————————————————————
`timescale 1ns/1ps
module key(
   reset,
         ack_out,
         g_ack_out
);

input reset, ack_out;
output g_ack_out;

IVHS inv_rst( .Z( n_reset ), .A( reset ) );

FD4QHS g_box( .Q( g_ack_out ), .D( feedback_g_ack ), .CP( n_ack_out ), .SD( n_reset ) );
IVHS feedback_inv_key( .Z(feedback_g_ack), .A(g_ack_out) );
IVHS cp_inv( .Z( n_ack_out ), .A( ack_out ) );

endmodule
```

C.2 lut_arb.vhd

```
-------------------------------------------------------------------------------
-- Title      : LUT dual write ports
-- Project    :
-------------------------------------------------------------------------------
-- File       : lut_arb.vhd
-- Author     : Christian Place Pedersen
-- Company    : Technical University of Denmark - IMM/CSE
-- Created    : 2005-08-28
-- Last update: 2006/08/30
-- Platform   :
-- Standard   : VHDL'93
-------------------------------------------------------------------------------
-- Description:
-------------------------------------------------------------------------------
-- Copyright (c) 2006
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2006-08-28  1.0      CPP     Created
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity lut_arb is

  generic (
    local_addr_wdth : integer := 24;
    data_wdth       : integer := 32);

  port (
    reset_n  : in  std_logic;
    write0_i : in  std_logic;
    data0_i  : in  std_logic_vector(data_wdth-1 downto 0);
    addr0_i  : in  std_logic_vector(local_addr_wdth-1 downto 0);
    done0_o  : out std_logic;
    write1_i : in  std_logic;
    data1_i  : in  std_logic_vector(data_wdth-1 downto 0);
    addr1_i  : in  std_logic_vector(local_addr_wdth-1 downto 0);
    done1_o  : out std_logic;
    write_o  : out std_logic;
    data_o   : out std_logic_vector(data_wdth-1 downto 0);
    addr_o   : out std_logic_vector(local_addr_wdth-1 downto 0);
    done_i   : in  std_logic
    );

end lut_arb;

architecture behav of lut_arb is

  signal last : std_logic;

begin  -- behav

  process (reset_n, write0_i, data0_i, addr0_i, write1_i,
           data1_i, addr1_i, done_i, last)
  begin  -- process
    if reset_n = '0' then
      last <= '1';
    else
      if last = '0' then
        if write1_i = '1' then
          write_o <= write1_i;
          data_o <= data1_i;
          addr_o <= addr1_i;
          done1_o <= done_i;
          last <= '1';
        else
          write_o <= write0_i;
          data_o <= data0_i;
          addr_o <= addr0_i;
          done0_o <= done_i;
```

```
            last <= '0';
          end if;
        else
          if write0_i = '1' then
            write_o <= write0_i;
            data_o <= data0_i;
            addr_o <= addr0_i;
            done1_o <= done_i;
            last <= '0';
          else
            write_o <= write1_i;
            data_o <= data1_i;
            addr_o <= addr1_i;
            done1_o <= done_i;
            last <= '1';
          end if;
        end if;
      end if;
  end process;

end behav;
```

C.3 gen_acknowledge_39.v

```
module gen_acknowledge_39(
  req_in,
  data_in,
  req_out,
  ack_out,
  data_out,
  reset
);

input  req_in;
input  [38:0] data_in;
output req_out;
input  ack_out;
output [38:0] data_out;
input reset;

reg req_out;
reg [38:0] data_out;
reg eop, gen;

parameter DLY = 0.4;

not(n_reset, reset);

always @(n_reset) begin
  if (n_reset == 1'b1) begin
    req_out <= 1'b0;
    data_out <= 38'b0;
    eop <= 1'b1;
    gen <= 1'b0;
  end
end

always begin
  wait (req_in == 1'b1);
  if (eop) begin
    data_out = {data_in[38], data_in[5], data_in[6],
                data_in[7], data_in[8], data_in[9], data_in[10],
                data_in[11], data_in[12], data_in[12], data_in[14],
                data_in[15], data_in[16], data_in[17], data_in[18],
                data_in[19], data_in[20], data_in[21], data_in[22],
                data_in[23], data_in[24], data_in[25], data_in[26],
                data_in[27], data_in[28], data_in[29], data_in[30],
                data_in[31], data_in[32], data_in[33], data_in[34],
                8'b00000000};
    gen = 1;
    #DLY;
    req_out = 1'b1;
    wait (ack_out == 1'b1);
    #DLY;
    req_out = 1'b0;
    wait (ack_out == 1'b0);
  end
  else if (gen) begin
    data_out = {2'b10, data_in[33:31],2'b01,32'b0}; // data valid response
    gen = 0;
    #DLY;
    req_out = 1'b1;
    wait (ack_out == 1'b1);
    #DLY;
    req_out = 1'b0;
    wait (ack_out == 1'b0);
  end
  wait (req_in == 1'b0);
end

always begin
  wait (req_in == 1'b1);
  wait (req_in == 1'b0);
  eop = data_in[38];
```

```
end
endmodule
```

C.4 BE_behav.v

```
// CPP
`timescale 1ns/1ps
module BE_5x5_39_behav(
    req_in,
    ack_in,
    data_in4,
    data_in3,
    data_in2,
    data_in1,
    data_in0,
    req_out,
    ack_out,
    data_out4,
    data_out3,
    data_out2,
    data_out1,
    data_out0,
    rst
);


// ports
input   rst;

input   [4:0]   req_in;
output  [4:0]   ack_in;
input   [38:0]  data_in0;
input   [38:0]  data_in1;
input   [38:0]  data_in2;
input   [38:0]  data_in3;
input   [38:0]  data_in4;
output  [4:0]   req_out;
input   [4:0]   ack_out;
output  [38:0]  data_out0;
output  [38:0]  data_out1;
output  [38:0]  data_out2;
output  [38:0]  data_out3;
output  [38:0]  data_out4;

wire [4:0] req_dmux0, req_dmux1, req_dmux2, req_dmux3, req_dmux4;
wire       req_merge0, req_merge1, req_merge2, req_merge3, req_merge4;
wire [4:0] select0, select1, select2, select3, select4;
wire [4:0] ack_dmux0, ack_dmux1, ack_dmux2, ack_dmux3, ack_dmux4;
wire       ack_merge0, ack_merge1, ack_merge2, ack_merge3, ack_merge4;
wire [38:0] data_dmux00, data_dmux01, data_dmux02, data_dmux03, data_dmux04,
            data_dmux10, data_dmux11, data_dmux12, data_dmux13, data_dmux14,
            data_dmux20, data_dmux21, data_dmux22, data_dmux23, data_dmux24,
            data_dmux30, data_dmux31, data_dmux32, data_dmux33, data_dmux34,
            data_dmux40, data_dmux41, data_dmux42, data_dmux43, data_dmux44;
wire [38:0] data_merge0, data_merge1, data_merge2, data_merge3, data_merge4;
//wire [38:0] data_latch0, data_latch1, data_latch2, data_latch3, data_latch4;


//assign data_out0 = data_latch0;
//assign data_out1 = data_latch1;
//assign data_out2 = data_latch2;
//assign data_out3 = data_latch3;
//assign data_out4 = data_latch4;

//Demux
// input port 0
BE_behav_demux input0(
  .req_in(req_in[0]),
  .ack_in(ack_in[0]),
  .data_in(data_in0),
  .req_out(req_dmux0),
  .ack_out(ack_dmux0),
  .data_out0(data_dmux00),
  .data_out1(data_dmux01),
  .data_out2(data_dmux02),
  .data_out3(data_dmux03),
  .data_out4(data_dmux04),
  .rst(rst)
);
// input port 1
BE_behav_demux input1(
  .req_in(req_in[1]),
  .ack_in(ack_in[1]),
  .data_in(data_in1),
  .req_out(req_dmux1),
  .ack_out(ack_dmux1),
  .data_out0(data_dmux10),
  .data_out1(data_dmux11),
  .data_out2(data_dmux12),
  .data_out3(data_dmux13),
  .data_out4(data_dmux14),
  .rst(rst)
);
// input port 2
BE_behav_demux input2(
  .req_in(req_in[2]),
  .ack_in(ack_in[2]),
  .data_in(data_in2),
  .req_out(req_dmux2),
  .ack_out(ack_dmux2),
  .data_out0(data_dmux20),
  .data_out1(data_dmux21),
  .data_out2(data_dmux22),
  .data_out3(data_dmux23),
  .data_out4(data_dmux24),
  .rst(rst)
);
// input port 3
BE_behav_demux input3(
  .req_in(req_in[3]),
  .ack_in(ack_in[3]),
  .data_in(data_in3),
  .req_out(req_dmux3),
  .ack_out(ack_dmux3),
  .data_out0(data_dmux30),
  .data_out1(data_dmux31),
  .data_out2(data_dmux32),
  .data_out3(data_dmux33),
  .data_out4(data_dmux34),
  .rst(rst)
);
// input port 4
BE_behav_demux input4(
  .req_in(req_in[4]),
  .ack_in(ack_in[4]),
  .data_in(data_in4),
  .req_out(req_dmux4),
  .ack_out(ack_dmux4),
  .data_out0(data_dmux40),
  .data_out1(data_dmux41),
  .data_out2(data_dmux42),
  .data_out3(data_dmux43),
  .data_out4(data_dmux44),
  .rst(rst)
);

// Merge
// output port 0
BE_behav_merge merge0(
  .req_in({req_dmux4[0],req_dmux3[0],req_dmux2[0], req_dmux1[0],req_dmux0[0]}),
  .ack_in({ack_dmux4[0],ack_dmux3[0],ack_dmux2[0], ack_dmux1[0],ack_dmux0[0]}),
  .data_in0(data_dmux00),
  .data_in1(data_dmux10),
  .data_in2(data_dmux20),
  .data_in3(data_dmux30),
  .data_in4(data_dmux40),
  .req_out(req_merge0),
  .ack_out(ack_merge0),
  .data_out(data_merge0),
  .rst(rst)
```

```
);
// output port 1
BE_behav_merge merge1(
  .req_in({req_dmux4[1],req_dmux3[1],req_dmux2[1], req_dmux1[1],req_dmux0[1]}),
  .ack_in({ack_dmux4[1],ack_dmux3[1],ack_dmux2[1], ack_dmux1[1],ack_dmux0[1]}),
  .data_in0(data_dmux01),
  .data_in1(data_dmux11),
  .data_in2(data_dmux21),
  .data_in3(data_dmux31),
  .data_in4(data_dmux41),
  .req_out(req_merge1),
  .ack_out(ack_merge1),
  .data_out(data_merge1),
  .rst(rst)
);
// output port 2
BE_behav_merge merge2(
  .req_in({req_dmux4[2],req_dmux3[2],req_dmux2[2], req_dmux1[2],req_dmux0[2]}),
  .ack_in({ack_dmux4[2],ack_dmux3[2],ack_dmux2[2], ack_dmux1[2],ack_dmux0[2]}),
  .data_in0(data_dmux02),
  .data_in1(data_dmux12),
  .data_in2(data_dmux22),
  .data_in3(data_dmux32),
  .data_in4(data_dmux42),
  .req_out(req_merge2),
  .ack_out(ack_merge2),
  .data_out(data_merge2),
  .rst(rst)
);
// output port 3
BE_behav_merge merge3(
  .req_in({req_dmux4[3],req_dmux3[3],req_dmux2[3], req_dmux1[3],req_dmux0[3]}),
  .ack_in({ack_dmux4[3],ack_dmux3[3],ack_dmux2[3], ack_dmux1[3],ack_dmux0[3]}),
  .data_in0(data_dmux03),
  .data_in1(data_dmux13),
  .data_in2(data_dmux23),
  .data_in3(data_dmux33),
  .data_in4(data_dmux43),
  .req_out(req_merge3),
  .ack_out(ack_merge3),
  .data_out(data_merge3),
  .rst(rst)
);
// output port 4
BE_behav_merge merge4(
  .req_in({req_dmux4[4],req_dmux3[4],req_dmux2[4], req_dmux1[4],req_dmux0[4]}),
  .ack_in({ack_dmux4[4],ack_dmux3[4],ack_dmux2[4], ack_dmux1[4],ack_dmux0[4]}),
  .data_in0(data_dmux04),
  .data_in1(data_dmux14),
  .data_in2(data_dmux24),
  .data_in3(data_dmux34),
  .data_in4(data_dmux44),
  .req_out(req_merge4),
  .ack_out(ack_merge4),
  .data_out(data_merge4),
  .rst(rst)
);

// Latch
//output port 0
BE_behav_latch latch0(
  .req_in(req_merge0),
  .ack_in(ack_merge0),
  .data_in(data_merge0),
  .req_out(req_out[0]),
  .ack_out(ack_out[0]),
  .data_out(data_out0),
  .rst(rst)
);
//output port 1
BE_behav_latch latch1(
  .req_in(req_merge1),
  .ack_in(ack_merge1),
  .data_in(data_merge1),
  .req_out(req_out[1]),
  .ack_out(ack_out[1]),
  .data_out(data_out1),
  .rst(rst)
);
//output port 2
BE_behav_latch latch2(
  .req_in(req_merge2),
  .ack_in(ack_merge2),
  .data_in(data_merge2),
  .req_out(req_out[2]),
  .ack_out(ack_out[2]),
  .data_out(data_out2),
  .rst(rst)
);
//output port 3
BE_behav_latch latch3(
  .req_in(req_merge3),
  .ack_in(ack_merge3),
  .data_in(data_merge3),
  .req_out(req_out[3]),
  .ack_out(ack_out[3]),
  .data_out(data_out3),
  .rst(rst)
);
//output port 4
BE_behav_latch latch4(
  .req_in(req_merge4),
  .ack_in(ack_merge4),
  .data_in(data_merge4),
  .req_out(req_out[4]),
  .ack_out(ack_out[4]),
  .data_out(data_out4),
  .rst(rst)
);

endmodule


// 2 input(data, req and ack) merge
module BE_behav_2merge (
  req_in,
  ack_in,
  data_in0,
  data_in1,
  req_out,
  ack_out,
  data_out,
  rst
);

input  [1:0] req_in;
output [1:0] ack_in;
input  [38:0] data_in0;
input  [38:0] data_in1;
output req_out;
input  ack_out;
output [38:0] data_out;
input  rst;

parameter DLY = 0.4;

reg [38:0] data_out;
reg [1:0] ack_in;
reg req_out, int_req_out;
reg last;
reg eop;
wire ev_req;

always @(rst) begin
  if (rst == 1'b1) begin
    last <= 1'b0;
    eop <= 1'b1;
    ack_in <= 1'b0;
  end
end

or(ev_req , req_in[0], req_in[1]);

always begin
wait (ev_req == 1'b1);
if (eop == 1'b1) begin
  if (last) begin // req_in[1] was selected last
    if (req_in[0] == 1'b1) begin
      last <= 1'b0;
    end
    else if (req_in[1] == 1'b1) begin
      last <= 1'b1;
    end
  end
  else begin
    if (req_in[1] == 1'b1) begin
      last <= 1'b1;
    end
    else if (req_in[0] == 1'b1) begin
      last <= 1'b0;
    end
  end
end
wait (ack_out == 1'b1);
eop = (last ? data_in1[38] : data_in0[38]);
#DLY;
if (last) begin
  ack_in = 2'b10;
  wait (req_in[1] == 1'b0);
end
else begin
  ack_in = 2'b01;
  wait (req_in[0] == 1'b0);
end
wait (ack_out == 1'b0);
ack_in <= 2'b00;
end

always @(data_in0 or data_in1 or last or req_in) begin
  if (last) begin
    data_out <= data_in1;
    req_out <= req_in[1];
  end
  else begin
    data_out <= data_in0;
    req_out <= req_in[0];
  end
end

//always @(int_req_out) begin // delay req_out. needed for the pgm_split
//  req_out = #1 int_req_out;
//end

endmodule

// Selects based on bit 37:35 of data_in.
// rotates and inverts, to form the return path
module BE_behav_demux(
  req_in,
  ack_in,
  data_in,
  req_out,
  ack_out,
  data_out0,
  data_out1,
  data_out2,
  data_out3,
  data_out4,
  rst
);

input  req_in;
```

```
output ack_in;
input  [38:0] data_in;
output [4:0] req_out;
input  [4:0] ack_out;
output [38:0] data_out0;
output [38:0] data_out1;
output [38:0] data_out2;
output [38:0] data_out3;
output [38:0] data_out4;
input  rst;

parameter DLY = 0.2;

reg  [38:0] data_out0, data_out1, data_out2, data_out3, data_out4, data;
reg  [4:0] req_out;
reg  ack_in, int_req_out;
wire ev_ack;
reg  [2:0] sel;
reg  eop;
reg  [3:0] count;

always @(rst) begin
  if (rst == 1'b1) begin
    int_req_out <= 1'b0;
    req_out <= 5'b0000;
    ack_in <= 1'b0;
    sel <= 3'b000;
    eop <= 1'b1;
    count <= 4'b0;
  end
end

or(ev_ack, ack_out[0], ack_out[1], ack_out[2], ack_out[3], ack_out[4]);

always begin
  wait (req_in == 1'b1);
  if (eop == 1'b1) begin
    sel <= data_in[37:35];
    count <= data_in[3:0]; // hop count
  end
  #DLY;
  int_req_out <= 1'b1;
  wait (ev_ack == 1'b1);
  #DLY;
  int_req_out <= 1'b0;
  eop <= data_in[38];
  wait (ev_ack == 1'b0);
  ack_in <= 1'b1;
  wait (req_in == 1'b0);
  ack_in <= 1'b0;

end

always @(eop or data_in or count) begin
  if (eop == 1'b1) begin
    data[38] <= data_in[38];
    data[3:0] <= data_in[3:0]+1;
    case (count)
    0: begin
      data[37:7] <= data_in[34:4];
      data[6:4] <= ~{data_in[35], data_in[36], data_in[37]};
    end
    1: begin
      data[37:10] <= data_in[34:7];
      data[9:7] <=  ~{data_in[35], data_in[36], data_in[37]};
      data[6:4] <= data_in[6:4];
    end
    2: begin
      data[37:13] <= data_in[34:10];
      data[12:10] <=  ~{data_in[35], data_in[36], data_in[37]};
      data[9:4] <= data_in[9:4];
    end
    3: begin
      data[37:16] <= data_in[34:13];
```

```
        data[15:13] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[12:4] <= data_in[12:4];
      end
      4: begin
        data[37:19] <= data_in[34:16];
        data[18:16] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[15:4] <= data_in[15:4];
      end
      5: begin
        data[37:22] <= data_in[34:19];
        data[21:19] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[18:4] <= data_in[18:4];
      end
      6: begin
        data[37:25] <= data_in[34:22];
        data[24:22] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[21:4] <= data_in[21:4];
      end
      7: begin
        data[37:28] <= data_in[34:25];
        data[27:25] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[24:4] <= data_in[24:4];
      end
      8: begin
        data[37:31] <= data_in[34:28];
        data[30:28] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[27:4] <= data_in[27:4];
      end
      9: begin
        data[37:34] <= data_in[34:31];
        data[33:31] <=  ~{data_in[35], data_in[36], data_in[37]};
        data[30:4] <= data_in[30:4];
      end
      default: begin
        data <= data_in;
      end
      endcase
    end
    else begin
      data <= data_in;
    end
end

always @(sel, int_req_out, data) begin
  case(sel)
  5'b001:
   begin
     data_out1 <= data;
     req_out[1] <= int_req_out;
   end
  5'b010:
   begin
     data_out2 <= data;
     req_out[2] <= int_req_out;
   end
  5'b110:
   begin
     data_out3 <= data;
     req_out[3] <= int_req_out;
   end
  5'b101:
   begin
     data_out4 <= data;
     req_out[4] <= int_req_out;
   end
  default:
   begin
     data_out0 <= data;
     req_out[0] <= int_req_out;
   end
  endcase
end
endmodule
```

```
module BE_behav_merge(
  req_in,
  ack_in,
  data_in0,
  data_in1,
  data_in2,
  data_in3,
  data_in4,
  req_out,
  ack_out,
  data_out,
  rst
);

input  [4:0] req_in;
output [4:0] ack_in;
input  [38:0] data_in0;
input  [38:0] data_in1;
input  [38:0] data_in2;
input  [38:0] data_in3;
input  [38:0] data_in4;
output req_out;
input  ack_out;
output [38:0] data_out;
input  rst;

wire req0_0, req0_1, req1_0;
wire [38:0] data0_0, data0_1, data1_0;
wire ack0_0, ack0_1, ack1_0;

BE_behav_2merge merge0_0(
  .req_in(req_in[1:0]),
  .ack_in(ack_in[1:0]),
  .data_in0(data_in0),
  .data_in1(data_in1),
  .req_out(req0_0),
  .ack_out(ack0_0),
  .data_out(data0_0),
  .rst(rst)
);
BE_behav_2merge merge0_1(
  .req_in(req_in[3:2]),
  .ack_in(ack_in[3:2]),
  .data_in0(data_in2),
  .data_in1(data_in3),
  .req_out(req0_1),
  .ack_out(ack0_1),
  .data_out(data0_1),
  .rst(rst)
);
BE_behav_2merge merge1_0(
  .req_in({req0_1, req_in[4]}),
  .ack_in({ack0_1, ack_in[4]}),
  .data_in0(data_in4),
  .data_in1(data0_1),
  .req_out(req1_0),
  .ack_out(ack1_0),
  .data_out(data1_0),
  .rst(rst)
);
BE_behav_2merge merge2_0(
  .req_in({req1_0, req0_0}),
  .ack_in({ack1_0, ack0_0}),
  .data_in0(data0_0),
  .data_in1(data1_0),
  .req_out(req_out),
  .ack_out(ack_out),
  .data_out(data_out),
  .rst(rst)
);

endmodule

module BE_behav_latch(
  req_in ,
  ack_in ,
  data_in ,
  req_out ,
  ack_out ,
  data_out ,
  rst
);

input  req_in;
output ack_in;
input  [38:0] data_in;
output req_out;
input  ack_out;
output [38:0] data_out;
input  rst;

parameter DLY = 0.4;
reg en;
reg req_out;
reg [38:0] data_out;

assign ack_in = en;

always @(rst) begin
  if (rst == 1'b1) begin
    data_out <= 39'b0;
    en <= 1'b0;
  end
end

always begin
  wait (req_in == 1'b1);
  #DLY;
  en = 1'b1;
  wait (ack_out == 1'b1);
  wait (req_in == 1'b0);
  #DLY;
  en = 1'b0;
  wait (ack_out == 1'b0);
end

always @(en) begin
  req_out <= #1 en; // delay so pgm_split works
  if (en == 1'b1)
    data_out <= data_in;
end

endmodule

module BE_behav_tb();

parameter DLY = 0.4;

reg [4:0] ack_out, req_in;
reg [38:0] data_in0, data_in1, data_in2, data_in3, data_in4;
reg rst;
wire [4:0] ack_in, req_out;

BE_5x5_39_behav dut(
    .req_in(req_in),
    .ack_in(ack_in),
    .data_in4(data_in4),
    .data_in3(data_in3),
    .data_in2(data_in2),
    .data_in1(data_in1),
    .data_in0(data_in0),
    .req_out(req_out),
    .ack_out(ack_out),
    .data_out4(),
    .data_out3(),
    .data_out2(),
    .data_out1(),
    .data_out0(),
    .rst(rst)
);

initial begin
  // Reset
  rst <= 1'b0;
  rst <= #DLY 1'b1;
  rst <= #10 1'b0;
end

always begin // port 0
  data_in0 <= 39'b000000000000000000000000000000000000000;
  req_in[0] <= 1'b0;
  wait (rst == 1'b1);
  wait (rst == 1'b0);
  #DLY;
  while (1) begin
    data_in0 <= 39'b011111111111111111111111111111111111111;
    req_in[0] <= #DLY 1'b1;
    wait (ack_in[0] == 1'b1);
    req_in[0] <= #DLY 1'b0;
    wait (ack_in[0] == 1'b0);
    data_in0 <= 39'b110101010101010101010101010101010101010;
    req_in[0] <= #DLY 1'b1;
    wait (ack_in[0] == 1'b1);
    req_in[0] <= #DLY 1'b0;
    wait (ack_in[0] == 1'b0);
    #DLY;
    data_in0 <= 39'b000111111111111111111111111111111111111;
    req_in[0] <= #DLY 1'b1;
    wait (ack_in[0] == 1'b1);
    req_in[0] <= #DLY 1'b0;
    wait (ack_in[0] == 1'b0);
    data_in0 <= 39'b110111111111111111111111111111111111111;
    req_in[0] <= #DLY 1'b1;
    wait (ack_in[0] == 1'b1);
    req_in[0] <= #DLY 1'b0;
    wait (ack_in[0] == 1'b0);
    #DLY;
  end
end

initial begin // port 1
  data_in1 <= 39'b000000000000000000000000000000000000000;
  req_in[1] <= 1'b0;
  wait (rst == 1'b1);
  wait (rst == 1'b0);
  #10;
  while (1) begin
    data_in1 <= 39'b000100000000000000000000000000000000000;
    req_in[1] <= #DLY 1'b1;
    wait (ack_in[1] == 1'b1);
    req_in[1] <= #DLY 1'b0;
    wait (ack_in[1] == 1'b0);
    data_in1 <= 39'b100000000000000000000000000000000000000;
    req_in[1] <= #DLY 1'b1;
    wait (ack_in[1] == 1'b1);
    req_in[1] <= #DLY 1'b0;
    wait (ack_in[1] == 1'b0);
    #1;
  end
end

initial begin // port 2
  data_in2 <= 39'b000000000000000000000000000000000000000;
  req_in[2] <= 1'b0;
  wait (rst == 1'b1);
  wait (rst == 1'b0);
  #8;
  while (1) begin
    data_in2 <= 39'b001000000000000000000000000000000000000;
    req_in[2] <= #DLY 1'b1;
    wait (ack_in[2] == 1'b1);
    req_in[2] <= #DLY 1'b0;
```

```
      wait (ack_in[2] == 1'b0);
      data_in2 <= 39'b100000000000000000000000000000000000000;
      req_in[2] <= #DLY 1'b1;
      wait (ack_in[2] == 1'b1);
      req_in[2] <= #DLY 1'b0;
      wait (ack_in[2] == 1'b0);
      #10;
    end
  end

  initial begin // port 3
    data_in3 <= 39'b000000000000000000000000000000000000000;
    req_in[3] <= 1'b0;
    wait (rst == 1'b1);
    wait (rst == 1'b0);
    #2;
    while (1) begin
      data_in3 <= 39'b000100000000000000000000000000000000000;
      req_in[3] <= #DLY 1'b1;
      wait (ack_in[3] == 1'b1);
      req_in[3] <= #DLY 1'b0;
      wait (ack_in[3] == 1'b0);
      data_in3 <= 39'b111111111111111111110000000000000000000;
      req_in[3] <= #DLY 1'b1;
      wait (ack_in[3] == 1'b1);
      req_in[3] <= #DLY 1'b0;
      wait (ack_in[3] == 1'b0);
      #6;
    end
  end

  initial begin // port 4
    data_in4 <= 39'b000000000000000000000000000000000000000;
    req_in[4] <= 1'b0;
    wait (rst == 1'b1);
    wait (rst == 1'b0);
    #5;
    while (1) begin
      data_in4 <= 39'b011000000000000000000000000000000000000;
      req_in[4] <= #DLY 1'b1;
      wait (ack_in[4] == 1'b1);
      req_in[4] <= #DLY 1'b0;
      wait (ack_in[4] == 1'b0);
      data_in4 <= 39'b100000000000000000000011111111111111111;
      req_in[4] <= #DLY 1'b1;
      wait (ack_in[4] == 1'b1);
      req_in[4] <= #DLY 1'b0;
      wait (ack_in[4] == 1'b0);
      #2;
    end
  end

  always @(req_out) begin
    ack_out <= #DLY req_out;
  end
endmodule

module merge_tb();

reg ack_out;
reg [1:0] req_in;
reg [38:0] d0, d1;
reg rst;
wire [1:0] ack_in;
wire [38:0] d;

parameter DLY = 0.4;

BE_behav_2merge dut(
  .req_in(req_in),
  .ack_in(ack_in),
  .data_in0(d0),
  .data_in1(d1),
  .req_out(req_out),
    .ack_out(ack_out),
    .data_out(d),
    .rst(rst)
  );

  initial begin
    // Reset
    rst = 1'b0;
    rst = #DLY 1'b1;
    rst = #10 1'b0;
    #2;
    d0 = 39'b011111111111111111111111111111111111111;
    req_in = 2'b01;
    wait (ack_in[0] == 1'b1);
    req_in = 2'b00;
    wait (ack_in[0] == 1'b0);
    #DLY;
    d1 = 39'b000000000000000000000011111111111111111;
    req_in = 2'b10;
    #3;
    d0 = 39'b111111111111111111111100000000000000000;
    req_in = #DLY 2'b11;
    wait (ack_in[0] == 1'b1);
    req_in = #DLY 2'b10;
    wait (ack_in[0] == 1'b0);
    wait (ack_in[1] == 1'b1);
    req_in = #DLY 2'b00;
    wait (ack_in[1] == 1'b0);
    #3;
    d1 = 39'b101010101001010101010010101001010101010;
  end

  always @(req_out) begin
    ack_out <= #0.4 req_out;
  end
  endmodule
```

C.5 The Test Net

```
// MANGO router test setup

// this test setup provides two local ports into a three node network,
// which is loaded by random traffic on all other ports.
// one port will have an initiator NA attached, the other a target NA

// setup: (the numbers at the links are the port numbers of the
// appropriate router)
//              :: an initiator NA can be connected to the local
//                 port of Router1 and 2
//              :: port1 of Router1 is connected to port3 of Router2
//              :: port4 of Router2 is connected to port2 of Router3
//              :: port3 of Router3 is connected to port1 of Router4
//              :: port2 of Router4 is connected to port4 of Router1
//              :: the target NA can be connected to the local port of Router3
//
//     +--------+          +--------+
//     |       |1         3|        |
//     |Router1|-----------|Router2 |
//     |      /            |      /
//     +------+            +------+
//       4|                  4|
//        |                   |
//        |                   |
//       2|                  2|
//     +--------+          +--------+
//     |       |1         3|        |
//     |Router4|-----------|Router3 |
//     |      /            |      /
//     +------+            +------+
//
//


`timescale 1ns/1ps

module MANGO_router_thesis_test_3port(
// test port 1
    RxReq_1,
    RxAck_1,
    RxData_1,
    TxAck_1,
    TxReq_1,
    TxData_1,
// test port 2
    RxReq_2,
    RxAck_2,
    RxData_2,
    TxAck_2,
    TxReq_2,
    TxData_2,
// test port 3
    RxReq_3,
    RxAck_3,
    RxData_3,
    TxAck_3,
    TxReq_3,
    TxData_3,
    reset
);



// parameters
parameter RxPorts = 4;
parameter TxPorts = 4;
parameter flit_data_wdth = 39;
```

```
// ports
  // port 1
output  [RxPorts −1:0]      RxReq_1;
input   [RxPorts −1:0]      RxAck_1;
output  [RxPorts*flit_data_wdth −1:0]      RxData_1;
output  [TxPorts −1:0]      TxAck_1;
input   [TxPorts −1:0]      TxReq_1;
input   [TxPorts*flit_data_wdth −1:0]      TxData_1;

  // port 2
output  [RxPorts −1:0]      RxReq_2;
input   [RxPorts −1:0]      RxAck_2;
output  [RxPorts*flit_data_wdth −1:0]      RxData_2;
output  [TxPorts −1:0]      TxAck_2;
input   [TxPorts −1:0]      TxReq_2;
input   [TxPorts*flit_data_wdth −1:0]      TxData_2;

  // port 3
output  [RxPorts −1:0]      RxReq_3;
input   [RxPorts −1:0]      RxAck_3;
output  [RxPorts*flit_data_wdth −1:0]      RxData_3;
output  [TxPorts −1:0]      TxAck_3;
input   [TxPorts −1:0]      TxReq_3;
input   [TxPorts*flit_data_wdth −1:0]      TxData_3;

input   reset;

// wires
wire    [3:0]   req_R1_GS;
wire    [3:0]   ack_R1_GS;
wire    [38:0]  data_R1_GS0,data_R1_GS1,data_R1_GS2,data_R1_GS3;
wire            req_R1_BE, ack_R1_BE;
wire    [38:0]  data_R1_BE;
wire    [4:0]   req_credit_R1;          // only for network ports, not for local port
wire    [4:0]   ack_credit_R1;
wire    [7:0]   vc_ctrl1_R1, vc_ctrl2_R1, vc_ctrl3_R1, vc_ctrl4_R1;
wire    [44:0]  link1_out0_R1, link2_out0_R1, link3_out0_R1, link4_out0_R1;
wire    [44:0]  link1_out1_R1, link2_out1_R1, link3_out1_R1, link4_out1_R1;
wire    [44:0]  pipe11a_0, n_pipe11b_0, pipe23a_0, n_pipe23b_0;
wire    [44:0]  pipe11a_1, n_pipe11b_1, pipe23a_1, n_pipe23b_1;
wire    [44:0]  pipe24a_0, n_pipe24b_0, pipe32a_0, n_pipe32b_0;
wire    [44:0]  pipe24a_1, n_pipe24b_1, pipe32a_1, n_pipe32b_1;
wire    [44:0]  pipe33a_0, n_pipe33b_0, pipe41a_0, n_pipe41b_0;
wire    [44:0]  pipe33a_1, n_pipe33b_1, pipe41a_1, n_pipe41b_1;
wire    [44:0]  pipe14a_0, n_pipe14b_0, pipe42a_0, n_pipe42b_0;
wire    [44:0]  pipe14a_1, n_pipe14b_1, pipe42a_1, n_pipe42b_1;
wire            link1_ack_R1, link2_ack_R1, link3_ack_R1, link4_ack_R1;

wire    [3:0]   req_R2_GS;
wire    [3:0]   ack_R2_GS;
wire    [38:0]  data_R2_GS0,data_R2_GS1,data_R2_GS2,data_R2_GS3;
wire            req_R2_BE, ack_R2_BE;
wire    [38:0]  data_R2_BE;
wire    [4:0]   req_credit_R2;          // only for network ports, not for local port
wire    [4:0]   ack_credit_R2;
wire    [7:0]   vc_ctrl1_R2, vc_ctrl2_R2, vc_ctrl3_R2, vc_ctrl4_R2;
wire    [44:0]  link1_out0_R2, link2_out0_R2, link3_out0_R2, link4_out0_R2;
wire    [44:0]  link1_out1_R2, link2_out1_R2, link3_out1_R2, link4_out1_R2;
wire            link1_ack_R2, link2_ack_R2, link3_ack_R2, link4_ack_R2;

wire    [3:0]   req_R3_GS;
wire    [3:0]   ack_R3_GS;
wire    [38:0]  data_R3_GS0,data_R3_GS1,data_R3_GS2,data_R3_GS3;
wire            req_R3_BE, ack_R3_BE;
wire    [38:0]  data_R3_BE;
wire    [4:0]   req_credit_R3;          // only for network ports, not for local port
wire    [4:0]   ack_credit_R3;
wire    [7:0]   vc_ctrl1_R3, vc_ctrl2_R3, vc_ctrl3_R3, vc_ctrl4_R3;
wire    [44:0]  link1_out0_R3, link2_out0_R3, link3_out0_R3, link4_out0_R3;
wire    [44:0]  link1_out1_R3, link2_out1_R3, link3_out1_R3, link4_out1_R3;
wire            link1_ack_R3, link2_ack_R3, link3_ack_R3, link4_ack_R3;

wire    [3:0]   req_R4_GS;
wire    [3:0]   ack_R4_GS;
wire    [38:0]  data_R4_GS0,data_R4_GS1,data_R4_GS2,data_R4_GS3;
wire            req_R4_BE, ack_R4_BE;
wire    [38:0]  data_R4_BE;
wire    [4:0]   req_credit_R4;          // only for network ports, not for local port
wire    [4:0]   ack_credit_R4;
wire    [7:0]   vc_ctrl1_R4, vc_ctrl2_R4, vc_ctrl3_R4, vc_ctrl4_R4;
wire    [44:0]  link1_out0_R4, link2_out0_R4, link3_out0_R4, link4_out0_R4;
wire    [44:0]  link1_out1_R4, link2_out1_R4, link3_out1_R4, link4_out1_R4;
wire            link1_ack_R4, link2_ack_R4, link3_ack_R4, link4_ack_R4;

wire    [44:0]  gen43_out0, gen44_out0;
wire    [44:0]  gen43_out1, gen44_out1;
wire    [44:0]  gen34_out0, gen31_out0;
wire    [44:0]  gen34_out1, gen31_out1;
wire    [44:0]  gen21_out0, gen22_out0;
wire    [44:0]  gen21_out1, gen22_out1;
wire    [44:0]  gen12_out0, gen13_out0;
wire    [44:0]  gen12_out1, gen13_out1;

//wire  [3:0]   steer3_1, steer2_1, steer1_1, steer0_1;
//wire  [3:0]   steer3_2, steer2_2, steer1_2, steer0_2;

wire            ack_sink12, ack_sink13;
wire    [7:0]   vc_ctrl_sink12, vc_ctrl_sink13;
wire            ack_sink21, ack_sink22;
wire    [7:0]   vc_ctrl_sink21, vc_ctrl_sink22;
wire            ack_sink31, ack_sink34;
wire    [7:0]   vc_ctrl_sink31, vc_ctrl_sink34;
wire            ack_sink43, ack_sink44;
wire    [7:0]   vc_ctrl_sink43, vc_ctrl_sink44;

// define background traffic generators
// (...also remember to set up the connections they use, by programming the routers...)
//`include "./DATA/generator_setup.param.v"
`include "autogen_setup.param.v"

// define sidechain usage (BE channels)
// ...this is based on the topology of the network
defparam router1.sidechain1 = 5'b11100;
defparam router1.sidechain2 = 5'b00000;
defparam router1.sidechain3 = 5'b00000;
defparam router1.sidechain4 = 5'b10100;
defparam router2.sidechain1 = 5'b00000;
defparam router2.sidechain2 = 5'b00000;
defparam router2.sidechain3 = 5'b01100;
defparam router2.sidechain4 = 5'b10100;
defparam router3.sidechain1 = 5'b00000;
defparam router3.sidechain2 = 5'b00100;
defparam router3.sidechain3 = 5'b01100;
defparam router3.sidechain4 = 5'b00000;
defparam router4.sidechain1 = 5'b11100;
defparam router4.sidechain2 = 5'b00100;
defparam router4.sidechain3 = 5'b00000;
defparam router4.sidechain4 = 5'b00000;
// routers coming up...

// test port 1

assign RxReq_1 = { req_R1_GS[3:1], req_R1_BE };
assign RxData_1 = { data_R1_GS3, data_R1_GS2, data_R1_GS1, data_R1_BE};
assign TxAck_1 = { ack_R1_GS[3:1], ack_R1_BE };

/* this work around is not needed anymore
assign steer3_1 = 4'b0000;
assign steer2_1 = 4'b0100;      // steer channel 2 to port 1, VC 4
assign steer1_1 = 4'b0111;      // steer channel 1 to port 1, VC 7
assign steer0_1 = 4'b0000;
*/


MANGO_router_5ports_39bits router1(
// local port (port 0), 4 GS chs + 1 BE ch
  // GS
```

```
    .req0_in( { TxReq_1[3:1], 1'b0 } ),
    .ack0_in( ack_R1_GS ),
    .data0_in0( 43'b0 ),        // inclusive steering to VC (4 bits... 1 of 4 upper VCs on each of the 4 output ports).
    .data0_in1( { 4'b0000, TxData_1[2*flit_data_wdth -1:flit_data_wdth] } ),     // steer3_1..steer0_1 not used anymore.
    .data0_in2( { 4'b0000, TxData_1[3*flit_data_wdth -1:2*flit_data_wdth] } ),
    .data0_in3( { 4'b0000, TxData_1[4*flit_data_wdth -1:3*flit_data_wdth] } ),
/*
    .data0_in1( { steer1_1, TxData_1[2*flit_data_wdth -1:flit_data_wdth] } ),    // steer3_1..steer0_1 not used anymore.
    .data0_in2( { steer2_1, TxData_1[3*flit_data_wdth -1:2*flit_data_wdth] } ),
    .data0_in3( { steer3_1, TxData_1[4*flit_data_wdth -1:3*flit_data_wdth] } ),
*/
    .req0_out( req_R1_GS ),      // router 1, port yy
    .ack0_out( { RxAck_1[3:1], 1'b0 } ),
    .data0_out0( data_R1_GS0 ),
    .data0_out1( data_R1_GS1 ),
    .data0_out2( data_R1_GS2 ),
    .data0_out3( data_R1_GS3 ),
  // BE
    .req_BE_in( TxReq_1[0] ),
    .ack_BE_in( ack_R1_BE ),
    .data_BE_in( TxData_1[flit_data_wdth -1:0] ),
    .req_BE_out( req_R1_BE ),
    .ack_BE_out( RxAck_1[0] ),
    .data_BE_out( data_R1_BE ),

// DI network ports and VC control

  // BE credits
    .req_credit_BE_in( { req_credit_R4[2], req_credit_sink13, req_credit_sink12, req_credit_R2[3], 1'b0 } ),
    .ack_credit_BE_in( ack_credit_R1 ),
    .req_credit_BE_out( req_credit_R1 ),
    .ack_credit_BE_out( { ack_credit_R4[2], ack_credit_sink13, ack_credit_sink12, ack_credit_R2[3], 1'b0 } ),
/*
    .req_credit_BE_in( req_credit_R1 ),
    .ack_credit_BE_in( { 1'b0, 1'b0, 1'b0, ack_credit_R2[3], 1'b0 } ),
    .req_credit_BE_out( { 1'b0, 1'b0, 1'b0, 1'b0, 1'b0 } ),
    .ack_credit_BE_out( ack_credit_R1 ),
*/
  // port 1
    .n_link1_in0( n_pipe23b_0 ),
    .n_link1_in1( n_pipe23b_1 ),
    .link1_ack_in( link1_ack_R1 ),
    .vc_ctrl1_in( vc_ctrl1_R1 ),        // this is an output (!)
    .n_link1_out0( link1_out0_R1 ),
    .n_link1_out1( link1_out1_R1 ),
    .link1_ack_out( ack_pipe11a_in ),
    .vc_ctrl1_out( vc_ctrl3_R2 ),       // this is an input (!)
  // port 2
    .n_link2_in0( gen12_out0 ),
    .n_link2_in1( gen12_out1 ),
    .link2_ack_in( link2_ack_R1 ),
    .vc_ctrl2_in( vc_ctrl2_R1 ),
    .n_link2_out0( link2_out0_R1 ),
    .n_link2_out1( link2_out1_R1 ),
    .link2_ack_out( ack_sink12 ),
    .vc_ctrl2_out( vc_ctrl_sink12 ),
  // port 3
    .n_link3_in0( gen13_out0 ),
    .n_link3_in1( gen13_out1 ),
    .link3_ack_in( link3_ack_R1 ),
    .vc_ctrl3_in( vc_ctrl3_R1 ),
    .n_link3_out0( link3_out0_R1 ),
    .n_link3_out1( link3_out1_R1 ),
    .link3_ack_out( ack_sink13 ),
    .vc_ctrl3_out( vc_ctrl_sink13 ),
  // port 4
    .n_link4_in0( link2_out0_R4 ),//n_pipe42b_0 ),
    .n_link4_in1( link2_out1_R4 ),//n_pipe42b_1 ),
    .link4_ack_in( link4_ack_R1 ),
    .vc_ctrl4_in( vc_ctrl4_R1 ),
    .n_link4_out0( link4_out0_R1 ),
    .n_link4_out1( link4_out1_R1 ),
    .link4_ack_out( link2_ack_R4 ),//ack_pipe14a_in ),
    .vc_ctrl4_out( vc_ctrl2_R4 ),
    .reset( reset )
);

// link sinks + generators
link_sink_sidechain sink12(
    .n_in0(link2_out0_R1 ),
    .n_in1(link2_out1_R1 ),
    .ack( ack_sink12 ),
    .vc_ctrl( vc_ctrl_sink12 ),
    .req_credit_out( req_credit_sink12 ),
    .ack_credit_out( ack_credit_R1[2] ),
    .reset( reset )
);
link_gen_sidechain gen12(
    .n_out0( gen12_out0 ),
    .n_out1( gen12_out1 ),
    .ack( link2_ack_R1 ),
    .vc_ctrl( vc_ctrl2_R1 ),
    .req_credit_in( req_credit_R1[2] ),
    .ack_credit_in( ack_credit_sink12 ),
    .reset( reset )
);

link_sink_sidechain sink13(
    .n_in0(link3_out0_R1 ),
    .n_in1(link3_out1_R1 ),
    .ack( ack_sink13 ),
    .vc_ctrl( vc_ctrl_sink13 ),
    .req_credit_out( req_credit_sink13 ),
    .ack_credit_out( ack_credit_R1[3] ),
    .reset( reset )
);
link_gen_sidechain gen13(
    .n_out0( gen13_out0 ),
    .n_out1( gen13_out1 ),
    .ack( link3_ack_R1 ),
    .vc_ctrl( vc_ctrl3_R1 ),
    .req_credit_in( req_credit_R1[3] ),
    .ack_credit_in( ack_credit_sink13 ),
    .reset( reset )
);

// pipelines
reg45_2ph_dr_inv_rst1_0 pipe11a(
    .n_in_t( link1_out1_R1 ),
    .n_in_f( link1_out0_R1 ),
    .ack_in( ack_pipe11a_in ),
    .out_t( pipe11a_1 ),
    .out_f( pipe11a_0 ),
    .n_ack_out( n_ack_pipe11a_out ),
    .reset( reset )
);
reg45_2ph_dr_inv_rst1_1 pipe11b(
    .in_t( pipe11a_1 ),
    .in_f( pipe11a_0 ),
    .n_ack_in( n_ack_pipe11a_out ),
    .n_out_t( n_pipe11b_1 ),
    .n_out_f( n_pipe11b_0 ),
    .ack_out( link3_ack_R2 ),
    .reset( reset )
);
reg45_2ph_dr_inv_rst1_0 pipe23a(
    .n_in_t( link3_out1_R2 ),
    .n_in_f( link3_out0_R2 ),
    .ack_in( ack_pipe23a_in ),
    .out_t( pipe23a_1 ),
    .out_f( pipe23a_0 ),
    .n_ack_out( n_ack_pipe23a_out ),
    .reset( reset )
);
reg45_2ph_dr_inv_rst1_1 pipe23b(
    .in_t( pipe23a_1 ),
    .in_f( pipe23a_0 ),
    .n_ack_in( n_ack_pipe23a_out ),
    .n_out_t( n_pipe23b_1 ),
    .n_out_f( n_pipe23b_0 ),
    .ack_out( link1_ack_R1 ),
    .reset( reset )
);

// test port 2

assign RxReq_2 = { req_R2_GS[3:1], req_R2_BE };
assign RxData_2 = { data_R2_GS3, data_R2_GS2, data_R2_GS1, data_R2_BE};
assign TxAck_2 = { ack_R2_GS[3:1], ack_R2_BE };

MANGO_router_5ports_39bits router2(
// local port (port 0), 4 GS chs + 1 BE ch
  // GS
    .req0_in( { TxReq_2[3:1], 1'b0 } ),
    .ack0_in( ack_R2_GS ),
    .data0_in0( 43'b0 ),
    .data0_in1( { 4'b0000, TxData_2[2*flit_data_wdth -1:flit_data_wdth] } ),
    .data0_in2( { 4'b0000, TxData_2[3*flit_data_wdth -1:2*flit_data_wdth] } ),
    .data0_in3( { 4'b0000, TxData_2[4*flit_data_wdth -1:3*flit_data_wdth] } ),
    .req0_out( req_R2_GS ),      // router 1, port yy
    .ack0_out( { RxAck_2[3:1], 1'b0 } ),
    .data0_out0( data_R2_GS0 ),
    .data0_out1( data_R2_GS1 ),
    .data0_out2( data_R2_GS2 ),
    .data0_out3( data_R2_GS3 ),
  // BE
    .req_BE_in( TxReq_2[0] ),
    .ack_BE_in( ack_R2_BE ),
    .data_BE_in( TxData_2[flit_data_wdth -1:0] ),
    .req_BE_out( req_R2_BE ),
    .ack_BE_out( RxAck_2[0] ),
    .data_BE_out( data_R2_BE ),

// DI network ports and VC control

  // BE credits
    .req_credit_BE_in( { req_credit_R3[2], req_credit_R1[1], req_credit_sink22, req_credit_sink21, 1'b0 } ),
    .ack_credit_BE_in( ack_credit_R2 ),
    .req_credit_BE_out( req_credit_R2 ),
    .ack_credit_BE_out( { ack_credit_R3[2], ack_credit_R1[1], ack_credit_sink22, ack_credit_sink21, 1'b0 } ),
/*
    .req_credit_BE_in( req_credit_R2 ), // this is an output (!)
    .ack_credit_BE_in( { ack_credit_R3[2], 1'b0, 1'b0, 1'b0, 1'b0 } ),
    .req_credit_BE_out( { 1'b0, req_credit_R1[1], 1'b0, 1'b0, 1'b0 } ), // this is an input (!)
    .ack_credit_BE_out( ack_credit_R2 ),
*/
  // port 1
    .n_link1_in0( gen21_out0 ),
    .n_link1_in1( gen21_out1 ),
    .link1_ack_in( link1_ack_R2 ),
    .vc_ctrl1_in( vc_ctrl1_R2 ),        // this is an output (!)
    .n_link1_out0( link1_out0_R2 ),
    .n_link1_out1( link1_out1_R2 ),
    .link1_ack_out( ack_sink21 ),
    .vc_ctrl1_out( vc_ctrl_sink21 ),    // this is an input (!)
  // port 2
    .n_link2_in0( gen22_out0 ),
    .n_link2_in1( gen22_out1 ),
    .link2_ack_in( link2_ack_R2 ),
    .vc_ctrl2_in( vc_ctrl2_R2 ),
    .n_link2_out0( link2_out0_R2 ),
    .n_link2_out1( link2_out1_R2 ),
    .link2_ack_out( ack_sink22 ),
    .vc_ctrl2_out( vc_ctrl_sink22 ),
  // port 3
    .n_link3_in0( n_pipe11b_0 ),
    .n_link3_in1( n_pipe11b_1 ),
    .link3_ack_in( link3_ack_R2 ),
    .vc_ctrl3_in( vc_ctrl3_R2 ),
    .n_link3_out0( link3_out0_R2 ),
    .n_link3_out1( link3_out1_R2 ),
    .link3_ack_out( ack_pipe23a_in ),
        .vc_ctrl3_out( vc_ctrl1_R1 ),
    // port 4
        .n_link4_in0( n_pipe32b_0 ),
        .n_link4_in1( n_pipe32b_1 ),
        .link4_ack_in( link4_ack_R2 ),
        .vc_ctrl4_in( vc_ctrl4_R2 ),
        .n_link4_out0( link4_out0_R2 ),
        .n_link4_out1( link4_out1_R2 ),
        .link4_ack_out( ack_pipe24a_in ),
        .vc_ctrl4_out( vc_ctrl2_R3 ),
        .reset( reset )
);

// link sinks
link_sink_sidechain sink21(
        .n_in0(link1_out0_R2 ),
        .n_in1(link1_out1_R2 ),
        .ack( ack_sink21 ),
        .vc_ctrl( vc_ctrl_sink21 ),
        .req_credit_out( req_credit_sink21 ),
        .ack_credit_out( ack_credit_R2[1] ),
        .reset( reset )
);
link_gen_sidechain gen21(
        .n_out0( gen21_out0 ),
        .n_out1( gen21_out1 ),
        .ack( link1_ack_R2 ),
        .vc_ctrl( vc_ctrl1_R2 ),
        .req_credit_in( req_credit_R2[1] ),
        .ack_credit_in( ack_credit_sink21 ),
        .reset( reset )
);

link_sink_sidechain sink22(
        .n_in0(link2_out0_R2 ),
        .n_in1(link2_out1_R2 ),
        .ack( ack_sink22 ),
        .vc_ctrl( vc_ctrl_sink22 ),
        .req_credit_out( req_credit_sink22 ),
        .ack_credit_out( ack_credit_R2[2] ),
        .reset( reset )
);
link_gen_sidechain gen22(
        .n_out0( gen22_out0 ),
        .n_out1( gen22_out1 ),
        .ack( link2_ack_R2 ),
        .vc_ctrl( vc_ctrl2_R2 ),
        .req_credit_in( req_credit_R2[2] ),
        .ack_credit_in( ack_credit_sink22 ),
        .reset( reset )
);

// pipelines
reg45_2ph_dr_inv_rst1_0 pipe24a(
        .n_in_t( link4_out1_R2 ),
        .n_in_f( link4_out0_R2 ),
        .ack_in( ack_pipe24a_in ),
        .out_t( pipe24a_1 ),
        .out_f( pipe24a_0 ),
        .n_ack_out( n_ack_pipe24a_out ),
        .reset( reset )
);
reg45_2ph_dr_inv_rst1_1 pipe24b(
        .in_t( pipe24a_1 ),
        .in_f( pipe24a_0 ),
        .n_ack_in( n_ack_pipe24a_out ),
        .n_out_t( n_pipe24b_1 ),
        .n_out_f( n_pipe24b_0 ),
        .ack_out( link2_ack_R3 ),
        .reset( reset )
);
reg45_2ph_dr_inv_rst1_0 pipe32a(
        .n_in_t( link2_out1_R3 ),
        .n_in_f( link2_out0_R3 ),
    .ack_in( ack_pipe32a_in ),
    .out_t( pipe32a_1 ),
    .out_f( pipe32a_0 ),
    .n_ack_out( n_ack_pipe32a_out ),
    .reset( reset )
);
reg45_2ph_dr_inv_rst1_1 pipe32b(
    .in_t( pipe32a_1 ),
    .in_f( pipe32a_0 ),
    .n_ack_in( n_ack_pipe32a_out ),
    .n_out_t( n_pipe32b_1 ),
    .n_out_f( n_pipe32b_0 ),
    .ack_out( link4_ack_R2 ),
    .reset( reset )
);


// test port 3

assign RxReq_3 = { req_R3_GS[3:1], req_R3_BE };
assign RxData_3 = { data_R3_GS3, data_R3_GS2, data_R3_GS1, data_R3_BE};
assign TxAck_3 = { ack_R3_GS[3:1], ack_R3_BE };

/* this work around is not needed anymore
assign steer3_2 = 4'b0000;
assign steer2_2 = 4'b0101;
assign steer1_2 = 4'b0000;
assign steer0_2 = 4'b0000;
*/

MANGO_router_5ports_39bits router3(
// local port (port 0), 4 GS chs + 1 BE ch
  // GS
    .req0_in( { TxReq_3[3:1], 1'b0 } ),
    .ack0_in( ack_R3_GS ),
    .data0_in0( 43'b0 ),        // inclusive steering to VC (4 bits... 1 of 4 upper VCs on each of the 4 output ports). T
    .data0_in1( { 4'b0000, TxData_3[2*flit_data_wdth -1:flit_data_wdth] } ),
    .data0_in2( { 4'b0000, TxData_3[3*flit_data_wdth -1:2*flit_data_wdth] } ),
    .data0_in3( { 4'b0000, TxData_3[4*flit_data_wdth -1:3*flit_data_wdth] } ),
/*
    .data0_in1( { steer1_2, TxData_2[2*flit_data_wdth -1:flit_data_wdth] } ),
    .data0_in2( { steer2_2, TxData_2[3*flit_data_wdth -1:2*flit_data_wdth] } ),
    .data0_in3( { steer3_2, TxData_2[4*flit_data_wdth -1:3*flit_data_wdth] } ),
*/
    .req0_out( req_R3_GS ),      // router 1, port yy
    .ack0_out( { RxAck_3[3:1], 1'b0 } ),
    .data0_out0( data_R3_GS0 ),
    .data0_out1( data_R3_GS1 ),
    .data0_out2( data_R3_GS2 ),
    .data0_out3( data_R3_GS3 ),
  // BE
    .req_BE_in( TxReq_3[0] ),
    .ack_BE_in( ack_R3_BE ),
    .data_BE_in( TxData_3[flit_data_wdth -1:0] ),
    .req_BE_out( req_R3_BE ),
    .ack_BE_out( RxAck_3[0] ),
    .data_BE_out( data_R3_BE ),

// DI network ports and VC control

  // BE credits
    .req_credit_BE_in( { req_credit_sink34, req_credit_R4[1], req_credit_R2[4], req_credit_sink31, 1'b0 } ),
    .ack_credit_BE_in( ack_credit_R3 ),
    .req_credit_BE_out( req_credit_R3 ),
    .ack_credit_BE_out( { ack_credit_sink34, ack_credit_R4[1], ack_credit_R2[4], ack_credit_sink31, 1'b0 } ),
/*
    .req_credit_BE_in( req_credit_R3 ),
    .ack_credit_BE_in( { 1'b0, 1'b0, 1'b0, 1'b0, 1'b0 } ),
    .req_credit_BE_out( { req_credit_sink34, req_credit_sink33, req_credit_R2[4], req_credit_sink31, 1'b0 } ),
    .ack_credit_BE_out( ack_credit_R3 ),
*/
  // port 1
    .n_link1_in0( gen31_out0 ),
    .n_link1_in1( gen31_out1 ),
    .link1_ack_in( link1_ack_R3 ),
    .vc_ctrl1_in( vc_ctrl1_R3 ),        // this is an output (!)
    .n_link1_out0( link1_out0_R3 ),
    .n_link1_out1( link1_out1_R3 ),
    .link1_ack_out( ack_sink31 ),
    .vc_ctrl1_out( vc_ctrl_sink31 ),    // this is an input (!)
  // port 2
    .n_link2_in0( n_pipe24b_0 ),
    .n_link2_in1( n_pipe24b_1 ),
    .link2_ack_in( link2_ack_R3 ),
    .vc_ctrl2_in( vc_ctrl2_R3 ),
    .n_link2_out0( link2_out0_R3 ),
    .n_link2_out1( link2_out1_R3 ),
    .link2_ack_out( ack_pipe32a_in ),
    .vc_ctrl2_out( vc_ctrl4_R2 ),
  // port 3
    .n_link3_in0( link1_out0_R4 ),//n_pipe41b_0 ),
    .n_link3_in1( link1_out1_R4 ),//n_pipe41b_1 ),
    .link3_ack_in( link3_ack_R3 ),
    .vc_ctrl3_in( vc_ctrl3_R3 ),
    .n_link3_out0( link3_out0_R3 ),
    .n_link3_out1( link3_out1_R3 ),
    .link3_ack_out( link1_ack_R4 ),//ack_pipe33a_in ),
    .vc_ctrl3_out( vc_ctrl1_R4 ),
  // port 4
    .n_link4_in0( gen34_out0 ),
    .n_link4_in1( gen34_out1 ),
    .link4_ack_in( link4_ack_R3 ),
    .vc_ctrl4_in( vc_ctrl4_R3 ),
    .n_link4_out0( link4_out0_R3 ),
    .n_link4_out1( link4_out1_R3 ),
    .link4_ack_out( ack_sink34 ),
    .vc_ctrl4_out( vc_ctrl_sink34 ),
    .reset( reset )
);

// link sinks
link_sink_sidechain sink31(
    .n_in0(link1_out0_R3 ),
    .n_in1(link1_out1_R3 ),
    .ack( ack_sink31 ),
    .vc_ctrl( vc_ctrl_sink31 ),
    .req_credit_out( req_credit_sink31 ),
    .ack_credit_out( ack_credit_R3[1] ),
    .reset( reset )
);
link_gen_sidechain gen31(
    .n_out0( gen31_out0 ),
    .n_out1( gen31_out1 ),
    .ack( link1_ack_R3 ),
    .vc_ctrl( vc_ctrl1_R3 ),
    .req_credit_in( req_credit_R3[1] ),
    .ack_credit_in( ack_credit_sink31 ),
    .reset( reset )
);

link_sink_sidechain sink34(
    .n_in0(link4_out0_R3 ),
    .n_in1(link4_out1_R3 ),
    .ack( ack_sink34 ),
    .vc_ctrl( vc_ctrl_sink34 ),
    .req_credit_out( req_credit_sink34 ),
    .ack_credit_out( ack_credit_R3[4] ),
    .reset( reset )
);
link_gen_sidechain gen34(
    .n_out0( gen34_out0 ),
    .n_out1( gen34_out1 ),
    .ack( link4_ack_R3 ),
    .vc_ctrl( vc_ctrl4_R3 ),
    .req_credit_in( req_credit_R3[4] ),
    .ack_credit_in( ack_credit_sink34 ),
    .reset( reset )
);
```

```
// pipelines
//reg45_2ph_dr_inv_rst1_0 pipe33a(
//     .n_in_t( link3_out1_R3 ),
//     .n_in_f( link3_out0_R3 ),
//     .ack_in( ack_pipe33a_in ),
//     .out_t( pipe33a_1 ),
//     .out_f( pipe33a_0 ),
//     .n_ack_out( n_ack_pipe33a_out ),
//     .reset( reset )
//);
//reg45_2ph_dr_inv_rst1_1 pipe33b(
//     .in_t( pipe33a_1 ),
//     .in_f( pipe33a_0 ),
//     .n_ack_in( n_ack_pipe33a_out ),
//     .n_out_t( n_pipe33b_1 ),
//     .n_out_f( n_pipe33b_0 ),
//     .ack_out( link1_ack_R4 ),
//     .reset( reset )
//);
//reg45_2ph_dr_inv_rst1_0 pipe41a(
//     .n_in_t( link1_out1_R4 ),
//     .n_in_f( link1_out0_R4 ),
//     .ack_in( ack_pipe41a_in ),
//     .out_t( pipe41a_1 ),
//     .out_f( pipe41a_0 ),
//     .n_ack_out( n_ack_pipe41a_out ),
//     .reset( reset )
//);
//reg45_2ph_dr_inv_rst1_1 pipe41b(
//     .in_t( pipe41a_1 ),
//     .in_f( pipe41a_0 ),
//     .n_ack_in( n_ack_pipe41a_out ),
//     .n_out_t( n_pipe41b_1 ),
//     .n_out_f( n_pipe41b_0 ),
//     .ack_out( link1_ack_R4 ),
//     .reset( reset )
//);

MANGO_router_5ports_39bits router4(
// local port (port 0), 4 GS chs + 1 BE ch
  // GS
    .req0_in( { 1'b0, 1'b0, 1'b0, 1'b0 } ),
    .ack0_in( ack_R4_GS ),
    .data0_in0( 43'b0 ),
    .data0_in1( 43'b0 ),
    .data0_in2( 43'b0 ),
    .data0_in3( 43'b0 ),
    .req0_out( req_R4_GS ),      // router 1, port yy
    .ack0_out( { 1'b0, 1'b0, 1'b0, 1'b0 } ),
    .data0_out0( data_R4_GS0 ),
    .data0_out1( data_R4_GS1 ),
    .data0_out2( data_R4_GS2 ),
    .data0_out3( data_R4_GS3 ),
  // BE
    .req_BE_in( 1'b0 ),
    .ack_BE_in( ack_R4_BE ),
    .data_BE_in( 39'b0 ),
    .req_BE_out( req_R4_BE ),
    .ack_BE_out( 1'b0 ),
    .data_BE_out( data_R4_BE ),

// DI network ports and VC control

  // BE credits
    .req_credit_BE_in( { req_credit_sink44, req_credit_sink43, req_credit_R1[4], req_credit_R3[3], 1'b0 } ),
    .ack_credit_BE_in( ack_credit_R4 ),
    .req_credit_BE_out( req_credit_R4 ),
    .ack_credit_BE_out( { ack_credit_sink44, ack_credit_sink43, ack_credit_R1[4], ack_credit_R3[3], 1'b0 } ),
  // port 1
    .n_link1_in0( link3_out0_R3 ),//n_pipe33b_0 ),
    .n_link1_in1( link3_out1_R3 ),//n_pipe33b_1 ),
    .link1_ack_in( link1_ack_R4 ),
    .vc_ctrl1_in( vc_ctrl1_R4 ),        // this is an output (!)
    .n_link1_out0( link1_out0_R4 ),
    .n_link1_out1( link1_out1_R4 ),
    .link1_ack_out( link3_ack_R3),//ack_pipe41a_in ),
    .vc_ctrl1_out( vc_ctrl3_R3 ),         // this is an input (!)
  // port 2
    .n_link2_in0( link4_out0_R1),//n_pipe14b_0 ),
    .n_link2_in1( link4_out1_R1),//n_pipe14b_1 ),
    .link2_ack_in( link2_ack_R4 ),
    .vc_ctrl2_in( vc_ctrl2_R4 ),
    .n_link2_out0( link2_out0_R4 ),
    .n_link2_out1( link2_out1_R4 ),
    .link2_ack_out( link4_ack_R1),//ack_pipe42a_in ),
    .vc_ctrl2_out( vc_ctrl4_R1 ),
  // port 3
    .n_link3_in0( gen43_out0 ),
    .n_link3_in1( gen43_out1 ),
    .link3_ack_in( link3_ack_R4 ),
    .vc_ctrl3_in( vc_ctrl3_R4 ),
    .n_link3_out0( link3_out0_R4 ),
    .n_link3_out1( link3_out1_R4 ),
    .link3_ack_out( ack_sink43 ),
    .vc_ctrl3_out( vc_ctrl_sink43 ),
  // port 4
    .n_link4_in0( gen44_out0 ),
    .n_link4_in1( gen44_out1 ),
    .link4_ack_in( link4_ack_R4 ),
    .vc_ctrl4_in( vc_ctrl4_R4 ),
    .n_link4_out0( link4_out0_R4 ),
    .n_link4_out1( link4_out1_R4 ),
    .link4_ack_out( ack_sink44 ),
    .vc_ctrl4_out( vc_ctrl_sink34 ),
    .reset( reset )
);

// link sinks
link_sink_sidechain sink43(
    .n_in0(link3_out0_R4 ),
    .n_in1(link3_out1_R4 ),
    .ack( ack_sink43 ),
    .vc_ctrl( vc_ctrl_sink43 ),
    .req_credit_out( req_credit_sink43 ),
    .ack_credit_out( ack_credit_R4[3] ),
    .reset( reset )
);
link_gen_sidechain gen43(
    .n_out0( gen43_out0 ),
    .n_out1( gen43_out1 ),
    .ack( link3_ack_R4 ),
    .vc_ctrl( vc_ctrl1_R4 ),
    .req_credit_in( req_credit_R4[3] ),
    .ack_credit_in( ack_credit_sink43 ),
    .reset( reset )
);

link_sink_sidechain sink44(
    .n_in0(link4_out0_R4 ),
    .n_in1(link4_out1_R4 ),
    .ack( ack_sink44 ),
    .vc_ctrl( vc_ctrl_sink44 ),
    .req_credit_out( req_credit_sink44 ),
    .ack_credit_out( ack_credit_R4[4] ),
    .reset( reset )
);
link_gen_sidechain gen44(
    .n_out0( gen44_out0 ),
    .n_out1( gen44_out1 ),
    .ack( link4_ack_R4 ),
    .vc_ctrl( vc_ctrl4_R4 ),
    .req_credit_in( req_credit_R4[4] ),
    .ack_credit_in( ack_credit_sink44 ),
    .reset( reset )
);

// pipelines
```

```
//reg45_2ph_dr_inv_rst1_0 pipe42a(
//      .n_in_t( link2_out1_R4 ),
//      .n_in_f( link2_out0_R4 ),
//      .ack_in( ack_pipe42a_in ),
//      .out_t( pipe42a_1 ),
//      .out_f( pipe42a_0 ),
//      .n_ack_out( n_ack_pipe42a_out ),
//      .reset( reset )
//);
//reg45_2ph_dr_inv_rst1_1 pipe42b(
//      .in_t( pipe42a_1 ),
//      .in_f( pipe42a_0 ),
//      .n_ack_in( n_ack_pipe42a_out ),
//      .n_out_t( n_pipe42b_1 ),
//      .n_out_f( n_pipe42b_0 ),
//      .ack_out( link4_ack_R1 ),
//      .reset( reset )
//);
//reg45_2ph_dr_inv_rst1_0 pipe14a(
//      .n_in_t( link4_out1_R1 ),
//      .n_in_f( link4_out0_R1 ),
//      .ack_in( ack_pipe14a_in ),
//      .out_t( pipe14a_1 ),
//      .out_f( pipe14a_0 ),
//      .n_ack_out( n_ack_pipe14a_out ),
//      .reset( reset )
//);
//reg45_2ph_dr_inv_rst1_1 pipe14b(
//      .in_t( pipe14a_1 ),
//      .in_f( pipe14a_0 ),
//      .n_ack_in( n_ack_pipe14a_out ),
//      .n_out_t( n_pipe14b_1 ),
//      .n_out_f( n_pipe14b_0 ),
//      .ack_out( link4_ack_R1 ),
//      .reset( reset )
//);

endmodule
```

C.6 The Test Bench

```
// MANGO OCP end2end test modified by CPP

// this test setup provides two OCP ports, a master and a slave,
// into a three node network, which is loaded by random traffic on
// all other ports.

// setup: (the numbers at the links are the port numbers of the
// appropriate router)
//                :: the initiator NA can be connected to the local
//                   port of Router1
//                :: port1 of Router1 is connected to port3 of Router2
//                :: port4 of Router2 is connected to port2 of Router3
//                :: the target NA can be connected to the local port of Router3
//
//    +--------+             +--------+
//    |        |1           3|        |
//    |Router1 |-------------|Router2 |
//    |     /  +---------+   |     /  +---------+
//    +------+ /initiator|   +------+ /initiator|
//       4|    |   NA    |      4|    |   NA    |
//        |    +---------+       |    +---------+
//        |                      |
//       2|                     2|
//    +--------+             +--------+
//    |        |1           3|        |
//    |Router4 |-------------|Router3 |
//    |     /                |     /  +-------+
//    +------+               +------+ / target|
//                                    |  NA   |
//                                    +-------+


`timescale 1ns/1ps


module MANGO_OCP_e2e_test_VC (
);


// parameters
parameter RxPorts = 4;
parameter TxPorts = 4;
parameter flit_data_wdth = 39;

// OCP parameters
  parameter addr_wdth = 32;
  parameter data_wdth = 32;
  parameter burstlength_wdth = 8;
  parameter Mthreadid_wdth = 3;
  parameter Sthreadid_wdth = 2;
  parameter connid_wdth = 2;


// wires
  // initiator 1
wire     [RxPorts-1:0]       RxReq_1;
wire     [RxPorts-1:0]       RxAck_1;
wire     [RxPorts*flit_data_wdth-1:0]      RxData_1;
wire     [TxPorts-1:0]       TxAck_1;
wire     [TxPorts-1:0]       TxReq_1;
wire     [TxPorts*flit_data_wdth-1:0]      TxData_1;

  // initiator 2
wire     [RxPorts-1:0]       RxReq_2;
wire     [RxPorts-1:0]       RxAck_2;
wire     [RxPorts*flit_data_wdth-1:0]      RxData_2;
wire     [TxPorts-1:0]       TxAck_2;
wire     [TxPorts-1:0]       TxReq_2;
wire     [TxPorts*flit_data_wdth-1:0]      TxData_2;
```

```
  // initiator 3
wire     [RxPorts -1:0]      RxReq_3;
wire     [RxPorts -1:0]      RxAck_3;
wire     [RxPorts*flit_data_wdth -1:0]      RxData_3;
wire     [TxPorts -1:0]      TxAck_3;
wire     [TxPorts -1:0]      TxReq_3;
wire     [TxPorts*flit_data_wdth -1:0]      TxData_3;

wire     rst;

wire     [2:0]                      M1_MCmd;
wire                                M1_SCmdAccept;
wire     [addr_wdth -1:0]           M1_MAddr;
wire     [data_wdth -1:0]           M1_MData, M1_SData;
wire     [burstlength_wdth -1:0]    M1_MBurstLength;
wire     [2:0]                      M1_MBurstSeq;
wire     [connid_wdth -1:0]         M1_MConnID;
wire     [Mthreadid_wdth -1:0]      M1_MThreadID, M1_STrheadID, M1_MDataThreadID;
wire     [1:0]                      M1_SResp;

wire     [2:0]                      M2_MCmd;
wire                                M2_SCmdAccept;
wire     [addr_wdth -1:0]           M2_MAddr;
wire     [data_wdth -1:0]           M2_MData, M2_SData;
wire     [burstlength_wdth -1:0]    M2_MBurstLength;
wire     [2:0]                      M2_MBurstSeq;
wire     [connid_wdth -1:0]         M2_MConnID;
wire     [Mthreadid_wdth -1:0]      M2_MThreadID, M2_STrheadID, M2_MDataThreadID;
wire     [1:0]                      M2_SResp;

wire     [2:0]                      S_MCmd;
wire     [addr_wdth -1:0] S_MAddr;
wire     [data_wdth -1:0] S_MData, S_SData;
wire     [burstlength_wdth -1:0]    S_MBurstLength;
wire     [2:0]                      S_MBurstSeq;
wire     [Sthreadid_wdth -1:0]      S_MThreadID, S_MDataThreadID, S_SThreadID;
wire     [1:0]                      S_SResp;

wire     [data_wdth -1:0]           yy;
wire     [Mthreadid_wdth -1:0]      qq;

OCP_Master #(
    .data_file ( "./DATA/OCP_Master1.in" ),
    .id ( "master1"))
  Master1(
    .OCPClk( clk_master1 ),
    .Reset_n( rst_master1 ),
    .OCPMCmd( M1_MCmd ),
//    .M_OCPSReset_n( rst_master  ),
    .OCPSCmdAccept( M1_SCmdAccept ),
    .OCPMAddr( M1_MAddr ),
    .OCPMData( M1_MData ),
    .OCPMBurstLength( M1_MBurstLength ),
    .OCPMBurstSeq( M1_MBurstSeq ),
    .OCPMBurstSingleReq( M1_MBurstSingleReq ),
    .OCPMBurstPrecise( M1_MBurstPrecise ),
    .OCPMReqLast( M1_MReqLast ),
    .OCPMDataLast( M1_MDataLast ),
    .OCPMConnID( M1_MConnID ),
    .OCPMThreadID( M1_MThreadID ),
    .OCPMDataValid( M1_MDataValid ),
    .OCPSDataAccept( M1_SDataAccept ),
    .OCPSResp( M1_SResp ),
    .OCPMRespAccept( M1_MRespAccept ),
    .OCPSData( M1_SData ),
    .OCPSRespLast( M1_SRespLast ),
    .OCPSThreadID( M1_STrheadID ),
    .OCPMDataThreadID( M1_MDataThreadID ),
    .OCPSInterrupt( M1_SInterrupt )
);

OCP_Master #(
    .data_file ( "./DATA/OCP_Master2.in" ),
    .id ( "master2"))
  Master2(
      .OCPClk( clk_master2 ),
      .Reset_n( rst_master2 ),
      .OCPMCmd( M2_MCmd ),
//    .M_OCPSReset_n( rst_master  ),
      .OCPSCmdAccept( M2_SCmdAccept ),
      .OCPMAddr( M2_MAddr ),
      .OCPMData( M2_MData ),
      .OCPMBurstLength( M2_MBurstLength ),
      .OCPMBurstSeq( M2_MBurstSeq ),
      .OCPMBurstSingleReq( M2_MBurstSingleReq ),
      .OCPMBurstPrecise( M2_MBurstPrecise ),
      .OCPMReqLast( M2_MReqLast ),
      .OCPMDataLast( M2_MDataLast ),
      .OCPMConnID( M2_MConnID ),
      .OCPMThreadID( M2_MThreadID ),
      .OCPMDataValid( M2_MDataValid ),
      .OCPSDataAccept( M2_SDataAccept ),
      .OCPSResp( M2_SResp ),
      .OCPMRespAccept( M2_MRespAccept ),
      .OCPSData( M2_SData ),
      .OCPSRespLast( M2_SRespLast ),
      .OCPSThreadID( M2_STrheadID ),
      .OCPMDataThreadID( M2_MDataThreadID ),
      .OCPSInterrupt( M2_SInterrupt )
);

OCP_Slave #(
      .id( "Slave1"))
  Slave1(
      .OCPClk( clk_slave ),
      .Reset_n( rst_slave ),
      .OCPMCmd( S_MCmd ),
//    .S_OCPMReset_n( S_MReset ),
      .OCPSCmdAccept( S_SCmdAccept ),
      .OCPMAddr( S_MAddr ),
      .OCPMData( S_MData ),
      .OCPMDataValid( S_MDataValid ),
      .OCPSDataAccept( S_SDataAccept ),
      .OCPMBurstLength( S_MBurstLength ),
      .OCPMBurstSeq( S_MBurstSeq ),
      .OCPMBurstSingleReq( S_MBurstSingleReq ),
      .OCPMBurstPrecise( S_MBurstPrecise ),
      .OCPMReqLast( S_MReqLast ),
      .OCPMDataLast( S_MDataLast ),
      .OCPMThreadID( S_MThreadID ),
      .OCPMDataThreadID( S_MDataThreadID ),
      .OCPSRespLast( S_SRespLast ),
      .OCPSThreadID( S_SThreadID ),
      .OCPSResp( S_SResp ),
      .OCPSData( S_SData ),
      .OCPMRespAccept( S_MRespAccept ),
      .OCPRespDone( S_RespDone ),
      .OCPSInterrupt( S_SInterrupt )
);

initiator initiator_NA1(
      .OCPClk( clk_master1 ),
      .Reset_n( rst_master1 ),

      // initiator request module interface
      .OCPMCmd_i( M1_MCmd ),
      .OCPSReset_no( ),          // leave this hanging.. its looped back from Reset_n...
      .OCPSCmdAccept_o( M1_SCmdAccept ),
      .OCPMAddr_i( M1_MAddr ),
      .OCPMData_i( M1_MData ),
      .OCPMBurstLength_i( M1_MBurstLength ),
      .OCPMBurstSeq_i( M1_MBurstSeq ),
      .OCPMBurstSingleReq_i( M1_MBurstSingleReq ),
      .OCPMBurstPrecise_i( M1_MBurstPrecise ),
      .OCPMReqLast_i( M1_MReqLast ),
      .OCPMDataLast_i( M1_MDataLast ),
      .OCPMConnID_i( M1_MConnID ),
      .OCPMThreadID_i( M1_MThreadID ),
      .OCPMDataValid_i( M1_MDataValid ),
      .OCPSDataAccept_o( M1_SDataAccept ),

      // initiator response module interface
      .OCPSResp_o( M1_SResp ),
      .OCPMRespAccept_i( M1_MRespAccept ),
      .OCPSData_o( M1_SData ),
      .OCPSRespLast_o( M1_SRespLast ),
      .OCPSThreadID_o( M1_STrheadID ),
      .OCPSResp_i( ),                           // unwarranted port! clean it up!
      .OCPSData_i( ),                           // unwarranted port! clean it up!
      .OCPSRespLast_i( ),                       // unwarranted port! clean it up!
      .OCPSThreadID_i( ),                       // unwarranted port! clean it up!
      .OCPMDataThreadID_i( M1_MDataThreadID ),
      .OCPSInterrupt_o( M1_SInterrupt ),

      // initiator network interface
      .RxReq_i( RxReq_1 ),
      .RxAck_o( RxAck_1 ),
      .RxData_i( RxData_1 ),
      .TxAck_i( TxAck_1 ),
      .TxReq_o( TxReq_1 ),
      .TxData_o( TxData_1 )
);

initiator initiator_NA2(
      .OCPClk( clk_master2 ),
      .Reset_n( rst_master2 ),

      // initiator request module interface
      .OCPMCmd_i( M2_MCmd ),
      .OCPSReset_no( ),           // leave this hanging.. its looped back from Reset_n...
      .OCPSCmdAccept_o( M2_SCmdAccept ),
      .OCPMAddr_i( M2_MAddr ),
      .OCPMData_i( M2_MData ),
      .OCPMBurstLength_i( M2_MBurstLength ),
      .OCPMBurstSeq_i( M2_MBurstSeq ),
      .OCPMBurstSingleReq_i( M2_MBurstSingleReq ),
      .OCPMBurstPrecise_i( M2_MBurstPrecise ),
      .OCPMReqLast_i( M2_MReqLast ),
      .OCPMDataLast_i( M2_MDataLast ),
      .OCPMConnID_i( M2_MConnID ),
      .OCPMThreadID_i( M2_MThreadID ),
      .OCPMDataValid_i( M2_MDataValid ),
      .OCPSDataAccept_o( M2_SDataAccept ),

      // initiator response module interface
      .OCPSResp_o( M2_SResp ),
      .OCPMRespAccept_i( M2_MRespAccept ),
      .OCPSData_o( M2_SData ),
      .OCPSRespLast_o( M2_SRespLast ),
      .OCPSThreadID_o( M2_STrheadID ),
      .OCPSResp_i( ),                           // unwarranted port! clean it up!
      .OCPSData_i( ),                           // unwarranted port! clean it up!
      .OCPSRespLast_i( ),                       // unwarranted port! clean it up!
      .OCPSThreadID_i( ),                       // unwarranted port! clean it up!
      .OCPMDataThreadID_i( M2_MDataThreadID ),
      .OCPSInterrupt_o( M2_SInterrupt ),

      // initiator network interface
      .RxReq_i( RxReq_2 ),
      .RxAck_o( RxAck_2 ),
      .RxData_i( RxData_2 ),
      .TxAck_i( TxAck_2 ),
      .TxReq_o( TxReq_2 ),
      .TxData_o( TxData_2 )
);


async_reset ctrl(
   .rst( rst)
);

MANGO_router_thesis_test_3port network(
// test port 1
     .RxReq_1( RxReq_1 ),
     .RxAck_1( RxAck_1 ),
     .RxData_1( RxData_1 ),
     .TxAck_1( TxAck_1 ),
     .TxReq_1( TxReq_1 ),
     .TxData_1( TxData_1 ),
// test port 2
     .RxReq_2( RxReq_2 ),
     .RxAck_2( RxAck_2 ),
     .RxData_2( RxData_2 ),
     .TxAck_2( TxAck_2 ),
     .TxReq_2( TxReq_2 ),
     .TxData_2( TxData_2 ),
// test port 3
     .RxReq_3( RxReq_3 ),
     .RxAck_3( RxAck_3 ),
     .RxData_3( RxData_3 ),
     .TxAck_3( TxAck_3 ),
     .TxReq_3( TxReq_3 ),
     .TxData_3( TxData_3 ),
     .reset( rst )
);

target target_NA(
        .OCPClk( clk_slave ),
        .Reset_n( rst_slave ),

        // OCP interface
        .OCPMCmd_o( S_MCmd ),
        .OCPMReset_no( S_MReset ),
        .OCPSCmdAccept_i( S_SCmdAccept ),
        .OCPMAddr_o( S_MAddr ),
        .OCPMData_o( S_MData ),
        .OCPMDataValid_o( S_MDataValid ),
        .OCPSDataAccept_i( S_SDataAccept ),
        .OCPMBurstLength_o( S_MBurstLength ),
        .OCPMBurstSeq_o( S_MBurstSeq ),
        .OCPMBurstSingleReq_o( S_MBurstSingleReq ),
        .OCPMBurstPrecise_o( S_MBurstPrecise ),
        .OCPMReqLast_o( S_MReqLast ),
        .OCPMDataLast_o( S_MDataLast ),
        .OCPMThreadID_o( S_MThreadID ),
        .OCPMDataThreadID_o( S_MDataThreadID ),
        .OCPSRespLast_i( S_SRespLast ),
        .OCPSThreadID_i( S_SThreadID ),
        .OCPSResp_i( S_SResp ),
        .OCPSData_i( S_SData ),
        .OCPMRespAccept_o( S_MRespAccept ),
        .OCPRespDone_i( S_RespDone ),
        .OCPSInterrupt_i( S_SInterrupt ),

        // Network interface
        .RxReq_i( RxReq_3 ),
        .RxAck_o( RxAck_3 ),
        .RxData_i( RxData_3 ),
        .TxAck_i( TxAck_3 ),
        .TxReq_o( TxReq_3 ),
        .TxData_o( TxData_3 )
);

endmodule
```

C.7 OCP Master

```
--------------------------------------------------------------------------------
-- Title       : OCP Master emulator
-- Project     : MANGO
--------------------------------------------------------------------------------
-- File        : OCP_Master.vhd
-- Author      : Christian Place Pedersen
--               http://christianplace.dk
-- Company     : Technical University of Denmark - IMM/CSE
-- Created     : 2006/08/21
-- Last update : 2006/09/15
-- Platform    :
-- Standard    : VHDL'93
--------------------------------------------------------------------------------
-- Description:
--------------------------------------------------------------------------------
-- Copyright (c) 2006
--------------------------------------------------------------------------------
-- Revisions  :
-- Date         Version  Author  Description
-- 2005/08/21   1.0      CPP     Created
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_textio.all;
-- use ieee.std_logic_unsigned.all;
use std.textio.all;
use work.my_misc.all;

entity OCP_Master is

  generic (
    addr_wdth : integer := 32;
    data_wdth : integer := 32;
    burstlength_wdth : integer := 8;
    connid_wdth : integer := 2;
    threadid_wdth : integer := 3;
    clk_period : time := 4 ns;
    data_file : string  := "./DATA/OCP_master1.in";
    id        : string  := "Master");

  port (
    OCPClk                : out std_logic;
    Reset_n               : out std_logic;
    OCPMCmd               : out std_logic_vector(2 downto 0);
    --OCPSReset_n         : in  std_logic;
    OCPSCmdAccept         : in  std_logic;
    OCPMAddr              : out std_logic_vector(addr_wdth-1 downto 0);
    OCPMData              : out std_logic_vector(data_wdth-1 downto 0);
    OCPMBurstLength       : out std_logic_vector(burstlength_wdth-1 downto 0);
    OCPMBurstSeq          : out std_logic_vector(2 downto 0);
    OCPMBurstSingleReq    : out std_logic;
    OCPMBurstPrecise      : out std_logic;
    OCPMReqLast           : out std_logic;
    OCPMDataLast          : out std_logic;
    OCPMConnID            : out std_logic_vector(connid_wdth-1 downto 0);
    OCPMThreadID          : out std_logic_vector(threadid_wdth-1 downto 0);
    OCPMDataValid         : out std_logic;
    OCPSDataAccept        : in  std_logic;
    OCPSResp              : in  std_logic_vector(1 downto 0);
    OCPMRespAccept        : out std_logic;
    OCPSData              : in  std_logic_vector(data_wdth-1 downto 0);
    OCPSRespLast          : in  std_logic;
    OCPSThreadID          : in  std_logic_vector(threadid_wdth-1 downto 0);
    OCPMDataThreadID      : out std_logic_vector(threadid_wdth-1 downto 0);
    OCPSInterrupt         : in  std_logic);

end OCP_Master;

architecture emulator of OCP_Master is

```

```
    file input : text open read_mode is data_file;
    constant CH0IN_SPEED : time := 0.4 ns;
    constant CH0OUT_SPEED : time := 0.2 ns;
    constant CH0_ACTIVITY : time := 0.1 ns;
    signal clk : std_logic;

  begin  -- emulator

    -- purpose: Master clock
    process
    begin  -- process
      loop
        clk <= '1';
        wait for clk_period;
        clk <= '0';
        wait for clk_period;
      end loop;
    end process;

    OCPClk <= clk;

    process                                    -- Transmitter
      variable l : line;
      variable MCmd : std_logic_vector(2 downto 0);
      variable MConnID : std_logic_vector(connid_wdth-1 downto 0);
      variable MThreadID : std_logic_vector(threadid_wdth-1 downto 0);
      variable MAddr : std_logic_vector(addr_wdth-1 downto 0);
      variable MData : std_logic_vector(data_wdth-1 downto 0);
    begin  -- process
      wait for 3 ns;
      OCPMCmd <= (others => '0');
      OCPMAddr <= (others => '0');
      OCPMData <= (others => '0');
      OCPMBurstLength <= "00000001";
      OCPMBurstSeq <= (others => '0');
      OCPMBurstSingleReq <= '0';
      OCPMBurstPrecise <= '0';
      OCPMReqLast <= '0';
      OCPMDataLast <= '0';
      OCPMConnID <= (others => '0');
      OCPMThreadID <= (others => '0');
      OCPMDataValid <= '0';
      OCPMRespAccept <= '0';
      OCPMDataThreadID <= (others => '0');

      -- Initiate reset cycle
      Reset_n <= '1';
      wait until clk = '0';
      wait until clk = '1';
      wait for CH0IN_SPEED;
      Reset_n <= '0';
      report ID & "␣Activating␣Reset" severity NOTE;
      -- Wait some clk cycles
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
      wait until clk = '1';
      wait until clk = '0';
    wait until clk = '1';
    wait until clk = '0';
    wait until clk = '1';
    wait for CH0IN_SPEED;
    Reset_n <= '1';
    report ID & "␣Deactivating␣Reset" severity NOTE;


    wait until clk = '0';
    wait until clk = '1';
    wait until clk = '0';
    wait until clk = '1';
    wait until clk = '0';
    wait until clk = '1';
    wait for CH0IN_SPEED;

   -- Start the test, read the input file
    while not endfile(input) loop
      readline(input, l);
      -- Each line has the following format (one whitespace as seperator):
      -- OCPMCmd OCPMConnID OCPMAddr OCPMData

      read(l,MCmd);                         -- First we read the MCmd
      -- And find out if we are IDLE(000), WRITE(001) or READ(010)
      -- Set up signals by reading values from the test file
      read(l,MConnID);
      read(l,MThreadID);
      read(l,MAddr);
      read(l,MData);
      case MCmd is
        when "000" =>                       -- IDLE
          report ID & "␣is␣IDLE" severity NOTE;
          OCPMCmd <= "000";
          wait until rising_edge(clk);
          wait for CH0IN_SPEED;
        when "001" =>                       -- WRITE
          report ID & ":␣Writing"
            & "␣ConnID:␣"   & to_string(MConnID)
            & "␣ThreadID:␣" & to_string(MThreadID)
            & "␣MAddr:␣"    & to_string(MAddr)
            & "␣MData:␣"    & to_string(MData)
            severity NOTE;
          OCPMCmd <= MCmd;
          OCPMConnID <= MConnID;
          OCPMThreadID <= MThreadID;
          OCPMAddr <= MAddr;
          OCPMData <= MData;
          -- Handshake
          report ID & "␣Waiting␣for␣SCmdAccept" severity NOTE;
          wait until OCPSCmdAccept = '1' and rising_edge(clk);
          report ID & "␣Got␣SCmdAccept␣going␣IDLE" severity NOTE;
          wait for CH0IN_SPEED;
          OCPMCmd    <= "000";
        when "010" =>
          report ID & ":␣Reading"
            & "␣ConnID:␣"   & to_string(MConnID)
            & "␣ThreadID:␣" & to_string(MThreadID)
            & "␣Address:␣"  & to_string(MAddr)
            severity NOTE;
          OCPMCmd <= MCmd;
          OCPMConnID <= MConnID;
          OCPMThreadID <= MThreadID;
          OCPMAddr <= MAddr;
          -- Handshake
          report ID & "␣Waiting␣for␣SCmdAccept" severity NOTE;
          wait until OCPSCmdAccept = '1' and rising_edge(clk);
          report ID & "␣Got␣SCmdAccept␣going␣IDLE" severity NOTE;
          wait for CH0IN_SPEED;
          OCPMCmd <= "000";                 --IDLE
          report ID & "␣Waiting␣for␣SResp" severity NOTE;
          wait until OCPSResp = "01" and rising_edge(clk);
          report ID & "␣Got␣SResp" severity NOTE;
          OCPMRespAccept <= '1';
          -- We are assuming single response
```

```
            report ID & "␣Recieved␣data:␣" & to_string(OCPSData) severity NOTE;
            wait until OCPSResp = "00" and rising_edge(clk);
            OCPMRespAccept <= '0';
          when "101" =>
            report ID & ":␣WRNP"
              & "␣ConnID:␣"   & to_string(MConnID)
              & "␣ThreadID:␣" & to_string(MThreadID)
              & "␣Address:␣"  & to_string(MAddr)
              & "␣MData:␣"    & to_string(MData)
             severity NOTE;
            OCPMCmd <= MCmd;
            OCPMConnID <= MConnID;
            OCPMThreadID <= MThreadID;
            OCPMAddr <= MAddr;
            OCPMData <= MData;
            -- Handshake
            report ID & "␣Waiting␣for␣SCmdAccept" severity NOTE;
            wait until OCPSCmdAccept = '1' and rising_edge(clk);
            report ID & "␣Got␣SCmdAccept␣going␣IDLE" severity NOTE;
            wait for CH0IN_SPEED;
            OCPMCmd <= "000";                    --IDLE
            report ID & "␣Waiting␣for␣SResp" severity NOTE;
            wait until OCPSResp = "01" and rising_edge(clk);
            report ID & "␣Got␣SResp" severity NOTE;
            OCPMRespAccept <= '1';
            wait until OCPSResp = "00" and rising_edge(clk);
            OCPMRespAccept <= '0';
          when others => report ID & "␣BAD␣MCmd␣(end␣of␣file␣perhaps?)"
                           severity NOTE;
            report ID & "␣is␣IDLE" severity NOTE;
            OCPMCmd <= "000";
            wait until rising_edge(clk);
            wait for CH0IN_SPEED;
        end case;
      end loop;
      wait;
      report ID & "␣End␣of␣file" severity FAILURE;
      loop
        report ID & "␣is␣IDLE" severity NOTE;
        OCPMCmd <= "000";
        wait until rising_edge(clk);
        wait for CH0IN_SPEED;
      end loop;
    end process;

end emulator;
```

C.8 OCP Slave

```
---------------------------------------------------------------------------
-- Title      : OCP Slave emulator
-- Project    : MANGO
---------------------------------------------------------------------------
-- File       : OCP_Slave.vhd
-- Author     : Christian Place Pedersen
--              http://christianplace.dk
-- Company    : Technical University of Denmark - IMM/CSE
-- Created    : 2006/08/22
-- Last update: 2006/09/16
-- Platform   :
-- Standard   : VHDL'93
---------------------------------------------------------------------------
-- Description:
---------------------------------------------------------------------------
-- Copyright (c) 2006
---------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2005/08/22  1.0      CPP     Created
---------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_textio.all;
-- use ieee.std_logic_unsigned.all;
use std.textio.all;
use work.my_misc.all;

entity OCP_Slave is

  generic (
    addr_wdth        : integer := 32;
    data_wdth        : integer := 32;
    ocp_data_wdth    : integer := 32;
    burstlength_wdth : integer := 8;
    threadid_wdth    : integer := 2;
    clk_period       : time := 7 ns;
    data_file        : string  := "OCP_Slave.in";
    id               : string  := "OCP_Slave");
  port (
    OCPClk              : out std_logic;
    Reset_n             : out std_logic;
    OCPMCmd             : in  std_logic_vector(2 downto 0);
    --OCPMReset_ni        : in  std_logic;
    OCPSCmdAccept       : out std_logic;
    OCPMAddr            : in  std_logic_vector(addr_wdth-1 downto 0);
    OCPMData            : in  std_logic_vector(ocp_data_wdth-1 downto 0);
    OCPMDataValid       : in  std_logic;
    OCPSDataAccept      : out std_logic;
    OCPMBurstLength     : in  std_logic_vector(burstlength_wdth-1 downto 0);
    OCPMBurstSeq        : in  std_logic_vector(2 downto 0);
    OCPMBurstSingleReq  : in  std_logic;
    OCPMBurstPrecise    : in  std_logic;
    OCPMReqLast         : in  std_logic;
    OCPMDataLast        : in  std_logic;
    OCPMThreadID        : in  std_logic_vector(threadid_wdth-1 downto 0);
    OCPMDataThreadID    : in  std_logic_vector(threadid_wdth-1 downto 0);
    OCPSRespLast        : out std_logic;
    OCPSThreadID        : out std_logic_vector(threadid_wdth-1 downto 0);
    OCPSResp            : out std_logic_vector(1 downto 0);
    OCPSData            : out std_logic_vector(ocp_data_wdth-1 downto 0);
    OCPMRespAccept      : in  std_logic;
    OCPRespDone         : out std_logic;
    OCPSInterrupt       : out std_logic);

end OCP_Slave;

architecture emulator of OCP_Slave is

  type STATE is (s0,s1);
  signal CurrentState, NextState : STATE;
  constant CH0IN_SPEED : time := 0.4 ns;
  constant CH0OUT_SPEED : time := 0.2 ns;
  constant CH0_ACTIVITY : time := 0.1 ns;
  signal clk, reset : std_logic;
  signal read_data : std_logic_vector(data_wdth-1 downto 0);

begin -- emulator

  -- purpose: Master clock
  process
  begin -- process
    loop
      clk <= '1';
      wait for clk_period;
      clk <= '0';
      wait for clk_period;
    end loop;
  end process;

  OCPClk <= clk;

  -- reset cycle
  process
  begin -- process
    Reset <= '1';
    wait until clk = '0';
    wait until clk = '1';
    wait for CH0IN_SPEED;
    Reset <= '0';
    report ID & "␣Activating␣Reset" severity NOTE;
   -- Wait some clk cycles
    wait until clk = '0';
    wait until clk = '1';
    wait until clk = '0';
    wait until clk = '1';
    wait until clk = '0';
    wait until clk = '1';
    wait for CH0IN_SPEED;
    Reset <= '1';
    report ID & "␣Deactivating␣Reset" severity NOTE;
    wait;
  end process;

  Reset_n <= Reset;

  next_state: process (clk, Reset)
  begin -- process
    if Reset = '0' then                  -- asynchronous reset (active low)
      CurrentState <= s0;
    elsif clk'event and clk = '1' then   -- rising clock edge
      CurrentState <= NextState;
    end if;
  end process;                           -- next_state

  fsm_logic: process (OCPMCmd, OCPMAddr, OCPMData,
                      OCPMDataValid, OCPMBurstLength, OCPMBurstSeq,
                      OCPMBurstSingleReq, OCPMBurstPrecise, OCPMReqLast,
                      OCPMDataLast, OCPMThreadID, OCPMDataThreadID,
                      OCPMRespAccept, CurrentState)
  begin -- process fsm_logic

    case CurrentState is
      when s0 =>                         -- Single request
        OCPSData <= (others => '0');
        OCPSRespLast <= '0';
        OCPSResp <= (others => '0');
        case OCPMCmd is
          when "000" =>                  -- Idle
            NextState <= s0;
            OCPSCmdAccept <= '0';
            OCPSDataAccept <= '0';
            OCPRespDone <= '0';
            OCPSInterrupt <= '0';
          when "001" =>                  -- Write
```

```
              -- only singe request so far
              OCPSCmdAccept <= '1';
              OCPSDataAccept <= '1';
              NextState <= s0;
              report ID & " Write to"
                & " Address: " & to_string(OCPMAddr)
                & " Data: "    & to_string(OCPMData)
                severity NOTE;
            when "010" =>                  -- Read
              OCPSThreadID <= OCPMThreadID;
              OCPSCmdAccept <= '1';
              NextState <= s1;
              read_data <= OCPMAddr;
              report ID & " read of"
                & " address " & to_string(OCPMAddr)
                severity NOTE;
            when others => null;
          end case;
        when s1 =>
          OCPSCmdAccept <= '1';
          --OCPSDataAccept <= '1';
          OCPSResp <= "01";
          OCPSRespLast <= '1';
          OCPSData <= read_data;   -- return the address as data
          OCPSCmdAccept <= '0';
          if OCPMRespAccept = '1' then
            NextState <= s0;
          else
            NextState <= s1;
          end if;
        when others =>
          NextState <= s0;
      end case;
    end process fsm_logic;

end emulator;
```