Faculty of Computer Science
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. 2813

FPGA Emulation of a GALS Network-on-Chip Interconnection

Alejandro Cook

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Hans-Joachim Wunderlich
Supervisors:	Dipl.-Inform. Melanie Elm
	Dipl.-Inf. Stefan Holst
Commenced:	July 04 2008
Completed:	February 03 2009
CR-Classification:	C.1.4

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

Chapter 1

Introduction

1.1 Network-on-Chip architecture motivation and goals

The periodic advances in semiconductor technology have made it possible to increase chip density and build ever larger VLSI systems. As the number of components in such a system grows beyond a certain limit [ZKCS02], the traditional bus becomes more difficult to implement and creates a bottleneck that slows down the entire system. This problem stems from the main idea behind a bus-based communication strategy: a common medium shared by all initiators and targets that, by definition, has to be spread across the entire chip.

From a performance point of view, the bus does not scale up well enough since an increasing number of initiators try to gain access to a shared resource and lock it for the duration of a full data transfer. This potentially leads to large bus contention and, as a result, the system throughput may become unacceptably low.

Several architectural optimizations have been used to overcome the inherent limitations of bus-based communication strategies. For example, the use of full crossbars may alleviate contention problems, given that two initiators are now able to communicate with two different targets concurrently. A well-designed bus hierarchy may also be a solution in some situations; it isolates two or more regions in the bus, so that the traffic in one region does not interfere with the traffic in another region as long as the traffic does not cross a hierarchy boundary. Although these solutions have been put to good use in the past, it is unlikely that they will support the clear trend towards parallel processing on Multi-Processor System on Chip (MPSoC) in coming years. On the one hand, crossbars are prohibitively expensive for even a moderate number of processing units [HN06], and on the other hand, a bus hierarchy requires some customization to make it suitable for the communication pattern of any given system [EEL07]. Therefore, it comes at a high price in terms of design reuse and time to market.

Networks on Chip (NoC) have been recognized as a promising strategy to interconnect the processing units in a large system [BB03, GDR05, VLC+07]. The key idea is to have several switches with a given number of ports, and direct connections between the ports of two adjacent switches. The processing unit is extended with a Network Interface (NI) to access the network and, in turn, the NI is connected to the switch via a dedicated port. The switch uses a crossbar to enable the flow of data to and from the ports and some buffers to improve latency and throughput.

The NoC paradigm does not incur in the area overhead of a full crossbar because the number of ports in a switch can be kept small. NoCs are scalable and modular; therefore more switches can be added in the network without major architectural redesigns. The data flow can be routed to its destination independently from the number of nodes in the network and with a lower performance degradation than that in a global arbitration scheme.

Networks on Chip make use of many of the key concepts found in wide area computer networks but their implementation is additionally constrained by challenges in chip manufacture. This is why, the most common and well known network protocols do not have a direct counterpart at chip level. Much research effort has been recently devoted to assess the trade-offs and design choices relevant to the Network-on-chip communication paradigm. They include traditional issues like latency and throughput, but this time much closer to the processor level, and some new challenges like power dissipation, wire delay, reusability and yield.

1.2 Globally asynchronous locally synchronous challenges

Even though technology scaling has made it possible to fit more devices into a single die, scalability and modularity continue to be an issue at the physical level [KGGV07]. Specifically, designing a clock network to control all the blocks in a chip has become increasingly complex. In addition, wire delay in today's manufacturing process dominates circuit speed, making the total delay over very long links unacceptable. This poses a problem for new designs since the desired behavior may no longer be obtained by aggregating pre-verified functional blocks.

Globally Asynchronous Locally Synchronous (GALS) systems use an asynchronous interconnect and thus decouple the timing issues of the independent synchronous blocks. Each of these synchronous islands can be developed using standard CAD tools and, for their interconnection, a special wrapper circuit is developed [KGGV07]. The use of a wrapper circuit enables design reuse and eases system integration, that is to say, a specialized functional block can be aggregated in the design with minimal design effort. Also, the use of separate clock domains in GALS systems lends itself well to power saving techniques, like frequency and voltage scaling [KGGV07].

The main challenge in the development of GALS systems is designing the wrappers or interfaces between two synchronous blocks in order to enable the flow of data across clock domains. If the input of a latch or flip-flop is driven in a different clock domain, the setup and hold times are no longer guaranteed by design, and an indeterminate voltage level may appear at the output for an unbounded amount of time. To avoid this condition, known as metastability, special blocks or synchronizers have to be in place to avoid occurrence, or at least reduce the probability, of a synchronization failure.

Given the heterogeneous nature of today's SoC and the previously mentioned physical problems in the design of complex systems, a GALS approach is very well suited for the interconnection of the potentially large number of processing units present in a NoC. In this regard, not only the processing elements can belong to different clock domains, but also the switches in the interconnection matrix may be driven by clocks that exhibit different timing relationships.

1.3 Emulation of a NoC on an FPGA

In recent years, NoCs have become a very attractive topic in the research community, and several GALS communication platforms have been developed. However, the network performance must traditionally be derived from theoretical models and simulation environments before any prototype can be built. In order to explore the NoC design space and enable functional validation under the requirements of a real application domain, an emulation approach becomes necessary. The aim of this Master Thesis is to assist the performance evaluation of the interconnection matrix of a large Network-on-Chip by using an multi-FPGA emulation platform. Current FPGAs are big enough to hold several processing units and prototype a network consisting of many nodes under very realistic conditions while obtaining a significant speedup when compared to a simulation experiment.

With this objective in mind, the VASH framework was developed. VASH comprises a set of hardware blocks that make up a simple, synthesizable NoC in a GALS environment, and a set of software tools to automate the generation of a NoC description, and fine-tune its parameters. VASH aims to serve as a development platform to functionally evaluate some of the concepts behind the most common NoC architectures.

In an effort to achieve a high degree of flexibility when exploring the the NoC design space, the VASH framework has been implemented as an open source platform in a high level programming language. Special care has been taken in the design of the framework, both on the software and the hardware side. It should be able to measure some relevant figures of merit and yet be extendable enough to support the architectural changes needed for a particular application. In order to account for some of the problems in current deep submicron technologies, VASH supports a mesochronous clocking scheme (all switches in the network operate at exactly the same frequency with a fixed phase difference) and implements two different synchronization strategies found in the literature, namely, a simple double flip-flop synchronizer and a lower latency latch-based synchronizer [LAB08].

The network hardware blocks in VASH include a network switch, a network interface (NI), a packet generator (PG) and a node controller (NC). The network switch is a modified version of a freely available switch form the University of Rostock[Imd]. It uses wormhole switching, X-Y routing, and has no virtual channel support. The packet generator injects packets into the network in order to create traffic. The network interface sends and receives packets to and from the local switch, and, finally, the node controller monitors the behavior of a node and communicates with a host PC.

The NoC components were validated using the *CHIPit Platinum* emulation system from *ProDesign*. This platform contains six large Xilinx Virtex-II FPGAs and includes the necessary software tools for the seamless integration of the hardware blocks in multiple FPGAs. In order to configure and monitor the network nodes from the host PC, the *UMR* bus, a proprietary protocol available from the emulation machine manufacturer, has been utilized.

Chapter 2

Literature review

2.1 Network-on-chip architecture overview

The recent interest in the NoC communication paradigm has been fostered by the increasing complexity in current systems-on-chip. NoCs aim to abstract away some of the projected physical issues in silicon technology; they achieve greater performance by employing a system-level view of on-chip communication.

The technology predictions in coming years envision the gradual increase in operating frequency and transistor density. This, however, makes energy consumption a major design consideration. Similarly, the propagation delay of global wires are likely to exceed one clock period. This issue can be alleviated by using pipelined interconnections, but they require the accurate estimation of all delay paths. Now, as a consequence of technology downsizing, the spreading of the physical parameter makes the exact wire delays difficult to determine. This variance in the delay may be a major issue when designing high-performance chips unless a worst-case timing analysis is used. However, this type of analysis may not utilize the underlying technology to its full potential.

Signal integrity and system reliability are also key design issues nowadays. On the one hand, signal integrity, the ability to transfer data without errors, is becoming harder to achieve at the physical level. This comes as a consequence of a reduced voltage swing and the complexity and ubiquitous nature of current integrated circuits. The reduced voltage swings, and their corresponding drop in noise margins, make the systems more vulnerable to electromagnetic interference by external sources and soft errors produced by the collision of thermal neutrons and alpha particles. Likewise, the additional complexity of future Systems-on-chip (SoCs) limits the efficiency of a thorough crosstalk analysis. Reliability, on the other hand, is the probability of a system to work correctly during a given period of time. SoCs have found their way into new application domains with critical safety constraints. They must, therefore, be designed to tolerate both permanent and transient malfunctions. Such systems should support a robust interconnection scheme and enable the seamless addition and removal of components, as well as a systematic approach for error detection, containment and correction distributed across the entire protocol stack. Such a layered approach to on-chip communication is useful when dealing with the inherent non-determinism that arises from the variability of the physical parameters. The advantages are twofold: firstly, the complexity of a design can be better managed and modeled in different abstraction

levels and, secondly, the eventual communication errors can be corrected at the logic or functional level. This makes it possible to relax the physical and electrical requirements .

NoCs offer a promising solution to the on-chip communication problem as they inherently show a layered structure to realize on-chip interconnections. In the following sections, a NoC protocol stack is briefly explained as presented in [MB06]. This layered architecture, as shown in Figure 2.1, resembles the familiar OSI model[Bla91] to characterize macroscopic networks. The rest of this chapter contains the definitions and concepts relevant to the present work.

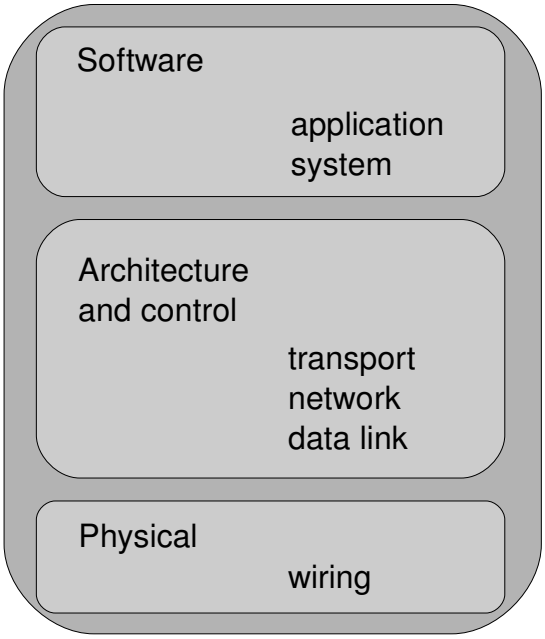

Figure 2.1: Micro-network stack [MB06]

2.1.1 Physical layer

This layer deals with the physical implementation of global wires as communication channels. In order to reduce power dissipation and retain or improve the speed of the communication channel, reduced voltage swing and current mode transmission are likely to be preferred over traditional rail-to-rail voltage signaling solutions. However, the trend to use a lower voltage level may also decrease communication reliability given that signal integrity is also compromised. NoCs favor error contention, detection and correction over stringent physical design. That is, they allow the occurrence of the unavoidable communication error and take appropriate measures, (Error Correcting Codes, for example) in higher levels of the protocol stack.

2.1.2 Data link layer

The data link layer hides the reliability issues in the physical link and provides the error-handling mechanism for safe on-chip data transfer. Besides the errors due to the inaccuracies in the manufacturing process, synchronization failures and contention resolution should also be dealt with in this layer. NoCs can employ Error Correcting Codes (ECC) to reach a minimum required communication reliability. However, their use is a major design consideration given their impact on power consumption.

2.1.3 Network and transport layers

The goal of the *network layer* is to discover the path the data has to travel to reach its final destination. In addition, this layer also deals with the information data flow between neighbor nodes in the network. In case of a *packet-switched* network, the choice of the *switching* and *routing* algorithms further specify the behavior of the data transmission. The routing algorithms specify where to send a packet from one node to the next in the path towards its destination. Broadly speaking, they can be classified as *deterministic* and *adaptive* algorithms. The former provide the same path every time the same two nodes need to exchange information in the network, while the latter monitor some network conditions, like data traffic and node congestion, in order to set up a more efficient communication route. This routing strategy is better suited for irregular networks with unpredictable data traffic.

The switching algorithms, on the other hand, control the way the data units that make up a packet are stored in the intermediate nodes along the packet route. The most common switching techniques used in NoCs are *store-and-forward* (SAF), *virtual cut-through* (VCT) and *wormhole*. As explained later in this chapter, they offer different trade-offs in terms of complexity, latency and minimum storage capacity.

The *transport layer* deals with the decomposing messages into packets and then further breaking them into smaller transmission units that flow across the network. At the receiving end, these units need to be reassembled to recover the intended message. At this layer, the most important design considerations include the packet size, the flow control mechanism to acknowledge the reception of a data unit, and the congestion control scheme to prevent network saturation in case of a slow or unresponsive network node.

2.1.4 Software layers

The *software layers* in the NoC domain comprises the system software and the application software. The system software provides an application program with an API to abstract away the hardware details in NoC communication. For this purpose, it is common practice to extend the hardware of a processing core with a *network interface* (NI) to gain access to the services provided by the network. The application software must be designed in such a way as to exploit the parallelism in the application domain, and effectively map the resulting instructions streams and communication patterns into a NoC configuration.

2.2 NoC topology

The network topology defines how the nodes are interconnected. In the context of a NoC, the network topology also influences how the nodes are placed on the chip. The most common topologies for NoC are explained in the following subsections.

2.2.1 Crossbar

As shown in Figure 2.2a, a *crossbar*[LLY03] is used when all routers are connected to all other routers in the network. Its implementation cost is prohibitively high even for a moderate number of processing units.

2.2.2 n-dimensional k-ary mesh

This network topology, also known as grid, has *n* dimensions, each with *k* nodes. The length of all the dimensions needs not be the same, this is why the *k* value is sometimes denoted with an n-vector that specifies the number of nodes along every direction. One node in this topology is connected only to its neighbor nodes in every dimension (see Figure 2.2b). The 2-dimensional grid is one of the most-often used topologies in NoC research[DGC07, HN06, VLC+07, EEL07] and is also the chosen topology for the VASH framework. In the following chapters, the NoC sizes are given as a duple *AxB*. In the context of this work, this means a two-dimensional grid with *A* and *B* nodes in the X and Y coordinates respectively.

2.2.3 k-ary n-cube (torus)

As Figure 2.2c shows, this network topology is similar to the n-dimensional k-ary mesh, but it provides a connection between the first and last node in each dimension. For example the k-ary 1 cube has k nodes in a single direction. Every node is connected to exactly two other nodes so that they form a simple ring[STN02].

2.2.4 d-dimensional k-ary tree

This network topology has *d* levels. The first level may have more than one router, and every router in the following levels has *k* children routers in the next level. The processing units are only attached to the leaf routers in the tree. For example, Figure 2.2d shows a 2-dimensional 3-ary tree. When the tree has more than one router in the first level, this topology is known as *fat tree*. This topology is used in the work presented in [PFT+07].

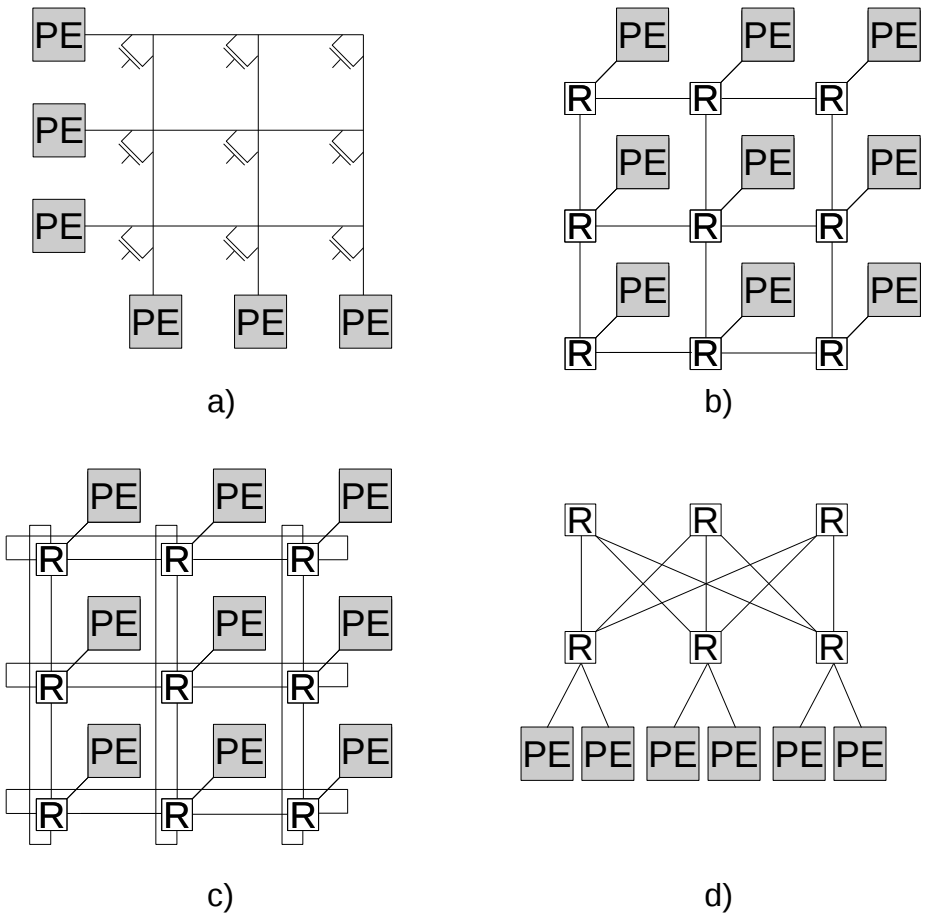

Figure 2.2: NoC topologies: a)Crossbar, b)Mesh, c)Torus, d)Fat tree

2.3 Switching techniques

The data transfer on a network link uses a fixed amount of parallel bits. This bundle of data wires is called a *phit* (physical unit). In order to control the data flow along the intermediate routers, phits need to be extended with additional bits to support synchronization and flow control. Such a unit is called a *flit* (flow control unit), and is at least as long as a phit. A given number of flits make up a *packet*, many of which constitute a *message*. The switching techniques deal with the transportation and storage of flits from the source node to the destination node. The two main ways in which the flit transmission can take place are: *circuit switching* and *packet switching*. In circuit switching, a physical path is set up between the communication endpoints prior to any flit exchange. After this connection is established, the sender can freely transmit data on this dedicated circuit. In packet switching, on the contrary, the packets in a message travel across the network independently, possibly over different paths and delays.

In the context of packet switching, the following subsections detail the most common algorithms for flit transmission.

2.3.1 Store-and-Forward (SAF) switching

In SAF switching a packet is forwarded to the next router when the latter has enough room in its internal buffer to hold the entire packet. As Figure 2.3 shows, a router can only accept the next packet once the current packet has been completely transmitted. SAF routing is one of the simplest switching algorithms, given that the complete packet is always transferred from router to router and, therefore, a packet cannot be permanently split among many storage buffers. In fact, there is no clear distinction between a packet and a flit. As a result, the required storage depth and minimum latency are at least equal to the size of a packet.

2.3.2 Virtual Cut-Through (VCT) switching

VCT switching is very similar to SAF switching, but this time individual flits can be forwarded to the next router as soon as there is enough room for the complete packet in the target router, i.e., there is no need to wait for the whole packet to be stored in a router in order to start forwarding flits (see Figure 2.3). As a result, the storage capacity is the same as in SAF switching but the latency in every packet hop is reduced.

2.3.3 Wormhole switching (WH)

As Figure 2.3 shows, in WH switching a flit is transferred to the next router as soon as the router has enough room to store it. This effectively reduces the required buffer capacity to one flit. However, if a router cannot accept an incoming flit, the storage for a packet may span many routers. As a consequence, this switching technique is more prone to congestion and deadlocks.

2.4 NoC Routing

The routing protocol defines the direction a packet must take at every router in order to reach its final destination. Since the routing protocol is exercised on every packet that traverses the network, its design and implementation are a major design concern for NoCs. Specifically, a good balance needs to be struck between sometimes contradictory metrics: power, area, performance and robustness to traffic change.

Roughly speaking, the routing algorithms can be classified as *static* or *dynamic*, and *distributed* or *source* routing.

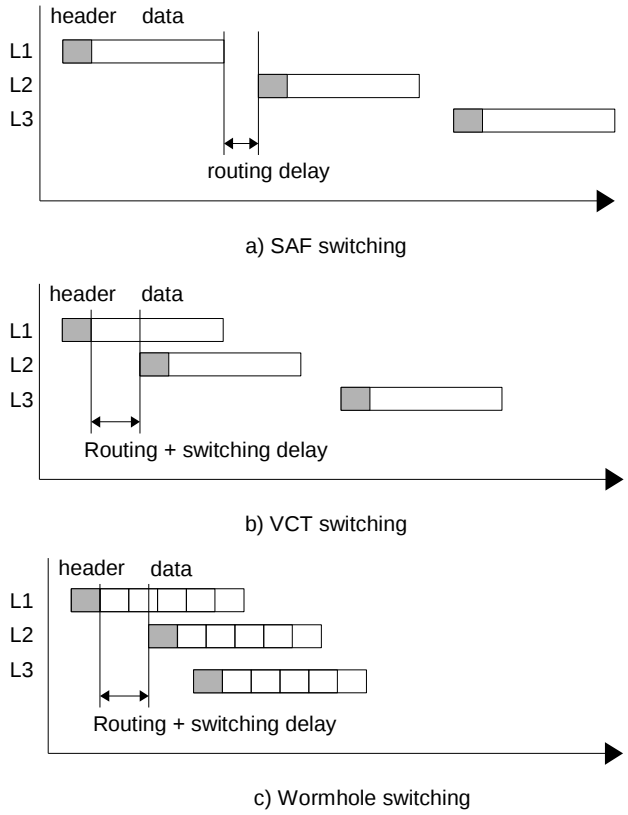

Figure 2.3: SAF, VCT and WH switching comparison

2.4.1 Static and dynamic routing

Static (deterministic) routing always uses the same path, regardless of the network traffic. *XY routing* is a very common static algorithm used in 2-dimensional meshes. In this algorithm the address of the nodes depend on their spatial XY coordinates, and every router compares its own address to the target address in the packet. Each router forwards the packet to one of its neighbor routers so as to match the X coordinate first and then the Y coordinate.

Decisions in dynamic routing, on the other hand, are made depending on the current network state. In *deflection routing*[LZJ06], for instance, when a packet enters a router, a routing table or function defines the preferred output port to forward the packet. If this port is not available, for example, because another packet is being transferred, another port is chosen for the incoming packet. The router does not have an extra buffer to hold such a packet, so it keeps bouncing around the network until it reaches its final destination.

2.4.2 Distributed and source routing

The routing algorithm can also be described according to where the routing decisions take place, and where the routing information is stored. In source routing, the routing information is kept in each NI and, when a packet is sent, its header is appended with all the directions the packet must take in every hop towards its destination. This routing strategy does not need intermediate routing tables but requires variable-length headers and specific global routing tables in every NI. In distributed routing, the target node address is included in the packet header, and each intermediate router must use a routing table or function to identify the appropriate output port. The XY routing algorithm is an example of this technique.

2.5 NoC Flow control

At the data-link level, the flow control deals with the allocation of resources as packets traverse the network. Resource allocation in NoCs is tightly related to fault tolerance. For example, when an error occurs in a network link and flit retransmission is employed, the flow control mechanism should support the data flow interruption, the error signaling and the re-allocation of the data buffers to previously transmitted flits. If retransmission is not supported in the data-link layer, error handling can also be accomplished using error-correcting codes, or using an end-to-end approach involving higher layers in the protocol stack.

At the transport layer, flow control serves a different purpose. Contention can effectively be resolved using techniques in the data-link layer, but these do not ensure that the target node has enough room in its input buffer to hold all incoming traffic. For this reason, a higher-level protocol has to be in place in order to account for slow or unresponsive nodes. That is to say, flow control at this level guarantees that all packets can be accepted on the receiving end of the communication.

2.5.1 Data-link layer flow control protocols

Several flow control protocols have been devised with different tradeoffs in performance, area, power and fault tolerance support.

The *STALL/GO* is one flow control protocol with good performance and no support for error handling. It uses one signal in the forward direction to signal there is data available, and one signal in the backward direction to condition the transmission of the next flit. The latter signal is thus able to stall incoming flits when the target buffer is full, and reactivate data transmission when some space in the buffer becomes available. This protocol offers reasonable performance given that, in the absence of congestion, one flit can be transmitted every clock cycle. Its main disadvantage is the lack of support for fault handling.

The *ACK/NACK* protocol provides support for transmission errors. The sender keeps an internal copy of every transmitted flit. Upon reception of a flit, the receiver may send back an ACK or a NACK token. If there was no transmission error, the receiver issues an ACK token and sender deletes the local copy of the flit. If, on the contrary, the receiver sends a NACK token, the sender rewinds its output queue and retransmits all flits coming after the erroneous one. The receiver also employs a GO-BACK-N

policy to discard a given amount of flits that may be in transit after the erroneous flit was detected. The ACK/NACK protocol requires more buffering resources than the STALL/GO method, although it provides fault tolerance by design. However, if NACKs are only produced as a response to sporadic errors, the performance degradation is minimal, but if they are used due to congestion, the performance drop and power consumption due to packet retransmissions may become a problem.

2.5.2 Transport layer flow control protocols

Flow control protocols in the transport layer can be divided into those that require resource reservations and those that do not. The former include methods as simple as dropping packets when there is no room for them, and other more complex solutions involving the routing of the packets. For example, when a packet can not be accepted in the target NI, it could be sent back to the source node. Another option is having the target node refuse the incoming packet temporarily, and using deflection routing to introduce the packet back into the network.

Methods that require resource reservations usually rely on end-to-end operations to grant network resources to a given communication flow. In credit-based flow control, the receiver keeps a credit counter and sends it to the sender periodically to update the number of packets it can receive. This means the receiver input buffer must be large enough to account for the delay in the transmission of the credits. *Virtual Circuit Buffers* can also be used in this context to help make sure a packet reaches an appropriate buffer. They provide dedicated storage for some network paths so that independent packet flows do not interfere with each other.

2.6 NoC Congestion control

Congestion appears as a result of *contention*, that is, the situation when two or more network packets need to use the same resource at the same time. As congestion severely limits the achievable bandwidth in a network, since available target nodes can no longer be reached, it should be avoided as much as possible. Figure 2.4 shows the scope of congestion control in comparison to that of flow control. While flow control regulates the data traffic between communication endpoints (master-slave pairs), congestion control regulates the temporally allocation of network resources (links and buffers) to the incoming packets in a network router.

In order to tackle congestion problems without resource reservation, techniques similar to those used for flow control can be employed, namely, packet dropping and dynamic routing schemes. Additionally, the packet injection rate can also be reduced to deal with the congestion problem at a higher level. Instead of routing packets around congested areas in the network, the NI may use statistical information, such as comparing offered load versus average latency distribution, in order to decrease the number of injected packets and maintain latency to a controlled level.

Techniques using resource reservation may eliminate congestion completely. For example, using Time Division Multiple Address (TDMA) to globally schedule the time and location of every packet ensures that every packet can be routed without contention[EDH06]. Although such technique requires a global notion of time, it does not necessarily imply a purely synchronous operation (single clock).

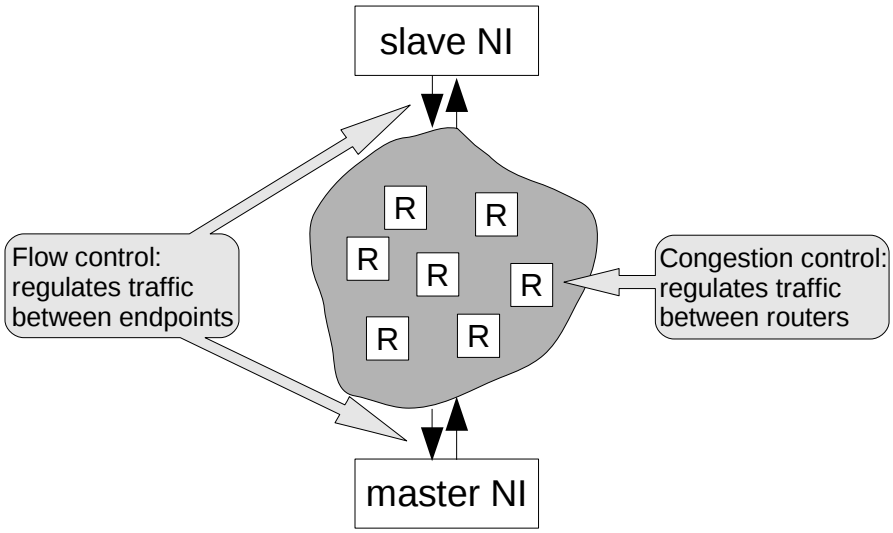

Figure 2.4: Scope of congestion and flow control

Rate-control schemes are another alternative when dealing with the congestion problem. In this strategy every node negotiates its traffic requirements with the network, and these traffic flows must be allowed to traverse the network concurrently. That is, the packet flows should not interfere with each other. The network links can be allocated to any combination of requested traffic flows as long as the total link capacity is not exceeded. If the negotiated traffic patterns are enforced, contention can be kept to a bounded value without a significant performance penalty.

2.7 Globally asynchronous locally synchronous (GALS) design styles

The recent interest in the GALS paradigm has been driven by the increasing difficulty in effectively distributing a global clock signal across the entire chip[TGL07]. Moreover, the GALS approach is also very well suited to some power-saving techniques, like voltage and frequency scaling. In addition, a GALS methodology may also reduce time to market by enabling the use of independent IP blocks, optimized for different clock frequencies and designed by separate teams.

The transmission of data across clock domains is the central issue in GALS design. In the following paragraphs GALS systems are classified according to the method they use to safely transmit data from one synchronous block to another.

Figure 2.5 shows the main GALS design styles, namely, *Pausible clocks*, *Asynchronous* and *Loosely asynchronous*.

One of the earliest GALS techniques proposes using pausible clocks to avoid metastability[Cha84]. The main idea behind this approach is to stop or stretch both the transmitter and receiver blocks while

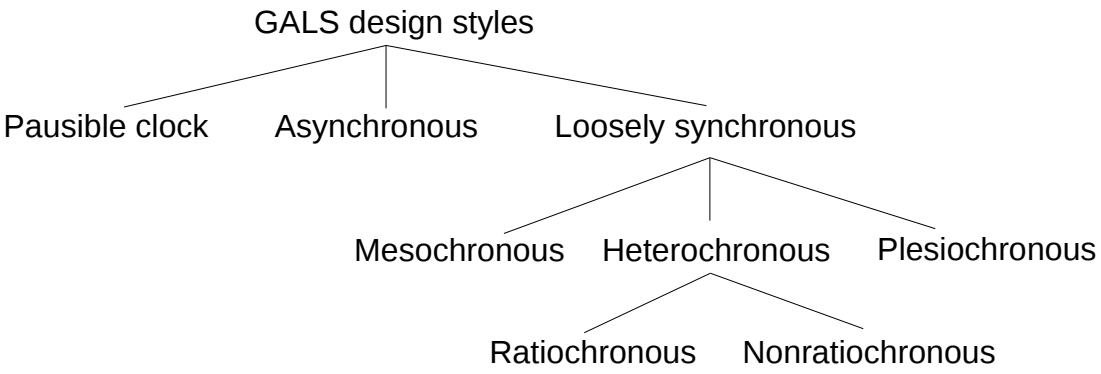

Figure 2.5: Taxonomy of GALS design styles

the data crosses a clock domain. As a consequence, the communication is very robust, as all timing violations are avoided by design. Another potential advantage of pausible clocks is power consumption: dynamic power can be reduced by stopping the receiver's clock, and the voltage level can be lowered during inactivity periods. In this approach, each synchronous island is wrapped in an asynchronous interface, and each clock signal is generated locally using a ring oscillator. The use of an asynchronous wrapper simplifies IP reuse, given that, once verified, the wrapper can be used in multiple hardware blocks without extensive timing analysis. However, the configuration of a ring oscillator for each synchronous island has been identified as a major design issue [TGL07]. Specifically, the jitter in the clock signal can be too high after the clock is re-enabled following a pause.

The *asynchronous* design style makes no assumption on the clock relationship of the synchronous islands. In order to transfer data to another clock domain, a synchronous signal must pass a synchronous-asynchronous interface in the sending domain, and an asynchronous-synchronous interface in the receiving domain. The communication between the asynchronous interfaces is usually Quasi-Delay Insensitive (QDI), that is, it makes no assumption other than isochronic forks about the relative delays on any of the circuit's wires. Isochronic forks assume the delays of all the fanout branches of a signal to be the same[PFT$^+$07, VLC$^+$07]. The resulting circuits, once verified, can be easily integrated in a standard Computer-Aided-Design (CAD) flow. The asynchronous interfaces are often implemented with asynchronous First-In-First-Out (FIFO) buffers that hide the synchronization problems from the synchronous blocks. However, the modeling and validation of the wrapper interfaces is difficult to achieve using standard techniques. The use of FIFOs is also expensive in silicon area when wide interconnects are used. It may also produce circuits with high communication latency, given that, as a general rule, 40 gate delays should be accounted for in order to resolve metastability[TGL07]. This extra delay prevents this synchronization method from being used in high-performance systems that require very high clock frequencies.

In *loosely synchronous* systems, there is a well-defined relationship between the clocks in the synchronous islands. This timing information is used to meet all timing constraints without stopping the clock or using FIFO buffers. This design style requires careful analysis of the logic path from the sender

domain to the receiver domain, but also eliminates the need for handshake signals. As a result, these synchronizers can achieve higher performance and more deterministic behavior. Loosely synchronous systems can be classified according to the following taxonomy:

Mesochronous: All synchronous islands operate at exactly the same frequency, but every clock signal has an unknown but stable phase difference. This situation is typical for a system with a single clock source, but different routing delays to distribute it to each part of the chip.

Plesiochronous: All synchronous islands operate at the same nominal frequency, but each synchronous block may present a small frequency mismatch. This situation is consistent with a system with multiple clock generators that cannot be perfectly synchronized.

Heterochronous: The synchronous blocks may operate at different nominal frequencies. *Rationochronous* systems are a special case of heterochronous systems, where every clock frequency is a rational multiple of another frequency in the system. Otherwise, the system is said to be *Nonrationochronous*.

2.8 GALS methodology issues

Although the GALS approach seems a promising solution to overcome the current challenges in SoC design, it has not found its way to mainstream adoption yet. While it is true that systems with complex interconnection structures, short time to market, or power constraints may benefit greatly from a GALS methodology, all the potential advantages do not justify the additional design effort to implement an asynchronous interface. One of the main reasons for such a slow adoption of the GALS paradigm is the lack of CAD tools to support development of the asynchronous circuits. Even in the mesochronous case, there is no widespread support for timing analysis on a logic path that spans multiple clock domains. As a consequence, the synchronizer design has to be performed manually, and therefore, the resulting circuit is tightly coupled to the application at hand. A related problem arises from the non-deterministic behavior of the arbiters and synchronizers in a GALS system. The uncertainty in the system operation makes validation and test difficult with traditional techniques aimed at fully synchronous circuits. As technology trends continue to push towards a GALS on-chip communication strategy, the CAD industry is expected to bring some degree of asynchronous support into the standard design flow. The VASH framework aims to assist the designer in the validation of GALS concepts within a NoC utilizing standard CAD tools.

Chapter 3

VASH architectural description

In this chapter the VASH hardware architecture is explained in detail. The configuration options are shown for each hardware block, and the reasons behind some of the design choices are given where appropriate. The software approach for network generation and configuration is also briefly discussed. However, the details of the software architecture is left for Chapter 4. At the end of this chapter the NoC validation strategy, both for single block and system-level operation, is thoroughly covered.

3.1 VASH hardware architecture

The VASH hardware architecture comprises the hardware blocks necessary for packets to be injected into and retrieved from each of the endpoint nodes in a NoC. For this purpose, special care was taken to make all the hardware components configurable enough, both at synthesis time (statically) and during emulation (dynamically), to measure the switch performance under different situations.

Figure 3.1 shows the hardware architecture overview of one node in the network; this top level hierarchy consists of a packet generator (packet_gen), a network interface (NI), a network switch and the umr_controller. A few remarks common to all blocks are given in the next paragraph while all design units are described in detail in the following subsections.

As Figure 3.1 shows, VASH makes use of FIFO buffers to hold incoming and outgoing data. These FIFOs have different read and write clocks and effectively decouple the timing relationship in each clock domain. The umr_controller needs to be clocked at exactly 75 MHz and the rest of the blocks can use any clock frequency as long as the delay along the critical path is not exceeded. A built-in Digital Clock Manager (DCM) can be used to shift the phase of a clock signal, and several network nodes can share the resulting output clock. The shifting can take place once, when the system is synthesized (fixed mode) or several times during system execution (variable mode). In the latter case, the control of the shifting logic should be driven by a separate clock and, as a consequence, a third clock domain must be accounted for.

The hardware architecture only supports the flow of same-length packets; the hardware blocks are synthesized for a given packet length even though the network fabric may support variable-length packets. The reason for this design choice was an effort to keep the implementation simple while still

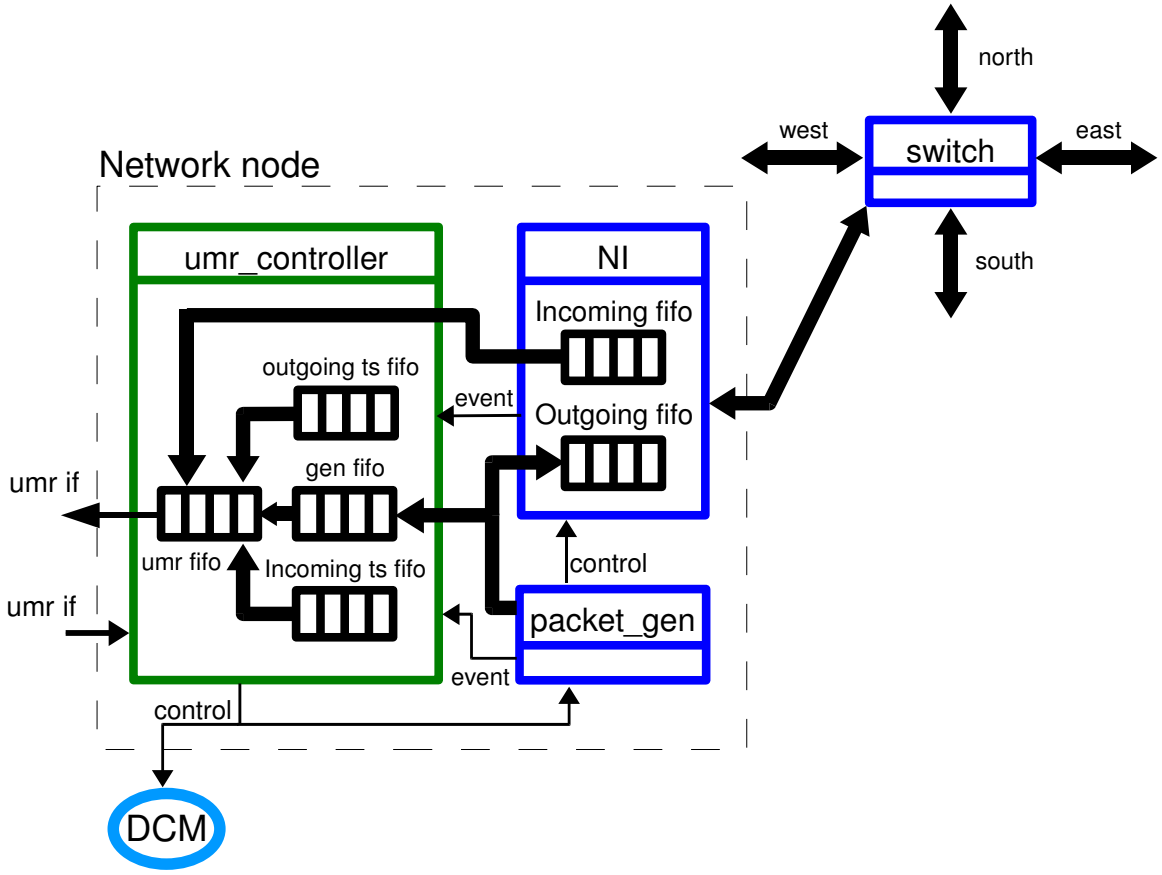

Figure 3.1: VASH hardware overview

maintaining minimum latency, that is, the FIFO management can be performed by simply monitoring the almost-full and almost-empty flags, that can be set to go active on a single threshold value. If the characteristics of the emulated design requires the use of variable-length packets, the current architecture can still be used by setting the FIFO threshold values according to the maximum packet length and always reading and writing this amount of phits. On the software side, the monitor program could remove the extra phits and reconstruct the original packets.

3.1.1 Packet generator

The packet generator places a given number of packets both in the outgoing FIFO of the NI and in an internal FIFO buffer.

The operation of the packet generator is stalled if any of target buffers does not have enough space to hold an entire packet. The internal FIFO is read by the umr_controller so outgoing packets can be sent

to the host PC for validation and latency calculation. The packet generator waits a fixed number of cycles (WAIT_VALUE in Table 3.1) before creating the next packet and then raises an outgoing event so that a timestamp can be captured at exactly the point in time when a new packet is written to the buffers.

The destination of the generated packet is produced by the internal block *address_gen*. The current VASH implementation generates sequential addresses using cascaded counters for the X and Y coordinates. Although this traffic pattern is not representative of a real network load, it is enough to validate the switch operation and its synchronization under heavy contention. Other more realistic traffic patterns could be obtained by providing another implementation of this block.

Figure 3.2 shows a simplified flow diagram of the operation of the packet generator. After a specified number of cycles has elapsed, another packet is transmitted only if the maximum number of packets has not been reached (MODE parameter in Table 3.1). The packet counter can be reset during execution to transmit several packet bursts (reload=1 in the figure). At this point, the fifo status flags are checked before requesting and validating a destination address. After obtaining a valid destination, the header phit and the remaining data phits are written in the target FIFOs.

The outgoing event is generated right after validating the target address. As a consequence, these necessary extra cycles do not affect the latency calculation.

Table 3.1 also shows additional configuration parameters for the node address and phit and packet length.

Table 3.1: Packet Generator configuration

Name	Type	Description
ADDRESS	Static	XY node address
PHIT_SIZE	Static	Number of bits in a phit
PACKET_LENGTH	Static	Number of flits in a packet
WAIT_VALUE	Static	Wait cycles between packet transmissions
MODE	Dynamic	Number of packets to send

3.1.2 Network interface

The network interface in the VASH framework stores the incoming and outgoing data (phits) and adds the necessary extra signals for synchronization and flow control (flit). The transmission of packets to the network switch takes place only when a complete packet is available in the outgoing FIFO and is also subject to the switch flow control mechanism. On the receiving side, the header flit is only accepted if there is enough space in the incoming FIFO. Since the NI is the endpoint of the communication link, once a packet is accepted, it may wait any number of cycles before reading the next flit as long as the switch minimum latency is still observed.

Table 3.2 shows the configuration parameters for this hardware block. The acknowledge pre-processing option has to do with the synchronization strategy and it is discussed later in this chapter.

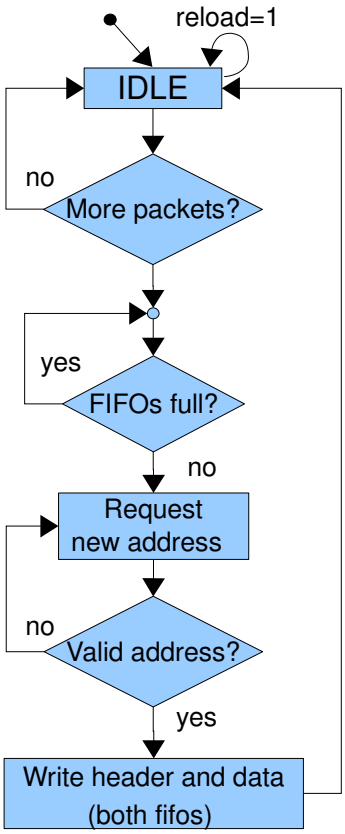

Figure 3.2: Packet generator flow diagram

Table 3.2: Network Interface configuration

Name	Type	Description
ADDRESS	Static	XY node address
PHIT_SIZE	Static	Number of bits in a phit
PACKET_LENGTH	Static	Number of bits in a packet
PULSE	Static	Acknowledge preprocessing
MODE	Dynamic	Number of cycles to wait before reading the next flit

3.1.3 Host PC interface

The *umr_controller* block handles the communication between any network node and the host PC via a proprietary protocol from ProDesign, the emulation machine manufacturer. This protocol, called *UMR*, uses a serial interface to transmit 32-bit words between the host PC and the umr_controller. In order to transmit packets to the host PC, a FIFO buffer is filled with a sequence of data words describing

either an outgoing or incoming packet. This word sequence is composed of a status word at the very beginning, two 32-bit words that make up a 64-bit timestamp, and the header and payload of a packet.

On the receiving side, the host PC can send one data word to the UMR controller and dynamically configure the operation of a network node. Tables 3.3 and 3.4 show the structure of the status word for read and write operations respectively.

A packet data sequence is transmitted to the host PC when the application software issues a read command to one of the network nodes. If there is no data available the read status bits in the status word are set to *NONE* and the application software should not expect any packet data in subsequent access to the node, otherwise these bits are set to *INCOMING* or *OUTGOING*. Polling the nodes this way may come at the expense of high overhead in certain load conditions. For the time being, any potential problem could be solved by making the internal FIFOs large enough, and, in future designs, the support for interrupts in the UMR protocol could be used to transfer data more efficiently.

The enable bits in the control word make it possible to simulate an unresponsive node and create congestion in the network. A global enable qualifies all other enable signals to start the operation of all nodes at exactly the same time. The global enable signal is driven by the node with address (0,0) and uses a global physical signal to reach all other nodes in the network. The FIFO flags, on the other hand, enable the network characterization by revealing potential bottlenecks in any of the network nodes.

Table 3.3: Status word write structure

Name	Offset	Size in bits	Description
GEN_EN	0	1	Packet generation enable
RCV_EN	1	1	NI receive enable
SND_EN	2	1	NI send enable
GLOBAL_EN	3	1	Global enable
REL	4	1	Packet generation reload
PG_MODE	5	2	Number of packets to transmit
SH_TRIGGER	7	1	Phase shifter trigger
SH_DIR	8	1	Phase shifter direction: increment/decrement
LAT	9	1	Flit latency in NI

3.1.4 Network-on-chip switch

The network switch used in this work was developed at the University of Rostock[Imd]; it supports 2-D mesh networks, 32-bit phits with 38-bit flits, X-Y routing and STALL/GO flow control. The main goal for this design choice was the significant time savings of a pre-verified hardware component as opposed to designing a new one from scratch. NocEm[NoC], another popular open source implementation option, was also considered as a candidate switch but, due to its coding style and better source code documentation, the simpler router from the University of Rostock was eventually used in the VASH framework.

Table 3.4: Status word read structure

Name	Offset	Size in bits	Description
OFE	0	1	Outgoing fifo full
OFF	1	1	Outgoing fifo empty
IFE	2	1	Incoming fifo full
IFF	3	1	Incoming fifo empty
SND_EN	4	1	NI send enable
RCV_EN	5	1	NI receive enable
PG_EN	7	1	Packet generator enable
TYPE	8	2	Read status
LOCK	10	1	Local clock stable
GLOBAL_EN	11	2	Global enable

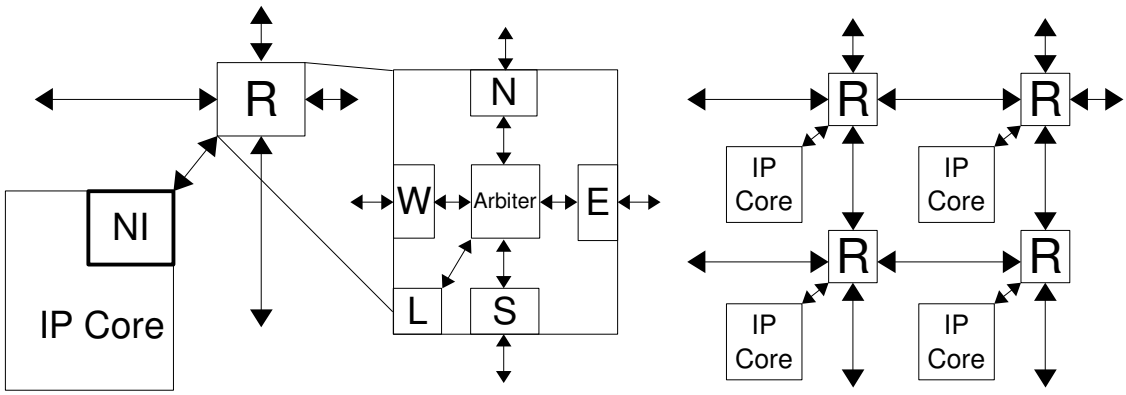

Figure 3.3: Structure of an NoC router

As shown in Figure 3.3 [KHT07], the network switch comprises four ports (north, east, south, and west) to communicate with neighbor switches, and a fifth port for the local network interface. Each port consists of two links, one for incoming and another for outgoing data. Table 3.5 shows the signals present in an outgoing link; they should be connected to an incoming link from either another switch or from the NI.

Once any switch or NI is ready to transmit a packet, it presents the first flit (header) on the data line and sets the *req* signal active. In the receiving switch, this request is arbitrated and relayed to the target port in the following clock cycle. At this point, the packet data is stored in the output buffer and is ready to be processed by the next switch in the path to the destination node. Each transmitted flit must be acknowledged by the receiving port. This stalls the data flow in case of congestion. After a port is granted for a data transfer, it monitors the *end-of-frame* signal (eof signal in table 3.5) and, when the last flit has been transmitted, it frees all resources associated with this data transfer in order to make the link available again. When there is no contention in the switches, an incoming flit can be acknowledged two clock cycles after the request signal is active, that is, one clock cycle to request the target port,

Table 3.5: Flit structure

Name	Size in bits	Direction	Description
req	1	out	Active when the sending switch requests a port in the receiving switch
sof	1	out	Start of frame. Active during the first flit of a new packet
eof	1	out	End of frame. Active the last flit of a packet
data	32	out	Packet data
be	2	out	Byte enable. Qualifies the bytes in the packet data
ack	2	in	Acknowledge signal. Active when the receiving switch has processed the current data word

and another clock cycle for the arbiter to respond. The arbiter uses a round-robin algorithm to resolve concurrent requests for the same port.

As Figure 3.4 shows, after the arbitration takes place, the data flow is similar to that of a shift register, where the node closest to the final destination controls when to perform the shifting and read new data. When the header flit has reached its final destination, the NI might read a new flit every clock cycle (no arbitration required). However, if transporting the data to the next switch requires more then one clock cycle, e.g a synchronizer is used, the NI needs to insert some wait cycles before reading the next flit. This is a direct consequence of using an output buffer with a single slot for the current flit.

Table 3.6 shows the configuration parameters for the network switch. The PULSE parameter configures the use of a pulse detector after the incoming acknowledge signal. Before it is used, this signal needs to be conditioned in some cases in order to account for the uncertainty in the synchronizer behavior. The use of the phase detector and its latency implications are detailed later in this chapter.

Table 3.6: Switch configuration

Name	Type	Description
ADDRESS	Static	XY node address
PHIT_SIZE	Static	Number of bits in a phit
PULSE	Static	Acknowledge preprocessing

3.1.5 Clock domain synchronizers

In this project, the synchronization approach is similar to that in [KHT07], that is, a synchronizer is used in the receiving clock domain to safely latch the signals *req* and *ack*. These signals are responsible for flow control and enable the asynchronous data transfer between switches. However, the data lines

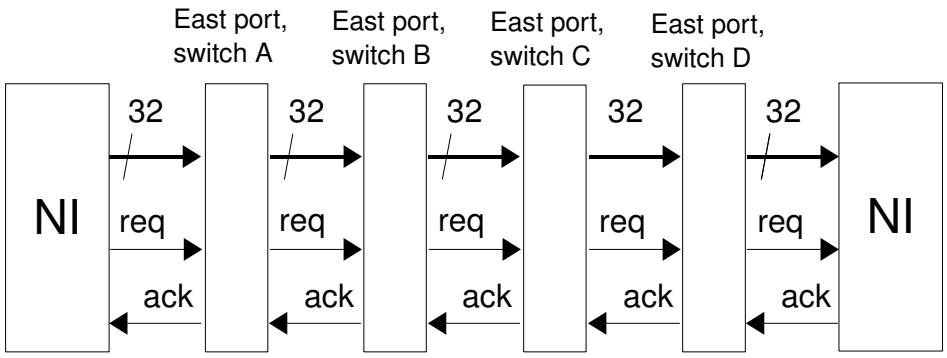

Figure 3.4: Switch flit flow

themselves are not synchronized, so an extra clock cycle has to be used to ensure they are stable before sampling them in the target domain. Figure 3.5 shows the structure of a mesochronous NoC and its Clock Domain Crossings (CDC). The current VASH implementation employs the same clock domain for the node and its local switch but, given that the NI contains a FIFO buffer with separate clock signals for the reading and writing port, these hardware blocks could use two different clock frequencies. Each switch can be set up to use a separate clock domain but, for actual implementation on the FPGA, the number of different clock signals is constrained by the number of global clock buffers. Likewise, the number of different clock phases in one FPGA depends on the number of available Digital Clock Managers (DCM).

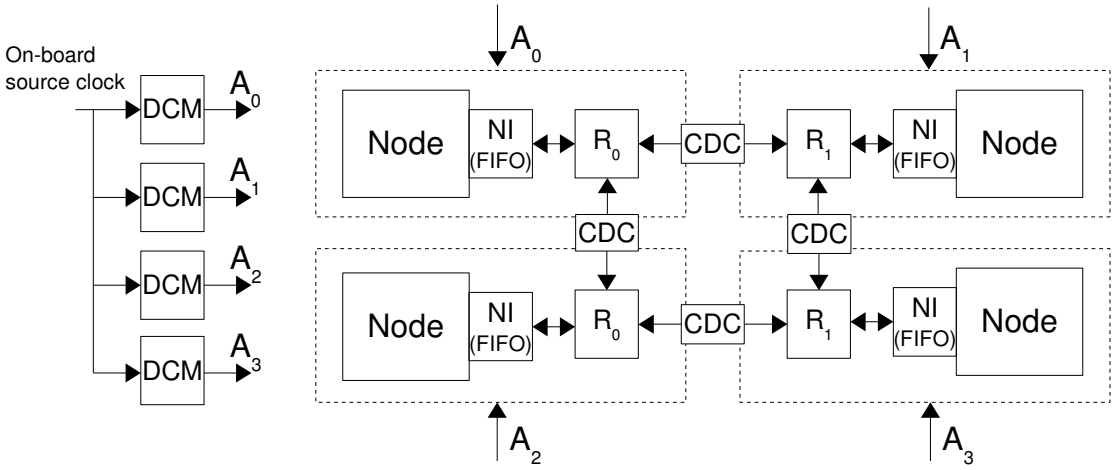

Figure 3.5: Clock domains (dotted lines) in the NoC

Double flip-flop synchronizer

This technique employs two cascaded flip-flops to reduce the probability of receiving an indeterminate value in the target clock domain. The asynchronous input to the first flip-flop may cause a timing violation, and therefore produce an metastable value at the input of the second flip-flop. However, this indeterminate value has one full period to resolve to a stable state before being latched in the second flip-flop. The exact probability of synchronization failure depends on the flip-flop's electrical parameters. For the target FPGA and frequency range, it is highly unlikely to propagate an indeterminate voltage level out of a synchronizer[Xila]. The double flip-flop synchronizer is able to avoid metastability but, in order to prevent data loss, a handshaking protocol needs to be employed.

Figure 3.6 shows how two cascaded flip-flops are used to transfer a packet flit between clock domains according to the STALL/GO flow control protocol described in section 2.5.1. The arbiter in the target switch takes at least one clock cycle to process the transition on *req*, this is enough time to let the *data* signal be stable and safely stored in the destination output port buffer. Contrary to the behavior of the *req* signal, the *ack* signal is active during one clock cycle in order to shift the incoming data one position forward. This means that if, due to a timing violation, the first stage of a double flip-flop synchronizer fails to follow a transition on the *ack* signal, a flit acknowledge can be lost. This inevitably leads to a network deadlock as some links in the network are locked while a switch expects an acknowledgement that is never received. To account for this uncertainty in the link, two architectural modifications were used. On the one hand, all *ack* and *req* signals are kept high for two clock cycles. This ensures that the acknowledgement is active for at least one clock cycle since the input of the first stage of the synchronizer is guaranteed to be active and stable. On the other hand, a pulse detector is used to detect transitions on the *ack* signal; this ensures that only one data flit moves forward each time the *ack* signal is active.

The right hand side of Figure 3.6 shows the worst-case latency in the path of the acknowledge signal from the destination switch to the source switch. This situation arises when both transitions on the *ack* signal are lost in the first synchronization flip-flop. As can be seen in the figure, q_4 goes low in clock cycle 4 so the output of pulse detector goes high in clock cycle 5; the flit data is shifted forward in clock cycle 6 and the receiving switch can read the new data in clock cycle 7.

Latch synchronizer

The latch synchronizer is implemented as proposed in [LAB08]. This synchronizer was originally designed to enable 3D NoCs, but it could also be used in traditional planar designs. The synchronization takes place at the receiving side of a communication link, and uses the clock of the sender domain as a strobe signal to store the incoming bundle of wires (data or control flow signals) in one of two parallel latches. The clock signal in the sending domain is transferred to the receiving domain as any other data signal. This simplifies the synchronization process given that all transmission lines should be subject to similar wire delays. The circuit description of this synchronizer is shown in Figure 3.7. In the front-end, the 1-bit counter alternates the latch where the next incoming flit is stored, while in the back-end, the 1-bit counter selects which latch to sample into the output flip-flops. The phase detector is used to evaluate the phase difference between the clock domains and initialize the counters so that the data at the input of the flip-flops is guaranteed to be stable. Given the mesochronous nature of the

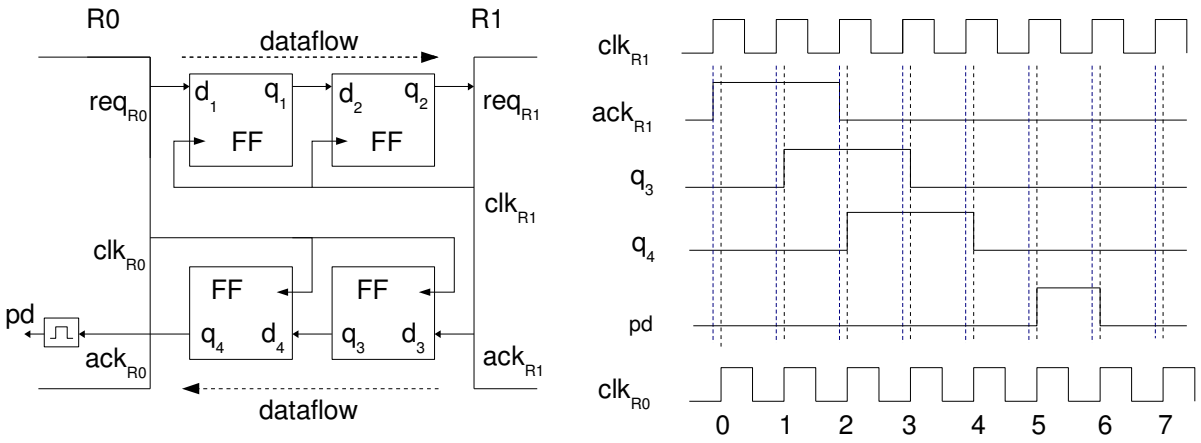

Figure 3.6: Double flip-flop synchronizer

link, the phase difference is constant during the operation of the whole system. This is why the counter initialization is only performed once, before any data transmission takes place.

The *ack* signal takes one clock cycle to reach one of the latches in the front-end and, one clock cycle later, it appears on the output flip-flop. At this point, the incoming data can be shifted forward (third clock cycle) so it is safe for the next switch to read the flit data on the fourth clock cycle.

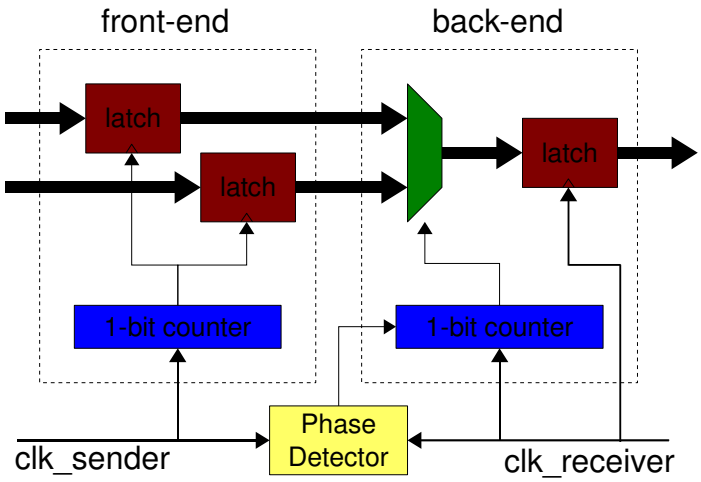

Figure 3.7: Mesochronous link synchronizer

In order to avoid metastability and data loss in the synchronizers, two conditions should be met: (i) the incoming data in the front-end must be latched safely and (ii) the data to be sampled in the back-end

must be stable. For condition (i) to be fulfilled, the latches must be carefully handled to control the point in time when they become transparent. Ideally, the latch enable signals should change at exactly the same time as the sender clock is toggled. However, this is not possible because the enable signals must be conditioned by local signals, introducing a delay t_{cond}. It should also be noted that the sender clock and the incoming data may no longer be perfectly synchronized given that they could travel through different routes on the chip. This introduces a timing skew $t_{routingskew}$ to account for any possible variation in wire delay. This means that the latch enable signals have, in the worst case, an offset of $t_{cond} + t_{routingskew}$. This value can be an advance if $t_{routingskew}$ is negative and larger than t_{cond}, or a delay otherwise.

In order to produce a robust synchronizer, the authors in [LAB08] propose the circuit shown in Figure 3.8 to generate the latch enable signals in the front-end. The flip-flop toggles its output on the positive edge of clk_sender but the latch enable signal becomes active when clk_sender is low, that is, half a clock cycle later. This way, each latch is transparent for one semiperiod every two cycles.

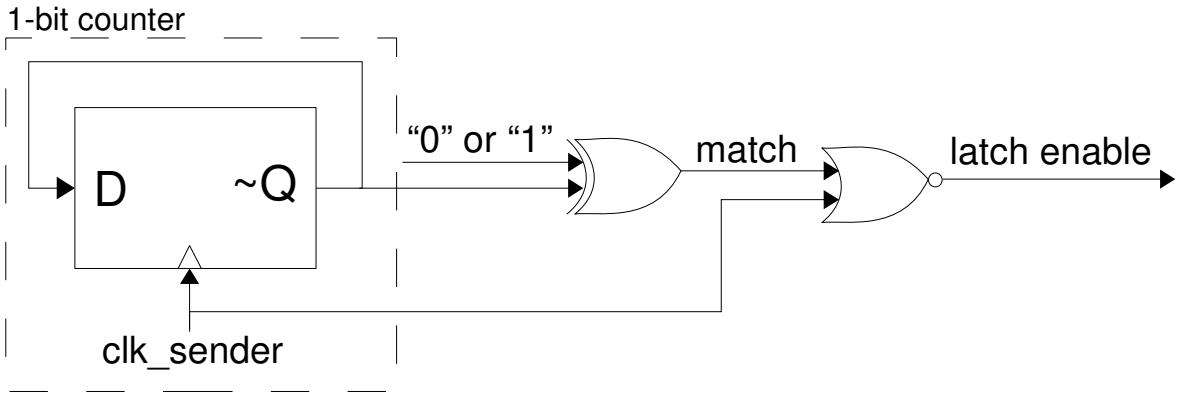

Figure 3.8: Circuit to produce the latch enable control signals

The enable circuit in Figure 3.8 poses a problem when implemented on an FPGA, namely, a clock signal should be taken out of the Global Clock Buffer (GCB) in order to drive combinational logic. In this case there is only one clock signal involved, so an implementation constraint (see section 4.4) could be applied to guarantee low skew at the local net that drives the nor gates of the two enable circuits. The revised circuit shown in Figure 3.9, however, retains the same behavior without a low skew resource at the expense of an additional flip-flop. This circuit has been implemented in the final version of the latch synchronizer.

In summary, the conditions for the correct operation of the latch synchronizer are:

(3.1.1a) $$t_{cond} + t_{routingskew} + t_{latchhold} < t_{datamin}$$

(3.1.1b) $$t_{clk} + t_{cond} + t_{routingskew} > t_{datamax} + t_{latchsetup}$$

(3.1.1c) $$t_{counter} + t_{comp} > t_{clk}/2$$

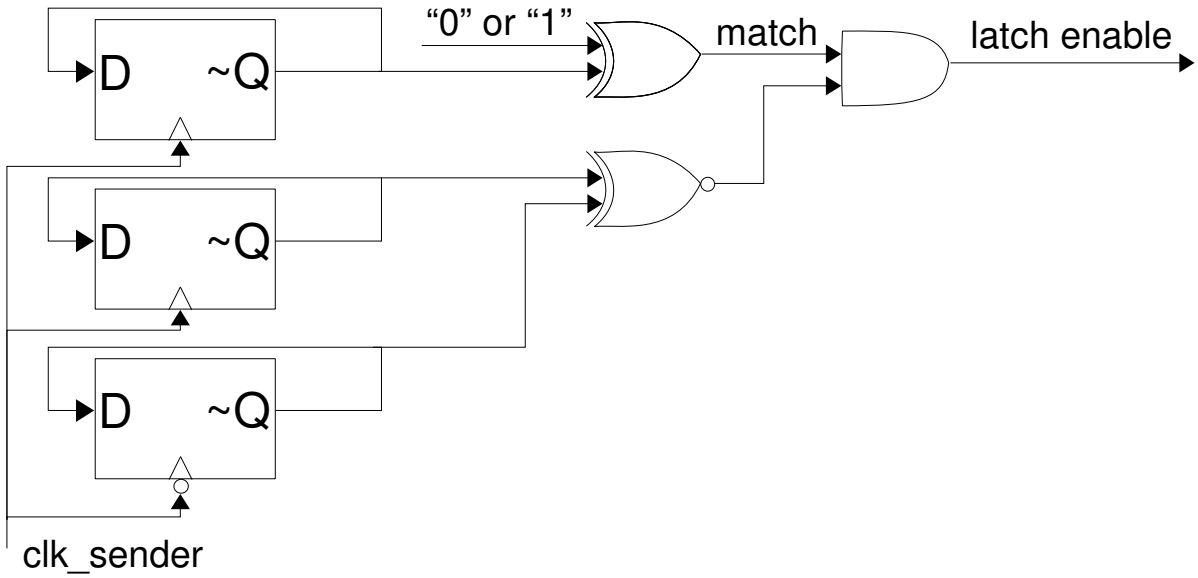

Figure 3.9: Implementation of *latch_enable* signal on the FPGA

where $t_{datamin}$ and $t_{datamax}$ are the minimum and maximum delay of the incoming data after the previous edge of *clk_sender* respectively.

Equation 3.1.1a guarantees that the latch enable come early enough not to let the next data flit be captured by the front-end latch. Equation 3.1.1a, on the other hand, guarantees that the latch enable signal come late enough to let the incoming data be stable before making the latch transparent. Lastly, equation 3.1.1c ensures the correct operation of the circuit in Figure 3.8, that is, the generation of the latch enable signals must be performed in one clock semiperiod.

Condition (ii) can be easily fulfilled with a proper initialization of the counters before system operation.

Table 3.7 shows the configuration options in the latch synchronizer source code. The INPUTS parameter specifies the width of the bundle of signals to be synchronized, while the rest of these parameters make system integration easier during validation, simulation and synthesis. For instance, the PHASE_SYNC parameter can be used to validate the operation of the synchronizers under different assumptions about the phase relationship in neighbor nodes. The INSERT_CB parameter, on the other hand, is useful when performing post place-and-route simulations. It instructs the synthesis tool whether to put the incoming clock in a separate clock buffer or in a Input/Output block (IOB).

Phase detector

The design of the phase detector was constrained by the same problem present in the latch enable circuit of section 3.1.5, but this time two clock signals need to drive combinational logic, and, for accurate

Table 3.7: Latch synchronizer configuration

Name	Type	Description
INPUTS	Static	Number of input data lines
PHASE_SYNC	Static	Value 0: input clock signals are not in phase
		Value 1: input clock signals are in phase
		Value 2: a phase detector should be included
INSERT_CB	Static	When true, a clock buffer is inserted for the back-end clock signal.

phase detection, their routing delays should be perfectly matched. With this problem in mind, and considering the lack of appropriate routing constraints (see section 4.4), one implementation option is shown in Figure 3.10. This design is based on the one presented in [Xil04]. The outputs of the flip-flops on the right hand side of the figure are held low after power-on and they are reset as soon as they both receive one positive clock edge. This behavior produces two pulses with transitions at exactly the same time if the clocks are in phase. Otherwise, only the falling edges happen at the same point in time. The rest of the circuit monitors these pulses in order to determine whether the clock signals are in phase. If this is the case, the output in the upper latch is always the same as that from the lower latch, so the data input (*i*) of the output latch is not high for enough time as to produce a high value on the *phase* signal. If the clocks are not in phase, the output in one of the intermediate latches is high while the output in the other latch is low. This makes the output of the XOR gate and the *phase* signal go high.

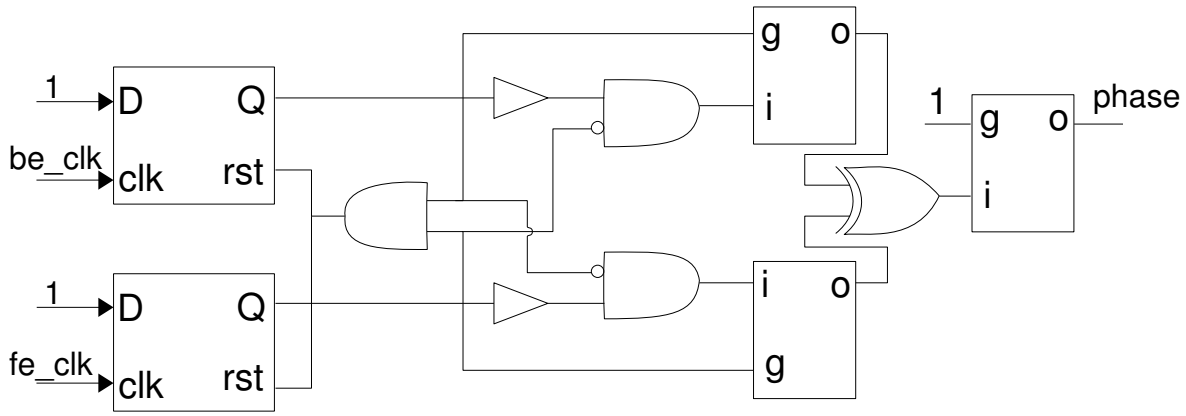

Figure 3.10: Phase detector

3.2 VASH software

The VASH software tools enable the generation of a synthesizable network description that contains a set of parameterized hardware blocks. The VASH software produces an internal representation of a NoC that can later be dumped in VHDL format and fed into standard synthesis tools. The VASH internal representation contains top-level design units that store both the instances of the VASH components and their interconnections. These design units can also be interconnected in order to distribute the network over several chips or FPGAs. The VASH software extends the DfX framework, an open-source development environment for hardware designs written in VHDL. DfX was developed in the Institute of Computer Architecture and Computer Engineering at Universität Stuttgart. Its goal is to provide an integrated solution to compilation, synthesis, simulation and testing of designs. The main extension for the VASH framework has been to support a hierarchical component design style and the programmatic configuration of all required hardware blocks with very little design effort.

3.3 Validation strategy

The correct operation of the fully synchronous hardware components in the VASH framework can be verified by simulating their RTL description in any of the standard event-driven simulators widely available in the CAD community. However, this approach fails to capture any timing issues in the mesochronous synchronizers. For this purpose, most synthesis tools are able to build a hierarchical description of a design after place and route. This model is accurate enough to reflect any timing violations.

The behavior of the double flip-flop synchronizer, however, is simple enough to be accurately described as behavioral VHDL code. This approach is useful to validate the mesochronous NoC without the overhead of processing a fully placed design.

To validate the operation of the latch synchronizer within the NoC, it is enough to verify that each outgoing packet reaches its destination. This validating strategy arises from the observation that (1) the *req* signal must be correctly relayed along the path of a packet to its target node and (2) the *ack* signal is active for one clock cycle each time a flit is transmitted. If this signal is lost, a deadlock occurs since the sending NI would not continue to inject flits. On the other hand, if the *ack* signal is active for two clock cycles, the receiving NI would read the incorrect packet data.

3.4 Latency issues and diagnosis

The initial emulation experiments with the latch synchronizer showed data errors and packet loss. The problems were not present when the double flip-flop synchronizer was used, so the problem was most likely a synchronization issue involving the use of the implementation constraints for assisted placing and routing (see section 4.4). Although the post place and route simulation model was able to correctly detect a problematic phase relationship in the back-end, and then transfer data safely between clock domains, its FPGA implementation did not produce the expected results. This situation was difficult

to debug with an external logic analyzer because of a problem in the ProDesign scripts to insert and make observation points visible in a test probe. The NoC was then emulated without the use of a phase detector, i.e., the configuration parameters in the latch synchronizer were used to statically specify whether the receiving and sending clock signals in a link were in phase. Apparently, the original problem was related to the way the implementation of the phase detector affects the overall behavior of the rest of the synchronizer, since once the output of the phase detector is fixed, the error behavior changes drastically.

The new errors stem from the fact that the exact timing relationship between the relevant clock signals is highly dependent on the implementation algorithm and cannot be predicted before the design is completely realized. As a result, the fixed phase offset may not accurately capture the real timing relationship in the clocks.

In this situation, a timing error may prevent the 1-bit counter in the back-end from being correctly initialized. It is worth noting that even when such an error occurs, the front-end continues to work correctly, so the incoming data is neither duplicated nor lost. The only observable difference in the synchronizer's operation is that 5 cycles need to be observed between activating the *ack* signal and reading the next data flit bundle. As explained in Section 3.1.5, the synchronizer latency under normal operation is 4 clock cycles. If the back-end counter is not correctly initialized, the incoming data is stored in the same latch being sampled in the back-end. However, no timing violation takes places since the latch becomes transparent half a clock cycle later, i.e., after the old data is transferred to the front-end. The data in the other latch is then sampled before the first latch is sampled again and the current data is transmitted out of the synchronizer. This extra sampling clock cycle makes up the additional clock cycle in the total synchronizer latency.

Further emulation experiments showed that data errors or packet loss never occur if a latency of 5 cycles or more is observed in the NI. Moreover, if all synchronizers are correctly set up, the nominal latency of 4 clock cycles can be achieved. However, given the limited insight into the implementation algorithm, the required manual configuration, although effective, can be difficult to obtain for large networks.

Chapter 4

Implementation and results

In this chapter the technical details in the development of the VASH framework are thoroughly explained. This includes the class diagrams and source code description on the software side, and the custom development procedure on the hardware side. The emulation infrastructure is also briefly discussed, and the performance and area of the double flip-flop and latch synchronizers are compared and analyzed.

4.1 Implementation resources

4.1.1 Development software tools

The Xilinx ISE 8.2 toolset was used to implement the hardware designs and build the necessary bitstreams for FPGA configuration. The software tools provided by ProDesign, the emulation machine manufacturer, set up the implementation flow for each FPGA automatically. The only manual step in the design specification is to include the target VHDL source files, clock signals and any custom implementation constraints in the appropriate script for synthesis, mapping and placing. In addition, some hardware, like the FIFO buffer and the DCM, were directly implemented using the Xilinx CoreGenerator tool. This tools enables the generation and configuration of the VHDL description of many commonly-used soft cores available in the Xilinx libraries.

All NoC simulations, both at the RTL and gate level, were performed using ModelSim 6.1 from Mentor Graphics.

4.1.2 Emulation machine

The CHIPit Platinum Edition prototyping system from ProDesign was used to realize the NoC emulation. The available system contains two boards with three Xilinx Virtex-II XC2V6000-6 each. Each FPGA has 184 fixed bi-directional connections to the two other FPGAs in the same board and 360 routed interconnections that may be used to reach any FPGA in the system. The emulation platform provides 8 clock domains with a frequency of 200 MHz at the board level and 100 MHz at the system level. The platform communicates with a host PC using the UMR protocol, a proprietary solution to read data

from or write data to a maximum of 32 endpoints in each board. ProDesign provides both the hardware cores to implement the protocol and a C software API[Pro07] ready to be linked into the user software. There are two implementations of the UMR software API: one exchanges data with the real hardware and the other communicates with a simulation model via a PLI interface. In the latter case, the user software and the simulation model are run concurrently in order to make debugging easier.

4.2 VASH software architecture

The VASH software architecture, as an extension of the DfX framework, consists of three new Java packages and some minor modification to the existing code base. The reason for this development effort is to support specific functionality useful for the generation of NoCs.

The new Java packages, *de.uni_stuttgart.iti.vash*, *de.uni_stuttgart.iti.vash.chipit* and *de.uni_stuttgart.iti.vash.parser* are from here onwards referred to as the last identifier in their qualified name, that is, *vash*, *chipit* and *parser*, respectively.

4.2.1 vash package

The vash package contains the classes that support the interconnection of hardware blocks inside a netlist. Figure 4.1 shows the class diagram of this package. The *NOCBuilder* abstract class provides common functionality to manipulate a netlist (creating and binding ports, for example) and to assist the naming of the instantiated hardware components. The classes *SingleClock* and *MesochronousClock* implement the *IClock* interface to provide a clock signal to the nodes in the NoC, while the classes that implement the *ISynchronizer* interface attach a synchronizer circuit to the NoC description. The parameters in the methods of this last interface include the parent netlist, the ports where the input signal is read from, the port where the synchronized signal is bounded and the clock signals in the receiving and sending clock domains.

Finally, the *NOCBuilder* class makes use of the rest of the classes in the package to build a netlist with the desired NoC characteristics. It creates the top-level input and output ports, instantiates and configures all nodes and switches, interconnects them together and stores any produced mapping or place and route constraints and all meta-data from the NoC generation, like the signal interconnection matrix in a multi-FPGA design. The output files generated by the *NOCBuilder* class are listed below.

connections.csv: This file describes the top-level interconnection matrix between FPGAs. It can be fed directly to the CHIPit manager tool.

priorities.txt: This file contains the routing priorities so that the clock signals are routed with higher precedence.

dfx_vash: Auto-generated VHDL package. It must be included in every synthesis script.

NOCXY_toplevel.txt: Group constraints for FPGA XY.

NOCXY_toplevel.vhd:Generated NOC for FPGA XY.

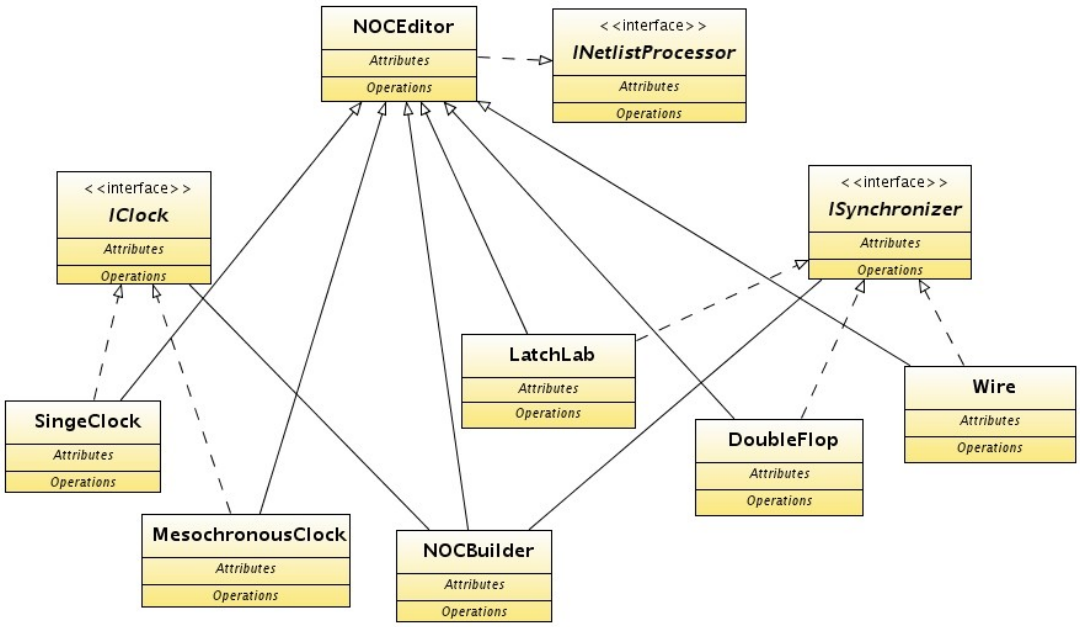

Figure 4.1: vash package class diagram

4.2.2 chipit package

The *chipit* package was developed in order to produce a description of the signal interconnections between FPGAs. This description can be dumped in *comma-separated-values* (csv) format so that it can later be directly imported into the design entry tools provided by the emulation machine manufacturer. Figure 4.2 shows the class diagram for this package.

As the generation of the NoC progresses, new *Signal* instances are added to the *ConnectionMatrix* class. *ConnectionMatrix* associates each *Signal* instance with a name so it can easily found when necessary. This class also keeps a map that relates every *Signal* instance to a set of signal connections. Each subclass of *Signal* is appropriately overridden so that, when the description is dumped, the correct signal type is produced in the output csv file.

4.2.3 parser package

The *parser* package contains the classes that hold the input parameters for the generation of a NoC. Figure 4.3 shows the class diagram for this package. The *INetworkInfo* interface provides a method to query a named signal mapping. This mapping describes the names of the signals that should be connected together to bind the ports of two neighbor switches. For instance, the class *SimpleNetwork* holds the mappings for the current switch, and, for each of its ports (north, west, south, east), it holds

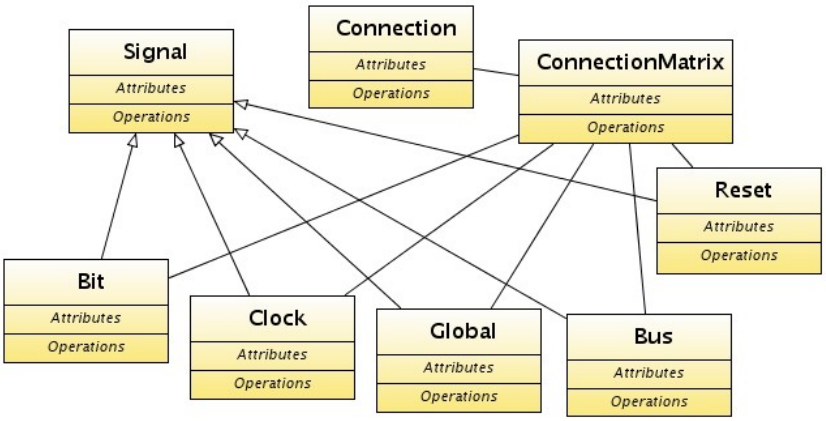

Figure 4.2: chipit package class diagram

several mapping entries so that, for example, the "north_tx_data" signal in one switch is bounded to the "south_rx_data" signal of the switch to the north.

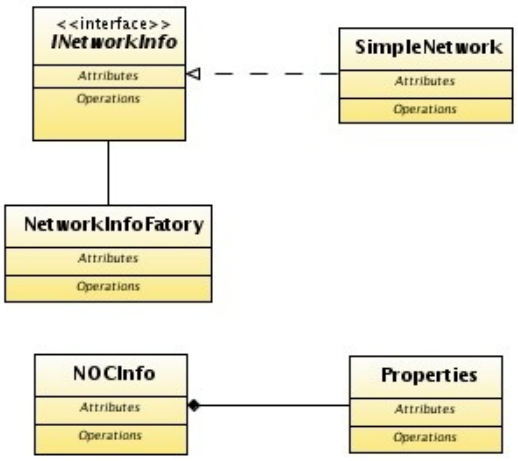

Figure 4.3: parser package class diagram

The *NOCInfo* class wraps a *java.util.Properties* instance in order to read the configuration parameters from a simple text file. The supported parameters are described below.

rows: The number of FPGAs en the Y axis.

columns: The number of FPGAs en the X axis.

rowX(with X from 0 to rows-1): the number of rows in this FPGA.

columnX (with X from 0 to columns-1): the number of columns in this FPGA.

switch = simplerouter: the name of the switch to use.

freq: NOC clock frequency.

packet_length: number of flits per packet (1 header plus data flits).

flit_size: number of bits in a flit.

pulse (true/false): if true, a pulse detector for the ack signal is instantiated in every port of the switch. Likewise, all ack signals are sent in two clock cycles. This must be set to true is the synchronizer is set to DOUBLE_FLOP.

counter_value: number of cycles the packet generator waits before sending another packet

clock_scheme (SINGLE/MESOCHRONOUS) SINGLE produces a single clock signal for all switches.

synchronizer_type (WIRE, DOUBLE_FLOP, LATCH_LAB): WIRE: no synchronizer, DOUBLE_FLOP: two cascaded flip-flop, LATCH_LAB: latch synchronizer.

phaseX: defines the clock phase for the X node. The nodes are named in ascending order, from left to right and bottom to top. For example, in a 2x2 network the node names are would be:

node2	node3
node0	node1

4.2.4 Generics support

The DfX framework was originally developed to store a gate-level internal representation of a VHDL circuit description. After a first compilation step, where the VHDL source code is parsed, both the entity and the implementing architectures are elaborated to produce a netlist. The netlist holds a set of gates and the signals to interconnect them. Furthermore, the *Netlist* class also extends the *Gate* class so as to make use of the composite design pattern to support hierarchical architectures. However, when an entity is elaborated, any generics parameter is stored as an internal constant to be used during the architecture elaboration. This approach is not well suited for placing and configuring hardware blocks as complex *Gate* instances.

In order to circumvent this issue without an extensive design revision, the relevant DfX classes were modified in order to build a netlist by elaborating only a VHDL entity. The resulting netlist has no internal gates but contains the ports that enable its connection to other hardware blocks. During the entity elaboration, the netlist is annotated with generics values so these can also be dumped together with the rest of the component information. The netlist annotations do not retain type information, that is, they simply map two *String* values to describe the formal and actual generics parameters.

One disadvantage of this approach is that, due to the entity elaboration, all the names in the port declaration are resolved. This does not allow the use of the generics values as a part of the port declaration. As an example, this solution does not support defining the length of a bit vector as

a function of one of the given generics values. For these situations it is recommended that the configuration parameter be left out of the generics list and included as a static constant in a separate package.

4.3 Timing margins

For the correct operation of the latch synchronizer, equations 3.1.1a, 3.1.1b, 3.1.1c need to be observed. The following analysis provides a first approximation of the circuit performance in the underlying Virtex-II FPGA technology.

The electrical characteristics of the Virtex-II FPGAs [Xil07] are summarized below:

$$\begin{aligned} T_{ILO} &= 0.44\,ns \text{ (Lookup table: F/G to X/Y outputs)} \\ T_{CKLO} &= 0.68\,ns \text{ (Latch clock to XQ/YQ outputs)} \\ T_{DICK} = T_{DYCK} = T_{DXCK} &= 0.27\,ns\ (t_{latchhold}) \\ T_{CKO} &= 0.57\,ns \text{ (CLK to XQ/YQ outputs)} \end{aligned} \tag{4.3.1}$$

On the one hand, for intra-FPGA communication it is always possible to use the transmitter clock buffer for the clk_sender signal, so $t_{routingskew} = 0$, while on the other hand, the off-chip routing delay has to be taken into account for inter-FPGA communication. The *srt_entry* [Pro] tool makes it possible to place a priority constraint on certain signals so that the higher the priority of a signal, the faster the physical link obtained. This constraint could be applied to the clk_sender signal in order to guarantee $t_{routingskew} \leq 0$.

The logic to drive the *latch_enable* signals can be implemented with one 4-input function, so t_{cond} becomes T_{ILO}. Given that for both inter- and intra-FPGA communication $t_{routingskew} \leq 0$, in both cases 3.1.1a can be safely reduced to $t_{cond} + t_{latchhold} < t_{datamin}$. Now, $t_{datamin}$ is at least T_{CKO}, so $t_{datamin} = T_{CKO} + t_{delay}$ and, applying the reduced equation 3.1.1a:

$$\begin{aligned} T_{ILO} + T_{DICK} &< T_{CKO} + t_{delay} \\ 0.44\,ns + 0.37\,ns - 0.57\,ns &< t_{delay} \\ 0.24\,ns &< t_{delay} \end{aligned} \tag{4.3.2}$$

From equation 4.3.2 it follows that the incoming data can be delayed with a single LUT buffer ($T_{ILO} > 0.24\,ns$).

If the critical path of the network is set by the inter-FPGA interconnection delay, equation 3.1.1b defines the maximum switch clock frequency. However, in order to use equation 3.1.1b, it is necessary to estimate $t_{routingskew}$ and $t_{datamin}$. According to the architecture of the CHIPit Platinum platform with 6 FPGAs, there can be at most three routing switches between any two FPGAs, and, as specified by the *str_timing* tool[Pro], these three switches make up an interconnection delay of 5 ns, while the minimum

possible delay (no switches at all) is 1 ns. This results in the worst case, where the clock signals are routed through the fastest physical link and at least one other network signal through the slowest link.

$t_{datamax}$ is, in turn, composed of $t_{datamin}$, the IOB output delay (IOB_{output}) and the IOB input (IOB_{input}) of the transmitting and receiving FPGA. According to [Xil07], these delays for LVTTL with 12 mA and slow slew rate are $IOB_{output} = 4.67\,ns$ and $IOB_{input} = 2.66\,ns$ so equation 3.1.1b becomes:

$$\begin{aligned} t_{clk} + T_{ILO} - 4\,ns &> IOB_{output} + IOB_{input} + T_{ILO} + T_{CKO} + 5\,ns + t_{latchsetup} \\ t_{clk} &> 14.31\,ns \end{aligned} \tag{4.3.3}$$

This results in a operating frequency of almost 70 MHz. However, this estimation may be too optimistic since it does not consider the routing delay from the input/output register to the input/output IO pad.

For intra-FPGA communication, $t_{routingskew} = t_{routing_{clk}}$ and $t_{datamax} = T_{CKO} + t_{routing_{data}}$. Moreover, $t_{routing_{clk}}$ can be safely assumed to be zero if the dedicated clock buffers are used for the transmitting clock. This means equation 3.1.1b becomes:

$$\begin{aligned} t_{clk} + T_{ILO} + t_{routing_{data}} &> T_{CKO} + t_{routing_{data}} + t_{latchsetup} \\ t_{clk} \setminus 2 - 0.5\,ns &> t_{routing_{data}} \end{aligned} \tag{4.3.4}$$

For a maximum supported frequency of 100 MHz, $t_{routing_{data}}$ should not be greater than 6.5 ns. This condition should be easily achieved by the synthesis tool.

In order to safely store data in the back-end, the initialization process reads the data in the 1-bit counter in the front-end and sets the 1-bit counter in the back-end so as to always sample a stable latch. A timing violation may occur when driving the 1-bit counter in the back-end from a signal in a different clock domain. To avoid a metastable condition, the phase detector monitors the phase difference between the two clock domains and produces a delayed version of the front-end 1-bit counter. This delayed signal can be used to avoid any hold and setup violations in the back-end counter.

4.4 FPGA primitive instantiation and guided place and route

Most of the hardware blocks used in the VASH framework are purely synchronous. Their functionality can be completely specified at the Register Transfer Level using a Hardware Description Language (HDL). Moreover, the semantics of the behavioral VHDL code makes use of the synchronous assumption and implies implementation directives for the synthesis tool to produce the desired hardware. In other words, as long as certain conditions hold, for instance, the critical path of the circuit does not exceed the operation frequency, the design is guaranteed to work as planned.

The latch synchronizer, however, drops the synchronous assumption. This prevents the functionality of the circuit from being described in behavioral VHDL code. This is why, for the development of this synchronizer, a hierarchical design style was used and some FPGA primitives were manually included

in the circuit description. Given that the synthesis tool cannot infer the intention of the designer from the processed circuit description, timing closure can only be obtained by placing meaningful constraints on the synthesizer, mapper and placer. Fortunately, the Xilinx software tools allow the use of a rich set of such implementation constraints [Xilb], which have been extensively used to fine-tune the synchronizer operation.

The *RLOC* constraint is used on almost all instantiated FPGA primitives. It forces the mapper and placer to fix the position of the target FPGA primitives relative to each other. This is useful to take advantage of the shortest routing path between two elements but, given that the primitive location has to be planned beforehand, a subtle change in the circuit functionality requires great design effort. This is not well suited for large projects where incremental design is mandatory.

The *KEEP* constraints is useful when placing delay buffers. It can be set on a signal to prevent the synthesis tool from optimizing away the driving logic. The *KEEP_HIERARCHY* constraint, on the other hand, prevents the synthesis tool from merging the logic across architecture boundaries. This directive is passed on to the placer so that the produced simulation model respects the designer's original hierarchy structure.

The *USELOWSKEWLINES* directs the router to assign a dedicated low skew resource to the target signal. FPGAs only have a fixed amount of low skew lines at certain locations on the chip. This is why there is no guarantee that the router will be able to meet this constraint for all signals.

The *MAXDELAY* constraint specifies the maximum allowed delay for a net. However, it was found that its use, even when *RLOC* constraints are also applied, did not guarantee that two signals exhibit a similar routing delay.

The *AREA_GROUP* constraint is useful to isolate a piece of logic so that no other hardware components can share the same on-chip resources. This constraint was used on the phase detector and latch synchronizer to make the behavior of their physical realization as similar as possible to the behavior in the simulation model.

4.4.1 Structural design style example

Listing 4.1 shows a code excerpt from the latch synchronizer source code. In order to instantiate the FPGA primitives, the *vcomponents* package from the *unisim* library has to be included at the top of the source file (not shown in listing 4.1). The top part of listing 4.1 shows the instantiation of a toggle flip-flop. The FDCP component is directly mapped to an on-chip flip-flop and all its formal ports must be bounded to user signals declared in this architecture's declarative part. These signals can be assigned using any design style meaningful to the synthesis tool, with a concurrent signal assignment as shown in the first line of the listing, for example. The bottom part of listing 4.1 shows how place and route constraints can be embedded in the description of the hardware. This piece of code uses a *generate* statement to implement a variable number of two-input multiplexers. As can be seen in the declarative part of the *generate* statement, the loop variable can be used inside the location constraint definition. This level of flexibility makes it easier to make design chances in a controlled, regular way, and was employed as often as possible.

Listing 4.1 Latch synchronizer code style example

```
be_cnt_inv <= not be_cnt;

BE_FF : FDCP
    generic map (INIT => '0')
    port map (Q   => be_cnt,
              C   => be_internal_clk,
              CLR => be_clr,
              D   => be_cnt_inv,
              PRE => be_pre);

MUX_DATA : for i in INPUTS-1 downto 0 generate
    attribute RLOC of MUX : label is "R" & "C" & natural'image(3-i);
begin

    MUX : process ( top_latch, bottom_latch, be_cnt)
    begin -- process MUX

        case be_cnt is
            when '1' =>
                data_mux(i) <= top_latch(i);
            when others =>
                data_mux(i) <= bottom_latch(i);
        end case;

    end process MUX;

end generate MUX_DATA;
```

4.4.2 Validation infrastructure

As mentioned in the previous section, the same software API can be used for emulation and simulation. The difference is made during linkage, where different shared libraries may be used. Algorithm 4.1 shows the steps followed to validate and measure the performance of the generated NoC.

The first action in every loop iteration is a phase adjustment in a given node. This allows the thorough validation of the network operation under many different phase relationships. The phase of a node is changed by instructing the local DCM to gradually increase or decrease the phase of the output clock signal. However, as pointed out in Xilinx Answer Records AR#13349[XARa] and AR#15130[XARb], special care must be taken to ensure correct operation in the negative phase range. For fixed mode operation, a workaround in the multiplexing of the clock signals is mandatory, while in variable mode, a special option must be passed to the bitstream generation tool. Unfortunately, the Xilinx CoreGenerator does not support any of this fixes and the ProDesign synthesis scripts do not easily manage bitstream options. This is why, the NoC validation was always performed using positive clock phases.

Algorithm 4.1 Validation software pseudocode

```
procedure MAIN
    loop
        phase_adjust()
        for i = 1 to NODES do
            set_up_node(i)
        end for
        global_enable()
        repeat
            for i = 1 to NODES do
                dsctr ← read_descriptor(i)
                if dsctr = INCOMING then
                    read_timestamp(i)
                    read_packet(i)
                    update_receive_list()
                else
                    if dsctr = OUTGOING then
                        read_timestamp(i)
                        read_packet(i)
                        update_sender_list()
                    end if
                end if
            end for
        until NO_MORE_PACKETS
        for Packet s : sender list do
            for Packet r : receiver list do
                compute_latency(s, r)
            end for
        end for
    end loop
end procedure
```

The clock frequency used in the NoC validation was at most 50 Mhz. The DCM is able to produce 64 distinct phase offsets in a positive range of 5 ns. The difference between any successive phase offsets is always the same in this range.

Before any measurement takes places, each network node must be correctly configured. For this purpose, a control data work is written from the host PC as presented in Table 3.3.

Once the global enable is activated, all nodes start operating at exactly the same time. At this point one byte is read from the target node to find out its status. If the node has a pending packet, it is read out and, depending on the packet type, stored in the sender or receiver list. After one whole packet at most is read from a node, the next node is polled and processed.

When no node in the network reports a pending packet within a loop iteration, the network is assumed to be idle and the latency calculation step begins. Each entry in the sender list is matched to an entry in the receiver list, their timestamps are compared, and the packet latency is calculated. If an entry in any list does not have a matching entry in the other list, a packet has been lost or corrupted and the NoC validation fails.

4.5 Area and latency comparison

Several NoCs with varying sizes were validated in the emulation machine. It was found that if the minimum clock cycle latency is observed in the NI, the network is able to transfer data packets without errors or deadlocks. In the case of the latch synchronizer, the phase configuration had to be set up manually. This required verifying each network link separately and changing the synchronizer phase configuration in case a problem was encountered. For this purpose, the network was set up so that all nodes receive packets but only one node is allowed to send them. Given that the X-Y routing algorithm is deterministic, missing packets can be traced back to a failing link.

Tables 4.1 and 4.2 show the measured latency in networks of different size using the double flip-flop and the latch synchronizer respectively. As the tables show, depending on the network size, the nodes were distributed over 1, 4 or 6 FPGAs. In each experiment, the sequential traffic pattern described in section 3.1.1 was used to transmit 25 packets per node, and the validation infrastructure of section 4.4.2 were employed to verify the network's correct functionality.

Table 4.1: Double flip-flop latency measurements

Size	FPGAs	Latency	Max. Frequency
2x2	1	36	85 MHz
3x3	1	147	85 MHz
4x4	4	202	58 MHz
5x5	6	387	65 MHz
6x6	6	696	65 MHz
7x7	6	1125	61 MHz

Table 4.2: Latch synchronizer latency measurements

Size	FPGAs	Latency	Max. Frequency
2x2	1	28	75 MHz
3x3	1	123	69 MHz
4x4	4	157	42 MHz
5x5	6	322	45 MHz
6x6	6	345	55 MHz
7x7	6	640	45 MHz

The tables also show the maximum achievable frequency in each experiment. The synthesis reports show that the critical path of the system is exercised during the port arbitration process in each network switch. This theoretical maximum frequency was 90 MHz, very close to the one obtained with the double flip-flop synchronizer when the design was implemented in a single FPGA. As expected, when more FPGAs are used, some ports need to be mapped into fixed IO buffers, incurring in longer wire delays. This produced a reduction in maximum frequency for four and six FPGAs. Interestingly enough, for both synchronization schemes, the lowest clock frequency was found in a 4x4 network on 4 FPGAs. Further insight into the mapping and routing algorithms, as well as into the interconnections between the FPGAs are needed to assess the reason for this behavior.

In the case of the latch synchronizer, the maximum clock frequency is lower and, given that the synchronous assumption was broken, the synthesis tool can no longer give an accurate description of the critical path. Also, the achieved maximum frequency was considerably lower than the performance estimations given in section 4.3. Further observation of the timing report suggests that the routing delay within each FPGA is grater than expected. As in the double flip-flop case, the maximum frequency is lower when more than one FPGA is used in the design.

Figure 4.4 shows a direct latency comparison between the two synchronization strategies. With only 4 nodes in the network the latency difference is almost negligible. However, as the network size grows, the latch synchronizer substantially outperforms the double flip-flop synchronizer. For example, in a 6x6 network the latch synchronizer takes almost half the clock cycles to transmit the same amount of packets. The network traffic in these experiments was close to the worst possible case in terms of contention and is not very representative of a real application data traffic. However, it is useful to establish an upper bound to the performance gain when using the latch synchronizer.

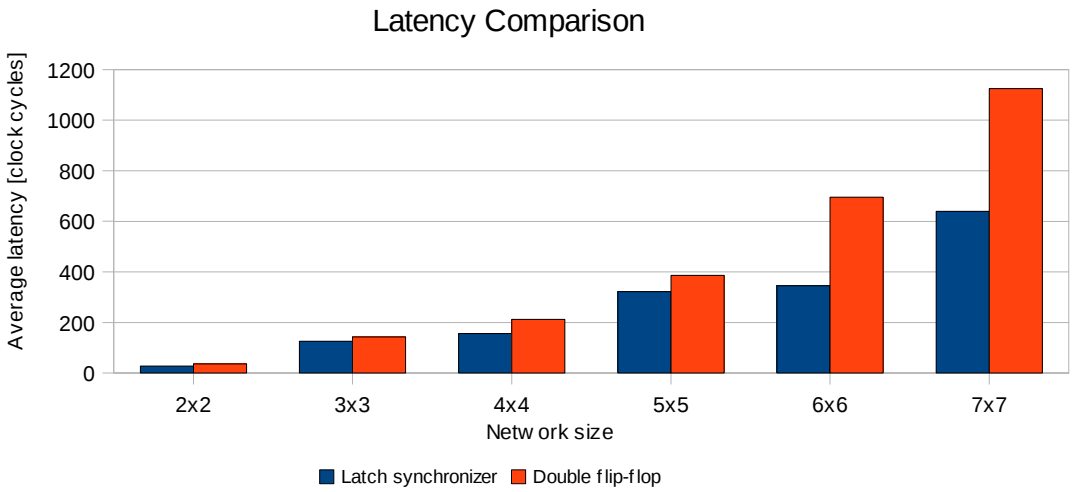

Figure 4.4: NoC latency comparison

Table 4.3 shows the amount of resources needed to implement several subnetworks in a single FPGA. The first column shows the number of network nodes that were synthesized in the corresponding row. Given that the target FPGA consists of 67584 LUTs and flip-flops, that is to say, almost five times more than the required resources in the largest 3x4 subnetwork, the networks using the smaller double flip-flop synchronizer do not show a significant area advantage over those with the latch synchronizer. The device utilization was in the worst case 22%. This leaves enough room in the FPGA to further develop all network components.

Table 4.3: Device utilization and comparison for one FPGA

Number of Nodes per FPGA	Number of flip-flops (Double flip-flop)	Number LUTs (Double flip-flop)	Number of flip-flops (Latch synchronizers)	Number of LUTs (Latch Synchronizer)
4	2807	5031	5387	2888
9	6766	12370	7079	14171
12	6199	12343	6687	13993

Chapter 5

Conclusion

In this Master Thesis, a framework to assist the rapid prototyping of NoC architectures has been developed. The VASH framework enables the FPGA emulation and functional validation of a NoC at the physical level and supports a mesochronous communication style. This emulation approach provides accurate results with a significant speedup over simulation models.

Current FPGA technology offers devices large enough to hold a large number of processing units and their interconnection matrix; the VASH framework is able to produce synthesizable VHDL to describe networks large enough to span several FPGAs. VHDL generics are widely used to set up the network behavior at elaboration time. VASH is written in Java programming language under an open license and, as a result, is flexible enough to manage many architectural choices in the NoC design space.

The hardware blocks in the emulated NoC support 38-bit flits, 32-bit phits, wormhole switching and X-Y routing and on-off flow control. They were implemented on a 6-FPGA emulation platform using Xilinx XC2V3000-6 devices. Given the clock tree distribution challenges in current VLSI manufacturing, two different synchronizer circuits were implemented to account for the clock skew over long interconnection links. The first synchronizer circuit consists of two cascaded flip-flops to reduce the probability of a synchronization failure. Although this synchronizer effectively enables the data transfer between clock domains, its use comes with a latency penalty. This penalty stems from the special manipulation of the request and grant signals needed for flow control, namely, each signal has to be kept active for at least two clock cycles and its control logic must be able to handle a delayed signal transition. This comes as a consequence of a stable but unknown output at the synchronizer.

The second synchronizer avoids any indeterminism in the link communication by observing the phase difference in the receiving and sending domain, and initializing two counter values before any communication takes place.

The emulation experiments show that, although the latch-based synchronizer works at a lower clock frequency, the latency overhead of the double flip-flop may impose a considerable performance penalty. This is especially true for large networks under high contention.

The VASH framework also opens the possibility to evaluate the NoC paradigm under realistic conditions and limited design effort; the significant time savings would make it possible to study the upper layers of the network protocol stack and its impact on application-level performance.

5.1 Recommendations and future work

The high level programming language used to develop VASH and the open source nature of the platform make it possible to extend the current functionality in order to cover a wider range of implementation options.

On the software side, the ability to instantiate hardware components programmatically and the control over their interconnections enable the validation of test and diagnosis infrastructures for NoCs. The VASH framework can be easily extended to support the necessary hardware not only to apply and evaluate test vectors, but also to inject complex faults in the interconnection links. The VASH framework also enables the use of different network topologies. For instance, there is no limitation that prevents wrapping the links around and producing k-ary n-cubes (tori). For any new topology to work, it should also be supported by the routing algorithm in the network switch. In order to further customize the network behavior or use more complex hierarchical blocks, it might be advantageous to retain type safety in the support of VHDL generics. This may be achieved by annotating the entity netlist with an elaborated type. In this case, the elaboration process must be partially performed as to include only the port and generic elements in the entity.

On the hardware side, VASH platform could be further developed to adopt a standard interconnection interface between a processing unit and the network interface. OCP and Wishbone have been used in this context before, and they would effectively enable the seamless integration of processor elements and, therefore, the evaluation of the higher software layers of the protocol stack.

As for performance, some architectural optimizations could be employed on the VASH building blocks. As demonstrated by [EEL07], the maximum clock frequency in the network switch can be substantially higher if the arbitration process is heavily pipelined. In addition, the output buffer storage in each port of the network switch can be increased to support flow control and flit transfers every clock cycle. This poses no problem in a purely synchronous system, but, if a synchronizer is used, its current implementation may need to be revised in order to cover an entire flit.

Bibliography

[BB03] D. Bertozzi, L. Benini. Xpipes. A network-on-chip architecture for gigascale systms-on-chip. In *Circuits and Systems Magazine, IEEE*. 2003. (Cited on page 7)

[Bla91] U. Black. *OSI: A Model for Computer Communications Standards*. Prentice Hall, 1991. (Cited on page 12)

[Cha84] D. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. Ph.D. thesis, Stanford University, 1984. (Cited on page 20)

[DGC07] R. R. Dobkin, R. Ginosar, I. Cidon. QNoC Asynchronous Router with Dynamic Virtual Channel Allocation. *Networks-on-Chip, International Symposium on*, p. 218, 2007. (Cited on page 14)

[EDH06] S. Evain, J.-P. Diguet, D. Houzet. NoC design flow for TDMA and QoS management in a GALS context. *EURASIP J. Embedded Syst.*, 2006(1):4–4, 2006. (Cited on page 19)

[EEL07] Ehliar, J. Eilert, D. Liu. A comparison of three FPGA optimized NoC architectures. In *Swedish System-on-Chip Conference (SSoCC)*. 2007. (Cited on pages 7, 14 und 54)

[GDR05] K. Goossens, J. Dielissen, A. Radulescu. The Ã†thereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design & Test of Computers*, 22:31–31, 2005. (Cited on page 7)

[HN06] C. Hilton, B. Nelson. Pnoc: a flexible circuit switch noc for FPGA-based systems. In *Computers and Digital Techniques, IEEE Proceedings*. 2006. (Cited on pages 7 und 14)

[Imd] "University of Rostock - Institute of Applied Microelectronics and Computer Engineering". *http://www.imd.uni-rostock.de/noc/*. (Cited on pages 9 und 27)

[KGGV07] M. Krstic, E. Grass, F. Gürkaynak, P. Vivet. Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook. *IEEE Design and Test of Computers*, 25:430–441, 2007. (Cited on page 8)

[KHT07] S. Kubisch, E. Heinrich, D. Timmermann. A Mesochronous Network-on-Chip for an FPGA. In *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS)*, pp. 113–120. 2007. (Cited on pages 28 und 29)

[LAB08] I. Loi, F. Angiolini, L. Benini. Developing Mesochronous Synchronizers to Enable 3D NoCs. In *Design, Automation and Test in Europe*, pp. 1414–1419. 2008. (Cited on pages 9, 31 und 33)

[LLY03] K. Lee, S.-J. Lee, H.-J. Yoo. A distributed crossbar switch scheduler for on-chip networks. *Custom Integrated Circuits Conference. IEEE Proceedings*, 25:671–674, 2003. (Cited on page 14)

[LZJ06] Z. Lu, M. Zhong, A. Jantsch. Evaluation of on-chip networks using deflection routing. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 296–301. 2006. (Cited on page 17)

[MB06] G. D. Micheli, L. Benini. *Networks on chips*. Morgan Kaufmann, 2006. (Cited on pages 5 und 12)

[NoC] "NoCem - Network on Chip emulator". *http://www.opencores.org/projects.cgi/web/nocem/overview*. (Cited on page 27)

[PFT+07] L. A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, S. Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design and Test of Computers*, 24:454–463, 2007. (Cited on pages 14 und 21)

[Pro] ProDesign. *CHIPit Platinum Edition : ASIC Emulation and Rapid Prototyping System. Handbook*. (Cited on page 44)

[Pro07] ProDesign. *UMRBus Communication System*, 2007. (Cited on page 40)

[STN02] D. Signenza-Tortosa, J. Nurmi. Proteo:A New Approach to Network-on-Chip. In *Proceedings of the IAS TED International Conference on Communication Systems and Networks (CSN'2)*. 2002. (Cited on page 14)

[TGL07] P. Teehan, M. Greenstreet, G. Lemieux. A Survey and Taxonomy of GALS Design Styles. *Design & Test of Computers, IEEE*, 24:418–428, 2007. (Cited on pages 20 und 21)

[VLC+07] P. Vivet, D. Lattard, F. Clermidy, E. Beigne, C. Bernard, Y. Durand, J. Durupt, D. Varreau. FAUST, an Asynchronous Network-on-Chip based Architecture for Telecom Applications. In *Design, Automation and Test in Europe*. 2007. (Cited on pages 7, 14 und 21)

[XARa] "Xilinx Answer Record 13349". *http://www.xilinx.com/support/answers/13349.htm*. (Cited on page 47)

[XARb] "Xilinx Answer Record 151310". *http://www.xilinx.com/support/answers/15130.htm*. (Cited on page 47)

[Xila] Xilinx. *Application Note XAPP094*. (Cited on page 31)

[Xilb] Xilinx. *Xilinx Constraints Guide*, 9.1i edition. (Cited on page 46)

[Xil04] Xilinx. *Clock and Data Recovery With Coded Data Streams*, 2004. (Cited on page 35)

[Xil07] Xilinx. *Virtex-II Platform FPGAs: Complete Data Sheet*, 2007. (Cited on pages 44 und 45)

[ZKCS02] C. Zeferino, M. Kreutz, L. Carro, A. Susin. A study on communication issues for systems-on-chip. In *Symposium on Integrated Circuits ans Systems Design Proceedings, IEEE*. 2002. (Cited on page 7)

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Alejandro Cook)