Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE @DIRECT®

INTEGRATION, the VLSI journal 38 (2004) 69–93

INTEGRATION
the VLSI journal

www.elsevier.com/locate/vlsi

HERMES: an infrastructure for low area overhead packet-switching networks on chip

Fernando Moraes*, Ney Calazans, Aline Mello, Leandro Möller, Luciano Ost

Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS), Av. Ipiranga, 6681, Prédio 30/BLOCO 4, 90619-900 Porto Alegre, RS, Brazil

Received 31 July 2003; received in revised form 14 January 2004; accepted 19 March 2004

Abstract

The increasing complexity of integrated circuits drives the research of new on-chip interconnection architectures. A network on chip draws on concepts inherited from distributed systems and computer networks subject areas to interconnect IP cores in a structured and scalable way. The main goal pursued is to achieve superior bandwidth when compared to conventional on-chip bus architectures. This paper reviews the state of the art in networks on chip. Then, it describes an infrastructure called Hermes, targeted to implement packet-switching mesh and related interconnection architectures and topologies. The basic element of Hermes is a switch with five bi-directional ports, connecting to four other switches and to a local IP core. The switch employs an XY routing algorithm, and uses input queuing. The main design objective was to develop a small size switch, enabling its immediate practical use. The paper also presents the design validation of the Hermes switch and of a network on chip based on it. A Hermes NoC case study has been successfully prototyped in hardware as described in the paper, demonstrating the functionality of the approach. Quantitative data for the Hermes infrastructure is advanced.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Network on chip; System on a chip; Core based design; Switches; On-chip interconnection

1. Introduction

Increasing transistor density, higher operating frequencies, short time-to-market and reduced product life cycle characterize today's semiconductor industry scenery [1]. Under these conditions, designers are developing ICs that integrate complex heterogeneous functional elements into

*Corresponding author. Tel.: +55-51-3320-3611; fax: +55-51-3320-3621.
E-mail addresses: moraes@inf.pucrs.br (F. Moraes), calazans@inf.pucrs.br (N. Calazans), alinev@inf.pucrs.br (A. Mello), moller@inf.pucrs.br (L. Möller), ost@inf.pucrs.br (L. Ost).

0167-9260/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.vlsi.2004.03.003

a single chip, known as a system on a chip (SoC). As described by Gupta et al. [2] and Bergamaschi et al. [3], SoC design is based on intellectual property (IP) cores reuse. Gupta et al. [2] define *core* as a pre-designed, pre-verified hardware piece that can be used as a building block for large and complex applications on an IC. Examples of cores are memory controllers, processors, or peripheral devices such as MAC Ethernet or PCI bus controllers. Cores may be either analog or digital, or even be composed by blocks using technologies such as microelectromechanical or optoelectronic systems [1,4]. Cores do not make up SoCs alone; they must include an interconnection architecture and interfaces to peripheral devices [4]. The interconnection architecture includes physical interfaces and communication mechanisms, which allow the communication between SoC components to take place.

Usually, the interconnection architecture is based on dedicated wires or shared busses. *Dedicated wires* are effective for systems with a small number of cores, but the number of wires around the core increases as the system complexity grows. Therefore, dedicated wires have poor reusability and flexibility. A *shared bus* is a set of wires common to multiple cores. This approach is more scalable and reusable, when compared to dedicated wires. However, busses allow only one communication transaction at a time, thus all cores share the same communication bandwidth in the system and scalability is limited to few dozens IP cores [5]. Using separate busses interconnected by bridges or hierarchical bus architectures may reduce some of these constraints, since different busses may account for different bandwidth needs, protocols and also increase communication parallelism. Nonetheless, scalability remains a problem for hierarchical bus architectures.

According to several authors, e.g. [5–9], the interconnection architecture based on shared busses will not provide support for the communication requirements of future ICs. According to ITRS, ICs will be able to contain billions of transistors, with feature sizes around 50 nm and clock frequencies around 10 GHz in 2012 [1]. In this context, a *network on chip* (NoC) appears as a probably better solution to implement future on-chip interconnects. An NoC is an on-chip network [8] composed by cores connected to switches, which are in turn connected among themselves by communication channels.

The rest of this paper is organized as follows. Section 2 presents basic concepts and features associated to NoCs. Section 3 presents an overview of current state of the art in NoCs, with emphasis on implemented approaches. A minimal NoC communication protocol stack is discussed in Section 4. Section 5 details the main contribution of this work, the proposal of an NoC infrastructure centered on a switch designed for packet-switching mesh and related interconnection architectures. An example NoC implementation and its functional validation are described in Section 6. In Section 7, some quantitative data regarding the Hermes[1] infrastructure are depicted, while Section 8 presents some conclusions and directions for future work.

2. NoC basic concepts and features

As described in [10,11], NoCs are emerging as a solution to the existing interconnection architecture constraints, due to the following characteristics: (i) energy efficiency and reliability

[1] In Greek mythology, Hermes is the messenger of Gods.

[7]; (ii) scalability of bandwidth when compared to traditional bus architectures; (iii) reusability; (iv) distributed routing decisions [8,9].

End to end communication in a system is accomplished by the exchange of *messages* among IP cores. Often, the structure of particular messages is not adequate for communication purposes. This leads to the concept of packet [12]. A *packet* is a standard form for representing information in a form adequate for communication. One packet may correspond to a fraction, one or even several messages. In the context of NoCs, packets are frequently a fraction of a message. Packets are often composed by a header, a payload, and a trailer. To ensure correct functionality during message transfers, an NoC must avoid deadlock, livelock and starvation [12]. *Deadlock* may be defined as a cyclic dependency among nodes requiring access to a set of resources so that no forward progress can be made, no matter what sequence of events happens. *Livelock* refers to packets circulating the network without ever making any progress towards their destination. It may be avoided with adaptive routing strategies. *Starvation* happens when a packet in a buffer requests an output channel, being blocked because the output channel is always allocated to another packet.

Two parts compose an interconnection network: the *services* and the *communication system*. Rijpkema et al. [10] define several *services* considered essential for SoC design, such as data integrity, throughput and latency guarantees. The implementation of these services is often based on protocol stacks such as the one proposed in the ISO OSI reference model. As mentioned in [5,8], when applied to NoCs the lower three layers (physical, link, and network) are technology dependent. The *communication system*, on the other hand, is what supports the information transfer from source to target. The communication system allows that every core send packets to every other core in the NoC structure. The NoC structure is a set of switches interconnected by *communication channels*. The way switches are connected defines the *network topology*. According to the topology, networks can be classified in one of two main classes: *static* and *dynamic* [13,14]. In *static* networks, each node has fixed point-to-point connections to some number of other nodes. Hypercube, ring, mesh, torus and fat-tree are examples of networks used to implement static networks. *Dynamic* networks employ communication channels that can be (re)configured at application runtime. Busses and crossbar switches are examples of dynamic networks.

The *communication mechanism*, *switching mode*, and *routing algorithm* are functions of the network topology and are used to compose the services provided by the NoC.

The *communication mechanism* specifies how messages pass through the network. Two methods for transferring messages are *circuit switching* and *packet switching* [14]. In *circuit switching*, a path is established before packets can be sent by the allocation of a sequence of channels between source and target. This path is called a *connection*. After establishing a connection, packets can be sent, and any other communication using the allocated channels is denied, until a disconnection procedure is executed. In *packet switching*, packets are transmitted without any need for connection establishment procedures.

Packet switching requires the use of a *switching mode*, which defines how packets move through the switches. The most important modes are *store-and-forward*, *virtual cut-through* and *wormhole* [15]. In *store-and-forward* mode, a switch cannot forward a packet until it has been completely received. Each time a switch receives a packet, its contents are examined to decide what to do, implying per-switch latency. In *virtual cut-through* mode, a switch can forward a packet as soon as the next switch gives a guarantee that a packet will be accepted completely [11]. Thus, it is necessary for a buffer to store a complete packet, like in store-and-forward, but in this case with lower latency communication. The *wormhole* switching mode is a variant of the virtual cut-through mode that avoids the need for large buffer spaces. A packet is transmitted between switches in units called *flits* (flow control digits—the smallest unit of flow control). Only the header flit has the routing information. Thus, the rest of the flits that compose a packet must follow the same path reserved for the header.

The *routing algorithm* defines the path taken by a packet between the source and the target. According to where routing decisions are taken, it is possible to classify the routing in *source* and *distributed* routing [12]. In *source* routing, the whole path is decided at the source switch, while in *distributed* routing each switch receives a packet and decides the direction to send it. According to how a path is defined to transmit packets, routing can be classified as *deterministic* or *adaptive*. In *deterministic* routing, the path is uniquely defined by the source and target addresses. In *adaptive* routing, the path is a function of the network traffic [12,15]. This last routing classification can be further divided into *partially* or *fully* adaptive. *Partially adaptive* routing uses only a subset of the available physical paths between source and target.

3. State of the art in NoCs

This Section is intended to provide a big picture of the state of the art in network-on-chip propositions, as currently found in the available literature. The results of the review are summarized in Table 1. The last row of Table 1 corresponds to the NoC infrastructure proposed here. In the Table, each row corresponds to an NoC proposition that could be found about which significant qualitative and quantitative implementation data were made available. The NoC implementation data considered relevant can be divided in three groups: (i) network and switch structural data, presented in the four first columns; (ii) performance data, in the following three columns; (iii) prototyping and/or silicon implementation data, in the last column. Although the authors do not pose claims about the completeness of this review, they consider it rather comprehensive.

Benini et al. made important contributions to the NoC subject area in their conceptual papers [7,8,16]. However, none of these documents contains any NoC implementation details.

A basic choice common to most reviewed NoCs is the use of packet switching, and this is not explicitly stated in the table. The exception is the aSOC NoC [19], where the definition of the route each message follows is fixed at the time of hardware synthesis. Two connected concepts, network topology and routing strategy are the subject of the first column in Table 1. The predominant network topology in the literature is the 2D Mesh. The reason for this choice derives from its three advantages: facilitated implementation using current IC planar technologies, simplicity of the XY routing strategy and network scalability. Another approach is to use the 2D torus topology, to reduce the network diameter [24]. The folded 2D torus [20] is an option to reduce the increased cost in wiring when compared to a standard 2D torus. One problem of mesh and torus topologies is the associated network latency. Three revised NoCs propose alternatives to overcome the problem. The SPIN [9,17,18] and the T-SoC [34,35] NoCs employ a fat-tree topology, while the Octagon NoC [22,23] proposes the use of a chordal ring topology. Both approaches lead to a smaller network diameter, with a consequent latency reduction. Concerning routing strategies,

Table 1
State of the art in NoCs

NoC	Topology/routing	Flit size	Buffering	IP-switch interface	Switch area	Estimated peak performance	QoS support	Implementation
SPIN—2000 [9,17,18]	Fat-tree/deterministic and adaptive	32 bits data + 4 bits control	Input queue + 2 shared output queue	VCI	0.24 mm^2 CMOS 0.13 μm	2 Gbits/s per switch	NA	ASIC layout 4.6 mm^2 CMOS 0.13 μm
aSOC—2000 [19]	2D mesh (scalable)/determined by application	32 bits	None	NA	50,000 transistors	NA	Circuit-switching (no wormhole)	ASIC layout CMOS 0.35 μm
Dally—2001 [20]	Folded 2D torus/XY source	256 bits data + 38 bits control	Input queue	NA	0.59 mm^2 CMOS 0.1 μm (6.6% of a tile)	4 Gbits/s per wire	GT-virtual channels	No
Nostrum—2001 [5,6]	2D mesh (scalable)/hot potato	128 bits data + 10 bits control	Input and output queues	NA	0.01 mm^2 CMOS 65 nm	NA	NA	NA
Sgroi—2001 [21]	2D mesh/NA	18 bits data + 2 bits control	NA	OCP	NA	NA	NA	NA
Octagon—2001 [22,23]	Chordal ring/distributed and adaptive	Variable data + 3 bits control	NA	NA	NA	40 Gbits/s	Circuit-switching	No
Marescaux—2002 [24,25]	2D torus (scalable)/XY blocking, hop-based, deterministic	16 bits data + 3 bits control	Input queue	Custom	611 slices VirtexII (6.58% area overhead XC2V6000)	320 Mbits/s per virtual channel at 40 MHz	2 virtual channels (to avoid deadlock)	FPGA VirtexII/VirtexII Pro
Bartic—2003 [26]	Arbitrary (parameterizable links)/deterministic, virtual-cut-through	Variable data + 2 bits control/link	Output queue	Custom	552 slices + 5 BRAMs VirtexII Pro for 5 bidirectional links router	800 Mbits/s per channel for 16-bit flits at 50 MHz	Injection rate control, congestion control	FPGA VirtexII Pro
Æthereal—2002 [10,11]	2D mesh/source	32 bits	Input queue	DTL (Philips proprietary standard)	0.26 mm^2 CMOS 0.12 μm	80 Gbits/s per switch	Circuit-switching	ASIC layout
Eclipse—2002 [27]	2D sparse hierarchical mesh/NA	68 bits	Output queue	NA	NA	NA	NA	No
Proteo—2002 [28–30]	Bi-directional ring/NA	Variable control and data sizes	Input and output queues	VCI	NA	NA	NA	ASIC layout CMOS 0.18 μm
SOCIN—2002 [31]	2D mesh (scalable)/XY source	n bits data + 4 bits control (parameterizable)	Input queue (parameterizable)	VCI	420 LCs APEX FPGAs (Estimated, for $n = 8$, bufferless)	1 Gbits/s per switch at 25 MHz	No	No

Table 1 (*continued*)

NoC	Topology/routing	Flit size	Buffering	IP-switch interface	Switch area	Estimated peak performance	QoS support	Implementation
SoCBus—2002 [32]	2D mesh/XY adaptive	16 bits data + 3 bits control	Single position input and output buffers	Custom	NA	2.4 Gbits/s per link	Circuit-switching	No
QNOC—2003 [33]	2D mesh regular or irregular/XY	16 bits data + 10 bits control (parameterizable)	Input queue (parameterize-ble) + Output queue (single position)	Custom	0.02 mm^2 CMOS 90 nm (Estimated)	80 Gbits/s per switch for 16-bit flits at 1 GHz	GT-virtual channels, (4 different traffic)	No
T-SoC—2003 [34,35]	Fat-tree/Adaptive	38 bits maximum	Input and output queues	Custom/ OCP	27000 to 36000 two input NAND gates	NA	GT-4 virtual channels	NA
Xpipes—2002 [36]	Arbitrary (design time)/ source static (street sign)	32, 64 or 128 bits	Virtual output queue	OCP	0.33 mm^2 CMOS 100 nm (Estimated)	64 Gbits/s per switch for 32-bit flits at 500 MHz	No	No
Hermes—2003 [37]	2D mesh (scalable)/XY	8 bits data + 2 bits control (parameterizable)	Input queue (parameterizable)	OCP	555 LUTs 278 slices VirtexII	500 Mbits/s per switch at 25 MHz	No	FPGA VirtexII

NA = data not available, GT = guaranteed throughput.

there is a clear lack of published information on specific algorithms. This indicates that further research is needed in this area. For instance, it is widely known that XY adaptive algorithms are prone to deadlock, but solutions exist to improve XY routing while avoiding deadlock risk [38].

The second important quantitative parameter of NoC switches is the flit size. From Table 1, it is possible to classify approaches in two groups, those focusing on future SoC technologies and those adapted to existing limitations. The first group includes the proposals of Dally [20] and Kumar [5,6], where wide switching channels are used (150–300 wires), without significantly affecting the overall SoC area. This can be achieved e.g. by using a future 60 nm technology for building 22 mm $\times$ 22 mm chip with a 10 $\times$ 10 NoC to connect 100 2 mm $\times$ 2 mm IPs [5]. However, this is clearly not yet feasible today. The second group comprises works with flit size ranging mostly from 8 to 64 bits, a data width similar to current processor architectures. The works providing a NoC prototype, Marescaux [24], Bartic [26] and Hermes [37], have the smallest flit sizes, 16, 16 and 8 bits, respectively.

The next parameter in Table 1 is the switch buffering strategy. Most NoCs employ input queue buffers. Since input queuing implies a single queue per input, this leads to lower area overhead, justifying the choice. However, input queuing presents the well-known *head-of-line* (HOL) blocking problem [10]. To overcome this problem, output queuing can be used [27], at a greater buffering cost, since this increases the total number of queues in the switch. An intermediate solution is to use virtual output queuing associated with time-multiplexed virtual channels, as proposed in the Xpipes NoC [36]. Another important parameter is the queue size, which implies the need to solve the compromise among the amount of network contention,[2] packet latency and switch area overhead. Bigger queues lead to small network contention, higher packet latency, and bigger switches. Smaller queues lead to the opposite situation. Section 7 exploits quantitative data regarding this compromise for the Hermes NoC.

The last structural parameter is the characteristic of the IP-switch interface. The use of standard communication interfaces for the on-chip environment is an evolving trend in industry and academia. They are devised to increase design reuse, and are accordingly seen as a needed feature to enable future SoC designs. NoCs with custom IP-switch interfaces, such as the ones proposed in [24,26,32,33], are less apt to aggregate third party IPs to the design in a timely manner. The two most prominent interface standards, VCI and OCP are each used by several of the NoC proposals presented in Table 1.

The fifth column collects results concerning the size of the switch. It is interesting to observe that two approaches targeted to ASICs [10,18], both with a 32-bit flit size, have similar dimensions, around 0.25 mm^2 for similar technologies. In addition, FPGA prototyped systems produced results ranging from 555 LUTs [37] to 1222 LUTS (611 slices) [25]. The observed difference comes from the fact that [25] employs virtual channels, while [37] does not, leading to a smaller switch area. These data seem to indicate the need to establish a relationship between switch size and SoC communication area overhead. It is reasonable to expect that the adoption of NoCs by SoC designers be tied to gains in on-chip communication performance. On the other hand, low area overhead when compared with e.g. standard bus architectures is another important issue. An SoC design specification will normally determine a maximum area overhead allowed for on-chip communication, as well as minimum expected communication performance,

[2] *Network contention* is a measure of the amount of network resources allocated to blocked packets.

possibly in an IP by IP basis. Switch size, flit size (i.e. communication channel width) and switch port cardinality are fundamental values to allow estimating the area overhead and the expected peak performance for on-chip communication. Adoption of NoCs is then tied to these quantitative assessments and to the ease with which designers are provided to evaluate the NoC approach in real designs.

Estimated peak performance, presented in the sixth column of Table 1, is a parameter that needs further analysis to provide a meaningful comparison among different NoCs. This column displays different units for different NoCs, which must accordingly be considered as merely illustrative of possible performance values. Most of the estimates are derived from the product of three values: number of switch ports, flit size, and estimated operating frequency. The wide variation of numbers is due mostly to the last two values. No measured performance data could be found in any reviewed publication. A first approach to measure the Hermes NoC performance is provided in Section 6.2. The value associated to the NoC proposed in [20] should be regarded with care. The reason for this is that the data reflects a technology limit that can be achieved by sending multiple bits through a wire at each clock cycle (e.g. 20 bits at each 200 MHz clock cycle [20]).

The next column concerns the quality of service (QoS) support parameter. The most commonly found form of guaranteeing QoS in NoCs is through the use of circuit switching. This is a way of ascertain throughput and thus QoS for a given communication path. The disadvantage of the approach is that bandwidth can be wasted if the communication path is not used at every moment during the period the connection is established. In addition, since most approaches combine circuit switching with best effort techniques, this brings as consequence the increase of the switch area overhead. This is the case for NoC proposals presented in [11,20,23]. Virtual channels are one way to achieve QoS without compromising bandwidth, especially when combined with time division multiplexing (TDM) techniques. This last technique, exemplified in [25,33,35], avoids that packets remain blocked for long periods, since flits from different inputs of a switch are transmitted according to a predefined time slot allocation associated with each switch output. It is expected that current and future SoC utilization will be dominated by streaming applications. Consequently, QoS support is a fundamental feature of NoCs.

Finally, it is possible to state that NoC implementation results are still very scarce. None of the four ASIC implementations found in the literature gives hints if the design corresponds to working silicon. On the other hand, only three NoCs have been reported to be prototyped in FPGAs, those proposed in [24,26,37].

4. NOCs protocol stack

The OSI reference model is a hierarchical structure of seven layers that define the requirements for communication among processing elements [39]. Each layer offers a set of services to the upper layer, using functions available in the same layer and in the lower ones. NoCs usually implement a subset of the lower layers, such as Physical, Data Link, Network, and Transport. These layers are described below for the NoC context.

The *physical layer* is responsible to provide mechanical and electrical media definitions to connect different entities at bit level [39]. In the present work, this layer corresponds to the

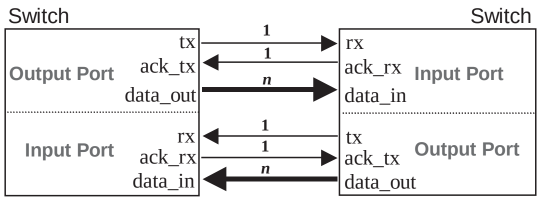


Fig. 1. Example of physical interface between switches.

communication between switches, as exemplified in Fig. 1 for the implementation proposed here. The physical data bus width must be chosen as a function of the available routing resources and available memory to implement buffering schemes. The output port in the example is composed by the following signals: (1) *Tx*: control signal indicating data availability; (2) *Data_out*: data to be sent; (3) *Ack_tx*: control signal indicating successful data reception. The input port in the example is composed by the following signals: (1) *Rx*: control signal indicating data availability; (2) *Data_in*: data to be received; (3) *Ack_rx*: control signal indicating successful data reception.

The *data link layer* has the objective of establishing a logical connection between entities and converting an unreliable medium into a reliable one. To fulfill these requirements, techniques of flow control and error detection are commonly used [12]. This work implements in the data link layer a simple handshake protocol built on top of the physical layer, to deal with flow control and correctly sending and receiving data. In this protocol, when the switch needs to send data to a neighbor switch, it puts the data in the *data_out* signal and asserts the *tx* signal. Once the neighbor switch stores the data from the *data_in* signal, it asserts the *ack_rx* signal, and the transmission is complete. Forward flow control can be used to reduce NoC latency, as proposed in Q-NoC [33]; however this requires employing synchronous communication between switches. One point favoring the use of explicit handshake protocols is the possibility to implement asynchronous interconnection between synchronous modules, enabling a Globally Asynchronous Locally Synchronous (GALS) approach. This alternative may also reduce clock skew requirements and provide lower power consumption [40].

The *network layer* is concerned with the exchange of packets. This layer is responsible for the segmentation and reassembly of flits, point-to-point routing between switches, and contention management. The network layer in this work implements the packet switching technique.

The *transport layer* is responsible to establish end-to-end communication from source to target. Services like segmentation and reassembly of packets are essential to provide a reliable communication [12]. Here, end-to-end communication is implemented in the local IPs cores.

5. Hermes switch

The main objective of an on-chip switch is to provide correct transfer of messages between IP cores. Switches usually have routing logic, arbitration logic and communication ports directed to other switches or cores. The communication ports include input and output channels, which can have buffers for temporary storage of information.

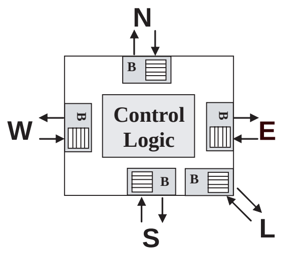


Fig. 2. Switch Architecture. B indicates input buffers.

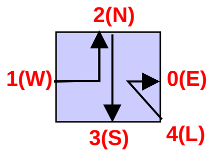
The Hermes switch has routing control logic and five bi-directional ports: East, West, North, South, and Local. Each port has an input buffer for temporary storage of information. The Local port establishes a communication between the switch and its local core. The other ports of the switch are connected to neighbor switches, as presented in Fig. 2. The routing control logic implements the arbitration logic and a packet-switching algorithm.

Among the switching modes presented in Section 2, wormhole was chosen because it requires less memory, provides low latency, and can multiplex a physical channel into more than one logical channel. Although the multiplexing of physical channels may increase the wormhole switching performance [41], this has not been implemented. The reason is to lower complexity and cost of the switch by using only one logical channel for each physical channel.

As previously described, the wormhole mode implies the division of packets into flits. The flit size for the Hermes infrastructure is parameterizable, and the number of flits in a packet is fixed at $2^{(\text{flit size, in bits})}$. An 8-bit flit size was chosen here for prototyping and evaluation purpose. The first and the second flit of a packet are header information, being respectively the address of the target switch, named *header flit*, and the number of flits in the packet payload. Each switch must have a unique address in the network. To simplify routing on the network this address is expressed in XY coordinates, where X represents the horizontal position and Y the vertical position.

5.1. Control logic

Two modules implement the control logic: *routing* and *arbitration*, as presented in Fig. 4. When a switch receives a header flit, the arbitration is executed and if the incoming packet request is granted, an XY routing algorithm is executed to connect the input port data to the correct output port. The algorithm compares the actual switch address ($xLyL$) to the target switch address ($xTyT$) of the packet, stored in the header flit. Flits must be routed to the local port of the switch when the $xLyL$ address of the actual switch is equal to the $xTyT$ packet address. If this is not the case, the xT address is first compared to the xL (horizontal) address. Flits will be routed to the East port when $xL<xT$, to West when $xL>xT$ and if $xL=xT$ the header flit is already horizontally aligned. If this last condition is true, the yT (vertical) address is compared to the yL address. Flits will be routed to South when $yL<yT$, to North when $yL>yT$. If the chosen port is busy, the header flit as well as all subsequent flits of this packet will be blocked. The routing request for this packet will remain active until a connection is established in some future execution of the procedure in this switch.



(a)

	0 (E)	1 (W)	2 (N)	3 (S)	4 (L)
Free	0	1	0	0	1
In	-	2	3	-	0
Out	4	-	1	2	-

(b)

Fig. 3. Three simultaneous connections in the switch (a), and the respective switching table (b).

When the *XY* routing algorithm finds a free output port to use, the connection between the input port and the output port is established and the *in*, *out* and *free* switching vectors at the switching table are updated. The *in* vector connects an input port to an output port. The *out* vector connects an output port to an input port. The *free* vector is responsible to modify the output port state from free (1) to busy (0). Consider the North port in Fig. 3(a). The output North port is busy (free = 0) and is being driven by the West port (out = 1). The input North port is driving the South port (in = 3). The switching table structure contains redundant information about connections, but this organization is useful to enhance the routing algorithm efficiency.

After all flits composing the packet have been routed, the connection must be closed. This could be done in two different ways: by a trailer, as described in Section 2, or using flit counters. A trailer would require one or more flits to be used as packet trailer and additional logic to detect the trailer would be needed. To simplify the design, the switch has five counters, one for each output port. The counter of a specific port is initialized when the second flit of a packet arrives, indicating the number of flits composing the payload. The counter is decremented for each flit successfully sent. When the counter value reaches zero, the connection is closed and the *free* vector corresponding position of the output port goes to one (free = 1), thus closing the connection.

A switch can simultaneously be requested to establish up to five connections. Arbitration logic is used to grant access to an output port when one or more input ports simultaneously require a connection. A dynamic arbitration scheme is used. The priority of a port is a function of the last port having a routing request granted. For example, if the local input port (index 4) was the last to have a routing request granted, the East port (index 0) will have greater priority, being followed by the ports West, North, South and Local. This method guarantees that all input requests will be eventually granted, preventing starvation to occur. The arbitration logic waits four clock cycles to treat a new routing request. This time is required for the switch to execute the routing algorithm. If a granted port fails to route the flit, the next input port requesting routing have its request granted, and the port having the routing request denied receives the lowest priority in the arbiter.

5.2. *Message buffering*

When a flit is blocked in a given switch, the performance of the network is affected, since the flits belonging to the same packet are blocked in other switches. To lessen the performance loss, a buffer is added to each input switch port, reducing the number of switches affected by the blocked flits. The inserted buffers work as circular FIFOs. In Hermes, the FIFO size is parameterizable, and a size eight has been used for prototyping purposes.

5.3. Switch functional validation

The Hermes switch was described in VHDL and validated by functional simulation. Fig. 4 presents some internal blocks of the switch and the signals of two ports (Local and East). Fig. 5 presents a functional simulation for the most important signals of Fig. 4. The simulation steps are described below, where numbering have correspondences in Figs. 4 and 5.

1. The switch ($xLyL=00$) receives a flit by the local port (index 4), signal *rx* is asserted and the *data_in* signal has the flit contents.
2. The flit is stored in the buffer and the *ack_rx* signal is asserted indicating that the flit was received.
3. The local port requests routing to the arbitration logic by asserting the *h* signal.
4. After selecting a port, the arbitration logic makes a request to the routing logic. This is accomplished by sending the header flit that is the switch target address (value 11) and the source of the input request (signal *incoming*, value 4, representing the local port) together with the request itself.
5. The XY routing algorithm is executed, the switching table is written, and the *ack_rot* signal is asserted indicating that the connection is established.
6. The arbitration logic informs the buffer that the connection was established and the flit can now be transmitted.
7. The switch asserts the *tx* signal of the selected output port and puts the flit in the *data_out* signal of this same port.
8. Once the *ack_tx* signal is asserted the flit is removed from the buffer and the next flit stored can be treated.
9. This second flit starts the counter indicating after how many clock cycles the connection must be closed.

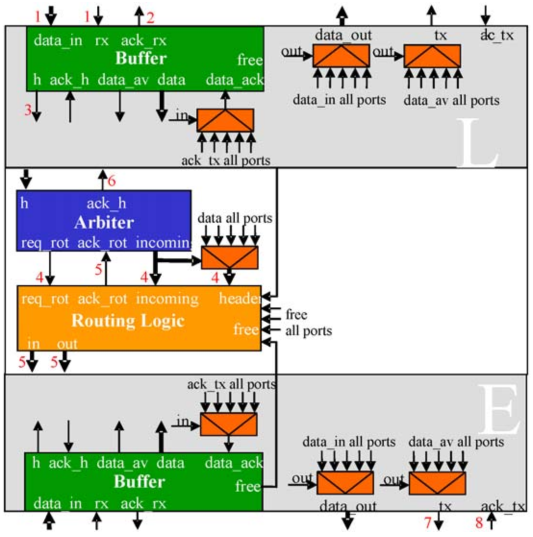


Fig. 4. Partial block diagram of the switch, showing two of the five ports. Numbers have correspondence to the sequence of events in Fig. 5.

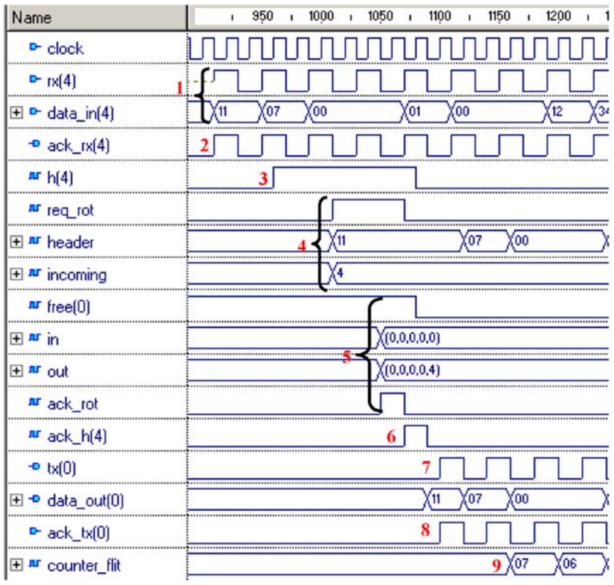


Fig. 5. Simulation of a connection between the local port and the east port.

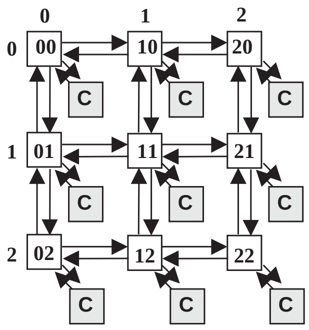


Fig. 6. 3×3 Mesh NoC structure. C marks IP cores, Switch addresses indicate the XY position in network.

6. Hermes network on chip

NoC topologies are defined by the connection structure of the switches. The Hermes NoC assumes that each switch has a set of bi-directional ports linked to other switches and to an IP core. In the mesh topology used in this work, each switch has a different number of ports, depending on its position with regard to the limits of the network, as shown in Fig. 6. For example, the central switch has all five ports defined in Section 5. However, each corner switch has only three ports.

The use of mesh topologies is justified to facilitate placement and routing tasks as stated before. The Hermes switch can also be used to build torus, hypercube or similar NoC topologies. However, building such topologies implies changes in switch connections and, more importantly, in the routing algorithm.

6.1. NoC functional validation

Packet transmission in the Hermes NoC was validated first by functional simulation. Fig. 7 illustrates the transmission of a packet from switch 00 to switch 11 in the topology of Fig. 6. The simulation shows the switch 10 input and output interface behaviors.

The simulation works as follows:

1. Switch 00 sends the first flit of the packet (address of the target switch) to the *data_out* signal at its East port and asserts the *tx* signal in this port.
2. Switch 10 detects the *rx* signal asserted in its West port and gets the flit in the *data_in* signal. It takes 10 clock cycles to route this packet (2 clock cycles to store it into the buffer, 2 for arbitration, 6 for routing). The flits that follow the header pass through the switch with a latency of 2 clock cycles each.
3. Switch 10 output South port indicates its busy state in the free(3) signal. Signals free(i) are elements of the free vector defined in Section 5.1.
4. Switch 10 puts the flit in *data_out* signal and asserts the *tx* signal of its South port. Next, Switch 11 detects asserted the *rx* signal of its North port. The flit is captured in the *data_in* signal and the source to target connection is now established.
5. The second flit of the packet contains the number of flits composing the payload.
6. After all flits are sent, the connection is closed and the free vector entries of each switch involved in the connection return to their free state.

The minimal latency in clock cycles to transfer a packet from source to target is given by:

$$latency = \left(\sum_{i=1}^{n} R_i\right) + P \times 2,$$

where n is the number of switches in the communication path (source and target included), R_i is the time required by the routing algorithm at each switch (at least 10 clock cycles), and P is the packet size. This number is multiplied by 2 because each flit requires 2 clock cycles to be sent.

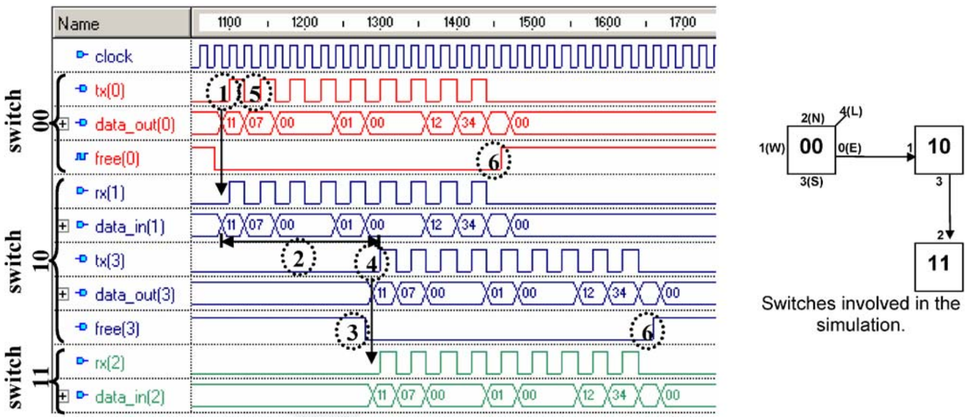


Fig. 7. Simulation of a packet transmission from switch 00 to switch 11 in topology of Fig. 6.

The latency to route the header and subsequent flits is due in part to the assumption that switch modules communicate always by explicit handshake signals, making the design highly modular and adaptable. Latency can be reduced using one of two alternatives. The first consists in combining the arbiter and the router into a single module, ignoring the modular design assumption. The second is by using alternative switch architectures, with distributed arbiters. See the proposal of Bartic et al. [26] for an example of such architecture. The first alternative leads to higher performance at the cost of the modularity and adaptability. The second alternative obtains performance from a significant increase in switch area.

6.2. *Switch peak performance*

The developed switch can establish only one connection at a time. However, a single switch can simultaneously handle up to five connections. The operating frequency was initially determined to be 25 MHz, for prototyping purposes. Each switch has five ports and each port transmits 8-bit flits. Since each flit takes two clock cycles to be sent, a switch presents a theoretical peak performance of 500 Mbits/s ((25 MHz/2)* 5 ports* 8 bits). This peak performance is indeed achieved in some moments as illustrated by the simulation results in Fig. 8, and explained below.

1. Address of the switch being simulated.
2. Target address of each incoming packet in the simulated switch, five headers arriving simultaneously.
3. Signal *incoming* indicates which port was selected to have its switching request granted, while the signal *header* indicates which is the target switch address of the selected packet.
4. First connection is established after 2.5 clock signals after the request: *flits* incoming from port 1 (West) exit at port 2 (North). To understand semantics of *mux_in* and *mux_out* signals, refer to Fig. 3(b).
5. Second connection is established after 8 clock signals after the previous one: *flits* incoming from port 2 (North) exit at port 4 (Local).

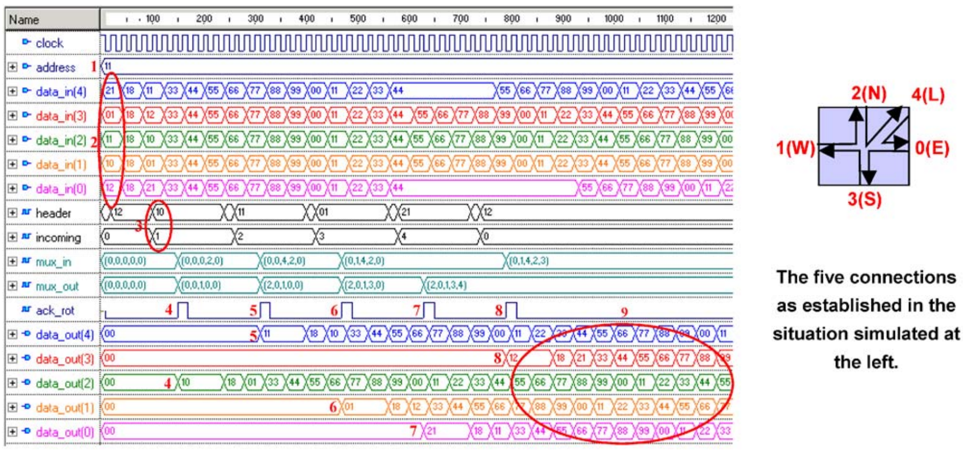


Fig. 8. Establishment of five simultaneously active connections in a single switch, to illustrate the peak performance situation.

6. Third connection is established: *flits* incoming from port 3 (South) exit at port 1 (West).
7. Fourth connection is established: *flits* incoming from port 4 (Local) exit at port 0 (East).
8. Fifth connection is established: *flits* incoming from port 0 (East) exit at port 3 (South).
9. After this sequence of events, the switch is working at peak performance, taking 2 clock cycles to switch 5 8-bit flits, i.e. 500 Mbits/s at a clock rate of 25 MHz.

7. Prototyping and results

The Hermes switch and NoC behavior has already been sketched in Sections 5 and 6. This Section is intended to present some quantitative data about these. Section 7.1 describes how to define a good compromise between latency and buffer size for 8-bit flits. Next, Section 7.2 presents data about the switch area consumption for different buffer and flit sizes. Finally, Section 7.3 provides results about FPGA prototyping.

7.1. Network latency and buffer sizing

A 5×5 mesh topology is employed to evaluate the network performance. The Hermes NoC is described in VHDL, while traffic generation and analysis is written in the C language. Co-simulation uses ModelSim and the FLI library [42], which allows VHDL and C to communicate.

7.1.1. Latency and buffer sizing without packet collision

The goal of the first conducted experiment is to define how to dimension the switch input buffers for the ideal situation where no packet collisions arise. As demonstrated later in this Section, this minimum buffer size is a good value, even for situations where collisions arise. The experiment was conducted as follows. A file containing 50 packets with 39 *flits* addressed to IPs located at different distances from the source IP is connected to the Local port of one switch, which serves as a traffic source. The tested distances between source and target varies from 1 to 5 hops. When a given *flit* enters the network, its *timestamp*[3] is stored, and when it arrives at the target switch, the total *flit* transmission time is stored in the *output file.* The plot of the simulation results is shown in Fig. 9.

The time spent to deliver packets grows linearly with the number of hops. For buffer sizes of six or more positions, the time remains almost constant, growing 10 clock cycles per hop. This happens because each switch spends some clock cycles to execute the arbitration and switching algorithms. If the buffer is too small, the switch cannot receive new *flits* until the destination port is chosen. Therefore, the buffer size to minimize latency has to be equal to the number of write operations that can be performed during the arbitration and switching algorithms execution. In the Hermes NoC, these algorithms consume 10 clock cycles and each write operation takes two clock cycles. Considering that the header *flit* must be in the buffer to be processed, the buffer size has to be at least six. With such buffer size, the *flits* are delivered as in an ideal pipeline. If the network works in this way, the formula below can be used to compute the total time to deliver

[3] *Timestamp* corresponds to the present simulation time.

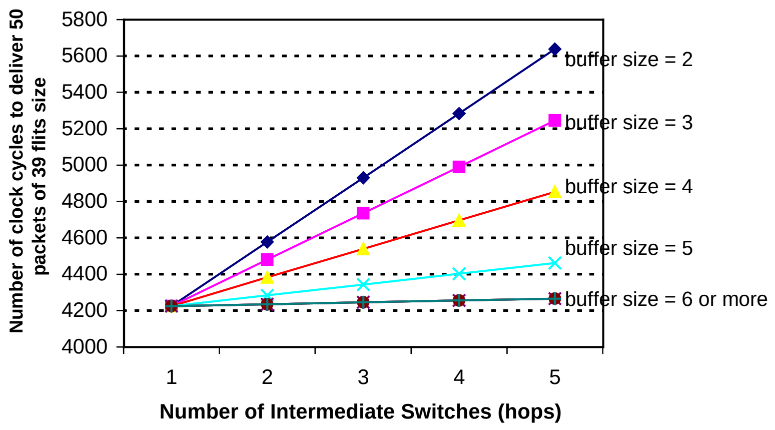


Fig. 9. Total time, in clock cycles, to deliver 50 packets of length 39 flits, for various 8-bit flit buffer sizes.

Table 2
NoC latency and throughput evaluation of 500 packets with random traffic for buffer sizes 8 and 16

	Buffer size = 8				Buffer size = 16			
	Traffic 1	Traffic 2	Traffic 3	Average	Traffic 1	Traffic 2	Traffic 3	Average
Average	260	275	271	**268**	312	324	326	**321**
Std. Deviation	170	199	167	**179**	203	208	201	**204**
Minimum	89	89	100	**93**	89	89	100	**93**
Maximum	1305	1618	1221	**1381**	1225	1644	1385	**1418**
Total time	5346	5559	5142	**5349**	4686	5088	4908	**4894**

Three sets of random data were used. Numbers in Table express clock cycles.

a set of packets:

$$Total\ time\ without\ packet\ collision = (ST + (NF - 1)^{*}2)^{*}NP,$$

where ST is the number of clock cycles to execute the arbitration and routing algorithms, 10 in the Hermes NoC, NF is the number of flits, 39 in this experiment; the –1 factor is used because the first flit (header) is processed in ST, $^{*}2$: each *flit* spends two clock cycles to be transmitted to the next switch, NP is the number of packets, 50 in this experiment.

Replacing the values in the above equation, the total time spent to deliver 50 packets with 39 flits is 4300 clock cycles, exactly the value observed in Fig. 9.

Buffers larger than the computed minimum size can be used to reduce contention, at the cost of some extra area. When dimensioning buffers during the NoC implementation, the designer has to consider the trade-off among area, latency, and throughput.

7.1.2. Latency and buffer sizing with random traffic and collision

The second experiment analyzes the NoC behavior in the presence of collisions, using random traffic. A random traffic generator and a process to store data concerning arriving *flits* were connected to each of the 25 switches. Two different buffer sizes were tested: 8 and 16 positions.

Table 2 presents the traffic results of simulating 500 packets passing through the network, where each switch sends 20 packets with 39 *flits* to random targets. Table 3 presents the traffic results of simulating 100,000 packets sent across the network. The two most relevant parameters are the *average* time to deliver a packet (first line), associated to the packet latency, and the *total time* to deliver all packets (last line), associated to the NoC throughput.

Tables 2 and 3 show that the average time to deliver a packet increased when doubling the buffer size (first line). This increased latency can be better understood analyzing Fig. 10. This Figure presents a header *flit* (number 1) arriving in two buffers with no empty space left. In the smaller buffer, the header has to wait that 7 *flits* be sent to the next switch before it can be treated, while in the bigger buffer the header waits a longer time.

The second line in Tables 2 and 3 presents the standard deviation of the average time to deliver the packets. To obtain the number of clock cycles to deliver 95% of the packets it suffices to add the average time to deliver a packet (first line) to the standard deviation. It is possible to observe that some packets stay in the network for a much longer time (fourth line—*maximum*). This may arise if a set of packets is transmitted to the same target or simply because of random collisions. Further analysis of these data is under way, in order to develop adequate traffic models and associated switching algorithms to reduce this problem.

The last line in Tables 2 and 3 presents the total time to deliver all packets. As in a pipeline, with additional buffer capacity the latency increases (as mentioned before) and the throughput is improved (8% in both experiments, 4894/5349 and 899,291/974,279). This improvement in throughput is due to the reduction in the network contention, since blocked flits use less network resources while waiting to be routed. The results indicate that buffers dimensioned with values

Table 3
NoC latency and throughput evaluation of 100,000 packets with random traffic for buffer sizes 8 and 16

	Buffer size = 8				Buffer size = 16			
	Traffic 1	Traffic 2	Traffic 3	Average	Traffic 1	Traffic 2	Traffic 3	Average
Average	281	280	281	281	348	347	348	348
Std. deviation	195	191	193	193	228	226	229	228
Minimum	99	99	89	96	100	100	89	96
Maximum	3073	2663	2601	2779	3350	3318	3025	3231
Total time	974,286	972,563	975,989	974,279	898,199	893,376	906,297	899,291

Three sets of random data were used. Numbers in Table express clock cycles.

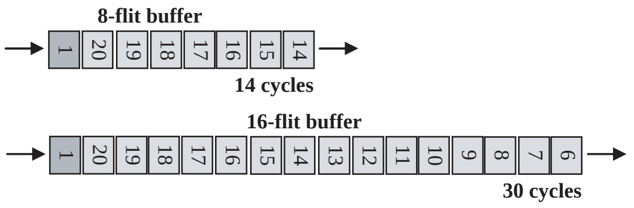


Fig. 10. Header *flit* latency for full buffers. In the first buffer, the header flit waits 14 clock cycles to be routed while in the second buffer it waits 30 clock cycles.

Table 4
Co-simulation time of a 5 × 5 Hermes NoC. Simulation time expressed for Modelsim running in a Sun Blade 2000 with 900 MHz clock frequency

Number of packets	500	1000	10,000	100,000
Co-simulation time (ms)	8978	16,898	162,002	1,614,682

near the minimum size for improving latency (6, in the case stated before) represent a good trade-off between latency and throughput while keeping area consumption small, as explained in Section 7.2.

It is also interesting to compare the performance of NoCs against shared bus architectures. Consider an ideal bus, able to send one word (the same width of the NoC flit) per clock cycle[4]. As the total number of words to be transmitted is respectively 19,500 and 3,900,000 (500 packets and 100,000 packets with 39 flits), it would be necessary 19,500 and 3,900,000 clock cycles to transmit all data. Data concerning a real NoC (Tables 2 and 3) show that it is necessary around 5300 and 975,000 clock cycles to transmit the same amount of data. In this situation, the NoC is almost 4 times faster than the ideal bus architecture. If real bus architectures are considered, NoCs are expected to present at least one order of magnitude of gain in performance over busses.

The results in Tables 2 and 3 were obtained with a pure XY switch algorithm. A fully adaptive XY algorithm was also employed, but then deadlock situations were observed. Deadlock-free adaptive switching algorithms are currently under implementation to overcome limitations of the pure XY algorithm.

Table 4 presents the average co-simulation time for the experiments showed in Tables 2 and 3. The co-simulation time grows linearly as a function of the number of transmitted packets.

The *load* offered by a given simulated traffic is defined as the percentage of the channel bandwidth used by each communication initiator [18]. The simulated traffic in the experiments reported here corresponds to a nominal load of 100%, since all cores are continually sending data to the NoC, without interruption between successive packets. In real situations, the system load is much smaller. This can be compared to data reported in [18], where the PI-bus architecture is shown to work well with load values below 4% and the SPIN NoC with load values below 28%. The Hermes NoC, due to its mesh topology, does support heavier traffic loads. The presented co-simulation time data correspond in fact to an upper bound simulation time, since, as mentioned before, in real benchmarks a much smaller load will be observed.

7.2. Switch area growth rate

The switch area consumption was estimated by varying two parameters: flit width and buffer size. The Leonardo Spectrum tool was used to synthesize the Hermes switch in two different technologies: Xilinx Virtex-II FPGAs and 0.35 μm CMOS ASIC. Synthesis was conducted with maximum effort, maximum area compaction, and hierarchy preservation (to allow collecting data about the individual modules composing the system).

[4] In practice, this is not feasible because of the latency associated to arbitration and bus protocols.

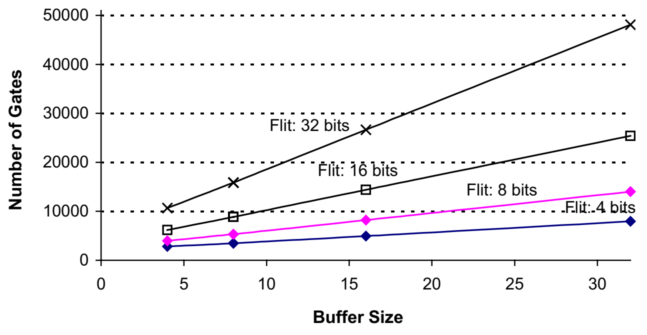


Fig. 11. Illustrating ASIC mapping growth rate for different switch size configurations.

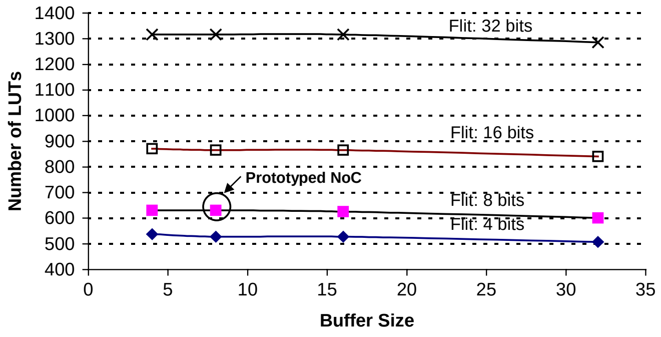


Fig. 12. Illustrating FPGA mapping growth rate for different switch size configurations.

Fig. 11 presents the area growth rate of ASIC mappings for different configurations of the switch, in equivalent gates. It is clear from the plotted data that the increase of the buffer size leads to a linear increase of the switch area for any flit size. In addition, the analysis of the raw data shows that the converse is also true, i.e. the increase of the flit size leads to a linear increase of the switch area for any buffer size. Another important result is that the buffer area dominates the switch area. For the smallest synthesized configuration, 4-flit buffers and 4-bit flit size, the switch logic consumes around 58% of the ASIC mapping area, and around 42% refers to buffer area. When the switch is configured with an 8-flit buffer and an 8-bit flit size, the buffer area takes 69% of the switch area. If the buffer and flit size increase to 32, buffers occupy 96% of the switch area.

In fact, the linear area growth shown in Fig. 11 is misleading, since this behavior appears only for buffer size steps in powers of 2. For example, the area growth rate is practically zero for buffers with dimension between 9 and 16 positions, for any flit size. This happens because the synthesis tool can only deal with memories which sizes are a natural power of two.

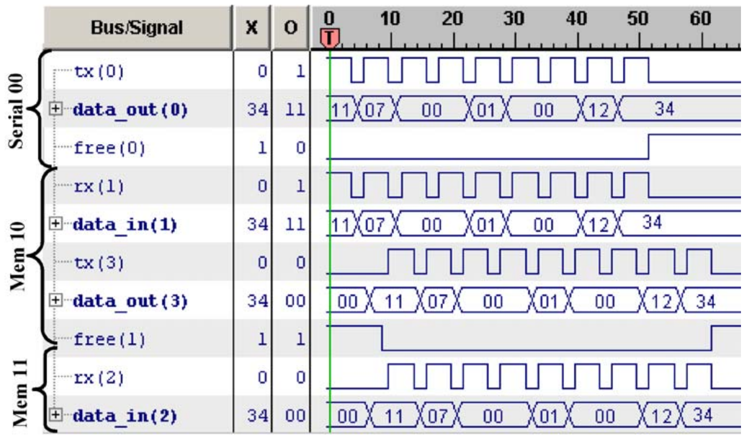
It would be expectable that the FPGA mapping behaves similar to the ASIC mapping. However, Fig. 12 presents a rather distinct behavior. The plot shows that independently of the buffer size, the LUT count, used as FPGA area unit, is practically invariant up to 32 bits. The fluctuations are due to the non-deterministic synthesis process. To really understand the area invariance it is necessary to delve into the FPGA device architecture and on how synthesis tools map hardware into this architecture. In this specific case, generic VHDL code was input to the

Leonardo tool, and the tool was instructed to perform LUT RAM inference. In Virtex families, each LUT can behave either as a 4-input truth table or as a small 16-bit RAM, named LUT RAM. When it is configured to be an LUT RAM, the component presents a 4-bit address input, to access up to 16 1-bit memory positions. Therefore, just one bit can be read from a LUT RAM at a time. For instance, if one 8-bit word must be read from a set of LUT RAMs, it is necessary to put eight LUT RAMs in parallel. Unfortunately, in this case, just one bit out of the 16 available per LUT will be used. On the other hand, if a 16-word buffer is used, only the same eight LUTs are needed. In the prototyping case study, the Leonardo tool inferred the switch buffers using Dual Port LUT RAMs. Dual Port LUT RAMs is a component that groups two LUTs. This is why the graphic is basically constant for buffer sizes until exactly 32 positions.

7.3. Prototyping

The Hermes NoC was prototyped using the Memec Insight Virtex-II MB1000 development kit. This kit is composed by three boards, the main one containing a million-gate Xilinx XC2V1000 456-pin FPGA device, memory and peripheral/communication devices [43]. A 2×2 NoC was implemented. To validate the prototyped NoC, two IP cores were developed: an RS-232 serial core and an embedded processor, named R8. The RS-232 *serial core* is responsible to send and receive packets to and from the network, providing an interface with a host computer. The R8 *processor* is a 40-instruction, 16-bit non-pipelined, load store architecture, with a 16×16 bit register file [44,45]. This processor was added to the NoC to validate the interconnection network as a multiprocessor platform. Each processor IP uses two internal 18 Kbits RAM blocks as instruction cache memory. The serial core was attached to switch 00 and the processor cores were attached to the other three switches.

Two software programs were used for hardware validation. The first one, developed in the scope of this work, provides communication between the development kit and the host computer. The second software is Xilinx ChipScope, which allows visualizing FPGA internal signals at run time [43]. Fig. 13 is a ChipScope snapshot showing the same signals presented in Fig. 7 functional



Bus/Signal	X	O
tx(0)	0	1
data_out(0)	34	11
free(0)	1	0
rx(1)	0	1
data_in(1)	34	11
tx(3)	0	0
data_out(3)	34	00
free(1)	1	1
rx(2)	0	0
data_in(2)	34	00

Fig. 13. ChipScope software snapshot, with data obtained directly from the prototyping board.

Table 5
2 × 2 Hermes NoC area data for XC2V1000 FPGA. LUTs are 4-input look-up-tables, a slice has 2 LUTs and 2 flip-flops and BRAMs are 18-Kbit RAM blocks

Resources	Used	Available	Used/total (%)
Slices	3058	5120	59.73
LUTs	6115	10,240	59.72
Flip flops	2968	11,212	26.47
BRAM	6	40	15.00

Table 6
2 × 2 Hermes NoC modules area report for FPGA and ASIC (0.35 μm CMOS). LUTs represent combinational logic. ASIC mapping represents the number of equivalent gates

	Virtex II mapping			ASIC mapping
	LUTs	FFs	BRAM	
Switch	555	172	—	3838
SR	210	233	—	2495
Serial	92	93	—	859
Serial + SR	608	563	—	5571
R8	538	114	—	2156
RAM + SR + R8	1111	576	2	6826

simulation. This picture shows that Hermes NoC works in FPGAs exactly as predicted by simulation, including the performance figures presented in Section 6.2.

The NoC with 4 IP cores (1 serial and 3 processor cores) and four switches was synthesized using the Leonardo synthesis tool. Table 5 presents area estimates generated by synthesis, where it can be seen that approximately 50% of the FPGA resources were employed.

Table 6 details the area usage of the NoC modules for two mappings, FPGA and ASIC. The switch itself takes 555 LUTs to be implemented, which represents around 5.4% (555/10240) of the available LUTs in the million-gate device, or around 9% (555/6115) of overhead in the implemented NoC. The Table also gives area data for three modules: serial and R8 processor. These modules were used to build an NoC-based on-chip multiprocessing system. SR is a send/receive wrapper module containing the interface between a switch and each IP core. Additional glue logic is needed to connect the IP core to SR, adding to the total gate count of the wrapped module.

This multiprocessor NoC platform is presently used to execute parallel programs, such as sorting algorithms [46].

8. Conclusions and future work

Networks on chip are a recent technology where much research and development work is left undone. From Section 3, it is possible to infer that scarce implementation data have been reported in the available literature. The Section 3 review is preliminary and shows mostly raw data found in the literature. It could be improved by reducing these data to a common ground, enabling an easier comparison of the different NoC proposals. Also, data about tools supporting NoC design and validation are already available, but were not addressed in this paper.

To the knowledge of the authors, the commercial offer of SoCs based on NoCs is not yet a reality. However, the potential advantages and current results of using NoCs lead already to the conclusion that they are a competitive technology. Among the problems for which NoCs appear as providing solutions, it is important to stress at least two: the enabling of SoC asynchronous communication between synchronous regions and SoC size scalability.

The body of knowledge about interconnection networks already available from the computer networks, distributed systems, and telecommunication subject areas is a virtually infinite source of results waiting to be mapped for the NoC domain. This mapping is anything but simple, since the constraints imposed by silicon to the implementation of network infrastructures are significant.

The Hermes infrastructure, switch, and NoC fulfilled the requirement of implementing a low area overhead and low latency communication for on-chip modules. The most relevant point of this work is the availability of a hardware testbed where NoC architectures, topologies, and algorithms can evolve, be implemented, and evaluated. A first application of the Hermes infrastructure is in the construction of a wireless multimedia application prototyping platform, named Fenix (www.brazilip.org/fenix). All design, implementation, and results data reported here are publicly available [46]. As required by the specification, the switch area is small. It is possible to note that the area of the IP cores is strongly influenced by the SR wrapper. The SR wrapper is still a preliminary structure, with buffers large enough to guarantee correct functionality of the communication. Better dimensioning of the SR and wrapping structures is an ongoing work.

It is already possible to compare area results obtained for the Hermes switch with some approaches found in the literature. First, Marescaux employed exactly the same prototyping technology and proposed switches that occupy 450 [24] and 611 [25] Virtex-II FPGA slices. Hermes switch employs 278 slices (555 LUTs), but it does not implement virtual channels. Second, the aSOC approach [19] mentions a switch ASIC implementation with an estimated transistor count of 50,000. The Hermes switch with the smallest possible buffer size (since aSOC does not use buffers) and a 32-bit flit size (the same as aSOC) has an estimated gate count of 10,000, which translates to 40,000 transistors.

The Hermes infrastructure provides in its current state support to the implementation of *best effort* (BE) NoCs only [10,11]. In BE, sent packets can be arbitrarily delayed by the network, as evidenced in Table 2 for the Hermes NoC. For applications with hard real time constraints, it is necessary to provide *guaranteed throughput* (GT) services. Another ongoing work is to provide the Hermes infrastructure with the possibility of addressing the implementation of GT NoCs. Maybe the most important kind of traffic to support in current SoCs is that arising from streaming applications, such as real-time video and audio. Further studies on the adequacy of the Hermes infrastructure for transporting streaming applications data are under way. Also, the Hermes IP to switch interface employs the OCP standard interface, providing enhanced reusability of the infrastructure and connectivity to available OCP compliant IP cores.

References

[1] International Sematech. International Technology Roadmap for Semiconductors—2002 Update, 2002. Available at http://public.itrs.net.

[2] R. Gupta, Y. Zorian, Introducing core-based system design, IEEE Des. Test Comput. 14 (4) (1997) 15–25.

[3] R. Bergamaschi, et al., Automating the design of SOCs using cores, IEEE Des. Test Comput. 18 (5) (2001) 32–45.

[4] G. Martin, H. Chang, System on chip design, in: The Nineth International Symposium on Integrated Circuits, Devices and Systems (ISIC'01), Tutorial 2, 2001.

[5] S. Kumar, et al., A network on chip architecture and design methodology, in: IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02), April 2002, pp. 105–112.

[6] M. Millberg, E. Nilsson, R. Thid, S. Kumar, A. Jantsch, The Nostrum backbone—a communication protocol stack for networks on chip, in: Proceedings of the VLSI Design Conference, January 2004.

[7] L. Benini, G. De Micheli, Powering networks on chips: energy-efficient and reliable interconnect design for SoCs, in: 14th International Symposium on Systems Synthesis (ISSS'01), October 2001, pp. 33–38.

[8] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, IEEE Comput. 35 (1) (2002) 70–78.

[9] P. Guerrier, A. Greiner, Ageneric architecture for on-chip packet-switched interconections, in: Design Automation and Test in Europe (DATE'00), March 2000, pp. 250–256.

[10] E. Rijpkema, K. Goossens, A. Rădulescu, Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, in: Design, Automation and Test in Europe (DATE'03), March 2003, pp. 350–355.

[11] E. Rijpkema, K. Goossens, P. Wielage, Arouter architecture for networks on silicon, in: 2nd Workshop on Embedded Systems (PROGRESS'2001), November 2001, pp. 181–188.

[12] J. Duato, S. Yalamanchili, L. Ni, Interconnection Networks: An Engineering Approach, Morgan Kaufmann, Los Altos, CA, 2002, 624p, revised edition.

[13] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, San Francisco, CA, 1996, 760p.

[14] K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill, New York, 1992, 672p.

[15] L. Ni, et al., A survey of wormhole routing techniques in direct networks, IEEE Comput. 26 (2) (1993) 62–76.

[16] T. Ye, L. Benini, G. De Micheli, Packetized on-chip interconnection communication analysis for MPSoC, in: Design Automation and Test in Europe (DATE'03), March 2003, pp. 344–349.

[17] A. Andriahantenaina, A. Greiner, Micro-network for SoC: implementation of a 32-port SPIN network, in: Design Automation and Test in Europe Conference and Exhibition (DATE'03), March 2003, pp. 1128–1129.

[18] A. Andriahantenaina, H. Charlery, A. Greiner, L. Mortiez, C. Zeferino, SPIN: a scalable, packet switched, on-chip micro-network, in: Design Automation and Test in Europe Conference and Exhibition (DATE'03), March 2003, pp. 70–73.

[19] J. Liang, S. Swaminathan, R. Tessier, aSOC: A scalable, single-chip communications architecture, in: IEEE International Conference on Parallel Architectures and Compilation Techniques, October 2000, pp. 37–46.

[20] W. Dally, B. Towles, Route packets, not wires: on-chip interconnection networks, in: 38th Design Automation Conference (DAC'01), June 2001, pp. 684–689.

[21] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, Addressing the system-on-chip interconnect woes through communication-based design, in: 38th Design Automation Conference (DAC'01), June 2001, pp. 667–672.

[22] F. Karim, A. Nguyen, S. Dey, An interconnect architecture for network systems on chips, IEEE Micro 22 (5) (2002) 36–45.

[23] F. Karim, A. Nguyen, S. Dey, R. Rao, On-chip communication architecture for OC-768 network processors, in: 38th Design Automation Conference (DAC'01), June 2001, pp. 678–683.

[24] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins, Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs, in: Field-Programmable Logic and Applications (FPL'02), September 2002, pp. 795–805.

[25] T. Marescaux, J.-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, R. Lauwereins, Networks on chip as hardware components of an OS for reconfigurable systems, in: Field-Programmable Logic and Applications (FPL'03), September 2003.
[26] A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, J.-Y. Mignolet, D. Verkest, S. Vernalde, R. Lauwereins, Highly scalable network on chip for reconfigurable systems, in: International Symposium on System-on-Chip (SOC'2003), November 2003.
[27] M. Forsell, A scalable high-performance computing solution for networks on chips, IEEE Micro 22 (5) (2002) 46–55.
[28] D. Sigüenza-Tortosa, J. Nurmi, Proteo: a new approach to network-on-chip, in: IASTED International Conference on Communication Systems and Networks (CSN'02), September 2002.
[29] I. Saastamoinen, M. Alho, J. Pirttimäki, J. Nurmi, Proteo interconnect IPs for networks-on-chip, in: IP Based SoC Design, October 2002.
[30] I. Saastamoinen, M. Alho, J. Nurmi, Buffer implementation for proteo networks-on-chip, in: International Symposium on Circuits and Systems (ISCAS'03), May 2003, pp. II-113–II-116.
[31] C. Zeferino, A. Susin, SoCIN: a parametric and scalable network-on-chip, in: 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03), September 2003, pp. 169–174.
[32] D. Wiklund, D. Liu, SoCBUS: switched network on chip for hard real time systems, in: International Parallel and Distributed Processing Symposium (IPDPS), April 2003.
[33] E. Bolotin, I. Cidon, R. Ginosar, A. Kolodny, QNoC: QoS architecture and design process for network on chip, The Journal of Systems Architecture, Special Issue on Networks on Chip 50 (2) (2004) 105–128.
[34] P. Pande, C. Grecu, A. Ivanov, R. Saleh, Design of a switch for network on chip applications, in: International Symposium on Circuits and Systems (ISCAS'03), May 2003, pp. 217–220.
[35] C. Grecu, P. Pande, A. Ivanov, R. Saleh, A scalable communication-centric SoC interconnect architecture, in: IEEE International Symposium on Quality Electronic Design (ISQED'2004), 2004.
[36] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, L. Benini, Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs, in: International Conference on Computer Design (ICCD'03), 2003, pp. 536–539.
[37] F. Moraes, A. Mello, L. Möller, L. Ost, N. Calazans, A low area overhead packet-switched network on chip: architecture and prototyping, in: IFIP Very Large Scale Integration (VLSI-SOC), 2003, pp. 318–323.
[38] C. Glass, L. Ni, The turn model for adaptive routing, J. Assoc. Comput. Mach. 41 (5) (1994) 874–902.
[39] J. Day, H. Zimmermman, The OSI reference model, Proc. IEEE 71 (12) (1983) 1334–1340.
[40] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson P. Nilsson, J. Öberg, P. Ellervee, D. Lundqvist, Lowering power consumption in clock by using globally asynchronous, locally synchronous design style, in: 36th Design Automation Conference (DAC'99), 1999, pp. 873–878.
[41] P. Mohapatra, Wormhole routing techniques for directly connected multicomputer systems, ACM Comput. Surv. 30 (3) (1998) 374–410.
[42] Model Technology. ModelSim Foreign Language Interface, Version 5.5e, 2001.
[43] Xilinx Inc. Virtex-II Platform FPGA User Guide. July 2002, available at: http://www.xilinx.com.
[44] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, E. Carara, From VHDL register transfer level to systemC transaction level modeling: a comparative case study, in: 16th Symposium on Integrated Circuits and Systems Design, (SBCCI'03), 2003, pp. 355–360.
[45] N. Calazans, F. Moraes, Integrating the teaching of computer organization and architecture with digital hardware design early in undergraduate courses, IEEE Trans. Educat. 44 (2) (2001) 109–119.
[46] A. Mello, Möller, L. SoC multiprocessing architectures: a study of different interconnection topologies. End of Term Work, FACIN–PUCRS, July 2003, 120p. (in Portuguese), available at http://www.inf.pucrs.br/~moraes/papers/tc_multiproc.pdf.