BaBaNoC: An Asynchronous Network-on-Chip Described in Balsa

Matheus T. Moreira, Felipe G. Magalhães, Matheus Gibiluka, Fabiano P. Hessel, Ney L. V. Calazans
Faculty of Computer Science - Pontifical Catholic University of Rio Grande do Sul – PUCRS - Porto Alegre, Brazil
{matheus.moreira, felipe.magalhaes, matheus.gibiluka}@acad.pucrs.br, {fabiano.hessel, ney.calazans}@pucrs.br

***Abstract*—The downscaling of silicon technology and the possibility of building MPSoCs, make intrachip communication a mainstream research topic. NoCs are an elegant solution to provide communication scalability and modularity. NoCs are already common in MPSoC design. Moreover, new technology challenges point to a growth in the use of non-synchronous NoCs. However, the design of asynchronous infrastructures with current EDA tools is challenging. That is due to the fact that most of these tools are oriented towards synchronous design. This work proposes and evaluates a fully asynchronous NoC router based on the Balsa language and framework. The design is validates through FPGA synthesis.**

Keywords-component; Asynchronous circuits, network-on-chip, semi-custom

I. INTRODUCTION

Current technologies allow the implementation of multi-processor systems-on-chip (MPSoCs), large amount of intellectual property cores (IP cores) interconnected through some communication architecture. The limitations of classic bus infrastructures led to the development of networks-on-chip (NoCs). These present higher levels of scalability and communication parallelism than buses. In fact, the use of NoCs is already a well established concept for creating effective intrachip communication architectures. Modern MPSoCs rely heavily on IP reuse, where IPs may employ particular standards and/or protocols and present varying design constraints. Often, IPs' requirements determine the use of specific communication protocols and/or operating frequencies. This renders the design of MPSoCs with multiple frequency domains far more natural than fully synchronous ones [1].

Systems with multiple frequency domains that communicate with each other through some synchronization mechanism are classified as globally-asynchronous locally-synchronous (GALS) [2]. In such systems, synchronization interfaces must be used at specific points, to allow distinct clock domains to communicate. Clearly, NoCs are natural candidates to include such synchronization interfaces, and this is an active research area [3]. These NoCs can themselves be fully synchronous, GALS or fully asynchronous, depending on their router design and on the router-to-router and router-to-IP interfaces design. The present work addresses the class of fully asynchronous NoCs.

The drawback is that the automation provided for the design of asynchronous circuits is still in early stages of development, because most commercial tools focus exclusively on the synchronous design paradigm. Consequently, all classes of asynchronous circuits have limited electronic design automation (EDA) support. Also, designing asynchronous circuits requires specific components, which are typically absent from standard cell libraries. In short, the design of non-synchronous circuits is often classified as a difficult task.

This work describes the design of an asynchronous router, called Balsa Based Router (abbreviated to BaBaRouter) to support the design of GALS SoCs. The Balsa language [4] allowed the straightforward description of this fully asynchronous NoC router. The circuit is an asynchronous version of the well-known Hermes NoC router [5]. Itsdesign was validating through its synthesis for different Xilinx FPGAs, supported by the Balsa Framework.

II. CONCEPTS

A. Asynchronous Circuits

Asynchronous circuits can be classified according to several criteria. One of the most employed criterion is based on the delays of wires and gates [6]. Accordingly, the most robust and restrictive delay model is the delay insensitive (DI) model [7] [8], which operates correctly regardless of gate and wire delay values. Unfortunately, this class is too restrictive. The addition of an assumption on wire delays in some carefully selected forks enables to define the quasi-delay-insensitive (QDI) circuit class. Here, signal transitions occur at the same time only at each end point of the mentioned forks, which are called isochronic forks. In fact, many works report QDI as the most suited class for practical asynchronous circuits, such as [9].

In order to implement QDI circuits, DI codes [7] [8] are required. These guarantee robustness to wire delay variations, because the request signal is embedded within the data signal. An example is the class of *m-of-n codes*. Given **n** and **m**, with **m<n**, an m-of-n code consists in the set of all n-bit code words with Hamming weight (i.e. the number of bits in 1 in the code word) equal to **m**. For example, the well-known one-hot code is an example of 1-of-n code. The use of m-of-n codes coupled to a protocol that establishes how valid codes succeed one another in a data flow allows obtaining communication with absolute insensitivity to delay variations in individual wires. Handshake protocols can be either *2-phase* or *4-phase [6]* both illustrated in Figure 1for a 1-of-2 code. Usually, 2-

978-1-4799-2409-7/13$31.00

phase protocols operate faster, but require more hardware than 4-phase protocols.

One of the most used approaches to achieve delay insensitivity consists in representing each data bit in a circuit by a pair of wires, in what is called dual-rail (DR) encoding (where each bit is represented using a 1-of-2 code). Let **d.t** (data true or 1) and **d.f** (data false or 0) be the names of two wires representing a single data bit. One example of 2-phase handshake using a 1-bit DR code appears in Figure 1(a). Here, a transition in wire **d.f** signals a **logical 0** value, which is recognized by a transition in the signal acknowledge (**ack**). A transition in **d.t** signals a **logical 1** value, which is again acknowledged by a transition in **ack**.

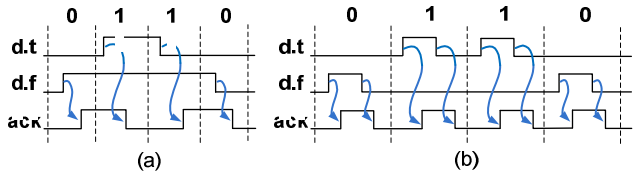

Figure 1 –Handshake protocols types: (a) 2-phase (b) 4-phase.

Figure 1(b) shows an example 4-phase protocol using DR code. Logical levels in wires uniquely identify data bit values. Again, let **d.t** and **d.f** be the names of two wires representing one bit of some DR code. Valid bit values are always valid code words of the 1-of-2 code ("**01**" for bit 0 and "**10**" for 1). However, after a value is acknowledged, all wires must return to a predefined value, here the **all-0s** spacer. The spacer itself is an invalid DR code word. This protocol is accordingly denominated *return to zero* or RTZ. In Figure 1(b), the first communicated data value is a logical 0, encoded by **d.t=0** and **d.f=1**. After the value is acknowledged by a low-to-high transition in the **ack** signal, a spacer is issued, in this case **d.t=0** and **d.f=0**. Next, the **ack** signal switches to 0, signaling reception of the spacer, and a new transmission may occur. Any 4-phase protocol requires spacers when using m-of-n codes.

B. *Balsa*

With the growing interest for asynchronous circuits, different research tools have been proposed to automate the process of designing them. Among these, Balsa stands as a comprehensive, open source environment [6]. The tool was designed and is maintained by the Advanced Processor Technologies Group from the University of Manchester [4]. In fact, Balsa is both a language to describe asynchronous circuits and a framework to simulate and synthesize them. The compilation of a Balsa description is transparent, since language constructs are directly mapped into handshake components. In this way, it is relatively easy for the designer to visualize the circuit-level architecture of a Balsa description. Moreover, modifications in a Balsa description reflect in predictable changes in the resulting circuit, which means that the designer has a clear control of the generated hardware.

Balsa requires the designer to furnishthe order of handshake events through Balsa language operators. These events imply the communication between handshake components (data exchange or control only) and can be specified as occurring sequentially or concurrently. Moreover, handshake components are transparent and are derived from higher abstraction language constructs, such as *if/else*, *case*, *select* and *arbitrate*. The two latter are very important for asynchronous circuits; they allow the designer to control and arbitrate events without a discrete notion of time. Therefore, one of the main advantages of using Balsa to describe and implement asynchronous circuits is that specific communication control signals between handshake components are abstracted.

After translating Balsa descriptions into a network of handshake components, different asynchronous templates can be assumed to map this to a target technology. For standard-cell implementations, a library with specific components is required, for FPGAs implementations, only the basic libraries are required. After the mapping, the netlists can be imported into back-end commercial tools for physical implementation and optimization, power and timing analysis.

III. RELATED WORK

As far as the authors could verify, six other fully asynchronous NoC routers are described in the literature: Asynchronous MANGO [10], ASPIN [11], QNoC [12], A-NoC [13], and Hermes-A/Hermes-AA [14] [15]. The latter derived from the work of the research group of the authors.

MANGO, asynchronous QNoC and A-NoC claim support to quality of service through the use of virtual channels and/or special circuits. A-NoC is the most developed of the proposals and presents the best overall performance. It has been successfully used to build at least two complete integrated circuits. Since Hermes-A and Hermes-AA are very similar, we restrict attention to the latter. Hermes-AA presents adaptive routing schemes to deal with unpredictable application dynamic behaviors, allowing packets to take different paths through the network every time congestion is detected.

The development of HermesA and Hermes-AA demonstrated that adapting typical tools and standard cell libraries to design asynchronous circuits is challenging. Hermes-AA functional description took months to be implemented and validated. Its logic synthesis used lots of manual labor to generate a functional netlist. The difficulties faced during the development of Hermes-AA were a motivation for the research this paper describes. Accordingly, the BaBaRouter first functional version was described and validated in the Balsa Framework during one week. Balsa automatically produced functional netlists for physical synthesis input.

All asynchronous NoCs reviewed in the literature report the use of synchronous design flows and tools for implementing asynchronous routers. This generates more workload, where the focus of the work becomes the adaptation of the circuit, to guarantee the asynchronous circuit works. Besides, asynchronous modules are handcrafted. This means that manual design is omnipresent to implement handshake elements, asynchronous control

logic and even basic components. Also, the BaBaRouter design employs a fully QDI asynchronous template. The only reviewed works that pledge the same approach are A-NoC and Hermes-A/Hermes-AA. The use of a QDI template, even if it increases design complexity somehow, is justified by the fact that it enables overall delay insensitivity, leading to designs that are robust to process, voltage and temperature variations.

In short, the main two differences between the work presented herein and the others discussed above are: (*i*) the use of a high level language and framework for design entry and synthesis and (*ii*) an automated connection between the high level framework and the physical synthesis framework. These enable facilitated design space exploration for asynchronous circuits, a feature absent in previous works.

IV. THE BABAROUTER ARCHITECTURE

The BaBaRouter was described using the Balsa language. The router can be implemented for different asynchronous templates, due to the fact that a Balsa descriptionabstracts data encoding methods, as well as communication protocols. Thus, specific design template decisions are made during synthesis and mapping to a specific technology, not during design capture. Moreover, as the Hermes router, BaBaRouter is parameterizable. In fact, the latter has exactly the same functionality and block structure as the Hermes router. Figure 2 displays the block diagram of the BaBaRouter.

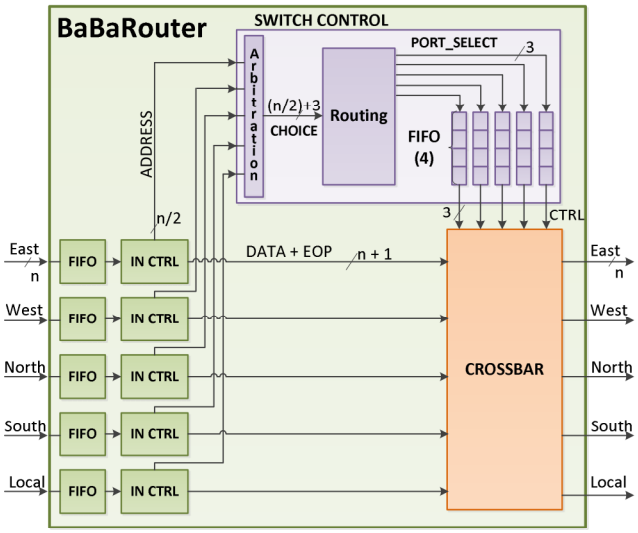

Figure 2 – BaBaRouter implementation block diagram. Communication and synchronization takes place through handshake channels. Control signals are implicit in the figure. The flit size is denoted by n, which is used to specify the adopted internal channel widths.

Four fundamental block types form the BaBaRouter: (*i*) the input FIFO buffers and (*ii*) the input control (IN CTRL), which together are equivalent to a Hermes input buffer, (*iii*) the switch control and (*iv*) the crossbar. All blocks are built with handshake components and each block is itself a handshake component. Consequently, blocks use handshake channels to communicate and synchronize. All channels in Figure 2 abstract existing request and acknowledge control signals. Also, at most five input and five output ports compose the router: East (0), West (1), North (2), South (3) and Local (4), each with input and output interfaces. Again, ports 0-3 interconnect routers and port 4 interconnect router and IP.

A. *The Input FIFO and IN CTRL Modules*

Sequentially connected registers form the input FIFO. Width and depth of the FIFOs are parameterizable by the designer. Neighbor registers handshake to each other as data flows through the FIFO. The width of router data channels is the same as the router flit width. Without loss of generality, this work assumes the use of 8-bit flits and 8-flit deep FIFOs.

The IN CTRL block treats packet control information. A packet in BaBaRouter has a variable size and follows the Hermes format, as Figure 3 shows.

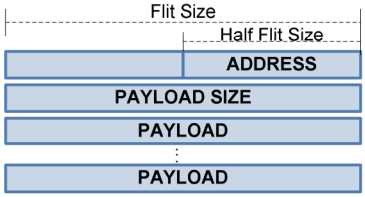

Figure 3 – BaBaRouter packet structure.

The first flit has the packet destination address in its lower half, followed by the packet payload size in the second flit, trailed by the payload in itself, i.e. all following flits. When the IN CTRL of a given port detects a new communication, it sends the address to the switch control through a (n/2)-bit channel, where n is the flit width. Then, it sends the whole address flit, together with an inactive end of package (EOP) signal (value '0'), to the crossbar, through an (n+1)-bit channel. Upon receiving the next flit (the payload size) IN CTRL sets an internal register to the value contained in this flit, resets its internal counter and propagates the flit to the crossbar, again with the '0' EOP value. Next, IN CTRL increments its counter for each new flit received and propagates the flit to the crossbar with EOP='0'. When it detects the last flit, by comparing its counter to the internal register that keeps the packet size, it signals EOP='1' when sending the last flit of the packet. Note that empty payloads are allowed.

B. *The Switch Control Module*

The switch control is responsible for computing a router output port for a packet, using the address received from IN CTRL and the internal router address. All input ports share the switch control module to route packets. Therefore, requests from these ports to the switch must be arbitrated. For instance, if two ports receive a new package at the same time and the delays of each path to the switch control are equivalent, two requests to use the switch control block arrive at the same time. Choosing what packet to route first, the switch control arbitrates requests through the Arbitration block. This can be achieved at high level using the *arbitrate* Balsa construct.

The arbitration choice comprises the selected destination address (n/2 bits) and the port identifier that requested it (3

bits). This output is input to the Routing block, which computes the path the packet will follow. Currently, packets are routed using the XY algorithm. Other routing algorithms can be easily adapted, due to the modularity inherent to asynchronous circuits and in particular to Balsa descriptions and the BaBaRouter architecture. The Routing module produces a 3-bit code for choosing one of the five output ports. The switch control has five 3-bit, 4-position output FIFOs, one for each router output port. The FIFO queue requests to a specific output port. For instance, consider that a router with address "11" receives a new packet in the East port with destination "11". The switch control will compute that the packet must follow to the Local output port. Then, it will write in the local output FIFO the value of the East router input (0). In other words, it will inform that the Local output port needs to be reserved for the East input port. Each new request to the Local output port will be queued in the output FIFO. The FIFO will be full when all input ports, but the Local, request the local output port. Note that it is impossible that any routing request is pending to enter a full FIFO, because at any moment at most four input ports may request a same output port. This justifies the fixed size of the FIFOs and guarantees that the switch control alone will never cause communication to stall.

FIFOs in the switch control outputs are useful to avoid starvation while ensuring fair service to all ports, when multiple inputs request the same output port. A same input port is not allowed to request the same output port twice consecutively. This approach is indeed fairer than the Round Robin arbiter of Hermes. Besides this approach costs little hardware. Implementing the original Hermes round robin in an asynchronous style would be a more area-consuming and complex task.

C. *The Crossbar*

Figure 4shows the simplified crossbar block diagram. The crossbar binds an output port to an input port. This is done with the information generated by the switch control module. When the switch control routes a packet to an output port, it signals to the crossbar through the CTRL channels (see Figure 2) which input port must be bound to a given output port. The crossbar then binds the ports and propagates the packets received from the IN CTRL until an EOP='1' is received. Remember all modules in Figure 4are handshake components. The DEMUX control triggers a handshake between this component and the MERGE module that selects one of its input flows to send to its output port data lines. After EOP='1' is received that port can be bound to a new input port. Only when the whole packet is transferred, the crossbar finishes the communication with the switch control for a given output port, generating an acknowledge signal. Thus, a new communication for any of the remaining output ports can take place at any time.

BaBaRouter can have data flowing from different inputs to different output ports concurrently. The router maximum throughput is reached when all input ports are granted to communicate with different output ports. In this case, five paths are simultaneously established across the router.

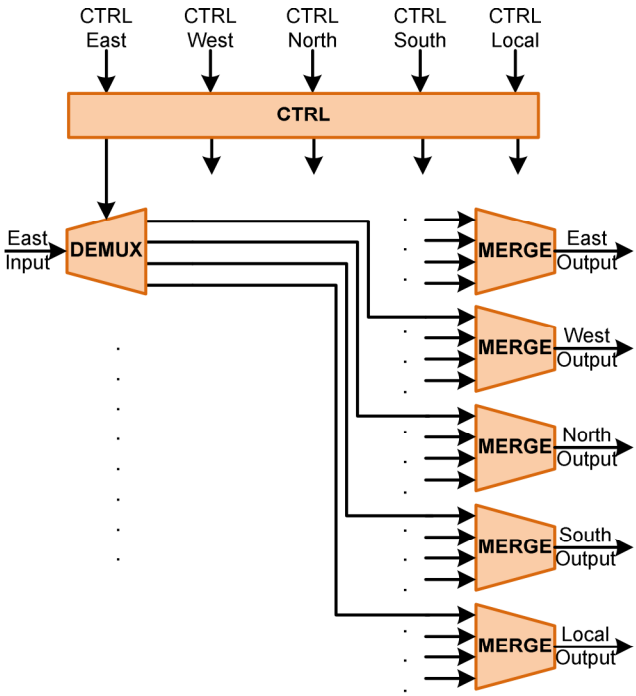

Figure 4 – BaBaRouter Crossbar simplified block diagram.

The crossbar consumes a large amount of hardware, becausefor each input port four channels must exist, one for each possible output port. The choice of what channel to use can be viewed as a demultiplexer (DEMUX) where the control input derives from information coming from the crossbar CTRL block. This block receives data from the switch control along with EOP signals (not shown in Figure 4) and binds each input port to an output port, producing control signals to the demultiplexers in each input. Channels destinedto a same output port are merged to the actual output port. This was the best approach that the authors could find for a Balsa description of a crossbar, to guarantee concurrency of communication for different input/output port pairs.

D. *Dataflow in BaBaRouter*

As Figure 5 shows, when the first flit of a packet is received in an input of the router, it is propagated through the input FIFO and reaches the IN CTRL, which feeds the switch control and the crossbar. The switch control decides the path that the packet must follow and associates an output port to the input port. This can be done concurrently for all output ports, as the switch control generates routing information.

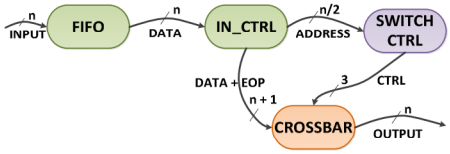

Figure 5 –BaBaRouter dataflow for the first flit of each packet.

As Figure 6 shows, when the following flits are received in the input port, they follow directly to the output port until the last flit passes. This is due to the fact that all active

CTRL channels of Figure 2are locked until the respective IN CTRL block signals EOP='1'. When the crossbar detects this situation, it sends the last flit to the output and clears the associated CTRL channel, by issuing an acknowledge signal to it. From this point on, that output port is free.

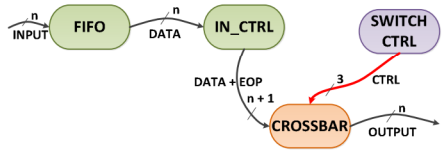

Figure 6 –BaBaRouter dataflow for flits after the first.

V. THE BABAROUTER IMPLEMENTATION FLOW

Figure 7 illustrates the proposed design flow, which was adopted for BaBaRouter. The functional behavior of this router was initially described in the Balsa language and compiled into handshake components through the Balsa compiler (Balsa-c), producing a Breeze Netlist. This netlist contains handshake components only, and can be simulated in the Balsa Framework to validate its functional behavior (with the Balsa Simulator).

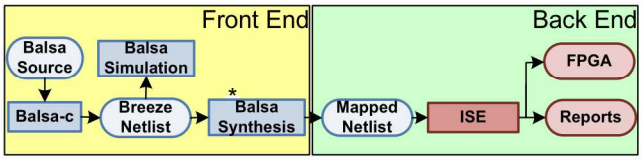

Figure 7 – The BaBaRouter design flow.

After extensive simulation of different random traffic scenarios, the design can be mapped to a specific technology. In this work, BaBaRouter was validated through its synthesis targeting Xilinx FPGAs, which arecompatible with the Balsa framework for generating QDI, four-phase DR style circuits. This synthesis produces a netlist of gates, which is described in Verilog and is fully compatible with commercial back-end tools. The generated mapped netlist is imported into the ISE Frameworkin order to be synthesized for a given FPGA.

The Balsa Front End allows design capture, simulation at the level of communicating processes, automatic synthesis and technology mapping. Design capture and simulation are independent of the final choice for an asynchronous template, which is left for the Balsa synthesis step (marked with * in Figure 7). In this way, freedom to explore the design space is guaranteed, because there is no constrain to choose asynchronous template at design capture time.This is in contrast to the use of structural design approaches such as the one employed for the design of Hermes-AA, where the asynchronous template must be chosen from the start.

It is useful to study the Balsa description of some blocks of the design, to assess the usefulness of the Front End. Figure 8shows the description of the BaBaRouter input FIFO and the register it employs. The first procedure (*Register*, in line 3) is the description of an asynchronous register with a parameter *width*, which defines how many bits it can store, an input i and an output o. The procedure also uses a variable x (line 8), that stores the data. Register input and output are just handshake channels; the storage hardware itself is denoted by the declared variable. The description of the Register behavior comprises lines 10 to 13, delimited by the reserved words begin andend. The semicolon in line 11 denotes that commands in lines 11 and 12 must be executed in sequence. Concurrent commands are also explicitly specified with the two vertical bars operator (||, as noted e.g. in line 29). The *Register* procedure consists of an endless loop (lines 10 to 14) that waits for a handshake request in input channel i and as it occurs, stores the input data in variable *x*(line 11). Next, it requests a communication in the output channel o to propagate the data stored in *x* (line 12). At the end of the loop (in line 13), input *i* waits for a new request. Thesemicolon of line 11 is responsible for the sequential semantics of the communications *i*→*x* and *x*→*o*.

```
1  import [balsa.types.basic]
2
3  procedure Register (
4          parameter width : type;
5          input i : width;
6          output o : width
7  ) is
8          variable x : width
9  begin
10         loop
11                 i -> x;
12                 o <- x
13         end
14 end
15
16 procedure fifo (
17         parameter depth : cardinal;
18         parameter width : type;
19         input i : width;
20         output o : width
21 ) is
22         procedure reg is Register(width)
23 begin
24         if depth = 1 then
25                 reg(i,o)
26         | depth >= 2 then
27                 local array 1 .. depth-1 of channel c : width
28                 begin
29                         reg(i,c[1]) ||
30                         reg(c[depth-1],o) ||
31                         for || i in 1 .. depth-2 then
32                                 reg(c[i], c[i+1])
33                         end
34                 end
35         else print error, "zero length fifo specified"
36         end
37 end
```

Figure 8 – Description of BaBaRouter *Register* and *fifo*.

The Register design is used as a module to build an asynchronous *fifo*, the next procedure in Figure 8. The *fifo* consists in a parameterizable number (*depth*) of interconnected *registers*. The *width* of the *register* instance is also parameterizable. The *fifo* interface consists in one input channel *i* and one output channel *o*, which share the same data width as the (internal) *registers*. The internal declaration of procedure *reg* binds to the *Register* module parameterized by the *fifo width* (line 22). This language construction is similar to the declaration of a component e.g. in VHDL. The *fifo* behavioral description (lines 24 to 36) tests the *depth*, using a construction with semantics similar to that of a conditional generate command in VHDL. If the

depth is 1 (line 24), a single register is instantiated (line 25). Otherwise (line 26), the description instantiates one register for the first position (line 29), one register for the last position (line 30) and depth-2 registers for the intermediate positions (lines 31 to 33), using a language construction similar to a for generate in VHDL. Note that all registers are configured to operate in parallel, through the concurrency operator (||) at the end of each line and in the for construction of line 31. In this way, there is no sequential operation at the fifo level, only inside each register. Communication and flow control are implied in the handshake and are independent of the specific asynchronous design style of the implementation. Also, as it can be seen from Figure 8, changes in Balsa the description lead to predictable changes in the resulting hardware.

VI. Experiments and Results

In order to validate the Babarouter presented in this work, a synthesis flow using Xilinx ISE Framework was defined. Also, a comparison was made between our balsa-based router implementation and the Hermes' regular VHDL implementation. Four different FPGA families were used for comparing the routers: Virtex5, Virtex6, Kintex 7 and Artix 7. As mentioned before, netlists generated by Balsa are fully compatible with ISE.

To make a fair comparison both routers were designed with similar parameters. However, Hermes stands as the synchronous implementation and the Babarouter as the asynchronous version. The adopted configuration was:

- 16 bits flit size
- 16 positions intermediary buffers size
- Round-robin arbitration
- Wormhole
- Handshake synchronization protocol

In this work, two outputs were taken into account for comparison: the area used to implement each router over the FPGA and the estimated power consumption. Table 1 presents the results obtained for the Hermes implementation. In the Table it is possible to see six columns: **I)** FPGA, which stands for the board family; **ii)** REGISTERs, as the name suggests, is the number of registers used to implement the logic over the specified FPGA family; **iii)** LUTs stands as the percentage of LUTS used; **iv)** TP is the estimated total power consumed by the router; **v)** DP stands for dynamic power. In this case the dynamic power is a function of the variation over the inputs, automatically estimated by the power tool, and; **vi)** SP, which is static power, or the power the router consumes when it's not performing any actions.

On the Table 2, the results regarding the Babarouter implementation are shown. As in Table 1, the same six columns are presented.

Table 1 – Obtained results for Hermes.

Hermes				
FPGA	**REGISTERs**	**LUTs**	**TP**	**DP**
Virtex 5	0.11%	0.35%	3.85	0.122
Virtex 6	0.06%	0.43%	5.905	0.107
Kintex7	0.04%	0.27%	0.444	0.09
Artix7	0.05%	0.30%	0.189	0.091

Table 2 – Obtained results for BaBaRouter.

BaBaRouter				
FPGA	**REGISTERs**	**LUTs**	**TP**	**DP**
Virtex 5	0.00%	2.37%	3.121	0.002
Virtex 6	0.00%	3.12%	5.797	0.003
Kintex7	0.00%	1.65%	0.356	0.003
Artix7	0.00%	2.18%	0.099	0.002

Table 3 presents a comparison between both implementations. The table shows the energy consumption difference in percentage and the difference in resources used in number of times. Analyzing all tables we can make some initial conclusions.

1. As expected, the dynamic energy consumption of the asynchronous version is much smaller. This happens due to to the very nature of these circuits to only consume where and when they are needed.
2. Babarouter's overall power consumption is smaller. Again, this is a characteristic of asynchronous circuits, and these results only confirmed what were already expected.

The overall area used is smaller on the synchronous version, which was also expected. While synchronous circuits can count on several tools to support its development, asynchronous circuits still lacks of such support. In this way a big gap between the two approaches become easily visible, as it is not possible, yet, to get a solution for asynchronous circuits as good as we can get for its synchronous counterparts. On the other hand, the asynchronous implementation didn't use any registers, what's desirable in this case, as registers are more expensive resources in FPGAs, and could be used to implement another IPs.

Table 3 – Results comparison.

Results Comparasion				
FPGA FAMILY	**Regs**	**LUTs**	**TP**	**DP**
Virtex 5	0.00	6.72x	-18,94%	-98,36%
Virtex 6	0.00	7.21x	-1,83%	-97,20%
Kintex7	0.00	6.16x	-19,82%	-96,67%
Artix7	0.00	7.27x	-47,62%	-97,80%

VII. CONCLUSIONS

This work presented the design and implementation of an asynchronous QDI DR NoC router. The router was described in a language specifically designed for asynchronous circuits, Balsa. As far as the authors could verify, this is the first asynchronous NoC router to be implemented using a high level design approach. Also, preliminary results show that by using Balsa to implement asynchronous intrachip networks power efficient designs can be obtained. Although it does incur overheads, the BaBaRouter is naturally tolerant to operational variations when compared to a synchronous NoC router. This is due to innate characteristics of asynchronous circuits and is advantageous in current technologies.

Future work includes exploring different routing algorithms, which can easily be added to the router. Also other asynchronous templates can be used to re-implement the router, including bundled-data and 1-of-4 encoding, combined with 2-phase or 4-phase protocols. In addition, the authors envisage the construction of an automatic generator for BaBaRouter, along with a high level simulation and evaluation environment.

REFERENCES

[1] International Technology Roadmap for Semiconductors. "Design Section", 2009,available at http://www.itrs.net.

[2] D. Chapiro. "Globally Asynchronous Locally Synchronous Systems". PhD Thesis, Stanford University, 1984, 134p.

[3] A. Agarwal, C. Iskander, and R, Shankar. "Survey of network on chip (NoC) architectures &contributions". Journal of Engineering, Computing & Architecture, vol. 3(1), 2009, 15p.

[4] D. Edwards, A. Bardsley, L. Janin, L. Plana, W. Toms,Balsa: A Tutorial Guide,Version 3.5, APT Group, University of Manchester, 2006, 157p.

[5] F. Moraes, N. Calazans, A. Mello, L. Möller, L. Ost, HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip. Integration, the VLSI Journal 38(1) (October 2004) 69-93.

[6] J. Sparsø and S. B. Furber. "Principles of asynchronous circuit design – a systems perspective".Kluwer Academic Publishers, Boston, 2001, 360 p.

[7] T. Verhoeff. Delay-insensitive codes- an overview. Distributed Computing, vol.3(1), 1988, pp.1-8.

[8] W. J. Bainbridge, W. B. Toms, D. A. Edwards, and S. B. Furber. "Delay-insensitive, point-to-point interconnect using m-of-n codes". In:ASYNC´03, 2003, pp. 132- 140.

[9] A. J. Martin."Formal program transformations for VLSI circuit synthesis".In: Formal Development of Programs and Proofs, E. W. Dijkstra, Editor, Addison-Wesley, 1989, pp. 59-80.

[10] T. Bjerregaard, J. Sparsø, A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip,in: Design, Automation and Test in Europe (DATE'05), 2005, pp. 1226-1231.

[11] A. Sheibanyrad, I. Miro-Panades, A. Greiner, Multisynchronous and Fully Asynchronous NoCs for GALS Architectures, IEEE Design and Test of Computers 25(6)(November-December 2008) 572 - 580.

[12] R. Dobkin, R. Ginosar, A. Kolodny,QNoC asynchronous router, Integration the VLSI Journal42(2) (February 2009) 103-115.

[13] Y. Thonnart, E. Beigné, P. Vivet,Design and Implementation of a GALS Adapter for ANoC Based Architectures, in: 14th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'09), 2009, pp. 13-22.

[14] J. Pontes, M. Moreira, F. Moraes, N. Calazans, Hermes-A - An Asynchronous NoC Router with Distributed Routing, in: International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'10), LNCS vol. 6448, 2010. pp. 150-159.

[15] J. Pontes, M. Moreira, F. Moraes, N. Calazans, Hermes-AA: A 65nm asynchronous NoC router with adaptive routing, in: IEEE International SOC Conference (SOCC'10), 2010, pp. 493-498.