PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE ENGENHARIA

FACULDADE DE INFORMÁTICA

ENGENHARIA DE COMPUTAÇÃO

DESIGN AND IMPLEMENTATION OF AN ASYNCHRONOUS NOC ROUTER USING A TRANSITION-SIGNALING BUNDLED-DATA PROTOCOL

MATHEUS GIBILUKA

END OF TERM WORK

ADVISOR: PROF. DR. NEY LAERT VILAR CALAZANS

CO-ADVISOR: PROF. MSC. MATHEUS TREVISAN MOREIRA

Porto Alegre
2013

MATHEUS GIBILUKA

DESIGN AND IMPLEMENTATION OF AN ASYNCHRONOUS NOC ROUTER USING A TRANSITION-SIGNALING BUNDLED-DATA PROTOCOL

End of Term work presented as part of the activities to obtain a degree of Computer Engineering at the Faculty of Engineering in the Pontifical Catholic University of Rio Grande do Sul.

ADVISOR: PROF. DR. NEY LAERT VILAR CALAZANS

CO-ADVISOR: PROF. MSC. MATHEUS TREVISAN MOREIRA

Porto Alegre
2013

ABSTRACT

Current silicon technologies enable the integration of billions of transistors in a single chip, supporting the creation of complex systems on a chip (SoCs). Networks on Chip (NoCs) constitute a suitable alternative for traditional SoC interconnect architectures, as they provide a high level of scalability and parallelism, supporting the ever-increasing number of cores in single chip. Additionally, synchronous design issues that were easily overcome in previous decades - such as clock distribution, skew, and power consumption - are becoming increasingly complex to solve in modern state of the art technology designs. Together, these trends constitute a good motivator for the development of an asynchronous SoC interconnect architecture. This work presents the design and implementation of an asynchronous NoC router using a transition-signaling bundled-data protocol. Additionally, a methodology for synthesis of bundled-data circuits using commercial CAD tools, together with an automated environment for enforcing relative timing constraints, is proposed. The router design was validated through behavioral simulation, and its basic block (a port) was synthesized, validating the implementation through post-synthesis simulation.

Keywords: Networks on chip, asynchronous circuits.

“Stay hungry,
stay foolish.”

Steve Jobs

CONTENTS

LIST OF ABBREVIATIONS

ACDC	Asynchronous Constraints for Design Compiler
DI	Delay Insensitive
EDA	Electronic Design Automation
FIFO	First In First Out
FSM	Finite State Machine
GALS	Globally Asynchronous Locally Synchronous
GAPH	Grupo de Apoio ao Projeto de Hardware
IC	Integrated Circuit
II	Input Interface
IP	Intellectual Property
ITRS	International Technology Roadmap for Semiconductors
MUTEX	Mutual Exclusion
NoC	Network-on-Chip
NRZ	Non Return-to-Zero
OI	Output Interface
PVT	Process, Voltage, Temperature
QDI	Quasi-Delay-Insensitive
RTZ	Return-to-Zero
SI	Speed Independent
SoC	System on a Chip
ST	Self Timed
STA	Static Timing Analysis
YeAH!	Yet Another Hermes

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

1. INTRODUCTION

According to the International Technology Roadmap for Semiconductors (ITRS), increasing transistor density, higher operating frequencies, short time-to-market, and reduced product life cycle characterize today's semiconductor industry [ITR11]. Smaller transistor feature sizes, now reaching ultra-deep submicron levels, enable the integration of billions of transistors in a single chip, supporting the creation of complex systems on a chip (SoCs). Typically, an SoC is composed by intellectual property (IP) cores, an interconnection architecture, and interfaces to peripheral devices [MOR04]. IP cores are pre-designed functional blocks such as a processor or a memory controller. The reuse of cores helps reducing the system's time to market.

Traditionally, the interconnect architecture of SoCs was based on dedicated wires or shared buses. The former approach is only applicable to systems containing a small number of cores, since the number of connecting wires increases quite fast as the number of cores grows. Shared buses are more scalable, but only one communication at a time is allowed, limiting its scalability to a few dozen IP cores [MOR04]. Networks on chip (NoCs), on the other hand, provide a higher level of communication parallelism and even higher scalability, when compared to bus-based interconnects [PON10b]. An NoC is an infrastructure that manages communication between IP cores. A NoC-based SoC typically consists of cores connected to routers, which are connected among themselves by communication channels [MOR04]. Each router may handle several simultaneous connections, increasing the available bandwidth. The TILE64™ [BEL08], the Cell processors [KAH05], the Intel Single Chip Cloud Computer [WIJ11] and the Intel Xeon Phi Coprocessor [INT13] are commercial examples of SoCs using an NoC as interconnect architecture.

Most of today's digital circuits are designed employing the synchronous paradigm: an externally generated global clock signal is used to create a discrete notion of time. This signal controls every storage element, as Figure 1.1(a) illustrates. When the clock transitions, registers sample the data at the input ports, and the stored values are displayed at the output ports – making data flow from one register to the next. The clock period is determined by the computation time of the slowest combinational logic block. This guarantees that data is only sampled when all signals are stable. Therefore, feedback hazards and signal glitches that may occur as combinational circuits stabilize can be ignored, greatly simplifying the design process. However, synchronous design issues that were easily overcome in previous decades - such as clock distribution, skew, and power consumption, are becoming increasingly complex to solve in modern state of the art technology designs. For instance, the clock signal in a high-speed processor represents an average of 45% of the total power [AMD05]. This makes complex synchronous systems less attractive for low power applications.

Asynchronous circuits are an alternative to tackle the problems created in advanced nodes by the use of the synchronous paradigm. As Figure 1.1(b) exemplifies, this class of circuits does not employ a clock signal. Instead, local handshakes between adjacent components perform the necessary synchronization, communication and sequencing of operations [SPA01]. This potentially reduces power consumption, and eliminates problems related to clock distribution and skew [HAU95] [SPA01]. However, the lack of techniques, methodologies and electronic design automation (EDA) tools fully supporting asynchronous systems prevents traditional circuit designers from taking full advantage of asynchronous circuits [PON10a] [HAU95].

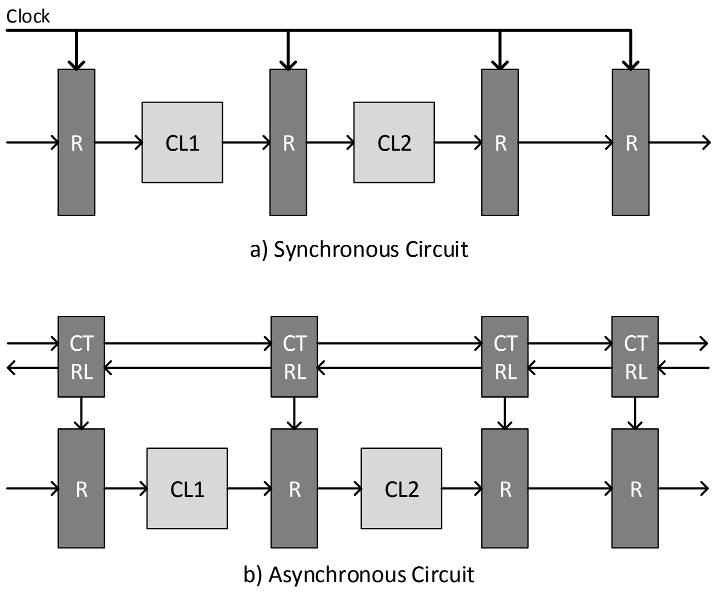


Figure 1.1 - Example of (a) a synchronous circuit, and (b) an asynchronous circuit. CLi represent combinational logic blocks, R symbolizes registers, and CTRL indicates control logic. Adapted from [SPA01].

A globally asynchronous locally synchronous (GALS) [CHA84] system is intermediate between a fully synchronous and a fully asynchronous design. In a GALS system, distinct clock signals are used to govern different modules. Internally, each module works as a fully synchronous system; different modules may operate at different clock rates. GALS techniques eliminate the burden of creating a clock distribution network across the whole chip, simplifying the achievement of timing closure in SoC designs [PON10b]. However, synchronization interfaces must be employed to enable reliable communication between distinct clock domains, introducing latency penalties in the system [PON10b] [SHE08].

Networks on chip can be implemented as fully synchronous, GALS, or asynchronous circuits. To avoid a bottleneck at the interconnection, a fully synchronous NoC needs to operate at a frequency high enough to meet bandwidth requirements of the most demanding core, which may result in significant, wasteful power consumption on routers that do not require high throughput [GEB10]. A GALS NoC allows controlling the operating frequency (and bandwidth) of each router, potentially reducing the dynamic power consumption when compared to a single-frequency NoC. However, as mentioned before, synchronization interfaces must be added between routers and cores operating in different clock domains [SHE08].

An asynchronous NoC allows a more flexible integration of components with different timing characteristics. Besides presenting reduced overall power consumption when compared with other implementation styles, zero dynamic power is naturally achieved when the NoC is idle [GIL11] [GEB10] [BJE05]. Also, latency overheads caused by the synchronization interfaces in GALS NoCs reduce the data throughput of the system. An approach to ease this issue is to reduce as much as possible the number of synchronization points throughout the SoC. By employing an asynchronous NoC as communication architecture, only two synchronization interfaces need to be traversed by each packet: one at the sender's output port, and another at the receiver's input port [PON10a].

Since the last decade, the research on asynchronous routers and NoCs has gained momentum [PON10b]. Asynchronous NoCs can reduce interconnect overall power consumption, while eliminating increasingly complex clock-related design problems [GIL11] [BJE05]. However, most of the previous works make use of delay insensitive data encodings, usually resulting in large area overheads, or bundled-data level-signaling implementations, which increase the overall latency of the system, when compared to transition-signaling protocols [GHI13].

The main goal of this work was the design of a bundled-data asynchronous network-on-chip router. Along with it, a methodology for synthesis of bundled-data circuits using commercial CAD tools was proposed and an automated environment for enforcing relative timing constraints was developed. In this work, the Author explored several topics not covered during the undergraduate program in Computer Engineering, in particular asynchronous circuits and NoC design, while gaining practical knowledge about the digital system design process. Furthermore, due to the lack of automated design and synthesis tools for asynchronous systems, many challenges had to be overcome to synthesize bundled-data circuits. This presented a unique opportunity to study about how EDA tools work, and learn how to adapt them to provide some degree of automation to the process of designing asynchronous systems.

The remainder of this work is organized as follows. Chapter 2 provides relevant background information. Chapter 3 presents the state of the art in asynchronous NoCs. Chapters 4 and 5 present developed work. The specification and architecture of the designed router is detailed in Chapter 4. The synthesis process is presented in Chapter 5. Final remarks and directions for future work are explored in Chapter 6.

2. CONCEPTS

This Chapter presents some basic concepts about Network-on-Chips (NoCs) and asynchronous circuits, which are needed to support this work.

2.1 Networks on Chip

On-chip communication implemented with dedicated wires is only effective for systems with a small number of cores, as it presents poor reusability and scalability. Shared buses are more scalable and reusable, but only one communication at a time is allowed and the bandwidth is shared among all cores. Moreover, it also lacks scalability, given that all cores share the same communication medium. The NoC approach address all these issues. It consists of a communication infrastructure in which cores are connected to routers and these communicate among themselves through channels. NoCs are reliable, energy efficient, reusable, and present much better bandwidth scalability when compared to traditional bus architectures [MOR04]. In fact, the use of NoCs is already a well-established concept for creating effective intrachip communication infrastructures for contemporary systems. The TILE64™ [BEL08], the Cell processors [KAH05], the Intel Single Chip Cloud Computer [WIJ11] and the Intel Xeon Phi Coprocessor [INT13] are commercial examples of NoC usage.

2.1.1 Hermes NoC

The Hermes NoC, proposed in [MOR04], comprises routers with a set of bidirectional ports that connect to an IP core and to other routers, according to a topology [MOR04]. Figure 2.1 exemplifies a 3x3 Hermes NoC with a 2D mesh topology. Each router has five bi-directional ports: East, West, North, South, and Local. The Local port links to a local IP core, and the other ports link to neighbor routers. Each router's address is expressed by its XY coordinates.

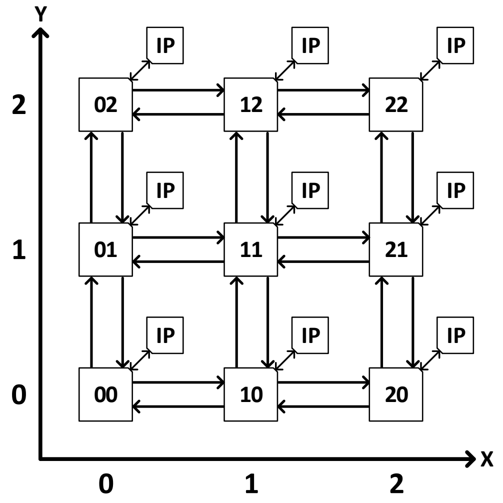


Figure 2.1 – 3x3 Hermes NoC, connected in a 2D mesh topology. Adapted from [MOR04].

IP cores communicate with each other through message exchange. To forward messages across the network, routers employ a wormhole packet switching approach; therefore, a packet is transmitted between routers in units called *flits*, an abbreviation of *flow control digits*. A 2-flit header and a variable length payload compose each packet: the first flit contains the packet's destination address and the second indicates the number of flits in the payload. When the header flit goes through a router, it reserves a path from an input port to an output port for the whole packet; the succeeding flits of the packet flow through this reserved path. Wormhole switching provides low latency, requires less memory, and facilitates the multiplexing of one physical channel into more than one logic channel [MOR04].

Each router can handle up to five simultaneous connections. The router's control logic is centralized in a single block and comprises two basic modules: routing and arbitration. The routing logic analyzes the packet's header, calculates to which port the packet should be sent, and connects the input port to the correct output port. The arbiter acts as a tiebreaker when more than one input port attempts to connect to the same output port. One of its goals it to prevent starvation, i.e. provide a balanced and fair usage of the output ports between input ports. Each input port has a buffer to help reduce performance degradation caused by a possibly blocked flit [MOR04].

2.2 Asynchronous Circuits

Most of today's digital systems are synchronous, which means they employ a global clock signal to synchronize the operation of all sequential components. This creates the abstraction of discrete time. In an asynchronous circuit, the coordination of modules is performed without a clock signal [MYE01]. Instead, handshake protocols are used to perform the necessary synchronization, communication and sequencing of operations [SPA01]. The discrete-time abstraction of synchronous circuits helps simplifying the design, but removing it can grant several other benefits, like lower power consumption, higher operating speed, lower electromagnetic noise emission, and eliminating clock distribution problems [HAU95] [SPA01]. However, depending on the handshake protocol and data-encoding scheme chosen, asynchronous control logic may introduce significant area, circuit speed, and power consumption overheads [SPA01]. Another drawback of asynchronous circuits is the lack of mature EDA tools supporting their design.

2.2.1 Delay Models

Asynchronous circuits can be classified with respect to the assumptions employed on gate and wire delays during their design. A delay-insensitive (DI) circuit can operate correctly no matter the magnitude of all delays found in the circuit. This means, referring for example to Figure 2.2, that the circuit in that Figure works for any arbitrary values of d_A, d_B, d_C, d_1, d_2, and d_3 [SPA01]. DI is the most robust class of asynchronous circuits, but it is very limited, since systems respecting this assumption can only contain C-elements as multiple-input operators [MAR90].

A circuit that operates correctly without assumptions regarding delays, except on some specific wire forks, called *isochronic forks*, is called quasi-delay-insensitive (QDI) [MYE01]. A fork is isochronic if the delays at all endpoints of the fork are identical (or differ by a very small amount that is controlled during design) [SPA01]. Considering the circuit represented in Figure 2.2, this means that either $d_2 = d_3$, or $\delta=|d_2 - d_3|$

is smaller than some given value κ. Even though this assumption may seem of little significance, it does change model properties considerably. By making forks isochronic, only one end of the fork needs to be sensed; therefore, when the sensing gate fires, the signal will have guaranteedly reached all ends of the fork [HAU95]. A further discussion about QDI circuits is outside the scope of this work, and can be found, for example, in references [SPA01] and [HAU95].

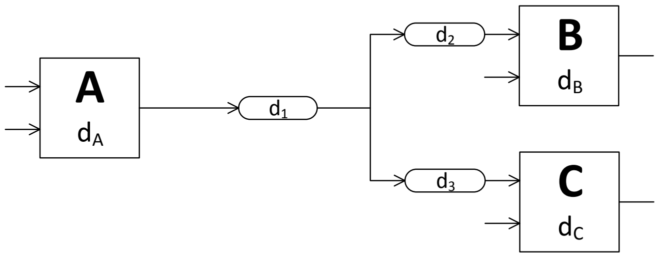


Figure 2.2 - A circuit fragment with gate and wire delays [SPA01].

A circuit is called speed-independent (SI) when it can operate correctly assuming unknown gate delays and zero-delay wires. Again, considering the circuit shown in Figure 2.2, this means that the circuit work regardless of the values of d_A, d_B, and d_C, but assuming that $d_1 = d_2 = d_3 = 0$. Of course, it is often unrealistic to assume zero-delay wires in today's semiconductor processes [SPA01].

Delay-insensitivity and speed-independence have mathematically well-defined properties. Circuits whose correct operation relies on more elaborate timing assumptions are called self-timed (ST) [SPA01]. An ST circuit assumes that both gate and wire delays are known, and delay lines are used to satisfy timing assumptions. These are often called *matched-delay* asynchronous circuits, and are usually simpler than QDI in terms of the required hardware [STE09]. However, some of the drawbacks are its worst-case operation and the sensitiveness to process, voltage, and temperature (PVT) variations [KRZ10]. However, several works available on literature report that these circuits are usually more suited to high speed and low power applications than those designed using QDI techniques and assumptions [GHI13][BHA13][NOW11][STE09].

2.2.2 Handshake Protocols

Instead of using a clock signal, asynchronous circuits use handshake protocols between components to perform synchronization [SPA01]. A handshake consists in the exchange of a request and an acknowledge signaling between an active and a passive element, establishing synchronization between both. The active component starts the communication by issuing a request (*req*); then it waits for the arrival of the corresponding acknowledge (*ack*), generated by the passive element [PEE96]. This communication mechanism can be implemented using two basic approaches: *transition-signaling* and *level-signaling*; also, implementations usually rely on one of four possible channel configurations: *nonput*, *push*, *pull*, and *biput* [KRZ10].

The level-signaling handshake protocol takes four signal events to complete a communication cycle – this signaling scheme is accordingly called a *four-phase protocol*, and sometimes named *return-to-zero* (RTZ) handshake [SPA01]. The events required to complete the handshake, as exemplified in Figure 2.3(a), are: *(1)* the active component initiates the communication by asserting *req*; *(2)* the passive component senses the request and, in response, asserts *ack*; *(3)* the active element detects the acknowledge and deasserts *req*; (*4)* finally, the passive component notices the deasserted request, and responds by deasserting *ack*. At this point the handshake is completed, and the active component may start another communication cycle. A big disadvantage of level-signaling protocols is its return-to-zero phase (events 3 and 4), as they cost extra time and energy [SPA01]. The transition-signaling protocol eliminates this disadvantage.

On the transition-signaling protocol, often called *two-phase protocol* or *non-return-to-zero* (NRZ) signaling, handshake events are encoded as signal transitions – there is no difference between a low-to-high and a high-to-low transition, as both imply the same type of event [SPA01]. This signaling scheme is illustrated in Figure 2.3(b). The active component starts the communication by issuing a request event – that is, switching the logic level of *req*. Then, the passive component senses the request transition, and ends the communication by sending out an acknowledge event – that is, changing the logic level of *ack*. At this point, the active component may start a new communication. Transition-signaling protocols can complete a handshake in half the time of a level-signaling one. However, depending on the used data encoding, the transition-signaling control unit may introduce more significant area and power overheads [SPA01].

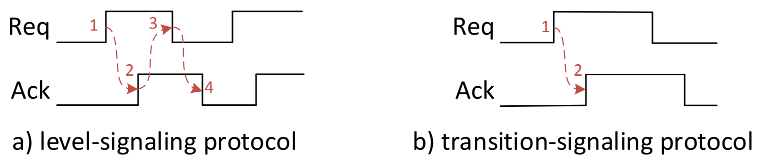


Figure 2.3– Handshake operations using (a) a level-signaling protocol and (b) a transition-signaling protocol. Adapted from [SPA01].

In addition to establishing synchronization between components, handshakes can also be used to transfer data between elements – this can be achieved by encoding data in the request, acknowledge, or in both events [PEE96]. Handshake channels that do not convey data are called nonput channels. They only have the request and acknowledge wires, as shown in Figure 2.4(a), and are used for synchronization purposes only. A push channel has, in addition to the *req* and *ack* wires, a *data* bus going from the active component to the passive component, as Figure 2.4(b) shows – it can be said that the active component *pushes* data through the channel [PEE96]. In this type of channel, data needs to be valid before issuing a request event. If the data flows from the passive to the active component, as depicted in Figure 2.4(c), it is said that the active component *pulls* the data through the channel – characterizing a pull channel configuration. A channel with such configuration requires that data be valid before the acknowledge event occurs. Data flowing both ways characterizes a biput channel – in this situation, data sent by the active element needs to be valid before the *req* event, and data coming from the passive component needs to be valid before the *ack* event. This channel configuration is illustrated in Figure 2.4(d).

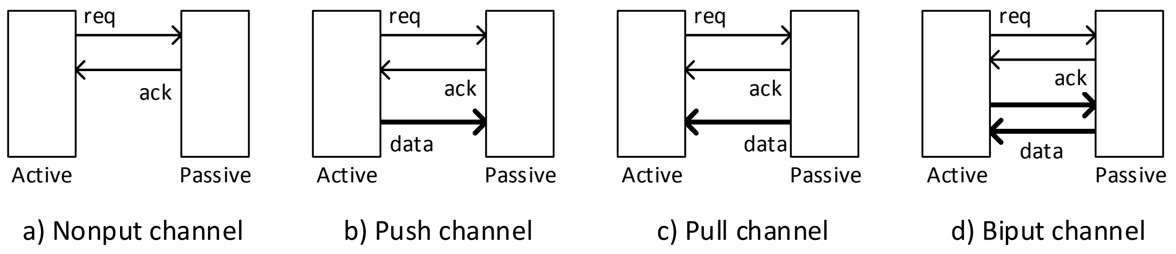


Figure 2.4 – Handshake channel types: (a) Nonput; (b) Push; (c) Pull; (d) Biput.

2.2.3 Bundled-Data

In a synchronous circuit, the role of the clock is to define points in time where data and control signals are stable and valid. Between cycles, signals may present hazards and make multiple transitions as combinational circuits stabilize [SPA01]. Asynchronous circuits, on the other hand, are usually designed to detect the arrival of data. Assuming a push channel, data validity must be inferred either from the data channel, as in QDI, or the explicit request signal, as in ST [KRZ10]. This choice is a trade-off between speed, robustness, area, and power [SPA01]. For instance, dual-rail is a very robust, delay-insensitive, data encoding scheme employed in several QDI designs. It consists of encoding each bit with a pair of wires – one, marked as the *true wire*, represents the bit value 1 when at the logic level 1, and the other, called *false wire*, denotes the bit value 0 when at the logic level 1 [SPA01]. When viewed together, each wire pair is a codeword that represents one bit of data or the absence of data (when both wires are at the logic level 0). Due to the DI characteristic of this encoding scheme, the passive element can detect data validity, when all bits have arrived, and, from this, infer the request event and generate an acknowledgement, which can be signaled using a single wire. However, the use of two wires per bit implies silicon area overhead and increases the circuit switching activity, given that every data sent requires the switching of exactly one wire per bit of information [KRZ10].

An alternative is the use of regular Boolean encoding, where one data bit is represented by one wire. This encoding scheme, however, does not convey data validity information. Therefore, an explicit signal (the request mentioned before) is required and the circuit designer needs to guarantee that data is valid before the detection of an event on this signal [SPA01]. The term *bundled-data* is commonly used to designate this kind of encoding: it suggests that the request and acknowledge wires are bundled with the data signals, hinting on the timing relationship between them. Figure 2.5(a) illustrates a bundled-data push channel. In the example, a delay line is used to match the delay of *req* to the worst-case delay of the data channel. A circuit using bundled-data scheme falls under the ST category, and either handshake protocol can be used.

The operations of the bundled-data level-signaling and the transition-signaling protocols are illustrated in Figure 2.5(b) and in Figure 2.5(c), respectively. The events required to complete the communication are the same discussed in Section 2.2.2. However, the designer needs to guarantee that, whenever there is a data exchange between an active and a passive components, data signals are valid and stable before the handshake event that notifies the data arrival is detected – that is, the request event on a push channel, the acknowledge event on a pull channel, and both events on a biput channel.

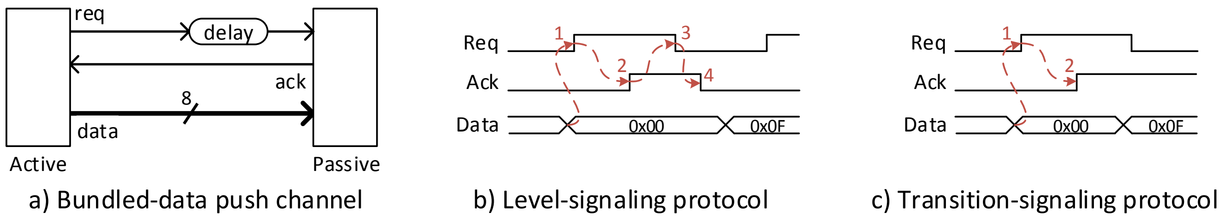

req
delay
ack
8
data
Active
Passive
a) Bundled-data push channel
Req
Ack
Data
1
2
3
4
0x00
0x0F
b) Level-signaling protocol
Req
Ack
Data
1
2
0x00
0x0F
c) Transition-signaling protocol

Figure 2.5 – An example 8-bit bundled-data channel scheme and its operation: (a) a bundled-data push channel; (b) an example of a level signaling communication; (c) an example transition-signaling communication. Adapted from [SPA01].

Therefore, asynchronous circuits' communication protocols are characterized by choices of: a data encoding scheme, a channel configuration, and a handshake protocol. Dual rail encoding is a good choice for robustness and delay-insensitivity, but it introduces significant power and area overheads [KRZ10].

Transition-signaling bundled-data circuits are a suitable choice for systems with high-speed requirements [GHI13] [BHA13] [NOW11] [STE09] [SPA01]. However, some extra design effort is needed to guarantee the fulfillment of delay constraints between data and handshake signals. Since the throughput of the circuit depends on the time it takes to complete a handshake cycle, level-signaling bundled-data has a speed penalty inherent to its handshake protocol [SPA01].

2.2.4 Asynchronous Logic Components

Unlike synchronous systems, asynchronous circuits operate with a continuous notion of time – that is, events, represented by signal transitions, may occur at any moment. To support this characteristic, several asynchronous logic components have been proposed throughout the years. Two of these components are used in this work: the C-element and the mutual exclusion (*mutex*) gates.

A mutex is used to arbitrate between events that can happen simultaneously. Its task is to forward independently generated request signals at inputs RA and RB to its corresponding outputs GA and GB guaranteeing that at most one output is active at any given time [SPA01]. The C-element is a state-holding gate that can be used for event synchronization: its current output value is held when its inputs are distinct; when all its inputs have the same value, the output changes to reflect that same value. For example, a 2-input C-element changes its output to 0 when both inputs are 0 and to 1 when both inputs are 1, for other combinations the output is kept unchanged [SPA01].

The ASCEnD cell library [MOR11a] [MOR11b], designed at the GAPH, includes more than six hundreds instances of the components mentioned in this Section in several forms, some of which were employed in this work.

2.2.5 High Performance Pipelines

Pipelining is a technique classically used to increase parallelism in a digital system and, consequently, increase the throughput of the latter. In synchronous systems, a pipeline design consists in partitioning complex logic blocks into smaller blocks, where adjacent blocks are separated by registers all of which are controlled by a single clock signal. Since asynchronous circuits do not have such a clock signal, a protocol to control the interaction of neighboring stages must be defined in an asynchronous pipeline, along with the choice of a data encoding and of storage elements [NOW11]. Together, these design choices compose a design template for asynchronous pipelines.

Figure 2.6(a) shows an abstract view of a synchronous pipeline. In such a system, data moves to the next stage at the end of every clock cycle, and this occurs simultaneously in all stages. To work properly, the pipeline clock period needs to be larger than the computation time of the slowest stage. In an asynchronous pipeline, exemplified in Figure 2.6(b), the data flow is controlled by handshakes between stages. Usually, a stage accepts new data if the previous stage is providing new data and the next stage has already stored the previous data item. Unlike what happens in synchronous pipelines, in an asynchronous pipeline different data values can flow at different rates, given that data propagation is not constrained by the worst case delay – which may improve the system's average latency and throughput [NOW11].

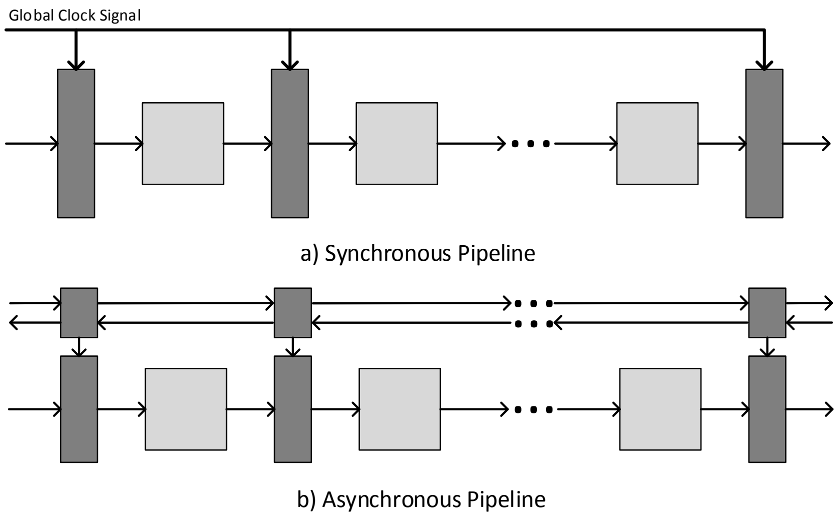


Figure 2.6 – An abstract view of pipelines: (a) a synchronous pipeline; (b) an asynchronous pipeline. Adapted from [NOW11].

Asynchronous pipelines can be classified based on the logic style of its datapath: static or dynamic. Classically, dynamic logic refers to circuit implementations where the pull-up network is removed, resulting in smaller area usage and higher switching speed [RAB03]. However, this type of implementation increases the design and validation effort [NOW11]. For this reason, this work considers only static logic pipelines.

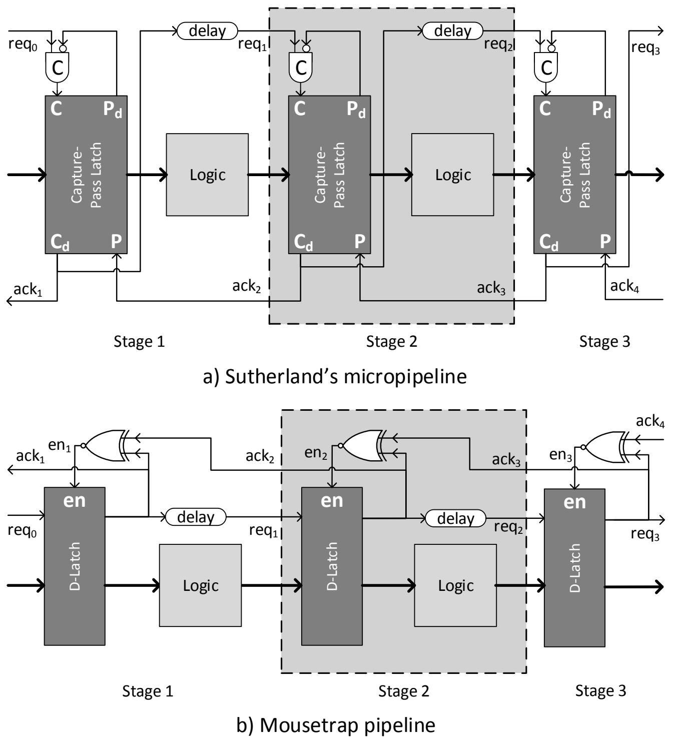


Figure 2.7 – Two transition-level asynchronous pipelines: (a) Sutherland's; (b) Mousetrap. Adapted from [NOW11].

Ivan Sutherland pioneered the transition-signaling bundled-data asynchronous pipeline in the late 1980s. Each stage of his proposed pipeline, shown in Figure 2.7(a), has a C-element as the control unit and a special capture-pass latch capable of sensing transitions at the inputs. The Mousetrap pipeline [SIN07], shown in Figure 2.7 (b), is based on Sutherland's micropipeline [SUT89], but features less complex signaling and lower overhead [NOW11]. Mousetrap is controlled by a 2-input XNOR gate, and data is stored in a bank of standard active-high level-sensitive D latches. Figure 2.8 exemplifies the operation of a Mousetrap pipeline stage: *1)* the incoming request (*req_1*) traverses the latch and causes the XNOR gate to switch its output to low (*en_2*), making the latch opaque; in parallel, the latch produces the acknowledge (*ack_2*) signal; *2)* once the acknowledge is received in the previous stage, the latter can start a new handshake; *3)* the output request (*req_2*) flows through the delay line and arrives at the next stage; *4)* the next stage acknowledges it, causing the XNOR gate to switch its output to high, making the latch of stage 2 transparent again, which is the original state for all latches. From this moment on, a new handshake cycle can start.

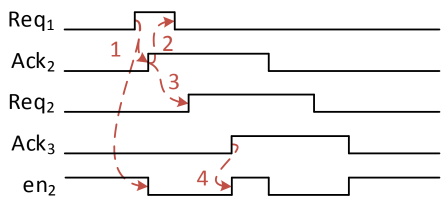
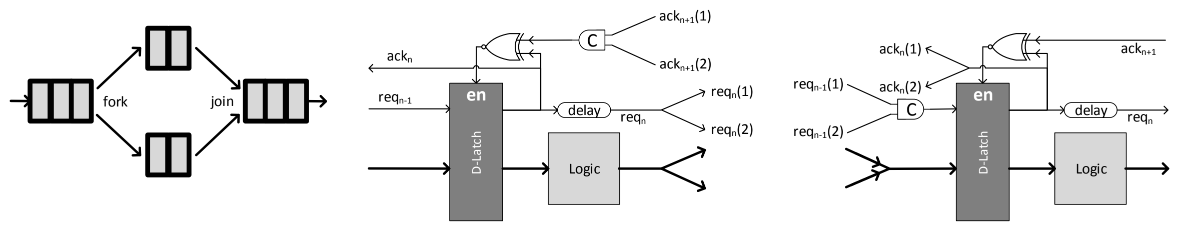


Figure 2.8 – Example of the Mousetrap pipeline operation for Stage 2 signals.

Many practical applications can be implemented with linear pipelines. However, complex system architectures often require nonlinear pipelines [SIN07]. This implies using pipelines with parallel stages, as Figure 2.9 – Nonlinear pipelines: (a) shows. The Mousetrap template can be easily adapted to support such systems. A pipeline fork stage can only accept a new request when all forking stages have acknowledged the current one. This behavior can be achieved by the use of a C-element to join the *ack* signals coming from parallel stages, as Figure 2.9(b) shows. Similarly, a join stage must wait until all requests from the parallel stages have arrived to start processing new data, which can be accomplished by employing a C-element at the incoming request signals, as Figure 2.9 – Nonlinear pipelines: (c) illustrates.



a) Nonlinear pipeline b) Mousetrap fork stage c) Mousetrap join stage

Figure 2.9 – Nonlinear pipelines: (a) Example of a nonlinear pipeline structure; (b) a Mousetrap fork stage; (c) a Mousetrap join stage. Adapted from [SIN07].

3. STATE OF THE ART

This Chapter presents an overview of the current literature on asynchronous network on chip routers based on 5-port architectures. As metrics used by each author differ from one another and the designs target different technologies using different delay models, there is not enough data to draw a fair comparison of all implementations. However, most authors provide area usage and throughput information in units of flits/s. Table 1 summarizes the attributes of the NoCs covered in this Section.

Table 1 – Comparison of 5-port asynchronous NoC routers.

Router	Asynchronous Style	Technology feature size	Flit Size	Area	Average Latency	Average Throughput
Ghiribaldi et al. [GHI13]	Bundled-data, transition-signaling	40nm	32 bits	4,691µm2	1.195ns	35.4 Gbytes/s per port
Hermes-AA [PON10b]	Dual-rail, level-signaling	65nm	8 bits	114,456µm2 for XY routing, 116,031µm2 for WF routing	3.709ns for input port, 1.351 for output port	7.7 Gbits/s per router
ASPIN [SHE08]	Bundled-data, level-signaling	90nm	32 bits	36,199µm2	1.5ns	4.4 Gbytes/s per router
ANoC [BEI05]	QDI, level-signaling	160nm	34 bits	0.25mm2	2ns and 2.5ns, depending on traffic priority	5.0 Gbytes/s per router
MANGO [BJE05]	Bundled-data, level-signaling	120nm	32 bits	0.188mm2	N.A.	15.4 Gbytes/s per router (under typical conditions)
Hermes-A [PON10a]	Dual-rail, level-signaling	180nm	8 bits	0.33mm2	N.A.	3.6 Gbits/s per router
Async. QNoC [DOB09]	Bundled-data, level-signaling	180nm	8 bits	N.A.	N.A.	1.1 Gbytes/s per router

3.1 The NoC of Ghiribaldi et al.

Ghiribaldi et al. [GHI13] propose an asynchronous NoC router based on the Mousetrap pipeline template, accordingly using a transition-signaling bundled-data protocol. The router features wormhole switching, 32-bit flits, and algorithmic dimension-order routing. The design was synthesized using a low-power standard-Vt 40nm cell library, with normal process, 1.2V supply voltage, and operating temperature of 300K.

The router presents in average a latency of 1,195ps, and a cycle time of 903ps. Latency is defined as the time it takes for a head flit to traverse from the input port to the output port, assuming an empty router and no congestion. Cycle time is the interval between two successive acknowledgments received at the router output port. The area usage is 4,691µm^2. Based on the average cycle time and on the flit size it is possible to compute the NoC average throughput: 35.4 Gbits/s per port.

3.2 The ANoC

Proposed by Beigné et al., the ANoC [BEI05] is a QDI router implemented with level-signaling handshake protocol and support for two virtual channels – one for real-time low latency packets, and other for best-effort traffic. It features wormhole routing, odd-even turn model adaptive routing algorithm, and 34-bit flits – 32 bits of data, plus 2 control bits. The router was synthesized using STMicroelectronic's HCMOS9LL 0.16µm technology. The area used by each router is 0.25mm^2 and the average cycle time is 4ns, giving the maximum throughput of 5Gbytes/sec, when all inputs and outputs run concurrently. The router latency is 2ns for the high priority virtual channel and 2.5ns for the low priority one.

3.3 The MANGO NoC

The MANGO [BJE05] NoC, proposed by Bjerregaard and Sparso, employs virtual channels to provide connection-oriented guaranteed services and connectionless best-effort routing. Communication is performed using a level-signaling bundled-data protocol. Each router implements wormhole packet switching, 32-bit flit size, and XY routing algorithm. The NoC was synthesized using 0.12µm CMOS standard cell technology. Under worst-case timing parameters (1.08V supply voltage and 125ºC operating temperature), each port presents a performance of 515MHz; under typical conditions, 795MHz was obtained. This corresponds to a worst-case maximum throughput of roughly 10GBytes/sec and a typical case of roughly 15.4GBytes/sec. The router area usage is 0.188mm^2.

3.4 The Asynchronous QNoC

Dobkin et al. [DOB09] proposed the asynchronous QNoC, a network on chip supporting quality-of-service in four distinct service levels, each with two virtual channels. The routers are implemented using a level-signaling bundled-data protocol, XY routing algorithm, and wormhole packet switching with 8-bit flit size. The NoC was synthesized using a 0.18µm CMOS standard cell library from Tower Semiconductor Ltd. The minimal router data cycle was 4.5ns, resulting in a throughput of 220Mflits/s per port, or 1.1GBytes/sec per router.

3.5 The ASPIN NoC

Proposed by Sheibanyrad et al., the ASPIN NoC [SHE08] uses both bundled-data and dual-rail data encoding: the latter is used only for long wires. Each router features wormhole packet switching, 32-bit flit size, and the XY routing algorithm. Synthesis was targeted at the STMicroelectronics 90nm low-voltage threshold technology. The area used by each router was 36,199µm^2. The maximum throughput is 131Mflits or 4.423Gbytes/s, and the average packet latency is 1.5ns.

3.6 The Hermes-A and Hermes-AA NoCs

Pontes et al. proposed two asynchronous delay-insensitive NoC router designs: Hermes-A [PON10a] and Hermes-AA [PON10b]. In both cases, routers implement wormhole packet switching, 8-bit flit width, and operate using a QDI dual-rail level-signaling protocol. The first design, Hermes-A, features a distributed XY routing algorithm and uses the XFab 180nm technology – with typical transistor models, 1.8V supply voltage, and operating temperature of 25ºC. The second design uses another option of routing algorithm: West-First (WF); Hermes-AA was synthesized using general-purpose standard-Vt 65nm technology from STMicroelectronics; the router operates at 25ºC with a supply voltage of 1V.

Hermes-A has a throughput of 727Mbits/s on each router link. In the best case, this performance level can be sustained on all five ports, resulting in a maximum throughput of approximately 3.6Gbits/s for the whole router. In this situation, the total power is 11.14mW. The total area usage is 0.33mm^2, of which 0.22mm^2 correspond to standard cell area.

Hermes-AA was able to produce 1.55Gbits/s of throughput on each router link – resulting in a maximum router throughput of 7.75Gbits/s, when all five ports are transmitting in parallel. According to the authors, per-port throughput can reach up to 6.3Gbits/s when asynchronous FIFOs are added to the input ports. However, this value goes back to the saturation throughput (1.55 Gbits/s) when FIFOs are full. Starting from an idle router, the input port latency reaches 3.709ns, and the output port latency is 1.351ns. When using the XY routing algorithm, the area usage is 114456µm^2 (75144µm^2 of standard cell area); with the WF algorithm, there was a slight increase of area: 116034µm^2 (76455µm^2 of standard cell area). For both routers, asynchronous cells such as C-elements were taken from the ASCEnD library [MOR11a] [MOR11b].

4. DESIGN

This work focuses on the design of a high-performance asynchronous NoC router based on the Hermes NoC. A transition-signaling bundled-data handshake protocol is a suitable choice for systems with high-speed requirements [SPA01], as they have the potential to lead to low area overheads, when compared to delay-insensitive encoding. Due to its simple control logic, the Mousetrap pipeline template [SIN07] is used for controlling the communication between components.

This Chapter details the specification and architecture of proposed router, and presents the functional validation of the design, obtained through behavioral simulation. The VHDL implementation files are available in Appendix A.

4.1 Specification

4.1.1 Target Architecture

The YeAH! NoC router, designed by the Hardware Design Support Group (GAPH) from the PUCRS, employs a fully distributed control logic to implement functionality similar to that of a Hermes router, introduced in section 2.1.1. In its design, routing and arbitration responsibilities are assigned, respectively, to each input and output interface of the router, instead of being centralized in a control block. This results in a highly modular design that avoids bottlenecks in the control unit, improving performance in congested networks. Figure 4.1 illustrates the architecture of the YeAH! router. Each port comprises two independent components: the input interface and the output interface. Each input interface has a set of independent control channels connected to all output interfaces. Up to five simultaneous internal connections can be handled by the router.

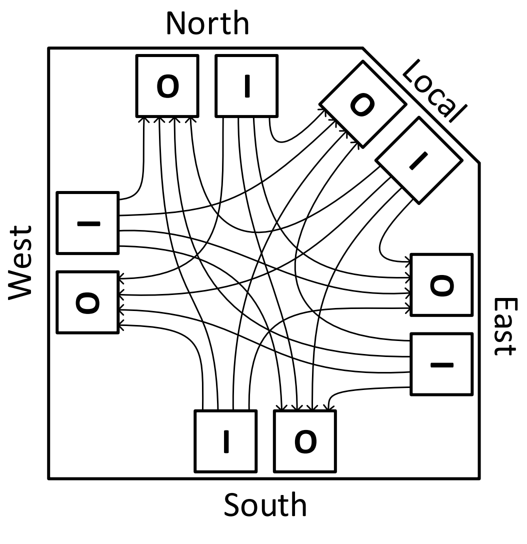


Figure 4.1 – Architecture of the YeAH! router.

Figure 4.2(a) displays the input interface architecture, which consists of an input buffer, a control unit, and the routing algorithm. When the Buffer outputs a header flit, the Control activates the Routing block that issues a request to the chosen output port. Once the request is granted, the packet is transmitted. The Control logic uses the payload size information contained in the second flit of each packet to identify header flits from payload flits. The buffer is implemented as a circular FIFO. Currently, the XY routing algorithm is the only one supported in YeAH!.

Each output interface, as Figure 4.2(b) illustrates, comprises an Arbiter and a (output) Control. The former implements a round-robin algorithm to select which input port should gain access to the output port when multiple simultaneous requests are pending. The (output) Control acts as a multiplexor, selecting which data channel should be sent to the output based on the arbiter's choice.

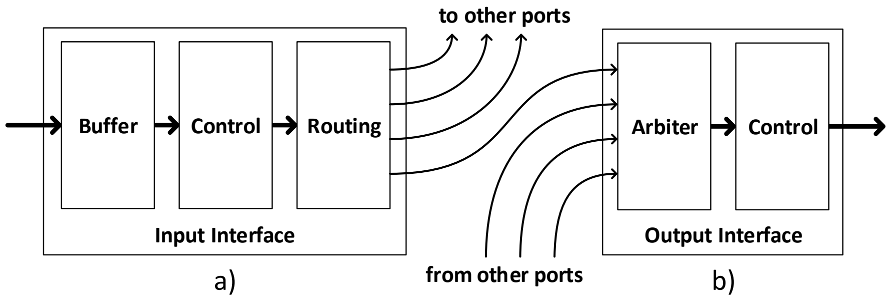


Figure 4.2 - Detailed view of input (a) and output (b) interfaces of the YeAH! router.

4.1.2 BaT-Hermes Specification

The architecture of the bundled-data transition-signaling Hermes router (BaT-Hermes) is based on the YeAH! router. Its highly decoupled architecture makes it a good starting point for the design of an asynchronous router as each individual module can be implemented as a handshaking module. In fact, the basic structure shown in Figure 4.1 can be kept, as communication between modules is replaced by handshake protocols. The router can have up to 5 ports, each composed of an Input and an Output Interface. Each Input Interface (II) is connected to all Output Interfaces (OI), but the one located in the same port – that is, there is no loopback connection. Figure 4.3 shows the top view of each interface. IIs are responsible for buffering and routing incoming packets to the correct OI. OIs, in turn, arbitrate between incoming packets from different IIs. Therefore, a 5-port router can handle up to five simultaneous connections, where all OIs receive a packet from a different II. BaT-Hermes is parameterizable with respect to flit size, buffer depth, and which ports are available. At this time, only the XY routing algorithm is implemented. However other routing algorithms can be easily implemented due to the high modularity of the router.

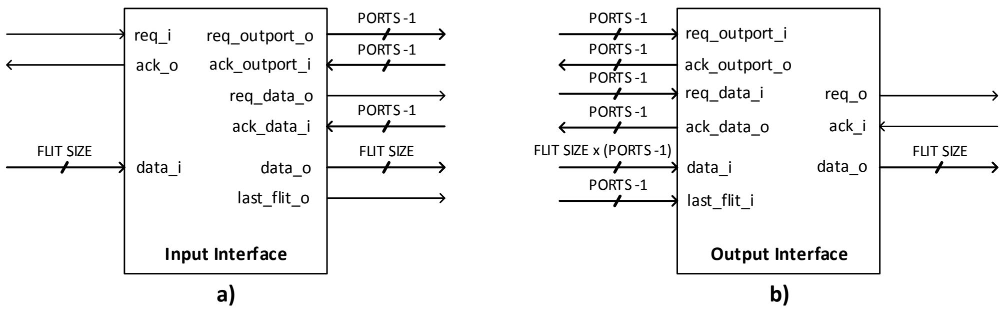


Figure 4.3 – Top view of a) Input Interface and b) Output Interface of BaT-Hermes.

The BaT-Hermes communication protocol determines the handshakes between Input and Output Interfaces. It defines that the first flit of each packet must be sent through a *req_outport* request – that is, a request event issued from the II's *req_outport_o* port to the OI's *req_outport_i* port -, while the remaining flits are sent over *req_data* requests – similarly, sent from II's *req_data_o* port to the OI's *req_data_i* port. Routing and arbitration tasks take place during the first handshake (through a *req_outport* request), in a specific II and OI, respectively. In the remaining handshakes, until the last flit is detected, flits flow directly from the II to the OI. The protocol also specifies that the *last_flit_i* signal needs to switch before the last flit is sent. All communication between interfaces is transition-encoded. The signals *req_data_o*, *last_flit_o* and *data_o* of each II are shared across all OIs, which arbitrate incoming requests. BaT-Hermes uses the same packet layout as the original Hermes NoC: a two-flit header, comprising the packet's target address and payload size, and a variable-length payload. The implementation details of each module is described in Section 4.2.

The key difference between the architectures of BaT-Hermes and the NoC proposed in [GHI13] is the router's control unit. In the former, the detection of head and tail flits is based on the payload size, contained in the packet header. The latter uses a 2-bit flit-type field, embedded in each flit, to identify head and tail. The approach taken on the latter greatly simplifies the control units when compared to the Finite State Machines (FSM) required by the former to count the number of flits sent, but reduces the amount of data carried by each flit, since two bits are always employed as control. Additionally, all communications between IIs and OIs in BaT-Hermes is performed with transition-signaling handshakes, while the router proposed in [GHI13] employs level-signaling handshakes to start and finish the transmission of packets – which increases the circuit's switching activity.

4.1.3 Timing Constraints

Bundled-data circuits rely on carefully designed delay-lines to ensure that handshake events take place only when data is valid. Due to the lack of EDA tools supporting such circuits, the timing relationships between wires and modules must be manually extracted. During the design phase of BaT-Hermes, the timing constraints of each circuit were identified and documented using the following technique: *i)* for each register, identify all data signal paths connected to the data input pin; *ii)* identify the control signal path of the register; *iii)* create a delay line at the control signal in order to guarantee that the path of step 1 is faster than the one of step 2. The list of timing constraints is available in Appendix B.

4.2 Architecture

This Section details the architecture of each component of BaT-Hermes. The techniques described in Section 4.3.1 were employed to obtain the waveforms used to illustrate the operation of the circuits.

4.2.1 Transition Merger and Phase Matcher

Transition-signaling handshake protocols relate changes in the logic value of control signals to actions. Therefore, it is imperative that the request and acknowledge signals remain stable until their related actions have taken place. Linear transition-signaling pipelines are designed to behave this way, and can be easily modified to support non-linear concurrent stages, as discussed in Section 2.2.5. Furthermore, a pipeline can fork to or join from stages that work in a mutually exclusive manner, that is, only one parallel stage will perform a handshake at a time, while all control signals of the remaining stages are kept stable. This construct, exemplified in Figure 4.4, is widely used in BaT-Hermes to decouple routing handshakes from data transferring handshakes, as the former takes longer to complete due to routing and arbitration overheads. Additional hardware is needed in order to support mutually exclusive joins and forks in transition-signaling pipelines.

The approach taken in BaT-Hermes design to implement forks and joins makes use of two circuits: the transition merger and the phase matcher. The former is used to merge independent signals of the same type to a single wire while keeping the number of transitions at the output consistent – that is, any transition in any of the inputs results in a transition at the output. Note that the inputs do not need to be synchronized and are agnostic of the logic level of the output. The phase-matcher is employed to keep the input control signals of deactivated stages stable while shared control signals switch to perform handshakes with the active stage. As Figure 4.4(a)-(b) show, a pipeline fork requires the use of a transition merger on the acknowledge signals and phase matchers at each parallel stage request signal input; on pipeline joins, as showed in Figure 4.4(c)-(d), phase matchers are used at each parallel stage acknowledge signal input, and a transition merger on the request signals.

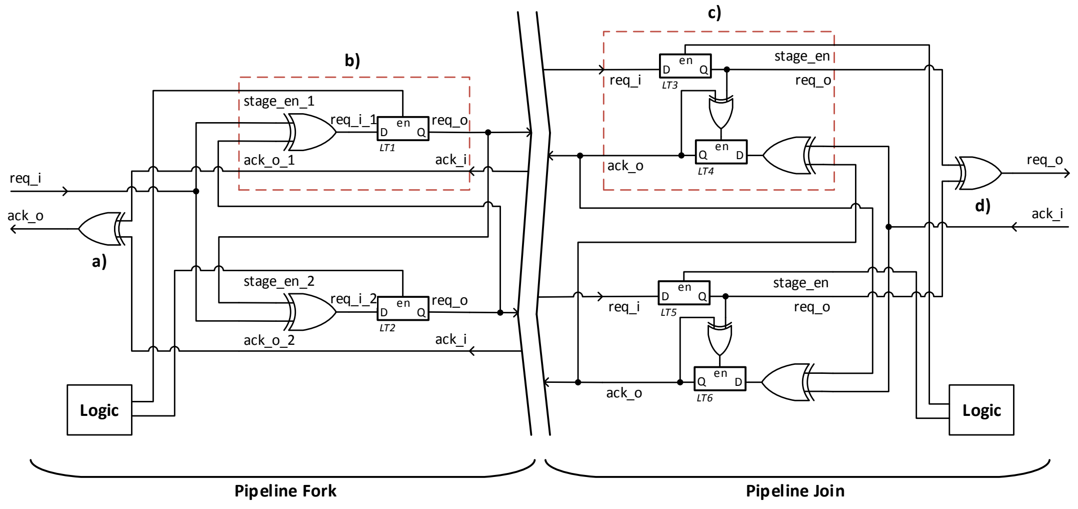


Figure 4.4 – Example of a mutually exclusive pipeline fork with a) transition merging and b) phase-matching circuits and pipeline join with c) phase-matching and d) transition merging circuits.

Both the transition merger and the phase matcher rely on the usage of *n*-input XOR gates, where *n* is the number of parallel stages in the pipeline fork or join. For instance, a transition merger circuit operating with the request signals, as Figure 4.4(d) shows, takes all output request signals from the parallel stages and generates one signal that switch every time one of its inputs switch, given the logic behavior of the XOR. The waveforms on Figure 4.5(a) show the operation of this circuit. As it can be seen, when *ack_o_1* switch to '1', the output *ack_o* also go to '1'. Next, when *ack_o_2* switch to '1', *ack_o* switch back to '0'. Finally, a new transition in *ack_o_1*, to '0', makes *ack_o* to switch, back to '1'. The phase matcher is used to mask transitions directed to the active stages, keeping the logic levels of disabled stages stable. Its operation is similar to the transition merger circuit, but for each handshake cycle its output transitions twice – canceling out the transition that originated the handshake. A latch is used to prevent these two transitions from propagating to the disabled stage. The XOR gate takes as input the shared control signal, *req_i* in the example of Figure 4.4(b), and one control signal from each of the other stages. Even though either control signal (req or ack) can be used, as both will eventually transition during the handshake cycle, it is more efficient to use the signal that switches first, *req_o* in this example, in order to propagate the transition to the other stages as soon as possible. Figure 4.5(b) shows the operation of the circuit depicted in Figure 4.4(b): *1)* a request action on *req_i* generates a transition at the output of every XOR gate connected to it; *2)* the latch on the disable stage (LT2) blocks the propagation of the transition, while the enabled stage (LT1) propagates it, generating a request event on that stage; *3)* the signal flows from the latch LT1 and arrives at the XOR gate at the disabled stage, switching it for the second time and restoring the previous logic level.

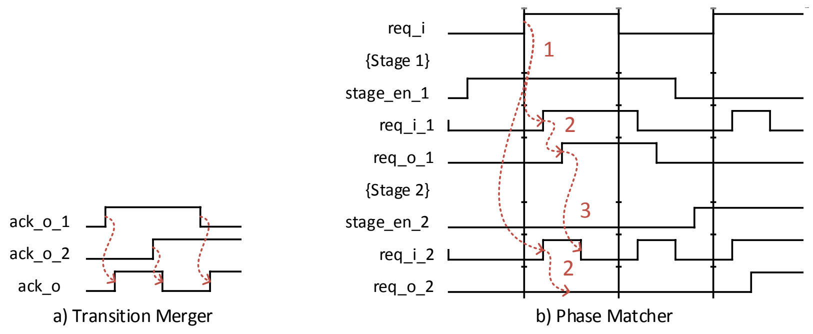


Figure 4.5 – Waveform showing the operation of the a) transition merger and the b) phase matcher circuit.

Although only one request will be received at any given time, as the stages work in a mutually exclusive manner, the pipeline join can be improved to support multiple simultaneous requests being made at the parallel stages. This can be done by the addition of the latch LT3, in Figure 4.4(c), and control logic to choose which stage will be enabled. If it is guaranteed by design that only one request will be made at a time, and during that handshake cycle the other stages will not attempt to start a handshake, the latch LT3 and the control logic can be removed. Also, in pipeline joins, the control of the phase matcher latch (LT4) is performed with an XOR gate. The latch is enabled when a request arrives and automatically disables itself when the acknowledge signal propagates through it. In pipeline forks this task is handled by the control logic used to activate the stage. The logic block that enables the parallels stages may be based on the data input of the stage or on previous handshakes. These circuits are also used in the NoC router proposed by Ghiribaldi et. al. [GHI13].

4.2.2 Input Interface

The II, shown in Figure 4.6, combines an Input Buffer and a Routing Control unit. The Input Buffer is composed of a parameterizable depth First-In First-Out (FIFO) queue and a Buffer Control, responsible for identifying header flits from the payload. The Routing Control implements an algorithm to select the target OI and signal a request to it. Transition-signaling handshake protocol is used in all communications between these components. This circuit implements the mutually exclusive parallel stages mentioned in Section 4.2.1: when a handshake is made through the *req/ack_outport* signals, the *req/ack_data* signals are not used, and vice versa.

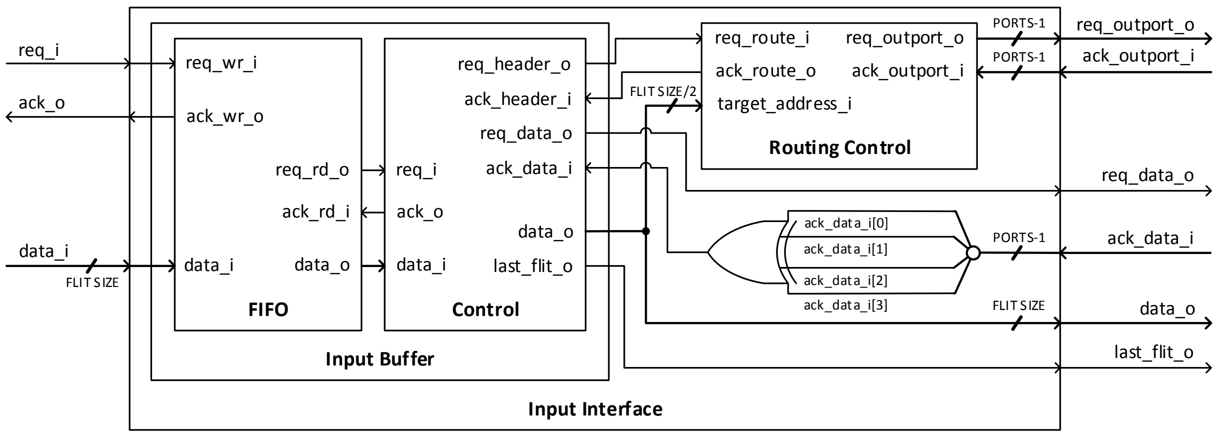


Figure 4.6 – Architecture of the Input Interface.

A pair of dedicated handshake signals (*req_outport_o* and *ack_outport_i*) is connected to each communicating OI - there is no loopback connection between the IIs and OIs on the same port. The Routing Control uses the *req_outport_o* wires to request access to the target OI. An *ack_outport_i* acknowledge means that the target interface has received the flit on *data_o* (target address flit), and is ready to receive other flits from the packet. Note that, albeit handshake cycles on these signals take a long time to complete due to routing and arbitration overheads, they are performed a single time for each packet, only in the first flit for directing the packet to the correct OI. Subsequent flits flow directly from the Input Buffer Control to the selected OI.

The Input Buffer Control uses the *req_data_o* signal, shared with all communicating OIs, to send the remaining flits. Each communicating OI has a dedicated *ack_data_i* wire used to acknowledge data requests destined to it. The shared request signal helps reduce the number of wires and control logic between IIs and OIs. Note that the OI was designed to support out-of-phase *req_data_o* and *ack_data_i signals*, in order to support the shared *req_data* wire. Hence, a transition merger circuit is used at the II to prevent out-of-phase signals in the Input Buffer Control. A transition on the *last_flit_o* signal indicates that the next data request will deliver the last flit of the packet – freeing the OI to accept requests from another IIs after the last flit is received.

Figure 4.7 shows the operation of the II: *1)* data received in the II is buffered in the FIFO and delivered to the Input Buffer Control; *2)* the Input Buffer Control receives data from the FIFO, detects that it is a target address flit, and sends a request to the Routing Control; *3)* the Routing Control choses the target output port based on the routing algorithm and requests its use; *4)* the OI of the selected port acknowledges the request, indicating that it is ready to receive the packet; *5)* the other flits are transmitted by the Input Buffer Control using the shared *req_data_o* signal; *6)* only the OI requested on step 3 acknowledges the request made on step 5; *7)* before the last flit of the packet is sent, the Input Buffer Control notifies the OI switching the *last_flit_o* signal. The FIFO works in parallel with the Input Buffer Control, buffering the flits sent to the input interface while the handshakes take place.

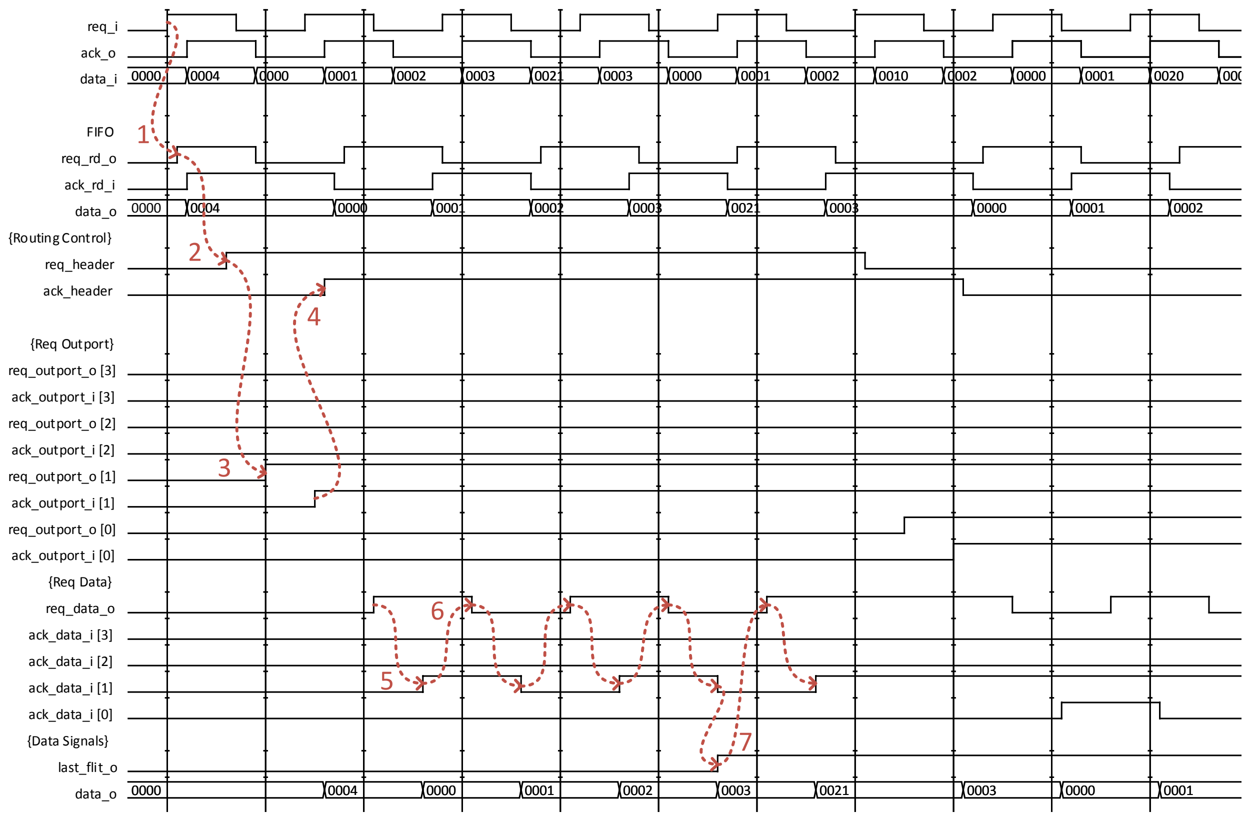


Figure 4.7 – Waveform showing the operation of the Input Interface on the Local port (4) of a BaT-Hermes router with address x11.

4.2.3 FIFO

Buffering is a technique commonly used in NoC design to help reducing performance degradation caused by a possibly blocked flit [MOR04]. The buffer used in BaT-Hermes is a transition-signaling bundled-data First-In First-Out (FIFO) circular queue proposed by Ghiribaldi et al. in [GHI13]. Figure 4.8 details the three building blocks of the FIFO: (a) write controller (*wr_ctrl*); (b) read controller (*rd_ctrl*); (c) 1-hot ring counter (*read/write_counter*). Write and read controllers work in pairs, as they access the same data register. The *wr_ctrl* circuit employs a phase matcher controlled by an XNOR gate and an enable signal (*en_i*) – very similar to the Mousetrap stage, introduced in Section 2.2.5. The signal *reg_en_o* controls the active-high latch that stores data. *full_o* is a request signal employed to inform the pairing *rd_ctrl* that new data is available to be read. *empty_o* is an acknowledge signal indicating that the data has been read. New data can only be written when the *wr_ctrl* is active (i.e. *en_i* is high) and the currently stored data has been read – which can be detected by the XNOR of *full_o* and *empty_i*, resembling a Mousetrap stage. The *rd_ctrl* works as the phase matcher circuit when employed in pipeline joins, as explained in Section 4.2.1. The ring counter is controlled by the XNOR of a request and an acknowledge signal (e.g. *req_i* and *ack_i*) – making it increment at the end of each handshake cycle, when *ack_i* transitions.

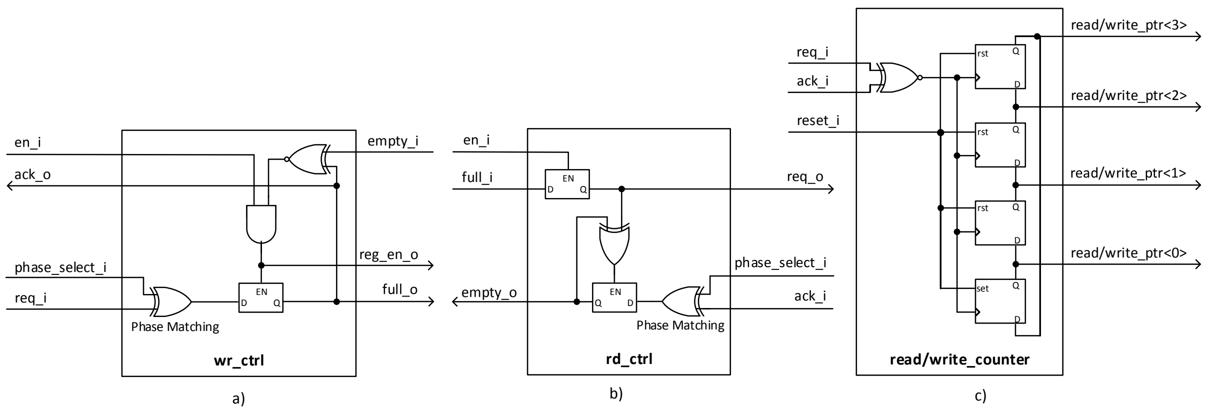


Figure 4.8 – Building blocks of the FIFO proposed in [GHI13]: a) write controller; b) read controller; c) 1-hot ring counter.

Figure 4.9 shows the FIFO's architecture. The pair of signals *req_wr_i* and *ack_wr_o* and *req_rd_o* and *ack_rd_i* are used to write to and read from the buffer, respectively. Note that a transition merger is used join the *ack_o* signals from the multiple *wr_ctrl* into a single *ack_wr_o* signal. Similarly, the *req_o* signal from the *rd_ctrl* employ the same circuit to generate *req_rd_o*. The enable signal of each *wr_ctrl* and *rd_ctrl* is generated by the *write_counter* and *read_counter*, respectively. The latter is also used as the mux selector for choosing the data output signals.

Since the order of activation of each controller is known, it is possible to reduce the number of signals used in the phase matcher to two: the shared control signal (*req_i* or *ack_i*) and a *phase_select_i* signal that is a function of the FIFO's depth and the controller's index.

- For even-depth FIFOs:
  - The *phase_select_i* of even-index controllers is the next *full_i*/*empty_i* signal ($\mathrm{phase\_select\_i[i]} \leftarrow \mathrm{full/empty[i+1]},\ \mathrm{when\ i\ mod\ 2\ =\ 0}$);
  - The *phase_select_i* of odd-index controllers is the previous *full_i/empty_i* signal ($phase\_select\_i[i] \leftarrow full/empty[i-1],\ when\ i\ mod\ 2\ \neq\ 0$).
- For odd-depth FIFOs:
  - The *phase_select_i* of even-index controllers is 0, which means that the *req_i/ack_i* signal can be connected directly to the latch;
  - The *phase_select_i* of odd-index controllers is 1, which means that the XOR gate can be replaced by an inverter.

The waveform presented in Figure 4.10 illustrates the operation of a 2-place FIFO: *1)* a request to write in the fifo is made (*req_wr_i* switches) and directed to the active write controller; *2)* the data latch controlled by the active *wr_ctrl* is disabled, storing the data, while both *full_o* and *ack_o* signals switch; *3) ack_o* propagates, switching *ack_wr_o* and incrementing the *write_counter*; *4)* simultaneously with step 3, *full_o* propagates to *rd_ctrl* switching *req_rd_o*, indicating that there is data in the queue; *5)* once this data is read, *ack_rd_i* switches, incrementing read_counter and freeing the data register to receive new data. These steps repeat for each position of the queue.

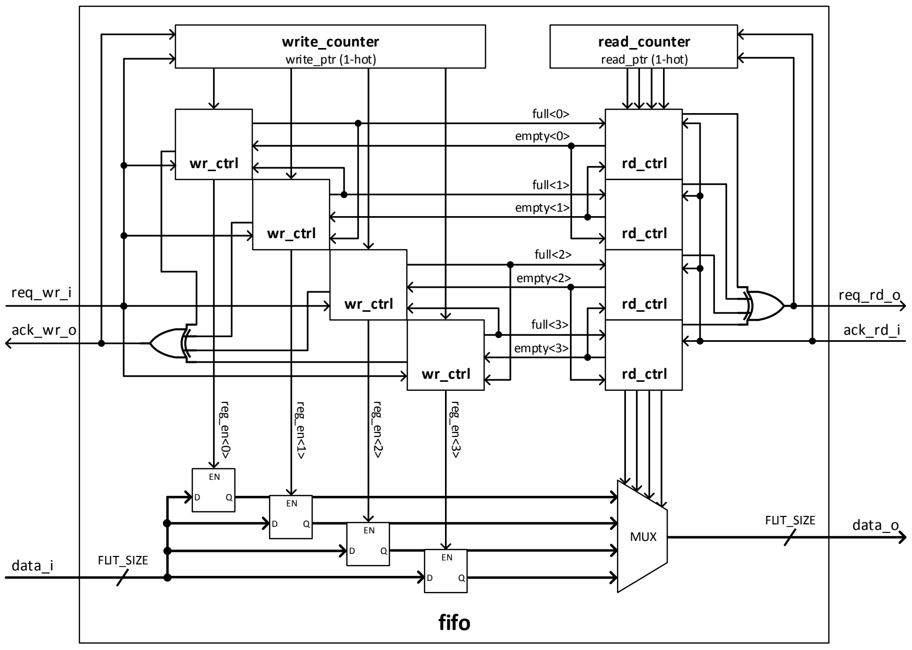


Figure 4.9 – Architecture of the transition-signaling bundled-data FIFO proposed in [GHI13].

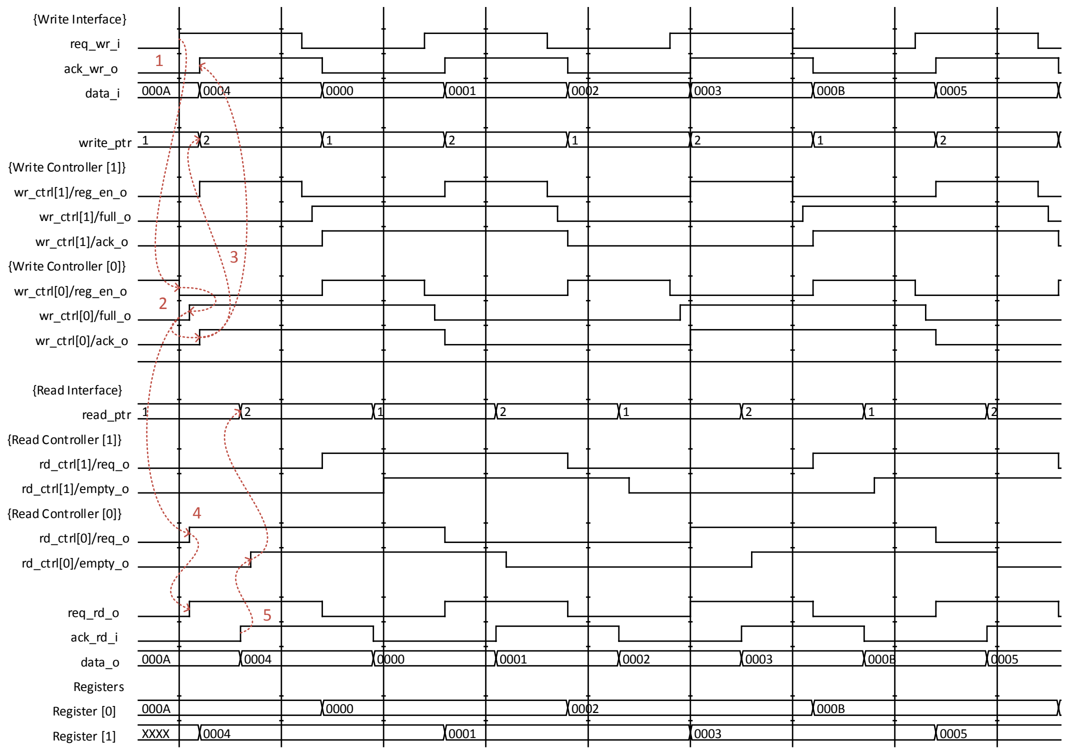


Figure 4.10 – Waveform showing the operation of a 2-place FIFO.

4.2.4 Input Buffer Control

The Input Buffer Control implements the BaT-Hermes communication protocol and is responsible for interacting with the OI. The protocol defines that the first flit of each packet must to be sent through a *req_outport_o* request, while the remaining flits are sent over *req_data_o* requests. It is also stated that the signal *last_flit_o* must switch its logic value before the request sending the last flit of the packet is made. The Input Buffer Control circuit, detailed in Figure 4.11, implements a Finite State Machine capable of fulfilling the requirements of the protocol. A Mousetrap stage (*LT1* and *LT2*) was inserted to reduce the average latency of the router. It allows the FIFO to fetch new data while the Input Buffer Control is busy. All latches and flip-flops of the circuit are reset to zero, except *FF2* and *FF5*.

The FSM circuit can be split in two blocks: a counter and the control logic for selecting which signal will be used to perform the handshake (*req_outport_o* or *req_data_o*), depending if the current flit is the first or the remaining flits of the packet. Flip-flops *FF1*, *FF2*, and *FF3* compose the counter circuit. The signal *flit_counter*, stored in *FF1* denotes the number of flits that remain to be sent. Its value is initialized with the payload size, read from the second flit, and decremented as each flit is sent. *last_flit_lvl* is a level-encoded signal stored in *FF2* that indicates when the handshake of the last flit is occurring. The output of *FF3*, called *size_flit*, controls the mux that selects which value will be stored in FF1: the flit in the *data_i* bus, or the current value of *flit_counter* decremented by 1. The data stored in these three registers are updated every time the Mousetrap stage accepts a request – therefore, the sampled values correspond to what was taking place on the previous handshake. For example, when the *last_flit_lvl* register (*FF2*) detects that the value of *flit_counter* is 1, it means that when the previous handshake took place there was still one flit remaining to be sent; consequently, the current flit is the last flit. This allows the computation of the next values to start as soon as the request is made, which means that the circuit doesn't need to be delayed if the computation time is smaller than the handshake round-trip time – that is the sum of: propagation delay of request signal, propagation delay of acknowledge signal, and the minimum amount of time it takes for the OI to issue an acknowledge.

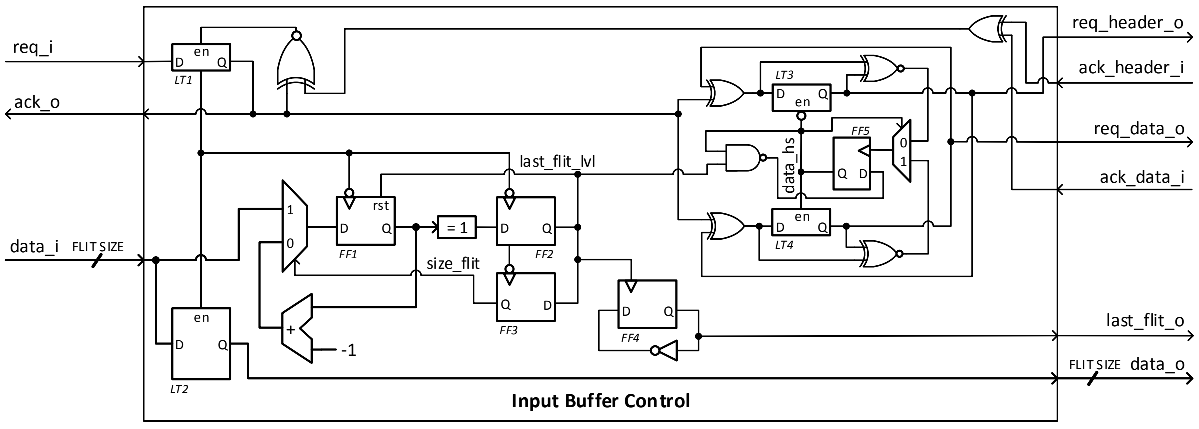


Figure 4.11 – Architecture of the Input Buffer Control.

The second block of the FSM, composed by *LT3*, *LT4*, *FF5* and the adjoining logic gates implements a mutually exclusive pipeline fork to control which output signal will perform the next request: *req_header_o* or *req_data_o*. A transition merger connects *ack_header_i* and *ack_data_i* to the Mousetrap stage. The Phase Matcher circuits are implemented with latches LT3 and LT4, plus connecting XOR gates. The signal *data_hs*, generated by *FF5*, controls the phase matcher's enable signal: when *data_hs* is asserted, the request is signaled by *req_data_o*; when deasserted, *req_header_o* performs the request. The value stored in *FF5* is updated as soon as the request signal propagates through the active phase matcher latch – the XNOR gate performs this detection: its output switches to low when the inputs have a different logic value. The mux prevents temporary transitions on the deactivated stage from reaching *FF5*. The new value of *FF5*, calculated by the NAND gate, is determined as follows: *i)* If *data_hs* is deasserted (*req_header_o* handshake), the next *data_hs* will be asserted (*data_hs* handshake); *ii)* If *data_hs* is asserted and *last_flit_lvl* is also asserted, the next *data_hs* will be deasserted; *iii)* otherwise, *data_hs* is kept asserted.

In order to reduce latency, the Input Buffer Control was designed to avoid stricting timing constraints on the request signal path. Even though the *last_flit_lvl* signal is shared between both blocks of the FSM, its propagation delay only affects *FF5*, as the signal at the input data pin needs to arrive before the one at the clock pin. As long as this delay is smaller than the round-trip time of the request signal, it does not affect the performance of the circuit.

Figure 4.12 shows the step-by-step operation of the Input Buffer Control circuit: *1) req_i* request is made; *2)* as soon as the data is stored in the Mousetrap stage, an acknowledge is issued by switching *ack_o*; *3)* in parallel with step 2, the flip-flops *FF1*, *FF2* and *FF3* update the stored values – note the *size_flit* signal rising to select *data_i* as the input of *FF1* for the next handshake. Also in parallel, the request propagates to *req_header_o*, and the *data_hs* signal is updated; *4)* when the acknowledge signal *ack_header_i* arrives, the payload size is stored in *flit_counter* and *size_flit* is disabled – changing the mux to decrement the counter. In parallel, the request is made using the signal *req_data_o*; *5)* the following flits are sent through *req_data_o*, while the *flit_counter* is decremented; *6)* when *flit_counter* reaches 1, it asserts the *last_flit_lvl* signal, indicating that the next flit is the last of the packet. As a consequence, the positive edge-triggered flip-flop *FF4* updates its output with the inverse of the current value, causing *last_flit_o* to switch. As soon as the *req_data_o* propagates through *LT4*, *data_hs* is deasserted, restoring the initial conditions of the circuit.

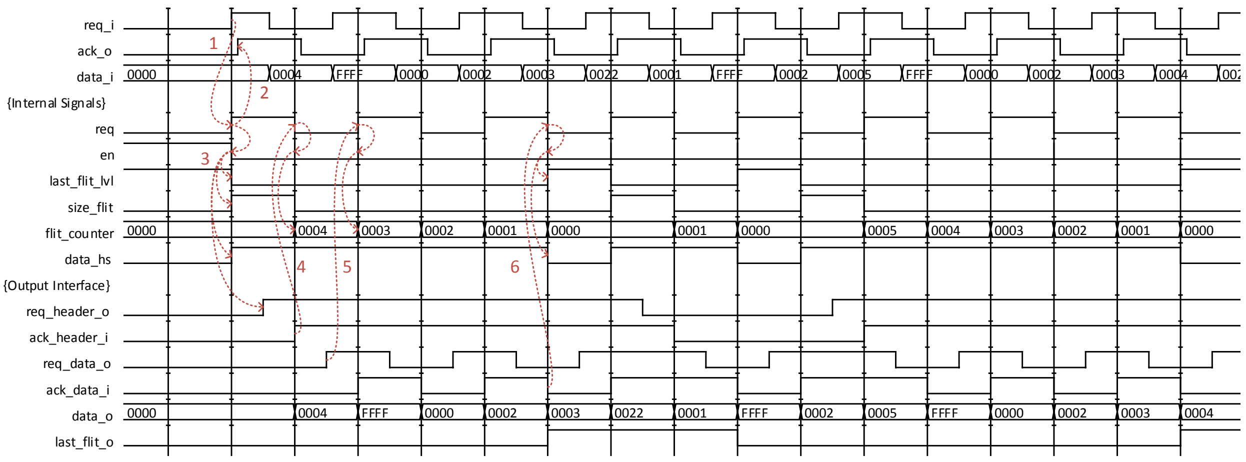


Figure 4.12 – Waveform showing the Input Buffer Control Operation.

The simulation technique used for functional validation doesn't simulate gate delays. For this reason, the signals seem to switch at the same time – for example, the *en* signal appears as a spike instead of a pulse long enough to fulfill the timing restrictions of the connected gates. Details about how the behavioral simulations were performed can be found in Section 4.3.1.

4.2.5 Routing Control

The Routing Control is responsible for requesting use of OIs. The circuit, as shown in Figure 4.13, is composed of a *Routing Unit* and a mutually exclusive fork with a parallel stage for each communicating OI. The routing algorithm is a combinational circuit that compares the value of signal *target_address_i* with the hardwired router address and port. Once the computation is ready, the target OI phase matcher is enabled, generating a request through the respective *req_outport_o*. An XOR gate is used to guarantee that the *Routing*

Unit is deactivated when routing is not taking place. The VHDL implementation of this module, through the use of “if generate” statements, can optimize the circuit by removing logic related to inexistent OI. This situation may happen, for example, in routers placed at the corner of a mash network.

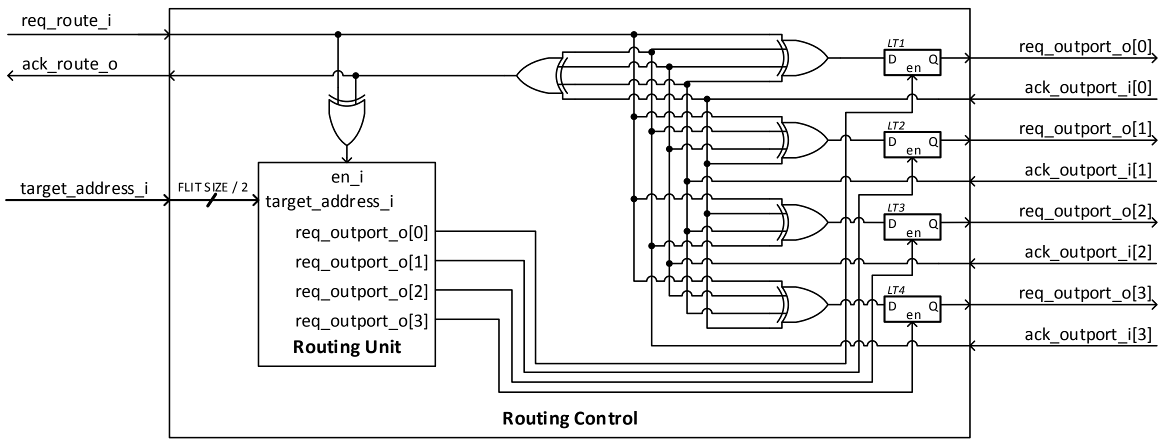


Figure 4.13 – Routing Control Architecture.

Figure 4.14 shows the operation of this circuit: *1)* the arrival of a *req_route_i* activates the *Routing Unit*. A request on the correct *req_outport_o* is made as soon as routing algorithm completes its computation; *2)* the acknowledge sent by the OI through the respective *ack_out_port_i* signal is forwarded to *ack_route_o*.

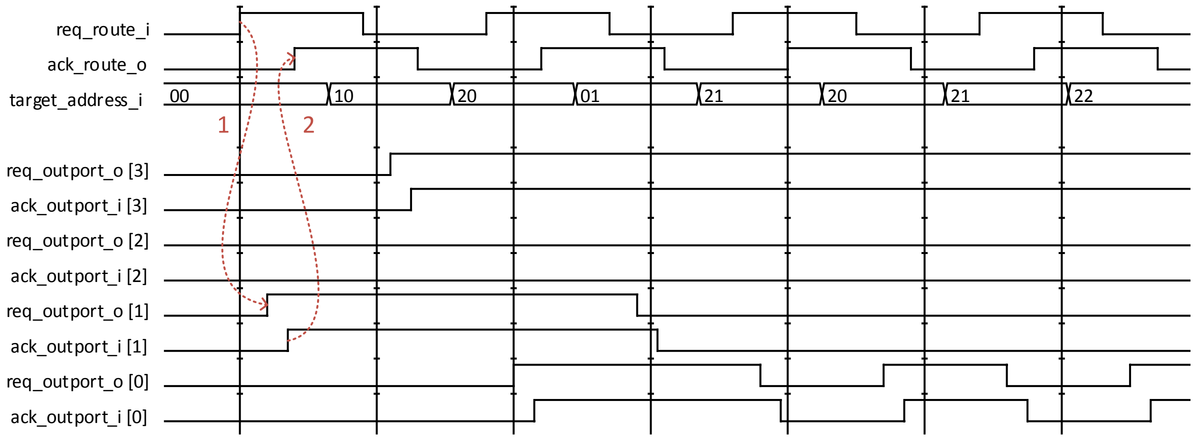


Figure 4.14 – Waveform showing the operation of the Routing Control at a local port on a BaT-Hermes router of address x11.

4.2.6 Output Interface

The OI of BaT-Hermes is responsible for arbitration tasks. Figure 4.15 shows the architecture of this circuit, which combines a set of *Outport Control* blocks with Phase Matchers at the *ack_i* inputs, an *Arbiter*, a multiplexor, and a Transition Merger on the request signals. Each *Outport Control* is connected to an II and is capable of handling out-of-phase *req_data_i/ack_data_o* signals events, in order to support the shared *req_data_i* signal. The *Arbiter* mediates simultaneous requests, guaranteeing that only one *Outport Control* can access the output channel at any time. The grant signal from the Arbiter (*arbiter_grant_i*) is also used to control the data multiplexor. When a *req_outport_i* request arrives, the *Outport Control* issues a level-encoded request signal to the *Arbiter*. Once this request is granted, and as long as the level-encoded request signal remains asserted, the *Outport Control* circuit can interact with the output channel through *req_o*, *ack_i*, and *data_o* signals.

Unlike the handshakes between the other components of BaT-Hermes, the *Arbiter* communicates using a level-signaling protocol. This was mandatory, since no transition-sensitive mutex is available in the cell library used.

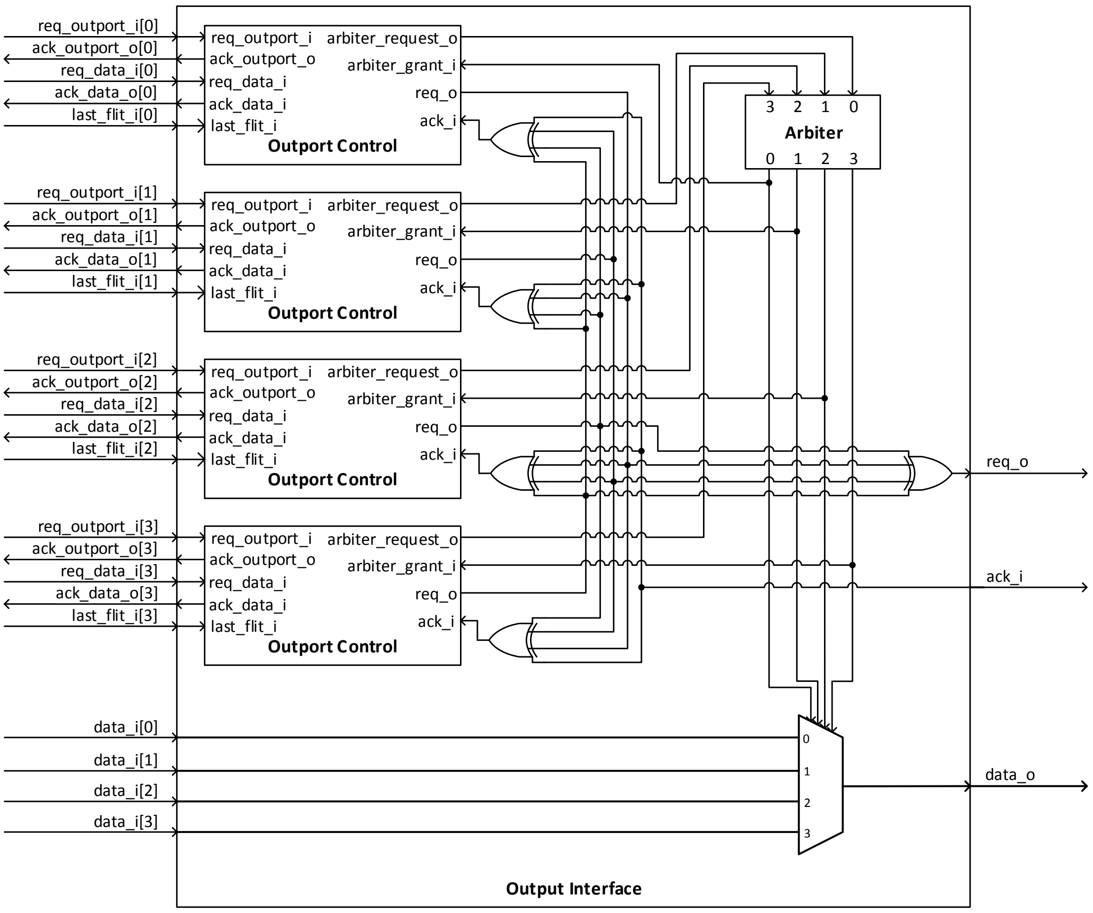


Figure 4.15 – Architecture of the Output Interface

Figure 4.16 illustrates the operation of the circuit: *1)* two simultaneous *req_outport_i* requests are made (0 and 1); *2)* the *Arbiter* grants access to the output channel to one of the *Outport Control* blocks that issued a request on step 1; *3)* Once the initial handshake is completed, the next flits are sent using the shared *req_data_i* signal; *4)* the *Outport Control* circuit handles the out-of-phase handshake signals gracefully; *5)* the last flit detector, built inside the *Outport Control*, senses the transition on the *last_flit_i* wire and deasserts the arbiter request signal; *6)* the arbiter grants access to the output channel to another *Outport Control* block. The transition of *req_data_i[0]* before the first handshake is generated by the test bench to simulate an out-of-phase *req_data_i* signal.

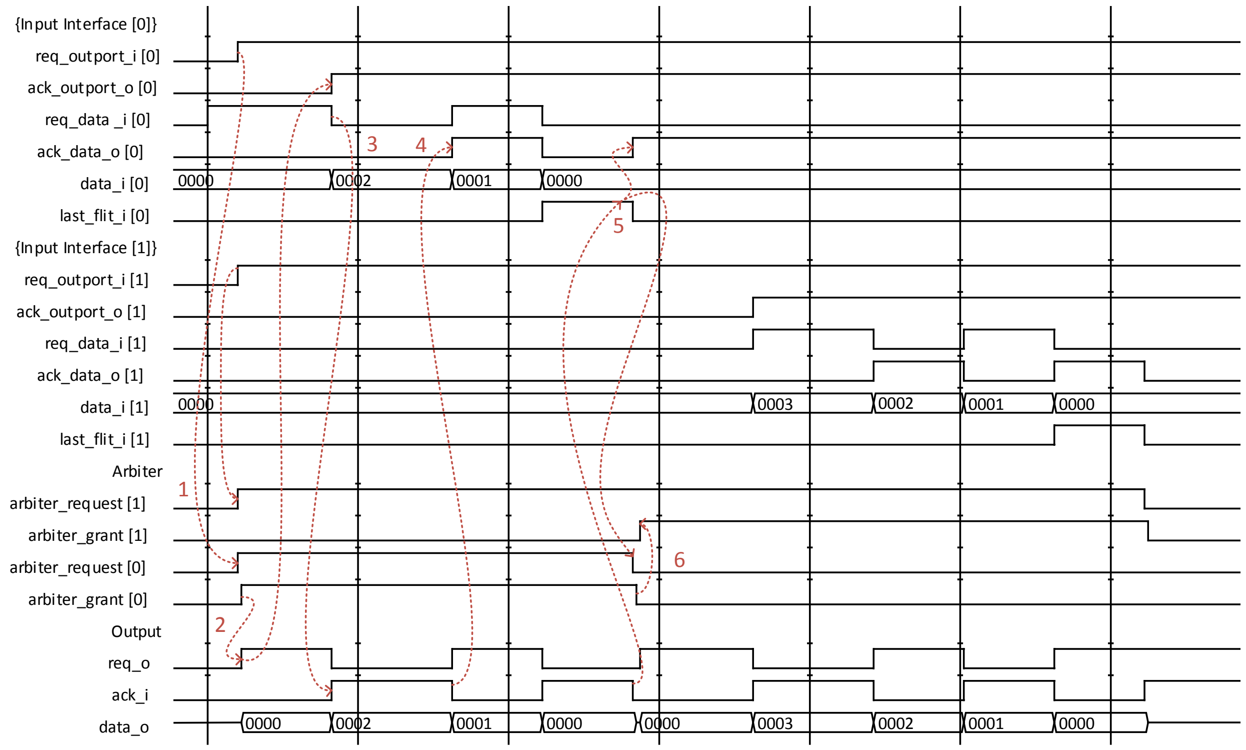


Figure 4.16 – Waveform showing the Output Interface communicating with two Input Interfaces.

4.2.7 Outport Control

The Outport Control implements the BaT-Hermes communication protocol and is responsible for controlling the interactions between II, *Arbiter* and output channel. The circuit, detailed in Figure 4.17, can be split into six blocks: *i)* last flit detector, formed by *FF2*, *FF3* and the adjoining XOR gate; *ii)* programmable phase matcher for the signal *req_data_i*, implemented with *FF1*, *LT2* and the connected XOR gates; *iii)* phase matcher circuit to control *ack_outport_o*, composed by *LT1* and the adjoining XOR gates; *iv)* phase matcher to control *ack_data_o*, comprised of *LT3* and the connected XOR gates; *v)* transition merger circuit, connecting the input request signals to *LT5*; *vi)* arbiter handshake control, implemented with *LT4* and the remaining gates.

The interaction between *Arbiter* and *Outport Control* is similar to a level-signaling handshake: first the *Outport Control* asserts the signal *arbiter_request_i* to request use of the output channel. Once the request is granted, the *Arbiter* signals the *Outport Control* by asserting *arbiter_grant_i*. From this moment on, the *Outport Control* has exclusive use of the output channel. This use can be relinquished by deasserting the signal *arbiter_request_o*. A new request can only be made after the deasserted *arbiter_grant_i* propagates to the *Outport Control*. The four-phase handshake prevents a new *req_outport_i* request, received soon after the end of the previous packet, from being sent to the output channel due to a delayed deassertion of *arbiter_grant_i*. The handshake behavior, implemented by the AND gate connected to the set pin of *LT4*, blocks new requests while the signal *arbiter_grant_i* is asserted. The grant signal, generated by the AND gate connecting *arbiter_request_o* and *arbiter_grant_i*, enables the latches *LT1*, *LT2*, and *LT5*, allowing handshake signals to propagate. This signal is disabled as soon as arbiter_request_o is deasserted in order to prevent incoming handshake signals from propagating. An example of this situation is given at the end of this Section.

The programmable phase matcher circuit is needed in order to support out-of-phase data requests. An XOR gate tests if the signal stored in *LT2* is equal to *req_data_i*. The output of this gate is stored in *FF1* on the low-to-high transition of the *grant* signal – that is, when the arbiter grant is given, through *arbiter_grant_i*. The stored value is fed to another XOR gate, which acts as a programmable inverter of *req_data_i*. When the signals are out-of-phase, logic '1' is stored in *FF1*, inverting *req_data_i* in order to match the value stored in *LT2*.

Similar to *FF1*, *FF3* stores the value of *last_flit_i* when the arbiter grant is given. The last flit of a packet can be detected when the *last_flit_i* becomes different from the value stored in *FF3*. This test is performed by the negative edge-sensitive flip-flop *FF2* at the end of each data handshake cycle. Logic '1' is stored in *FF2* when the arrival of the last flit is identified, which causes *LT4* to reset – deasserting *arbiter_request_o*. *FF2* is reset when the *grant* signal goes to zero.

BaT-Hermes has timing constraints to guarantee that the signals *req_data_i* and *last_flit_i* are stable before a *req_outport_i* request can be issued.

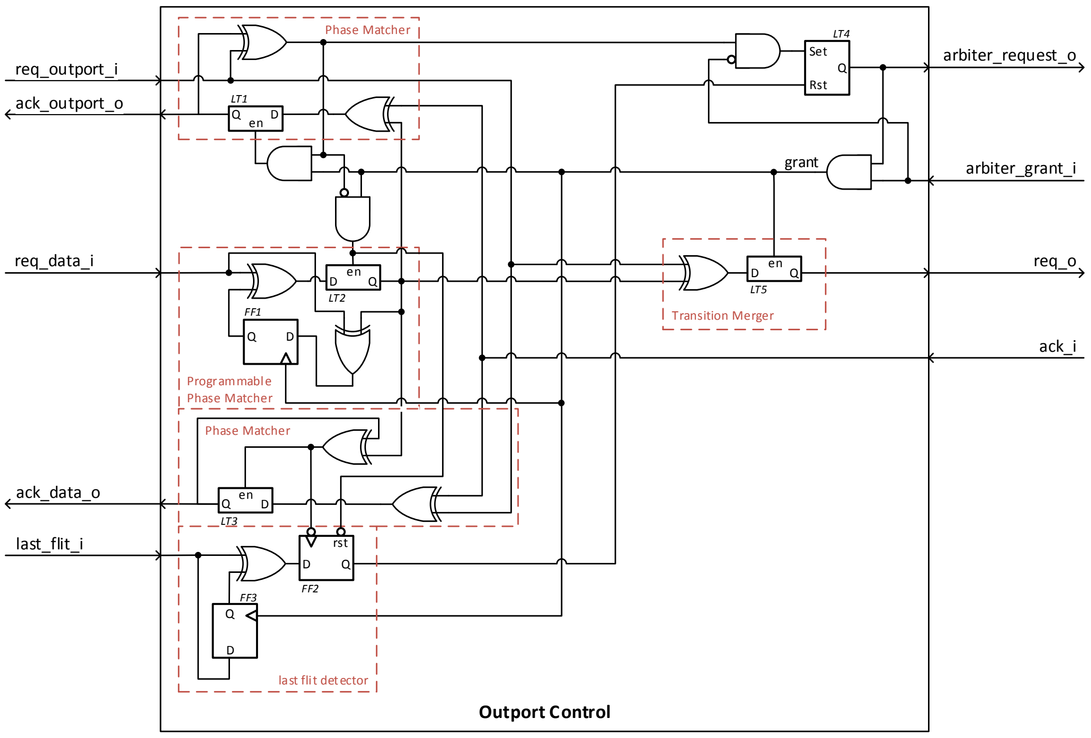


Figure 4.17 – Outport Control Architecture, with logic blocks highlighted.

Figure 4.18 exemplifies the *Outport Control* operation: *1)* the arrival of a *req_outport_i* request causes *LT1* to be set, requesting use of the output channel to the *Arbiter*; *2)* once the *Arbiter* grant is given, the last flit detector and programmable phase matcher are initialized in parallel, while the *req_outport_i* propagates to the output *req_o*; *3)* the *ack_i* signal propagates to *ack_outport_o*, making *LT1* become opaque and *LT2* transparent; *4)* a sequence of *req_data_i* requests and *ack_data_o* acknowledges takes place while the packet is transferred; *5)* at the end of the data handshake cycle the last flit detector notices the transition on *last_flit_i*, which makes the output of *FF3* (*last_flit*) to switch to '1', resetting *LT4*, and, as a consequence, deasserting *arbiter_request_o*. The internal grant signal falls as soon as the signal *arbiter_request_o* becomes a logic '0', resetting *FF3* – this is shown in the waveform as a spike on the *last_flit* signal. *6)* A new *req_outport_i* request arrives, but is blocked until the deasserted arbiter_grant_i signal propagates to the *Outport Control*. If the four-phase handshake with the *Arbiter* was not implemented, the request that was blocked in step 6 would have been propagated to the output channel, as the *arbiter_grant_i* signal was still high when the *req_outport_i* request arrived.

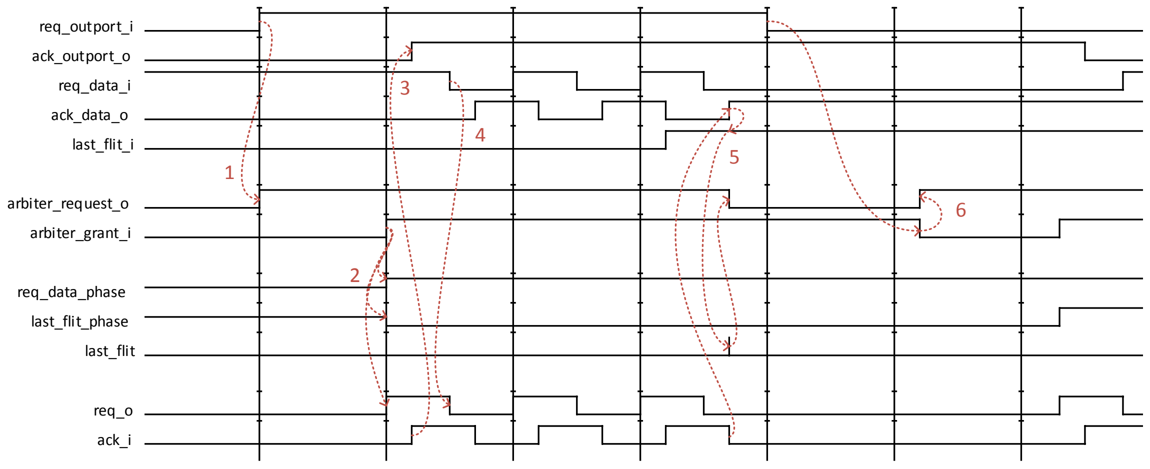


Figure 4.18 – Waveform showing the operation of an Output Control circuit.

4.2.8 Arbiter

The *Arbiter* is responsible for mediating simultaneous output channel usage requests, granting access to only one *Outport Control* at a time. As discussed in Section 2.2.4, arbitration tasks in asynchronous circuits are usually performed with mutual exclusion gates. Bat-Hermes requires a 4-input arbiter, as each OI can communicate with up to 4 IIs. Figure 4.19 shows the 4-input arbiter proposed by Ghiribaldi et. al., and used in BaT-Hermes. Details about its design can be found in [GHI13]. The C-element and mutex gates needed to implement this circuit are available in the ASCEnD Cell Library.

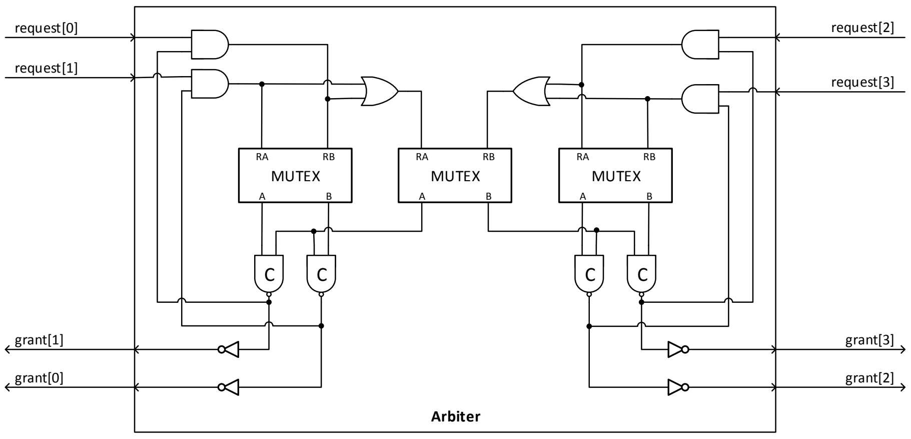


Figure 4.19 – 4-input arbiter design proposed in [GHI13].

Figure 4.20 illustrates the operation of the *Arbiter*: *1)* in the absence of contention the grant is immediately given; *2)* in the presence of contention, similar to the 2-input mutex, only one grant is given at a time and is retained for as long as the request signal is asserted; once the request is deasserted, grant is given to next request. This design provides a certain degree of fairness when choosing the next request to be granted: two requests signals that share the same mutex will not be given consecutive access if a request has been made on the other mutex. This is similar to a round robin policy implemented between the requests on the left and right sides of the arbiter [GHI13]. This behavior can be seen on the waveform below.

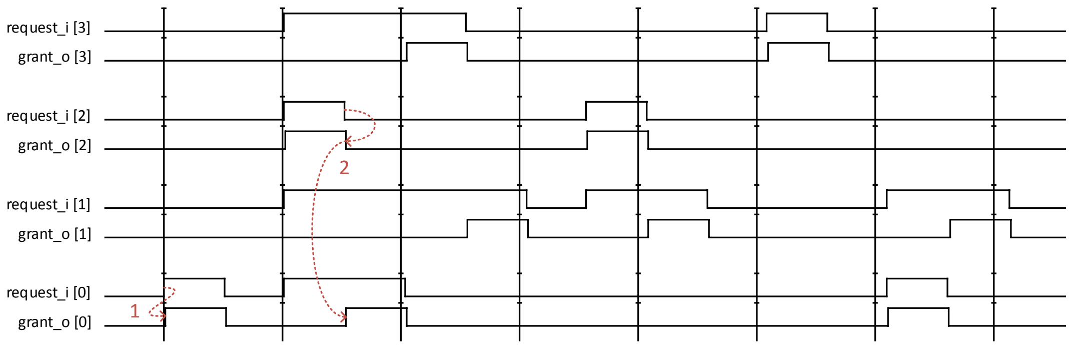


Figure 4.20 – Waveform showing the operation of the Arbiter.

4.3 Functional Validation

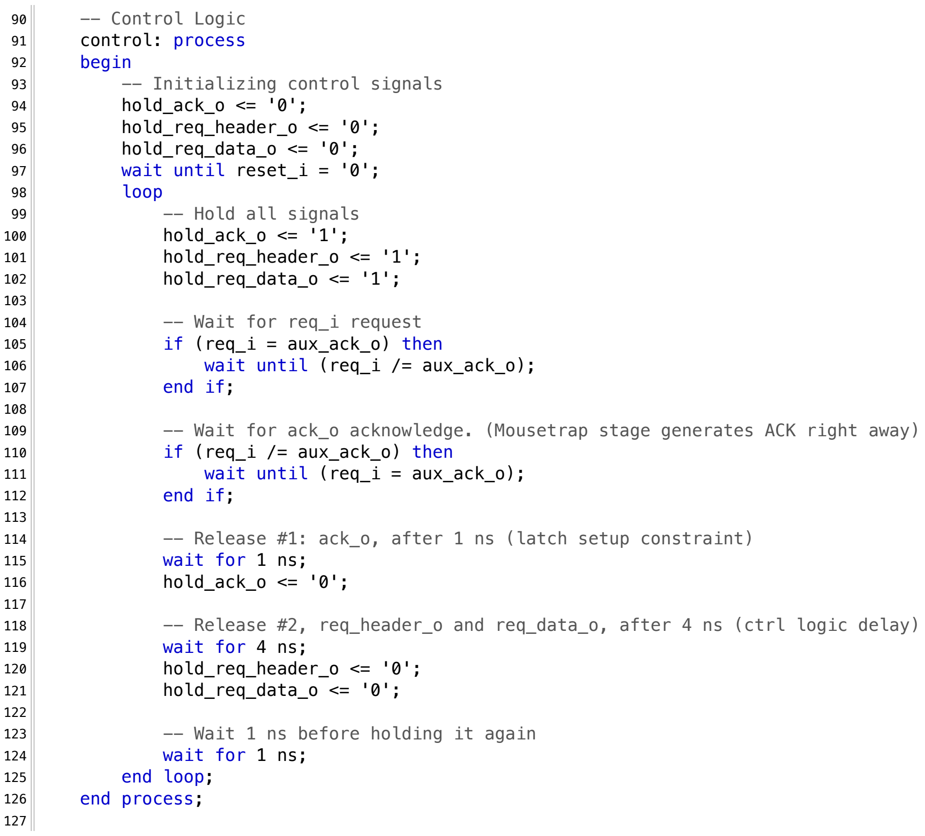
The function validation of BaT-Hermes was performed through behavioral simulation using Mentor Graphics' ModelSim. This section describes the techniques used to enable the simulation of an asynchronous circuit using a commercial tool designed for synchronous systems. Simulation results are also presented.

4.3.1 Simulation Wrappers

The behavioral simulation of asynchronous circuits using tools designed for synchronous system simulation can be problematic, since there is no clock signal giving a timing reference to the circuit. Nevertheless, there are several ways to overcome this problem. One simple technique is to use the VHDL statement "after" on signal assignments to simulate propagation delays. This method, however, requires a careful planning of the amount of delay applied to each signal in order to fulfill all timing constrains of the circuit. Another approach is to treat each module as an independent self-contained block with internal delay long enough to fulfill its timing requirements. For the simulation of BaT-Hermes, this technique was implemented in the form of circuit wrappers.

A wrapper is a VHDL entity with interface identical to the circuit under test (CUT). It employs external latches between the control ports of the instantiated CUT and the ports of the wrapper to control signal propagation. The order in which latches are enabled and disabled depends on the functionality of the block, and must emulate the circuit's desired sequence of operations – during synthesis, this is achieved by the use of relative timing constraints. Only modules capable of generating requests and acknowledges need to be wrapped – for example, a Mousetrap stage, that issues an acknowledge when a request is received. However, other blocks can be wrapped to simulate propagation delay. Extra latches may be needed in order to fulfill dependencies related to signals coming from other blocks.

Wrappers for the following circuits were created to simulate BaT-Hermes: *Input Buffer Control*, *FIFO Write Control*, *FIFO Read Control*, and *Routing Control*. The first two actively perform handshakes, while the remaining were wrapped to simulate the propagation delay. Implementation files for these wrappers are available in Appendix A. Figure 4.21 shows the control logic of the wrapper used to simulate the *Input Buffer Control* circuit. Signals whose name starts with "hold" are used to control the extra latches – by asserting them, the latches become opaque. Signals starting with "aux" are the outputs of the CUT that are connected to the inputs of their respective latches. In this wrapper, latches to hold the following control signals are used: *ack_o*, *req_header_o, and req_data_o*. Initially, after reset, the propagation of all these signals is blocked by making the latches opaque. Next, the wrapper waits for a request signal to arrive and for the respective acknowledge generated by the circuit under test – which, happens immediately, since there is no gate or wire delays. After 1 ns, the acknowledge signals is released, by deasserting *hold_ack_o*. This simulates the time needed by the data latch inside *Input Buffer Control* to store data. After 4 ns, the signals *req_header_o and req_data_o* are released. This delay simulates the propagation time of the logic inside the circuit under test. After that, all latches become opaque again, and the wrapper waits for a new request signal.



```vhdl
-- Control Logic
control: process
begin
    -- Initializing control signals
    hold_ack_o <= '0';
    hold_req_header_o <= '0';
    hold_req_data_o <= '0';
    wait until reset_i = '0';
    loop
        -- Hold all signals
        hold_ack_o <= '1';
        hold_req_header_o <= '1';
        hold_req_data_o <= '1';

        -- Wait for req_i request
        if (req_i = aux_ack_o) then
            wait until (req_i /= aux_ack_o);
        end if;

        -- Wait for ack_o acknowledge. (Mousetrap stage generates ACK right away)
        if (req_i /= aux_ack_o) then
            wait until (req_i = aux_ack_o);
        end if;

        -- Release #1: ack_o, after 1 ns (latch setup constraint)
        wait for 1 ns;
        hold_ack_o <= '0';

        -- Release #2, req_header_o and req_data_o, after 4 ns (ctrl logic delay)
        wait for 4 ns;
        hold_req_header_o <= '0';
        hold_req_data_o <= '0';

        -- Wait 1 ns before holding it again
        wait for 1 ns;
    end loop;
end process;
```

Figure 4.21 – Fragment showing the control logic of the wrapper used to simulate the Input Buffer Control circuit.

4.3.2 Simulation Results

Three test cases were used to perform functional validation of BaT-Hermes. In the first, three packets were sent from the east port to the north and local ports – the first and last packets to the latter. This test shows packets flowing through the router and illustrates the operation of the programmable phase matcher implemented in the *Outport Control*. Figure 4.22 shows the waveforms generated when the test was performed: *1)* the first flit was sent to the local port using the *req_outport* signal, causing the initialization of the programmable phase matcher associated with the *Outport Control* interacting with the east port; *2)* the next flits are sent through the shared req_data signal; *3)* the second packet is sent to the north port; by the end of the transmission, the phase of the shared *req_data* signal will be different than its initial state; *4)* similarly to step 1, the programmable phase matcher is initialized to allow the out-of-phase *req_data* signal interact with the *Outport Control*.

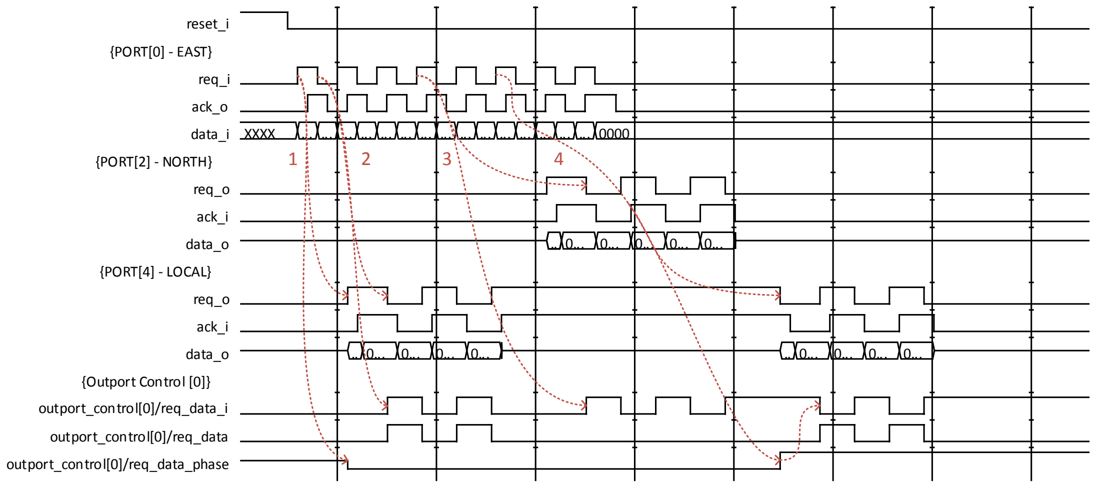


Figure 4.22 – Operation of BaT-Hermes when sending packets from the East port to the North and Local ports.

The second test simulates the router operating at its maximum throughput, which happens when there are five connections simultaneously active. In the situation shown in Figure 4.23(a), the connections happen as follows: *i)* from the east port to the local port; *ii)* from the local port to the north port; *iii)* from the north port to the west port; *iv)* from the west port to the south port; *v)* from the south port to the east port.

The last scenario, shown in Figure 4.23(b), tests the router with presence of contention: all input ports, with the exception of the local port, try to send packets to the local output port. It can be seen that the *Arbiter* circuit works as expected, allowing only one Input Interface access the Output Interface at any given time.

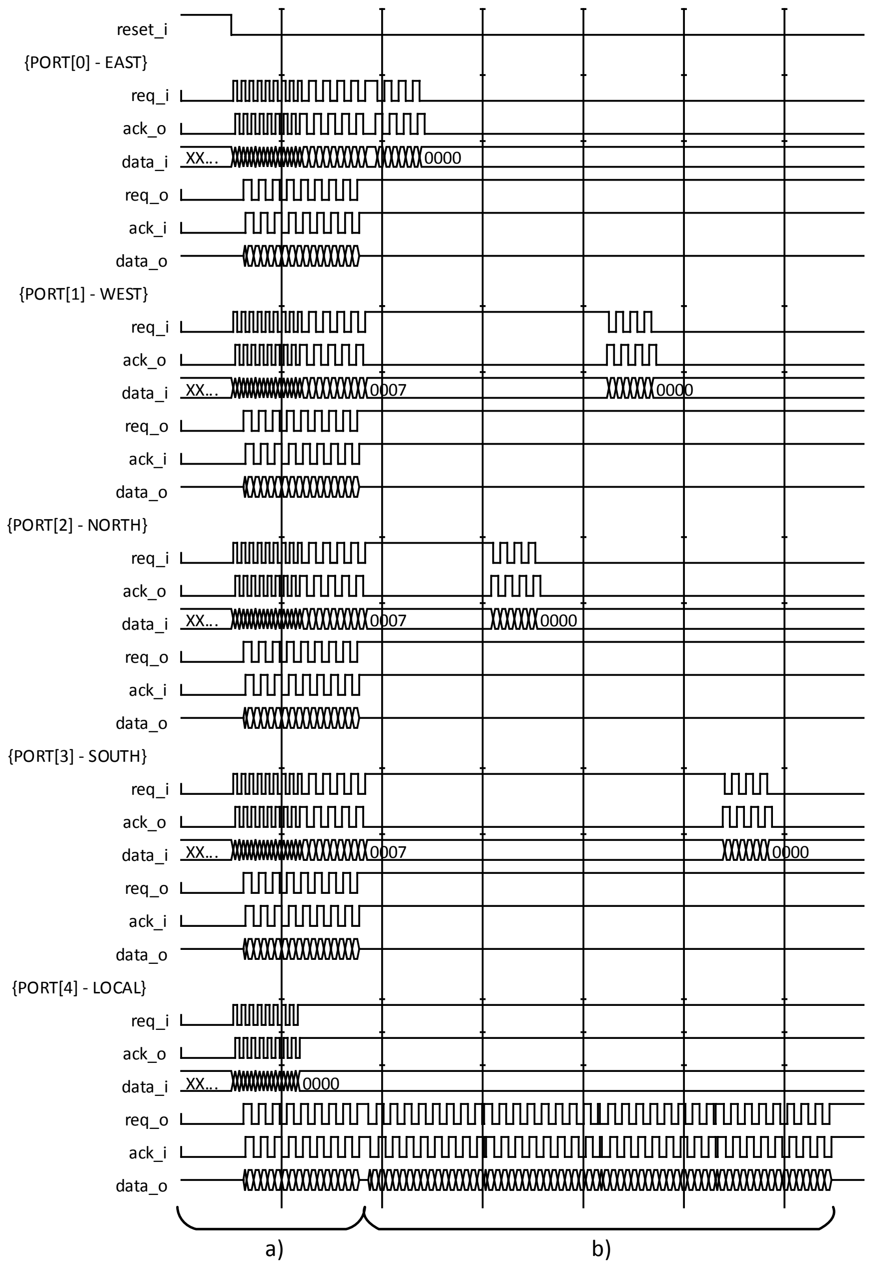


Figure 4.23 – BaT-Hermes a) operating at peak performance scenario, and b) with contention on the output of the local port.

5. IMPLEMENTATION

After functional validation, the Input and Output Interfaces of BaT-Hermes were synthesized to layout level using the STMicroelectronics 65nm CMOS technology. The ASCEnD cell library, designed at GAPH, provided the mutex and C-element cells required by the *Arbiter* circuit. The timing constraints identified during design were applied in the synthesis process to ensure the correct operation of the circuit. This Chapter details how the timing constraints were enforced and describes the synthesis flow used to generate the layouts. It also describes post-synthesis simulation with back-annotated delays performed to validate the circuit's functionality.

5.1 Relative Timing Constraints

Bundled-data circuits fall under the category of self-timed asynchronous circuits, as they rely on carefully designed delay lines to ensure that handshake events take place only when data is valid – that is, the request signal must arrive only after the data signal is stable. Relative timing constraints define signal-arrival order, and can be used to fulfill these requirements. These constraints relate a base path, an enforced path, and a delay line: the enforced-path delay must be greater than the base-path delay; if it is not, a delay line must be inserted in the enforced path to fulfill the requirement. Commercial EDA tools, however, are not adapted to deal with such constraints in a natural manner [GHI13].

5.1.1 Enforcing Relative Timing Constraints on Synopsys Tools

Even though Synopsys tools do not natively support relative timing constraints, an approach for enforcing them is proposed in [GHI13], but no EDA support is made available in that reference. The methodology consists in iteratively extracting delays from base paths, using the *get_timing_path* command, and applying to enforced paths, using the *set_min_delay* command, until all constraints are met. However, several issues, explained below, may arise when setting minimum delay values. In order to avoid such problems, the following guidelines were defined during the development of this work for creating or enforcing constraints:

Gi) A delay-line cannot be inserted in a path that is common to both base and enforced paths of the same constraint;

Gii) Delays cannot be set on paths with forks;

Giii) Given that *d(x,y)* calculates the path delay from point *x* to point *y*, the delay of a path that goes through a delay line should be calculated as: $\mathrm{d(a,b)} = \mathrm{d(a,m)} + \mathrm{d(m,n)} + \mathrm{d(n,b)}$, where a and b are, respectively, the start and endpoints of the path; and m and n are, respectively, the start and endpoints of the delay line;

Giv) A delay-line start and endpoint must be referenced by pin, not by net name.

The FIFO's write control circuit, introduced in Section 4.2.3, can be used to illustrate the issues related to the *set_min_delay* command. Figure 5.1(a) shows this circuit after logic synthesis. The detailed part has one relative timing constraint: signal *phase_select_i* must arrive at pin *D* of *full_reg* (*full_reg/D*) before *req_i*. If the delay is set on the path from *req_i* to *full_reg/D*, it is not possible to infer where the tool will insert the delay line, which can be: *a)* between *U10/Z* and *U9/D* or *U9/Z* and *full_reg/D* the delay is applied to both *req_i* and *phase_select_i*, preventing the constraint from ever being met, and justifying the first guideline (*Gi*); *b)* from the fork of *req_i* to *U9/B* or *U10/B* the delay is not applied to all paths from *req_i* to *full_reg/D*, therefore, more iterations may be needed in order to fix delays of remaining paths, which creates redundant delay lines, and justifies the second guideline (*Gii*); *c)* from *req_i* to its first wire fork, which is the right place to add the delay-line as it affects the whole path of *req_i* without interfering with *phase_select_i*. A timing path, defined by a timing start and endpoint, is used by Synopsys' Static Timing Analysis (STA) tool to calculate the delays across wires and gates. The timing at these points is broken by the STA tool, therefore delays can only be calculated to and from these points – the tool cannot compute the delay across a timing points. When the start and endpoints of a delay line are not timing start and endpoints, they become one, breaking the timing path through them. For this reason, the delay of a path that goes through a delay line must be measured as stated on the third guideline (*Giii*). Also for this reason, nets should not be used as start or endpoints, as it is not possible to infer in which cell the timing path will be broken, justifying the guideline number four (*Giv*).

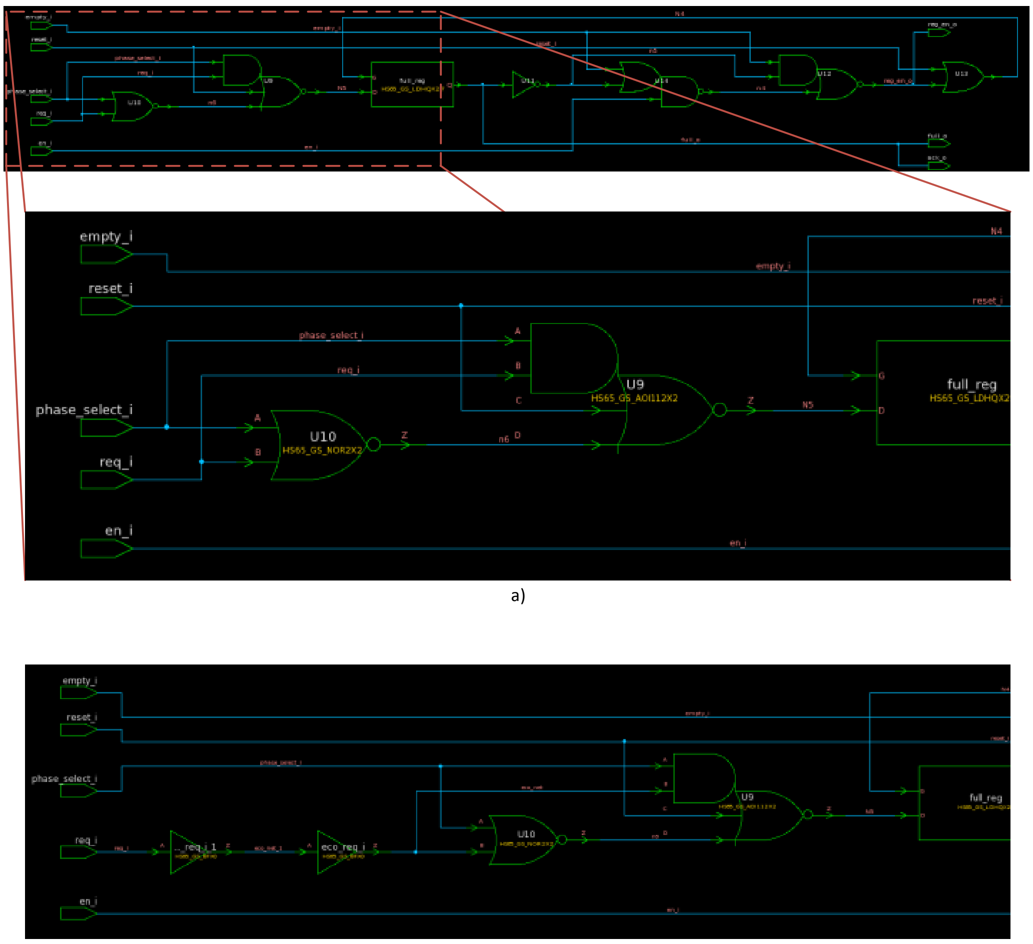


Figure 5.1 – FIFO's write controller circuit a) after logic synthesis and b) after adding extra cells to control the place where the delay line will be inserted.

As stated above, for the circuit shown in Figure 5.1(a) it is desired to insert a delay line between *req_i* and its first wire fork. However, wire forks are not referable in Synopsys' tools, and, according to guideline number four, a delay-line start and endpoints must not be referenced by a net name. The approach taken to overcome these issues is the insertion of extra cells in order to create a path where delay lines can be appropriately placed. These extra cells cannot change the circuit's logic - therefore, only buffers and pairs of inverters can be used. In an attempt to avoid inserting extra delay where interconnect delay alone are enough to fulfill timing requirements, a custom "wire cell", composed of a single wire with input and output pins, was created. This cell was called *HS65_GS_BFX0* to maintain the naming pattern used by the standard-cell library, and the property *size_only* was set, preventing it from being removed in logical optimizations, but allowing it to be replaced by actual buffers when needed. The extra cells were manually inserted, based on the timing constraints defined during design. Figure 5.1(b) shows the FIFO's write control circuit with a pair of wire cells inserted. The delay-line can be created from the startpoint of the wire cell connected to *req_i* to the endpoint of the wire cell connected to the wire fork, ensuring that (*Giv)* is respected.

Situations where, due to how the circuit was synthesized, it is not possible to find a place meeting the guidelines to insert the extra cells can be solved by creating a new entity wrapping the problematic circuit. Since the synthesis flow keeps the hierarchy of the circuit, the additional entity will have a port for each signal, creating a place to insert extra cells.

The wire cell was modeled as a single wire with an input and an output pin, where the output is directly connected to the input pins. Parasitics were extracted using Mentor Graphic's Calibre PEX tool. Electrical characterization was performed using Cadence's Encouter Library Characterization tool employing the same non-linear table stimuli used in the characterization of a minimum-size buffer (output capacitance vector and input slope vector). The abstract view, shown in Figure 5.2, was created using Cadence's Abstract Editor. Nota that the wire had to be split to generate the .LEF files, since the tool does not allow input and output pins to share the same net. However, this does not compromise the functionality of the circuit, given that this view is only required for place and route, and power and timing models employed the RC model of a full wire. Additionally, the wire cell had the same height of a standard-cell of the library, in order for it to be compatible during the placement; VDD and GND rails are also in the default position of the library. For this design, after the physical synthesis, all wire cells used in the IIs and OIs were substituted by buffers, suggesting that interconnect delay was not enough to fulfill the timing constraints on these circuits.

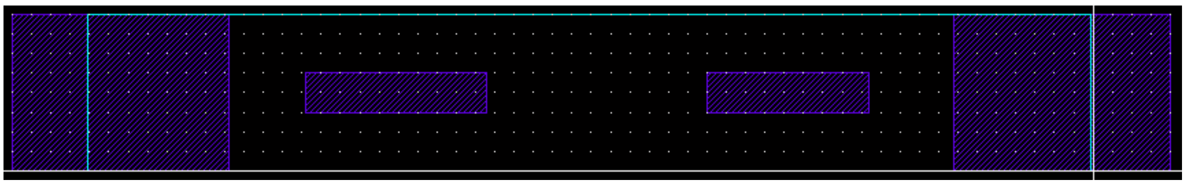
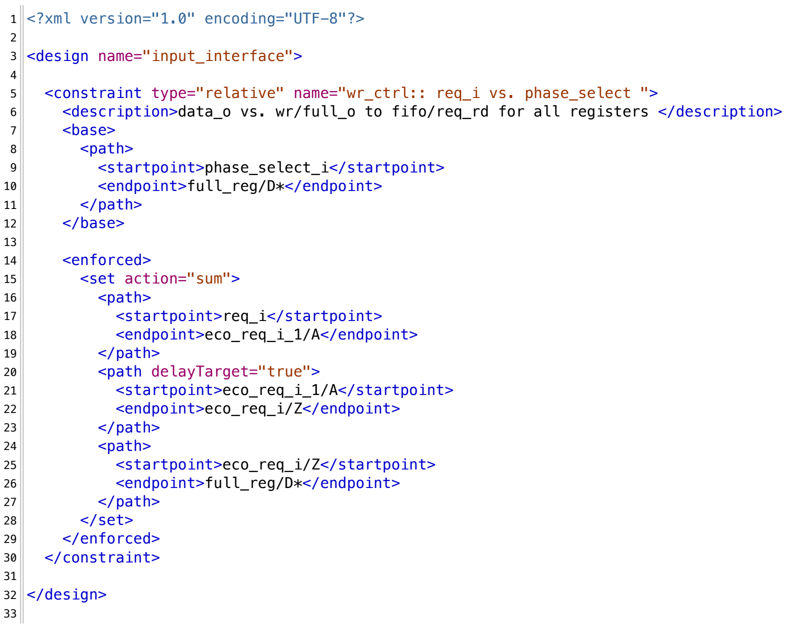


Figure 5.2 – Abstract view of wire cell, rotated by 90º.

5.1.2 Automated Constraint Enforcement

Scripts and functions were created to generate an automated environment for the iterative process of constraint enforcement described in the previous Section. This environment was called Asynchronous Constraints for Design Compiler (ACDC) and comprises a set of scripts written in TCL and Python that take as input an XML file describing the relative timing constraints of the circuit after logic synthesis, as exemplified in Figure 5.3. From this file, it generates TCL scripts capable of checking, setting and reporting the defined constraints inside Synopsys' tools. ACDC performs consistency checks in order to find mistakes in the manually generated XML file. At first, it checks if the file is well formatted and semantically correct – it verifies, for example, if there is exactly one delay target set on the enforced path, among many other verifications. After that, the validity of path names is check with Synopsys' tools, and constraints containing non-existent paths are disabled. If a delay target is shared between constraints, the tool calculates the maximum target delay value of all shared constraints, and applies it as the path's minimum delay, ensuring that all constraints are fulfilled.

The XML structure used by ACDC, shown in Figure 5.3, is very flexible and allows the representation of complex paths. Sets, which can be nested, are used to represent a group of paths. Each set has an action property that defines how its delay is calculated. The "sum" action returns the sum of all enclosed path- and set- delays, which is useful to represent linear paths. The "max" action returns the maximum delay among all enclosed paths and sets, and is helpful to represent path branches.



```xml
<?xml version="1.0" encoding="UTF-8"?>

<design name="input_interface">

  <constraint type="relative" name="wr_ctrl:: req_i vs. phase_select ">
    <description>data_o vs. wr/full_o to fifo/req_rd for all registers </description>
    <base>
      <path>
        <startpoint>phase_select_i</startpoint>
        <endpoint>full_reg/D*</endpoint>
      </path>
    </base>

    <enforced>
      <set action="sum">
        <path>
          <startpoint>req_i</startpoint>
          <endpoint>eco_req_i_1/A</endpoint>
        </path>
        <path delayTarget="true">
          <startpoint>eco_req_i_1/A</startpoint>
          <endpoint>eco_req_i/Z</endpoint>
        </path>
        <path>
          <startpoint>eco_req_i/Z</startpoint>
          <endpoint>full_reg/D*</endpoint>
        </path>
      </set>
    </enforced>
  </constraint>

</design>
```

Figure 5.3 – XML file describing the relative timing constraints of the FIFO Write Control circuit, shown in Figure 5.1(b).

Additionally, two TCL functions were created to simplify delay extraction and enforcement. The *custom_get_delay* function returns the delay between the start and endpoints used as, respectively, the first and second parameters of the function. A third optional parameter is the type of delay, which can be maximum (max, the default option), or minimum (min). *Custom_set_delay* wraps the *set_min_delay* command to check if the delay constraint was properly set – stopping the execution if an error occurs. It takes three parameters: the start and endpoints of the delay line, and the target delay value.

```
 1 # AC_set_constraints Function (Generated Automatically)
 2 proc AC_set_constraints {} {
 3   # Get Delays
 4   set AC_path0 [custom_get_delay req_i eco_req_i_1/A]
 5   set AC_path0_min [custom_get_delay req_i eco_req_i_1/A min]
 6   set AC_path1 [custom_get_delay phase_select_i full_reg/D*]
 7   set AC_path1_min [custom_get_delay phase_select_i full_reg/D* min]
 8   set AC_path2 [custom_get_delay eco_req_i_1/A eco_req_i/Z]
 9   set AC_path2_min [custom_get_delay eco_req_i_1/A eco_req_i/Z min]
10   set AC_path3 [custom_get_delay eco_req_i/Z full_reg/D*]
11   set AC_path3_min [custom_get_delay eco_req_i/Z full_reg/D* min]
12
13   # Set Constraints
14   echo "\n***************************************************** "
15   echo " Setting min_delay Constraints: "
16
17   ###### Set Constraints
18   # Constraint 'wr_ctrl:: req_i vs. phase_select ' :
19   set AC_aux_base [expr $AC_path1 ]
20   set AC_aux_enforced [expr [expr $AC_path0_min + $AC_path2_min + $AC_path3_min ]  ]
21   set AC_aux_delta [expr $AC_aux_base - $AC_aux_enforced + $AC_path2_min ]
22   set AC_cnst0 $AC_aux_delta
23   echo "\tConstraint 'wr_ctrl:: req_i vs. phase_select ' set. "
24
25   set AC_aux $AC_cnst0
26   custom_set_min_delay eco_req_i_1/A eco_req_i/Z $AC_aux
27
28 }
29
```

Figure 5.4 – Fragment of the TCL scripts generated by ACDC. This function sets minimum delay constraints.

Figure 5.4 shows the ACDC-generated function used to set the constraints described in the XML file shown in Figure 5.3. Lines 4 through 11 extract the minimum and maximum delays of each path referenced in the XML file and assign them to custom variables. Line 19 calculates the maximum delay of the base path (AC_path1), while the next line calculates the minimum delay of the enforced path – composed by the delay from the path's startpoint to the delay-line's startpoint (AC_path0_min), delay-line's delay (AC_path2_min), and delay from the delay-line's endpoint to the path's endpoint (AC_path3_min). Line 22 calculates the delay-target's delay, as shown in Equation 1(b). Lines 25 and 26 apply the previously calculated delay to the delay line. If the delay line was shared between constraints, the maximum delay target value of all constraints would be calculated in line 25.

Equation 1 – (a) The enforced-path's delay must be equal or grater to the base-path's delay; (b) equation to calculate the delay-line's delay, by isolating it from (a).

$$AC\_path0\_min + AC\_path2\_min + AC\_path3\_min \geq AC\_path1 \qquad \text{(a)}$$

$$AC\_path2\_min \geq AC\_path1 - (AC\_path0\_min + AC\_path3\_min) \qquad \text{(b)}$$

5.2 Synthesis

Synthesis was the most challenging step in the development of this work. Synthesis flows for bundled-data asynchronous circuits using commercial tools are scarse. In fact, the only reference found was [GHI13], where very little details are given about the synthesis process. During the many synthesis attempts throughout the development of this work, it was observed that, as systems become larger, more relative timing constraints are needed in order to ensure proper operation of the circuit. Also, as the number of constraints grows, more overlaps may happen, which can cause the addition of redundant delay lines, degrading the circuit's performance. Additionally, the number of iterations needed to fulfill the constraints increases.

Based on those observations, a synthesis flow was created in order to avoid conflicts between constraints. It consists of synthesizing each module of the system individually, in a bottom up manner. The post physical synthesis netlist of each circuit, generated after the respective timing constraints were met, is used, instead of its behavioral VHDL implementation, to instantiate the module. Even though it is not guaranteed that the constraints will still be fulfilled once the netlist is instantiated, as the placement and routing of the new circuit will be different from the one when the netlist was generated, it is fair to assume that, if needed, the amount of extra delay to meet the constraints will be small. Therefore, when synthesizing a circuit that instantiates other blocks, the relative timing constraints calculations can take into account the delays of these blocks, which are known at this stage (post-synthesis), possibly creating smaller delay lines. Note that the XML constraints file has to include the internal timing constraints of the instantiated modules as well, in order to guarantee that all constraints are met.

The following logic and physical flows were used to perform the synthesis of every component of BaT-Hermes. Synopsys' Design Compiler and IC Compiler were used to perform, respectively, logic and physical synthesis.

5.2.1 Logic Synthesis Flow

The logic synthesis flow can be divided in three phases: settings, initial synthesis, and optimization. In the first phase, tool configurations and design-specific settings are applied. Next, an initial synthesis is performed in order to map the design using the standard cell library. Finally, the design is optimized to a given target: low area, high performance, or low power.

In asynchronous systems, not only the logic function, but also the structure of the circuit defines the system's behavior [GHI13]. To guarantee that Design Compiler does not make structural changes during synthesis, logic optimizations can be disabled with the command *set_structure*. Next, timing loops of the instantiated modules need to be disabled to avoid the insertion of loopbreakers. A timing loop is a combinational circuit with feedback signal – for example, the Mousetrap stage control latch, where its output signal is fed back as the enable signal, after passing through an XNOR gate. As such loops prevent Design Compiler from performing static timing analysis, the tool inserts extra buffers in the feedback path and disables the timing through them, breaking the feedback path – these buffers are called loopbreakers. If loopbreakers are inserted in a path with a relative timing constraint, ACDC may not be able to calculate the delay of the path, and may fail. Also, the *dont_touch* flag is set for instantiated modules to preserve the delay lines and prevent logic changes.

After these initial settings, logic synthesis is performed with the command *compile_ultra*. The flag *no_autoungroup* is used to keep the hierarchical structure of the circuit. The optimization phase consists in setting constraints to achieve a desired target and running *compile_ultra* again to optimize the circuit. BaT-Hermes aims for high performance, therefore maximum delay constraints for critical paths were set with the *set_max_delay* command. Additionally, timing loops related to the newly synthesized circuit also have to be disabled. The insertion of extra buffers to allow the creation of delay lines can be performed after the optimizations. Before storing the design, the *size_only* flag is applied to the circuit to allow cell resize and buffer insertion while preventing logic changes.

5.2.2 Physical Synthesis Flow

Similarly to the logic synthesis, the physical synthesis flow can also be divided in three phases: settings, initial synthesis and constraint enforcement. The input to this flow is the mapped netlist generated in the logic synthesis. Initially, the timing loops of the circuit to be synthesized are disabled and the *dont_touch* property is applied to the circuit. The *set_cost_priority* command is used to increase the priority of minimum delay constraints, since they are used by ACDC to enforce the relative timing constraints. Next, the initial physical synthesis is performed using a standard synthesis flow.

After initial synthesis, the *dont_touch* property is replaced by *size_only* in order enable cell resize and buffer insertion. The buffers and inverters from the core library present asymmetric rise and fall times, which can greatly degrade the performance of the circuit if used to create the delay lines. This is because the circuit is based on transition signaling and, therefore, the worst case between rising and falling propagation delays is employed for dimensioning the delay lines. The bigger the difference between these delays, the bigger is the impact in the average latency, as the handshake protocol relies on alternated falling and rising edges of the request signal. The solution found was to build the delay lines with standard cells designed for clock tree synthesis, as they are symmetric with respect to rise and fall times. The ACDC environment is then employed to enforce the relative timing constraints.

5.2.3 BaT-Hermes Synthesis

The BaT-Hermes router is composed by a set of ports, each with IIs and OIs. Each interface can be synthesized individually and saved as a hard macro. The same OI macro can be instantiated across all ports, whereas a different II macro must be generated for each port, since the Routing Control logic depends on the address of the router and the in which port it will be used. The router is assembled by interconnecting IIs to OIs, as discussed in Section 4.1.2.

Unfortunately, due to the problems that had to be overcome in order to create the synthesis flow, a full router could not be synthesized in time to be included in this work. However, the local port of a router, featuring 16-bit flit size and an 8-flit input buffer, was successfully synthesized and validated.

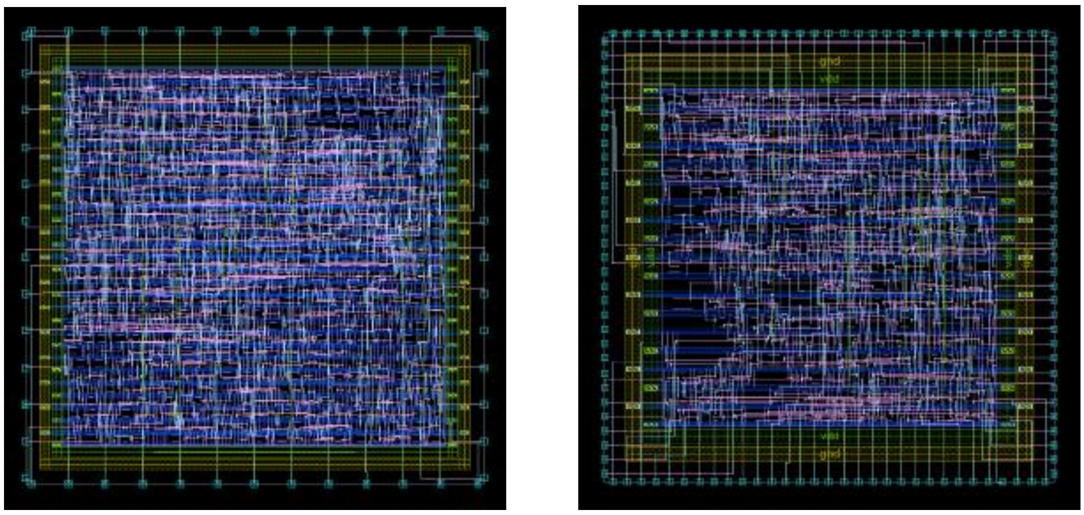


Figure 5.5 – Layout of synthesized a) Input Interface and b) Output Interface circuits.

The synthesis targeted the STMicroelectronics 65nm general-purpose standard-Vt CMOS technology. The ASCEnD cell library, available for this technology, provided the required asynchronous cells. Figure 5.5 shows the layout of the synthesized interfaces. Table 2 compares the total cell area of each circuit with the area occupied by buffers and inverters, most of which were used in delay lines. Post-synthesis functional validation is shown in Section 4.3.

Table 2 – Area used by the synthesized Input and Output Interfaces.

	Total Cell Area	Buffer and Inverter Area	% of Buffers and Inverters
Input Interface	4357μm^2	1489μm^2	34,1%
Output Interface	1428μm^2	600μm^2	42,0%

5.3 Post-Synthesis Validation

After physical synthesis, the netlist and .SDF file of the Input and Output Interfaces were exported and used to perform the post-synthesis functional validation with back-annotated delays. Cadence's SimVision simulator was employed in this task.

5.3.1 Input Interface

The test case applied to the II simulates an IP Core connected to the local port sending packets to all other cores of a 3x3 mesh network. Figure 5.6 shows an overview of the waveforms generated by this test case. Note the stalled *ack_o* signal (1) generated when the input buffer becomes full. The signals that communicate with the Output Interface were grouped to help identifying the destination of each request: Outport 0 groups signals sent to the east port; Outport 1 combine signals sent to the west port; signals intended for the north port are grouped under Outport 2; Outport 3 combines signals sent to the south port. Shared signals, like *data_o*, *req_data_o*, and *last_flit_o* were replicated in each of these groups to help to comprehend which handshakes are taking place at each instant. The signals from Outport 4 refer to the local port and are not shown, since loopback connections are not supported by BaT-Hermes.

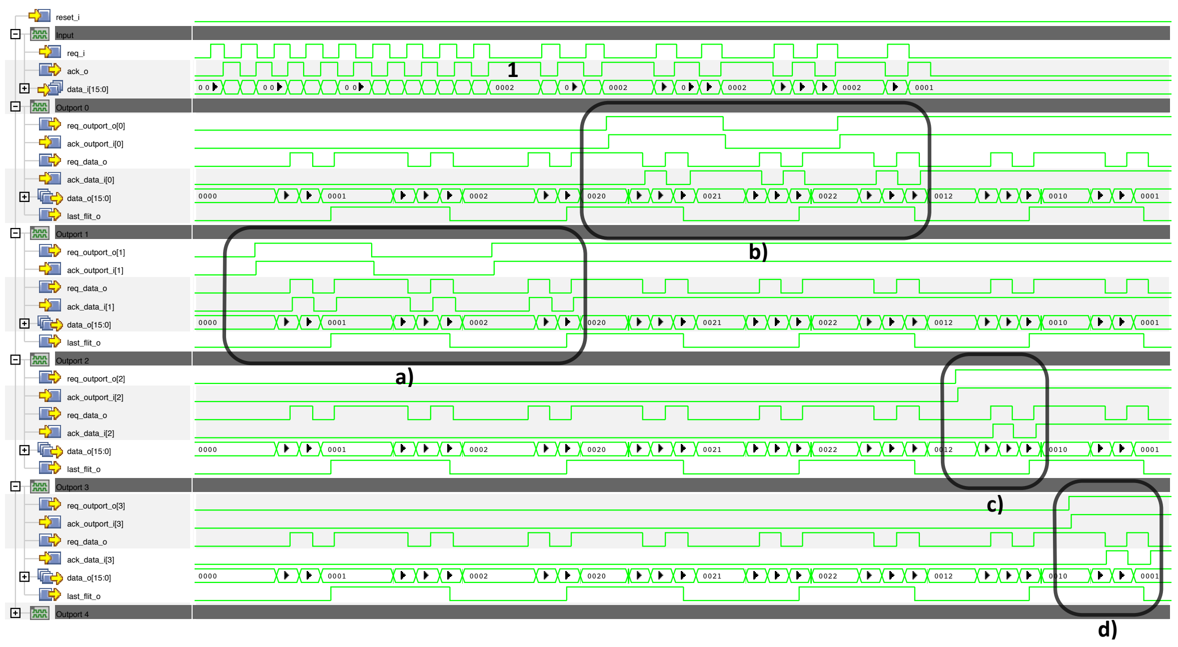


Figure 5.6 – Post-synthesis simulation of an Input Interface sending packets to the a) west, b) east, c) north, and d) south Output Interfaces.

Figure 5.7 shows a detailed view of Figure 5.6(a) to illustrate the flits flowing through the II. Arrows show the propagation of the first flit of each packet sent to the west OI.

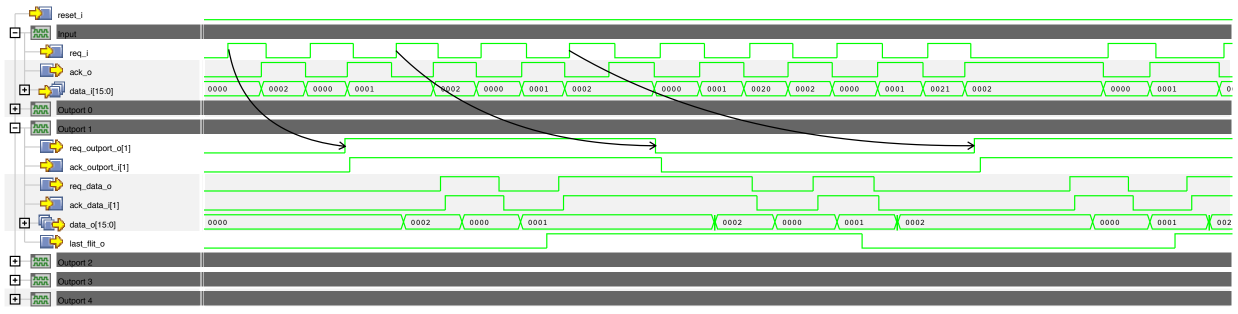


Figure 5.7 - Detailed view of Figure 5.6(a), showing flits propagating through the Input Interface.

The forward latency, measured as the time it takes for an incoming request to propagate to the output, is 2.255ns. The average time between data handshakes is 1.08ns. At the input, the average delay between successive acknowledges is 0.745ns.

5.3.2 Output Interface

The test case applied to the Output Interface simulates four IIs trying to send a packet simultaneously. Figure 5.8 shows the simulation waveforms. Note that the *req_data_i* (1) signals, connected to Outputs 0 and 2, are out-of-phase with the respective *ack_data_o*. This allows simulating the programmable phase matcher circuit, for the sake of validation. All *req_outport_i* (2) signals switch at the same time, competing for access to the output channel. Arbitration works as expected, grating access to only one Outport Control at any given time.

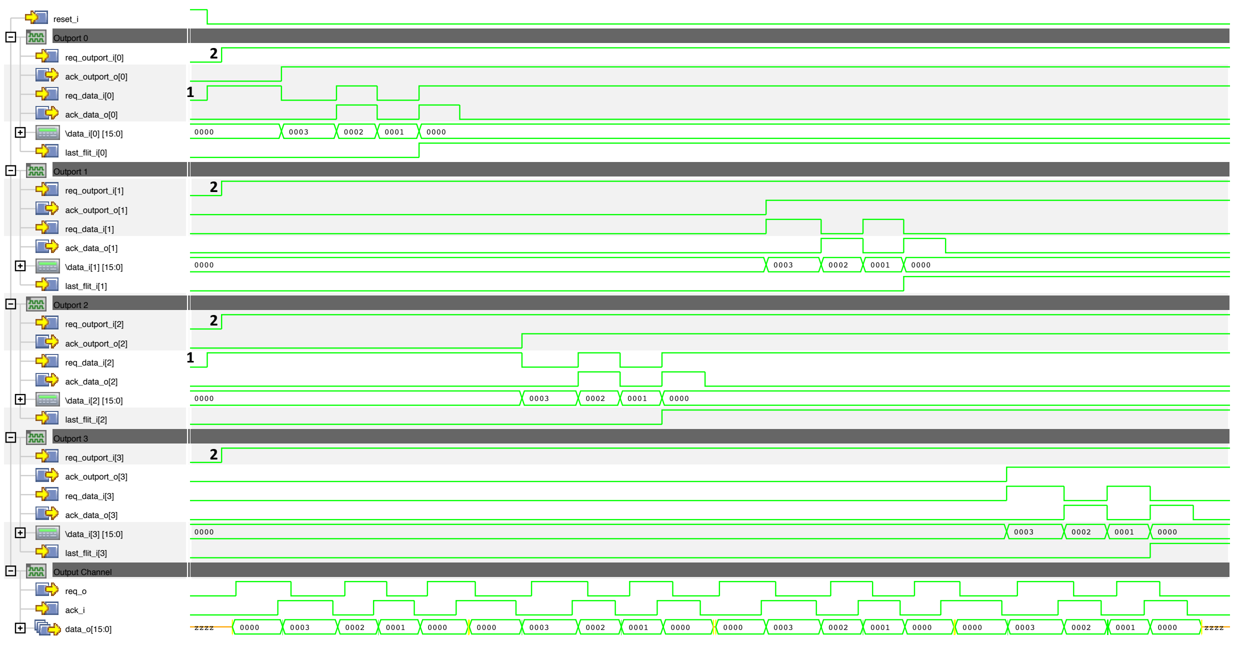


Figure 5.8 - Post-synthesis simulation of an Output Interface receiving four packets simultaneously.

The average forward delay, computed from the *req_data_i* to *req_o*, is 0.677ns. The average cycle time at the output channel is 0.95ps.

6. CONCLUSIONS AND FUTURE WORK

This work proposed an asynchronous transition-signaling bundled-data NoC router, called BaT-Hermes. The design, based on the highly decoupled architecture of the YeAH! router, is composed of several independent modules, that can be connected to assemble a router. A full BaT-Hermes was validated through behavioral simulation.

Unfortunately, the synthesis process was far more complex than anticipated and a full router could not be synthesized in time to be included in this work, and is left as a future work. However, one port, composed by an II and OI was synthesized to the STMicroelectronics 65nm CMOS technology and validated through post-synthesis simulation, enabling the validation of the proposed synthesis methodology and the ACDC environment. In this way, apart from the designed NoC router, another contribution from this work is the methodology for synthesis of bundled-data circuits using commercial CAD tools, along with automated environment for enforcing relative timing constraints, namely ACDC.

6.1 Future Work

This work presents many topics for further research. The most immediate is the synthesis of a full BaT-Hermes router and its comparison with the fully synchronous YeAH! router. Due to the many structural similarities between them, this is a great opportunity to assess the differences of each implementation style with respect to metrics like latency, throughput, area, and power consumption.

Another interesting topic for research is alternative ways to implement the router, aiming for a simpler synthesis process. One idea is to only use Mousetrap stages as storage elements, implementing the router as a "pure pipeline". This approach can greatly simplify the synthesis because relative timing constraints are restricted to each pair of stages. An interesting study could be conducted comparing the final circuit performance with the synthesis effort for various design techniques.

Other possibility for future work is on the improvement of the ACDC environment to support automatic constraint detection. Additionally, related work could be conducted about synchronization interfaces that need to be used when interfacing asynchronous and synchronous circuits.

REFERENCES

[AMD05] Amde, M.; Felicijan, T.; Efthymiou, A.; Edwards, D.; Lavagno, L. *"Asynchronous on-chip networks"*. IEE Proceedings – Computers and Digital Techniques, 152(2), Mar. 2005, pp. 273-285.

[BEI05] Beigné, E.; Clermidy, F.; Vivet, P.; Clourad, A.; Renaudin, M. *"An Asynchronous NOC Architecture Providing Low Latency Service and its Multi-level Design Framework*". Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05) , Mar. 2005, pp. 54-63.

[BEL08] Bell, S.; Amann, J.; Conlin, R.; Joyce, K.; Leung, V.; MacKay, J.; Reif, M.; Bao, L. ; Brown, J.; Mattina, M.; Chyi-Chang Miao; Ramey, C.; Wentzlaff, D.; Anderson, W.; Berger, E.; Fairbanks, N.; Khan, D.; Montenegro, F.; Stickney, J.; Zook, J. *"TILE64™ Processor: A 64-Core SoC with Mesh Interconnect"*. In: International Solid State Circuits Conference (ISSCC'08), 2008, pp. 87-89.

[BHA13] Bhadra, D.; Vij, V.; Stevens, K. *"A Low Power UART Design Based on Asynchronous Techniques"*. In: Midwest Symposium on Circuits and Systems (MWSCAS'13), 2013, pp. 21–24.

[BJE05] Bjerregaard, T.; Sparso, J. *"A Router Architecture for Connection-Oriented Service Guarantees in the Mango Clockless Network-on-Chip"*. In: Design, Automation and Test in Europe Conference and Exhibition (DATE'05), 2005, pp. 1226-1231.

[CHA84] Chapiro, D. M. "Globally-Asynchronous Locally Synchronous Systems". PhD Thesis, Stanford University, 1984, 134p.

[DOB09] Dobkin R.; Ginosar R.; Kolodny A. *"QNoC asynchronous router"*. Integration, the VLSI Journal, 42(2), Feb. 2009, pp. 103-115.

[GEB10] Gebhardt, D.; You, J.; Stevens, K. S. *"Comparing Energy and Latency of Asynchronous and Synchronous NoCs for Embedded SoCs"*. In: Fourth ACM/IEEE International Symposium of Networks-on-Chip (NOCS'10), 2010, pp. 115-122.

[GHI13] Ghiribaldi, A.; Bertozzi, D.; Nowick, S. M. *"A Transition-Signaling Bundled Data NoC Switch Architecture for Cost-Effective GALS Multicore Systems"*. In: Design, Automation & Test in Europe Conference & Exhibition (DATE´13), 2013, pp. 332-337.

[GIL11] Gill, G.; Attarde, S. S.; Lacourba, G.; Nowick, S. M. *"A Low-Latency Adaptive Asynchronous Interconnection Network Using Bi-Modal Router Nodes"*. In: Fifth ACM/IEEE International Symposium of Networks-on-Chip (NOCS'11), 2011, pp. 192-200.

[HAU95] Hauck, S. *"Asynchronous Design Methodologies: An Overview"*. Proceedings of the IEEE, 83(1), Jan. 1995, pp. 69-93.

[INT13] *"Intel Xeon Phi Core Micro-Architecture"*. Avaiable at: http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture.

[ITR11] ITRS. "International Technology Roadmap for Semiconductors 2011 Edition". Design Chapter. Available at http://www.itrs.net/Links/2011ITRS/2011Chapters/2011Design.pdf.

[KAH05] Kahle, J. A.; Day, M. N.; Hofstee, H. P.; Johns, C. R.; Maeurer, T. R. and Shippy D. "Introduction to the cell multiprocessor". IBM J. Res. Dev. 49, 4/5 (July 2005), 589-604.

[KRZ10] Krzysztof, I. *"CMOS Processors and Memories"*. Springer, New York, 2010, 382 p.

[MAR90] Martin, A. J. *"The limitations to delay-insensitivity in asynchronous circuits"*. In: 6th Conference on Advanced Research in VLSI, 1990, pp. 263-278.

[MOR04] Moraes, F.; Calazans, N. L. V.; Mello, A. V.; Möller, L. H.; Ost, L. C. *"HERMES: an infrastructure for Low Area Overhead Packet-switching Networks on Chip"*. Integration, the VLSI Journal, 38(1), Oct. 2004, pp. 69-93.

[MOR11a] Moreira, M. T.; Oliveira, B. S.; Pontes, J. J. H.; Calazans, N. L. V. *"A 65nm Standard Cell Set and Flow Dedicated to Automated Asynchronous Circuits Design"*. In: 24th IEEE International SoC Conference (SoCC'11), 2011, pp. 99-104.

[MOR11b] Moreira, M. T.; Oliveira, B. S.; Pontes, J. J. H.; Moraes, F. G.; Calazans, N. L. V. *"Adapting a C-Element Design Flow for Low Power"*. In: IEEE International Conference on Electronics, Circuits, and Systems, Beirut (ICECS'11), 2011, pp. 45-48.

[MYE01] Myers, C. *"Asynchronous Circuit Design"*. New York: John Wiley & Sons, Inc., 2001, 422p.

[NOW11] Nowick, S. M.; Singh M. *"High-Performance Asynchronous Pipelines: An Overview"*. IEEE Design & Test of Computers, 28(5), Sep.-Oct. 2011, pp. 8-22.

[PEE96] Peeters, A. *"Single-Rail Handshake Circuits"*. PhD Thesis, Eindhoven University of Technology, Jun. 1996, 186p.

[PON10a] Pontes, J. J. H.; Moreira, M. T.; Moraes, F. G.; Calazans, N. L. V. *"Hermes-A - An Asynchronous NoC Router with Distributed Routing"*. In: International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'10), LNCS, 6448, Grenoble, 2010, pp. 150-159.

[PON10b] Pontes, J. J. H.; Moreira, M. T.; Moraes, F. G.; Calazans, N. L. V. *"Hermes-AA - A 65nm Asynchronous NoC Router with Adaptive Routing"*. In: 23rd IEEE International SoC Conference (SOCC'10). IEEE Computer Society, Las Vegas, 2010, pp. 493-498

[RAB03] Rabaey, J. M.; Chandrakasan A.; Nikolic, B. *"Digital Integrated Circuits: A Design Perspective"*. Pearson Education, Upper Saddle River, 2003, 761p.

[SHE08] Sheibanyrad, A.; Miro-Panades, I.; Greiner, A. *"Multisynchronous and Fully Asynchronous NoCs for GALS Architectures"*. IEEE Design & Test of Computers, 25(6), Nov.-Dec. 2008, pp. 572-580.

[SIN07] Singh, M.; Nowick, S. M. *"MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines"*. IEEE Transactions on VLSI Systems, 15(6), Jun. 2007, pp. 684-698.

[SPA01] Sparso, J.; Furber, S. *"Principles of Asynchronous Circuit Design – A Systems Perspective"*. London: Kluwer Academic Publishers, 2001, 337p.

[STE09] Stevens, K. S.; Gebhardt, D.; You, J.; Xu, Y.; Vij, V.; Das, S.; Desai, K. *"The Future of Formal Methods and GALS Design*". Electronic Notes in Theoretical Computer Science, 245, 2009, pp. 115–134.

[SUT89] Sutherland, I. E. *"Micropipelines"*. Communications of the ACM, 32(6), Jun. 1989, pp. 720-738.

[WIJ11] van der Wijngaart, R. F.; Mattson, T. G. and Haas, W. "Light-weight communications on Intel's single-chip cloud computer processor". ACM SIGOPS Operating Systems Review, vol.40(1), February 2011, pp. 73-83.

A. BaT-Hermes VHDL Description

This appendix contains the VHDL implementation code and simulation wrappers for each module of BaT-Hermes.

A.1 Settings Package

```vhdl
--! @file hermes_bd_package.vhd
--! @brief Global definitions for the Hermes Bundled-data Network
--! @author Matheus Gibiluka, mgibiluka@me.com
--! @date 2013-07-13


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

package hermes_bd_package is
    -- The flit size should be modified with scripts
    -- since was not possible to change its value
    -- with genereics yet
    constant FLIT_SIZE : integer range 1 to 64 := 16;

    constant PORTS_NUMBER: integer := 5;

    -- Port IDs
    constant EAST  : integer := 0;
    constant WEST  : integer := 1;
    constant NORTH : integer := 2;
    constant SOUTH : integer := 3;
    constant LOCAL : integer := 4;

    -- Types
    type flit_size_array is array(natural range<>) of std_logic_vector(FLIT_SIZE-1 downto 0);

    -- Functions
    -- Helper functions useful to implement phase-matching
    function xor_all_bits_considering_comm_ports (X : std_logic_vector; COMM_PORTS : std_logic_vector; THIS_PORT: integer ) return std_logic;
    function xor_all_bits_but_one_considering_comm_ports (X : std_logic_vector; N : integer; COMM_PORTS : std_logic_vector; THIS_PORT : integer) return std_logic;

    function number_of_used_ports (COMM_PORTS : std_logic_vector; THIS_PORT : integer) return integer;


end package;


package body hermes_bd_package is

    -- Helper function to XOR all bits, considering COMM_PORTS and THIS_PORT values.
    function xor_all_bits_considering_comm_ports (X : std_logic_vector; COMM_PORTS : std_logic_vector; THIS_PORT: integer ) return std_logic is
        variable aux : std_logic := '0';
    begin
        for i in X'range loop
            if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) then
                aux := aux XOR X(i);
            end if;
        end loop;
        return aux;
    end xor_all_bits_considering_comm_ports;


    -- Helper function to XOR all bits but one, considering COMM_PORTS and THIS_PORT values.
    function xor_all_bits_but_one_considering_comm_ports (X : std_logic_vector; N : integer; COMM_PORTS : std_logic_vector; THIS_PORT : integer) return std_logic is
        variable aux : std_logic := '0';
    begin
        for i in X'range loop
            if ((i /= N) and (i /= THIS_PORT) and (COMM_PORTS(i) = '1')) then
                aux := aux XOR X(i);
            end if;
        end loop;
        return aux;
    end xor_all_bits_but_one_considering_comm_ports;

```

Figure A.1 – VHDL source code for the settings package (part 1 of 2).

```vhdl
     -- Helper function to calculate the actual number of outports being used
    function number_of_used_ports (COMM_PORTS : std_logic_vector; THIS_PORT : integer) return integer is
        variable cnt : integer := 0;
    begin
        for i in COMM_PORTS'range loop
            if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) then
                cnt := cnt + 1;
            end if;
        end loop;
        return cnt;
    end number_of_used_ports;

end package body;

```

Figure A.2 – VHDL source code for the settings package (part 2 of 2).

A.2 FIFO

```vhdl
--! @file fifo_wr_ctrl.vhd
--! @brief Write control unit for circular FIFO. All handshake signals use transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-07-24

------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--         reset_i---------->|             |
--                           |             |
--            en_i---------->|             |
--                           |             |
--  phase_select_i---------->|             |
--           req_i---------->|             |---------->full_o
--                           |             |
--           ack_o<----------|             |<----------empty_i
--                           |             |
--                           |             |---------->reg_en_o
--                            -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity fifo_wr_ctrl is
    port(
        reset_i        : in  std_logic; --! Active-high reset signal.
        en_i           : in  std_logic; --! Signal to enable this controller.
        req_i          : in  std_logic; --! Request signal indicating that new data is available to be written.
        ack_o          : out std_logic; --! Acknowledge signal relative to req_i. Indicates that the data has been stored.
        phase_select_i : in  std_logic; --! Signal used for full_o phase matching.
        full_o         : out std_logic; --! Request signal indicating that new data has been written on the register.
        empty_i        : in  std_logic; --! Acknowledge signal relative to full_o. Indicates that the data has been read.
        reg_en_o       : out std_logic  --! Signal to control the register(latch). High makes latch transparent(store); low turns it opaque(hold).
    );
end fifo_wr_ctrl;


architecture fifo_wr_ctrl of fifo_wr_ctrl is
    signal full   : std_logic; -- Same as full_o
    signal req    : std_logic; -- req_i signal phase-matched to full_o
    signal reg_en : std_logic; -- Signal that controls the full_o latch; Same as reg_en_o

begin
    full_o <= full;
    ack_o <= full;
    reg_en_o <= reg_en;

    -- Matching req_i signal to the phase of full_o
    req <= req_i XOR phase_select_i;

    -- Enables next write if the register's data has been read
    -- [It's this register's turn to write AND the register is empty (full/empty handshake is complete)]
    reg_en <= en_i AND (empty_i XNOR full);

    -- Latch that controls requests (full_o)
    process(reset_i, reg_en, req)
    begin
        if (reset_i = '1') then
            full <= '0';
        elsif (reg_en = '1') then
            full <= req;
        end if;
    end process;

end fifo_wr_ctrl;

```

Figure A.3 – VHDL source code for FIFO Write Control module.

```vhdl
--! @file fifo_wr_ctrl_wrapper.vhd
--! @brief Wrapper for fifo_wr_ctrl
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-07-24

------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--         reset_i---------->|             |
--                           |             |
--            en_i---------->|             |
--                           |             |
--  phase_select_i---------->|             |
--           req_i---------->|             |---------->full_o
--                           |             |
--           ack_o<----------|             |<----------empty_i
--                           |             |
--                           |             |---------->reg_en_o
--                            -------------
------------------------------------------------------------------------
-- Wrapper Behavior:
--   0) Output handshake signals should be held (ack_o, full_o)
--   1) Handshake starts when (full_o /= empty_i)
--   2) Release order: full_o > ack_o
--   3) Wait for Ack (full_o = empty_i)
--
--   * Unlike the fifo_rd_ctrl_wrapper, there is no need to hold the en_i
--     signal. The internal closed loop issue is handled by making
--     sure that the ack_o is released after the full_o. This guarantees
--     that the right value of full will be available when the write_ptr
--     changes.
--
--  ** Because there is no internal delay, the reg_en_o signal will appear
--     as an impulse on the simulation. This signal will look as expected
--     when the request "req_i" is made after both "empty_i" ack arrives
--     and "en_i" is asserted.
--     (I.e: empty_i > en_i > req_i, or en_i > empty_i > req_i)
--
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity fifo_wr_ctrl_wrapper is
    port(
        reset_i        : in  std_logic; --! Active-high reset signal.
        en_i           : in  std_logic; --! Signal to enable this controller.
        req_i          : in  std_logic; --! Request signal indicating that new data is available to be written.
        ack_o          : out std_logic; --! Acknowledge signal relative to req_i. Indicates that the data has been stored.
        phase_select_i : in  std_logic; --! Signal used for full_o phase matching.
        full_o         : out std_logic; --! Request signal indicating that new data has been written on the register.
        empty_i        : in  std_logic; --! Acknowledge signal relative to full_o. Indicates that the data has been read.
        reg_en_o       : out std_logic  --! Signal to control the register(latch). High makes latch transparent(store); low turns it opaque(hold).
    );
end fifo_wr_ctrl_wrapper;

architecture sim of fifo_wr_ctrl_wrapper is
     -- Control signals
    signal hold_ack_o    : std_logic;
    signal hold_full_o   : std_logic;

    -- Aux signals
    signal aux_ack_o    : std_logic;
    signal aux_full_o   : std_logic;

begin

    -- Instance of fifo_wr_ctrl
    fifo_wr_ctrl_i: entity work.fifo_wr_ctrl
    port map (
        reset_i        => reset_i,
        en_i           => en_i,
        req_i          => req_i,
        ack_o          => aux_ack_o,
        empty_i        => empty_i,
        full_o         => aux_full_o,
        phase_select_i => phase_select_i,
        reg_en_o       => reg_en_o
    );

    -- Control Logic
    control: process
    begin
        -- Initializing control signals
        hold_ack_o <= '0';
        hold_full_o <= '0';
        wait until reset_i = '0';
        -- Hold handshake signals
        hold_ack_o <= '1';
        hold_full_o <= '1';
```

Figure A.4 – VHDL source code for simulation wrapper of FIFO Write Control (part 1 of 2).

```vhdl
        loop
            -- Wait request (full_o transitions when a valid req_i transition is made)
            if ((en_i /= '1') OR (aux_full_o = empty_i)) then
                wait until ((en_i = '1') AND (aux_full_o /= empty_i));
            end if;
            -- Release #1: full_o, after 1 ns
            wait for 1 ns;
            hold_full_o <= '0';
            -- Release #2: ack_o, after 1 ns;
            wait for 1 ns;
            hold_ack_o <= '0';
            wait for 1 ns;
            -- Hold handshake signals
            hold_ack_o <= '1';
            hold_full_o <= '1';

            -- Wait acknowledge from full/empty handshake
            if (empty_i /= aux_full_o) then
                wait until empty_i = aux_full_o;
            end if;

        end loop;
    end process;


    -- Latches to hold signals
    latch_ack_o: process(reset_i, hold_ack_o, aux_ack_o)
    begin
        if ((reset_i = '1') OR (hold_ack_o = '0')) then
            ack_o <= aux_ack_o;
        end if;
    end process;

    latch_full_o: process(reset_i, hold_full_o, aux_full_o)
    begin
        if ((reset_i = '1') OR (hold_full_o = '0')) then
            full_o <= aux_full_o;
        end if;
    end process;

end sim;

```

Figure A.5 - VHDL source code for simulation wrapper of FIFO Write Control (part 2 of 2).

```vhdl
--! @file fifo_rd_ctrl.vhd
--! @brief Read control unit for circular FIFO. All handshake signals use transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-07-24

-------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                            |             |
--         reset_i----------->|             |
--                            |             |
--             en_i---------->|             |
--                            |             |
--           full_i-----------|             |---------->req_o
--                            |             |
--                            |             |<----------phase_select_i
--         empty_o<-----------|             |<----------ack_i
--                            |             |
--                            -------------
-------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity fifo_rd_ctrl is
    port(
        reset_i        : in  std_logic; --! Active-high reset signal.
        en_i           : in  std_logic; --! Signal to enable this controller.
        full_i         : in  std_logic; --! Request signal indicating that new data has been written on the register.
        empty_o        : out std_logic; --! Acknowledge signal relative to full_i. Indicates that the data has been read.
        phase_select_i : in  std_logic; --! Signal used for empty_o phase matching.
        req_o          : out std_logic; --! Request signal indicating that new data is available to be read.
        ack_i          : in  std_logic  --! Acknowledge signal relative to req_o. Indicates that the data has been read.
    );
end fifo_rd_ctrl;

```

Figure A.6 - VHDL source code for FIFO Read Control module (part 1 of 2).

```vhdl

architecture fifo_rd_ctrl of fifo_rd_ctrl is
    signal ack         : std_logic; -- ack_i signal phase-matched to empty_o
    signal req         : std_logic; -- Same as req_o
    signal empty       : std_logic; -- Same as empty_o
    signal waiting_ack : std_logic; -- Signal that controls the empty_o latch

begin
    req_o <= req;
    empty_o <= empty;

    -- Matching ack_i signal to the phase of empty_o
    ack <= ack_i XOR phase_select_i;

    -- Signal indicating that a req_o request has been made, but hasn't received ack_i acknowledge yet
    waiting_ack <= empty XOR req;

    -- Latch that controls requests (req_o)
    process(reset_i, en_i, full_i)
    begin
        if (reset_i = '1') then
            req <= '0';
        elsif (en_i = '1') then
            req <= full_i;
        end if;
    end process;

    -- Latch that controls acknowledges (empty_o)
    process(reset_i, waiting_ack, ack)
    begin
        if (reset_i = '1') then
            empty <= '0';
        elsif (waiting_ack = '1') then
            empty <= ack;
        end if;
    end process;

end fifo_rd_ctrl;

```

Figure A.7 - VHDL source code for FIFO Read Control module (part 2 of 2).

```vhdl
--! @file fifo_rd_ctrl_wrapper.vhd
--! @brief Wrapper for fifo_rd_ctrl
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-07-25

-------------------------------------------------------------------------
-- Interface description:
--                               -------------
--                              |             |
--          reset_i----------->|             |
--                              |             |
--              en_i--------->|             |
--                              |             |
--            full_i----------|             |---------->req_o
--                              |             |
--                              |             |<----------phase_select_i
--          empty_o<----------|             |<----------ack_i
--                              |             |
--                               -------------
-------------------------------------------------------------------------
-- Wrapper Behavior:
--   0) Enable and output handshake signals should be held (en_i, full_o)
--   1) Handshake starts when (full_i /= empty_o) and (en_i = '1')
--   2) Release en_1 after 1 ns. To simulate delay and account for close
--      loop signal inside fifo_rd_ctrl (req_o controls the empty_o latch)
--   3) Wait for Ack (full_i = empty_i)
--   4) Release empty_o after 1 ns (To simulate delay)
--
--   * The closed loop signal makes the latch that controls empty_o become
--     opaque instantly due to a phase mismatch introduced by the delay
--     of the empty_next_i signal. To fix this without having to change
--     the fifo_rd_ctrl we had to simulate a delay on the assertion of the
--     en_i signal (there is no wrapper arround the pointer counter). By
--     holding en_i for the duration of the empty_next_i delay, this
--     "race condition" is resolved (for simulation purpose).
-------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity fifo_rd_ctrl_wrapper is
    port(
        reset_i        : in  std_logic; --! Active-high reset signal.
        en_i           : in  std_logic; --! Signal to enable this controller.
        full_i         : in  std_logic; --! Request signal indicating that new data has been written on the register.
        empty_o        : out std_logic; --! Acknowledge signal relative to full_i. Indicates that the data has been read.
        phase_select_i : in  std_logic; --! Signal used for empty_o phase matching.
        req_o          : out std_logic; --! Request signal indicating that new data is available to be read.
        ack_i          : in  std_logic  --! Acknowledge signal relative to req_o. Indicates that the data has been read.
    );
end fifo_rd_ctrl_wrapper;
```

Figure A.8 - VHDL source code for simulation wrapper of FIFO Read Control (part 1 of 2).

```vhdl

architecture sim of fifo_rd_ctrl_wrapper is
     -- Control signals
    signal hold_en_i    : std_logic;
    signal hold_empty_o : std_logic;

    -- Aux signals
    signal aux_empty_o : std_logic;
    signal aux_en_i    : std_logic;

begin

    -- Instance of fifo_rd_ctrl
    fifo_rd_ctrl_i: entity work.fifo_rd_ctrl
    port map (
        reset_i        => reset_i,
        en_i           => aux_en_i,
        full_i         => full_i,
        empty_o        => aux_empty_o,
        phase_select_i => phase_select_i,
        req_o          => req_o,
        ack_i          => ack_i
    );

    -- Control Logic
    control: process
    begin
        -- Initializing control signals
        hold_en_i <= '0';
        hold_empty_o <= '0';
        wait until reset_i = '0';
        loop
            -- Hold all signals
            hold_en_i <= '1';
            hold_empty_o <= '1';

            -- Wait for valid handshake (rd_ctrl enabled and full_i request)
            if ((en_i /= '1') OR (full_i = aux_empty_o)) then
                wait until ((en_i = '1') AND (full_i /= aux_empty_o));
            end if;
            -- Release #1: en_i, after 1 ns
            wait for 1 ns;
            hold_en_i <= '0';
            -- Wait for ack_i acknowledge
            if (aux_empty_o /= full_i) then
                wait until (aux_empty_o = full_i);
            end if;
            -- Release #2: empty_o, after 1 ns
            wait for 1 ns;
            hold_empty_o <= '0';
            -- Wait 1 ns before holding it again
            wait for 1 ns;
        end loop;
    end process;


    -- Latches to hold signals
    latch_en_i: process(reset_i, hold_en_i, en_i)
    begin
        if ((reset_i = '1') OR (hold_en_i = '0')) then
            aux_en_i <= en_i;
        end if;
    end process;

    latch_empty_o: process(reset_i, hold_empty_o, aux_empty_o)
    begin
        if ((reset_i = '1') OR (hold_empty_o = '0')) then
            empty_o <= aux_empty_o;
        end if;
    end process;

end sim;

```

Figure A.9 - VHDL source code for simulation wrapper of FIFO Read Control (part 2 of 2).

```vhdl
--! @file fifo.vhd
--! @brief Handshake-based FIFO using transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-07-14

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--  BUFFER_DEPTH: Defines the depth of the buffer
------------------------------------------------------------------------
-- Interface description:
--                           -------------
--                          |             |
--        reset_i---------->|             |
--                          |             |
--       req_wr_i---------->|             |---------->req_rd_o
--                          |             |
--       ack_wr_o<----------|             |<----------ack_rd_i
--                          |             |
--                  WIDTH   |             |   WIDTH
--          data_i====/====>|             |=====/====>data_o
--                          |             |
--                           -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use work.hermes_bd_package.all;

entity fifo is
    generic(
        BUFFER_DEPTH : integer := 8 --! Defines the depth of the buffer.
    );
    port(
        reset_i  : in  std_logic; --! Active-high reset signal.
        req_wr_i : in  std_logic; --! Request signal indicating that new data is available to be written.
        ack_wr_o : out std_logic; --! Acknowledge signal relative to req_wr_i. Indicates that the data has been stored.
        data_i   : in  std_logic_vector(FLIT_SIZE-1 downto 0); --! Data input.
        req_rd_o : out std_logic; --! Request signal indicating that new data is available to be read.
        ack_rd_i : in  std_logic; --! Acknowledge signal relative to req_rd_o. Indicates that the data has been read.
        data_o   : out std_logic_vector(FLIT_SIZE-1 downto 0) --! Data output.
    );
end fifo;

architecture circular of fifo is
    signal write_ptr   : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Write Pointer (1-hot)
    signal read_ptr    : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Read Pointer (1-hot)
    signal ack_wr_ctrl : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Acknowledge signals generated by wr_ctrl (data has been written)
    signal req_rd_ctrl : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Request signals generated by rd_ctrl (data is available to be read)
    signal full        : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Request signals generated by wr_ctrl (data is available to be read)
    signal empty       : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Acknowledge signals generated by rd_ctrl (data has been read)
    signal reg_en      : std_logic_vector(BUFFER_DEPTH-1 downto 0); -- Register enable signals generated by wr_ctrl.

    signal ack_wr      : std_logic; -- Same as ack_wr_o
    signal req_rd      : std_logic; -- Same as req_rd_o
    signal next_wr_ptr : std_logic; -- Control signal for Write Counter (increment on rising edge)
    signal next_rd_ptr : std_logic; -- Control signal for Read Counter (increment on rising edge)

    type reg_t is array (0 to BUFFER_DEPTH-1) of std_logic_vector(FLIT_SIZE-1 downto 0);
    signal reg : reg_t; -- Register bank (latches)

begin

    -- Write Interface
    --=========================================================================
    ack_wr <= xor_reduce(ack_wr_ctrl); -- Generates ack_wr with the correct phase
    ack_wr_o <= ack_wr;

    -- Write Counter
    next_wr_ptr <= req_wr_i XNOR ack_wr; -- Rising edge means that handshake was completed
    write_counter: process(reset_i, next_wr_ptr)
    begin
        if (reset_i = '1') then
            write_ptr <= (others=>'0');
            write_ptr(0) <= '1';
        elsif next_wr_ptr'event and (next_wr_ptr = '1') then
            -- Ring counter
            write_ptr <= write_ptr(BUFFER_DEPTH-2 downto 0) & write_ptr(BUFFER_DEPTH-1) ;
        end if;
    end process;

    -- Write Control
    wr_ctrl: for i in 0 to BUFFER_DEPTH-1 generate
        -- Generate for even-depth FIFO
        wr_ctrl_even: if ((BUFFER_DEPTH rem 2) = 0) generate
```

Figure A.10 - VHDL source code for FIFO module (part 1 of 3).

```
            --  Generating control for even-depth FIFO and even control index
            wr_full_gen_1: if ((i rem 2) = 0) generate
                wr_ctrl_i: entity work.fifo_wr_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => write_ptr(i),
                    req_i          => req_wr_i,
                    ack_o          => ack_wr_ctrl(i),
                    empty_i        => empty(i),
                    full_o         => full(i),
                    phase_select_i => full(i+1),
                    reg_en_o       => reg_en(i)
                );
            end generate wr_full_gen_1;

            -- Generating control for even-depth FIFO and odd control index
            wr_full_gen_2: if ((i rem 2 ) /= 0) generate
               wr_ctrl_i: entity work.fifo_wr_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => write_ptr(i),
                    req_i          => req_wr_i,
                    ack_o          => ack_wr_ctrl(i),
                    empty_i        => empty(i),
                    full_o         => full(i),
                    phase_select_i => full(i-1),
                    reg_en_o       => reg_en(i)
                );
            end generate wr_full_gen_2;
        end generate wr_ctrl_even;

        -- Generate for odd-depth FIFO
        wr_ctrl_odd: if ((BUFFER_DEPTH rem 2) /= 0) generate
            --  Generating control for odd-depth FIFO and even control index
            wr_full_gen_3: if ((i rem 2) = 0) generate
                wr_ctrl_i: entity work.fifo_wr_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => write_ptr(i),
                    req_i          => req_wr_i,
                    ack_o          => ack_wr_ctrl(i),
                    empty_i        => empty(i),
                    full_o         => full(i),
                    phase_select_i => '0',
                    reg_en_o       => reg_en(i)
                );
            end generate wr_full_gen_3;

            -- Generating control for even-depth FIFO and odd control index
            wr_full_gen_4: if ((i rem 2 ) /= 0) generate
               wr_ctrl_i: entity work.fifo_wr_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => write_ptr(i),
                    req_i          => req_wr_i,
                    ack_o          => ack_wr_ctrl(i),
                    empty_i        => empty(i),
                    full_o         => full(i),
                    phase_select_i => '1',
                    reg_en_o       => reg_en(i)
                );
            end generate wr_full_gen_4;
        end generate wr_ctrl_odd;
    end generate wr_ctrl;


    -- Read Interface
    --==========================================================================
    req_rd <= xor_reduce(req_rd_ctrl); -- Generates req_rd with the correct phase
    req_rd_o <= req_rd;

    -- Read Counter
    next_rd_ptr <= ack_rd_i XNOR req_rd; -- Rising edge means that handshake was completed
    read_counter: process(reset_i, next_rd_ptr)
    begin
        if (reset_i = '1') then
            read_ptr <= (others=>'0');
            read_ptr(0) <= '1';
        elsif next_rd_ptr'event and (next_rd_ptr = '1') then
            -- Ring counter
            read_ptr <= read_ptr(BUFFER_DEPTH-2 downto 0) & read_ptr(BUFFER_DEPTH-1) ;
        end if;
    end process;

    -- Read Control
    rd_ctrl: for i in 0 to BUFFER_DEPTH-1 generate
        -- Generate for even-depth FIFO
        rd_ctrl_even: if ((BUFFER_DEPTH rem 2) = 0) generate
            -- Generating control for even-depth FIFO and even control index
            rd_ctrl_gen_1: if ((i rem 2) = 0) generate
                rd_ctrl_i: entity work.fifo_rd_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
```

Figure A.11 - VHDL source code for FIFO module (part 2 of 3).

```
                    en_i           => read_ptr(i),
                    full_i         => full(i),
                    empty_o        => empty(i),
                    phase_select_i => empty(i+1),
                    req_o          => req_rd_ctrl(i),
                    ack_i          => ack_rd_i
                );
            end generate rd_ctrl_gen_1;

            -- Generating control for even-depth FIFO and odd control index
            rd_ctrl_gen_2: if ((i rem 2 ) /= 0) generate
                rd_ctrl_i: entity work.fifo_rd_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => read_ptr(i),
                    full_i         => full(i),
                    empty_o        => empty(i),
                    phase_select_i => empty(i-1),
                    req_o          => req_rd_ctrl(i),
                    ack_i          => ack_rd_i
                );
            end generate rd_ctrl_gen_2;
        end generate rd_ctrl_even;

        -- Generate for odd depth FIFO
        rd_ctrl_odd: if ((BUFFER_DEPTH rem 2) /= 0) generate
            -- Generating control for odd-depth FIFO and even control index
            rd_ctrl_gen_3: if ((i rem 2) = 0) generate
                rd_ctrl_i: entity work.fifo_rd_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => read_ptr(i),
                    full_i         => full(i),
                    empty_o        => empty(i),
                    phase_select_i => '0',
                    req_o          => req_rd_ctrl(i),
                    ack_i          => ack_rd_i
                );
            end generate rd_ctrl_gen_3;

            -- Generating control for odd-depth FIFO and odd control index
            rd_ctrl_gen_4: if ((i rem 2 ) /= 0) generate
                rd_ctrl_i: entity work.fifo_rd_ctrl_wrapper
                port map (
                    reset_i        => reset_i,
                    en_i           => read_ptr(i),
                    full_i         => full(i),
                    empty_o        => empty(i),
                    phase_select_i => '1',
                    req_o          => req_rd_ctrl(i),
                    ack_i          => ack_rd_i
                );
            end generate rd_ctrl_gen_4;
        end generate rd_ctrl_odd;
    end generate rd_ctrl;

    -- Data Path
    --==========================================================================

    -- Data registers (latches)
    reg_gen: for i in 0 to BUFFER_DEPTH-1 generate
        data_reg: process(reg_en(i), data_i)
        begin
            if (reg_en(i) = '1') then
                reg(i) <= data_i;
            end if;
        end process;
    end generate reg_gen;

    -- MUX that selects which register will be read
    reg_mux_gen: for i in 0 to BUFFER_DEPTH-1 generate
        data_o <= reg(i) when read_ptr(i) = '1' else
                  (others => 'Z');
    end generate reg_mux_gen;

end circular;
```

Figure A.12 - VHDL source code for FIFO module (part 3 of 3).

A.3 Input Buffer

```vhdl
--! @file mousetrap_ctrl.vhd
--! @brief Control stage of the Mousetrap pipeline template.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-11-10

--------------------------------------------------------------------------
-- Interface description:
--                             -------------
--                            |             |
--          reset_i---------->|             |
--                            |             |
--            req_i---------->|             |---------->req_o
--                            |             |
--            ack_o<----------|             |<----------ack_i
--                            |             |
--                            |             |---------->en_o
--                            |             |
--                             -------------
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity mousetrap_ctrl is
    port(
        reset_i : in  std_logic; --! Active-high reset signal.
        req_i   : in  std_logic; --! Incoming Request signal.
        ack_o   : out std_logic; --! Acknowledge relative to req_i.
        req_o   : out std_logic; --! Outgoing request signal. Same as req_i, after passing through the latch.
        ack_i   : in  std_logic; --! Acknowledge relative to req_o.
        en_o    : out std_logic  --! Active-high enable signal used to control the latch. May be used to control a data
latch.
    );
end mousetrap_ctrl;

architecture mousetrap_ctrl of mousetrap_ctrl is
    signal req : std_logic; -- req_i after passing through latch.
    signal en  : std_logic; -- Signal to control the latch.

begin
    en <= req XNOR ack_i;
    en_o <= en;
    req_o <= req;
    ack_o <= req;

    -- Latch
    process(reset_i, en, req_i)
    begin
        if (reset_i = '1') then
            req <= '0';
        elsif (en = '1') then
            req <= req_i;
        end if;
    end process;

end mousetrap_ctrl;

```

Figure A.13 - VHDL source code for Mousetrap stage.

```vhdl
--! @file request_splitter.vhd
--! @brief Circuit to split the request signal into req_header and req_data, handling phase-matching and latch-enable signals.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-11-10

------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--        reset_i---------->|             |
--                           |             |
--          req_i---------->|             |---------->req_header_o
--          ack_o<----------|             |<----------ack_header_i
--                           |             |
--                           |             |---------->req_data_o
--last_flit_lvl_i----------->|             |<----------ack_data_i
--                           |             |
--                            -------------
------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity request_splitter is
    port(
        reset_i         : in  std_logic; --! Active-high reset signal.
        req_i           : in  std_logic; --! Request signal.
        ack_o           : out std_logic; --! Acknowledge relative to req_i (ack_header_i xored with ack_data_i).
        req_header_o    : out std_logic; --! Header request signal. The next req will be redirected to req_data_o.
        ack_header_i    : in  std_logic; --! Acknowledge relative to req_header_o.
        req_data_o      : out std_logic; --! Data request signal. This request will be repeatedly made until last_flit_lvl_i becomes high, then the next one will be redirected to req_header_o.
        ack_data_i      : in  std_logic; --! Acknowledge relative to req_data_o.
        last_flit_lvl_i : in  std_logic  --! Signal indicating that the next request to be made is a header request.
    );
end request_splitter;

architecture request_splitter of request_splitter is
    signal req_header_phasematched : std_logic; -- req_i after being phase-matched to req_header.
    signal req_header              : std_logic; -- Same as req_header_o.
    signal req_data_phasematched   : std_logic; -- req_i after being phase-matched to req_data.
    signal req_data                : std_logic; -- Same as req_data_o.
    signal data_hs                 : std_logic; -- Signal that indicates when a data handshake will be performed.
    signal hs_complete             : std_logic; -- Signal that indicates when the handshake was completed).


begin
    ack_o <= ack_header_i XOR ack_data_i;
    req_header_o <= req_header;
    req_data_o <= req_data;


    -- Latch Enable Controller (Low-to-high transition when a handshake is completed)
    hs_complete <= (req_data XNOR req_data_phasematched) when (data_hs = '1') else
                   (req_header XNOR req_header_phasematched);

    process(reset_i, hs_complete)
    begin
        if (reset_i = '1') then
            data_hs <= '0';
        elsif (hs_complete'event and (hs_complete = '1')) then
            data_hs <= not(data_hs and last_flit_lvl_i);
        end if;
    end process;


    -- req_header Latch
    req_header_phasematched <= req_i XOR req_data;

    process(reset_i, data_hs, req_header_phasematched)
    begin
        if (reset_i = '1') then
            req_header <= '0';
        elsif (data_hs = '0') then
            req_header <= req_header_phasematched;
        end if;
    end process;


    -- req_header Latch
    req_data_phasematched <= req_i XOR req_header;

    process(reset_i, data_hs, req_data_phasematched)
    begin
        if (reset_i = '1') then
            req_data <= '0';
        elsif (data_hs = '1') then
            req_data <= req_data_phasematched;
        end if;
    end process;

end request_splitter;
```

Figure A.14 - VHDL source code for request splitter circuit.

```vhdl
--! @file input_buffer_ctrl.vhd
--! @brief Control unit for the input_buffer. All handshake signals use transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! Originally created in 2013-08-06.
--! @date 2013-11-10

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Interface description:
--                           -------------
--                          |             |
--        reset_i---------->|             |
--                          |             |
--          req_i---------->|             |---------->req_header_o
--                          |             |
--          ack_o<----------|             |<----------ack_header_i
--                          |             |
--                          |             |---------->req_data_o
--                          |             |
--                          |             |<----------ack_data_i
--                          |             |
--                FLIT_SIZE |             | FLIT_SIZE
--          data_i====/====>|             |=====/====>data_o
--                          |             |---------->last_flit_o
--                          |             |
--                           -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.hermes_bd_package.all;

entity input_buffer_ctrl is
    port(
        reset_i      : in  std_logic; --! Active-high reset signal.
        req_i        : in  std_logic; --! Request signal indicating that new data is available on data_i.
        ack_o        : out std_logic; --! Acknowledge relative to req_i. Indicates that the data was consumed.
        data_i       : in  std_logic_vector(FLIT_SIZE-1 downto 0); --!  Data input.
        req_header_o : out std_logic; --! Request signal indicating that the flit on data_o is the packet header.
        ack_header_i : in  std_logic; --! Acknowledge signal relative to req_header_o. Indicates that header was consumed.
        req_data_o   : out std_logic; --! Request signal indicating that the flit on data_o is payload.
        ack_data_i   : in  std_logic; --! Acknowledge relative to req_data_o. Indicates that the payload was consumed.
        data_o       : out std_logic_vector(FLIT_SIZE-1 downto 0); --! Data output.
        last_flit_o  : out std_logic  --! Transition-encoded signal indicating that the flit on data_o is the last flit of the current packet.
    );
end input_buffer_ctrl;

architecture input_buffer_ctrl of input_buffer_ctrl is
    signal en            : std_logic; -- Signal that controls the registers in the data path.
    signal req           : std_logic; -- Request signal after passing through the Mousetrap Control.
    signal ack           : std_logic; -- Acknowledge signal after passing through the request_splitter.
    signal data          : std_logic_vector(FLIT_SIZE-1 downto 0); -- Latch that holds data_i.
    signal flit_counter  : std_logic_vector(FLIT_SIZE-1 downto 0); -- Holds how many flits still need to be sent
    signal last_flit_lvl : std_logic; -- Level-encoded active-high signal indicating that the flit on the data latch is the last of the current packet.
    signal size_flit     : std_logic; -- Level-encoded active_high signal indicating that the flit on the data latch has the size of the current packet.
    signal last_flit     : std_logic; -- Same as last_flit_o.


begin
    data_o <= data;

    -- Mousetrap stage at the input
    --===========================================================================
    -- Control Latch
    mousetrap_ctrl_i: entity work.mousetrap_ctrl
    port map (
        reset_i => reset_i,
        req_i   => req_i,
        ack_o   => ack_o,
        req_o   => req,
        ack_i   => ack,
        en_o    => en
    );

    -- Data Latch
    process(reset_i, en, data_i)
    begin
        if (reset_i = '1') then
            data <= (others => '0');
        elsif (en = '1') then
            data <= data_i;
        end if;
    end process;


    -- Request Splitter, to generate req_header_o and req_data_o
```

Figure A.15 - VHDL source code for Input Buffer Control module (part 1 of 2).

```vhdl
    --==========================================================================
    request_splitter_i: entity work.request_splitter
    port map (
        reset_i         => reset_i,
        req_i           => req,
        ack_o           => ack,
        req_header_o    => req_header_o,
        ack_header_i    => ack_header_i,
        req_data_o      => req_data_o,
        ack_data_i      => ack_data_i,
        last_flit_lvl_i => last_flit_lvl
    );


    -- Control State Machine
    --==========================================================================
    -- flit_counter Flip-Flop (Sampled at the request transition, reset by last_flit_lvl)
    process(last_flit_lvl, en)
    begin
        if (last_flit_lvl = '1') then
            flit_counter <= (others => '0');
        elsif (en'event and (en = '0')) then
            if (size_flit = '1') then
                flit_counter <= data_i;
            else
                flit_counter <= flit_counter - 1;
            end if;
        end if;
    end process;


    -- last_flit_lvl Flip-Flop (Sampled at request transition)
    process(reset_i, en)
    begin
        if (reset_i = '1') then
            last_flit_lvl <= '1';
        elsif (en'event and (en = '0')) then
            if (flit_counter = x"1") then
                last_flit_lvl <= '1';
            else
                last_flit_lvl <= '0';
            end if;
        end if;
    end process;


    -- size_flit Flip-Flop (Sampled at request transition
    process(reset_i, en)
    begin
        if (reset_i = '1') then
            size_flit <= '0';
        elsif (en'event and (en = '0')) then
            size_flit <= last_flit_lvl;
        end if;
    end process;


    -- Transition-encoded last_flit_o Generator
    --==========================================================================
    -- last_flit_o Flip-Flop (Sampled at low-to-high transition of last_flit_lvl)
    last_flit_o <= last_flit;

    process(reset_i, last_flit_lvl)
    begin
        if (reset_i = '1') then
            last_flit <= '0';
        elsif (last_flit_lvl'event and (last_flit_lvl = '1')) then
            last_flit <= not(last_flit);
        end if;
    end process;


end input_buffer_ctrl;

```

Figure A.16 - VHDL source code for Input Buffer Control module (part 2 of 2).

```vhdl
--! @file input_buffer_ctrl_wrapper.vhd
--! @brief Wrapper for input_buffer_ctrl
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-08-06

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Interface description:
--                         -------------
--                        |             |
--      reset_i---------->|             |
--                        |             |
--        req_i---------->|             |---------->req_header_o
--                        |             |
--        ack_o<----------|             |<----------ack_header_i
--                        |             |
--                        |             |---------->req_data_o
--                        |             |
--                        |             |<----------ack_data_i
--                        |             |
--              FLIT_SIZE |             | FLIT_SIZE
--        data_i====/====>|             |=====/====>data_o
--                        |             |---------->last_flit_o
--                        |             |
--                         -------------
------------------------------------------------------------------------
-- Wrapper Behavior:
--   0) Output signals related to handshake should be held (ack_o,
--      req_header_o, req_data_o)
--   1) Input handshake starts when req_i /= ack_o.
--   2) Wait for input handshake acknowledge (req_i = ack_o). If stage is
--      empty, this will happen right away.
--   3) Release ack_o after 1 ns. [Simulating the latch hold constraint.
--      These latches in question hold req_i and data_i.]
--   4) Release req_header_o and req_data_o after 4 ns. [Simulating the
--      logic delay of the control circuit.]
--
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use work.hermes_bd_package.all;

entity input_buffer_ctrl_wrapper is
    port(
        reset_i      : in  std_logic; --! Active-high reset signal.
        req_i        : in  std_logic; --! Request signal indicating that new data is available on data_i.
        ack_o        : out std_logic; --! Acknowledge relative to req_i. Indicates that the data was consumed.
        data_i       : in  std_logic_vector(FLIT_SIZE-1 downto 0); --!  Data input.
        req_header_o : out std_logic; --! Request signal indicating that the flit on data_o is the packet header.
        ack_header_i : in  std_logic; --! Acknowledge signal relative to req_header_o. Indicates that header was consumed.
        req_data_o   : out std_logic; --! Request signal indicating that the flit on data_o is payload.
        ack_data_i   : in  std_logic; --! Acknowledge relative to req_data_o. Indicates that the payload was consumed.
        data_o       : out std_logic_vector(FLIT_SIZE-1 downto 0); --! Data output.
        last_flit_o  : out std_logic  --! Active-high signal indicating that the flit on data_o is the last flit of the current packet.
    );
end input_buffer_ctrl_wrapper;

architecture sim of input_buffer_ctrl_wrapper is
     -- Control signals
    signal hold_ack_o        : std_logic;
    signal hold_req_header_o : std_logic;
    signal hold_req_data_o   : std_logic;

    -- Aux signals
    signal aux_ack_o        : std_logic;
    signal aux_req_header_o : std_logic;
    signal aux_req_data_o   : std_logic;

begin

    -- Instance of input_buffer_ctrl
    input_buffer_ctrl_i: entity work.input_buffer_ctrl
    port map (
        reset_i      => reset_i,
        req_i        => req_i,
        ack_o        => aux_ack_o,
        data_i       => data_i,
        req_header_o => aux_req_header_o,
        ack_header_i => ack_header_i,
        req_data_o   => aux_req_data_o,
        ack_data_i   => ack_data_i,
        data_o       => data_o,
        last_flit_o  => last_flit_o
    );

    -- Control Logic
    control: process
    begin
```

Figure A.17 - VHDL source code for simulation wrapper of Input Buffer Control (part 1 of 2).

```vhdl
        -- Initializing control signals
        hold_ack_o <= '0';
          hold_req_header_o <= '0';
          hold_req_data_o <= '0';
        wait until reset_i = '0';
        loop
            -- Hold all signals
            hold_ack_o <= '1';
            hold_req_header_o <= '1';
                hold_req_data_o <= '1';

            -- Wait for req_i request
            if (req_i = aux_ack_o) then
                wait until (req_i /= aux_ack_o);
            end if;

                  -- Wait for ack_o acknowledge. (Mousetrap stage generates ACK right away)
                  if (req_i /= aux_ack_o) then
                wait until (req_i = aux_ack_o);
            end if;

            -- Release #1: ack_o, after 1 ns (latch setup constraint)
            wait for 1 ns;
            hold_ack_o <= '0';

                  -- Release #2, req_header_o and req_data_o, after 4 ns (ctrl logic delay)
                wait for 4 ns;
                hold_req_header_o <= '0';
                  hold_req_data_o <= '0';

            -- Wait 1 ns before holding it again
            wait for 1 ns;
        end loop;
    end process;


    -- Latches to hold signals
    latch_ack_o: process(reset_i, hold_ack_o, aux_ack_o)
    begin
        if ((reset_i = '1') OR (hold_ack_o = '0')) then
            ack_o <= aux_ack_o;
        end if;
    end process;

    latch_req_header_o: process(reset_i, hold_req_header_o, aux_req_header_o)
    begin
        if ((reset_i = '1') OR (hold_req_header_o = '0')) then
            req_header_o <= aux_req_header_o;
        end if;
    end process;

    latch_req_data_o: process(reset_i, hold_req_data_o, aux_req_data_o)
    begin
        if ((reset_i = '1') OR (hold_req_data_o = '0')) then
            req_data_o <= aux_req_data_o;
        end if;
    end process;

end sim;
```

Figure A.18 - VHDL source code for simulation wrapper of Input Buffer Control (part 2 of 2).

```vhdl
--! @file input_buffer.vhd
--! @brief Handshake-based Input Buffer using transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-07-13

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--  BUFFER_DEPTH: Defines the depth of the buffer
------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--         reset_i---------->|             |
--                           |             |
--           req_i---------->|             |---------->req_header_o
--                           |             |
--           ack_o<----------|             |<----------ack_header_i
--                           |             |
--                           |             |---------->req_data_o
--                           |             |
--                           |             |<----------ack_data_i
--                           |             |
--                 FLIT_SIZE |             | FLIT_SIZE
--           data_i====/====>|             |=====/====>data_o
--                           |             |---------->last_flit_o
--                           |             |
--                            -------------
------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use work.hermes_bd_package.all;

entity input_buffer is
    generic(
        BUFFER_DEPTH : integer := 8 --! Defines the depth of the buffer.
    );
    port(
        reset_i      : in  std_logic; --! Active-high reset signal.
        req_i        : in  std_logic; --! Request signal indicating that new data is available to be written.
        ack_o        : out std_logic; --! Acknowledge signal relative to req_i. Indicates that the data has been stored.
        data_i       : in  std_logic_vector(FLIT_SIZE-1 downto 0); --! Data input.
        req_header_o : out std_logic; --! Request signal indicating that the flit on data_o is the packet header.
        ack_header_i : in  std_logic; --! Acknowledge signal relative to req_header_o. Indicates that header was consumed.
        req_data_o   : out std_logic; --! Request signal indicating that the flit on data_o is payload.
        ack_data_i   : in  std_logic; --! Acknowledge signal relative to req_data_o. Indicates that the payload was consumed.
        data_o       : out std_logic_vector(FLIT_SIZE-1 downto 0); --! Data output.
        last_flit_o  : out std_logic --! signal indicating that the flit on data_o is the last flit of the current packet.
    );
end input_buffer;

architecture input_buffer of input_buffer is
    signal data : std_logic_vector(FLIT_SIZE-1 downto 0); -- Signal that holds the data coming out of the buffer
    signal req_rd : std_logic; -- Request signal from buffer (new data to be read)
    signal ack_rd : std_logic; -- Acknowledge relative to req_rd (data was read)

begin

    -- Buffer Instantiation
    fifo_i: entity work.fifo
    generic map(
        BUFFER_DEPTH => BUFFER_DEPTH
    )
    port map(
        reset_i  => reset_i,
        req_wr_i => req_i,
        ack_wr_o => ack_o,
        data_i   => data_i,
        req_rd_o => req_rd,
        ack_rd_i => ack_rd,
        data_o   => data
    );

    -- Controller Instantiation
    buffer_ctrl_i: entity work.input_buffer_ctrl_wrapper
    port map (
        reset_i      => reset_i,
        req_i        => req_rd,
        ack_o        => ack_rd,
        data_i       => data,
        req_header_o => req_header_o,
        ack_header_i => ack_header_i,
        req_data_o   => req_data_o,
        ack_data_i   => ack_data_i,
        data_o       => data_o,
        last_flit_o  => last_flit_o
    );

end input_buffer;
```

Figure A.19 - VHDL source code for Input Buffer module.

A.4 Routing Control

```vhdl
--! @file routing_ctrl.vhd
--! @brief Control unit for the input router. All handshake signals use transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-08-09

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--   ROUTING_ALGORITHM: Currently only XY
--      ROUTER_ADDRESS: Address of the router in which this port is inserted.
--           THIS_PORT: Defines which port is this interface representing (East, West, North, South, Local).
--          COMM_PORTS: Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
------------------------------------------------------------------------
-- Interface description:
--                             -------------
--                            |             |
--        reset_i----------->|             |
--                            |             |PORTS_NUMBER
--    req_route_i----------->|             |=====/====>req_outport_o
--                            |             |
--    ack_route_o<-----------|             |PORTS_NUMBER
--                            |             |<=====/====ack_outport_i
--                FLIT_SIZE/2|             |
--target_address_i====/====>|             |
--                            |             |
--                             -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use work.hermes_bd_package.all;

entity routing_ctrl is
    generic(
        ROUTING_ALGORITHM : string := "XY"; --! Choice of routing algorithm.
        ROUTER_ADDRESS    : std_logic_vector((FLIT_SIZE/2)-1 downto 0) := x"11"; --! Address of the router.
        THIS_PORT         : integer := LOCAL; --! Defines which port is this interface representing  (East, West, North, South, Local).
        COMM_PORTS        : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
    );
    port(
        reset_i          : in  std_logic; --! Active-high reset signal.
        req_route_i      : in  std_logic; --! Signal requesting a route towards the target_address_i router.
        ack_route_o      : out std_logic; --! Acknowledge relative to req_route_i. Indicates that the outport towards the destination is ready to receive data.
        target_address_i : in  std_logic_vector((FLIT_SIZE/2)-1 downto 0); --! Address of the destination router.
        req_outport_o    : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting the use of an output port.
        ack_outport_i    : in  std_logic_vector(PORTS_NUMBER-1 downto 0)  --! Acknowledge signal relative to req_outport_i. Indicates that the port use has been granted.
    );
end routing_ctrl;

architecture routing_ctrl of routing_ctrl is
    signal outport_en  : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Signal to enable a given port's handshake
    signal req_outport : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Used to phase-match req_route_i with each req_outport_o
    signal ack_route   : std_logic; -- Same as ack_route_o
    signal routing_en  : std_logic; -- Signal to enable all handshakes (Request has been made, but hasn't received acknowledge yet)


begin
    routing_en <= req_route_i XOR ack_route; -- Enables the computation of the output port; High when handshake hasn't been completed yet
    ack_route <= xor_all_bits_considering_comm_ports(ack_outport_i, COMM_PORTS, THIS_PORT); -- Generates ack_route with the correct phase
    ack_route_o <= ack_route;

    -- Routing algorithm
    xy_routing_i : if (ROUTING_ALGORITHM = "XY") generate
        routing_unit_i : entity work.routing_unit(XY)
        generic map (
            ROUTER_ADDRESS => ROUTER_ADDRESS,
            THIS_PORT      => THIS_PORT,
            COMM_PORTS     => COMM_PORTS
        )
        port map (
            en_i             => routing_en,
            target_address_i => target_address_i,
            req_outport_o    => outport_en
        );
    end generate xy_routing_i;


    -- Latches and phase-matchers for req_outport_o
    req_outport_ctrl_gen: for i in 0 to PORTS_NUMBER-1 generate

```

Figure A.20 - VHDL source code for Routing Unit module.

```vhdl
--! @file routing_ctrl.vhd
--! @brief Control unit for the input router. All handshake signals use transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-08-09

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--   ROUTING_ALGORITHM: Currently only XY
--      ROUTER_ADDRESS: Address of the router in which this port is inserted.
--           THIS_PORT: Defines which port is this interface representing (East, West, North, South, Local).
--          COMM_PORTS: Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
------------------------------------------------------------------------
-- Interface description:
--                           -------------
--                          |             |
--        reset_i---------->|             |
--                          |             |PORTS_NUMBER
--    req_route_i---------->|             |=====/====>req_outport_o
--                          |             |
--    ack_route_o<----------|             |PORTS_NUMBER
--                          |             |<=====/====ack_outport_i
--               FLIT_SIZE/2|             |
--target_address_i====/====>|             |
--                          |             |
--                           -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use work.hermes_bd_package.all;

entity routing_ctrl is
    generic(
        ROUTING_ALGORITHM : string := "XY"; --! Choice of routing algorithm.
        ROUTER_ADDRESS    : std_logic_vector((FLIT_SIZE/2)-1 downto 0) := x"11"; --! Address of the router.
        THIS_PORT         : integer := LOCAL; --! Defines which port is this interface representing  (East, West, North, South, Local).
        COMM_PORTS        : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
    );
    port(
        reset_i          : in  std_logic; --! Active-high reset signal.
        req_route_i      : in  std_logic; --! Signal requesting a route towards the target_address_i router.
        ack_route_o      : out std_logic; --! Acknowledge relative to req_route_i. Indicates that the outport towards the destination is ready to receive data.
        target_address_i : in  std_logic_vector((FLIT_SIZE/2)-1 downto 0); --! Address of the destination router.
        req_outport_o    : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting the use of an output port.
        ack_outport_i    : in  std_logic_vector(PORTS_NUMBER-1 downto 0)  --! Acknowledge signal relative to req_outport_i. Indicates that the port use has been granted.
    );
end routing_ctrl;

architecture routing_ctrl of routing_ctrl is
    signal outport_en  : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Signal to enable a given port's handshake
    signal req_outport : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Used to phase-match req_route_i with each req_outport_o
    signal ack_route   : std_logic; -- Same as ack_route_o
    signal routing_en  : std_logic; -- Signal to enable all handshakes (Request has been made, but hasn't received acknowledge yet)


begin
    routing_en <= req_route_i XOR ack_route; -- Enables the computation of the output port; High when handshake hasn't been completed yet
    ack_route <= xor_all_bits_considering_comm_ports(ack_outport_i, COMM_PORTS, THIS_PORT); -- Generates ack_route with the correct phase
    ack_route_o <= ack_route;

    -- Routing algorithm
    xy_routing_i : if (ROUTING_ALGORITHM = "XY") generate
        routing_unit_i : entity work.routing_unit(XY)
        generic map (
            ROUTER_ADDRESS => ROUTER_ADDRESS,
            THIS_PORT      => THIS_PORT,
            COMM_PORTS     => COMM_PORTS
        )
        port map (
            en_i             => routing_en,
            target_address_i => target_address_i,
            req_outport_o    => outport_en
        );
    end generate xy_routing_i;


    -- Latches and phase-matchers for req_outport_o
    req_outport_ctrl_gen: for i in 0 to PORTS_NUMBER-1 generate

```

Figure A.21 - VHDL source code for Routing Control module (part 1 of 2).

```vhdl
        -- Test if there will be communication with port i
        used_req_outport_ctrl_gen:if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) generate

            -- Phase-matcher [req_route_i XORed with all ack_outport_i but ack_outport_i(i)]
            req_outport(i) <= req_route_i XOR (xor_all_bits_but_one_considering_comm_ports(ack_outport_i,i, COMM_PORTS,
THIS_PORT));

            -- Latch
            req_outport_latch_i: process(reset_i, outport_en, req_outport(i))
            begin
                if (reset_i = '1') then
                    req_outport_o(i) <= '0';
                elsif (outport_en(i) = '1') then
                    req_outport_o(i) <= req_outport(i);
                end if;
            end process;
        end generate;
    end generate;

end routing_ctrl;
```

Figure A.22 - VHDL source code for Routing Control module (part 2 of 2).

```vhdl
--! @file routing_ctrl_wrapper.vhd
--! @brief Wrapper for routing_ctrl
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-08-09

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--   ROUTING_ALGORITHM: Currently only XY
------------------------------------------------------------------------
-- Interface description:
--                              -------------
--                             |             |
--         reset_i---------->|             |
--                             |             |PORTS_NUMBER
--     req_route_i---------->|             |=====/====>req_outport_o
--                             |             |
--     ack_route_o<----------|             |PORTS_NUMBER
--                             |             |<=====/====ack_outport_i
--                             |             |
--                 FLIT_SIZE/2|             |
--router_address_i====/====>|             |
--                             |             |
--                 FLIT_SIZE/2|             |
--target_address_i====/====>|             |
--                             |             |
--                              -------------
------------------------------------------------------------------------
-- Wrapper Behavior:
--   0) Output signals related to handshake should be held (ack_o,
--      req_header_o, req_data_o)
--   1) Input handshake starts when req_i /= ack_o.
--   2) Wait for input handshake acknowledge (req_i = ack_o). If stage is
--      empty, this will happen right away.
--   3) Release ack_o after 1 ns. [Simulating the latch hold constraint.
--      These latches in question hold req_i and data_i.]
--   4) Release req_header_o and req_data_o after 4 ns. [Simulating the
--      logic delay of the control circuit.]
--
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use work.hermes_bd_package.all;

entity routing_ctrl_wrapper is
    generic(
        ROUTING_ALGORITHM : string := "XY"; --! Choice of routing algorithm.
        ROUTER_ADDRESS    : std_logic_vector((FLIT_SIZE/2)-1 downto 0) := x"11"; --! Address of the router.
        THIS_PORT         : integer := LOCAL; --! Defines which port is this interface representing  (East, West, North,
South, Local).
        COMM_PORTS        : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines to which ports this interface
connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
    );
    port(
        reset_i          : in  std_logic; --! Active-high reset signal.
        req_route_i      : in  std_logic; --! Signal requesting a route towards the target_address_i router.
        ack_route_o      : out std_logic; --! Acknowledge relative to req_route_i. Indicates that the outport towards the
destination is ready to receive data.
        target_address_i : in  std_logic_vector((FLIT_SIZE/2)-1 downto 0); --! Address of the destination router.
        req_outport_o    : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting the use of an output port.
        ack_outport_i    : in  std_logic_vector(PORTS_NUMBER-1 downto 0)  --! Acknowledge signal relative to
req_outport_i. Indicates that the port use has been granted.
    );
end routing_ctrl_wrapper;
```

Figure A.23 - VHDL source code for simulation wrapper of Routing Control (part 1 of 2).

```vhdl
architecture sim of routing_ctrl_wrapper is
     -- Control signals
    signal hold_ack_route_o   : std_logic;
    signal hold_req_outport_o : std_logic;

    -- Aux signals
    signal aux_ack_route_o  : std_logic;
    signal aux_req_outport_o : std_logic_vector(PORTS_NUMBER-1 downto 0);

begin

    -- Instance of routing_ctrl
    routing_ctrl_i: entity work.routing_ctrl
    generic map(
        ROUTING_ALGORITHM => ROUTING_ALGORITHM,
        ROUTER_ADDRESS    => ROUTER_ADDRESS,
        THIS_PORT         => THIS_PORT,
        COMM_PORTS        => COMM_PORTS
    )
    port map (
        reset_i          => reset_i,
        req_route_i      => req_route_i,
        ack_route_o      => aux_ack_route_o,
        target_address_i => target_address_i,
        req_outport_o    => aux_req_outport_o,
        ack_outport_i    => ack_outport_i
    );

    -- Control Logic
    control: process
    begin
        -- Initializing control signals
        hold_ack_route_o <= '0';
        hold_req_outport_o <= '0';
        wait until reset_i = '0';
        loop
            -- Hold all signals
            hold_ack_route_o <= '1';
            hold_req_outport_o <= '1';

            -- Wait for req_route_i request
            if (req_route_i = aux_ack_route_o) then
                wait until (req_route_i /= aux_ack_route_o);
            end if;

            -- Release #1: req_outport_o, after 4 ns (routing logic delay)
            wait for 4 ns;
            hold_req_outport_o <= '0';

            -- Wait for ack_outport_i acknowledge.
            if (req_route_i /= aux_ack_route_o) then
                wait until (req_route_i = aux_ack_route_o);
            end if;

            -- Release #2, ack_route_o, after 1 ns (phase-matching delay)
            wait for 1 ns;
            hold_ack_route_o <= '0';

            -- Wait 1 ns before holding it again
            wait for 1 ns;
        end loop;
    end process;


    -- Latches to hold signals
    latch_ack_route_o: process(reset_i, hold_ack_route_o, aux_ack_route_o)
    begin
        if ((reset_i = '1') OR (hold_ack_route_o = '0')) then
            ack_route_o <= aux_ack_route_o;
        end if;
    end process;

    latch_req_outport_o: process(reset_i, hold_req_outport_o, aux_req_outport_o)
    begin
        if ((reset_i = '1') OR (hold_req_outport_o = '0')) then
            req_outport_o <= aux_req_outport_o;
        end if;
    end process;

end sim;
```

Figure A.24 - VHDL source code for simulation wrapper of Routing Control (part 2 of 2).

A.5 Input Interface

```vhdl
--! @file input_interface.vhd
--! @brief Handshake-based Input Interface using transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-08-09

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--  ROUTING_ALGORITHM: Choice of routing algorithm.
--       BUFFER_DEPTH: Defines the depth of the buffer.
--     ROUTER_ADDRESS: Address of the router in which this port is inserted.
--          THIS_PORT: Defines which port is this interface representing (East, West, North, South, Local).
--         COMM_PORTS: Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--         reset_i---------->|             |
--                           |             |PORTS_NUMBER
--           req_i---------->|             |======/===>req_outport_o
--                           |             |PORTS_NUMBER
--           ack_o<----------|             |<=====/====ack_outport_i
--                           |             |
--                           |             |---------->req_data_o
--                           |             |PORTS_NUMBER
--                           |             |<=====/====ack_data_i
--                           |             |
--                 FLIT_SIZE |             | FLIT_SIZE
--           data_i====/====>|             |=====/====>data_o
--                           |             |---------->last_flit_o
--                           |             |
--                            -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use work.hermes_bd_package.all;

entity input_interface is
    generic(
        ROUTING_ALGORITHM : string := "XY"; --! Choice of routing algorithm.
        BUFFER_DEPTH      : integer := 8; --! Defines the depth of the buffer.
        ROUTER_ADDRESS    : std_logic_vector((FLIT_SIZE/2)-1 downto 0) := x"11"; --! Address of the router.
        THIS_PORT         : integer := LOCAL; --! Defines which port is this interface representing  (East, West, North, South, Local).
        COMM_PORTS        : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
    );
    port(
        reset_i       : in  std_logic; --! Active-high reset signal.
        req_i         : in  std_logic; --! Request signal indicating that new data is available on data_i.
        ack_o         : out std_logic; --! Acknowledge signal relative to req_i. Indicates that the data has been stored.
        data_i        : in  std_logic_vector(FLIT_SIZE-1 downto 0); --! Data input.
        req_outport_o : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting use of an output port.
        ack_outport_i : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge signal relative to req_outport_o. Indicates that the port use has been granted, and the flit on data_o has been consumed.
        req_data_o    : out std_logic; --! Request signal indicating that new payload is available on data_o.
        ack_data_i    : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge relative to req_data_o. Indicates that the payload was consumed.
        data_o        : out std_logic_vector(FLIT_SIZE-1 downto 0); --! Data output.
        last_flit_o   : out std_logic --! Active-high signal indicating that the flit on data_o is the last flit of the current packet.
    );
end input_interface;

architecture input_interface of input_interface is
    signal data       : std_logic_vector(FLIT_SIZE-1 downto 0); -- Same as data_o
    signal req_header : std_logic; -- Signal requesting a route towards the address on data_i.
    signal ack_header : std_logic; -- Acknowledge from req_header. The routing is completed, and data_o has been stored
    signal ack_data   : std_logic; -- Acknowledge from req_data (same as all ack_data_i XORed)


begin
    data_o <= data;

    -- XOR of all relevant ack_data_i
    ack_data <= xor_all_bits_considering_comm_ports(ack_data_i, COMM_PORTS, THIS_PORT);

    -- Buffer Instantiation
    input_buffer_i: entity work.input_buffer
    generic map(
        BUFFER_DEPTH => BUFFER_DEPTH
    )
    port map(
        reset_i      => reset_i,
        req_i        => req_i,
        ack_o        => ack_o,
        data_i       => data_i,
```

Figure A.25 - VHDL source code for Input Interface module (part 1 of 2).

```vhdl
        req_header_o => req_header,
        ack_header_i => ack_header,
        req_data_o   => req_data_o,
        ack_data_i   => ack_data,
        data_o       => data,
        last_flit_o  => last_flit_o
    );

    -- Routing control
    routing_ctrl_i: entity work.routing_ctrl_wrapper
    generic map(
        ROUTING_ALGORITHM => ROUTING_ALGORITHM,
        ROUTER_ADDRESS    => ROUTER_ADDRESS,
        THIS_PORT         => THIS_PORT,
        COMM_PORTS        => COMM_PORTS
    )
    port map(
        reset_i          => reset_i,
        req_route_i      => req_header,
        ack_route_o      => ack_header,
        target_address_i => data((FLIT_SIZE/2)-1 downto 0),
        req_outport_o    => req_outport_o,
        ack_outport_i    => ack_outport_i
    );

end input_interface;

```

Figure A.26 - VHDL source code for Input Interface module (part 2 of 2).

A.6 Outport Control

```vhdl
--! @file outport_ctrl.vhd
--! @brief Control unit for each outport (Phase converter between input port and output port).
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-10-02

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Interface description:
--                           -------------
--                          |             |
--        reset_i---------->|             |
--                          |             |
--  req_outport_i---------->|             |---------->arbiter_request_o
--                          |             |
--  ack_outport_o<----------|             |<----------arbiter_grant_i
--                          |             |
--                          |             |
--      req_data_i--------->|             |---------->req_o
--                          |             |
--      ack_data_o<---------|             |<----------ack_i
--                          |             |
--                          |             |
--     last_flit_i--------->|             |
--                          |             |
--                           -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use work.hermes_bd_package.all;

entity outport_ctrl is
    port(
        reset_i           : in  std_logic; --! Active-high reset signal.
        req_outport_i     : in  std_logic; --! Signal requesting use of the outport.
        ack_outport_o     : out std_logic; --! Acknowledge relative to req_outport_i. Indicates that port use has been granted and the data at data out port was received.
        arbiter_request_o : out std_logic; --! Level-encoded signal requesting use of the outport.
        arbiter_grant_i   : in  std_logic; --! Level-encoded grant relative to arbiter_request_o. This signal is kept high as long as the arbiter_request_o is asserted.
        req_data_i        : in  std_logic; --! Signal indicating that a new data flit is ready to be sent.
        ack_data_o        : out std_logic; --! Acknowledge relative to req_data_i. Indicates that the data flit has been sent.
        last_flit_i       : in  std_logic; --! Transition-encoded signal indicating that the current flit is the last flit of the packet.
        req_o             : out std_logic; --! Signal indicating that there is a new flit ready to be sent.
        ack_i             : in  std_logic  --! Acknowledge relative to req_o. Indicates that the flit has been sent.
    );
end outport_ctrl;

architecture outport_ctrl of outport_ctrl is
    signal outport_request : std_logic; -- Level-encoded signal that indicates when a request at [req/ack]_outport_i is detected
    signal ack_outport     : std_logic; -- Same as ack_outport_o
    signal req_data        : std_logic; -- Phase-matched version of req_data_i
    signal req_data_phase  : std_logic; -- Signal to indicate when the phase of req_data_ needs to be inverted
    signal ack_data        : std_logic; -- Same as ack_data_o
```

Figure A.27 - VHDL source code for Outport Control module (part 1 of 3).

```
    signal data_request    : std_logic; -- Level-encoded signal that indicates when a request at [req/ack])_data_i is
detected
    signal last_flit       : std_logic; -- last_flit_i sampled at the acknowledge of [req/ack]_data handshake
    signal last_flit_phase : std_logic; -- Sample of last_flit_i taken when abriter_grant is given
    signal arbiter_grant   : std_logic; -- arbiter_grant_i "filtered", to avoid a race condition (guarantees that
arbiter_grant_i goes low when arbiter_request goes low)
    signal arbiter_request : std_logic; -- Same as arbiter_request_o
    signal data_hs         : std_logic; -- Signal used to indicate when a data handshake can take place. (bec

begin
    ack_outport_o <= ack_outport;
    ack_data_o <= ack_data;
    arbiter_request_o <= arbiter_request;

    outport_request <= req_outport_i XOR ack_outport; -- High when request has been made, but ack hasn't been issued yet
    data_request <= req_data XOR ack_data; -- High when request has been made, but ack hasn't been issued yet
    arbiter_grant <= arbiter_grant_i AND arbiter_request; -- Part of a race-condition avoidance mechanism

    data_hs <= arbiter_grant AND not(outport_request); -- High after [req/ack]_outport handshake takes place

    -- arbiter_request latch
    process (reset_i, arbiter_grant_i, outport_request, last_flit)
    begin
        if ((reset_i = '1') or (last_flit = '1')) then
            arbiter_request <= '0';
        elsif ((arbiter_grant_i = '0') AND (outport_request = '1')) then
            arbiter_request <= '1';
        end if;
    end process;


    -- req_o latch
    process (reset_i, arbiter_grant, req_outport_i, req_data)
    begin
        if (reset_i = '1') then
            req_o <= '0';
        elsif (arbiter_grant = '1') then
            req_o <= req_outport_i XOR req_data; -- Phase-matching of req_o
        end if;
    end process;


    -- ack_outport latch
    process (reset_i, arbiter_grant, outport_request, ack_i, req_data)
    begin
        if (reset_i = '1') then
            ack_outport <= '0';
        elsif ((arbiter_grant = '1') AND (outport_request = '1')) then
            ack_outport <= ack_i XOR req_data; -- Phase-matching of ack_outport
        end if;
    end process;


    -- req_data latch
    process (reset_i, arbiter_grant, outport_request, req_data_i, req_data_phase)
    begin
        if (reset_i = '1') then
            req_data <= '0';
        elsif (data_hs = '1') then
            req_data <= req_data_i XOR req_data_phase; -- Phase-matches req_data
        end if;
    end process;


    -- req_data_phase flip-flop
    process (arbiter_grant)
    begin
        if (arbiter_grant'event AND (arbiter_grant = '1')) then
            req_data_phase <= req_data_i XOR req_data; -- High if they are different
        end if;
    end process;


    -- ack_data latch
    process (reset_i, data_request, ack_i, req_outport_i)
    begin
        if (reset_i = '1') then
            ack_data <= '0';
        elsif (data_request = '1') then
            ack_data <= ack_i XOR req_outport_i; -- Phase-matches ack_data
        end if;
    end process;


    -- last_flit flip-flop
    process(reset_i, data_hs, last_flit_i, data_request)
    begin
        if ((reset_i = '1') OR (data_hs = '0')) then
            last_flit <= '0';
        elsif (data_request'event AND (data_request = '0')) then
            last_flit <= last_flit_i XOR last_flit_phase; -- Phase-matches last_flit
        end if;
    end process;
```

Figure A.28 - VHDL source code for Outport Control module (part 2 of 3).

A.7 Arbiter

```vhdl
--! @file arbiter.vhd
--! @brief Arbiter for output interface. Based on design proposed by Ghiribaldi et. al.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-10-03

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--                   SIZE    |             |    SIZE
--        request_i=====/====>|             |=====/====>grant_o
--                           |             |
--                           |             |
--                            -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use work.hermes_bd_package.all;

entity arbiter is
    generic(
        SIZE : integer := 4 --! Number of requests supported (max. 4).
    );
    port(
        request_i : in  std_logic_vector(SIZE-1 downto 0); --! Vector of level-encoded requests.
        grant_o   : out std_logic_vector(SIZE-1 downto 0)  --! Vector of grants, based of the requests made in request_i.
    );
end arbiter;

architecture arbiter of arbiter is
    -- Declaration of MUTEX implemented in ascend_behavioral.v and ascend_behavioral_udps.v
    component MUTEX2
        port (
            RA : in  std_logic;
            RB : in  std_logic;
            AA : out std_logic;
            AB : out std_logic
        );
    end component;

    -- Declaration of C-Element implemented in ascend_behavioral.v and ascend_behavioral_udps.v
    component GPSVT_BP_CSYM2X1
        port (
            Q : out std_logic;
            A : in  std_logic;
            B : in  std_logic
        );
    end component;


begin

    -- Arbiter with 1 input (Just to make it very generic)
    arb1: if (SIZE = 1) generate
        grant_o <= request_i;
    end generate arb1;


    -- Arbiter with 2 inputs
    arb2: if (SIZE = 2) generate
        -- Instanciation of Mutex
        mutex_1: MUTEX2
        port map (
            RA => request_i(0),
            RB => request_i(1),
            AA => grant_o(0),
            AB => grant_o(1)
        );
    end generate arb2;


    -- Arbiter with 3 inputs
    arb3: if (SIZE = 3) generate
        signal grant       : std_logic_vector(1 downto 0);
        signal request     : std_logic_vector(1 downto 0);
        signal grant_mutex : std_logic_vector(3 downto 0);
        signal mutex_2_ra  : std_logic;

    begin
        grant_o(1 downto 0) <= grant(1 downto 0);
        grant_o(2) <= grant_mutex(3);


        request(0) <= request_i(0) AND not(grant(1));
        request(1) <= request_i(1) AND not(grant(0));
        mutex_2_ra <= request(0) OR request(1);
```

Figure A.29 - VHDL source code for Arbiter module (part 1 of 3).

```vhdl

        -- Instanciation of Mutexes
        mutex_1: MUTEX2
        port map (
            RA => request(0),
            RB => request(1),
            AA => grant_mutex(0),
            AB => grant_mutex(1)
        );

        mutex_2: MUTEX2
        port map (
            RA => mutex_2_ra,
            RB => request_i(2),
            AA => grant_mutex(2),
            AB => grant_mutex(3)
        );

        -- Instanciation of C-Elements
        cel_1: GPSVT_BP_CSYM2X1
        port map (
            Q => grant(0),
            A => grant_mutex(0),
            B => grant_mutex(2)
        );

        cel_2: GPSVT_BP_CSYM2X1
        port map (
            Q => grant(1),
            A => grant_mutex(1),
            B => grant_mutex(2)
        );

    end generate arb3;


    -- Arbiter with 4 inputs
    arb4: if (SIZE = 4) generate
        signal grant       : std_logic_vector(3 downto 0);
        signal request     : std_logic_vector(3 downto 0);
        signal grant_mutex : std_logic_vector(5 downto 0);
        signal mutex_3_ra  : std_logic;
        signal mutex_3_rb  : std_logic;

    begin
        grant_o <= grant;

        request(0) <= request_i(0) AND not(grant(1));
        request(1) <= request_i(1) AND not(grant(0));
        request(2) <= request_i(2) AND not(grant(3));
        request(3) <= request_i(3) AND not(grant(2));
        mutex_3_ra <= request(0) OR request(1);
        mutex_3_rb <= request(2) OR request(3);

        -- Instanciation of Mutexes
        mutex_1: MUTEX2
        port map (
            RA => request(0),
            RB => request(1),
            AA => grant_mutex(0),
            AB => grant_mutex(1)
        );

        mutex_2: MUTEX2
        port map (
            RA => request(2),
            RB => request(3),
            AA => grant_mutex(2),
            AB => grant_mutex(3)
        );

        mutex_3: MUTEX2
        port map (
            RA => mutex_3_ra,
            RB => mutex_3_rb,
            AA => grant_mutex(4),
            AB => grant_mutex(5)
        );

        -- Instanciation of C-Elements
        cel_1: GPSVT_BP_CSYM2X1
        port map (
            Q => grant(0),
            A => grant_mutex(0),
            B => grant_mutex(4)
        );

        cel_2: GPSVT_BP_CSYM2X1
        port map (
            Q => grant(1),
            A => grant_mutex(1),
            B => grant_mutex(4)
        );

        cel_3: GPSVT_BP_CSYM2X1
```

Figure A.30 - VHDL source code for Arbiter module (part 2 of 3).

```
        port map (
            Q => grant(2),
            A => grant_mutex(2),
            B => grant_mutex(5)
        );

        cel_4: GPSVT_BP_CSYM2X1
        port map (
            Q => grant(3),
            A => grant_mutex(3),
            B => grant_mutex(5)
        );
    end generate arb4;

end arbiter;

```

Figure A.31 - VHDL source code for Arbiter module (part 3 of 3).

A.8 Output Interface

```
--! @file output_interface.vhd
--! @brief Handshake-based Output Interface using transition signaling protocol.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-10-07

------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
------------------------------------------------------------------------
-- Generics for configuration:
--   THIS_PORT: Defines which port is this interface representing (East, West, North, South, Local).
--  COMM_PORTS: Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
------------------------------------------------------------------------
-- Interface description:
--                           -------------
--                          |             |
--        reset_i---------->|             |
--                          |             |
--              PORTS_NUMBER|             |
--  req_outport_i======/===>|             |
--              PORTS_NUMBER|             |
--  ack_outport_o<=====/====|             |
--                          |             |
--              PORTS_NUMBER|             |
--     req_data_i=====/====>|             |---------->req_o
--              PORTS_NUMBER|             |
--     ack_data_o<=====/====|             |<----------ack_i
--                          |             |
--  PORTS_NUMBER x FLIT_SIZE|             | FLIT_SIZE
--         data_i=====/====>|             |=====/====>data_o
--              PORTS_NUMBER|             |
--    last_flit_i=====/====>|             |
--                          |             |
--                           -------------
------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use work.hermes_bd_package.all;

entity output_interface is
    generic(
        THIS_PORT  : integer := LOCAL; --! Defines which port is this interface representing  (East, West, North, South, Local).
        COMM_PORTS : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
    );
    port(
        reset_i       : in  std_logic; --! Active-high reset signal.
        req_outport_i : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting use of the output port.
        ack_outport_o : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge signal relative to req_outport_i. Indicates that the port use has been granted, and the flit on data_i has been consumed.
        req_data_i    : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Request signal indicating that new payload is available on data_i.
        ack_data_o    : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge relative to req_data_i. Indicates that the payload was consumed.
        data_i        : in  flit_size_array(PORTS_NUMBER-1 downto 0); --! Data input.
        last_flit_i   : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Active-high signal indicating that the flit on data_i is the last flit of the current packet.
        req_o         : out std_logic; --! Request signal indicating that new data is available on data_o.
        ack_i         : in  std_logic; --! Acknowledge signal relative to req_o. Indicates that the data has been stored.
        data_o        : out std_logic_vector(FLIT_SIZE-1 downto 0) --! Data output.
    );
end output_interface;

architecture output_interface of output_interface is
    constant OUTPORTS_NUMBER : integer := number_of_used_ports(COMM_PORTS, THIS_PORT); -- Constant to indicate the number
```

Figure A.32 - VHDL source code for Output Interface module (part 1 of 2).

```vhdl
    constant OUTPORTS_NUMBER : integer := number_of_used_ports(COMM_PORTS, THIS_PORT); -- Constant to indicate the number of OUTPORTS in this interface

    signal arbiter_req_aux : std_logic_vector(OUTPORTS_NUMBER-1 downto 0); -- Auxiliary signal to connect arbiter_request to the arbiter
    signal arbiter_grn_aux : std_logic_vector(OUTPORTS_NUMBER-1 downto 0); -- Auxiliary signal to connect arbiter_grant to the arbiter
    signal arbiter_request : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Signal used to make an arbiter request
    signal arbiter_grant   : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Signal used to indicate an arbiter grant
    signal ctrl_req        : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Request signals generated by outport_ctrl
    signal ctrl_ack        : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Phase-matched Acknowledge signals for the outport_ctrl


    -- Helper function to connect arbiter_request according to the COMM_PORTS
    function arbiter_request_connect(X : std_logic_vector; N : integer; COMM_PORTS : std_logic_vector; THIS_PORT: integer) return std_logic_vector is
        variable aux : std_logic_vector(N-1 downto 0);
        variable j   : integer := 0;
    begin
        for i in COMM_PORTS'range loop
            if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) then
                aux(j) := X(i);
                j := j + 1;

            end if;
        end loop;
        return aux;
    end arbiter_request_connect;

    -- Helper function to connect arbiter_grant according to the COMM_PORTS
    function arbiter_grant_connect(X : std_logic_vector; N : integer; COMM_PORTS : std_logic_vector; THIS_PORT: integer) return std_logic_vector is
        variable aux : std_logic_vector(PORTS_NUMBER-1 downto 0);
        variable j   : integer := 0;
    begin
        for i in COMM_PORTS'range loop
            if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) then
                aux(i) := X(j);
                j := j + 1;
            end if;
        end loop;
        return aux;
    end arbiter_grant_connect;

begin
    -- Req phase encoder
    req_o <= xor_all_bits_considering_comm_ports(ctrl_req, COMM_PORTS, THIS_PORT);


    -- Outport Control Unit
    outport_ctrl_gen: for i in 0 to PORTS_NUMBER-1 generate
        used_outport_ctrl_gen: if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) generate
            ctrl_ack(i) <= ack_i XOR xor_all_bits_but_one_considering_comm_ports(ctrl_req,i, COMM_PORTS, THIS_PORT);

            outport_ctrl_i: entity work.outport_ctrl
            port map (
                reset_i           => reset_i,
                req_outport_i     => req_outport_i(i),
                ack_outport_o     => ack_outport_o(i),
                arbiter_request_o => arbiter_request(i),
                arbiter_grant_i   => arbiter_grant(i),
                req_data_i        => req_data_i(i),
                ack_data_o        => ack_data_o(i),
                last_flit_i       => last_flit_i(i),
                req_o             => ctrl_req(i),
                ack_i             => ctrl_ack(i)
            );
        end generate used_outport_ctrl_gen;
    end generate outport_ctrl_gen;

    -- Map arbiter's wires (this is needed due to the use of COMM_PORTS to determine which ports are connected and which are not, making it more generic)
    arbiter_req_aux <= arbiter_request_connect(arbiter_request, OUTPORTS_NUMBER, COMM_PORTS, THIS_PORT); -- Connect the used arbiter_request to the arbiter
    arbiter_grant <= arbiter_grant_connect(arbiter_grn_aux, OUTPORTS_NUMBER, COMM_PORTS, THIS_PORT); -- Connect grants from arbiter to the arbiter_grants used

    -- Arbiter Instance
    arbiter_i: entity work.arbiter
    generic map(
        SIZE => OUTPORTS_NUMBER
    )
    port map(
        request_i => arbiter_req_aux,
        grant_o   => arbiter_grn_aux
    );

    -- MUX to select data input
    mux_gen: for i in 0 to PORTS_NUMBER-1 generate
        used_mux_gen: if ((i /= THIS_PORT) and (COMM_PORTS(i) = '1')) generate
            data_o <= data_i(i) when arbiter_grant(i) = '1' else
                      (others => 'Z');
        end generate used_mux_gen;
    end generate mux_gen;

end output_interface;
```

Figure A.33 - VHDL source code for Output Interface module (part 2 of 2).

A.9 BaT-Hermes

```vhdl
--! @file hermes_bd_port.vhd
--! @brief Entity to group an input_interface with an output_interface.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-10-14

--------------------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
--------------------------------------------------------------------------------------
-- Generics for configuration:
-- ROUTING_ALGORITHM: Choice of routing algorithm.
--      BUFFER_DEPTH: Defines the depth of the buffer
--    ROUTER_ADDRESS: Address of the router in which this port is inserted
--         THIS_PORT: Defines which port is this interface representing (East, West, North, South, Local)
--        COMM_PORTS: Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are left disconnected.
--------------------------------------------------------------------------------------
-- Interface description:
--                                    -------------
--                                   |             |
--                 reset_i---------->|      I      |
--                                   |      N      |PORTS_NUMBER
--            inport_req_i---------->|      P      |======/===>inport_req_outport_o
--                                   |      U      |PORTS_NUMBER
--            inport_ack_o<----------|      T      |<=====/====inport_ack_outport_i
--                                   |             |
--                                   |      I      |---------->inport_req_data_o
--                                   |      N      |PORTS_NUMBER
--                                   |      T      |<=====/====inport_ack_data_i
--                                   |      F      |
--                       FLIT_SIZE |      A      | FLIT_SIZE
--            inport_data_i====/====>|      C      |=====/====>inport_data_o
--                                   |      E      |---------->inport_last_flit_o
--                                   |             |
--                                    -------------
--                                   |             |
--                       PORTS_NUMBER|      O      |
--   outport_req_outport_i======/===>|      U      |
--                       PORTS_NUMBER|      T      |---------->outport_req_o
--   outport_ack_outport_o<=====/====|      P      |
--                                   |      U      |<----------outport_ack_i
--                       PORTS_NUMBER|      T      |
--      outport_req_data_i=====/====>|             |
--                       PORTS_NUMBER|      I      |
--      outport_ack_data_o<=====/====|      N      |
--                                   |      T      |
--           PORTS_NUMBER x FLIT_SIZE|      F      |
--          outport_data_i=====/====>|      A      | FLIT_SIZE
--                       PORTS_NUMBER|      C      |=====/====>outport_data_o
--     outport_last_flit_i=====/====>|      E      |
--                                   |             |
--                                    -------------
--------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use work.hermes_bd_package.all;

entity hermes_bd_port is
    generic(
        ROUTING_ALGORITHM : string := "XY"; --! Choice of routing algorithm.
        BUFFER_DEPTH      : integer := 8; --! Defines the depth of the buffer.
        ROUTER_ADDRESS    : std_logic_vector((FLIT_SIZE/2)-1 downto 0) := x"11"; --! Address of the router.
        THIS_PORT         : integer := LOCAL; --! Defines which port is this interface representing  (East, West, North, South, Local).
        COMM_PORTS        : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines to which ports this interface connects (index THIS_PORT is ignored). Ports with '0' are disconnected.
    );
    port(
        reset_i               : in  std_logic; --! Active-high reset signal.
        -- Input Interface Signals
        inport_req_i          : in  std_logic; --! Request signal indicating that new data is available on inport_data_i.
        inport_ack_o          : out std_logic; --! Acknowledge signal relative to req_i. Indicates that the data on inport_data_i has been stored.
        inport_data_i         : in  std_logic_vector(FLIT_SIZE-1 downto 0); --! Data input of the input interface.
        inport_req_outport_o  : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting use of an output port.
        inport_ack_outport_i  : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge signal relative to req_outport_o. Indicates that the port use has been granted, and the flit on input_data_o has been consumed.
        inport_req_data_o     : out std_logic; --! Request signal indicating that new payload is available on input_data_o.
        inport_ack_data_i     : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge relative to req_data_o. Indicates that the payload was consumed.
        inport_data_o         : out std_logic_vector(FLIT_SIZE-1 downto 0); --! Input interface data output.
        inport_last_flit_o    : out std_logic; --! Active-high signal indicating that the flit on data_o is the last flit of the current packet.
        -- Output Interface Signals
        outport_req_outport_i : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Signal requesting use of the output port.
        outport_ack_outport_o : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge signal relative to req_outport_i. Indicates that the port use has been granted, and the flit on outport_data_i has been consumed.
        outport_req_data_i    : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Request signal indicating that new
```

Figure A.34 - VHDL source code for router port (part 1 of 2).

```vhdl
payload is available on outport_data_i.
        outport_ack_data_o    : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge relative to req_data_i.
Indicates that the payload was consumed.
        outport_data_i        : in  flit_size_array(PORTS_NUMBER-1 downto 0); --! Output interface data input.
        outport_last_flit_i   : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Active-high signal indicating that the
flit on data_i is the last flit of the current packet.
        outport_req_o         : out std_logic; --! Request signal indicating that new data is available on outport_data_o.
        outport_ack_i         : in  std_logic; --! Acknowledge signal relative to req_o. Indicates that the data has been
stored.
        outport_data_o        : out std_logic_vector(FLIT_SIZE-1 downto 0) --! Output interface data output.
    );
end hermes_bd_port;

architecture hermes_bd_port of hermes_bd_port is
begin

    -- Input Interface Instance
    input_interface_i: entity work.input_interface
    generic map(
        ROUTING_ALGORITHM => ROUTING_ALGORITHM,
        BUFFER_DEPTH      => BUFFER_DEPTH,
        ROUTER_ADDRESS    => ROUTER_ADDRESS,
        THIS_PORT         => THIS_PORT,
        COMM_PORTS        => COMM_PORTS
    )
    port map (
        reset_i       => reset_i,
        req_i         => inport_req_i,
        ack_o         => inport_ack_o,
        data_i        => inport_data_i,
        req_outport_o => inport_req_outport_o,
        ack_outport_i => inport_ack_outport_i,
        req_data_o    => inport_req_data_o,
        ack_data_i    => inport_ack_data_i,
        data_o        => inport_data_o,
        last_flit_o   => inport_last_flit_o
    );

    -- Output Interface Instance
    output_interface_i: entity work.output_interface
    generic map (
        THIS_PORT  => THIS_PORT,
        COMM_PORTS => COMM_PORTS
    )
    port map (
        reset_i       => reset_i,
        req_outport_i => outport_req_outport_i,
        ack_outport_o => outport_ack_outport_o,
        req_data_i    => outport_req_data_i,
        ack_data_o    => outport_ack_data_o,
        data_i        => outport_data_i,
        last_flit_i   => outport_last_flit_i,
        req_o         => outport_req_o,
        ack_i         => outport_ack_i,
        data_o        => outport_data_o
    );

end hermes_bd_port;

```

Figure A.35 - VHDL source code for router port (part 2 of 2).

```vhdl
--! @file hermes_bd_router.vhd
--! @brief A bundled-data transition-signaling handshake-based parameterizable NoC router.
--! @author Matheus Gibiluka, matheus.gibiluka@acad.pucrs.br
--! @date 2013-10-14

--------------------------------------------------------------------------------------
-- Dependencies:
-- > hermes_bd_package.vhd
--------------------------------------------------------------------------------------
-- Generics for configuration:
-- ROUTING_ALGORITHM: Choice of routing algorithm.
--      BUFFER_DEPTH: Defines the depth of each input buffer.
--    ROUTER_ADDRESS: Address of the router.
--             PORTS: Defines which ports are implemented in the router. Ports with '0' are left unconnected.
--------------------------------------------------------------------------------------
-- Interface description:
--                            -------------
--                           |             |
--         reset_i---------->|             |
--                           |             |
--              PORTS_NUMBER |             | PORTS_NUMBER
--            req_i======/===>|             |======/===>req_o
--              PORTS_NUMBER |             | PORTS_NUMBER
--            ack_o<=====/====|             |<=====/====ack_i
--                           |             |
--   PORTS_NUMBER x FLIT_SIZE|             |PORTS_NUMBER x FLIT_SIZE
--            data_i====/====>|             |=====/====>data_o
--                           |             |
--                            -------------
--------------------------------------------------------------------------------------


```

Figure A.36 - VHDL source code for BaT-Hermes router (part 1 of 2).

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.hermes_bd_package.all;

entity hermes_bd_router is
    generic(
        ROUTING_ALGORITHM : string := "XY"; --! Choice of routing algorithm.
        BUFFER_DEPTH      : integer := 8; --! Defines the depth of the buffer.
        ROUTER_ADDRESS    : std_logic_vector((FLIT_SIZE/2)-1 downto 0) := x"11"; --! Address of the router.
        PORTS             : std_logic_vector(PORTS_NUMBER-1 downto 0) := "11111" --! Defines which ports are implemented in the router. Ports with '0' are left unconnected.
    );
    port(
        reset_i : in  std_logic; --! Active-high reset signal.
        req_i   : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Request signal indicating that new data is available on data_i.
        ack_o   : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge signal relative to req_i. Indicates that the data on data_i has been stored.
        data_i  : in  flit_size_array(PORTS_NUMBER-1 downto 0); --! Data input for the input ports.
        req_o   : out std_logic_vector(PORTS_NUMBER-1 downto 0); --! Request signal indicating that new data is available nm data_o.
        ack_i   : in  std_logic_vector(PORTS_NUMBER-1 downto 0); --! Acknowledge signal relative to req_o. Indicates that the data has been stored.
        data_o  : out flit_size_array(PORTS_NUMBER-1 downto 0) --! Output interface for the output ports.
    );
end hermes_bd_router;

architecture hermes_bd_router of hermes_bd_router is
    type ports_array_t is array(0 to PORTS_NUMBER-1) of std_logic_vector(PORTS_NUMBER-1 downto 0);

    signal req_outport     : ports_array_t; -- Signal to hold each inport_req_outport_o. Use: req_outport(input_port)(output_port)
    signal req_outport_inv : ports_array_t; -- Signal to hold each inport_req_outport_o. Use: req_outport(output_port)(input_port)
    signal ack_outport     : ports_array_t; -- Signal to hold each outport_ack_outport_o. Use: ack_outport(output_port)(input_port)
    signal ack_outport_inv : ports_array_t; -- Signal to hold each outport_ack_outport_o. Use: ack_outport(input_port)(output_port)
    signal req_data        : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Signal to hold each inport_req_data_o. Use: req_data(input_port)
    signal ack_data        : ports_array_t; -- Signal to hold each inport_req_data_o. Use: (output_port)(input_port)
    signal ack_data_inv    : ports_array_t; -- Signal to hold each inport_req_data_o. Use: (input_port)(output_port)
    signal data            : flit_size_array(PORTS_NUMBER-1 downto 0); -- Signal to hold each inport_data_o. Use: data(input_port)
    signal last_flit       : std_logic_vector(PORTS_NUMBER-1 downto 0); -- Signal to hold each inport_last_flit_o Use: last_flit(input_port)

begin
    -- Generation of inverted wires to facilitate connection between input and output interfaces
    inv_wire_gen: for i in 0 to PORTS_NUMBER-1 generate
        inv_wire2_gen: for j in 0 to PORTS_NUMBER-1 generate
            req_outport_inv(j)(i) <= req_outport(i)(j);
            ack_outport_inv(j)(i) <= ack_outport(i)(j);
            ack_data_inv(j)(i) <= ack_data(i)(j);
        end generate inv_wire2_gen;
    end generate inv_wire_gen;


    -- Generating ports
    port_gen: for i in 0 to PORTS_NUMBER-1 generate
        used_port_gen: if (PORTS(i) = '1') generate

            hermes_bd_port_i: entity work.hermes_bd_port
            generic map(
                ROUTING_ALGORITHM => ROUTING_ALGORITHM,
                BUFFER_DEPTH      => BUFFER_DEPTH,
                ROUTER_ADDRESS    => ROUTER_ADDRESS,
                THIS_PORT         => i,
                COMM_PORTS        => PORTS
            )
            port map (
                reset_i               => reset_i,
                inport_req_i          => req_i(i),
                inport_ack_o          => ack_o(i),
                inport_data_i         => data_i(i),
                inport_req_outport_o  => req_outport(i),
                inport_ack_outport_i  => ack_outport_inv(i),
                inport_req_data_o     => req_data(i),
                inport_ack_data_i     => ack_data_inv(i),
                inport_data_o         => data(i),
                inport_last_flit_o    => last_flit(i),
                outport_req_outport_i => req_outport_inv(i),
                outport_ack_outport_o => ack_outport(i),
                outport_req_data_i    => req_data,
                outport_ack_data_o    => ack_data(i),
                outport_data_i        => data,
                outport_last_flit_i   => last_flit,
                outport_req_o         => req_o(i),
                outport_ack_i         => ack_i(i),
                outport_data_o        => data_o(i)
            );
        end generate used_port_gen;
    end generate port_gen;

end hermes_bd_router;
```

Figure A.37 - VHDL source code for BaT-Hermes router (part 1 of 2).

B. Relative Timing Constraints

This appendix contains graphical representations of the relative timing constraints found in BaT-Hermes. Wires are color-coded green and red, representing, respectively, base and enforced paths. Delay lines are inserted in the latter. Constraints between blocks are accounted in the input and output signals of each circuit – for example, the timing restrictions between the FIFO and the Input Buffer Control are accounted in the *req_rd_o*, *ack_rd_i*, *data_o, req_i*, *ack_o* and *data_i* signals of these modules.

B.1 FIFO

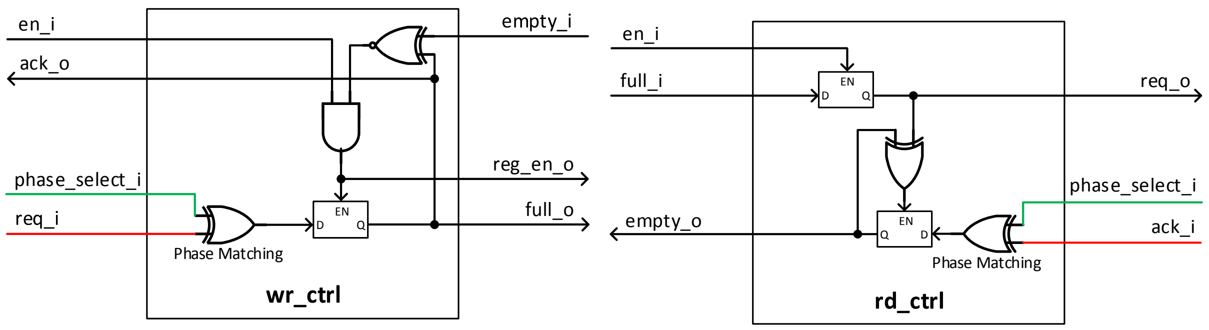


Figure B.1 – FIFO: constraint ensuring that *phase_select_*i arrives before *req_i* in the write control circuit and *phase_select_*i arrives before *ack_i* in the read control circuit.

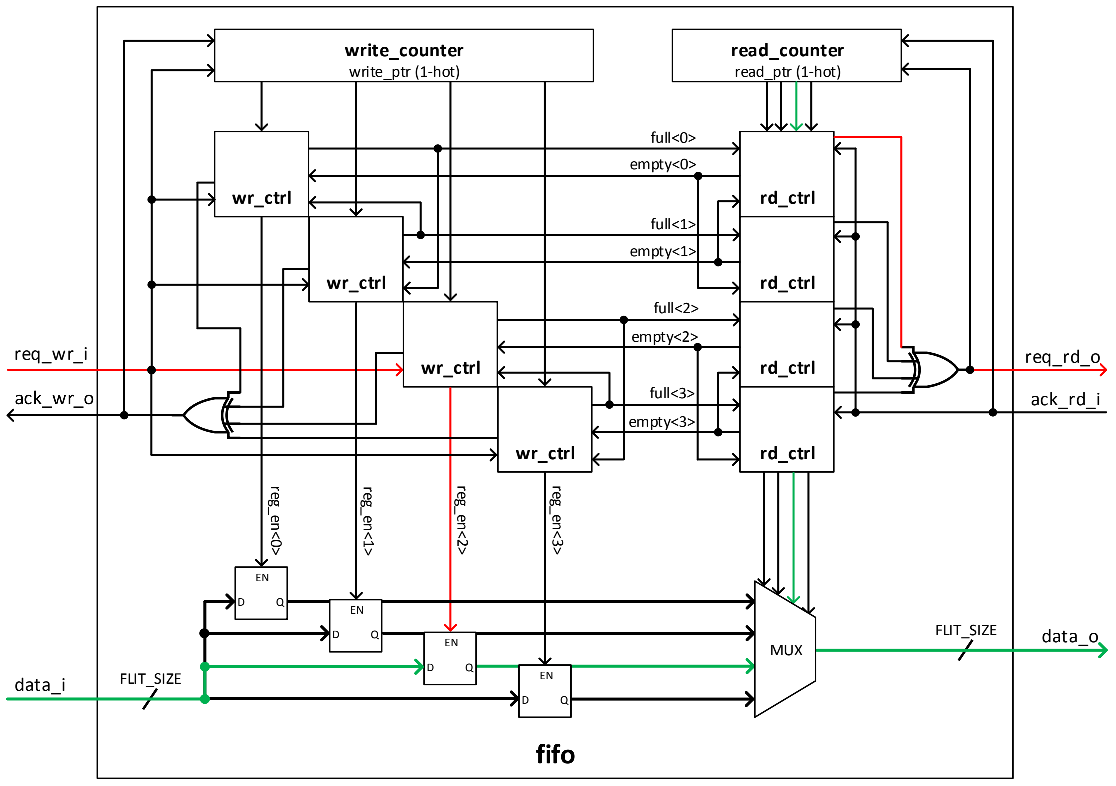


Figure B.2 – FIFO: constraints ensuring that, for each buffer position, *data_i* arrives before *req_wr_i* and *data_*o arrives before *req_rd_*o.

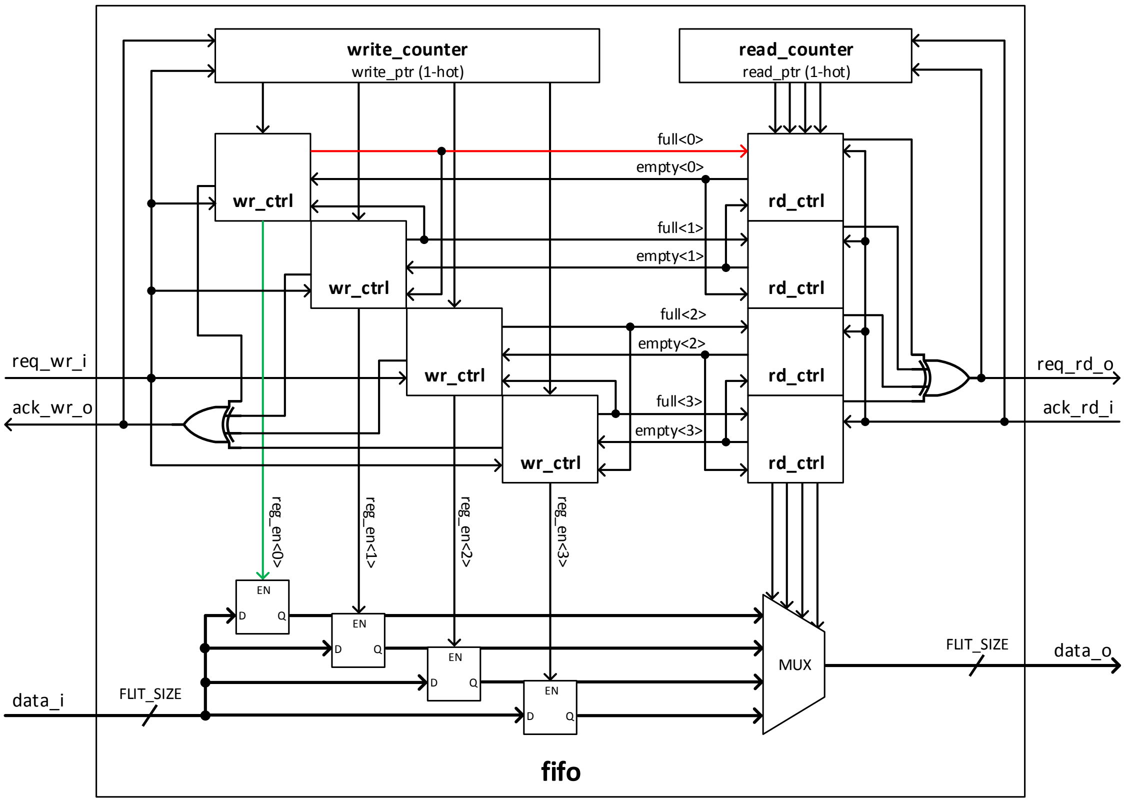


Figure B.3 – FIFO: constraint ensuring that, for each buffer position, *reg_en* arrives at the register before *full*_o arrives at the read control circuit.

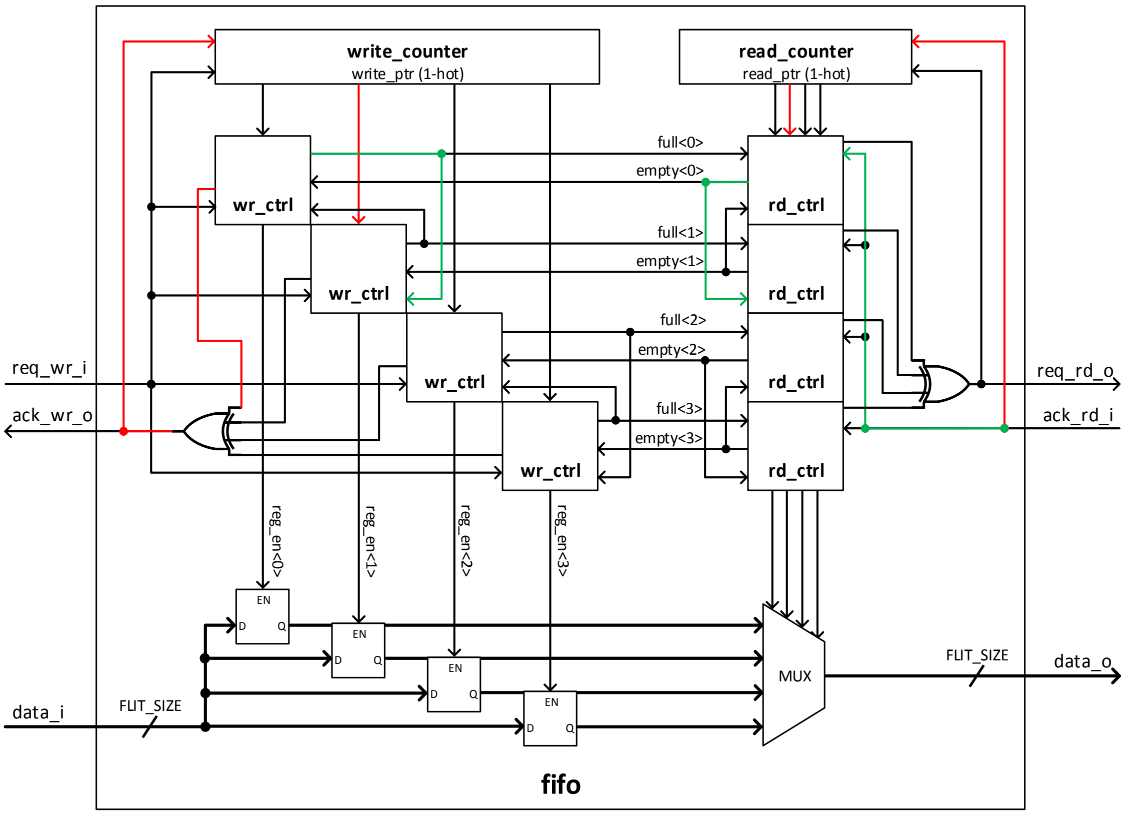


Figure B.4 – FIFO: constraints ensuring that, for each read and write control circuit, *phase_select*_i arrives before *en_i*.

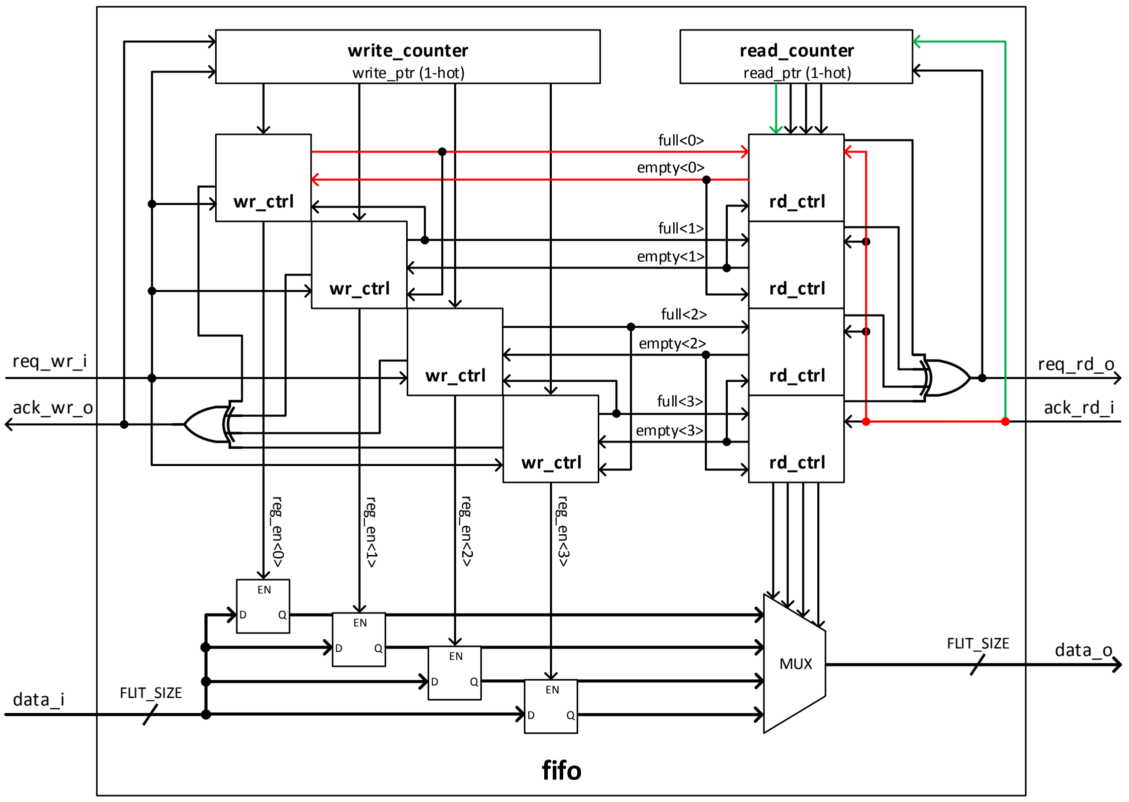


Figure B.5 – FIFO: constraints ensuring that, for each read control circuit, *en_*i is disabled before a new *full_o* request can be made.

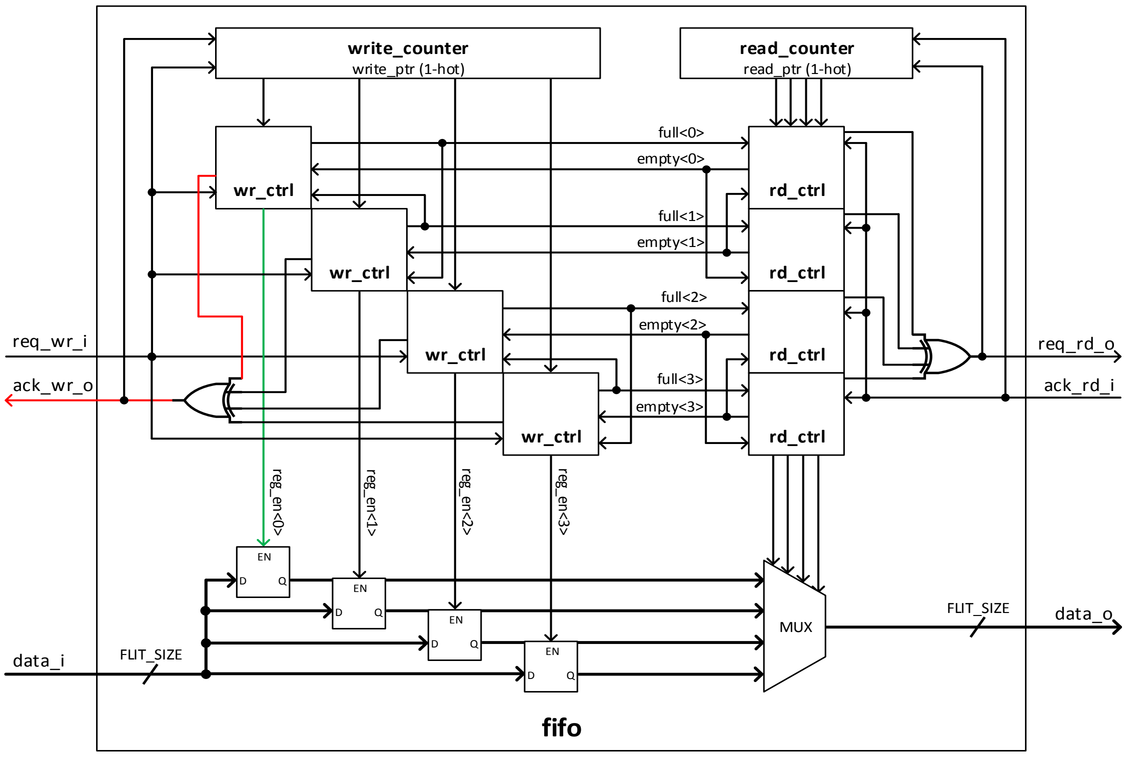


Figure B.6 – FIFO: constraints ensuring that, for each register, the data is stored before an *ack_wr_o* is issued.

B.2 Input Buffer Control

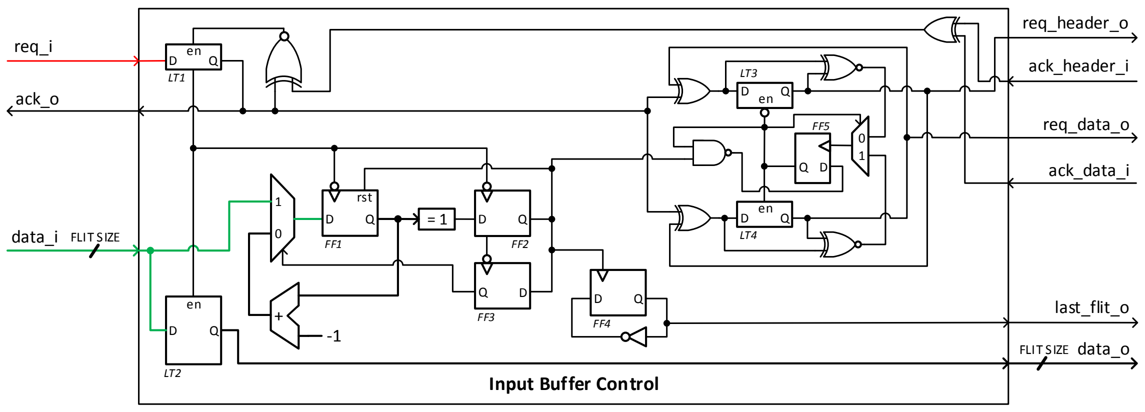


Figure B.7 – Input Buffer Control: constraint ensuring that *data_i* arrives before *req_i*.

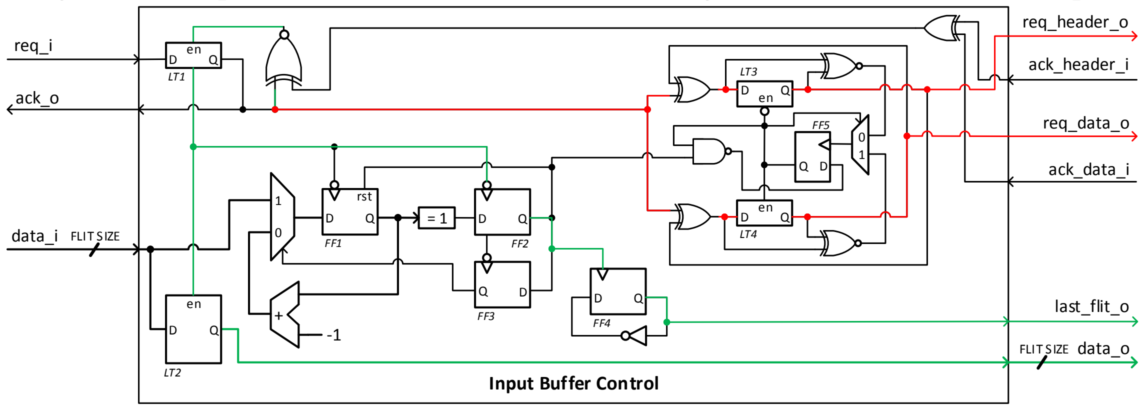


Figure B.8 – Input Buffer Control: constraints ensuring that *data_o and last_flit_o* arrives before *req_header_*o or *req_data_o*.

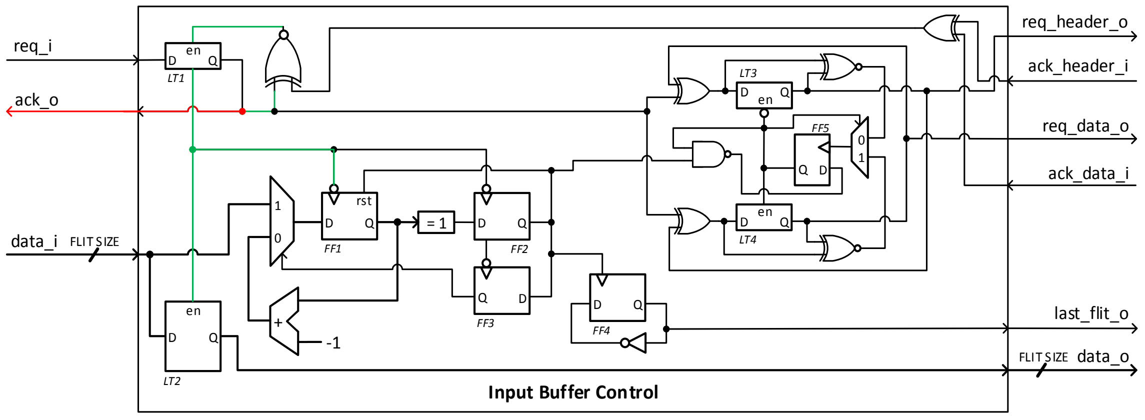


Figure B.9 – Input Buffer Control: constraint ensuring that *data_o is properly stored in LT2 and FF1* before *ack_*o is issued.

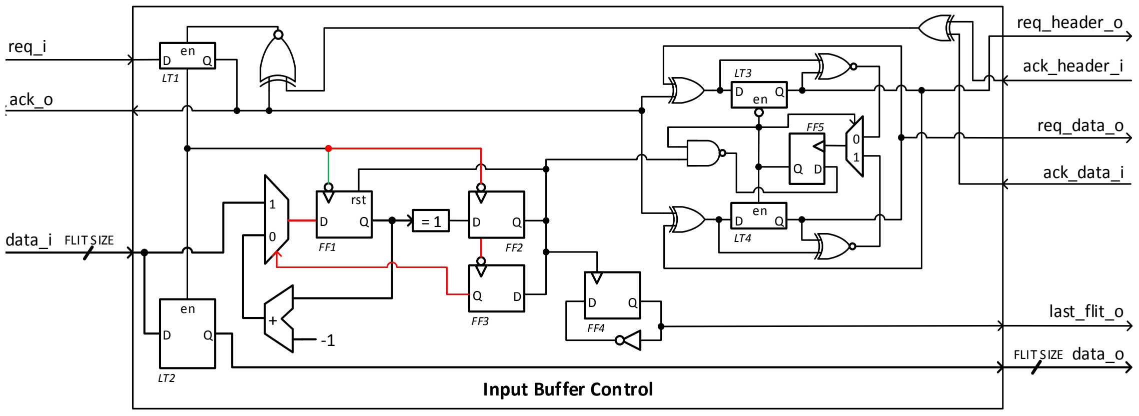


Figure B.10 – Input Buffer Control: constraint ensuring that, when a request is received, the mux remains stable until FF1 stores the data currently at its input.

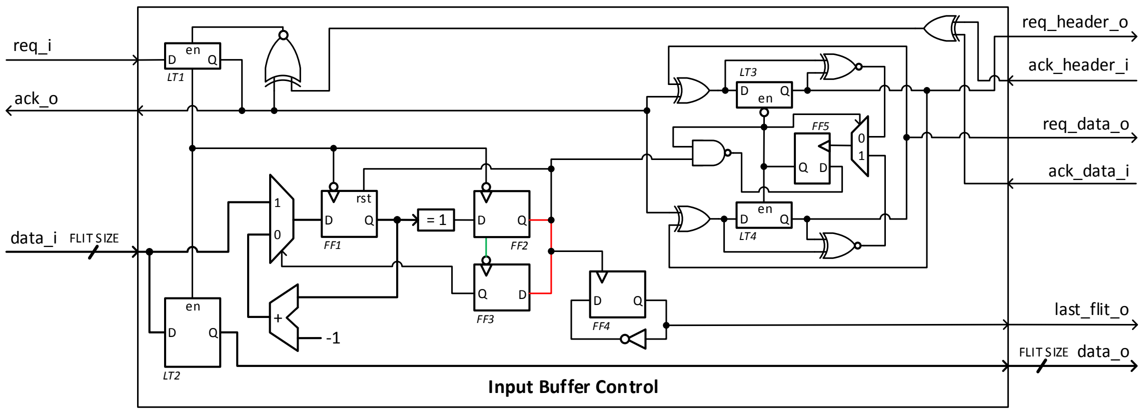


Figure B.11 – Input Buffer Control: constraint ensuring that, when a request is received, the data at the input pin of FF3 only changes after it has been captured.

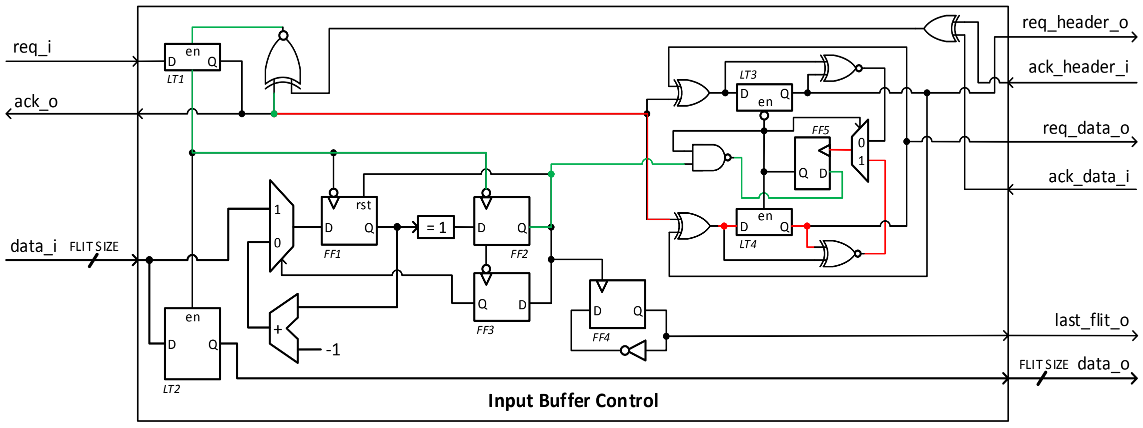


Figure B.12 – Input Buffer Control: constraint ensuring that, when a request is received, FF5 only captures the data at its input pin after *last_flit_lvl* has propagated.

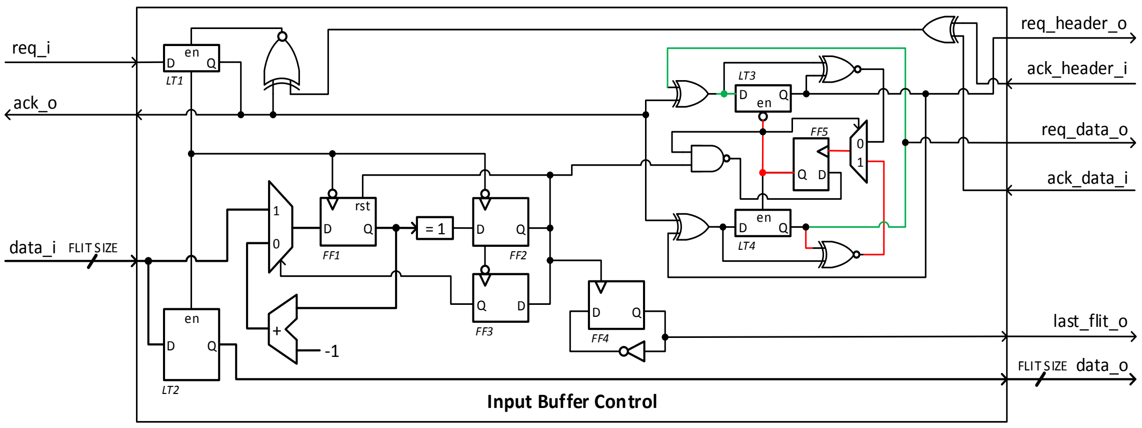


Figure B.13 – Input Buffer Control: constraint ensuring that LT3 is enabled only after all its input signals have propagated.

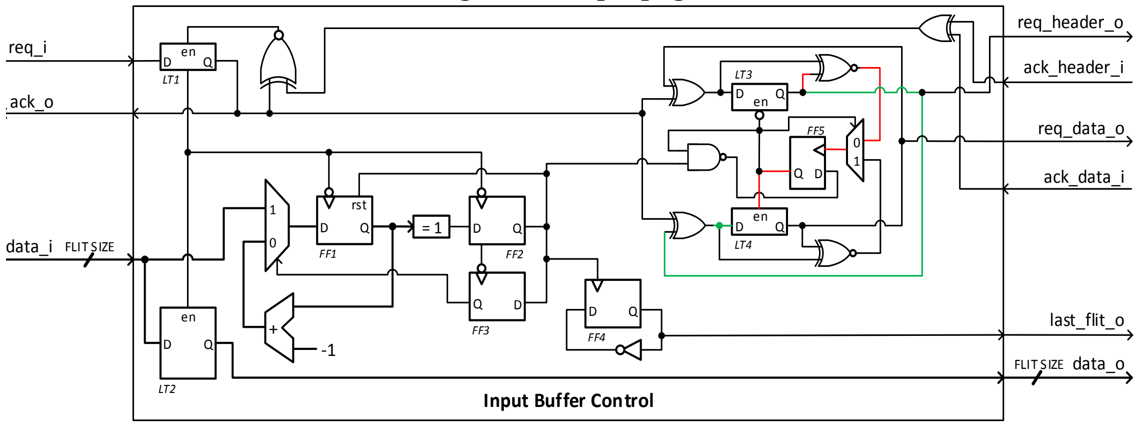


Figure B.14 – Input Buffer Control: constraint ensuring that LT4 is enabled only after all its input signals have propagated.

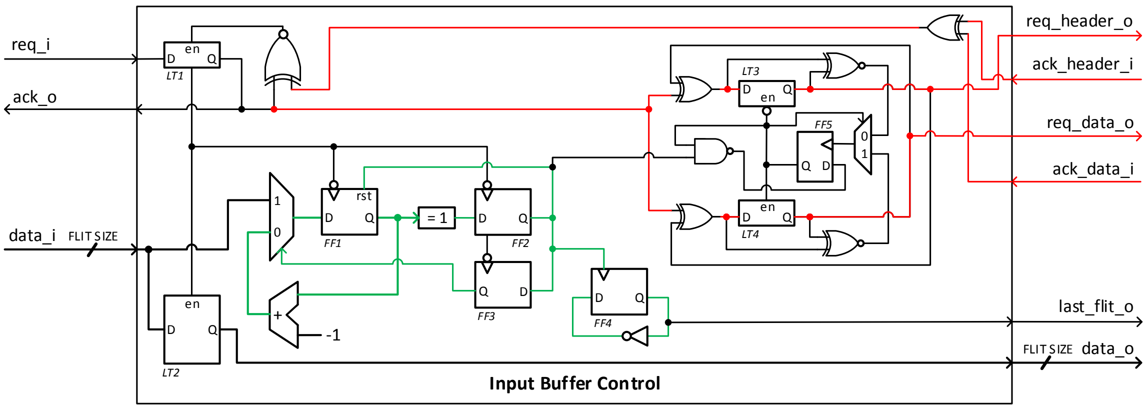


Figure B.15 – Input Buffer Control: constraint ensuring that all FSM data signals propagate during a handshake cycle.

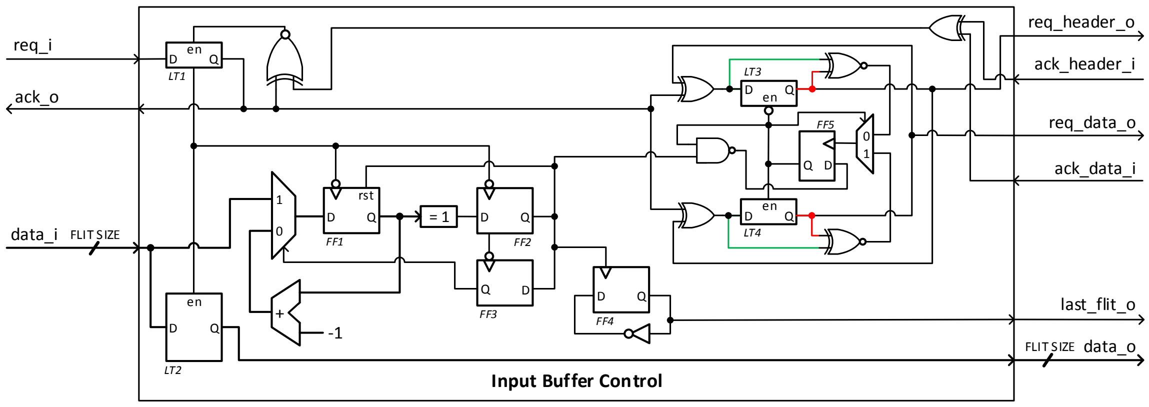


Figure B.16 – Input Buffer Control: constraints ensuring the minimum period of the signal used to clock FF5 is large enough to fulfill the flip-flop's timing requirements.

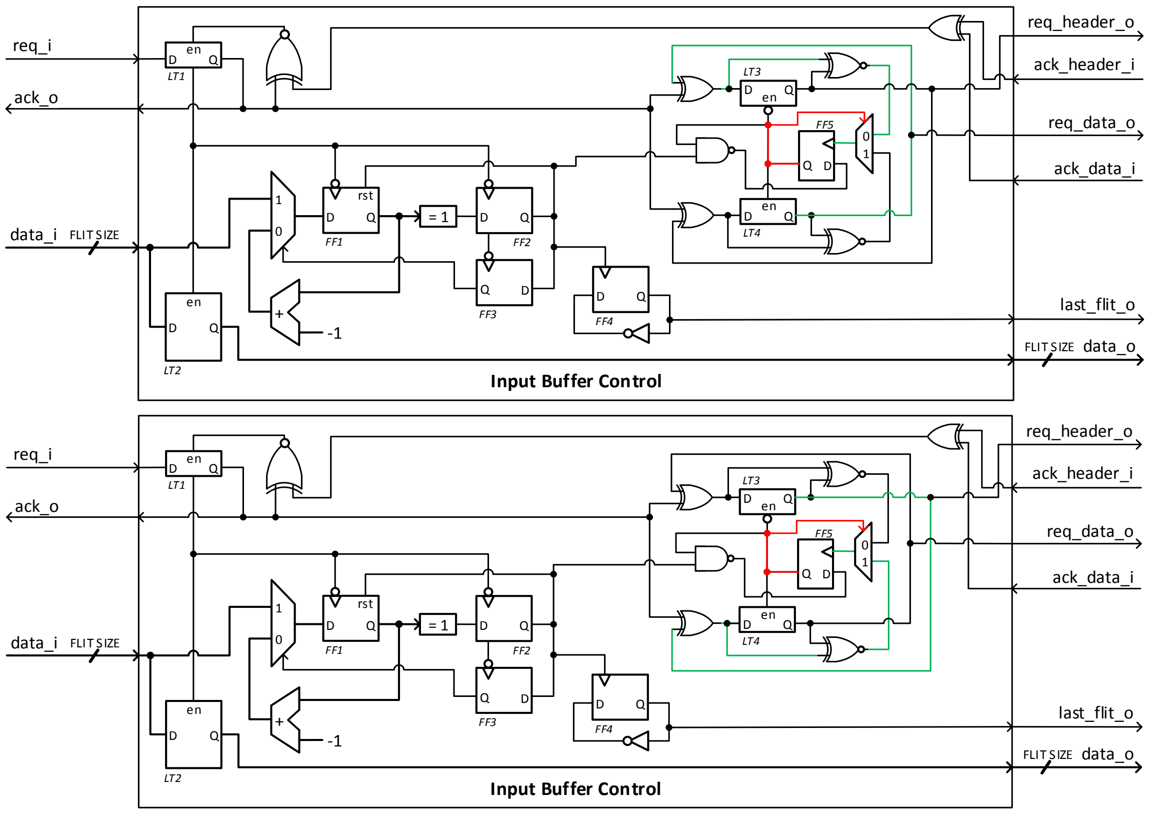


Figure B.17 – Input Buffer Control: constraints ensuring that the control signal of the mux only changes after the signals at the clock input of FF5 have propagated.

B.3 Routing Control

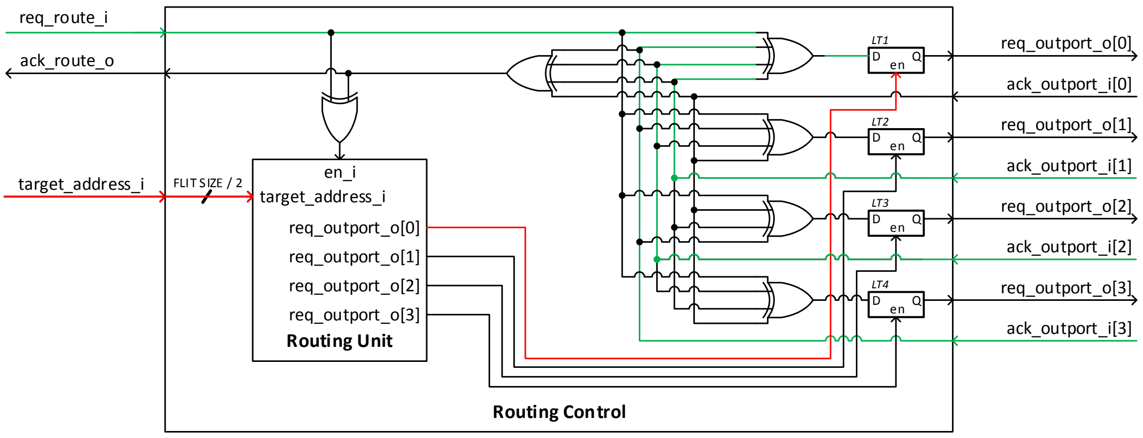


Figure B.18 – Routing Control: constraints ensuring that, for each latch connected to a *req_outport_o* signal, the signals at the input propagate before the enable signal.

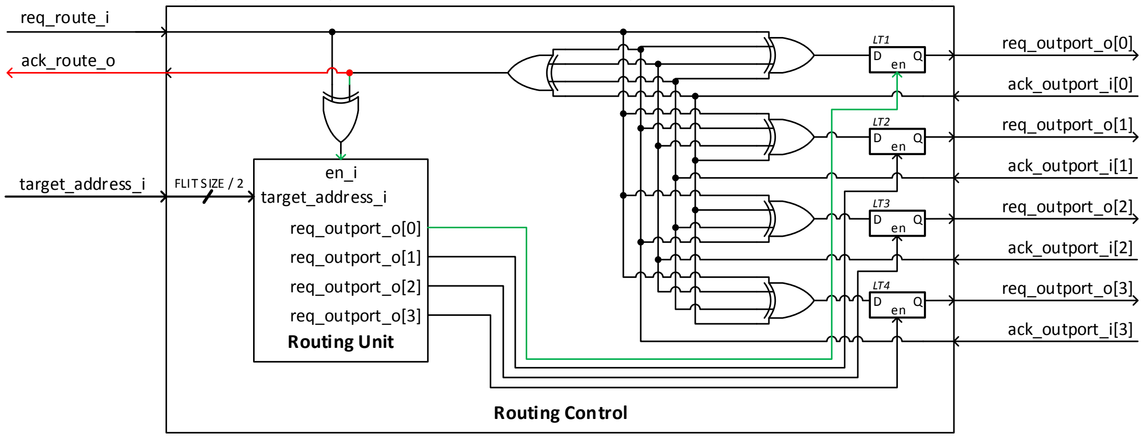


Figure B.19 – Routing Control: constraints ensuring that, for each latch connected to a *req_outport_o* signal, the enable signal disabling the latch propagates before *ack_route_o*.

B.4 Outport Control

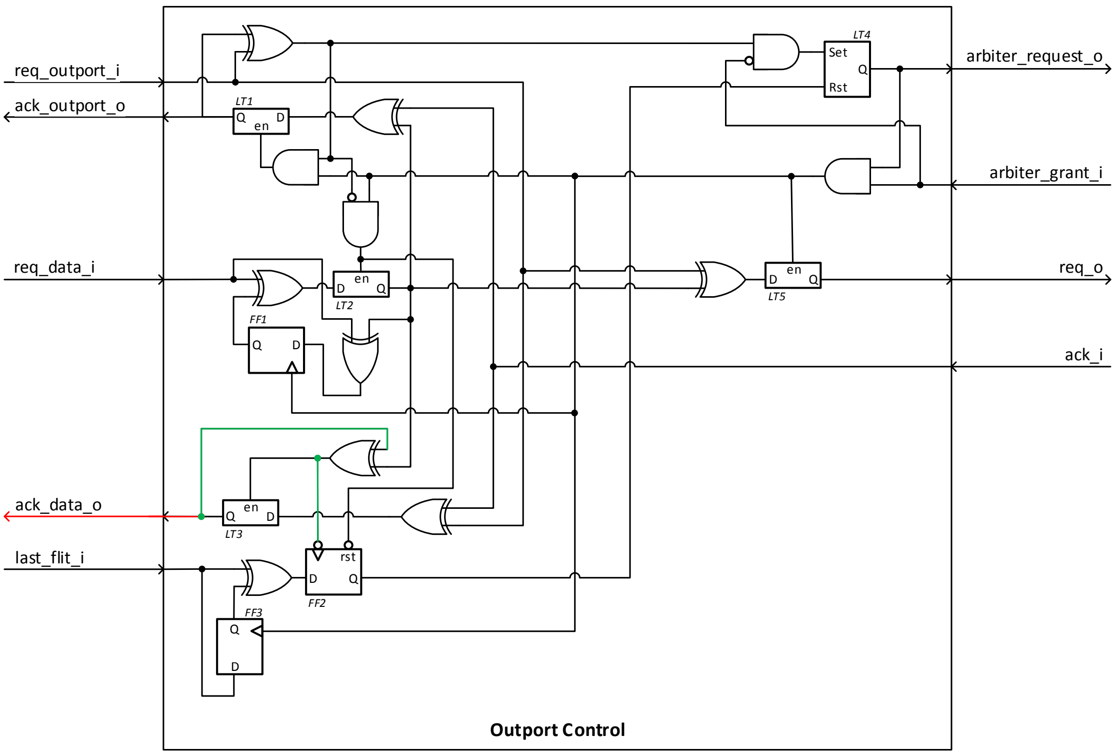


Figure B.20 – Outport Control: constraint ensuring the last flit detector is activated before the connected Input Interface receives the *ack_data_o* acknowledge.

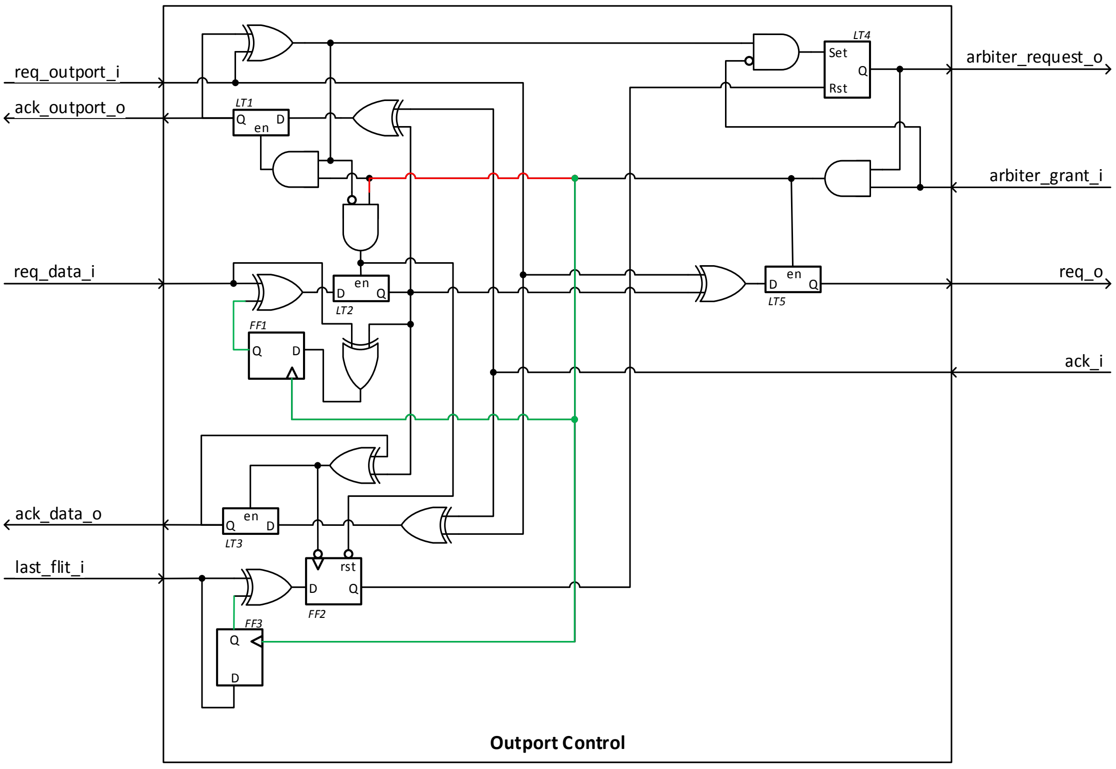


Figure B.21 – Outport Control: constraint ensuring the programmable phase matcher and the last flit detector are initialized before an *ack_outport_o* acknowledge is issued.

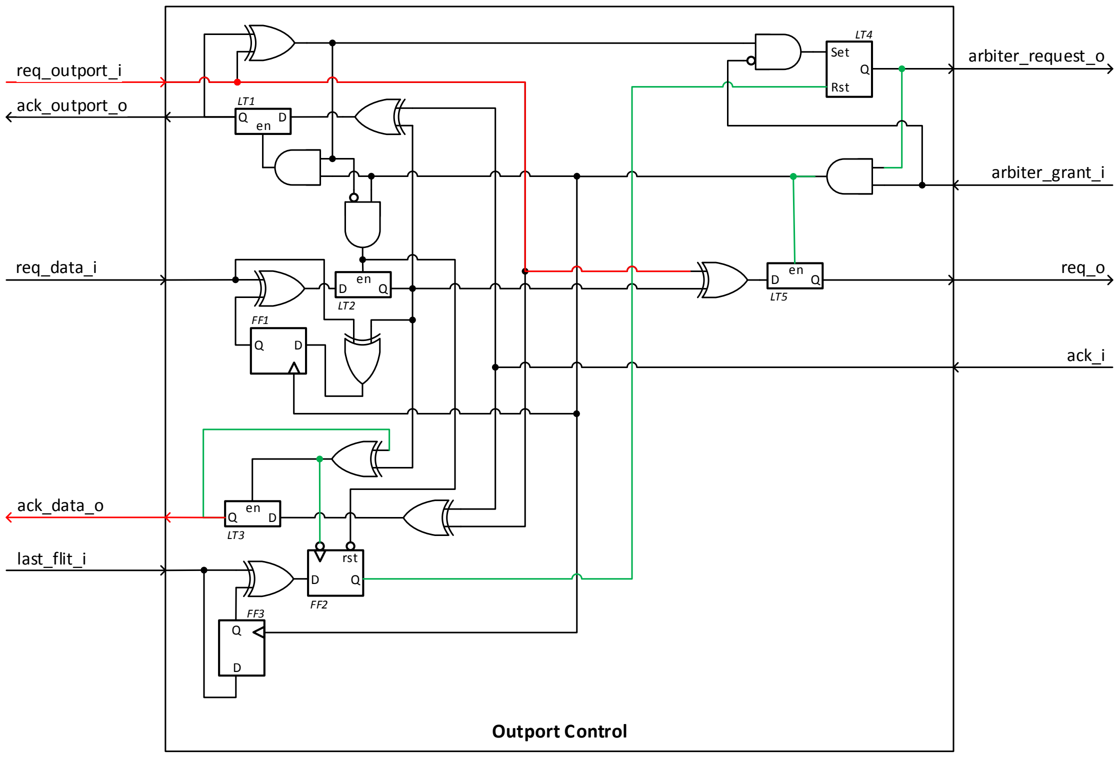


Figure B.22 – Outport Control: constraint ensuring that a new *req_outport_i* request is received after LT5 has been disabled.

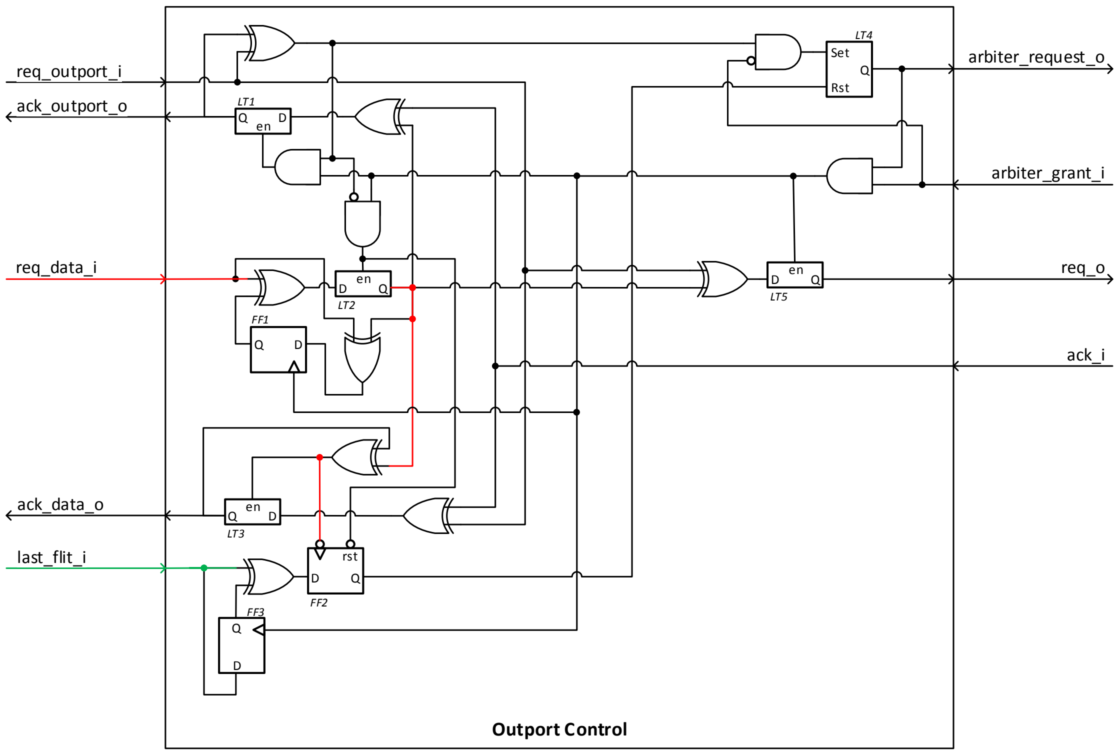


Figure B.23 – Outport Control: constraint ensuring that the last flit detector only tests a valid *last_flit_i* signal.

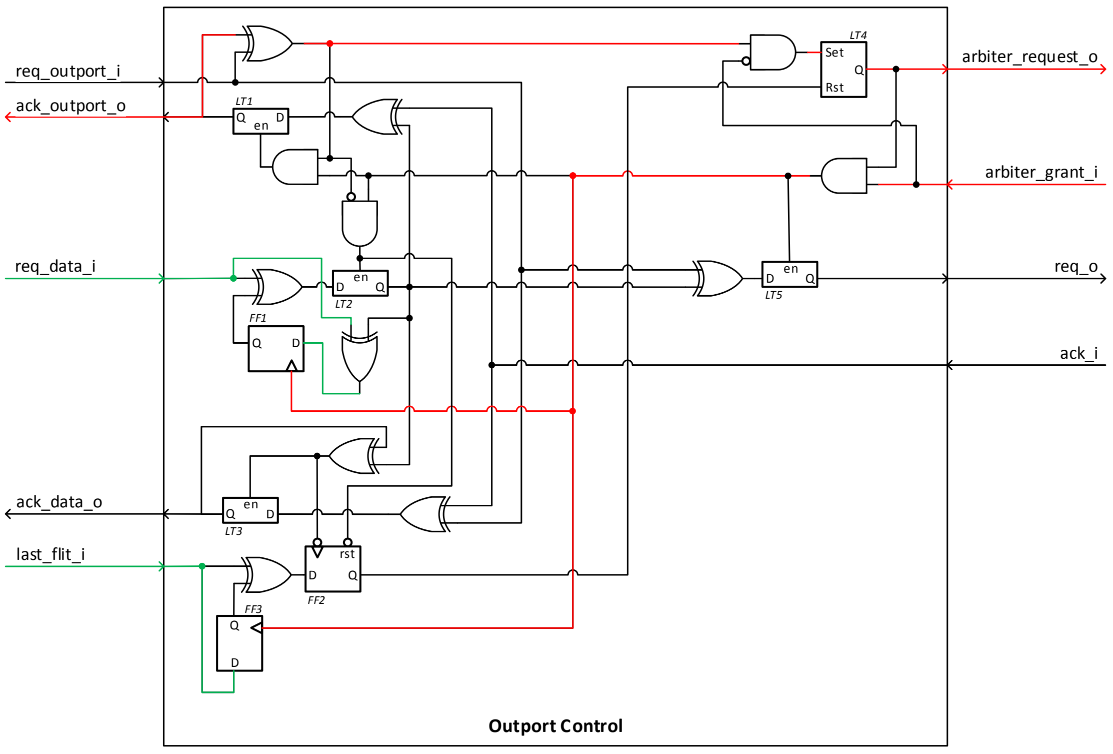


Figure B.24 – Outport Control: constraints ensuring that the signals to be stored in FF1 and FF3 arrive before the arbiter grant is given.

B.5 Output Interface

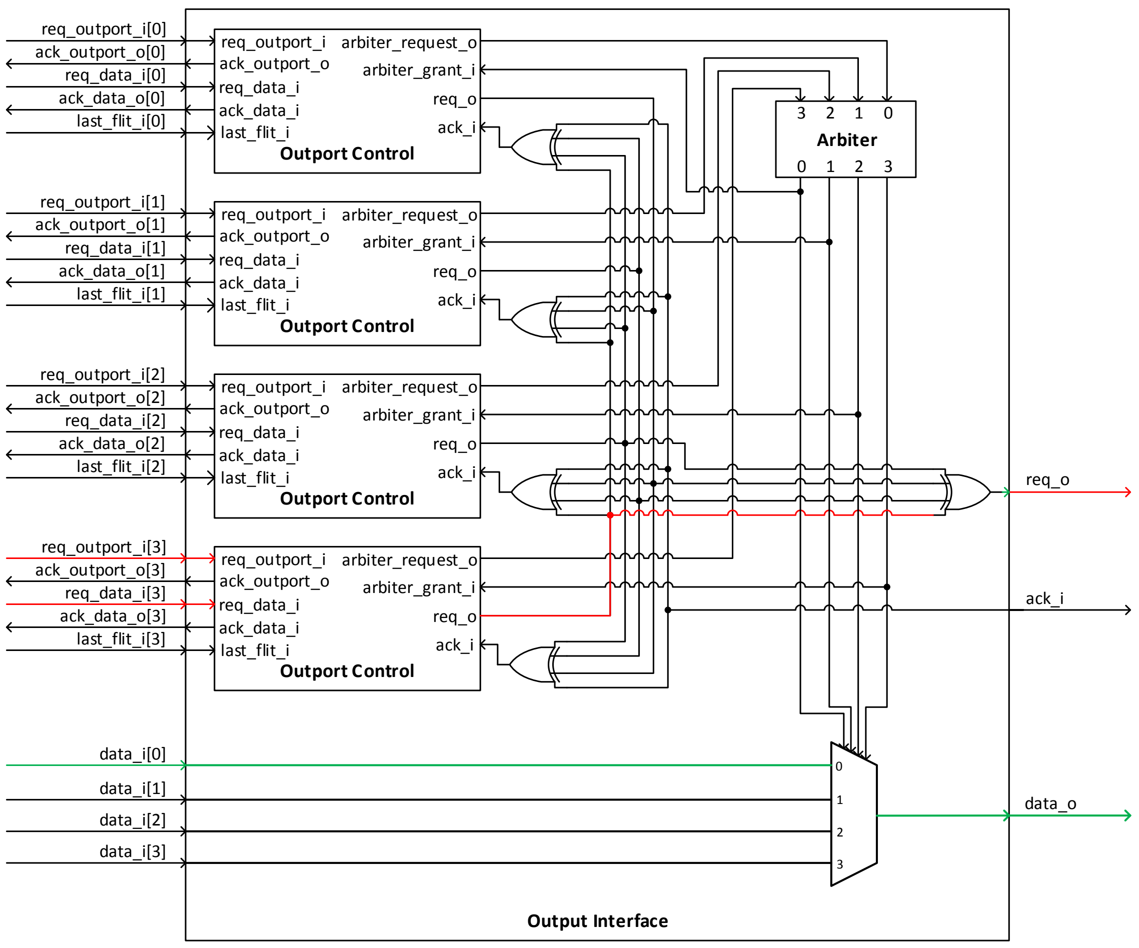


Figure B.25 – Output Interface: constraint ensuring that *data_o* signal is stable before a *req_o* request is issued.