João Gonçalves de Almeida

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

Developing Globally-Asynchronous Locally-Synchronous Systems through the IOPT-Flow Framework

Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador:	Doutor Filipe de Carvalho Moutinho, Professor Auxiliar, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
Co-orientador:	Doutor Rogério Alexandre Botelho Campos Rebelo, Investigador, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Júri:

Presidente:	Doutor José António Barata de Oliveira - FCT/UNL
Arguente:	Doutora Anikó Katalin Horváth da Costa - FCT/UNL
Vogal:	Doutor Filipe de Carvalho Moutinho - FCT/UNL

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2019

Developing Globally-Asynchronous Locally-Synchronous Systems through the IOPT-Flow Framework

To my family

Acknowledgements

First and foremost, I would like to thank both of my advisers Professor Filipe de Carvalho Moutinho and Professor Rogério Alexandre Botelho Campos Rebelo for the opportunity and for the support and insight provided throughout the development of this dissertation.

I would like to thank the *Faculdade de Ciências e Tecnologia* from *Universidade NOVA de Lisboa* for the opportunities provided during these five years.

A special thanks to Professor Fernando Joaquim Ganhão Perreira for his availability and assistance.

And lastly, I would like to thank my friends for their friendship and my family for love and encouragement.

Abstract

Throughout the years, synchronous circuits have increased in size and complexity, consequently, distributing a global clock signal has become a laborious task. Globally-Asynchronous Locally-Synchronous (GALS) systems emerge as a possible solution; however, these new systems require new tools.

The DS-Pnet language formalism and the IOPT-Flow framework aim to support and accelerate the development of cyber-physical systems. To do so it offers a tool chain that comprises a graphical editor, a simulator and code generation tools capable of generating C, JavaScript and VHDL code. However, DS-Pnets and IOPT-Flow are not yet tuned to handle GALS systems, allowing for partial specification, but not a complete one.

This dissertation proposes extensions to the DS-Pnet language and the IOPT-Flow framework in order to allow development of GALS systems. Additionally, some asynchronous components were created, these form interfaces that allow synchronous blocks within a GALS system to communicate with each other.

Keywords: DS-Pnets; IOPT-Flow; GALS; Asynchronous components; model-based development.

Resumo

Ao longo dos anos, os circuitos síncronos têm vindo a crescer em dimensão e complexidade, consequentemente, distribuir um sinal clock tornou-se uma tarefa árdua. Sistemas Globalmente-Assíncronos Localmente-Síncronos (GALS) surgem como uma possível solução, no entanto, estes requerem novas ferramentas.

O formalismo DS-Pnet e o conjunto de ferramentas IOPT-Flow têm como objetivo suportar e acelerar o desenvolvimento de sistemas ciber-físicos. E, por essa razão, disponibilizam um conjunto de ferramentas que incluem um editor gráfico, um simulador e ferramentas para geração de código seja ele C, JavaScript ou VHDL. No entanto, as DS-Pnets e as IOPT-Flow não estão ajustadas para sistemas GALS, permitindo especificar parcialmente, mas não por completo.

Esta dissertação propõe extensões ao formalismo DS-Pnet e à IOPT-Flow framework de forma a permitir o desenvolvimento de sistemas GALS. Adicionalmente, são propostos alguns componentes, estes formam interfaces que permitem comunicação entre blocos síncronos dentro de um sistema GALS.

Palavras-chave: DS-Pnets; IOPT-Flow; GALS; Componentes assíncronos; Desenvolvimento baseado em modelos.

Content

List of Figures

List of Tables

Acronyms

Ack - Acknowledge

DS-Pnet - Dataflow, Signals and Petri nets

FIFO - First In First Out

FPGA - Field Programmable Gate Array

GALS - Globally Asynchronous Locally Synchronous

HDL - Hardware Description Language

HLS - High-Level Synthesis

IOPT-Flow - Framework that employs the DS-Pnet Language

IOPT-net - Input-Output Place-Transition Petri-net class

IOPT-tools - Framework that employs the IOPT-net language

ISIM - Xilinx ISE Simulator

NoC - Network-on-Chip

Req - Request

SoC - System-on-Chip

1

Chapter One - Introduction

1.1 - Motivation

Circuits are ever evolving, over the years, they have greatly increased in size and complexity. For this reason, distributing a global clock over the entire circuit with minimal clock skew has become an issue subject to great analysis and consideration by designers and researchers. [1][2][3]

More attention is given to alternative options such as asynchronous systems and Globally-Asynchronous Locally-Synchronous (GALS) systems. A GALS system is halfway between a synchronous and an asynchronous system, by breaking down a fully synchronous system into several smaller blocks, these are their own synchronous circuits with their respective clock generators and frequencies. Consequently, since each block has its own individual clock signal, an asynchronous interface is required for communication.

A GALS design provides benefits from both sides, much of the standard synchronous methodology and design tools may still be employed. However, asynchronous interfaces connect the synchronous blocks which suggests a naturally modular design. This modular design is the greater benefit of employing a GALS design since each block can be reused and replaced without any additional cost, because they are not constrained by a global clock. [1][4]

On one hand, asynchronous interfaces between synchronous blocks imprint latency and introduce a bottleneck in the design that may limit throughput. On the other hand, while it is not a guarantee, a GALS system provides the possibility to fine tune each clock signal may achieve higher power efficiency and lower electromagnetic noise. Lastly, the potential to reuse and replace synchronous blocks will inevitably speed up development time giving designers an edge on the time-to-market. [2][5]

1.2 - Objectives

While an ample range of different tools for developing circuits exist some authors argue that there is a lack of GALS specific tools, especially when it comes to verification and testing, this is due to the uncertainty of asynchronous communication. For this reason, one of the objectives of this work is to analyse existing tools and methodologies for the development of GALS systems. [6][7][8]

The DS-Pnet (Dataflow, Signals and Petri nets) modelling language combines Petri net nodes and data-flow nodes to support the development of cyber-physical systems. Petri net nodes inherit the main characteristics of Input-Output Place-Transition (IOPT) nets and are responsible for the reactive part of the controllers, meaning they control the state of the system and its evolution. Data-flow nodes employ mathematical expressions to calculate new values (data processing), the output value of such nodes is an expression based on the inputs. [9][10][11]

IOPT-Flow modelling framework is a web-based set of tools that supports DS-Pnets. It comprises a graphical editor, a simulator/debugger and automatic code generators, supporting a complete design work-flow. The tool outputs C and JavaScript software code or VHDL hardware descriptions. [11]

Model-based development formalisms based on Petri nets bring several advantages, such as rapid prototyping, simulation and automatic model-checking tools. Additionally, graphical languages are inherently more accessible to unexperienced users, which facilitates communication between different roles of the development team.

Petri net-based formalisms tend to lack input/output and data processing capabilities, which can sometimes make them unsuitable for real-world applications. The DS-Pnet language formalism and its associated framework, IOPT-Flow, introduce these capabilities, evoking model-based development formalism's advantages while covering for their shortcomings.

Additionally, while IOPT-Flow is not a commercial framework with a vast and varied set of tools, such as LabView or Matlab/Simulink, it also does not impose expensive license costs or large applications that need to be attached to the user's personal computer. Nonetheless, it offers a higher-level abstraction level than VHDL while being less verbose, however, it is still an abstraction level below most languages used in High-Level Synthesis (HLS) which results in generated code that is more optimized. Lastly, it continues to be developed with extensions as soon as this year, which include a GUI builder.

This work aims to extend both the DS-Pnet formalism and the library of its tool framework (IOPT-Flow) to include asynchronous interfaces. In doing so it should also be possible to specify different time-domains to fully realize a GALS system. The intended result is an environment for quick development/prototyping of GALS systems, further extending the capabilities of IOPT-Flow allowing the development of new and different systems.

1.3 - Document Structure

This document is organized in five different chapters, where the first is a quick introduction to the document of which this section belongs. Chapter two is a literature review of the subject at hand, synchronous and asynchronous circuits, Globally-Asynchronous Locally-Synchronous (GALS) circuits and their different architectures, already existing tools for FPGA design and the DS-Pnet and the IOPT-Flow framework. Chapter three provides a description of the asynchronous interfaces that were created utilizing the IOPT-Flow framework, five different types of interfaces were created, with each following either a 2-phase or a 4-phase communication protocol, the five categories of interface are Simple, SimpleBuffer, BurstBuffer, Data and Databuffer of which the first three transmit events and the latter transmit data. Chapter four, first, provides an analysis of the current DS-Pnet language and IOPT-Flow framework's capability of modelling a

GALS circuit, and second, discusses possible adjustments that can be made to improve its ability to accommodate the modelling of a GALS circuit. Chapter five presents a set of simulations that aim to validate the created components and alterations made to IOPT-Flow framework's simulator. Lastly, Chapter six offers a conclusion and proposes future work.

2

Chapter Two - Literature Review

2.1 - Sequential Circuits

Sequential circuits distinguish themselves for their capability of remembering previous states, meaning the value of the outputs is not only determined by the value of the inputs but also by the value of the previous states. Two types of sequential circuits will be further explored bellow, namely synchronous and asynchronous circuits. [12]

2.1.1- Synchronous Circuits

Synchronous circuits are sequential circuits that require a global clock signal to advance to the next state. For this reason, this type of circuit is predictable and by consequence safer to design around, thriving over its asynchronous counterpart in most of today's circuits. [12]

These circuits separate themselves by their simplicity of design and understanding and their predictability that results in safer and more robust systems that can more easily avoid hazards.

However, the global clock signal brings some disadvantages as well, mainly due to the fact that this signal is required to reach every component simultaneously. That is to say, the speed at which the circuit can run is limited by this signal. The clock signal itself is limited by the longest path in the system requiring to account for the worst case scenario and clock skew. Furthermore, since the whole circuit is stimulated by the global clock signal additional energy is spend

on unused sections of the circuit, generating a large electromagnetic pulse. Additionally, as a result of the need to design the clock signal specifically for the circuit in hand the circuit is not easily adapted or repurposed. [1][13]

2.1.2 - Asynchronous Circuits

Asynchronous circuits are sequential circuits that are not governed by a global clock signal. This allows them to progress as fast as its physical capabilities allow. Furthermore, the absence of a global clock signal also implies lower power dissipation and lower electromagnetic emissions. The need to optimize rarely used paths is heavily devalued allowing design time to be invested elsewhere for a higher overall gain since the worst-case scenario does not limit the entire system. External outputs are more easily accommodated in comparison to its counterpart due to its asynchronous nature. It also exhibits a better potential for migration which could cut down on design time since sections from other works can be adapted. [13][14]

While asynchronous circuits have only been used as small circuits such as peripherals, due to low scalability in synchronous circuits, have been gaining increasingly more attention. [13]

However, this type of design is challenging since hazards and race conditions are a constant issue and their lack of predictability makes them harder to test, mainly due to a lack of design tools. Other issues involve intermediate states, disparate arrival times cause different results and further time spent during handshake protocols which ultimately results in a necessity for a larger number of transistors. [14]

2.2 - Globally-Asynchronous Locally-Synchronous (GALS) Circuits

Globally-Asynchronous Locally-Synchronous systems aspire to achieve a middle ground between synchronous and asynchronous circuits taking advantage of qualities from both sides. The main principle is that there are synchro-

nous islands on an asynchronous environment. That is, the whole system is divided into multiple blocks and each block has its own local clock, however the blocks communicate with each other asynchronously.

Over the years, the increase in die size, transistor count and complexity has increased the amount of design effort put into the clock distribution. It's a challenging task to distribute a low skew clock with high frequency. A GALS design solves this issue by replacing the global clock with multiple local clocks, inherently generating lower electromagnetic impulses. Having multiple blocks isolated from each other implies that the blocks are independent and, for this reason, can be repurposed, this is the main advantage GALS exhibits. Furthermore, the elimination of the global clock means that not every section of the system consumes energy, only the active blocks do, while by itself a GALS system doesn't mean a lower consumption of energy it allows for the creation of more energy sensible systems by fine-tuning the local clocks and even pausing the clock while it is not necessary. [1][2]

Despite the issue not being as severe as in a fully asynchronous design, the lack of tools for designing and testing are the biggest hurdle for a GALS approach, however, some of the tools used for synchronous design can be adapted or used for partial testing. With the addition of asynchronous interfaces between blocks, the partition of the blocks becomes a critical aspect of the design process that requires considerable attention and design effort, furthermore, latency and some loss in throughput is to be expect when in comparison to a fully synchronous design, due to the crossing of timing domains. [6]

In [4], an older but still relevant article, after many years of research in the subject of GALS some observations have been made. Contrary to speculation at the time, GALS systems are not automatically faster, smaller or sufficient to reduce power. Each module can be fine-tuned to run at its own optimal rate. However, communication between blocks incurs a penalty. Large systems with rarely used slow paths can profit from a GALS methodology. Where GALS' true advantages lie, is on its methodology, the ability to break down complex large systems in multiple smaller and simpler blocks. Blocks become re-usable which allows for the exchange of blocks according to specifications. Inherently, a GALS system can be more easily integrated due to its ability to communicate through

an asynchronous environment. A GALS methodology might be the only approach for large SoC (system-on-chip) systems.

2.3 - Architectures for GALS

A GALS system can be characterized by the way the synchronous blocks interact with each other. The two main schemes utilized are the introduction of an asynchronous wrapper that envelops the block enabling it to communicate asynchronously and the introduction of a FIFO buffer that hides the synchronization problems. A third scheme will be presented; however, this scheme is far less popular than the other two, simply put each block communicates with a central storage unit as opposed to with each other.

In figure 2.1, an overall GALS architecture graph is presented based on the literature found, it's important to note that many of the techniques utilized for the FIFO-based scheme may also be utilized for the Asynchronous wrapper scheme; however, the most commonly found in literature was the clock control.

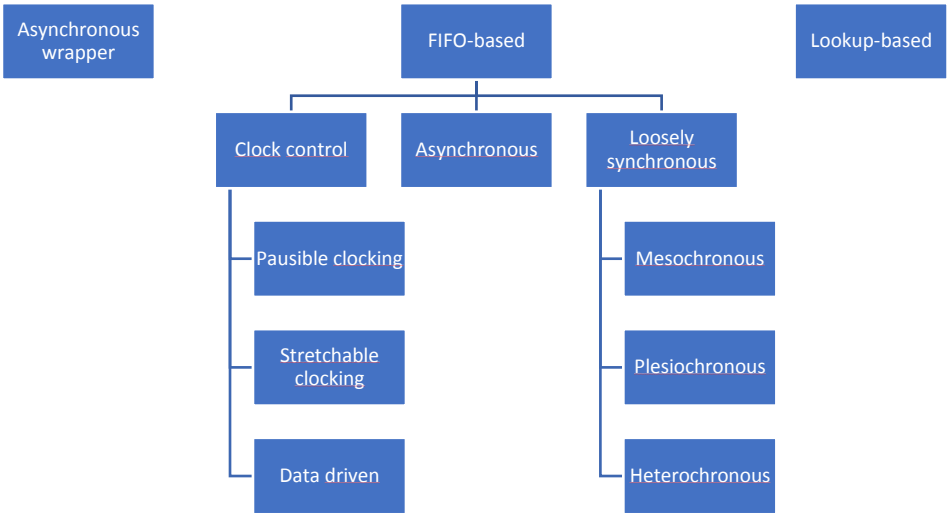

Figure 2.1 - Graph of GALS architectures.

2.3.1 - Asynchronous wrapper

The asynchronous wrapper allows communication via use of handshaking schemes, input and output ports are added to the synchronous block that handle request and acknowledge signals to assure that data can be safely transmitted.

These ports are usually in control of the clock generator and whenever it is not safe to transmit data the clock is stopped and the block ceases operation.

These handshakes require additional power and limit performance since only one data value can be transmitted at a time and multiple signals are required for each value. For these reasons this architecture as fallen in popularity. [7]

In [3], each synchronous island is enveloped by an asynchronous wrapper that contains one clock generator and a controller for each input and output. To avoid metastability during data transfer between synchronous islands the clock generator needs to be pausible, so that whenever data transfer coincides with a sampling clock edge the clock can be paused. However, such incidents are rare and have little impact on throughput.

Information leaving or entering the synchronous island is accompanied by a pair of request-acknowledge signals. A four-phase protocol is used in which a handshake cycle has 4 sequential events (req↑, ack↑, req↓, ask↓), were information is guaranteed to be valid between req↑ and ack↓.

The local clock should be large enough to accommodate the worst-case scenario for its local circuit, same as any synchronous system. A pause request will delay the next clock cycle stretching the low phase of the clock. A mutual exclusion element is necessary to arbitrate between requests, for this reason the signals are required to be persistent.

Two subdivisions of port controllers are described, demand-ports (D-ports) and poll-ports (P-ports). These types of ports differentiate from each other by the way they influence the clock generation, D-ports are useful whenever the synchronous island cannot continue to function without new data, the clock is paused in order to prevent unnecessary power usage. P-ports are useful whenever available data is not immediately needed, the clock is only paused to guarantee that information is safely transferred. Simply put, D-ports have a “sleep while waiting” design and P-ports have a “proceed while waiting” design.

2.3.2 - FIFO-based scheme

The usage of a FIFO buffer connecting two synchronous blocks hides the synchronization problem, the blocks only need to be able to send and request

data from the FIFO. The introduction of the FIFO buffer increases area overhead but also inherently means that a synchronous block does not need to wait for its receiver to be available it only requires for the FIFO to not be full, this leeway increases throughput. There are multiple design styles through which blocks can achieve communication with the FIFO, either clock control where the clock of the block is affected as to avoid metastability, asynchronously through the use of synchronizers or loosely synchronous methods were the bounds of the frequency are known.

It's important to note that some of the methods described could be applied without the FIFO but were either not intended that way or benefit from its usage.

2.3.2.1 - Clock Control

Multiple clocked synchronous domains can be controlled by clock control techniques in order to communicate. The synchronization strategy stops the clock when data transfer takes place in order to avoid metastability. Different clocking schemes can be applied, each dictate how and when the clock is stopped. The clock generator is controlled by asynchronous port controllers similar to the asynchronous wrapper, however these ports do not communicate with other ports, instead they communicate with the FIFO buffer. Input and output ports are implemented accordingly to the amount of inputs and outputs. To ensure safe transmission computation on the synchronous block is halted during communication between blocks. [15]

In [16], an early implementation of a pausible clock control scheme is described, of which the principles still stand. Mutual exclusion elements and arbiters are described as well as their role in the circuit, these elements are responsible for the arbitration between the internal clock's sampling edges and external signal transitions, since they share a common resource. In order for mutual exclusion elements to function effectively signals need to be persistent. A ring oscillator substitutes the commonly used crystal oscillator due to the added possibility of pause. Furthermore, an arbiter is required for multiple inputs. As many connections as necessary can be implemented, however, a large amount will make the arbiter impractical. A FIFO is used between blocks for one direction communication and two for bidirectional communication. Communication with

the FIFO is done through handshake signals. Using SPICE, the behaviour of a simplified architecture was simulated, consisting of two modules communicating in a single direction through the use of pausible clock control. The FIFO is not necessary to establish communication; however, its inclusion helps to smother bursts of data common when dealing with modules that function at different rates. During simulation, the receiver operates at higher clock frequency than the sender, so the FIFO never fills up.

In [15], three different clocking schemes are described and compared, them being:

- Pausible clocking scheme

Characterized by its free running clock. A mutual exclusion element is used to interrupt the clock whenever data needs to be transferred, providing safe communication.

- Stretchable clocking scheme

Similarly to the pausible clocking scheme it has a free running clock. The difference is that the pause is anticipated. This adjustment leads to an increase in throughput. The computational segment is paused once a request signal is present.

- Data driven clocking scheme

Contrary to the previous schemes, the clock is not free running, the clock edges occur with the presence of input data. Resulting in a reduction in power consumption due to only being active when data is received.

In [15], experiments show that with the increase in frequency of the producer block (block that produces data to be sent) the number of pauses also increases, as expected since lower frequencies mask the delay caused by the data transfer. While a pausible clocking scheme and a stretchable clocking scheme have similar number of pauses, once the frequency of the producer block is higher than the frequency of the consumer block (block that consumes data it receives) the latency, of pauses, of the stretchable clocking scheme is much longer than those of the pausible clocking scheme. Furthermore, once the frequency of the producer block surpasses the consumer block the FIFO will tend to fill, once

full more pauses will occur, since pauses take longer to resolve on a stretchable clocking scheme it is much more impacted by a full buffer.

It can be observed that the throughput increases with the frequency of the producer block until the FIFO is full, for this reason the ideal frequency is such that both blocks run at a high frequency without filling the buffer. However, the data driven has the best throughput followed by the stretchable clocking scheme.

Power consumption increases with the increase in frequency, as expected. Data driven offers less power consumption as well, while the other two exhibit similar results.

2.3.2.2 - Asynchronous

In order to avoid metastability, synchronizers are added between the timing domains. These synchronizers consist of a battery of flip-flops that resolve the problem but add latency, for this reason while it is a simple solution it is not the most elegant nor the most efficient.

In [17], communication across an elastic network fabric is analysed and evaluated, measuring efficiency and overhead. A NoC (network-on-chip) possesses a large potential that is further enhanced through the use of GALS methodologies. As a consequence, many design aspects need to be considered, and an important aspect is the interface protocol between the network and the cores. Latency insensitive protocols (LIP) have some potential advantages in GALS systems, therefore it has been subject of testing in elastic systems.

Elastic systems are similar to clocked systems, however, two control wires (valid and stall) implement a handshake between sender and receiver. Stall signals propagate backwards and allows that a second data arriving at the flip-flop can be stored in the second latch, this is possible due to independent control for each of the two latches that comprise a flip-flop. Three interfaces were implemented, them being a fully synchronous interface, a clocked interface and an asynchronous interface.

The network fabric has low latency, high throughput and a static worst-case latency for all transmissions due to significantly different targets when compared to other NoC designs. Since the blocks run at their own optimal frequency, data

transmitted must be synchronized, this synchronisation has a significant overhead and is an important factor in a GALS system.

Two synchronizing interfaces are compared the aforementioned clocked and asynchronous. Each composed of FIFOs and synchronizers. Both FIFOs are simple four-deep linear shift, the clocked FIFO is implemented as four elastic half buffers and takes four full clock cycles to propagate data, where in the asynchronous FIFO, data propagates very quickly from tail to the head, this asynchronous FIFO is implemented using domino gates.

The clocked synchronizer is implemented with two flip-flops and some additional logic for handshaking. The asynchronous designs use a "fast synchronizer" that substitutes flip-flops for mutual exclusion elements, being able to send data and perform handshakes within a single clock cycle, whereas the clocked synchronizer takes two clock cycles in each direction. However, while a clocked interface requires a single synchronizer the asynchronous interface requires two, this happens due to crossing the different timing domains, where the asynchronous FIFO behaves like its own domain.

The three designs were implemented onto a chip with four components, the computational module, network interfaces, network fabric and a scan interface. Multiple chips can be used to emulate a SoC design, the SoC design was the intended application, and however, due to space limitations the system had to be broken down.

Two sets of results were had into account, characterization of the elastic network and a comparison between clocked and asynchronous interfaces.

In the former, the test is done in a way that bypasses the synchronizing interface, this way the links are operating at their maximum utilization factor. The latency is effectively equal to the number of buffering elements, that is to say that this test is meant to provide an analysis of the system at its ideal performance without the latency introduced by the synchronizing interfaces. The latter set of results has the computational module and the network fabric clocked at different frequencies, this is done to compare the performance of each interface. The asynchronous design exhibits reduced latency and a close to no synchronization cost, as speculated previously.

2.3.2.3 - Loosely Synchronous

By exploiting known bounds of the frequency, it is possible to only allow communication whenever it is safe to transmit losing only a few cycles from time to time which deems handshaking unnecessary resulting in a higher throughput. However, timing analysis between blocks is required and changes to the clock frequency require further timing analysis. This design style can be further classified according to its timing relationships, such relationships have been classified by Messerschmitt [18] and are as follows:

In [18], some basic concepts are first described, a Boolean signal represents one of two possible values, due to physical limitations there is an undesired period of transition between states, however with the goal of taxonomy of such signals this and similar behaviours will be ignored, in order to keep such classifications simple. Furthermore, a Boolean signal's frequency and phase are those of its associated clock. When a signal has a constant frequency it is said to be *isochronous* (from the Greek root "iso" meaning "equal"), if it is otherwise non-constant, the signal is designated *anisochronous* (meaning "not equal").

Secondly, when two signals are present we can define their relationship.

When two signals are both *isochronous*, have the same frequency offset and have a zero instantaneous phase difference, they can be designated *synchronous* (from the Greek root "syn" meaning "together"), any signal that does not follow these criteria is then designated *asynchronous* (meaning "not together"), any two *anisochronous* signals are therefore *asynchronous* expect when they exhibit the same transitions, however this is a special and unusual case, only possible if both signals originate from the same circuit.

For a GALS architecture the relevant relationships are the *asynchronous* relationships, because locally synchronous blocks communicate through an asynchronous environment, such relationships can be further categorized.

Two isochronous signals with the exact same frequency and yet with a stable but different phase, may be designated *mesochronous* (from the Greek root "meso" for "middle), an example would be two signals sourced from the same generator but suffering varying delays.

Two signals that have nominally the same average frequency, however not exactly the same, such relationship is designated *plesiochronous* (from the Greek root "plesio" for "different"). This happens when two different sources generate the same signal.

Finally, if two signals have nominally different average frequencies they are designated *heterochronous* (from the Greek root "hetero" for "different"). However, a further separation has been made by [6], a *heterochronous* relationship can be classified as either *ratiochronous*, if the frequency of one signal an exact rational multiple to the other signal and there is a predictable periodic phase relationship due to a common source clock, or otherwise *nonratiochronous* in which the explained scenario is not true.

In [19], a network-on-chip (NoC) is introduced in a multiprocessor system-on-chip (MPSoC) to realize dozens or hundreds of CPU cores. Each packet is segmented into flits with a 26bit header and a 64-bit payload. A 2d-mesh topology with XY-routing is used. Three different behaviours can be configured on the router, synchronous, mesochronous or asynchronous.

For a synchronous router, incoming flits are registered, then flits are buffered in a FIFO queue to smother possible bursts of data. The FIFO is also used for flow control. Flits can be transmitted until the FIFO is nearly full, nearly as opposed to full to accommodate some latency in signal transition. The flit is then transmitted to its respective output. Any flit as a minimum latency of two clock cycles to transverse through the router.

For a mesochronous router, the synchronous router is extended with mesochronous links to allow unknown phase shifts between routers, this relaxes the global clock tree. However, the register has to be replaced by a mesochronous synchronizer, a tightly coupled mesochronous synchronizer (TCMS) is used. Additionally, three clock cycles are required, which results on an increase in the depth of the FIFO by one.

Lastly, for an asynchronous router, communication is carried out through a two-phase handshake protocol, in which any change in the request signal implies a new flit. The clusters still operate synchronously and may be operated by their own independent clock. Synchronizers are needed to cross domains, addition-

ally, metastability needs to be accounted for. The block responsible for the routing decision is pure combinational logic and is the same for all designs. For flit buffering, mousetrap circuits are used in every port. Multiple mousetraps may be combined to make a FIFO, enabling higher throughput at the expense of latency. The biggest challenge is arbitration between requests, and for that reason an arbiter is added.

For a single cluster, a synchronous and a mesochronous router occupy about the same area, while the mesochronous router replaces the register for a synchronizer effectively increasing its size it compensates for it by having less wires connecting each router, due to relaxed timing constrains. The asynchronous router requires only 42% of the synchronous router's area, due to the usage of latches instead of flip-flops.

Power consumption is determined by gate-level simulations of two connected cluster nodes. Power consumption was registered while both on an idle state and during active communication. The synchronous router consumes the most for both an idle state and during active communication, the mesochronous router achieves slightly better results and the asynchronous router dissipates only 22.4% of the synchronous router's power while on idle state and 53% during active communication.

Contrary to the other two designs the asynchronous router's latency and throughput are not dependent on a global clock, it is instead only limited by the local handshake and wire delay. On average, the asynchronous NoC shows 15% increased maximum throughput and 25% decreased minimum latency, the asynchronous router benefits from fast handshaking handled by mousetraps. Additionally, the mousetrap in combination with fast arbitration by the asynchronous join elements results in a better handling of packet collision.

Lastly, a comparison between the three designs for the global clock tree on top level is made. The global clock three needs to connect to all cluster nodes, however timing checks need only to be met for the synchronous design. With the intent of analysing scalability, at this point, only the results for the clock tree top level are had into account. Once again, while the mesochronous design exhibits a small advantage over the synchronous design, the asynchronous design has a

much lower power consumption, 25% for 256 CPUs, this is due to smaller clock trees within the cluster nodes.

All in all, while modern synchronous methods allow for some scaling, a mesochronous or asynchronous design is more efficient.

2.3.3 - Comparison between different FIFO

In [20], with the apparition of Network-on-Chip (NoC) as new approaches for high throughput and scalable design of Multi-Processor Systems-on-Chip (MPSoS), FIFO buffers stand as an important component for asynchronous design, namely GALS.

A Muller pipeline is structurally simple, consisting of a simple C element and an inverter gate, once full only every other stage stores data and each stage should tightly interact with neighbouring stages.

A fully-decoupled latch controller can be introduced so that all stages store data when the pipeline is full and each stage handshaking on the input channel completes without any interaction with the output channel. But, having a more complicated controller reduces throughput and slows down the handshaking.

Such pipelines require combinational logics between stages. However, in a FIFO data is stored without processing, furthermore data is only read and written on a single stage. The result is less power consumption in comparison with a pipeline.

An option for a FIFO implementation is a modular FIFO which follows the asP* handshaking protocol and is synthesizable by standard cell logics without any particular asynchronous cell. However, it requires some timing considerations due to not following the conventional 4-phase handshake protocol.

Another option is a FIFO based on a fully-decoupled pipeline. The FIFO exhibits the throughput of a fully-decoupled FIFO in addition to some multiplexing and demultiplexing penalties.

It's of note that all presented FIFOs are all self-timed.

Furthermore, [21] proposes a FIFO architecture consisting of a RAM, read and write pointers, full and empty detectors and input and output handshake

controllers. This design completely follows the 4-phase bundled data handshake protocol.

Data is written into the RAM in the slot indicated by the write pointer while the RAM is not full. An ack signal is sent when the RAM is not full, and the request is asserted, additionally the full and empty detectors are updated. When the RAM is not empty a request is sent to the output port indicating new data is available.

Memory is implemented with latches and pass-transistors. However, it could be replaced by a memory plane to increase performance. Addresses are one-hot, which discards the need for an address decoder. Handshake controllers consist of an asymmetric C-element and n AND gates for input channel. Once a request is asserted the signal is then blocked until the FIFO has free space. When free space is available and ack is zero it enables the load signal regardless of the request signal. FIFO requests to send out its data when it is not empty once the ack signal is lowered. Upon reset the read and write pointers should be zero as to point to the first memory slot. When both pointers are the same it indicates that the FIFO is empty or full. To distinguish between full and empty, one extra bit is added to both pointers and they toggle whenever their pointers circulate and become zero again. Since the write pointer is triggered on the ack's rising edge the not-full flag can be evaluated before the ack signal lowers. However, if the FIFO is full the detector should not change state until the read operation ends, in order to avoid read/write races.

Through accurate SPICE simulation the proposed FIFO and some other designs, the aforementioned Muller, Fully-Decoupled, asP* based and Domino-controlled, were simulated with the intent to evaluate them. Verilog was used at gate level for describing some available FIFOs that were then used as buffers.

All of the FIFOs require elaborate and engineering timing assumptions for reliable operation. No wire delays were had into account. A random Galois LFSR sequence of data words are used as to obtain results that don't depend on certain inputs.

All FIFOs were made from the same library and the conditions were identical. Results for comparisons should therefore be reasonable. Furthermore, different depths were tested.

The resulting throughputs seem to be sorted by complexity were the highest throughput is achieved by the simpler FIFO design, this is, the proposed FIFO. Only the Muller and the Fully-Decoupled pipelines maintain throughput with the increase of depth. However, the Muller and the Fully-Decoupled pipelines have a sharp increase in energy per word with the increase of depth, Domino's energy consumption is much lower than the remaining due to a lack of Flip-Flops.

The proposed FIFO also seems to have a lower latency than all the others in must conditions, however if the FIFO is empty the Domino-controlled will have a slightly less latency.

For further analysis, the proposed FIFO and the Domino-controlled are employed as buffers in routers of a 4x4 mesh network. The NoC used is fully asynchronous based on QNoC and ASPIN using two cascaded Flip-Flops to connect synchronous IP cores to the asynchronous network.

Each router addressable with a two-dimensional number contains five ports connecting it to its neighbours and to the local IP cores. To route packets between ports the asynchronous NoC uses distributed X-First algorithm to guarantee the in-order-delivery property. Furthermore, each packet is divided in flits and when a header flit of a packet is received the packet is forwarded to the corresponding output port.

Parameters from the SPICE library are extracted and back-annotated in a Verilog HDL library.

The router is modelled in gate level using the mentioned library and the 4x4 mesh network is constructed by behavioural model local IP cores.

Both the proposed FIFO and the domino are modelled and used as buffers of the routers' input ports. FIFOs are 5-stages deep with 34bit data word length. The results show that the proposed FIFO exhibits less packet latency and more network saturation threshold compared to its Domino controlled counterpart.

2.3.4 - Lookup-Based Scheme

A storage mapping unit (SMU) is added to each block. These SMU allow the block to read and write from a central storage, removing the need to exchange signals. Removing inter-component signals simplifies design, additionally multiple data values can be stored without having to wait. [7]

In [21], a Kahn Process Network (KPN) consists of processes that communicate via point-to-point unbounded FIFO channels. A KPN model can be validated via functional simulation. Then, it is possible to construct a GALS system from a KPN model, by refining it with a refinement components library. The communication between processes is facilitated by asynchronous communication with a shared on-chip storage location. However, the KPN properties must be preserved through the refinement.

Refining a KPN model into a GALS model requires that unbounded storage elements are replaced by bounded storage elements, additionally blocking read and blocking write conditions shall be enforced on the process. To achieve these two requirements an on-chip lookup storage unit (LUS) is used to store data, a LUS is bounded and allows communication between processes since all processes read from and write on the LUS, furthermore each process is augmented by storage mapping out unit (SMU).

The LUS is split into different segments, each segment is used for communication between two processes. Segments have limited size that should be enough to accommodate delays on behalf of the consumer process. An index points to the location of a data segment, additionally the bounds of the segments are known. Data stored has a single control bit that indicates whether the data is valid. To protect data integrity each segment has a read and a write controller.

A SMU is added to each process, this SMU governs the execution of the process. It is capable of storing data relevant to the process, this is, inputs and outputs. Furthermore, the SMU is responsible for communicating with the LUS, each communication is established through control and data signals.

By maintaining the Kahn principles, each process is shown to be deterministic, continuous and monotonic, additionally the behaviours of the processes in the GALS architecture are latency equivalent to its KPN counterpart.

2.4 - Tools for the Development of Circuits

2.4.1 - Hardware Description Language

A Hardware description language (HDL) is used to describe the structure and behaviour of electronic circuits. The two most popular design languages for FPGA are VHDL and Verilog, they are capable of description, synthesis, simulation of logic circuits and multiple levels of design description. [22]

In [22], a comparison between Verilog and VHDL is presented, although the paper's focus is on VHDL. Verilog is described as less verbose, easier to learn and to model abstract hardware structures, while VHDL is described as having good documentation, synthesis and simulation capabilities, with emphasis on reusability.

However, other languages may be used to describe circuits. MyHDL, a Python package that outputs to either Verilog or VHDL, it still requires the developer to understand design techniques while leveraging the power of the Python language. SystemVerilog is an extension of Verilog in combination with OpenVera and SuperLog, that can be used for developing the RTL for the FPGA or it can be used for verification. [23]

Many commercial simulators are available such as Xilinx's ISE and Vivado, Intel's Quartus II, Mentor Graphic's ModelSim and Synopsys' VCS, as well as some free and open-source simulators such as Icarus Verilog, Verilator and QUCS.

2.4.2 - High Level Synthesis

On one hand, describing components in a hardware description language (HDL) allows the creation of specialized hardware with great potential; however, advanced hardware expertise is required and even then, the process is cumbersome, resulting in long development times. [24]

On the other hand, High Level Synthesis (HLS) adds another layer of abstraction to the design process. Many tasks are automated, resulting in a greatly improved design time at the expense of some chip area and performance. Some of the automated tasks include, resource allocation, scheduling and resource binding. Additionally, interface synthesis is handled, between the generated circuit and periphery. [24][25]

While a HLS tool will be capable of generating a hardware realisation the result will be bloated and inefficient. A software mind set does not always translate well into a hardware environment, such as, algorithms based on pointer arithmetic or recursion, will have issues since memory is distributed and many variables are stored in registers. Additionally, any concurrent system design will cause issues due to the need of hardware synchronization [26]

The amount of code written by the designers is heavily reduced, saving time and reducing risk of error. Additionally, some HLS tools can generate test benches to reduce verification time. [25]

Field Programmable Gate Arrays (FPGAs) and HLS make a strong combination for rapid prototyping and a fast time to market. [24][25]

HLS tools start from a high-level software programmable language (HLL), such as C, System C or Matlab, and generate a circuit specification in HDL that performs the same function. [24]

In [25], a survey on the HLS tools compares them on criteria such as abstraction level, learning curve, documentation and design exploration. However, this survey is from 2012 in a field with active research where the tools are constantly evolving. Nevertheless, the tools analysed are Xilinx AccelDSP, Agility Compiler, AutoPilot, BlueSpec, Catapult C, Compaan, C-to-Silicon, CyberWorkBench, DK Design Suite, Impulse CoDeveloper, ROCCC and Synphony C.

In [24], a more recent survey from late 2015, offers an evaluation of past and present HLS tools both commercial and academic, as well as an in-depth evaluation and discussion of some of the selected tools in terms of performance and area metrics. Some of the academic tools include LegUp, Bambu and DWARV, while some of the commercial tools include VivadoHLS, CyberWorkBench, Synphony C and ROCCC.

2.4.3 - GALS Specific Tools

2.4.3.1 - Introduction

Some GALS specific tools and methodologies exist; however, these tools are still recent and have room to grow. To mention a few, in [27], Malik et al. present a new system-level programming language SystemJ. SystemJ is a multiclock Java based language that supports the GALS model of computation. Later, in [28], Malik et al. propose DSystemJ, it is an extension of the SystemJ language that enhances it with dynamic creation and process mobility. Another method may be to extend synchronous languages to develop GALS systems, taking advantage of the fact that the systems have a trustworthy base. Such is the case for Esterel, Lustre and Signal languages that were proposed for distributed systems development in [29].

2.4.3.2 - IOPT-nets and IOPT-Tools

Furthermore, other Petri net-based modelling tools have been used for the development of GALS. Due to a scarcity of Petri net-based tools for embedded systems design surges the Input-Output Place-Transition Petri-net class that extends the Place-Transition Petri net Class with input and output capabilities, guard functions and output expressions. IOPT-Tools (figure 2.2) is a web-based framework that employs the IOPT Petri net class to support the implementation of embedded system controllers, it possesses a graphical editor complemented with model-checking and system verification tools and is capable of generating either C code for software solutions or VHDL code for hardware descriptions. Later, a model-based development approach for Globally-Asynchronous Locally-Synchronous Distributed Embedded Systems was proposed relying and extending the IOPT Petri net class and its associated framework, IOPT-Tools. [30][31][32][33]

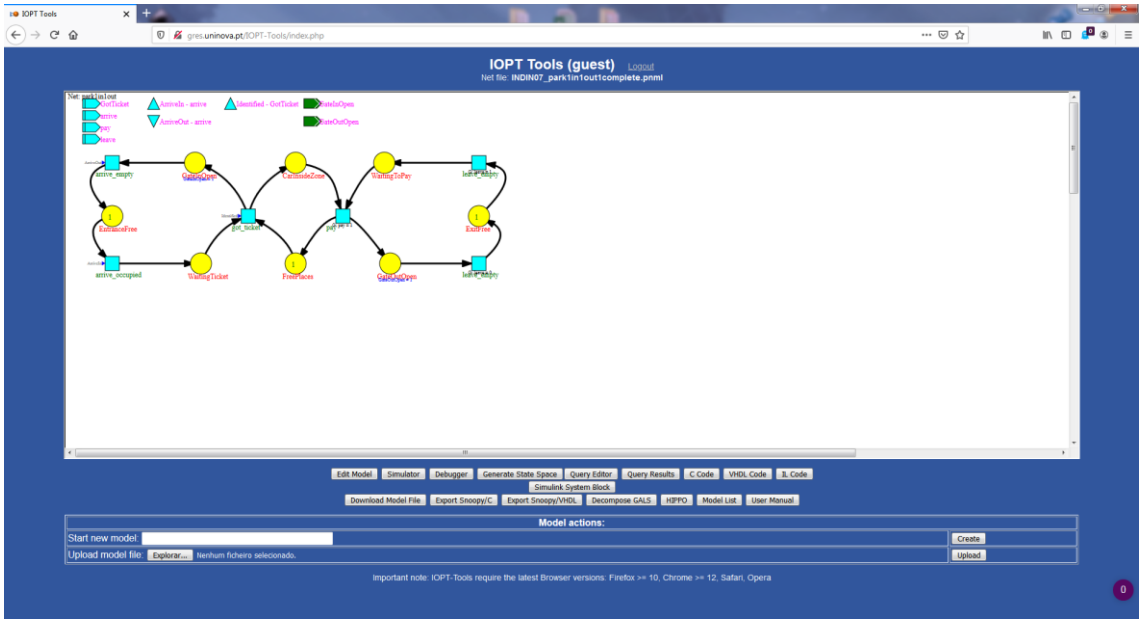

Figure 2.2 - IOPT-Tools Framework browser window.

2.4.3.3 - A Toolset for Modelling and Verification of GALS Systems.

For example, in [8], a toolset for design and verification of GALS systems is presented, this toolset is based on Communicating Reactive State Machines (CSRM), a visual formal language. Furthermore, it integrates a graphical editor, simulator and verification engine. A CRSM is a network of nodes. These nodes are locally synchronous, can execute concurrently and communicate with each other on point-to-point channels.

The model is verified by translating the system into Promela. Promela code ensures that the status of signals and states in the system are evaluated correctly, additionally, each node should include a reactive kernel and an environment process. Spin code has been modified to automatically and counter examples generated are translated into traces and displayed in the simulator.

2.4.3.4 - Calibrating a Local Clock

In [34], a flexible method that through dedicated hardware circuits and off-line software analysis allows local clock generators to be calibrated. However, this solution requires each clock generator to be connected by a serial synchronous low-speed link. Each clock generator has allocated a unique address in the chain. A main clock controller exists and is connected to the serial chain. This

controller is responsible for generating the serial sequence that calibrates the selected clock generator to the desired frequency. The serial link communication is achieved with frames that carry the commands. Two types of command are enough for a ring oscillator, programming frames and calibration frames. Programming frames write initial configuration values to the local clock divider, and calibration frames perform incremental modification of the delay chain of the ring oscillator.

Configuration and calibration is performed by a FSM sequence based on both the data read from the ROM and the local clock period. A configuration signal is sent to each local clock generator and afterwards calibrated. A calibration command can either increment or decrement in small amounts.

The clock parameter is stored in the ROM so they can easily be accessed by the controller. These values are computed by an external tool that makes sure the measurement of the clock frequency is kept to a minimum without impacting the overall programming time.

A system with three clock generators was simulated to test different target frequencies and different reference clock frequencies. It can be noticed that, a higher reference clock produces better results in both programming time and fewer errors, however it shouldn't be so high that it doesn't accommodate a significant number of local generated clock periods. Higher local generated clocks also produce better results because less time is needed to measure them. A possible approach to achieving faster programming time would be to have a higher frequency that are then divided locally.

2.4.3.5 - Adapting Existing FPGA Tools for a GALS System

The authors of [35] defend that, Field Programmable Gate Arrays (FPGA) are becoming larger and distributing a global clock is becoming a burden, for that reason GALS are becoming an attractive alternative, however the lack of tools is a pressing issue. And so, methodology for implementing asynchronous GALS components using existing FPGA resources becomes an option.

A GALS system requires asynchronous elements to establish asynchronous communication between synchronous islands. For a pausible clock, a delay element and a Muller-C element are indispensable, an arbiter, while not essential, greatly simplifies communication.

Look-up tables (LUT) and Flip-Flops (FF) are the only synchronous FPGA resources numerous enough to enable implementation of such asynchronous elements. However, the LUT must generate hazard-free output and have close delay matching between each input and corresponding output.

The asynchronous component implementation is based on a Place & Route (P&R) process that ensures component functionality. FPGA synthesis tools optimize redundant paths which may not be desired when designing asynchronous components, and for that reason needs to be constrained. These constrains are necessary for GALS communications.

A delay element may be preserved in a FPGA by either creating a group constraint with at least two elements or creating a KEEP constraint. However, the first option is preferred due to KEEP not allowing P&R. A delay element required two LUTs in an empty group and otherwise a single LUT.

A Muller-C element establishes the precedence of asynchronous events. A Muller-C element only changes its output once every input is the same, which protocols take advantage to establish communication. In order to implement a Muller-C element two options are presented, using a single LUT or using tow LUT and a FF. The first option has the obvious advantage of requiring less resources, however the second option is generally preferred due to its robustness.

The arbiter must grant access to the signal that first arrives, however due to its asynchronous inputs it's subject to metastability which must be accounted for. Three implementations exist. The first is a latch-based implementation on two LUTs and two feedback paths, the second is similar but requires an additional feedback path. The third option is to once again exploit the robustness provided by the FFs, four LUTs and four FFs are required, despite more resources spent and slower action this method is preferred, due to its added robustness.

2.5 - DS-Pnet & IOPT-Flow

Model-based development formalisms based on Petri nets bring several advantages, such as rapid prototyping, simulation and automatic model-checking tools, however, they are not always suitable for real-world embedded system applications as they lack input/output and data processing capabilities. To overcome this, the IOPT-nets and, later, DS-Pnets were proposed. [11]

2.5.1 - DS-Pnet (Dataflow, Signals and Petri nets)

The DS-Pnet modelling language is a directed graph that combines Petri net nodes with data-flow nodes to aid in the development of cyber-physical systems, it offers high level design concepts that hide low level platform details, simplifying and accelerating development. [10]

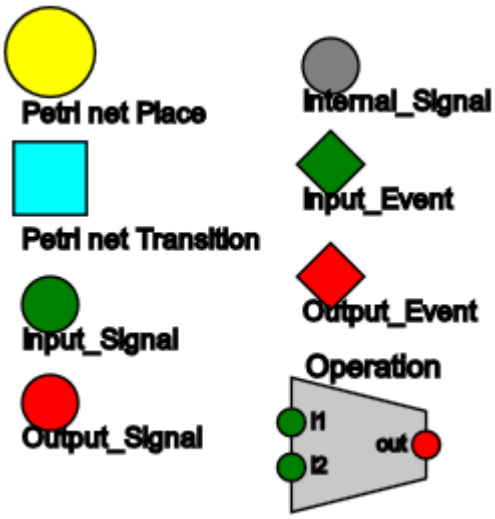

Figure 2.3 - DS-Pnet elements.

It is composed of five types of nodes, Petri net places, Petri net transitions, input/output signals, input/output events and data-flow operations, these are represented in figure 2.3. Additionally, two types of arcs connect such nodes, normal-arcs and read-arcs. [10]

Data-flow nodes, or operations, employ mathematical expressions to calculate new values. Operations are connected to other nodes via read-arcs and each output is an expression where its operands are its inputs. Computational time is considered instantaneous; therefore, feedback loops are forbidden. Counters should be employed to work around this issue. [10]

Input and output signals and events are used to communicate with the real world. Signals have a lasting effect were events are instances. [10]

Petri net places and transitions follow the IOPT Petri net class and are used to control the cyber-physical systems. The state of the system and its evolution is controlled by these nodes. [10]

Normal-arcs are the traditional Petri net arcs used for communication between Petri net places and transitions. Read-arcs transmit data between the remaining nodes. [10]

The DS-Pnet language stands in between HDL and HLS. It is simple for someone without experience to comprehend due to being a graphical language, while avoiding the pitfall of a software mind set and as a result the generated HDL code will be much closer to a handmade one.

2.5. 2 - IOPT-Flow Framework

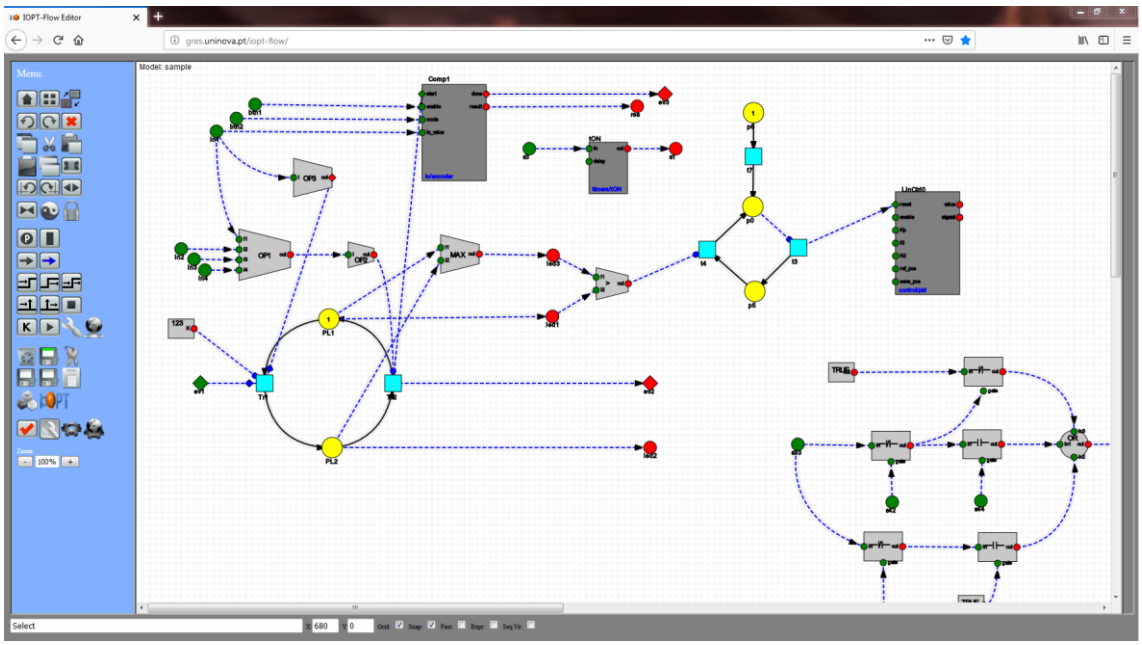

Figure 2.4- IOPT-Flow Framework browser window.

The IOPT-Flow modelling framework is a web-based set of tools for design of embedded and cyber-physical system controllers. The chosen modelling language is named DS-Pnet and it employs elements from the IOPT-Flow Petri nets class and from data-flows. The IOPT-Flow framework, depicted in figure 2.4, is composed of a few tools, namely, a graphical editor capable of model edition, a simulator capable of simulation, a debugger supporting monitoring and remote control and an automatic code generator that produces the final code to run on the prototype hardware. [11]

Upon opening the framework, the user is greeted with the editor tool, it's from this tool that all others are invoked. The window is divided into three sections, the left section is the toolbox where all the tool's functionalities reside, in the middle is the drawing are where the model will be implemented and lastly on the right the selected element's properties will be shown. The toolbox allows for typical edition tools such as element creation, copy and paste, undo and redo, load and save, etc. Additionally, buttons specific to the framework exist, node-fusion, complementary places and automatic semaphore creation, as well as, check syntax errors and invoke the simulator and code generation tools. [11]

The simulator tool, is used to simulate and debug models without the need to compile and deploy code onto physical hardware. Simulations run on the user's web browser using automatically generated JavaScript code. To simplify debugging a simulation may be run continuously or step-by-step. Additionally, simulator history is recorded for later analysis for high speed simulations. This recorded history can either be exported in CSV or viewed as graphical waveforms on the tool itself. [11]

Lastly, the goals of the framework go past modelling and simulation, it is also possible to generate software or hardware code to run on embedded devices. The automatic code generation tools were implemented using XSLT transformations. Currently, XSLT transformations for C, VHDL and JavaScript are available. [11]

2.6 - Literature Review Conclusions

Globally-Asynchronous Locally-Synchronous (GALS) circuits show potential going forward, since they take advantages from both synchronous and asynchronous circuits, in particular its scalability and its modular nature.

Some different architectures exist when it comes to GALS systems, most notably asynchronous wrappers and FIFO based schemes. The former envelops the synchronous block in an asynchronous wrapper that handles all communication with both other synchronous blocks and environmental inputs and outputs, and the latter has the synchronous blocks communicate through a FIFO buffer that covers any synchronization issues.

Currently, a myriad different tools capable of developing and implementing algorithms into hardware each with their own set of advantages and disadvantages.

Labview and Matlab/Simulink have seen heavy use by the academic community with some incursions in the industrial environments. There are huge frameworks with a vast and varied set of tools. However, these are commercial products, with expensive license costs whose applications are comparatively quite large and need to be attached to the user's personal computer. On the other hand, the IOPT-Flow framework can be easily accessed on any simple phone, tablet or computer due to being web based with the more computing intensive tasks running on the user's web browser. Although Simulink supports Petri nets and dataflows, the IOPT-Flow has been specifically designed for the purpose of supporting the design of embedded and distributed cyber-physical systems, bringing high performance implementations and precise semantics. Furthermore, this tool chain allows for reading and writing input and output signals and events without the need for additional blocks supplied by the hardware venders. [9]

The IOPT-Tools support many of the advantages that IOPT-Flow brings, however, it is not as adequate to handle the complex data manipulations due to the lack of dataflow nodes that the IOPT-Flow framework provides. IOPT- Tools has had some development in the field of GALS systems that the IOPT-Flow has not yet. And so, this dissertation aims to develop the IOPT-Flow framework in that direction.

3

Chapter Three - Asynchronous Components

3.1- Communication Protocol

In a GALS system, an asynchronous interface is required in order to transmit information between two blocks. To this end, some components were created, utilizing the IOPT-Flow framework. Each asynchronous interface is composed of a transmitter-receiver pair. As their names imply, information is loaded onto the transmitter and from there sent to the receiver from which it may be later retrieved. Synchronization is achieved through pairs of Request/Acknowledge signals, these signals may communicate either via a 2-phase or 4-phase handshake protocol. A description of these protocols is readily presented, additionally, figure 3.1 displays an UML sequence diagram of a simple example for each of the protocols.

On a handshake protocol, communication is initiated by setting the request signal. Then upon receiving this signal the other component will respond by setting the acknowledge signal. If data is being transmitted, then it may only be valid during a portion of the exchange. [36]

For a 2-phase handshake protocol only two events are exchanged. The initial values of these signals are irrelevant since transitions are recognized rather than the values themselves. Nevertheless, these signals will usually be synched, meaning, they'd hold the same value while not communicating. If after communication both signals are high then the exchange is designated as an up handshake and if, otherwise, both signals are low it is designated as a down handshake. First, the exchange is initiated by the request signal's shift in value. Then,

upon recognizing this transition the other component will shift the acknowledge signal's value. The exchange is considered finished once the component that initiated the exchange recognizes a transition on the acknowledge signal, and so, a new handshake may occur. [36]

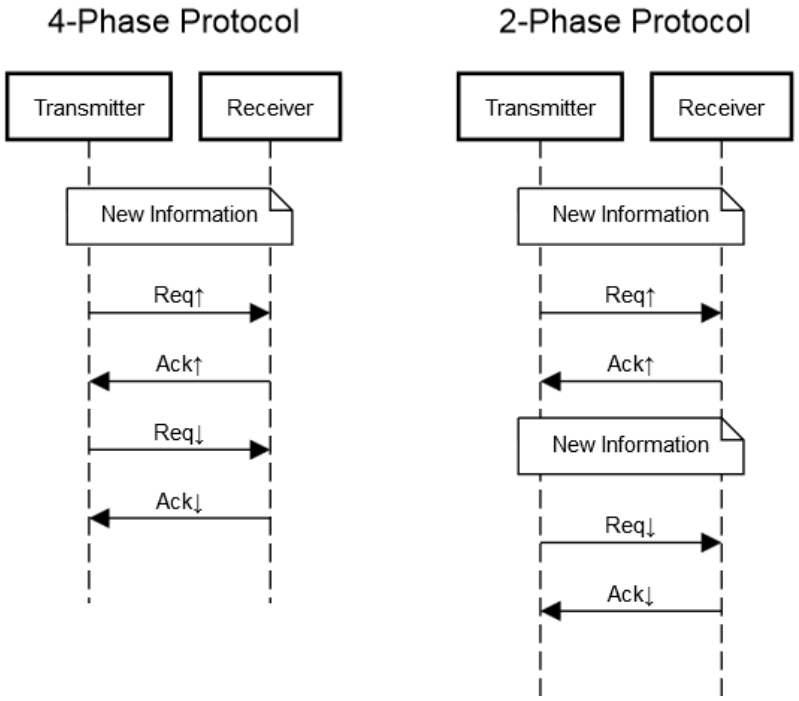

Figure 3.1 - Examples of a push channel for both the 4-phase and 2-phase protocol.

On the other hand, the 4-phase handshake protocol requires four events. It is composed of an up handshake followed by a down handshake. In this protocol, the signal's values are read and not necessarily their transitions. Once the down handshake concludes a new cycle may begin. [36]

When transmitting data, two types of channels may be distinguished, push channels and pull channels. These pertain to when data is considered valid. On a push channel data is valid during the handshake, the sender initiates communication by issuing the data-valid (request) signal. On a pull channel data is valid between handshakes, it is the receiving party that requests data. [36]

For a 2-phase handshake, only one data-valid scheme exists for each channel. For a push channel, data is valid from the moment the request signal transition occurs to the moment the acknowledge signal transition is recognized. As for a pull, channel data is valid from the moment the acknowledge signal transition occurs until the next request signal transition is recognized. [36]

For a 4-phase handshake, three different data-valid schemes exist for each channel, these schemes can be designated as early, late or broad. First addressing the push channel, for the early scheme, data is valid from Req↑ until Ack↑, for the late scheme, data is valid from Req↓ until Ack↓, and lastly, for a broad scheme, data is valid from Req↑ until Ack↓.

Now addressing the pull channel, for the early scheme, data is valid from Ack↑ until Req↓, for the late scheme data is valid from Ack↓ until Req↑, and lastly, for the broad scheme, data is valid from Ack↑ until Req↓. [36]

Furthermore, push channels are more commonly found in literature.

The interfaces created may be placed into five categories. These categories will be explained in detail further bellow and they are Simple, SimpleBuffer, BurstBuffer, Data and DataBuffer. For each category two pairs of transmitter/receiver components were created, one of them follows the 2-phase communication protocol and the other the 4-phase communication protocol. Additionally, a simple communication example was created for each one.

For the components capable of transmitting data, push channels were used and while data is set at the same instance as Req↑ it is only read by the receiver after Req↓. This was the chosen approach in order to relax the wiring. If the data was read immediately upon receiving Req↑, the data wire had to reach the receiver before the request signal, since data is read upon receiving Req↓, the data wire has three events to reach the receiver.

Bellow, some communication graphs will be presented, they will display exchange of signals between the synchronous block and the components described in this chapter. Due to simplicity and quick understanding, only some crucial signals will be visible, by consequence, it is not completely accurate. Additionally, AB1 and AB2, respectively refer to the synchronous block to which the transmitter is attached and the synchronous block to which the receiver is attached.

A Delay event is present in every component in order to artificially simulate communication delay in the framework, this event does not contribute to the goals of the component and should be later removed, and that is to say, it should only be present for simulation while the IOPT-Flow framework is not prepared for asynchronous communication. Additionally, removing the Delay events may

require some simple adjustments in some components, specifically the ones that communicate via 2-phase protocol.

3.2 - Components Description

3.2.1 - Simple Interface

The Simple interface is only capable of transmitting events. In other words, no data is transmitted, instead the transmitter merely notifies the receiver that an event has occurred. Only one event can be transmitted at a time.

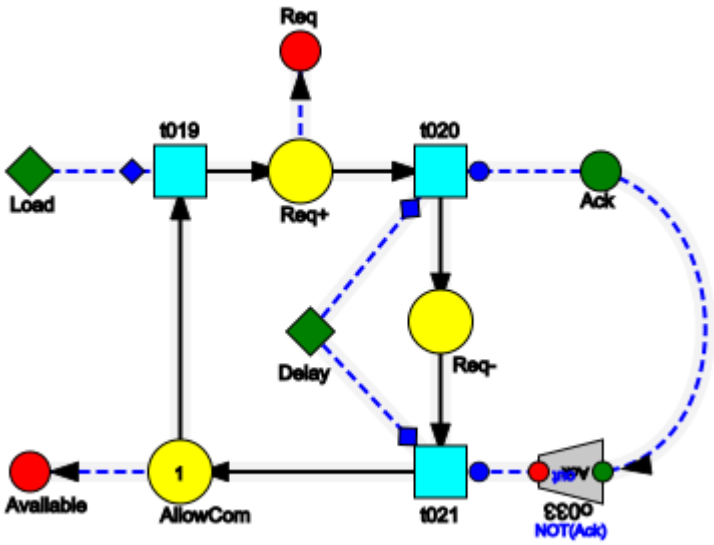

Figure 3.2 - Simple 4-Phase Transmitter.

Figure 3.2 depicts the Simple 4-phase transmitter, in its initial state. This state is held until the next load event, this load event is the means by which the synchronous block informs the transmitter that a new event should be transmitted. The component's outputs are the Available signal and the Req (for request) signal, both are connected to Petri net places from which their values are dependent of, this is, their value is high whenever the Petri net place they are connected to holds at least one mark and low otherwise. Once the load event occurs, the transition t019 will fire consuming the mark in AllowCom (Available signal↓) and creating a new mark on Req+ (Req signal↑). Once the Ack (for acknowledge) signal is high the transition t020 fires, causing the mark in Req+ (Req signal↓) to be consumed and a new mark on Req- to be created. Once Ack signal is low the transition t021 will fire consuming the mark on Req- and creating a new mark on

AllowCom (Available signal ↑) and by doing so the component returns to its initial state.

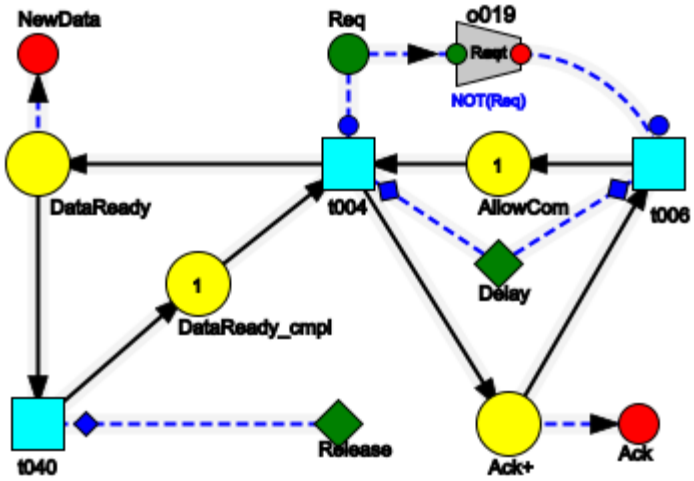

Figure 3.3 - Simple 4-Phase Receiver.

In figure 3.3, the Simple 4-phase receiver, in its initial state, is visible. This state is held until the Req signal is high, meaning that the transmitter desires to communicate a new event. In this component a place named DataReady_cmpl exists, this place is complementary to DataReady, as its name implies, its sole purpose is to impede DataReady of having more than one mark at any giving time, as this scenario would disrupt the system. In a similar fashion, the upcoming components will also have complementary places, these will not be further described as their purpose is directly related to the place they are complementary to. Once again, the component's outputs are connected to Petri net places from which their values are dependent of, them being the NewData output signal and the Ack output signal. Just as the receiver perceives that the Req signal has been set to high the transition t004 will fire, as a consequence, the mark in AllowCom is consumed and marks are created in DataReady (NewData↑) and Ack+ (Ack↑). Since NewData is now high the receiver's synchronous block has been notified that a new event has been transmitted, once the synchronous block is ready to take in information, the Release event should be evoked in order to have the transition t040 fire, in consequence, consuming DataReady (NewData↓) and allowing for new information to be received. Meanwhile, once Req is set to low the transition t006 will fire, consuming the mark on Ack+ (Ack↓) and creating a mark on AllowCom. Once Req is low and DataReady is empty the system is again in its initial state.

Figures 3.4 and 3.5 depict, respectively, the Simple 2-phase transmitter and receiver, in their initial state. As it would be expected, they function in a similar manner to their 4-phase protocol counterparts. Many of the elements are conserved, in particular the same inputs and outputs are still present and connected to the same Petri net places. Nonetheless, one main difference is that these components communicate by detecting both the rising and falling edges instead of simply reading the signals. The differences between these two protocols will be further explained, starting with the transmitter.

Whenever a load event occurs, assuming the transmitter is available, either the transition t021 (Req↓) or the transition t019 (Req↑) will fire consuming AllowCom (Available↓), whichever transition fires is determined by whether a mark is present on either Req+ or Req-, a mark will always be consumed in one of these Petri net places and created on the other, effectively controlling the Req output signal. At any given time, a single mark will exist in either of these places. The other main difference is how the response from the receiver impacts the system, namely, once a shift in the Ack signal is detected a mark will be created in AllowCom (Available↑), this shift is detected by the operation that is connected to the Ack signal.

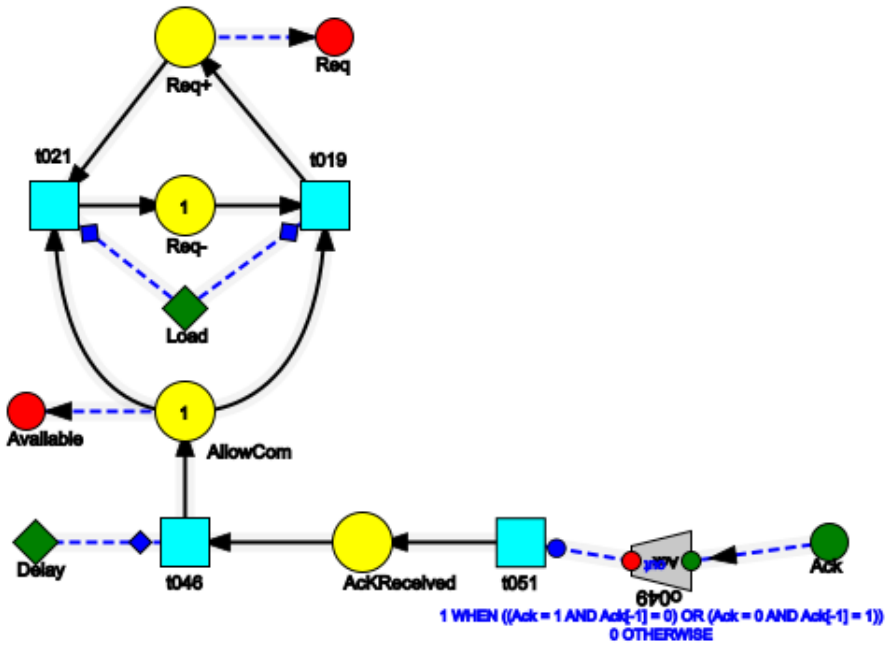

Figure 3.4 - Simple 2-Phase Transmitter.

As for the receiver, it is readily visible that the left section of the component remains untouched. Once again, divergences are mainly laid out on the sections of the component responsible for communication, as it would be expected. Likewise, the transmitter's Req+ and Req-, the transitions t029 and t028 are responsible for consuming the mark on either Ack+ (Ack↓) or Ack- (Ack↑) and creating a new mark on the other, once more, effectively controlling the Ack output signal. A shift in the Req signal will indicate that a new event is being transmitted, and as long as DataReady is empty a new cycle may begin, else the system will wait for DataReady to become empty.

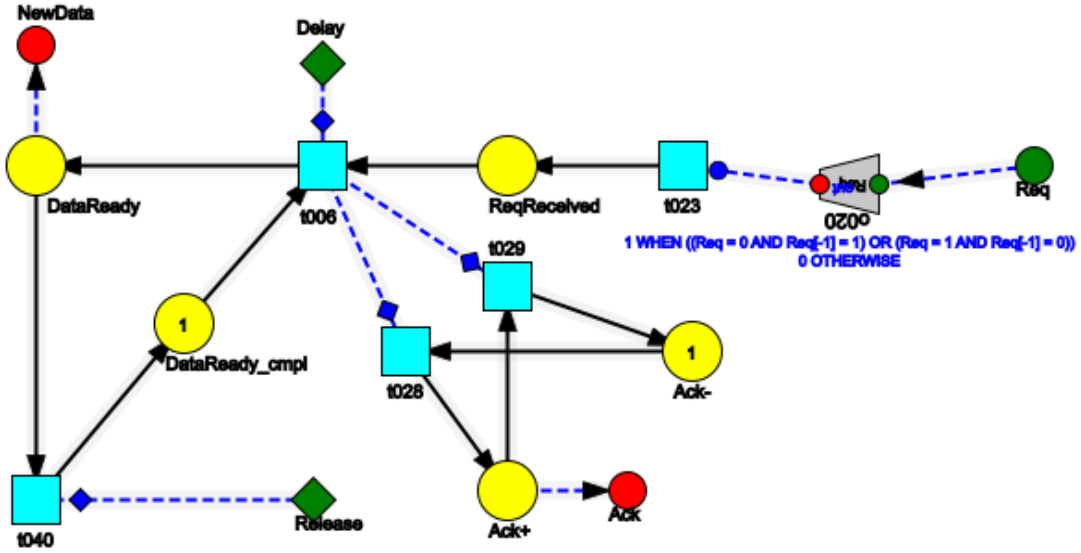

Figure 3.5 - Simple 2-Phase Receiver

Figures 3.6 and 3.7 depict a Simple communication example for each protocol, the former for the 4-phase protocol and the latter for the 2-phase protocol. As was mentioned before, the components have the same inputs and outputs regardless of which communication protocol was employed.

As it is visible on each figure, the transmitter's inputs are a load event (this event should be the result of a Petri net transition being fired as shown in the figure), a delay event and an Ack signal. On the other hand, the transmitter's outputs are an Available signal and a Req signal. While the transmitter is available for communication the Available output signal should be high and otherwise low. The transition connected to the load input event should only fire if the Available output signal is high, whenever new events need to be transmitted. The Req and Ack signals are the Request and Acknowledge signals used for communication between blocks.

The receiver's inputs are a release event (similar to the load event it should be connected to a Petri net transition), a delay event and a Req signal. The receiver's outputs are a NewData signal and an Ack signal. The NewData output signal is high whenever an event has been received from the other block and low otherwise. After the new information has been retrieved by the receiver's block, the Petri net transition connected to the Release event should fire which will result in NewData going from high to low, however, this transition should fire only if NewData is high.

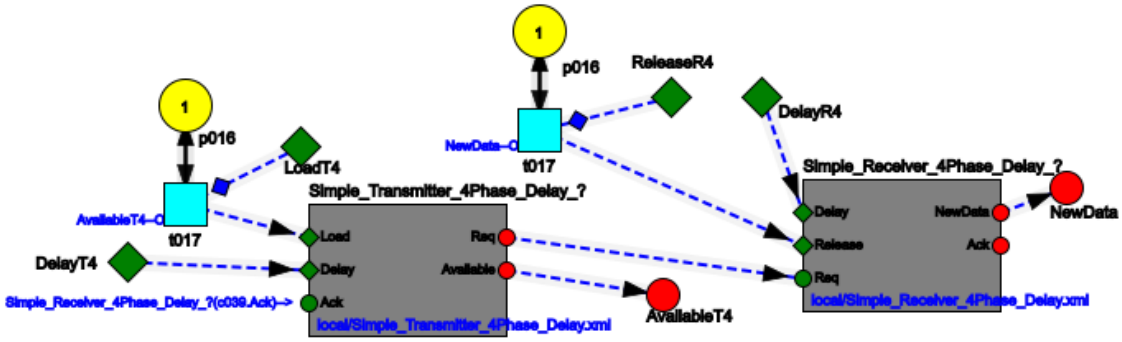

Figure 3.6 - Communication between Simple 4-phase transmitter and receiver.

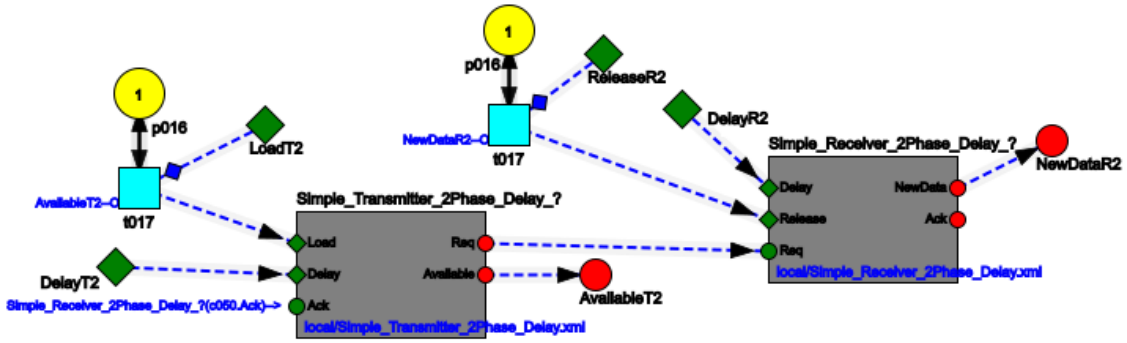

Figure 3.7 - Communication between Simple 2-phase transmitter and receiver.

3.2.2 - SimpleBuffer Interface

The SimpleBuffer interface is akin the Simple interface, as one might infer from their names, except for the presence of a buffer. This buffer is capable of holding a pre-defined number of events before becoming unavailable in contrast to the single one from the Simple interface.

Figure 3.8 illustrates an example of how this interface might be applied, as was earlier explained, AB1 is the synchronous block to which the transmitter is

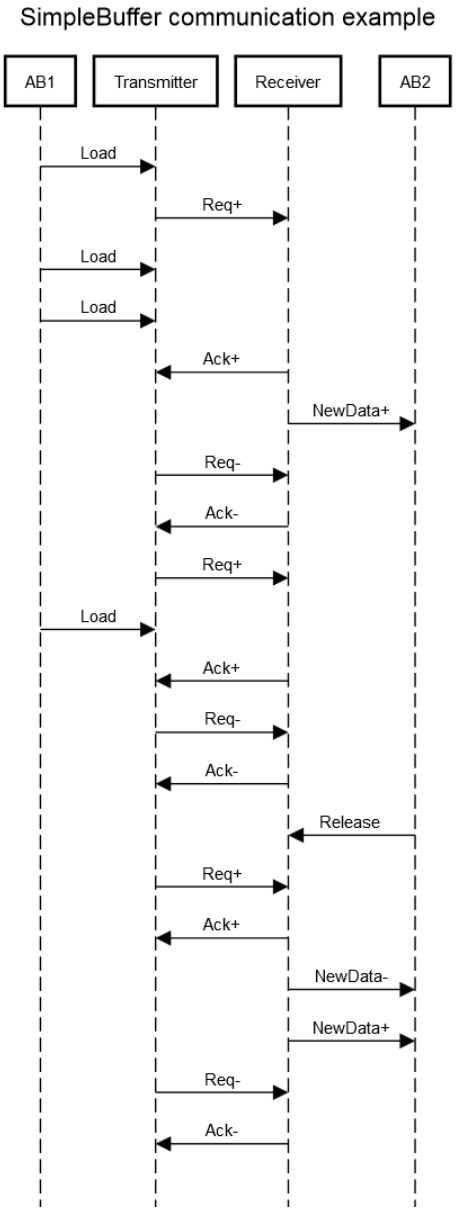

Figure 3.8 - SimpleBuffer 4-phase communication example.

connected to and, in a similar manner, AB2 is the synchronous block to which the receiver is connected to. Three events are loaded in quick succession before obtaining a response from the receiver, once the first load event occurs the Req is set to high to establish communication. Then, the receiver responds raising the Ack signal, in the meanwhile, NewData also raises indicating to AB2 that a new event has been received. Communication proceeds as expected, later, another event is loaded without disrupting said communication. By the point the receiver holds two events, a Release event occurs from AB2, NewData is lowered and then raised again, since it had more events to transmit.

Figure 3.9 portrays the SimpleBuffer 4-phase transmitter, in its initial state. As it would be expected it is very similar to the corresponding Simple transmitter. Only a few elements were added, for instance, whenever a Load event occurs, if available, the transition t004 will fire creating a mark on the Buffer place, this Buffer holds the amount of events yet to be communicated. The operation o005 is high while the buffer is not yet full, it does so by comparing the BufferSize signal input with the buffer. The Available signal is directly connected to this operation. The BufferEV output signal exists mainly as a debugging method and may be later removed.

Figure 3.10 depicts the SimpleBuffer 4-phase receiver, in its initial state. Once more, it is very similar to the corresponding Simple receiver. Only a few elements were added, specifically, between the transition t004 and DataReady. A Petri net place, Buffer, holds the amount of events that have been communicated

but are yet to be read. Furthermore, an additional condition is imposed onto the t004 transition, gating off new information if the buffer is full.

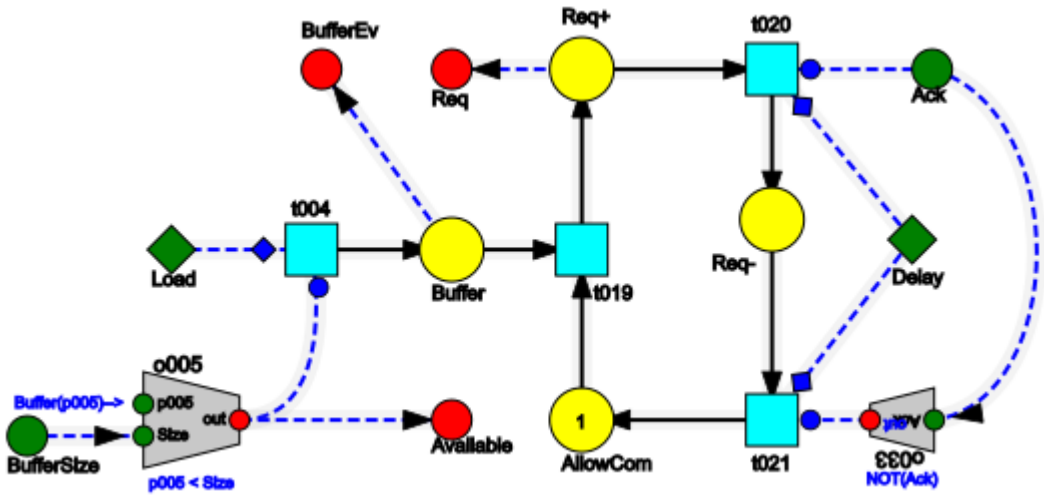

Figure 3.8 - SimpleBuffer 4-phase transmitter.

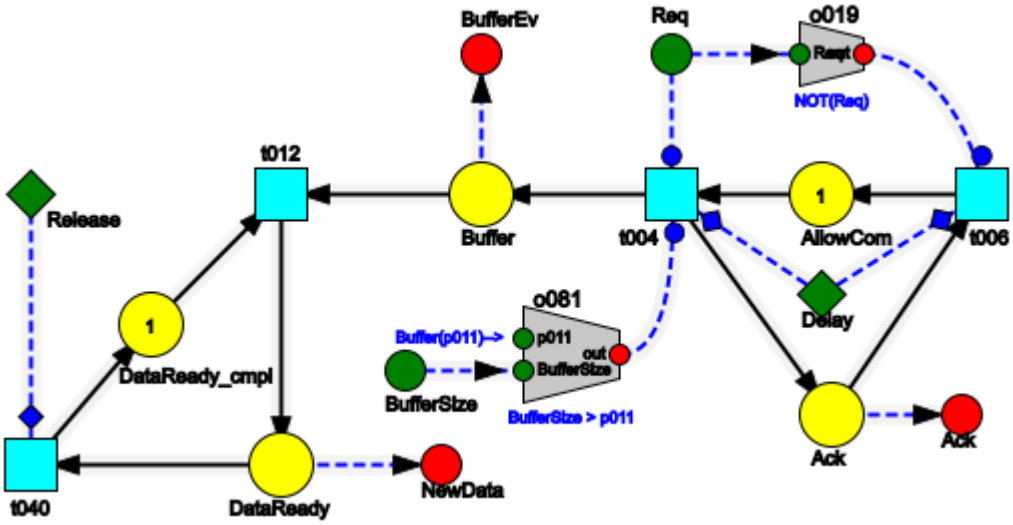

Figure 3.9 - SimpleBuffer 4-phase receiver.

Figures 3.11 and 3.12 depict, respectively, the SimpleBuffer 2-phase transmitter and receiver. Once more, these two components are similar to both the Simple 2-phase components and to the SimpleBuffer 4-phase components.

Figures 3.13 and 3.14 illustrates a simple communication example of the SimpleBuffer components, for both the 4-phase and the 2-phase protocols. What is shown is very similar to figures 3.6 and 3.7, with the only additions being the

BufferSize input signal and the BufferEV output signal. The BufferSize signal input dictates the size of the buffer up to a maximum of 255 (albeit this value could be easily increased with few adjustments), this value should be set at the start, as to permit events to be transmitted. However, this value can be altered knowing that increasing it simply allows for more events onto the buffer and decreasing it for less events, although decreasing this value does not in and of itself decrease the amount of events to be sent/read on the buffer, causing it to possibly have more events on the buffer than the buffer size.

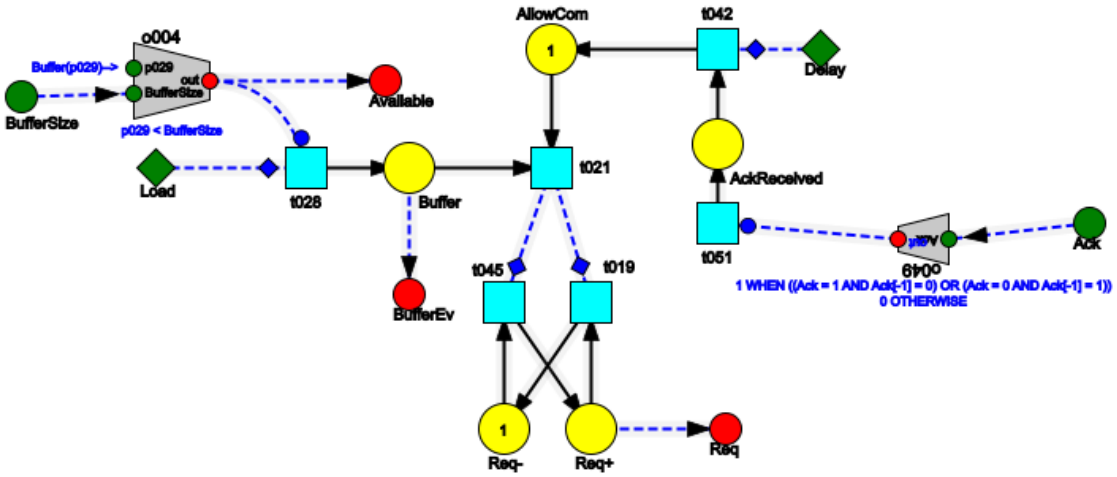

Figure 3.10 - SimpleBuffer 2-phase transmitter.

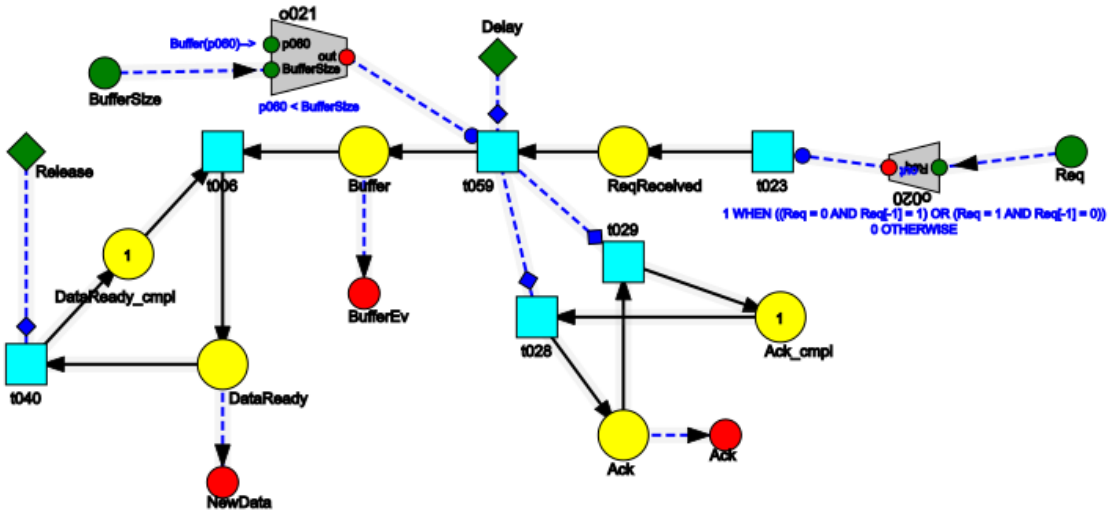

Figure 3.11 - SimpleBuffer 2-phase receiver.

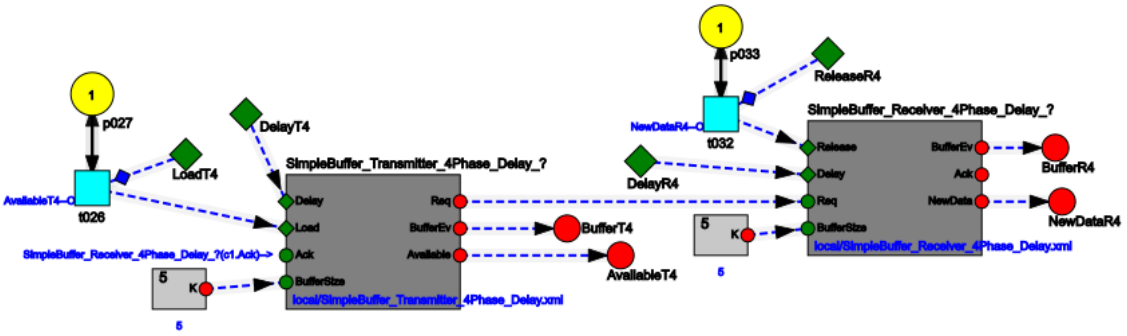

Figure 3.12 - Communication between SimpleBuffer 4-phase transmitter and receiver.

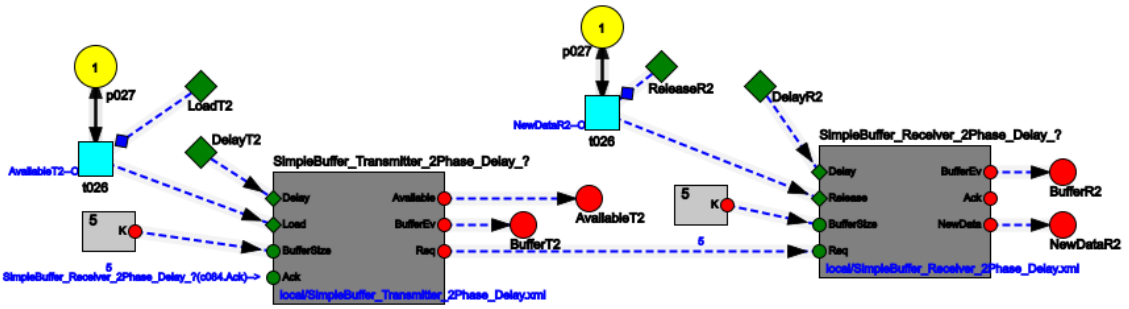

Figure 3.13 - Communication between SimpleBuffer 2-phase transmitter and receiver.

3.2.3 - BurstBuffer Interface

The BurstBuffer interface continues to build upon the previous interfaces. It transmits events and has an adjustable buffer, however, it differs in its ability to transmit multiple events at once. Nonetheless, despite having this ability, events are loaded onto the transmitter one at a time and retrieved from the receiver, likewise, one at a time.

Figure 3.15 illustrates an example of how this interface might be applied. Four events are loaded in quick succession, yet, once the first event is loaded, since communication is available, is it immediately communicated, and only upon its conclusions are the following events communicated. OutData, the data signal that connects the transmitter and the receiver, is set to 1, since one event is being transmitted, then, Req is set to high. However, since a late channel is being employed, the data signal is only read after the lowered Req is received. Once the first communication cycle has finished, OutData is set to 3 and Req is set to high beginning a whole new cycle.

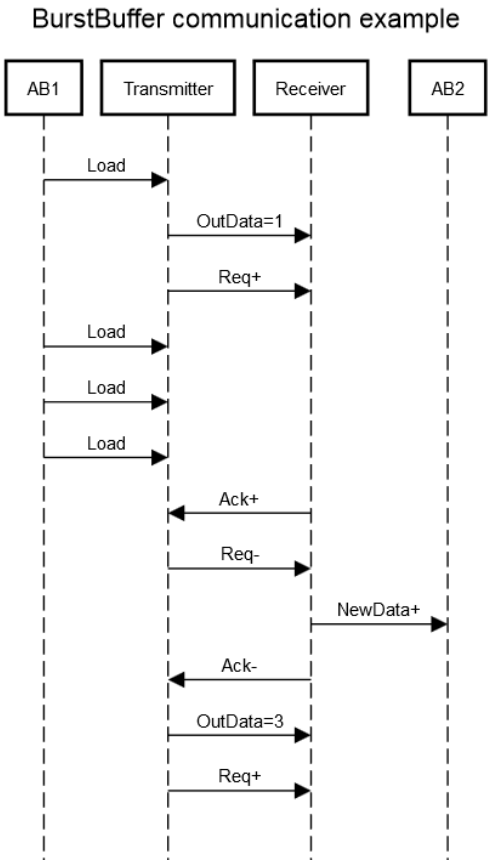

Figure 3.15 - BurstBuffer 4-phase communication example.

Figure 3.16 exhibits the BurstBuffer 4-phase transmitter, in its initial state. The communication section and the load mechanism are retained from the previous interfaces. Modifications lie on the buffer and how information is transmitted. Operation o038 increments whenever a load event occurs and resets whenever the transition t019 fires, which signifies the start of communication. The internal signal IS1 reflects the output from o038. The transition t019 fires whenever there is both a mark to consume in AllowCom and there are events to transmit. Operation o019 verifies if there are events in the Buffer and if so enables t019. Operation o039 holds the Data to be sent, if AllowCom has a mark then it becomes transparent and its output is the value on IS1, otherwise, it holds its output, this is important because data should be held during communication and loaded once a new communication starts.

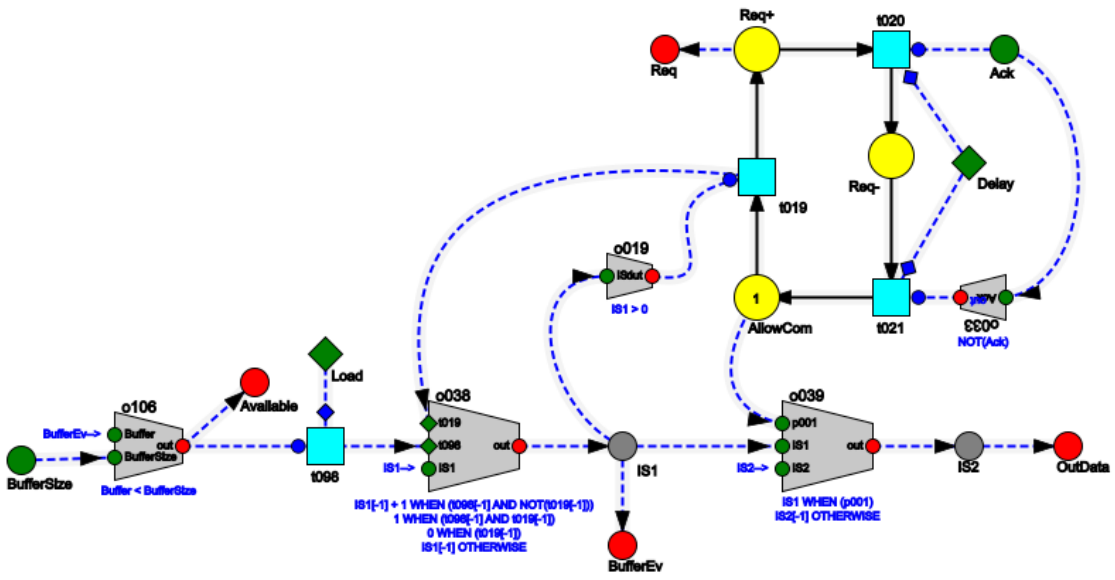

Figure 3.14 - BurstBuffer 4-phase transmitter.

Figure 3.17 displays the BurstBuffer 4-phase receiver, in its initial state. While at first it might appear disjointed and peculiar when compared to the previous interfaces, yet, many of the main functionalities are still present. The communication section and the section that outputs events to the synchronous block remain untouched apart from the means by which they are triggered. Operation o007 holds its output, aside from, whenever o019 rises (Req↓), were InData is added to the output value held, additionally, whenever t012 fires the output value is decremented by one. The transition t012 will fire if DataReady is empty and there are events stored in the buffer. The transition t004 will fire if there is a mark on AllowCom to be consumed, Req is high and it's enabled by the operation o029. The operation o029 is high only if InData plus the Buffer is not higher than BufferSize.

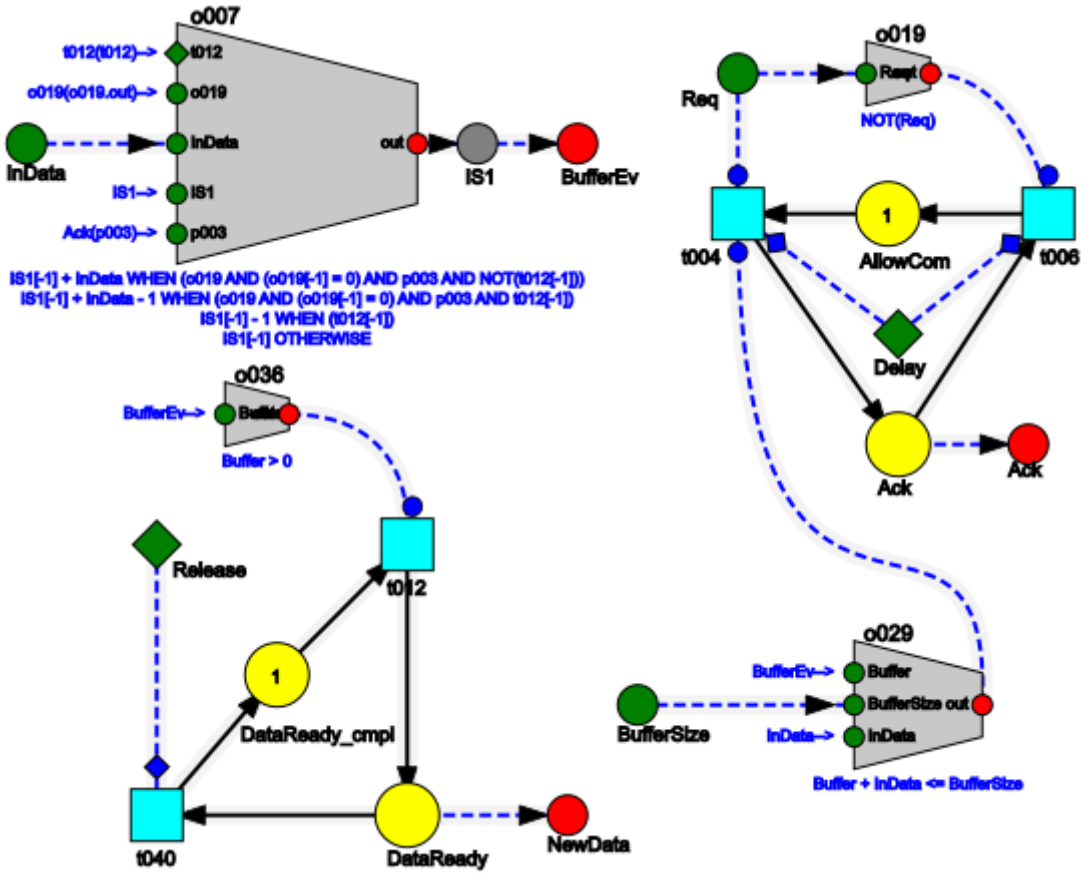

Figure 3.15 - BurstBuffer 4-phase receiver.

Figures 3.18 and 3.19 display, respectively, the BurstBuffer 2-phase transmitter and receiver. These components are a mixture of both their 4-phase counterparts and the 2-phase components from previous interfaces, the exception being the addition of the operation o001 in the receiver that introduces a delay of one cycle so that data is read before the acknowledge shifts value.

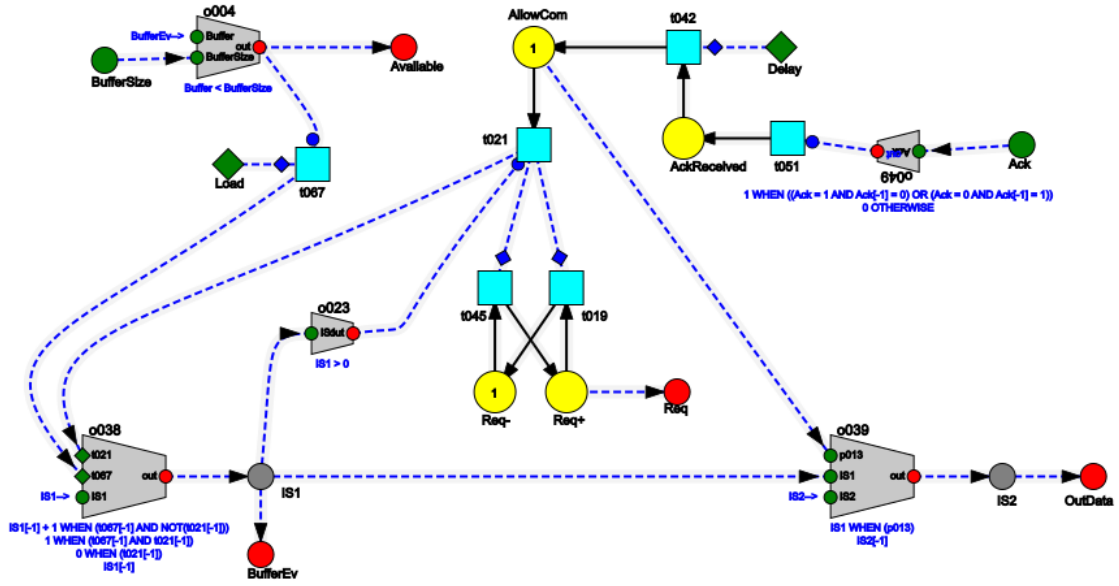

Figure 3.16 - Burstbuffer 2-phase transmitter.

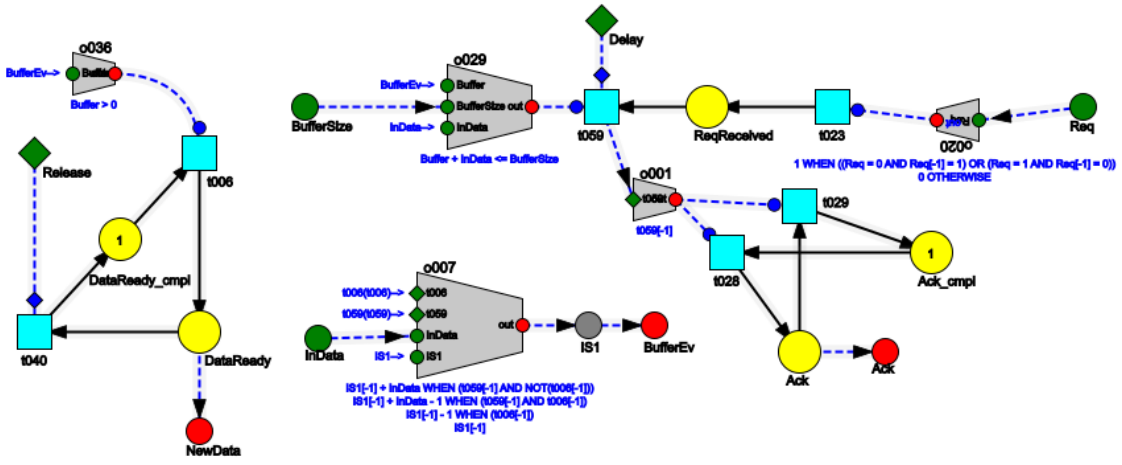

Figure 3.17 - BurstBuffer 2-phase receiver.

Figure 3.20 depicts a simple communication example between the BurstBuffer components. To avoid redundancy, since both components are undistinguishable once encapsulated, only the 4-phase protocol is shown. It is very similar to the previous interface, were it differs is in its ability to transmit data, and so, for this reason, the transmitter has an additional output, OutData, and the receiver an additional input, InData.

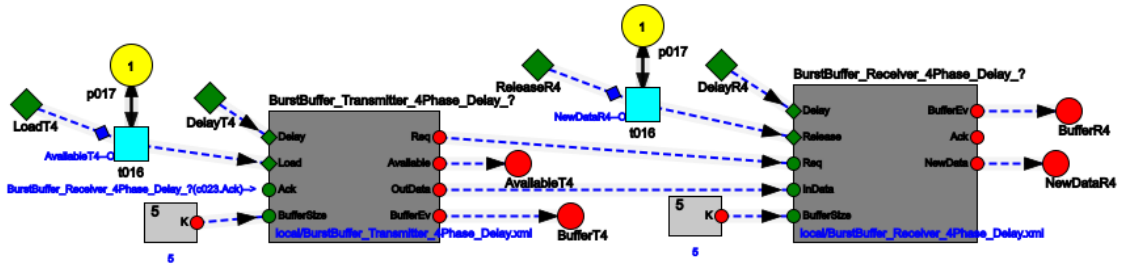

Figure 3.18 – Communication between BurstBuffer 4-phase transmitter and receiver.

3.2.4 - Data Interface

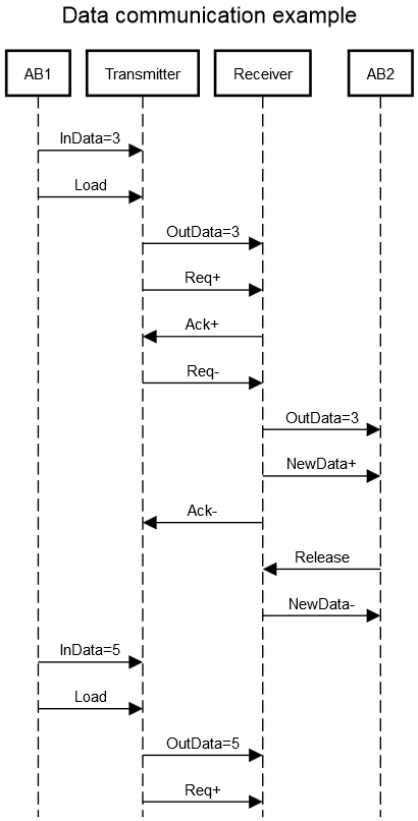

Figure 3.21 - Data 4-phase communication example.

The Data interface is most alike the Simple interface, apart from the fact that it transmits data instead of events. For each communication cycle a data signal is added in order to transmit information. A late approach with a push channel is used, meaning data is valid between the moment the Request signal is set to low until the moment the transmitter receives the lowered Acknowledge signal.

In Figure 3.21, an example of communication using the Data interface is presented. In it, two values are transmitted, first 3 and then 5. A complete cycle is visible for the first value and only the start is visible for the second value.

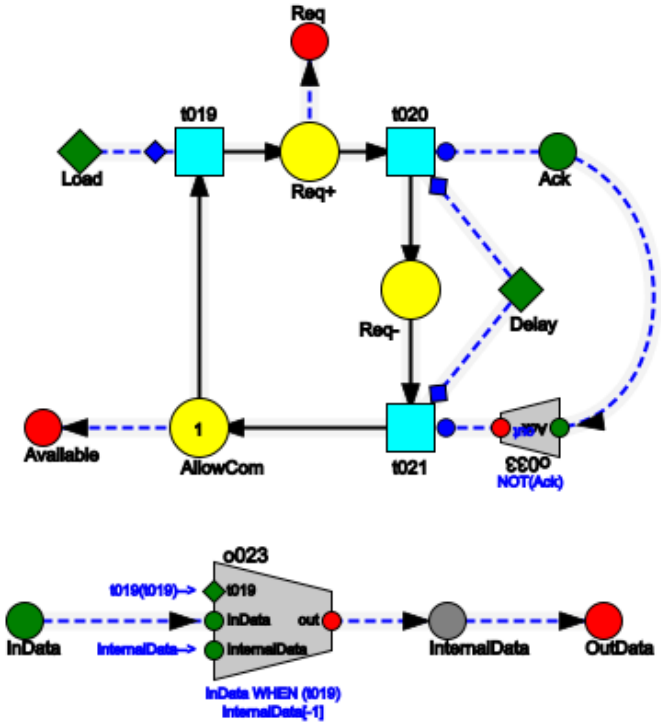

Figure 3.19 – Data 4-phase transmitter.

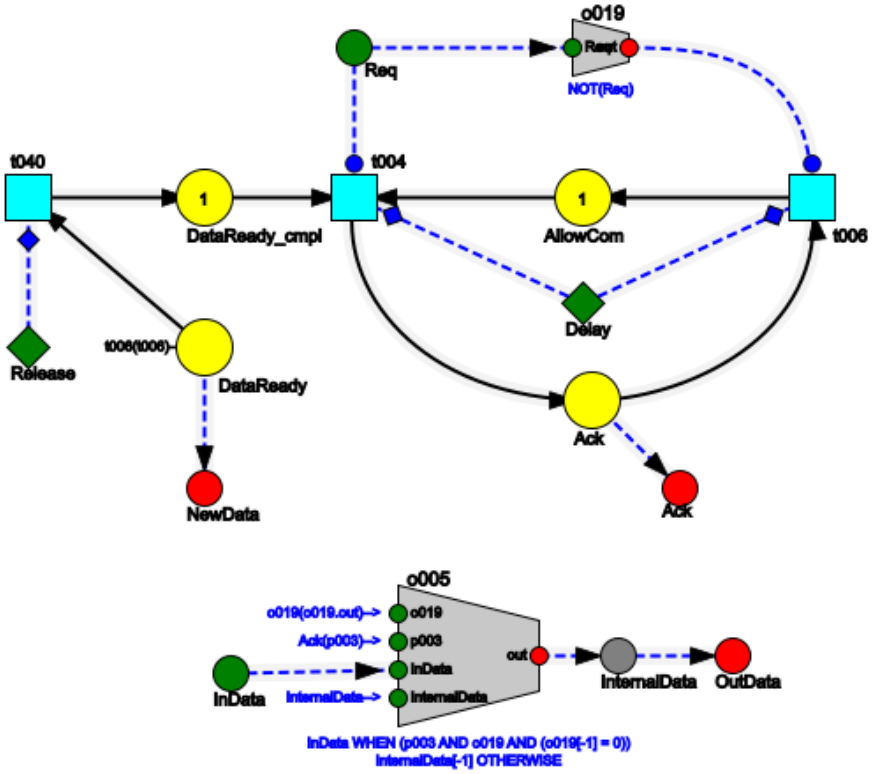

Figure 3.20 - Data 4-phase receiver.

All of the Data interface components are very similar to those of the Simple interface. In figures 3.22 and 3.23, the Data 4-phase transmitter and receiver are, respectively, displayed in their initial state. The only difference between the transmitter and that of the Simple interface is the addition of the operation o023 and its connected signals, however, this operation was also added to the receiver with some minor differences. This operation becomes transparent, meaning its output is equal to its input, InData, whenever the transition t019 fires, for the transmitter, or the operation o019 rises (Req↓), for the receiver. For the transmitter, OutData is the output signal through which data is communicated, it's for that reason that it's important that the operation holds its value during communication and change it only upon a new communication cycle. For the receiver, OutData is the output signal from which the synchronous block retrieves data, likewise, it is necessary that this value is held until it is released. Additionally, the receiver's DataReady output signal will only rise once the Req signal is lowered contrary to how it is in the Simple interface which would rise with t004 firing.

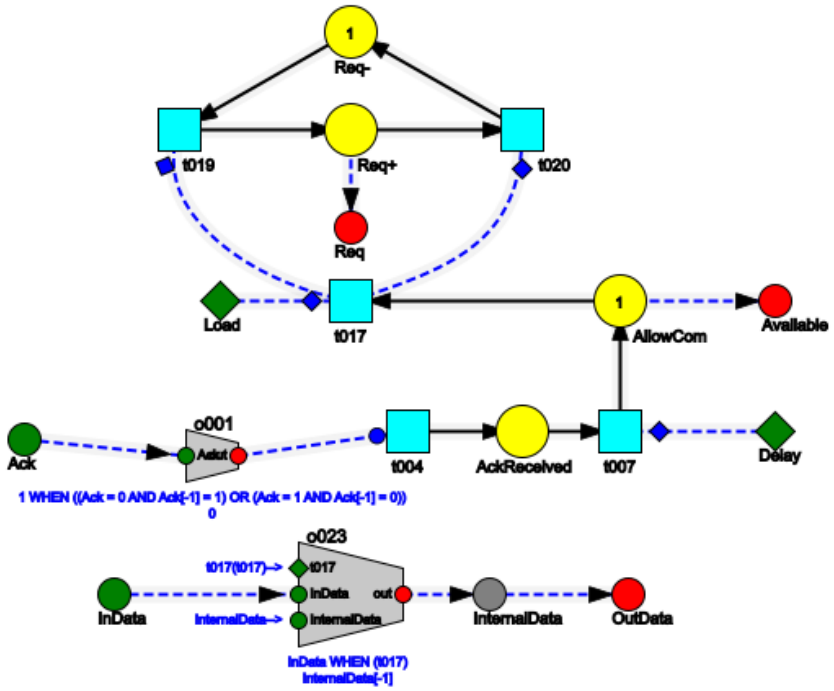

Figure 3.21 - Data 2-phase transmitter.

Similarly, figures 3.24 and 3.25 depict, respectively, the Data 2-phase transmitter and receiver in their initial state. Just as the 4-phase components, these components also function in the same manner as the correspondent Simple components with the introduction of the operation responsible for holding data values, and its related signals.

Lastly, Figure 3.26 displays a simple communication example between the Data 4-phase components. The single alteration from the Simple communication interface is the addition of the data signals in and out of each component.

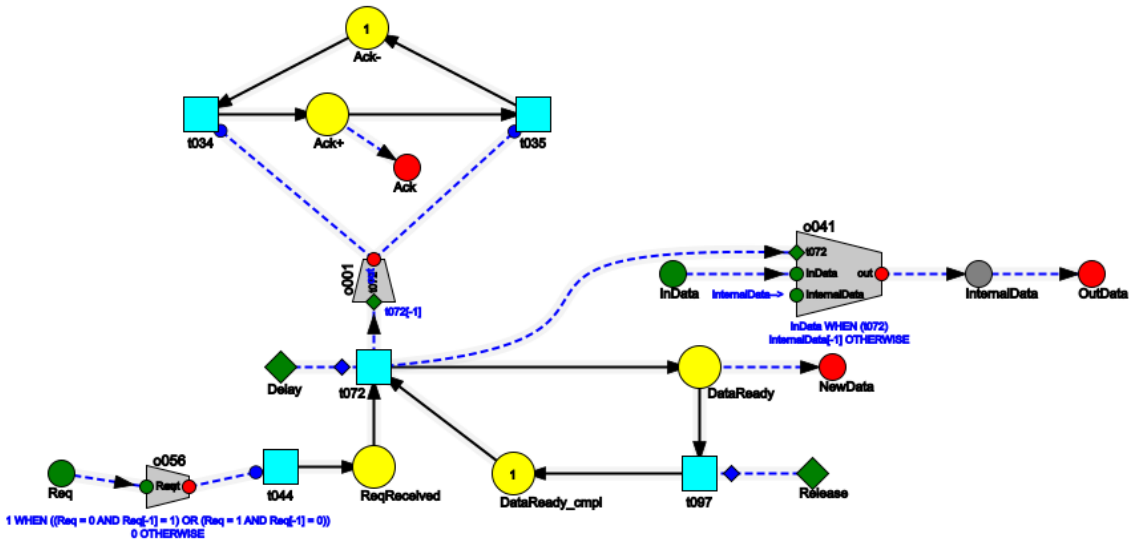

Figure 3.22 - Data 2-phase receiver.

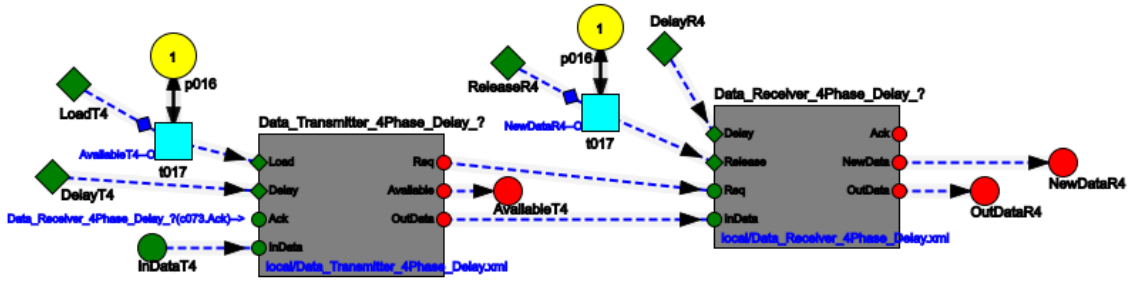

Figure 3.23 - Communication between Data transmitters and receivers.

3.2.5 - DataBuffer Interface

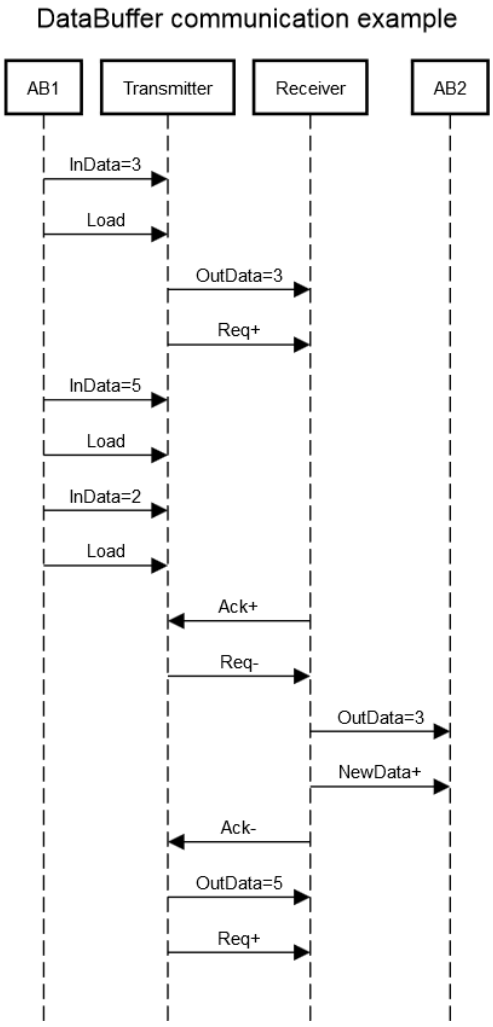

Figure 3.27 - DataBuffer 4-phase communication example.

The last interface is the DataBuffer interface. Resembling the Data interface as it transmits data between synchronous blocks, moreover, it is capable of holding more than one value. The current components are ready for three values. However, this quantity could be increased with the addition of more operations, similar to those already implemented. These values are transmitted/retrieved in the same sequence they are loaded/received.

Figure 3.27 is a simple example of communication between two synchronous blocks using the DataBuffer interface. Three values are quickly loaded onto the transmitter before the Receiver is able to respond. While not visible in the diagram, the transmitter would've been full after the third value was loaded, resulting in the available signal being set to low. Then communication proceeds as usual and once a new cycle begins the second value is sent.

The DataBuffer components are the most distinct, this is due to their ability to hold multiple data values, for each data value an operation is required, the register, and, regardless of how many data values there are, two additional pointers and a single multiplexer.

The register has three inputs, Indata, the incoming data, the write pointer output, and lastly, its own output. Whenever the output from the write pointer currently corresponds to the register's number but it did not in the previous cycle, the register's output will mirror InData, otherwise its output value will be held.

The write pointer has four inputs, the load event, the available signal, a constant that represents the amount of registers and its own output. The write

pointer will increment its output by one every time the load event fires, unless the available signal is low, in which case it will hold its output, or its output matches the amount of registers in which case it will cycle back to one. If the load event does not occur the current output is held. For the receiver, the load event is replaced by both the Petri net place named Ack and the operation o019, with the goal of triggering simultaneously with the transition t006, as a result the write pointer on the receiver will have five inputs instead of four, albeit its behaviour is identical.

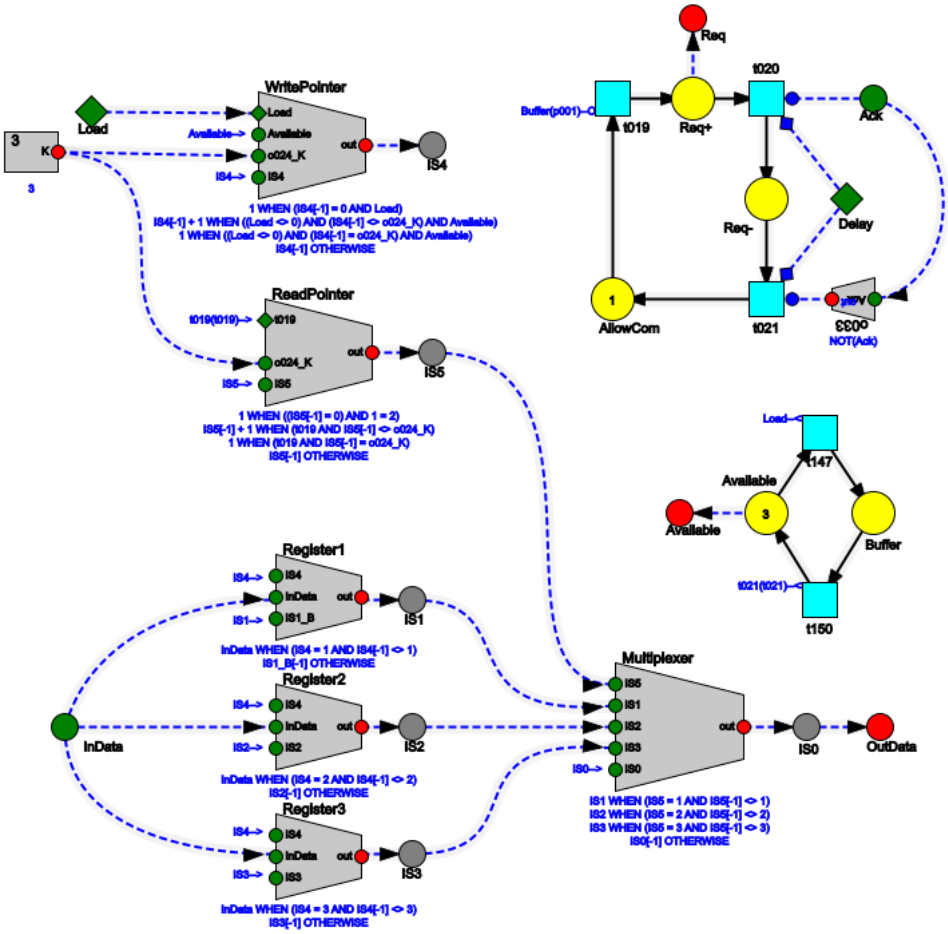

Figure 3.24 - DataBuffer 4-Phase Transmitter.

The read pointer is very similar to the write pointer, the only differences are the absence of the available input and instead of the load event, the input that increments the output is the start of communication, t019 being fired, for the transmitter. For the receiver, the load event is replaced by the transition t067 in the data retrieval mechanism.

The multiplexer has as its inputs each of the register's outputs, the read pointer's output and its own output. It will hold its own output, unless, the read pointer alters its value, in which case, it will mirror the value of the register whose number corresponds to the pointer's output.

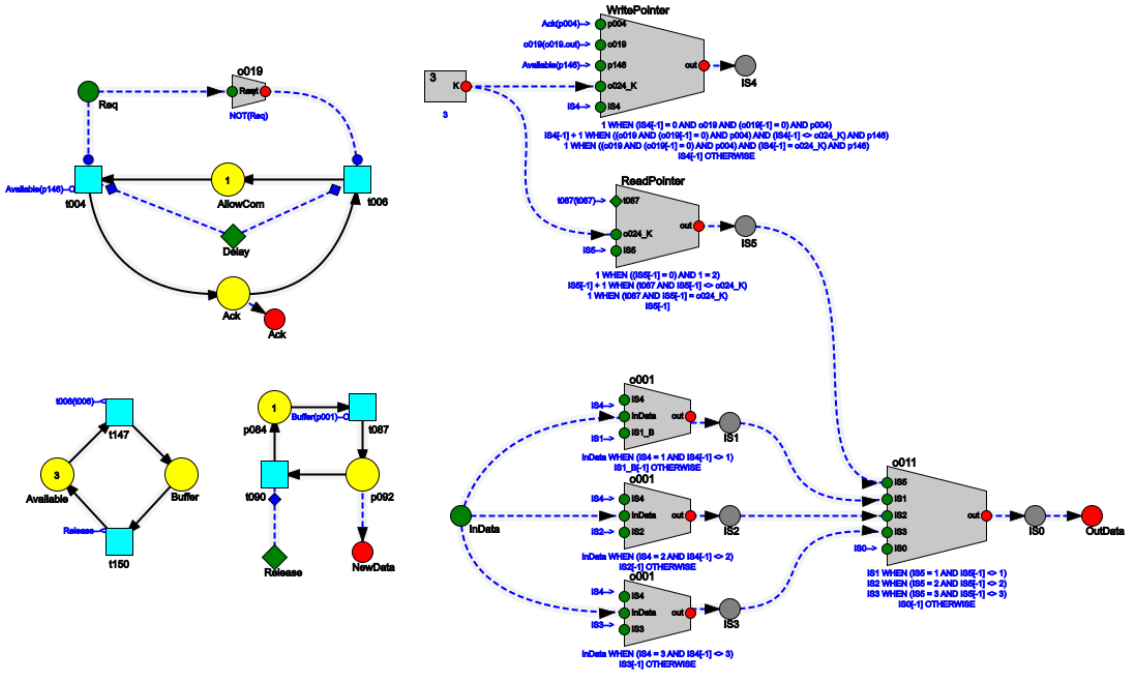

Figure 3.25 - DataBuffer4-Phase Receiver.

Figures 3.28 and 3.29 depict respectively the DataBuffer transmitter and receiver. Both have three registers, each capable of holding a data value, a write pointer, that determines in which register incoming data should be kept, a read pointer, that determines from which register outgoing data should be read, data to be transmitted in case of the transmitter and retrieved by the synchronous block in case of the receiver, and lastly a multiplexer that selects the register accordingly to the read pointer. Additionally, both components have a simple mechanism composed of two Petri net transitions (t147 and t150) and two Petri net Places (Available and Buffer) that keep count of the number of data values currently being held. The remaining mechanisms of the component are akin those of previous interfaces, namely the communication segments and, in case of the receiver, the data retrieval segment.

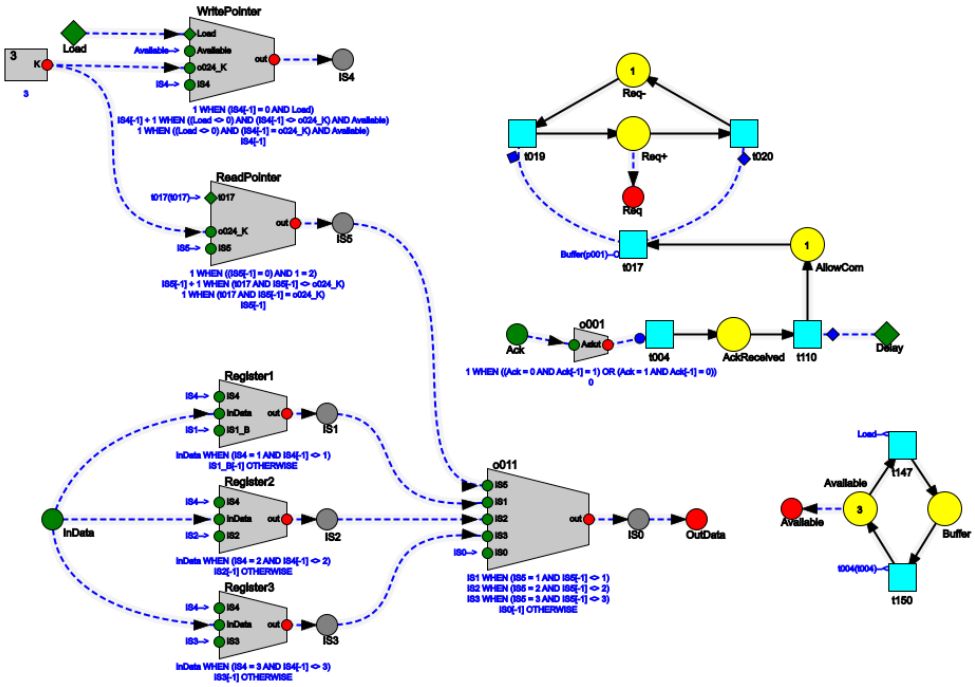

Figure 3.26 - DataBuffer 2-Phase Transmitter.

Similarly, figures 3.30 and 3.31 depict, respectively, the DataBuffer 2-phase transmitter and receiver in their initial state. These are very similar to their 4-phase counterparts, possessing the same elements described above.

Lastly, figure 3.32 presents the DataBuffer 4-phase communication example between its respective components. Its appearance is identical to the Data communication example as the single difference is the components themselves.

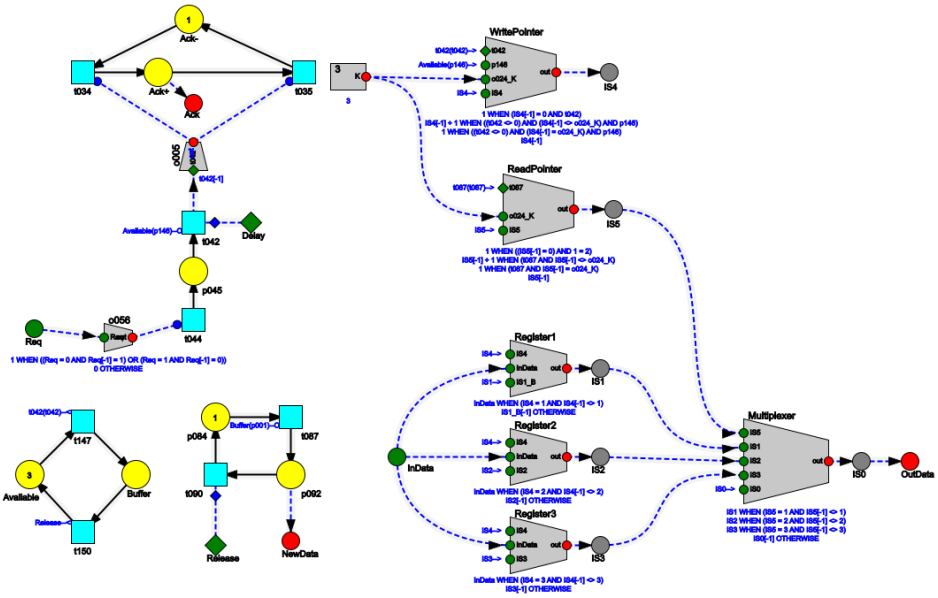

Figure 3.27 - DataBuffer 2-Phase Receiver

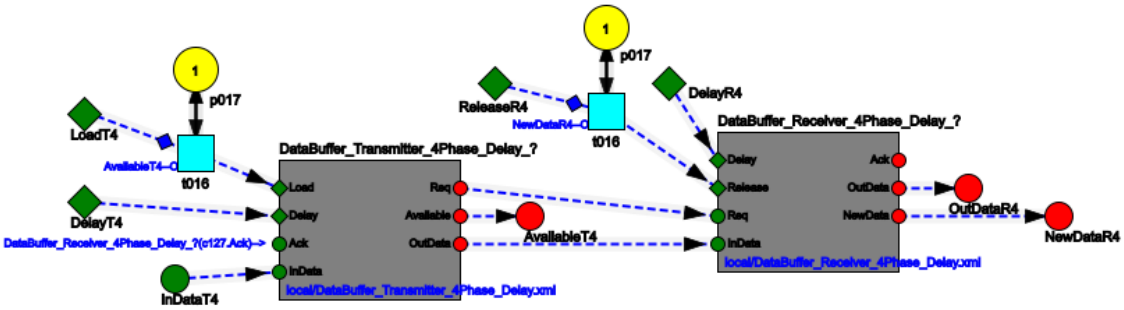

Figure 3.28 - DataBuffer 4-Phase Communication.

4

Chapter Four - Extending DS-Pnets and IOPT-Flow for GALS

The DS-Pnet modelling formalism and the IOPT-Flow framework are first and foremost a set of tools that aims to accelerate the development of synchronous cyber-physical systems, and so, in their current state, some features are not prepared for a GALS system, such as the execution semantics, which are globally synchronous, and the generation of hardware code with a single clock signal.

There is a necessity to model a system composed of several synchronous blocks capable of interacting asynchronously with one another, then this model needs to be simulated within the framework and the generated code should allow for each synchronous block to have its own clock signal.

4.1 - GALS-DS-Pnet Definition

A GALS-DS-Pnet formalism emerges with the need to extend the DS-Pnet formalism described in [9] and [10]. A GALS-DS-Pnet is always a top-level model that encompasses components that are described with by the DS-Pnet formalism.

A GALS-DS-Pnet model contains components, signals, events, read-arcs, synchronizing signals and signal edges, and is described as a tuple $\mathrm{GALS-DS-Pnet} = (\mathrm{C}, \mathrm{S}, \mathrm{E}, \mathrm{R}, \mathrm{s}_0, \mathrm{ss}, \mathrm{se})$ satisfying the following requirements:

1. C is a finite set of components
2. S is a finite set of signals
3. E is a finite set of events

4. R is a finite set of read arcs with:
   $R \subseteq (C \times C) \cup (S \times C) \cup (E \times C) \cup (S \times S) \cup (C \times S) \cup (C \times E) \cup (E \times E)$
5. $\forall s \in S, \#\{(x \times s) | (x \times s) \in R\} \leq 1$ (signals may have at most one input arc)
6. $\forall e \in E, \#\{(x \times e) | (x \times e) \in R\} \leq 1$ (events may have at most one input arc)
7. s_0 is the initial signal values partial function with mapping $S \nrightarrow N_0$
8. ss is the signal that synchronizes a component where $ss: C \rightarrow S$
9. se is the edge to which the component is synchronized where $se: C \rightarrow e, \ e \in \{falling, rising\}$

In essence, a GALS-DS-Pnet model is a top-level model that contains components, which employ the DS-Pnet formalism with its execution semantics defined in [9] and [10], that are connected to other components and/or signals and events. The components are also connected to a synchronizing signal and a signal edge, these synchronize said components.

4.2 – Execution Semantics

Since multiple clocks are present, the execution step will inevitably be different. In this subchapter, first will be presented the current execution semantics (Figure 4.1), which is implemented in the IOPT-Flow simulator (in JavaScript) and in hardware (in VHDL), and finally, it is proposed an execution semantic that addresses a GALS system.

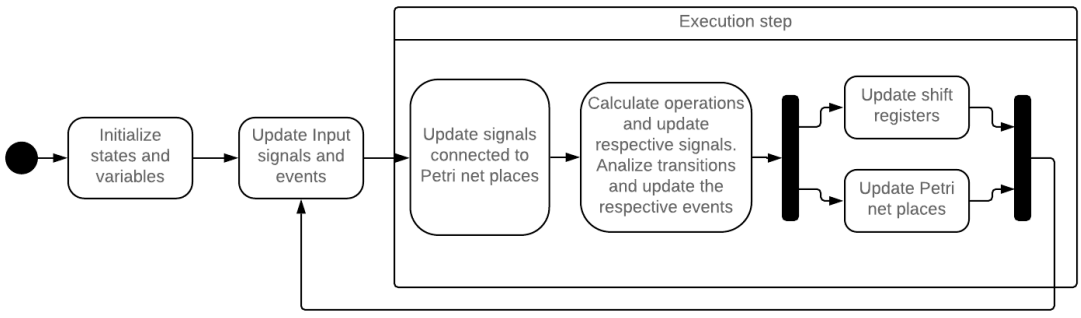

Figure 4.1 - Current execution semantics.

4.1.1 – JavaScript Implementation

The IOPT-Flow framework is capable of generating JavaScript code, this code is used in the framework's simulator. First, Petri net places and variables are initialized, according to their initial values. Then, signals (both internal and output) that are connected to Petri net places are updated. Next, accordingly to their sequence in the model, operations are calculated, and transitions are analysed, updating their connected signals and events. To clarify, it is often the case that operations and transitions depend on each other, all these need to be updated during the same execution step since they correspond to combinational logic rather than a state of the system, however, since they have dependencies they will be addressed in a determined sequence starting with the operations and/or transitions that do not depend on other operations and/or transitions. It is for this reason that cyclic behaviour is not allowed. Lastly, the step concludes with both shift registers and Petri net places being updated.

4.1.2 – Hardware Implementation

For a hardware implementation of the VHDL code generated by the IOPT-Flow framework, two moments may be identified, the first moment is the rising edge of the clock, during which every synchronized element will be updated, these include shift register, Petri net places and any synchronized signal or event. Then, the moment between two rising edges of the clock signal, combinational logic will take place, and these include calculating operations, analysing transitions and any unsynchronized signals.

Having the hardware's behaviour into account it is possible to view its execution step similarly to how the JavaScript code operates. To start, the input signals are updated accordingly to the currently available state, during the clock signal's rising edge. Then, all combinational logic takes place, calculating operations, analysing transitions and updating any unsynchronized signals. Lastly, during the following rising edge, shift registers and Petri net places are updated, accordingly to the calculations that take place during the combinational logic. It's of note that the next execution step starts simultaneously with the end of the current one. Updating synchronized signals is considered at the start of the step since its values are used during combinational logic, on the other hand, updating

shift registers and Petri net places are considered at the end of the step since they are altered by the combinational logic.

To clarify, signals that depend of Petri net places are only updated on the following, regardless of whether they are synchronized or not, this is true for both hardware and the JavaScript code since the places themselves are only updated at the end of a step. Additionally, if a signal does not depend of a place it is updated during combinational logic, if it is not synchronized, or during the following rising edge, if it is synchronized.

4.1.3 – Proposal for GALS

In order to adapt the current tools to accommodate a GALS system, few changes need to take place. Figure 4.2, through an UML state diagram, displays the proposed execution semantics for a GALS system in the DS-Pnet formalism. It is immediately evident that it uses the current execution semantics as a foundation, the only needed change is the possibility to have the multiple synchronous blocks independent from each other.

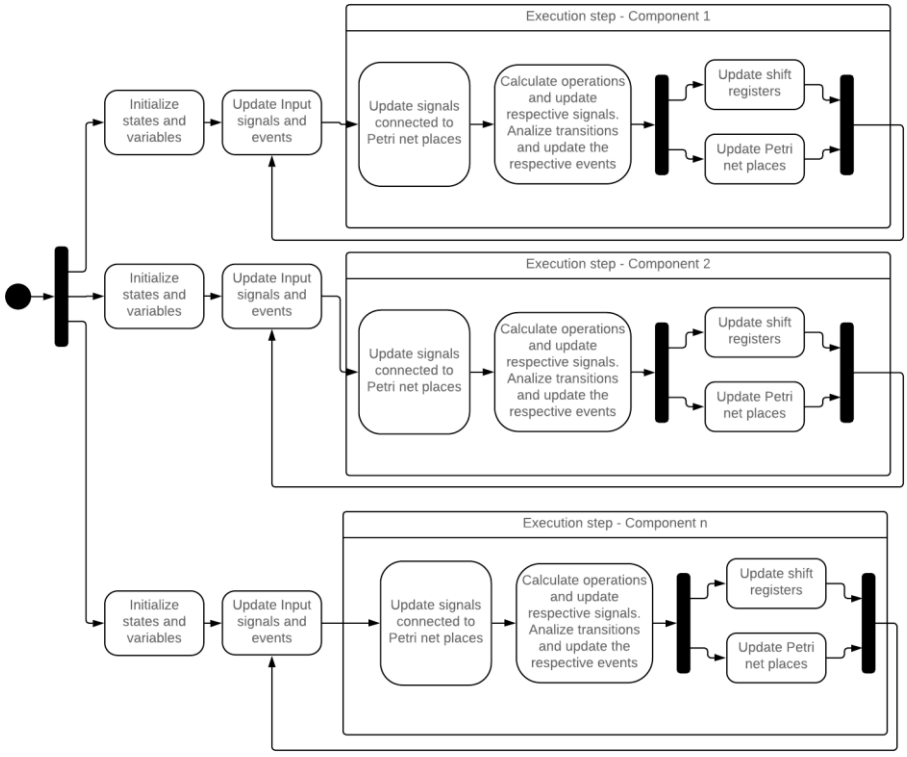

Figure 4.2 - Proposed execution semantics for GALS.

4.3 – IOPT-Flow Tools

In this subchapter, are proposed some adjustments to the IOPT-Flow framework's tools, namely, the editor, the simulator and the VHDL code generator.

4.3.1 - Editor

For future work, it would be advantageous if it was possible to assign multiple clock signals to different elements without much additional effort for the designer. As a suggestion, this can be achieved by selecting both a synchronous signal and a synchronizing edge within special tags that can be assigned to each component, thus, having components act as synchronous blocks.

Howbeit, as it was mentioned before the set of tools provided by the IOPT-Flow framework has the goal of accelerating the development of cyber-physical systems, and, for this reason, the possibility of designing GALS systems should not hinder the design of synchronous systems.

Accordingly, the most appropriate option would be to present the clock signal tag in the properties bar where it could be easily accessed but does not clutter the view for those that do not intend to use it. This could be implemented with either a dedicated drop box which would allow for the user to select the desired synchronous signal or add a new one, additionally, a different approach would be for the user to type a command in the comment section, or another akin section, in the properties window that would assign that element or component the desired clock signal, hiding the GALS feature in plain sight, so that someone that does not intend to use said function could easily avoid it. Another necessary extension would be the possibility to choose either the falling or the rising edge of the clock, since access to a specific edge might be required in order to interact with some components, one such example is the DDR sDRAM present in some FPGA boards, which employ interfaces that require a specific synchronizing edge.

4.3.2 - Simulator

The first issue that needs addressing is the lack of multiple synchronizing signals since, by definition, these are required for a GALS system. There are, however, some workarounds that involve creating events, these are regular input events that are attached to the system's elements, grouping said elements by the clock event they are attached to.

Each transition belonging to a certain synchronous block should be restrained by the same clock event. Additionally, each operation that has memory, this is, has a delay in one of its inputs, belonging to that same synchronous block is required to have a new input to accommodate this new clock event, and, its conditions should guarantee that its output should only change in case the clock event is high. However, some issues arise when the operation has a delay bigger than one. If the delay is one then the previous value is held so the new previous value is the same, yet, when the delay is bigger than one, the states are shifted, meaning that, the state in "–n" will be the state in "-(n+1)".

In order to circumvent this issue, multiple operations should be utilized, the amount of operations required is dictated by the size of the delay, since each operation will have a maximum memory of one. The implication is that for a "-n" delay "n" operations are required.

These are, nevertheless, band aid solutions that don't address the inherent problem and are, therefore, messy, which contradicts the desired goal for this set of tools. Additionally, these workarounds are mainly for testing on the IOPT-Flow simulator, and so, the resulting generated code will not be completely correct and will require some supplementary adjustments.

For the IOPT-Flow simulator, there should be an execution step function for each synchronizing signal. A cycle in the simulator should start with all the input values being read, as it is currently implemented, afterwards the execution step functions, whose clock signal is active, should be called. The order in which these functions are called are irrelevant since their inputs are read at the start and remain in unaltered until the next step.

Figure 4.3 depicts, through an UML state diagram, the aforementioned execution semantics for a JavaScript simulation.

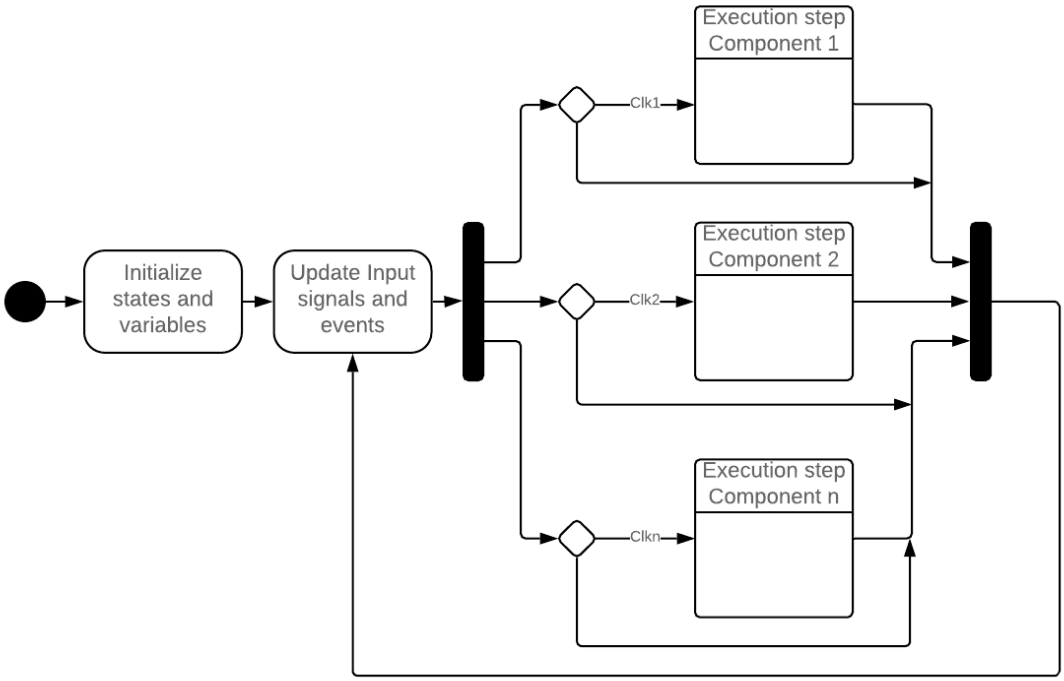

Figure 4.3 - Proposed execution semantics for JavaScript simulation.

4.3.3 - VHDL Code Generator

Upon generating VHDL code, each component should be synchronized to the signal and edge previously specified in the model. Additionally, each component should have its own execution process in the main file in order to assign signals.

Lastly, currently, once VHDL code is generated within the tool, both input and output signals are synchronized to the clock by default. This is a good measure to avoid metastability; however, it would be beneficial to have the option to not synchronize them. This option would allow for a designer to create a model within the IOPT-Flow framework and implement it into a larger synchronized model without having an unnecessary delay.

5

Chapter Five - Validation

5.1 - Asynchronous Components

In order to validate the components presented in chapter three, communication between each pair of components was simulated in both the IOPT-Flow simulator and in the Xilinx ISE Simulator (ISim).

First, the simulations were performed on the IOPT-Flow framework, since it offers real time response of the system during stimuli thus making it easier to obtain the desired order of events. Then, a new Xilinx project was created using the automatically generated VHDL code from the IOPT-Flow framework; however, by default the generated code synchronizes internal and output signals. While these synchronizations do not impact how the components behave, they introduce delays, which make comparing both simulations more difficult, and so, for each input and output signal, the synchronization was removed. Additionally, when creating a test bench some signals may be generated as "std_logic_vectors", even though they are "integer range" signals in the VHDL file under simulation. This means that these "std_logic_vector" signals should be changed to "integer range". Then, after creating the test bench, the input signals were introduced to match those of the IOPT-Flow simulation, lastly, the simulation on the ISim was performed. The results for each interface are shown below.

Both waveforms have roughly the same signals represented in the same order, though, on the IOPT-Flow framework an overwhelming amount of signals would be shown and so, to simplify, the image has been edited to only display relevant information. As for ISim, by default only the input and output signals

are displayed, this includes both clock and control signals, however, any internal signal may be added to the Wave Window, in this case both the acknowledge and request signals. These images have also been edited in order to take better advantage of the space, without any signals being added, removed or altered in any way. Furthermore, all signals are synchronized with the rising edge of the clock.

5.1.1 - Simple

5.1.1.1 - 4-Phase

For the first simulation, only the bare minimum stimuli were employed, in order to transmit a single event, at the earliest point, apart from the Release signal that was only active after communication had concluded, although it could have risen as soon as NewData rose, in order to better differentiate its impact from the other signals.

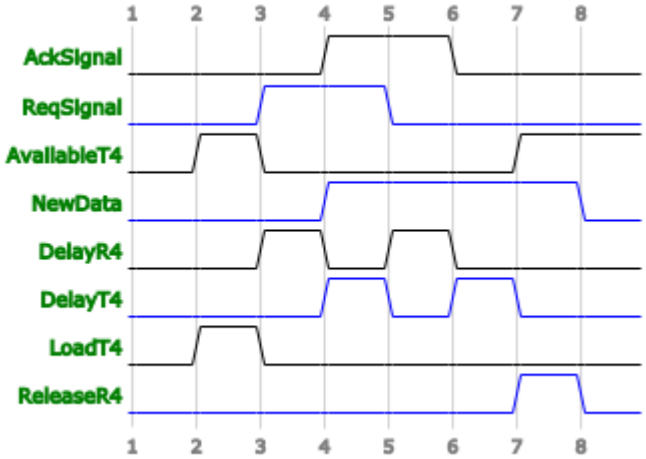

Figure 5.1 - Simulation of the Simple 4-Phase interface on the IOPT-Flow framework.

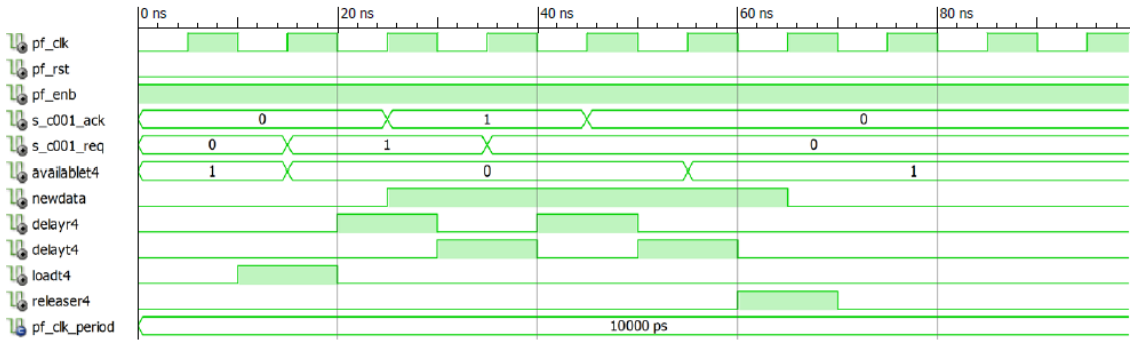

Figure 5.2 - Simulation of the Simple 4- Phase interface on the Xilinx ISE.

Initially, the available signal is the only signal to be high, informing that the transmitter is ready for communication. And so, in order to start transmission, the load signal becomes high (at step 2 in figure 5.1 and at 10ns in figure 5.2) for a single clock cycle, corresponding to the duration of an input event, causing the available signal to lower and the request signal to rise. Then, for the next four clock cycles, both communication delay signals will intersperse between one another, becoming high for a single clock cycle each. Acknowledge and Request signals will lower and rise interspersed according to the delay signals, following the 4-phase protocol. The NewData signal will rise (at step 4 in figure 5.1 and at 25ns in figure 5.2) once the request is first received, additionally, once the lowered acknowledge is received the Available signal will rise, allowing for new information to be loaded. Lastly, the Release signal indicates that the event has been retrieved by the synchronous block releasing the receiver and enabling a new communication cycle.

5.1.1.2 - 2-Phase

As for the 2-phase protocol of the Simple interface, a simple simulation was performed, only the minimum input signals were introduced for the transmission of two events. Because the 2-phase protocol reads transitions instead of values, the delay restriction will always consume an extra clock cycle, as for an actual implementation, as stated before, this delay restriction would be removed and so this extra clock cycle would not need to be expended.

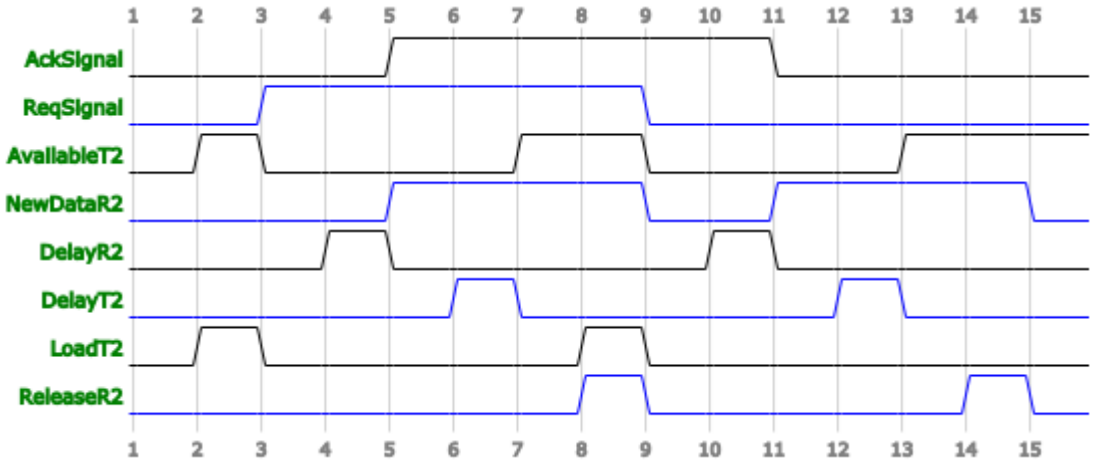

Figure 5.3 - Simulation of the Simple 2-Phase interface on the IOPT-Flow framework.

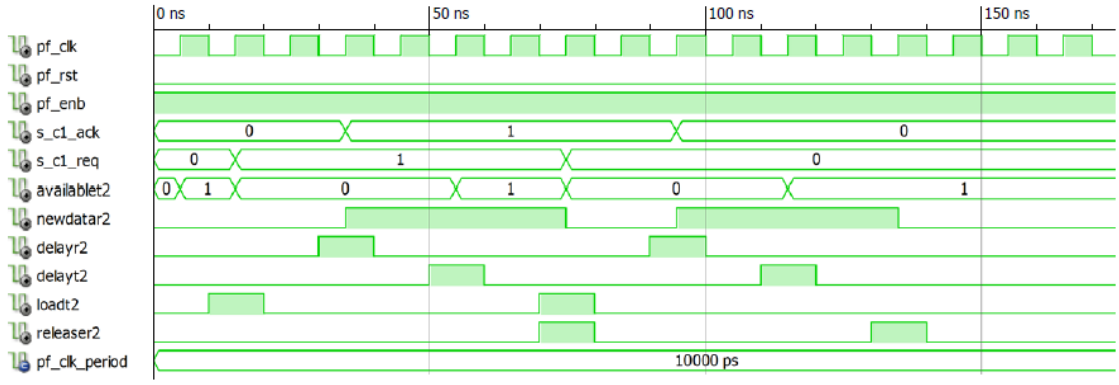

Figure 5.4 - Simulation of the Simple 2-Phase interface on the Xilinx ISE.

This set of simulations is, naturally, akin to its 4-phase counterpart, however, two communication cycles are represented instead of one. The Release signal occurs at the end of each communication cycle (for example, at step 8 in figure 5.3 and at 70ns in figure 5.4), in order to be easily distinguished. Some correlations may be noted between the input and output signals. The Load signal once risen will lower the available signal and shift the request signal (for example, at step 2 in figure 5.3 and at 10ns in figure 5.4). The delay signals will enable the communication signals to reach the opposite component, rising and lowering the communication signals and rising the NewData signal. The Release signal indicates that the new information has been retrieved and so the NewData signal will lower.

5.1.2 - SimpleBuffer

5.1.2.1 - 4-Phase

For this set of simulations, three events will be communicated, these are quickly loaded onto the transmitter before a new communication is established, then, one at a time, the transmitter's buffer will empty and the receiver's buffer will fill, once communication reaches a conclusion the three events are retrieved by the synchronous block. Additionally, two new signals are visible, a buffer for both the transmitter and the receiver.

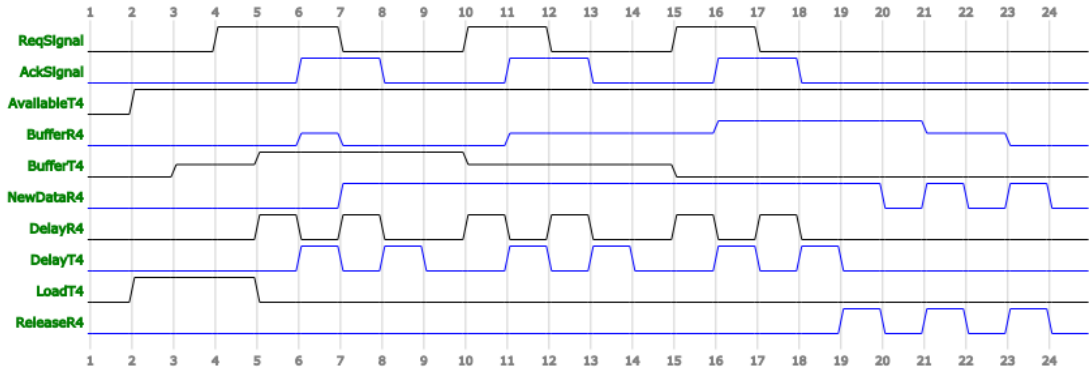

Figure 5.5 - Simulation of the SimpleBuffer 4-Phase interface on the IOPT-Flow framework.

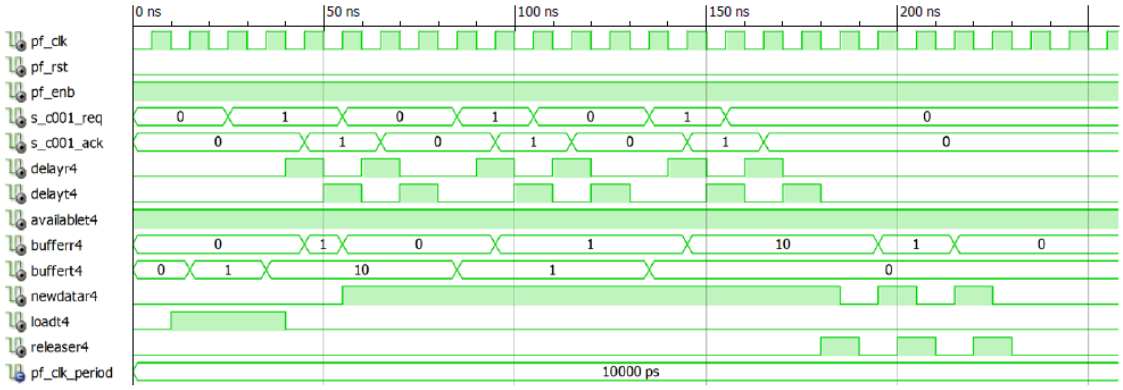

Figure 5.6 - Simulation of the SimpleBuffer 4-Phase interface on the Xilinx ISE.

The Load signal (at step 2 in figure 5.5 and at 10ns in figure 5.6) is held for three clock signals in order to load three events onto the buffer, afterwards, the buffer will only have two events, this happens because one of the events is consumed, raising the request signal. Only a single event may be consumed from the buffer in order to initiate communication, if communication is already taking place then no more events will be consumed since only one event may be communicated at a time. A similar effect may be visible on the receiver's buffer, the first event to reach the buffer will be held for a single clock cycle before being consumed, raising the NewData Signal (at step 7 in figure 5.5 and at 55ns in figure 5.6). Furthermore, while NewData is high no events will be consumed from the receiver's buffer since only one event may be retrieved at a time. While for the Simple interface the request signal rises immediately with the load signal, the SimpleBuffer interface takes an extra clock cycle since the events have to first pass through the buffer.

5.1.2.2 - 2-Phase

For this set of simulations, three events are transmitted. The behaviour is alike that of its 4-phase counterpart, except for the communication protocol that is, again, alike the Simple 2-phase.

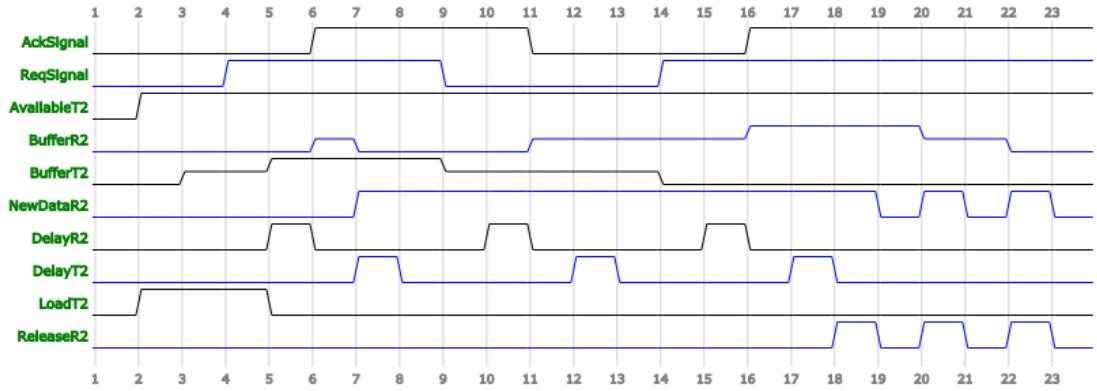

Figure 5.7 – Simulation of the SimpleBuffer 2-Phase interface on the IOPT-Flow framework.

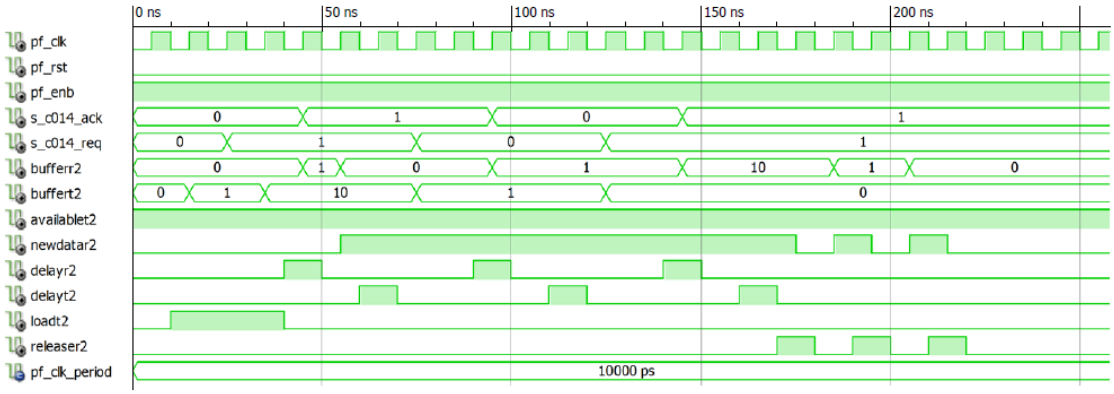

Figure 5.8 - Simulation of the SimpleBuffer 2-Phase interface on the Xilinx ISE.

When the receiver's delay signal is risen communication is allowed to proceed causing the receiver's buffer to be incremented by one (at step 5 in figure 5.7 and at 45ns in figure 5.8). The clock cycle that follows the receiver's buffer increment will decrement said buffer in order to rise the NewData signal, again, this does not happen during the following increments as there is already one event being communicated. The effect of the transmitter's delay will take a clock cycle

to decrement the transmitter's buffer and switch the request signal's value, again, due to the existence of a buffer.

5.1.3 - BurstBuffer

5.1.3.1 - 4-Phase

For this set of simulations, 4 events are first loaded onto the transmitter, then two communication cycles take place, the first communicating a single event and the next communicating the remaining three. These events are retrieved one at a time after the second communication cycle. Since data is being transmitted, a new signal is introduced in order to transmit said data, the DataWire signal for the IOPT-Flow simulation and, the equivalent signal, indata, for the Xilinx ISE.

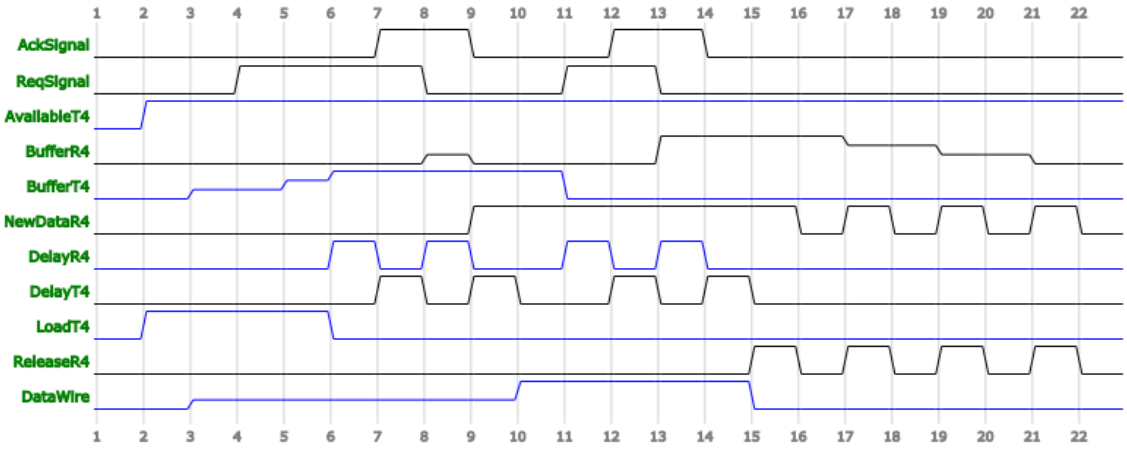

Figure 5.9 - Simulation of the BurstBuffer 4-Phase interface on the IOPT-Flow framework.

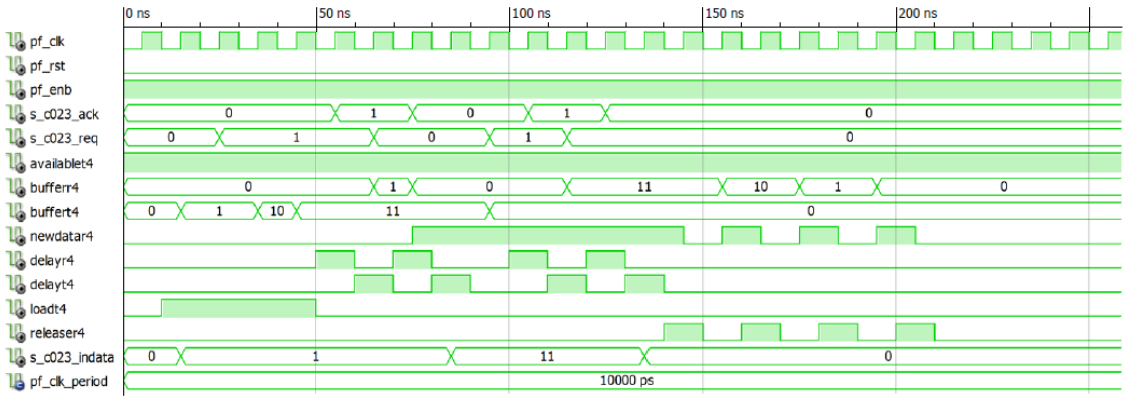

Figure 5.10 - Simulation of the BurstBuffer 4-Phase interface on the Xilinx ISE.

The Load signal (at step 2 in figure 5.9 and at 10ns in figure 5.10) is high for four clock cycles, resulting in a value of one in the DataWire signal and a value of three in the transmitter's buffer. The component does not know how many multiple events will be transmitted, and so, as soon as one is loaded it immediately starts communication, thus, a single event is transmitted. After the first communication cycle, an event is received by the receiver and is added to the buffer, since NewData is low, this event is immediately consumed, rising NewData (at step 9 in figure 5.9 and at 75ns in figure 5.10). When the second communication cycle begins, the three events are removed from the transmitter's buffer and transmitted on the DataWire (at step 10 in figure 5.9 and at 85ns in figure 5.10). It's relevant to note how since a late scheme is employed only after the lowered request is received, is the data from the DataWire added to the receiver's buffer. Again, the events are retrieved one at a time after communication has finished.

5.1.3.2 - 2-Phase

As for the 2-phase protocol set of simulations, four events are also transmitted through two communication cycles, and then retrieved by the receiving synchronous block. Again, the first cycle transmits a single event and the second transmits the remaining.

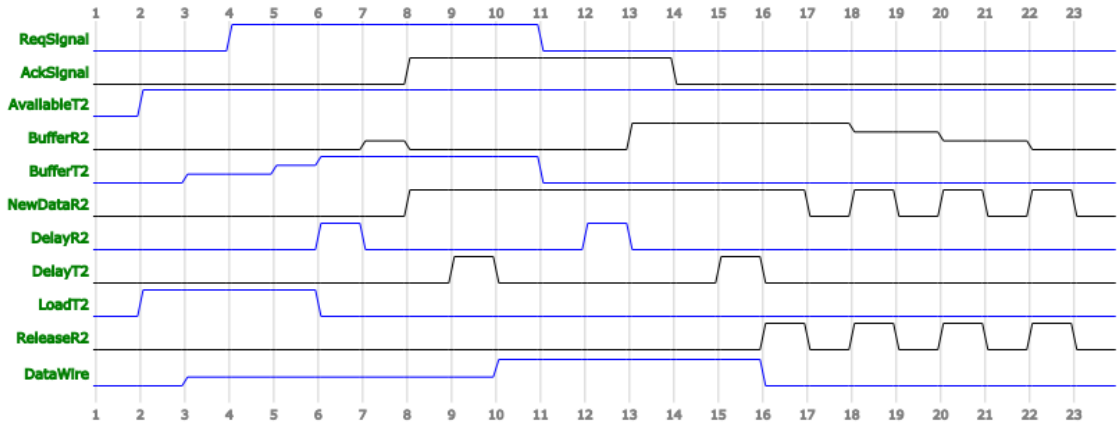

Figure 5.11 - Simulation of the BurstBuffer 2-Phase interface on the IOPT-Flow framework.

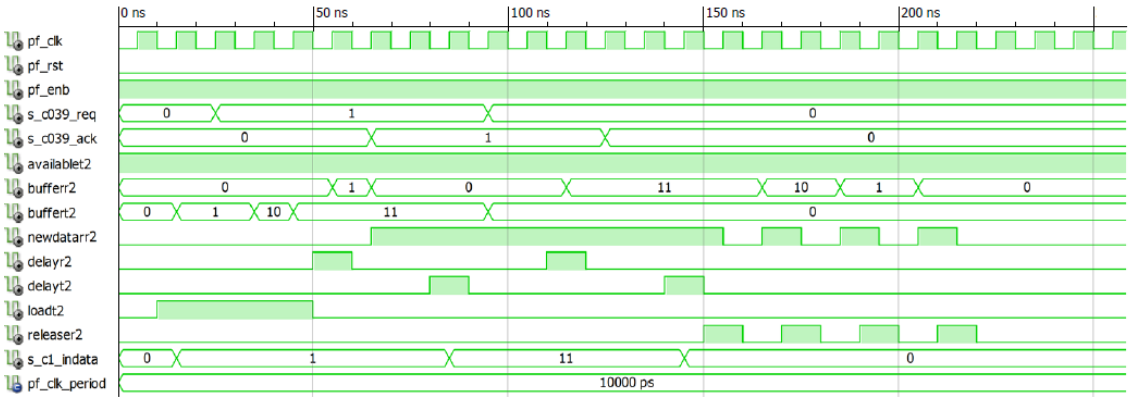

Figure 5.12 - Simulation of the BurstBuffer 2-Phase interface on the Xilinx ISE.

For the 2-phase protocol, a single pair of request-acknowledge signals is exchanged during one communication cycle, and so, data is received at the first delay signal, when the request signal is received (at step 6 in figure 5.11 and at 50ns in figure 5.12). Upon receiving the first event, the receiver's buffer is incremented, but since the NewData signal is lowered, the buffer will be decremented immediately on the following clock cycle, raising the NewData signal. The second delay in the communication cycle will enable the following communication cycle to start, and the DataWire signal's value will hold a value correspondent to the events stored in the transmitter's buffer.

5.1.4 - Data

5.1.4.1 - 4-Phase

The remaining interfaces transmit data instead of events. The data interface is the most straightforward after the Simple interface. For this set of simulations, a single data value, 5, is being transmitted. The DataWire signal in the IOPT-Flow waveform corresponds to the second indata signal, s_c073_indata, in the Xilinx ISE.

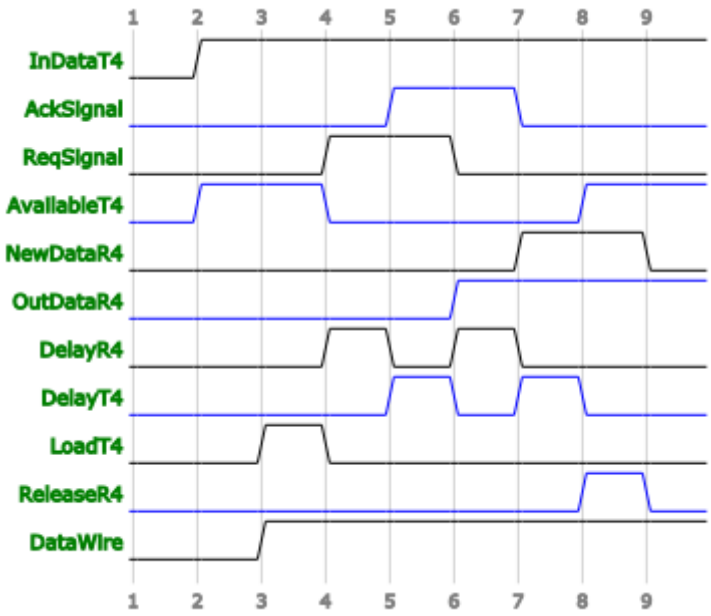

Figure 5.13 - Simulation of the Data 4-Phase interface on the IOPT-Flow framework.

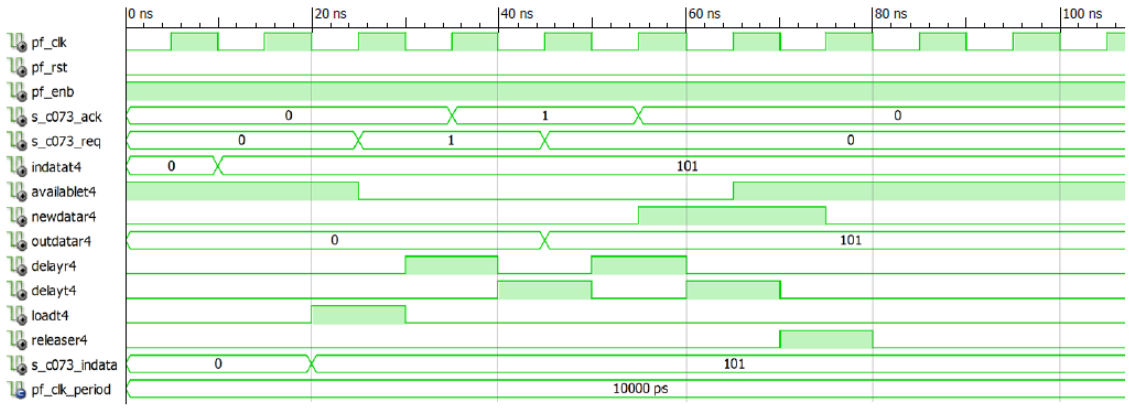

Figure 5.14 - Simulation of the Data 4-Phase interface on the Xilinx ISE.

First, the desired value is set in InData, for these simulations the chosen value was 5. This value is first set one cycle before the Load signal rises (at step 3 in figure 5.13 and at 20ns in figure 5.14); however, it would have an equivalent result were they to be set simultaneously. Upon receiving the Load signal, the value to transmit is read and mimicked by the DataWire signal (at step 3 in figure 5.13 and at 20ns in figure 5.14). The communication cycle behaves as the previous interfaces; however, the incoming data value is only read when the receiver receives the lowered request, due to a late data-valid scheme. Once the DataWire is read, the OutData signal will reflect its value (at step 6 in figure 5.13 and at 45ns in figure 5.14). On the following cycle the NewData signal will rise. Lastly, the Release signal behaves as before.

5.1.4.2 - 2-Phase

For the 2-phase data interface set of simulations, a single value, 5, is being transmitted.

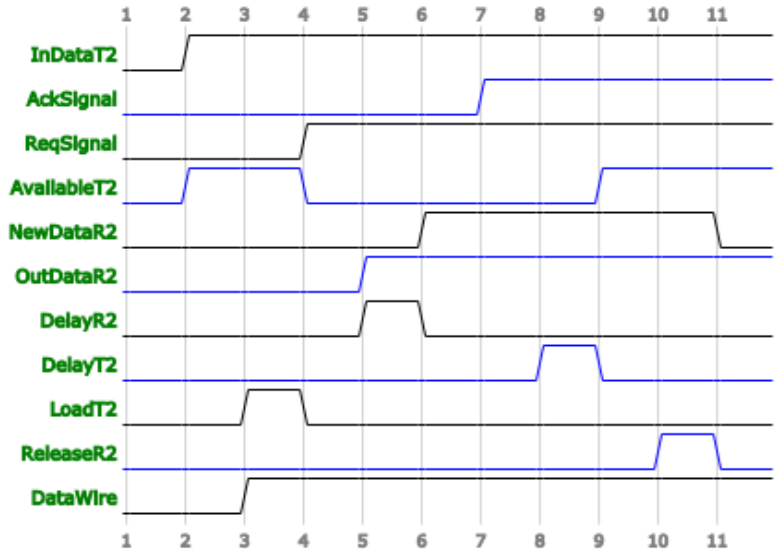

Figure 5.15 - Simulation of the Data 2-Phase interface on the IOPT-Flow framework.

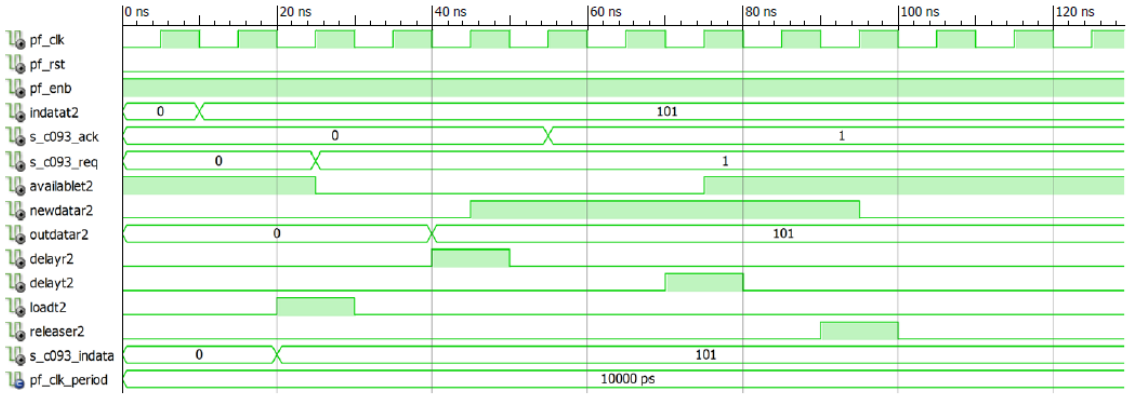

Figure 5.16 - Simulation of the Data 2-Phase interface on the Xilinx ISE.

The desired value is first set, then, the Load signal is risen for a single clock cycle (at step 3 in figure 5.15 and at 20ns in figure 5.16). The DataWire signal value is set, mimicking that of the InData signal . On the first delay signal, when the shift in the request signal is received, the data value is read and set on the OutData signal in order for it to be retrieved by the receiver's synchronous block. The second delay signal will release (at step 8 in figure 5.15 and at 70ns in figure 5.16) the transmitter allowing for a new communication cycle to begin.

5.1.5 - DataBuffer

5.1.5.1 - 4-Phase

This last interface is capable of holding multiple data values at a time. And so, in this set of simulations, three data values are transmitted, in order, 5, 10 and 15, three communication cycles are required. The values are then retrieved one at a time.

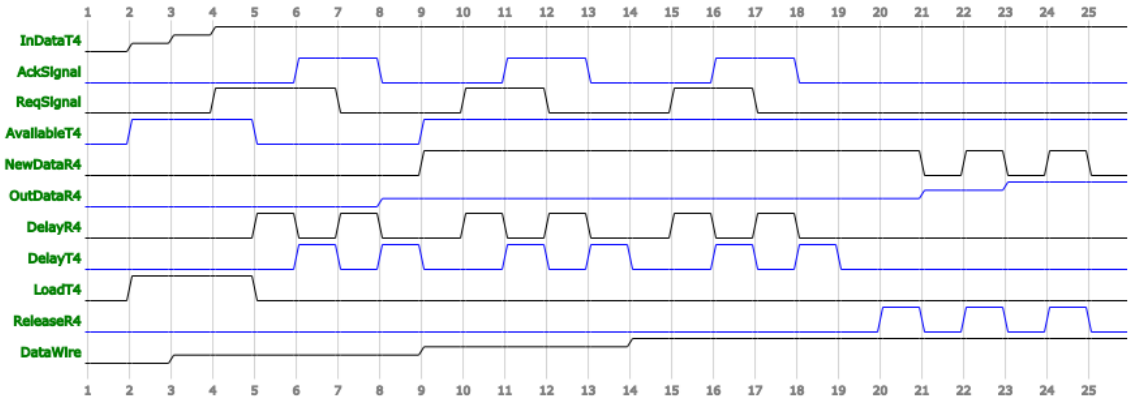

Figure 5.17 - Simulation of the DataBuffer 4-Phase interface on the IOPT-Flow framework.

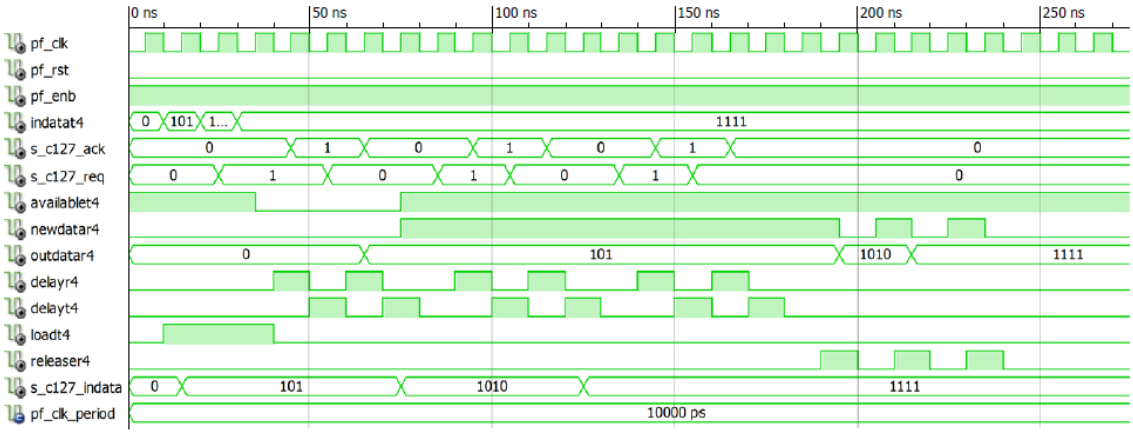

Figure 5.18 - Simulation of the DataBuffer 4-Phase interface on the Xilinx ISE.

The simulation starts with a Load signal that is high for three clock cycles accompanied by multiple values in the InData signal, these values are 5, 10 and 15 (at step 2 in figure 5.17 and at 10ns in figure 5.18). As the data values are loaded the buffer is incremented, after the third and last data value is loaded, into the transmitter, the buffer reaches its maximum capacity causing the Available signal to lower (at step 5 in figure 5.17 and at 35ns in figure 5.18). A clock cycle after the

first data value is loaded, the communication cycle starts with the rise of the Request signal, and however, it will stay halted until the first delay signal rises. Communication proceeds dependent on the delay signals, similar to the previous interfaces. The data value from the transmitter is read by the receiver on the third delay signal, of each communication cycle, the first data value read will also rise the NewData signal. Since three data values are being transmitted, three communication cycles are required in order to transmit them all. Lastly, these values are retrieved by the receiver's synchronous block one at a time.

5.1.5.2 - 2-Phase

For the last simulation set, three data signals will be transmitted. These are quickly loaded before the first delay signal rises. The following communication cycles transmit the data values one at a time. And lastly, these values are retrieved from the receiver's synchronous block.

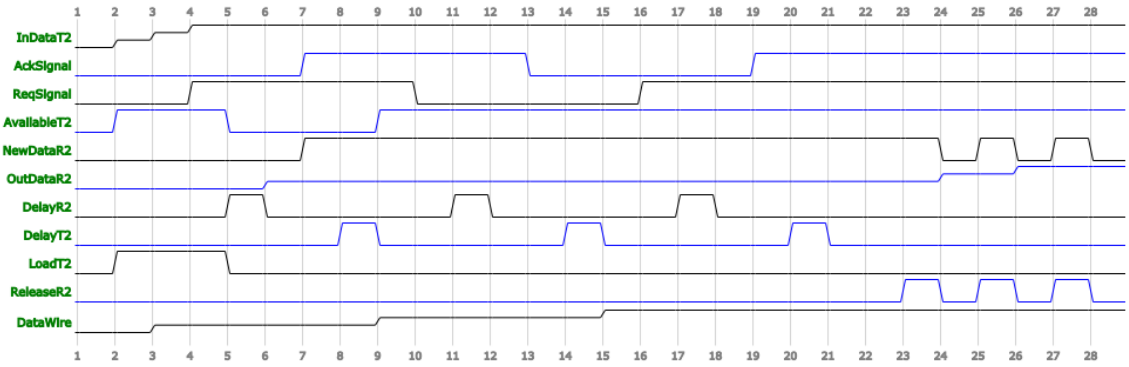

Figure 5.19 - Simulation of the DataBuffer 2-Phase interface on the IOPT-Flow framework.

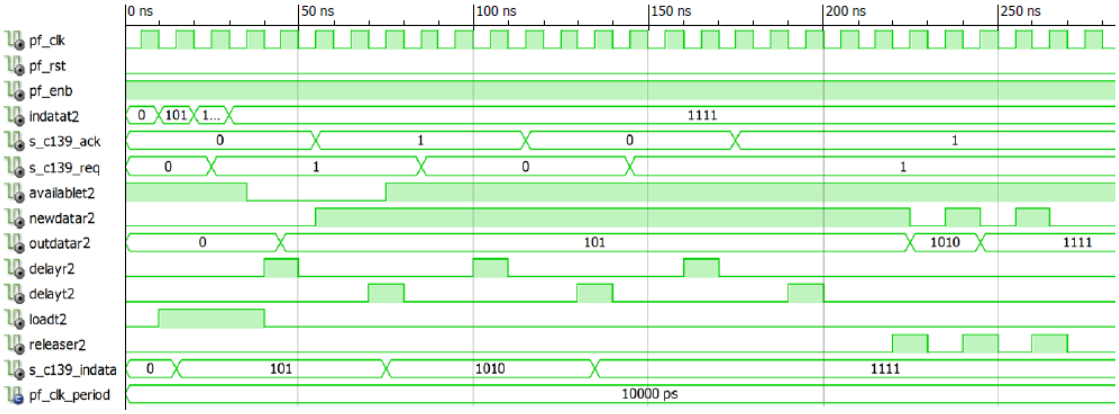

Figure 5.20 - Simulation of the DataBuffer 2-Phase interface on the Xilinx ISE.

Three values are loaded in quick succession before communication starts, these values are 5, 10 and 15 (at step 2 in figure 5.19 and at 10ns in figure 5.20). Afterwards, the delay signals will intersperse allowing for the Request and Acknowledge signals to be read, thus, transmitting the stored data values one at a time from the transmitter to the receiver. Lastly, the values will be retrieved by the receiver's synchronous block.

5.2 - Implementing Several Tests on an FPGA Board

After testing the asynchronous components in the IOPT-Flow simulator and the Xilinx ISE simulator, some more complex tests were employed. These were performed on a Nexys 4 DDR board (Figure 5.21), although the Nexys board possesses a single 100MHz crystal oscillator, the input clock can drive MMCMs or PLLs to generate several clock signals at various frequencies, and thus is capable of simulating GALS circuits. Furthermore, for all of the following tests the delay events were removed from the asynchronous components, as it was explained these were merely present to facilitate testing and debugging during development. Moreover, every single input coming from the other synchronous block should have a simple synchronizer in order to avoid metastability. The models were created in the IOPT-Flow framework and then exported as VHDL code with the function "Generate Modular VHDL Code", next a new project was created in the Xilinx ISE to which the generated VHDL source files were added. It is important to use the modular option in order to differentiate the multiple synchronous blocks to be able to later attribute to each a different clock signal.

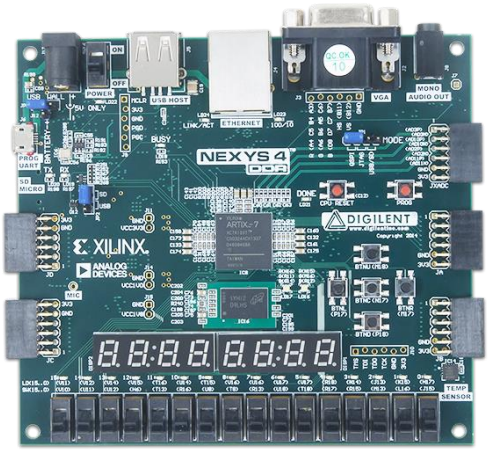

Figure 5.21 - Nexys 4 DDR board from Xilinx.[1]

Within the Xilinx ISE, first, the multiple clock signals must be created, the simpler approach to do so is to add a new "IP (CORE Generator & Architecture Wizard)" source file, then, under Clocking, select Clocking Wizard. Once the Clocking wizard window opens the desired clock signals should be chosen, upon conclusion the source file will be automatically generated. Lastly, the instantiation template should be added to the main VHDL module source file.

Since DS-Pnets and IOPT-Flow are not yet ready for a GALS system, some additional adjustments need to be made to the generated code. Enable signals are generated; however, because they are not used, they (PF_Enb and sPF_Enb) should be either commented or deleted, and set to '1' in each component's port map. Furthermore, any input or output that is initialized as an "integer range" should instead be initialized instead as a "std_logic_vector", this is done in order to be able to access the IO pins (LEDs, buttons, switches…).

Each value that was initialized as a "std_logic_vector" instead of an "integer range" will have some mismatch when atributions are made with internal signals. And so, there is a need for the library "IEEE.numeric_std.all" to convert signals from integer range to std_logic_vector the signal should be first converted to unsigned using the function "to_unsigned()" and then converted to std_logic_vector using the function "std_logic_vector()", to do the inverse conversion simply use the function "conv_integer()".

[1] https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual [37]

After the clocking source has been created and the instantiation template added to the main source file: the clocking port map should be adjusted; for the CLK_IN signal should be atributed PF_CLK; for the RESET should be attributed '0' or a different signal from the one that resets the synchronous blocks; and any other signal should have its own dedicaded internal signal. These signals will then be applied in the component's port maps. Additionally, a new reset signal should be created for each clock signal and synchronized to it, this will then be the new reset signals in the component's port maps. Furthermore, each input signal should be synchronized to the respective clock signal. Lastly, an execution process exists, in the main source, for all components and should instead be split accordingly to each synchronous block.

5.2.1 - Counting Events Transmitted with a Simple 4-phase Interface

The first test to be implemented onto the FPGA was a simple one. Two synchronous blocks are present, a transmitter and a receiver. The transmitter, will always load a new event if available, this is done by having a transition connected to both the available signal and the load event, additionally, there is an operation that counts the amount of times the Ack signal was lowered in order to count the amount of transmitted events, the result from this operation will be connected to an output signal named count_ack. As for the receiver, it merely responds to the transmitter, it will immediately release any new data, in a similar fashion, this is done through the use of a transition connected to both the NewData signal and the Release event.

There will be two clock signals, one will be the board's clock with a frequency of 100MHz and another with a frequency of 50MHz generated with the board's MMCMs and PPLs.

In terms of inputs and outputs, the pair of request/acknowledge signals is present as well as count_ack on the transmitter, this count_ack has a size of 26 bits and the 16 most significant ones will be displayed on the board's LEDs, since the first 10 bits will more too fast to be observable in real time.

Three models are created on the IOPT-Flow, the main, the receiver and the transmitter, the main model is simply the two others encapsulated and connected, the other two are represented in figures 5.22 and 5.23, respectively.

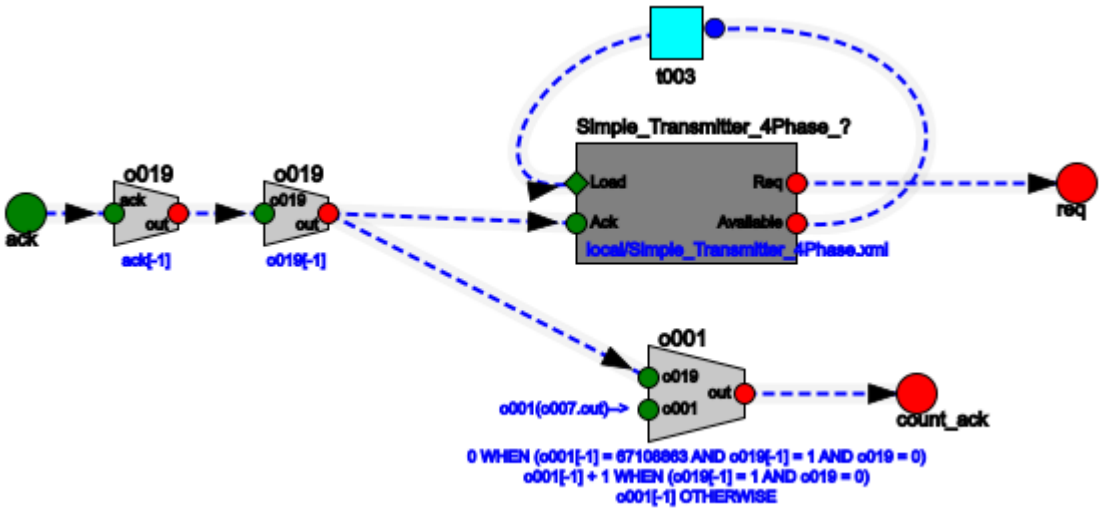

Figure 5.22 - Synchronous block transmitter, test one.

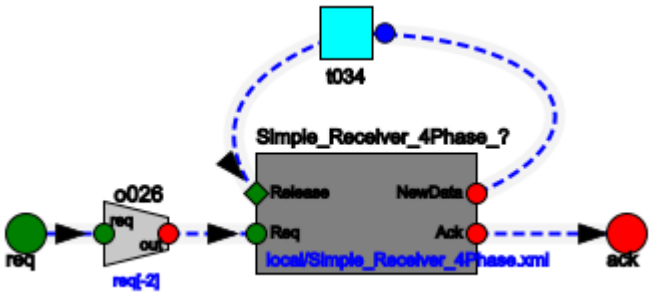

Figure 5.23 - Synchronous block receiver, test one.

In figure 5.24, is represented a snippet of the simulation performed in the ISim. It is visible that the count_ack increases once an ack is received, at the end of 1,217µs 1024 transmissions have succeeded, with a transmission from the moment the request rises until the next one does so takes seven clock cycles, or 0.7us.

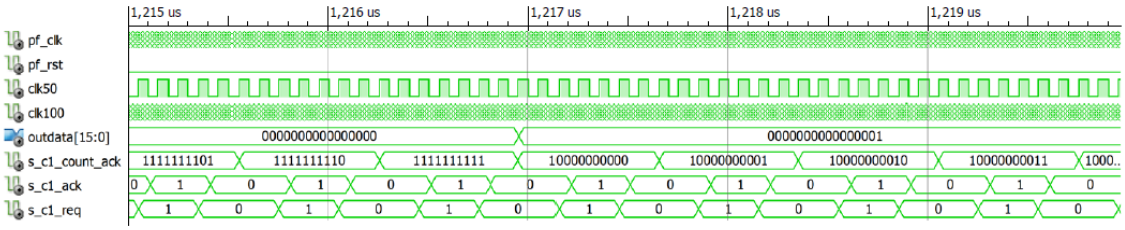

Figure 5.24 - Xilinx ISE wave graph, simulation one.

5.2.2 - Transmitting Events with a BurstBuffer 4-phase Interface

The second test is somewhat more complex than the previous one. Again, two synchronous blocks are present, the transmitter and the receiver. Here, the transmitter will possess a BurstBuffer 4-phase transmitter that will send events as long as the Available signal is up, similarly to the previous test, and the receiver will possess a BurstBuffer 4-phase receiver that will release the events, likewise, as soon as they are received. However, the transmitter will send a total of 256 events for a total of one byte and then stop, upon receiving exactly 256 events, if more are received an LED will be risen and the system is halted, an event will be sent through a Simple 4-phase interface. Upon receiving this event the system will restart, meaning that the transmitter will transmit new 256 events and the receiver will clear its total and receive the new 256 events, always restarting upon their retrieval.

There will be two clock signals, one will be the board's clock with a frequency of 100MHz and another with a frequency of 10MHz generated with the board's MMCMs and PPLs. The frequency was lowered for future tests in order for the results to be more easily visualized in real time.

In terms of inputs and outputs, there will be two pairs of request/acknowledge signals, as well as a data signal from the transmitter to the receiver. Lastly, the receiver will have an additional error signal, for when more events are transmitted that they were supposed to, and the total signal that will display the events that have been retrieved by the receiver. During test on the board, the error signal LED will not ever light up and the total signal, that will be displayed as eight LEDS, will appear to have them all turned on permanently, this happens because they signal moves faster than the eyes can observe due to the high frequency. Nonetheless, if there were an error the total signal would halt at a value different than 255 and the error led would light up.

Again, three models are created on the IOPT-Flow, the main, the receiver and the transmitter, the main model is simply the two others encapsulated and connected, the other two are represented in figures 5.25 and 5.26, respectively.

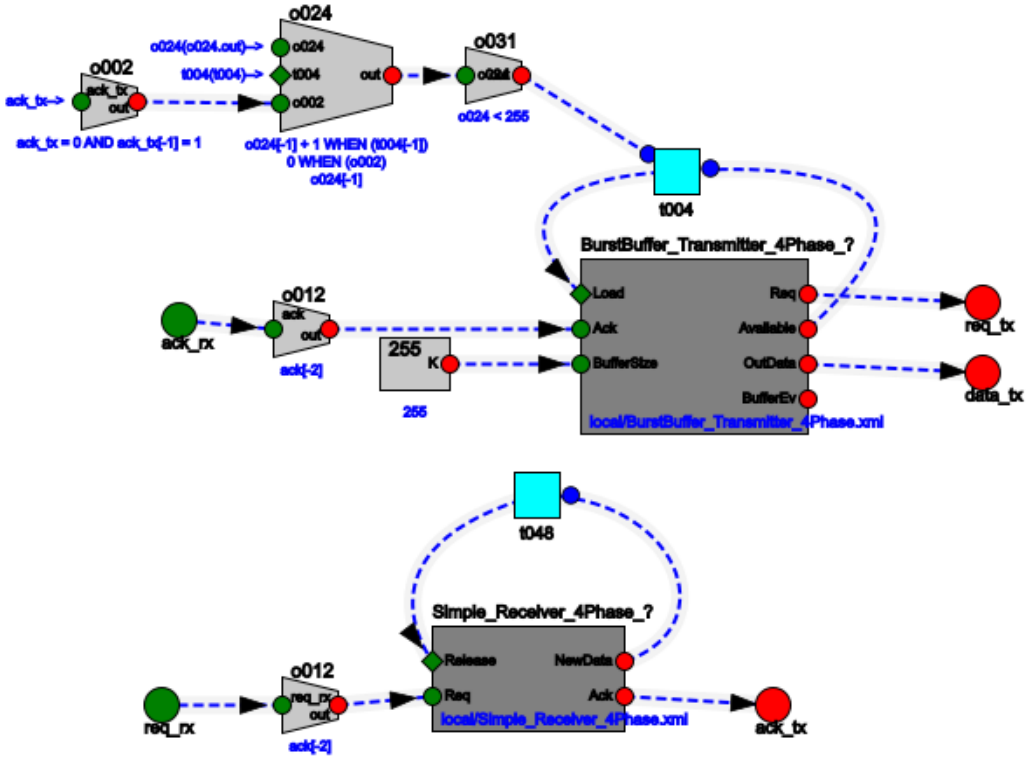

Figure 5.25 - Synchronous block transmitter, test two.

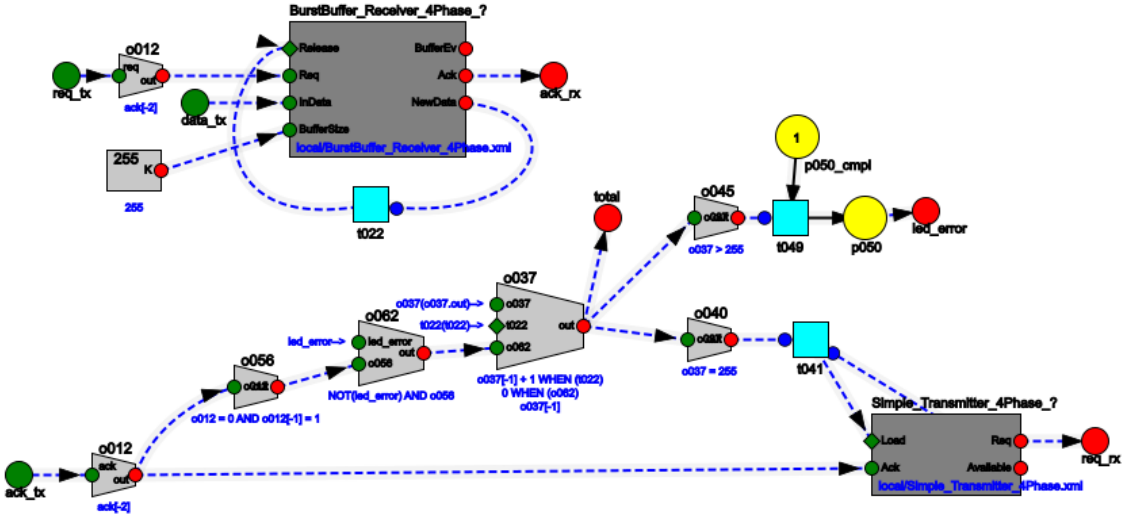

Figure 5.26 - Synchronous block receiver, test two.

In figure 5.27, is represented a snippet of the simulation performed in the Xilinx ISE. This particular section represents the interval in which the old cycle concludes and a new one begins. The first pair of communication signals correspond to the transmission that resets the system, consequently, the second pair of communication signals and their correspondent data signal begin to send events in form of data.

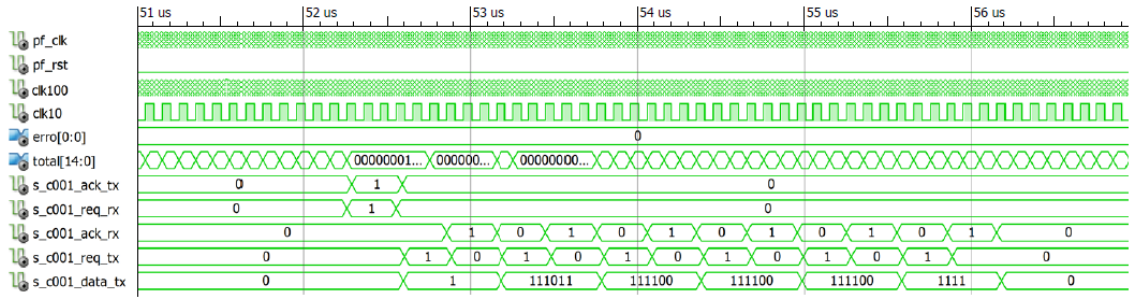

Figure 5.27 - Xilinx ISE wave graph, simulation two.

5.2.3 - Data Transmission

5.2.3.1 - With a Data 4-Phase Interface

The third test focusses on data transmission, more specifically, the Data 4-phase interface. Two synchronous blocks are present, a data value with a size of 8 bits will be loaded onto the transmitter block that will then transmit this value to the receiver block, causing it to, consequently, display the received data value. Contrary to the previous tests, the load and release of data is manual, meaning, they have dedicated buttons on the board.

The clock signals will remain from the previous interface and be maintained until stated otherwise.

In terms of inputs and outputs, the pair of request/acknowledge signals and the data signal are the only ones that connect the two synchronous blocks. The transmitter will have a load input signal, which is connected to a button on the board, and the indata input signal, which, in its place, is connected to eigth switches, one for every bit. On the other hand, the receiver will have the release input signal, connected to a button, and the outdata output signal connected to eight LEDs, again, one for every bit. To clarify, the indata input signal is the data value loaded onto the transmitter, the data signal is the one that connects the transmitter to the receiver and outdata signal is the data value that is retrieved from the receiver.

Once more, three models are created on the IOPT-Flow, the main, the receiver and the transmitter, the main model is simply the two others encapsulated and connected, the other two are represented in figures 5.28 and 5.29, respectively.

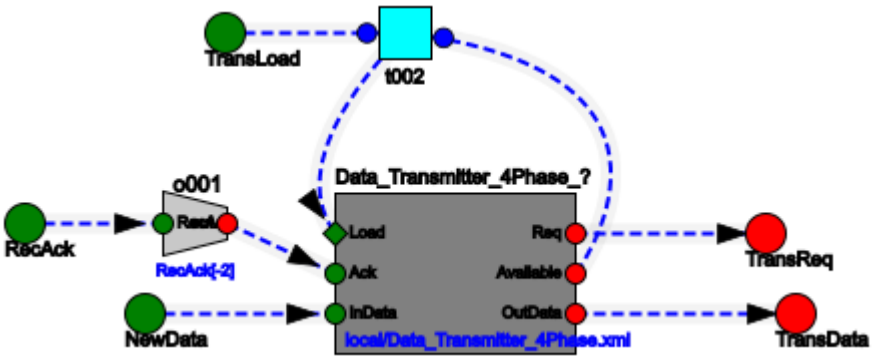

Figure 5.28 - Synchronous block transmitter, test three.

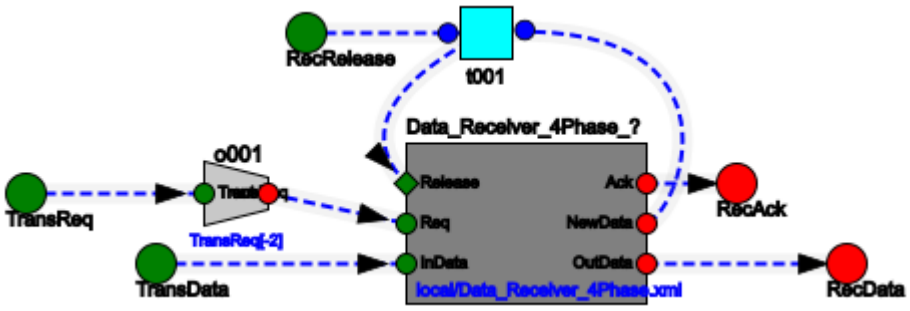

Figure 5.29 - Synchronous block receiver, test three.

In figure 5.30, is represented a snippet of the simulation performed in the Xilinx ISE. Here, the NewData signal is set at 200ns, the load signal is high from 300ns and 400ns. It's important to clarify that the clock signals take some time to be generated, and for this reason, the Load signal is only recognized much later than would be expected. Despite this, communication ensues and moments later the OutData signal reflects the data values introduced earlier.

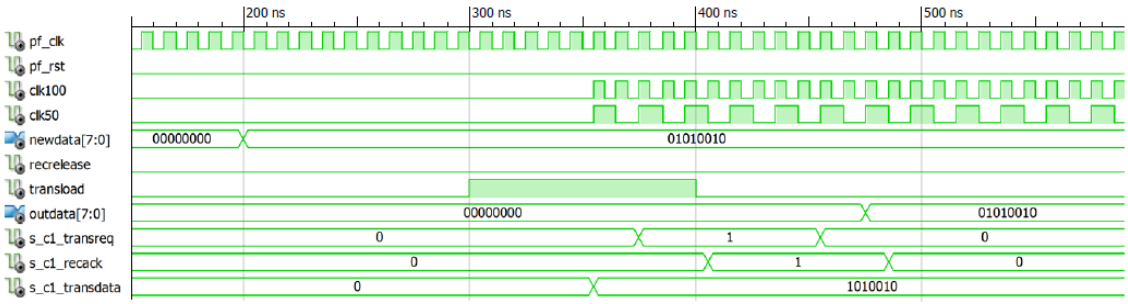

Figure 5.30 - Xilinx ISE wave graph, simulation three.

5.2.3.2 - With a DataBuffer 4-Phase Interface

This test close resembles the previous one. There are two differences, the first and most obvious is that a different interface is being used. As it was mentioned before in chapter 3, the Databuffer interface not only transmits data but also is capable of holding up to three values in its buffer. Due to the presence of this buffer extra care has to be had with the load input, and so the second difference from the previous interface is the presence of an operation that is only high if the load input signal is currently high but was not during the previous cycle. This operation was employed in order to avoid loading multiple events with a single press due to the clock signal frequency being so high. An identical operation was employed for the release input signal for the same reasons.

One more time, three models are created on the IOPT-Flow, the main, the receiver and the transmitter, the main model is simply the two others encapsulated and connected, the other two are represented in figures 5.31 and 5.32, respectively.

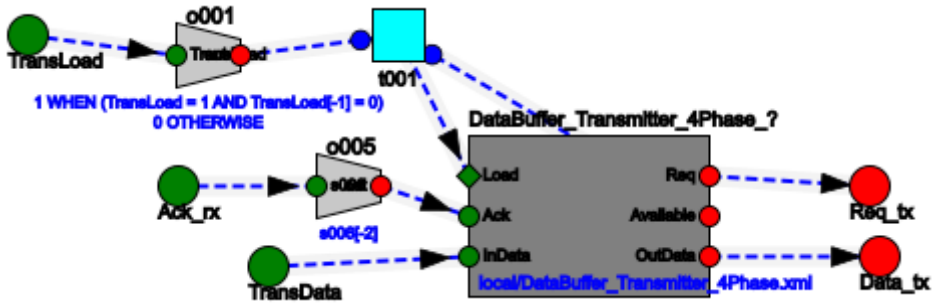

Figure 5.31 - Synchronous block transmitter, test four.

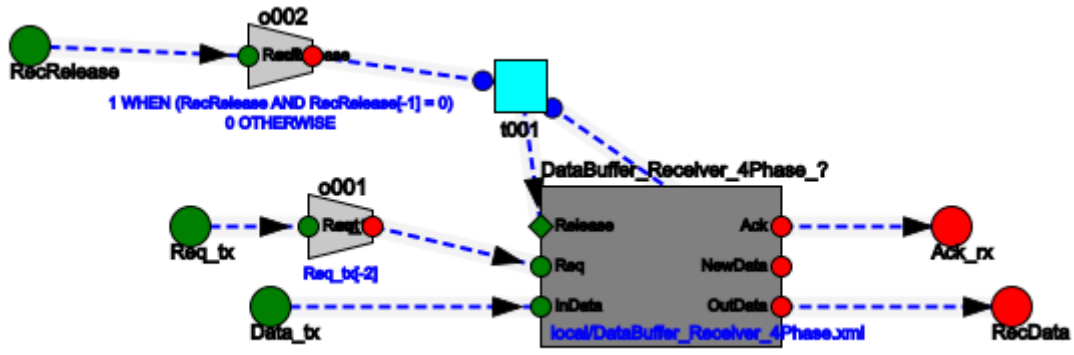

Figure 5.32 - Synchronous block receiver, test four.

5.2.4 - 2-Phase Implementations of Previous Tests

In order to validate the interfaces that employ a 2-phase protocol instead of a 4-phase protocol, three additional tests were executed. These are adaptations from previous tests, more specifically, counting events transmitted with a Simple 4-phase interface, transmitting events with a BurstBuffer 4-phase interface and data transmission with a Databuffer 4-phase interface, respectively, from sub-chapters 5.2.1, 5.2.2 e 5.2.3.2. The single small adjustment, besides the switch of interfaces, was that for all implementations, the clock signals were 100MHz and 10MHz.

From figure 5.33 to figure 5.38 are represented the synchronous blocks for the 2-phase implementations in the order they were presented in, for each pair of figures, first is displayed the transmitter and then the receiver.

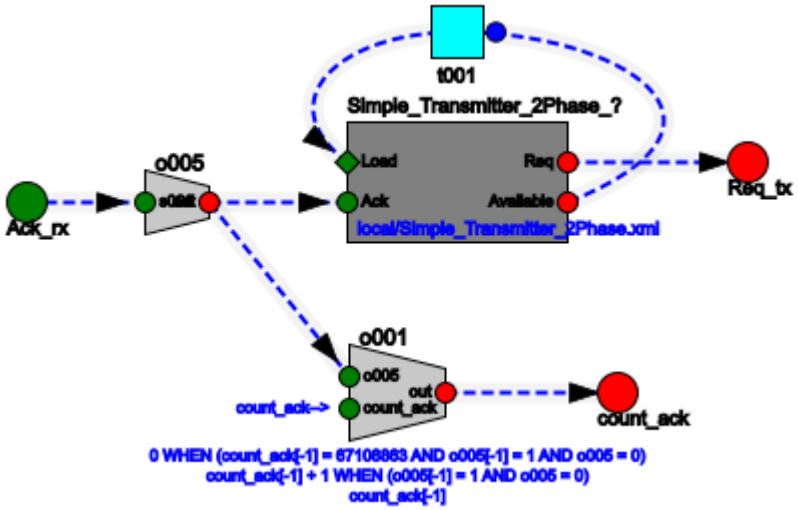

Figure 5.33 - Synchronous block transmitter, test four.

Figure 5.34 - Synchronous block receiver, test five.

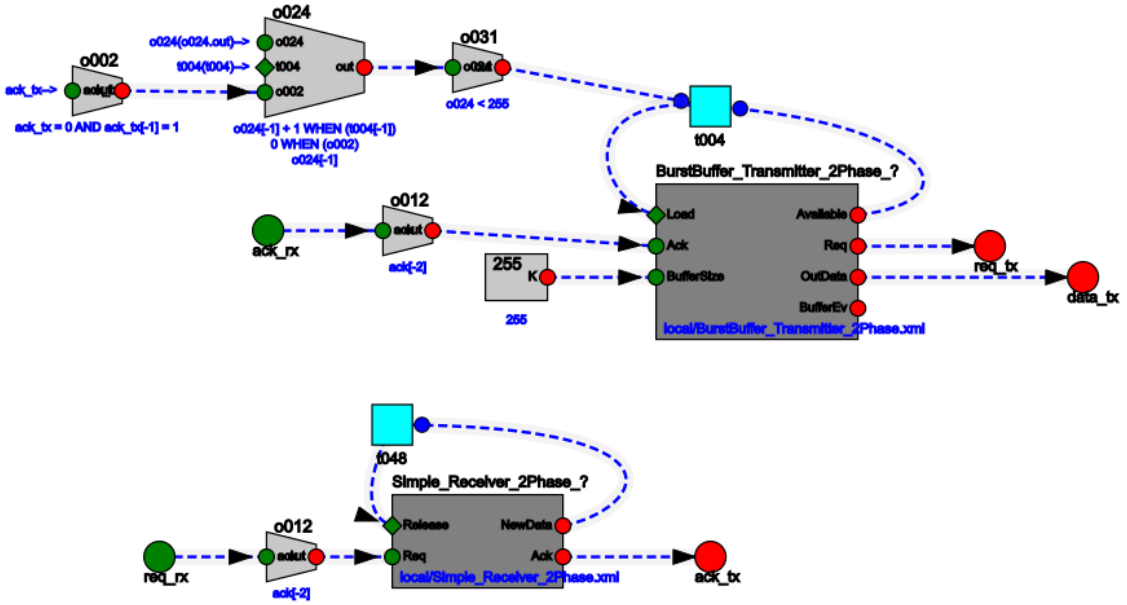

Figure 5.35 - Synchronous block transmitter, test six.

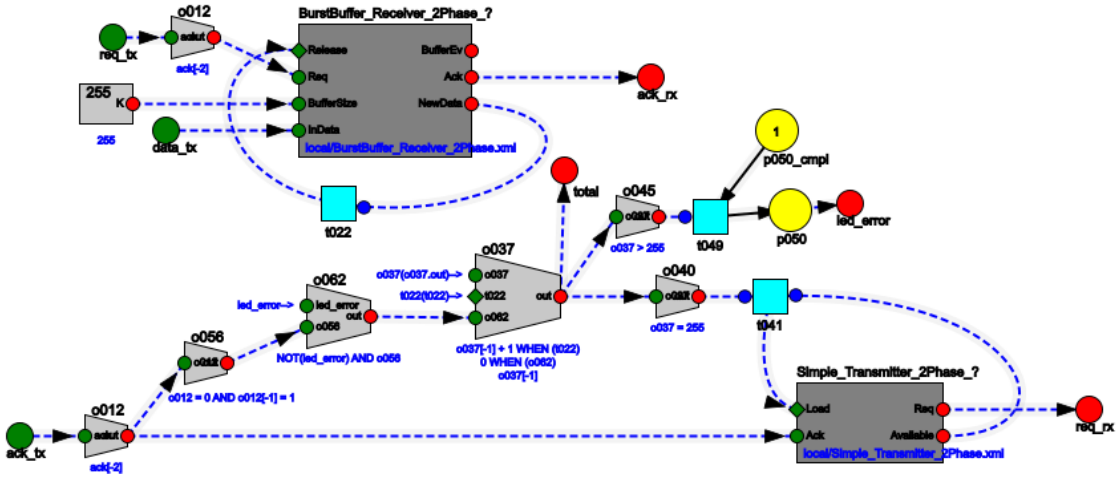

Figure 5.36 - Synchronous block receiver, test six.

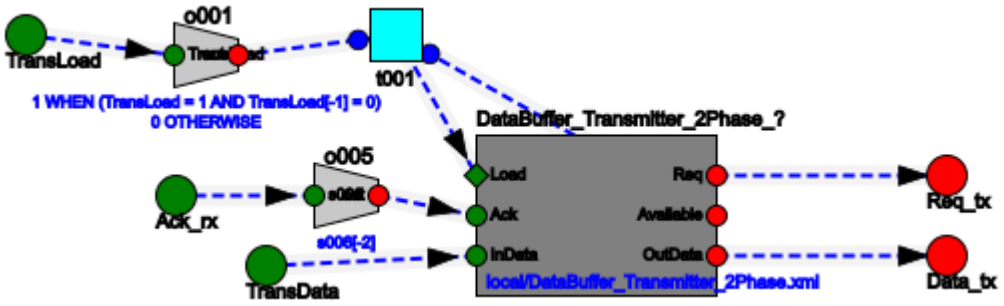

Figure 5.37 - Synchronous block transmitter, test seven.

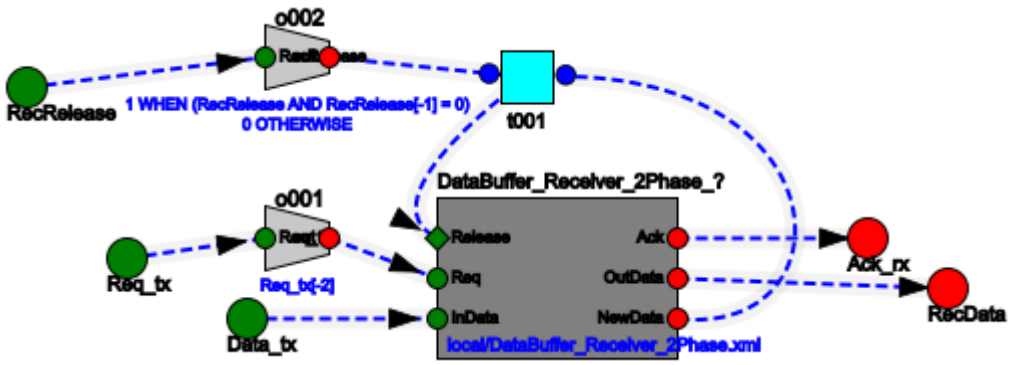

Figure 5.38 - Synchronous block receiver, test **seven.**

In figure 3.9, is represented a snippet of the simulation performed in the Xilinx ISE. Again, this is a 2-phase implementation of the first simulation and for that reason it is very similar to it. It is, however, important to note that one of the clock signal's frequency is much lower, and yet the count_ack signal reaches the same value shortly after.

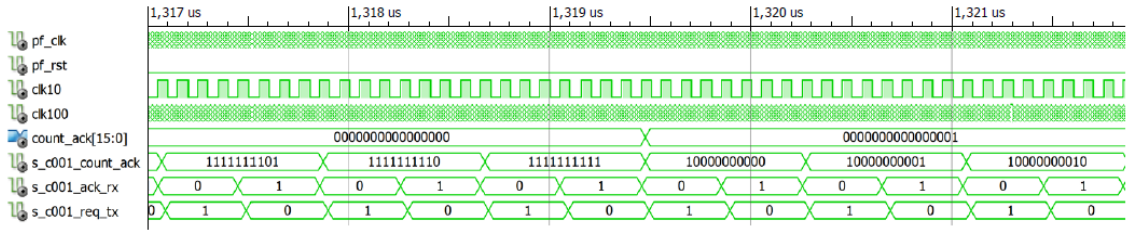

Figure 5.39 - Xilinx ISE wave graph, simulation seven.

5.2.5 - Comparing Clock Edges to Event Transmission

5.2.5.1- With a Simple 4-Phase Interface

This test is similar to the first one, it merely transmits events uninterrupted. However, clock edges are counted, and events loaded are counted, each through the means of an operation. Furthermore, both of these can be stopped through the press of a button, in order for it to be possible to register the values of the counters on the FPGA board.

The counters will hold values with a maximum of 31 bits, since integers in VHDL has a maximum restriction of 32 unsigned bits. The FPGA board used has a total of sixteen LEDs, consequently, switched are used to access the desired bits of each counter. Two switches were used, the first determines what half of the

integer is displayed, the first sixteen bits or the latter sixteen bits, and the second switch determines which counter is displayed.

Two models were created in the IOPT-Flow framework, the main and the transmitter models, for the receiver model was used the one from the first test. The transmission model is represented in figure 3.40.

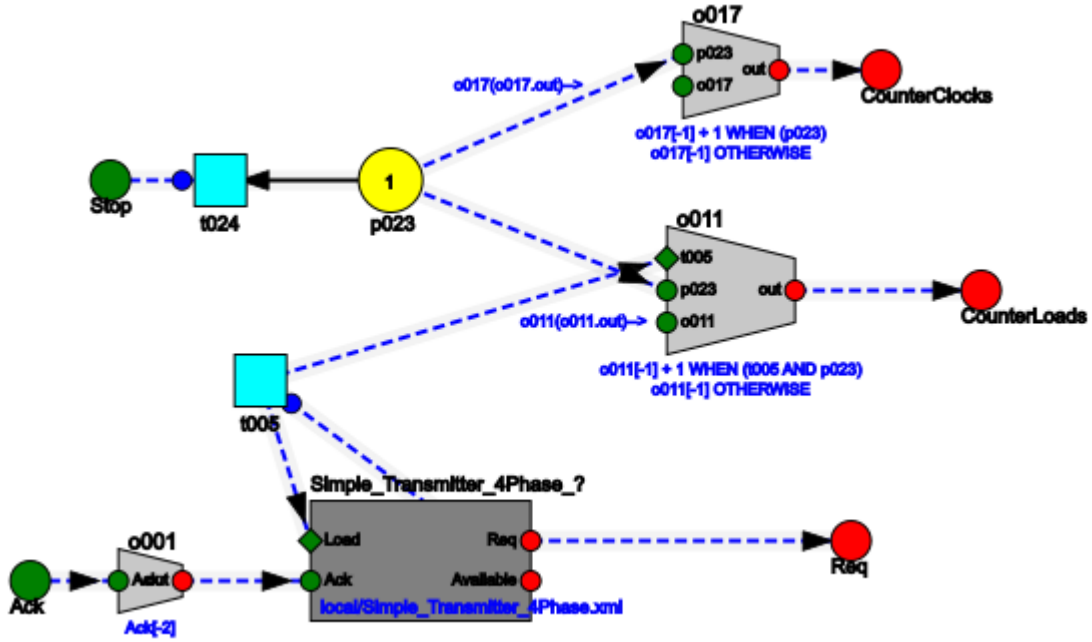

Figure 5.40 - Synchronous block transmitter, test eight.

In figure 3.41, is represented a snippet of the simulation performed in the Xilinx ISE. The switch inputs were not utilized since both counters are visible during simulation on the computer. It can be noted that it takes the 695 clock edges to reach 100 event loads, since it takes seven clock edges to transmit an event, assuming the transmitter clock frequency is lower.

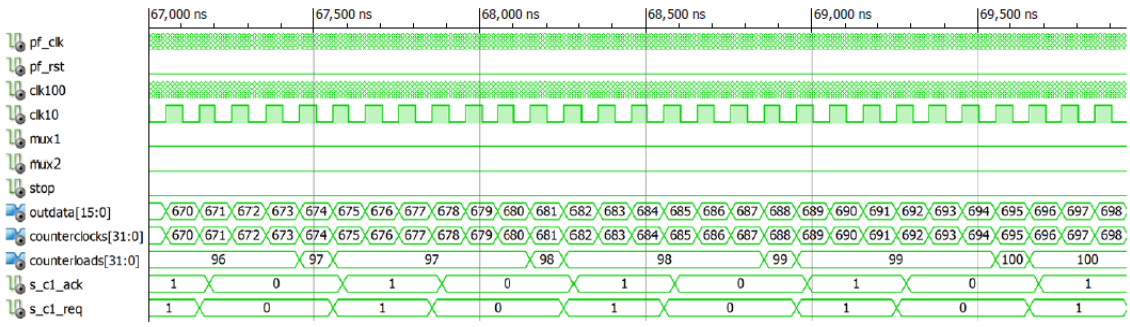

Figure 5.41 - Xilinx ISE wave graph, simulation eight.

5.2.5.1- With a BurstBuffer 4-Phase Interface

This test is similar to the previous one, though it instead employs a Burst-Buffer interface. Here, three actions are counted, the clock edge, the event loads and the burst transmissions. Once again, these counters may be halted by pressing a button. Moreover, there will be an additional switch in order to access the three counters, the first switch will still be responsible for which half of the counter it is visible, and the other two switches will determine which counter is visible.

Again, three models are created on the IOPT-Flow, the main, the receiver and the transmitter, the main model is simply the two others encapsulated and connected, the other two are represented in figures 5.42 and 5.43, respectively.

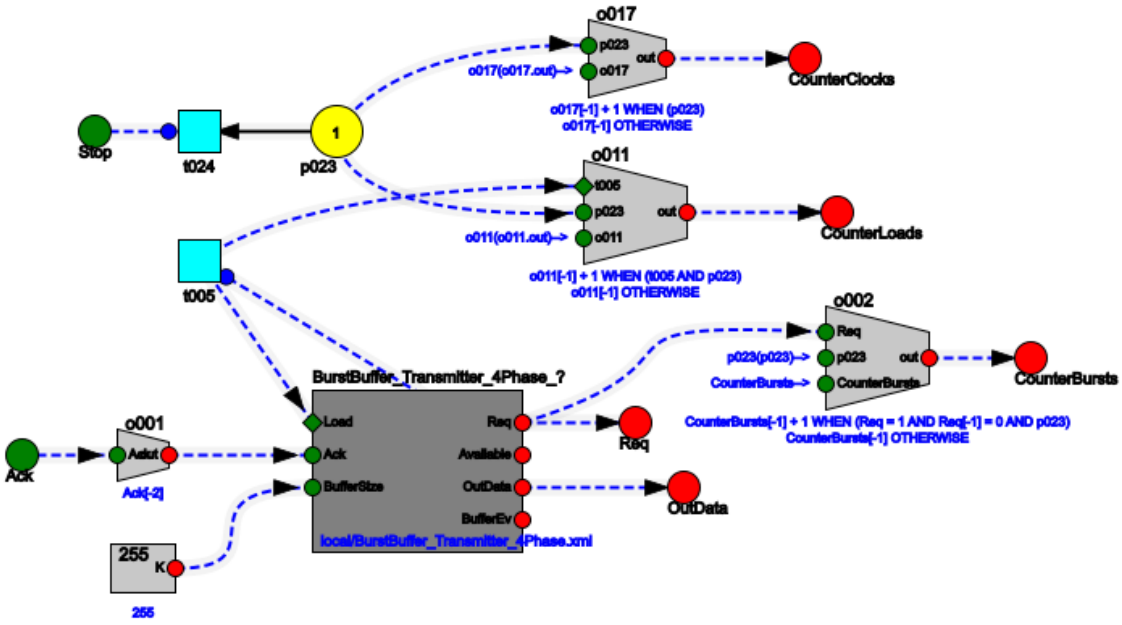

Figure 5.42 - Synchronous block transmitter, test nine.

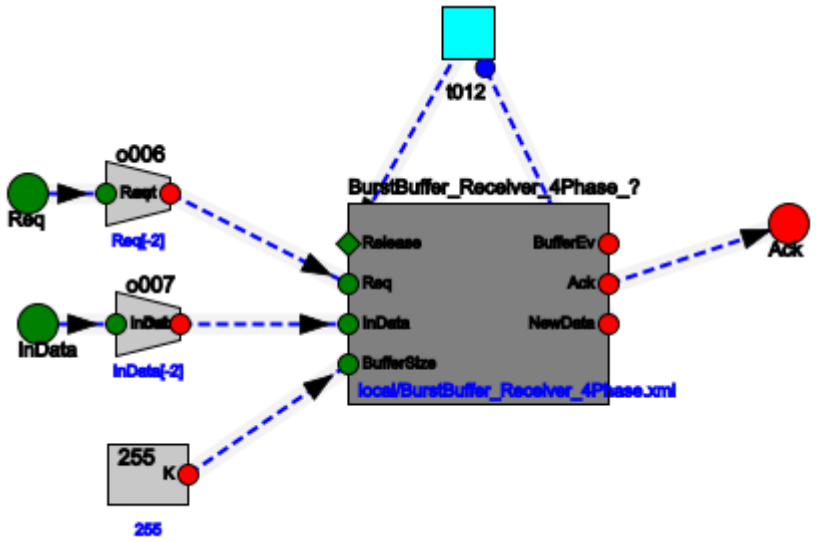

Figure 5.43 - Synchronous block receiver, test nine.

In figure 5.44, is represented a snippet of the simulation performed in the Xilinx ISE. Here, the load counter will accompany the clock edge counter while the transmitter's buffer is not full, however once it does become full the load counter will stop and the clock edge counter will continue. During this simulation, due to the receiver's clock signal having a lower frequency it will struggle to retrieve all the events stalling communications. After some time, the transmitter's buffer will fill between each communication cycle, since, again, the receiver is much slower and it will take longer to retrieve all the events than the transmitter takes to load completely load the buffer.

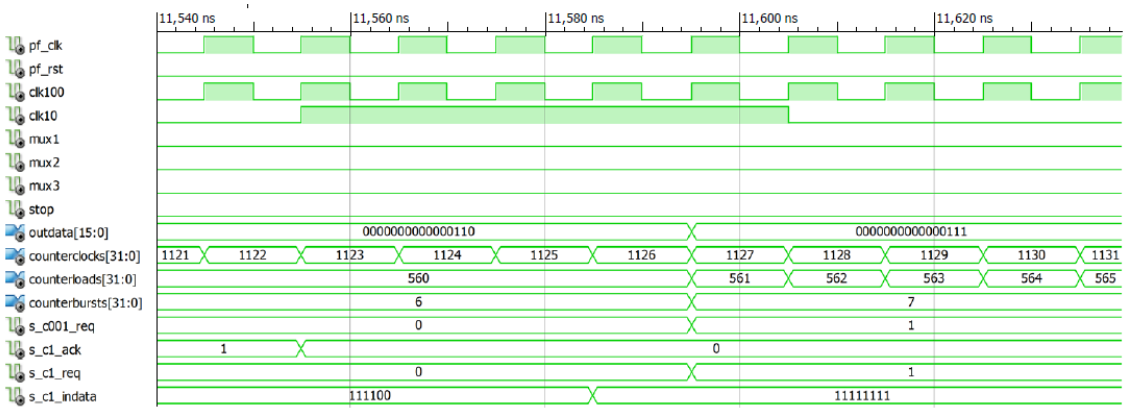

Figure 5.44 - Xilinx ISE wave graph, simulation nine.

5.2.6 - Data verification

For the last test, data is sent from one asynchronous block to the other and then returned, if it matches, the value is incremented by one and the process repeats. The default maximum size for data transmission is one byte, which will make it difficult to observe the values changing in real time, nonetheless, on the board's LEDs will be displayed both the data value sent and the data value receives, this is done simultaneously using eight LEDs to represent one value and the remaining eight to represent the other.

One of the clock signal's frequency has been lowered, now one clock signal will have a frequency of 5MHz and the other a clock frequency of 10MHz. It would be desirable to have an even lower frequency in order to easily observe the values being changed on the board, but such is not possible.

One more time, three models are created on the IOPT-Flow, the main, the receiver and the transmitter, the main model is simply the two others encapsulated and connected, the other two are represented in figures 5.45 and 5.46, respectively.

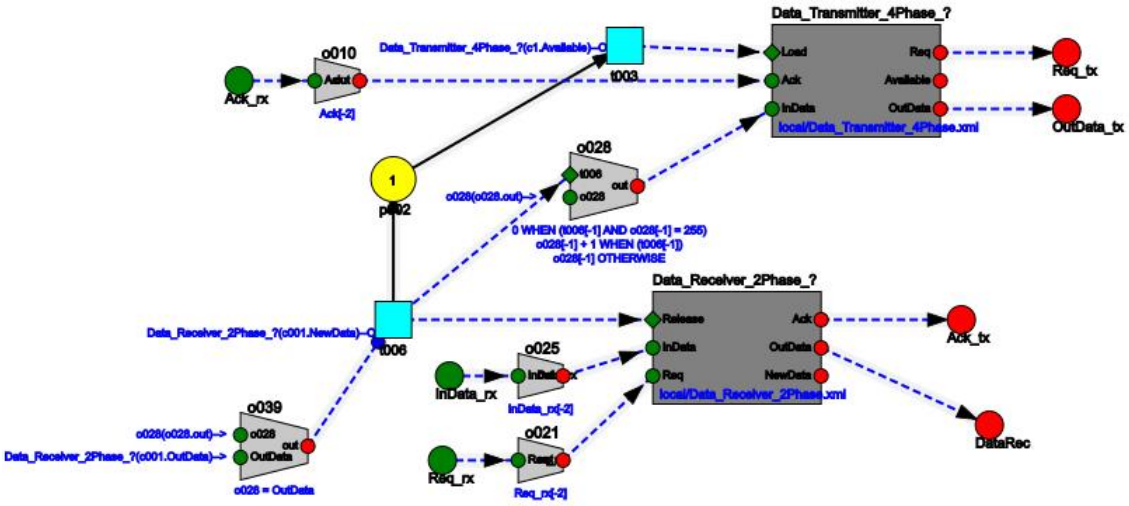

Figure 5.45 - Synchronous block transmitter, test ten.

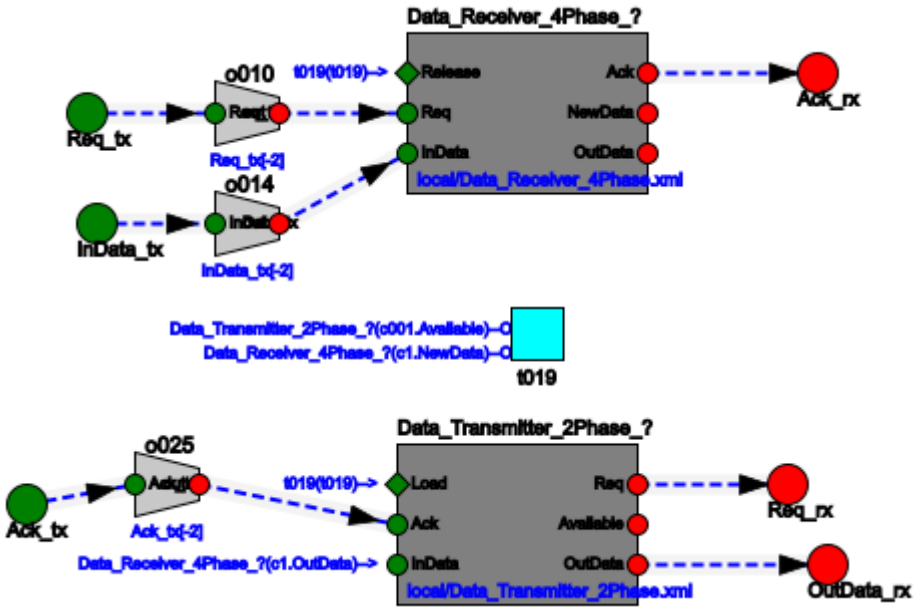

Figure 5.46 - Synchronous block receiver, test ten.

In figure 5.47, is represented a snippet of the simulation performed in the Xilinx ISE. The first pair of communication signals corresponds to the transmitter sending data to the receiver, and the second pair corresponds to the opposite. As it would be expected, the data sent during the first transmission corresponds to that of the second transmission.

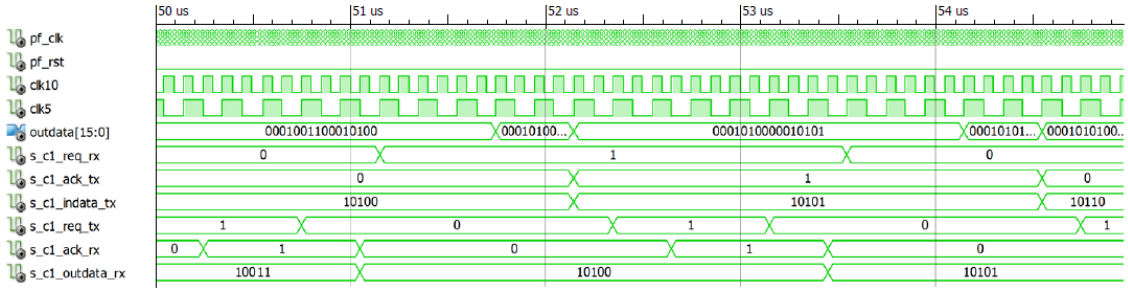

Figure 5.47 - Xilinx ISE wave graph, simulation ten.

5.3 - GALS Simulation in the IOPT-Flow Framework

Some adjustments were carried out to realize a GALS simulation in the IOPT-Flow Framework, in the interest of validating the execution semantics proposed in chapter four. On a copy of the tool, in order to avoid altering the current framework as these changed might hinder it, a specialized simulation mode was

implemented that would not run the generated JavaScript code and would instead run JavaScript code that was loaded from a fixed directory. Additionally, an input signal should be added for each desired clock signal. The JavaScript code that is run during this specialized simulation mode should take root in the one that is generated by the tool but is altered to implement the execution semantics proposed for GALS in chapter four.

Multiple simulations were employed in the modified IOPT-Flow framework, and in this document two are presented, the first simulation from the previous subchapter that employs a Simple 4-phase interface to transmit events uninterrupted and the fourth simulation from the previous subchapter that employs a DataBuffer 4-phase interface to transmit data from one synchronous block to the other. Figures 5.48 and 5.49 illustrate, respectively, the results of said simulations.

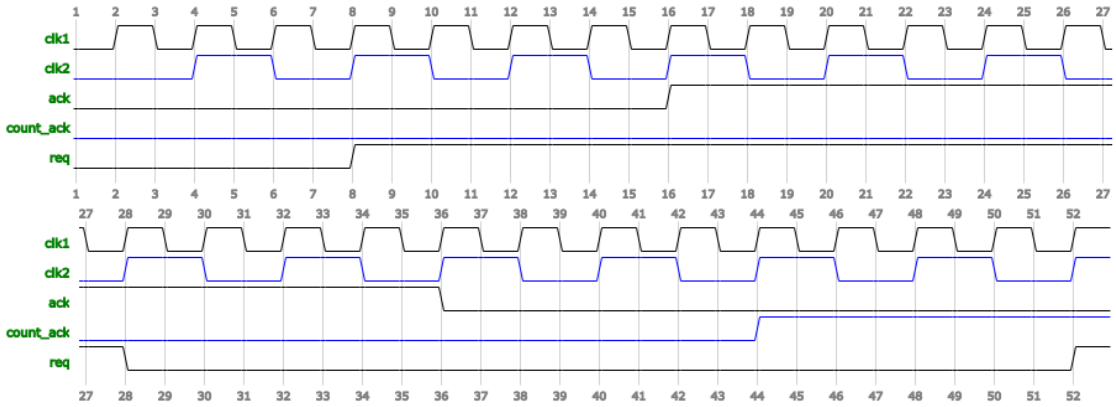

Figure 5.48 - IOPT-Flow wave graph with proposed execution semantics, simulation one.

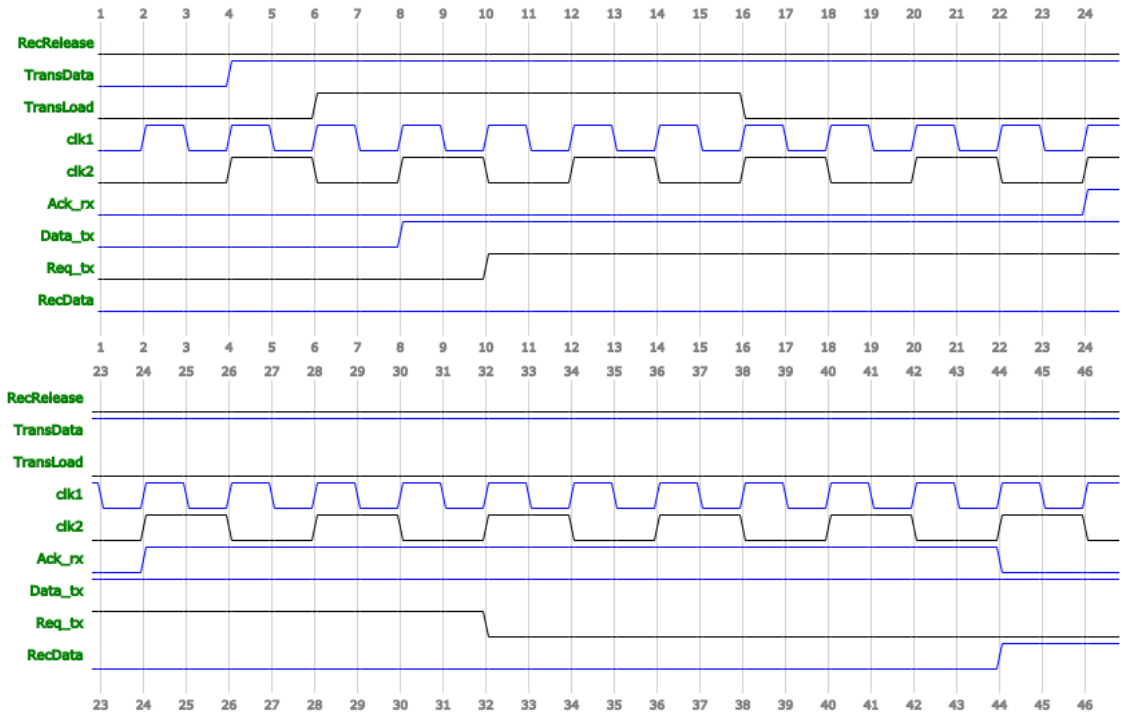

Figure 5.49 - IOPT-Flow wave graph with proposed execution semantics, simulation two.

5.4 - Results Analysis

Some results, related to execution time, may be extracted from the previous simulations, these results are presented in the following tables and the values are expressed in number of steps/clocks.

Table 1 presents information regarding the transmitter components that transmit events, four circumstances are measured, in number of steps, between:

- Load to Req - The number of steps between the Load event and a shift in the Request signal.
- Ack to Req - A shift in the Acknowledge signal and a shift in the Request signal, this circumstance doesn't occur in components that employ the 2-phase protocol.
- Ack to next Req - A shift in the Acknowledge signal and the shift in the Request signal that initiates a new communication cycle, this is only applicable when there is information to be read in the buffer.
- Ack to Available - A shift in the Acknowledge signal and the rise of the Available signal.

Transmitter		Load to Req	Ack to Req	Ack to next Req	Ack to Available
4-Phase	Simple	1	1	-	1
	SimpleBuffer	2	1	2	2
	BurstBuffer	2	1	2	2
2-Phase	Simple	1	-	-	1
	SimpleBuffer	2	-	2	2
	BurstBuffer	2	-	2	2

Table 5.1 - Transmitter components, event transmission.

It is immediately noticeable that the Simple interface saves a step when compared to the other interfaces, since these spend a step, traversing the buffer. Additionally, the SimpleBuffer and the BurstBuffer interfaces require the same amount of steps; however, the latter may transmit all its events at once due to its additional data signal. Finally, the 2-phase components require less events to complete a full communication cycle at the expense of additional elements.

Table 2 presents information regarding the transmitter components that transmit data, five circumstances are measured, in number of steps, between:

- Load to Req - The Load event and a shift in the Request signal.
- Ack to Req - A shift in the Acknowledge signal and a shift in the Request signal, this circumstance doesn't occur in components that employ the 2-phase protocol.
- Ack to next Req - A shift in the Acknowledge signal and the shift in the Request signal that initiates a new communication cycle, this is only applicable when there is information to be read in the buffer.
- Ack to Available - A shift in the Acknowledge signal and the rise of the Available signal.
- Load to DataWire - A Load event and a value being set in the DataWire signal, some components measure 0 steps because the output signal is not synchronized and it is only dependent of combinational logic which takes less that one step to update.

Transmitter		Load to Req	Ack to Req	Ack to next Req	Ack to Available	Load to DataWire
4-Phase	Data	1	1	-	1	0
	DataBuffer	2	1	2	1	1
2-Phase	Data	1	-	-	1	0
	DataBuffer	2	-	2	1	1

Table 5.2 - Transmitter components, data transmission.

For these components, the major difference is in the presence of the buffer. For the Data interface, data is loaded immediately, between steps, since it is held by operations, this is, combinational logic.

Table 3 presents information regarding the receiver components that receive events, three circumstances are measured, in number of steps, between:

- Req to Ack - A shift in the Request signal and a shift in the Acknowledge signal.
- Req to NewData - A shift in the Request signal and the rise of the NewData signal, it is considered that NewData is empty. Additionally, some components have an arrow, this represents whether the information is received in either the rising or falling edge of the Request signal.
- Release to NewData - The Release event and the rise of the NewData signal, this is only applicable to components that employ a buffer and said buffer has more information to be retrieved.

Receiver		Req to Ack	Req to NewData	Release to NewData
4-Phase	Simple	1	1 (↑)	-
	SimpleBuffer	1	2 (↑)	2
	BurstBuffer	1	1 (↓)	2
2-Phase	Simple	2	2	-
	SimpleBuffer	2	3	2
	BurstBuffer	3	3	2

Table 5.3 - Receiver components, event transmission.

Regarding the Receiver components, there is a bigger disparity between amount of steps required for the 4-phase and the 2-phase components. This disparity is due to the method by which transitions in signals are read. The 4-phase BurstBuffer component seems to have a quicker response to the Request signal, yet NewData reacts to the signal's falling edge, contrary to the other interfaces that react to the signal's rising edge.

Table 4 presents information regarding the receiver components that receive data, four circumstances are measured, in number of steps, between:

- Req to Ack - A shift in the Request signal and a shift in the Acknowledge signal
- Req to NewData - A shift in the Request signal and the rise of the NewData signal, it is considered that NewData is empty.
- Release to NewData - The Release event and the rise of NewData signal this is only applicable to components that employ a buffer and said buffer has more information to be retrieved.
- Req to OutData - A shift in the Request signal and a value being set in OutData, some components measure 0 steps because the output signal is not synchronized and it is only dependent of combinational logic which takes less that one step to update.

Receiver		Req to Ack	Req to NewData	Release to NewData	Req to OutData
4-Phase	Data	1	1	-	0
	DataBuffer	1	2	2	1
2-Phase	Data	3	2	-	1
	DataBuffer	3	3	2	2

Table 5.4 - Receiver components, data transmission.

Once again, the amount of steps required for the 4-phase and 2-phase protocol varies, again, due to the method by which signal transitions are read.

6

Chapter Six - Conclusion and Future Work

During this work, an analysis of the current state of Globally-Asynchronous Locally-Synchronous (GALS) systems was made. A GALS system is composed of several synchronous blocks that can interact with one another through an asynchronous environment, as a result, the design is modular with the possibility to fine tune each block to achieve higher power efficiency and lower electromagnetic frequencies. Additionally, a modular design also implies that the synchronous blocks might be easily reused or replaced. However, this modular design comes at the expense of some latency, as a consequence of asynchronous communication.

In terms of architectures for GALS systems there are two that stand out, first, the asynchronous wrapper that encases the synchronous block, which resolves any synchronization issues with neighbour blocks. And, secondly, the FIFO-based scheme where communication is done indirectly through a FIFO buffer, which resolves any synchronization issue.

The DS-Pnet modelling language and its tool framework, IOPT-Flow, emerge to accelerate the development of cyber-physical systems. They bring the advantages of model-based development to bolster a tool chain that encompasses a graphical editor, a simulator/debugger and automatic code generators.

This work proposes a set of asynchronous wrappers that in this work are referred to as asynchronous components. A library of these asynchronous components was developed using DS-Pnets and the IOPT-Flow framework and is

available at http://gres.uninova.pt/iopt-flow/. These components form interfaces that support asynchronous interaction between synchronous blocks in a GALS environment.

Five categories of asynchronous interfaces may be distinguished: the Simple interface is the most plain and it is merely capable of transmitting events; the SimpleBuffer interface introduces a buffer, that is capable of holding a number of events that may, at a later moment, be either transmitted or retrieved; the BurstBuffer interface is similar to the SimpleBuffer interface, but instead of transmitting one event at a time, it transmits events through a data channel; the Data interface is similar to the Simple interface, but allows for the transmission of data instead of events; the Databuffer interface introduces a buffer of size three to the Data interface. Additionally, for each interface two communication protocols were implemented, the 2-phase and the 4-phase communication protocols.

Each asynchronous interface was modelled in the IOPT-Flow framework and then, it was generated the VHDL code, which was used in Xilinx ISE project. Each component is simulated in both the IOPT-Flow and the Xilinx ISE and a comparison is presented. Then, other ten example models, utilizing the components, were implemented on a Nexys 4 DDR board, were a different synchronizing signals were assigned to each of the components. These synchronizing signals were generated with MMCMs and PLLs present in the FPGA, through the Clocking Wizard IP.

Although DS-Pnets support the development of synchronous components, they do not completely support GALS systems. In this work DS-Pnets were extended to support these systems. It is proposed a definition for a GALS-DS-Pnet model. This is a top-level model that is created on top of the current DS-Pnet model, enabling a GALS design. Then, the current execution semantics from the DS-Pnets formalism and its implementation in both the simulator (JavaScript) and hardware code (VHDL) are discussed. Next, suggestions are made for the IOPT-Flow's tools namely, its editor, simulator and VHDL code generator. Afterwards, two models containing the asynchronous components proposed, in chapter three, were simulated in a test version of the IOPT-Flow framework which employed the proposed execution semantics.

From the simulations and tests, it was possible to conclude that 2-phase components will generally be faster, since they require half the communication signals; however, 4-phase components will generally be more compact since reading signals is easier than reading transitions of said signals. When comparing the different interfaces, it can be concluded that the Simple interface is the most compact, adding a buffer provides flexibility to the system at the cost of space and communicating the buffered events through a data signal will rise the efficiency at the cost of said extra signal.

The proposed execution semantics were successfully implemented in the IOPT-Flow simulator, and were capable of fully simulating GALS systems. Additionally, multiple GALS systems were successfully implemented in an FPGA board.

From this work resulted two papers. They were submitted and accepted, respectively for YEF-ECE 2019 - 3rd International Young Engineers Forum on Electrical and Computer Engineering, titled "A survey of IOPT-Flow for GALS systems development", which has already been presented and published, and for IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society, titled "Asynchronous interfaces for IOPT-Flow to support GALS systems", which is scheduled to be presented in October. Both papers had the following list of authors, "João Almeida, Filipe Moutinho and Rogerio Campos-Rebelo".

Lastly, in terms of future work, the proposed extensions should be implemented, that is to say, that it should be possible to associate both synchronizing clocks and synchronizing edges to components in the graphical editor, automatically generate JavaScript code that supports GALS and automatically generating VHDL code with the assigned synchronizing clocks and synchronizing edges.

Bibliography

[1] Jain, R. C., Dinesh Padole, Madhuri B. Kulkarni, Abhijeet and Amit Singhal. “Design of Globally Asynchronous Locally Synchronous (GALS) System using FPGA.”, International Journal Of Enhanced Research In Science Technology & Engineering, (2014).

[2] A. Iyer and D. Marculescu, "Power and performance evaluation of globally asynchronous locally synchronous processors," Proceedings 29th Annual International Symposium on Computer Architecture, Anchorage, AK, USA, 2002, pp. 158-168.

[3] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber and W. Fichtner, "Globally-asynchronous locally-synchronous architectures to simplify the design of on-chip systems," Twelfth Annual IEEE International ASIC/SOC Conference (Cat. No.99TH8454), Washington, DC, USA, 1999, pp. 317-321.

[4] Gürkaynak, Frank K., Stephan Oetiker, Norbert Felber, Hubert Kaeslin, and Wolfgang Fichtner. "Is there hope for GALS in the future?." In 4th ACiD Workshop of the European commission’s fifth framework programme. 2004.

[5] M. Kishinevsky, S. K. Shukla and K. S. Stevens, "Guest Editors' Introduction: GALS Design and Validation," in IEEE Design & Test of Computers, vol. 24, no. 5, pp. 414-416, Sept.-Oct. 2007.

[6] P. Teehan, M. Greenstreet and G. Lemieux, "A Survey and Taxonomy of GALS Design Styles," in IEEE Design & Test of Computers, vol. 24, no. 5, pp. 418-428, Sept.-Oct. 2007.

[7] Syed Suhaib, Deepak Mathaikutty, Sandeep Shukla, "Dataflow Architectures for GALS", Electronic Notes in Theoretical Computer Science, Volume 200, Issue 1, 2008, Pages 33-50.

[8] Ramesh S., Sonalkar S., D'silva V., Chandra R. N., Vijayalakshmi B. (2004) A Toolset for Modelling and Verification of GALS Systems. In: Alur R., Peled D.A. (eds) Computer Aided Verification. CAV 2004. Lecture Notes in Computer Science, vol 3114. Springer, Berlin, Heidelberg

[9] Pereira, F.J.G. (2017) "The DS-Pnet modeling formalism for cyber-physical system development", Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, http://hdl.handle.net/10362/27876.

[10] Pereira, F. and Gomes, L. (2016) " Combining Data-Flows and Petri Nets for Cyber-Physical Systems Specification", in: Camarinha-Matos L.M., Falcão A.J., Vafaei N., Najdi S. (eds) Technological Innovation for Cyber-Physical Systems. DoCEIS 2016. IFIP Advances in Information and Communication Technology, vol 470. Springer, Cham.

[11] F. Pereira and L. Gomes, "The IOPT-Flow framework pairing Petri nets and data-flows for embedded controller development," IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, Florence, 2016, pp. 4832-4837.

[12] Cuesta García, Luís Miguel ; Gil Padilla, António J. ; Remiro Dominguez, Fernando ; Ruivo, Tomás ; Jerónimo, Alberto - Electrónica digital : álgebra de Boole : circuitos combinacionais e sequenciais : automatismos : memórias. Lisboa...[et al.] : Mcgraw-Hill, cop.1994. VII, [2], 44l p.

[13] C. H. Van Berkel, M. B. Josephs and S. M. Nowick, "Applications of asynchronous circuits," in Proceedings of the IEEE, vol. 87, no. 2, pp. 223-233, Feb. 1999.

[14] Spars, Jens, and Steve Furber. Principles asynchronous circuit design. Kluwer Academic Publishers, 2002.

[15] S. Dasgupta and A. Yakovlev, "Modeling And Performance Analysis of GALS architectures," 2006 International Symposium on System-on-Chip, Tampere, 2006, pp. 1-4.

[16] K. Y. Yun and R. P. Donohue, "Pausible clocking: a first step toward heterogeneous systems," Proceedings International Conference on Computer Design. VLSI in Computers and Processors, Austin, TX, USA, 1996, pp. 118-123.

[17] Junbok You, Yang Xu, Hosuk Han, Kenneth S. Stevens, "Performance Evaluation of Elastic GALS Interfaces and Network Fabric", Electronic Notes in Theoretical Computer Science, Volume 200, Issue 1, 2008, Pages 17-32.

[18] Messerschmitt, David. (1990). Synchronization in digital system design. Selected Areas in Communications, IEEE Journal on. 8. 1404 - 1419. 10.1109/49.62819.

[19] J. Ax, N. Kucza, M. Vohrmann, T. Jungeblut, M. Porrmann and U. Rückert, "Comparing Synchronous, Mesochronous and Asynchronous NoCs for GALS Based MPSoCs," 2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), Seoul, 2017, pp. 45-51.

[20] M. Fattah, A. Manian, A. Rahimi and S. Mohammadi, "A High Throughput Low Power FIFO Used for GALS NoC Buffers," 2010 IEEE Computer Society Annual Symposium on VLSI, Lixouri, Kefalonia, 2010, pp. 333-338.

[21] S. Suhaib, B. A. Jose, S. K. Shukla and D. A. Mathaikutty, "Formal transformation of a KPN specification to a GALS implementation," 2008 Forum on Specification, Verification and Design Languages, Stuttgart, 2008, pp. 84-89.

[22] O. K. Chinedu, E. C. Genevera and O. O. Akinyele, "Hardware description language (HDL): An efficient approach to device independent designs for VLSI market segments," 3rd IEEE International Conference on Adaptive Science and Technology (ICAST 2011), Abuja, 2011, pp. 262-267.

[23] Taylor, Adam, " 10 Ways To Program Your FPGA", EETimes (Internet article), November's 6th, 2016, https://www.eetimes.com/document.asp?doc_id=%201329857&page_number=1.

[24] R. Nane et al., "A Survey and Evaluation of FPGA High-Level Synthesis Tools," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp. 1591-1604, Oct. 2016.

[25] Meeus, W., Van Beeck, K., Goedemé, T. et al. Des Autom Embed Syst, Design Automation for Embedded Systems (2012) 16: 31. https://doi.org/10.1007/s10617-012-9096-8

[26] Bailey, Donald. (2015). The advantages and limitations of high level synthesis for FPGA based image processing. Proceedings of the 9th International Conference on Distributed Smart Cameras, 134-139. 10.1145/2789116.2789145.

[27] Avinash Malik, Zoran Salcic, Partha S. Roop, Alain Girault, SystemJ: A GALS language for system level design, Computer Languages, Systems & Structures, Volume 36, Issue 4, 2010, Pages 317-344.

[28] A. Malik, A. Girault and Z. Salcic, "A GALS Language for Dynamic Distributed and Reactive Programs," 2011 Eleventh International Conference on Application of Concurrency to System Design, Newcastle Upon Tyne, 2011, pp. 173-182.

[29] A. Gamatie and T. Gautier, "The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded Systems," in IEEE Transactions on Parallel and Distributed Systems, vol. 21, no. 5, pp. 641-657, May 2010.

[30] L. Gomes, F. Moutinho, F. Pereira, J. Ribeiro, A. Costa and J. Barros, "Extending input-output place-transition Petri nets for distributed controller systems development," 2014 International Conference on Mechatronics and Control (ICMC), Jinzhou, 2014, pp. 1099-1104.

[31] L. Gomes, F. Moutinho and F. Pereira, "IOPT-tools — A Web based tool framework for embedded systems controller development using Petri nets," 2013 23rd International Conference on Field Programmable Logic and Applications, Porto, 2013, pp. 1-1.

[32] F. Moutinho and L. Gomes, "Asynchronous-Channels Within Petri Net-Based GALS Distributed Embedded Systems Modeling," in *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2024-2033, Nov. 2014.

[33] F. Moutinho, L. Gomes, A. Costa and J. Pimenta, "Asynchronous wrappers configuration within GALS systems specified by Petri nets," *2012 IEEE International Symposium on Industrial Electronics*, Hangzhou, 2012, pp. 1357-1362.

[34] R. Jipa, "Dedicated solution for local clock programing in GALS designs," 2008 International Semiconductor Conference, Sinaia, 2008, pp. 393-396.

[35] R. Gagne, J. Belzile and C. Thibeault, "Asynchronous component implementation methodology for GALS design in FPGAs," 2009 Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference, Toulouse, 2009, pp. 1-4.

[36] A. Peeters and K. van Berkel, "Single-rail handshake circuits," Proceedings Second Working Conference on Asynchronous Design Methodologies, London, UK, 1995, pp. 53-62.

[37] Digilent, Nexys 4 DDR Referemce Manual, reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual, accessed 18/12/2019.