UNIVERSIDADE NOVA DE LISBOA

FACULDADE DE CIÊNCIAS E TECNOLOGIAS

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA

Petri Nets Based Components

Within

Globally Asynchronous Locally Synchronous systems

Por

Henrique Afonso Ferreira

Dissertação apresentada na Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa para a obtenção do grau de Mestre em Engenharia Electrotécnica e Computadores

Orientador: Prof. Doutor Luís Gomes

Lisboa

2010

SUMÁRIO

O objectivo principal do trabalho é o de desenvolver uma solução para a interligação de componentes constituintes de um sistema GALS – Globally Asynchronous, Locally Synchronous.

Os componentes referidos são executados em paralelo obtidos como resultado da partição de um modelo expresso em redes de Petri (RdP), realizada utilizando o editor de RdP SNOOPY-IOPT em conjugação com a ferramenta SPLIT e com a ferramenta de geração automática de código VHDL a partir das representações PNML dos modelos RdP resultantes da partição (as ferramentas referidas foram desenvolvidas no âmbito do projecto FORDESIGN e encontram-se disponíveis em http://www.uninova.pt/FORDESIGN).

Serão analisadas soluções típicas disponíveis para garantir a correcta comunicação entre componentes do sistema GALS, bem como caracterizada e desenvolvida uma solução adequada para a interligação dos componentes associados aos sub-modelos RdP. O objectivo final (não atingido com esta dissertação) é o de obter uma ferramenta que permita gerar o código da solução de interligação partindo dos componentes associados considerando uma aplicação específica.

A solução proposta para a interligação de componentes foi codificada em VHDL e as plataformas de implementação utilizadas para teste incluem as FPGAs da Xilinx das famílias Spartan-3 e Virtex-II.

ABSTRACT

The main goal is to develop a solution for the interconnection of components constituent of a GALS - Globally Asynchronous, Locally Synchronous – system.

The components are implemented in parallel obtained as a result of the partition of a model expressed a Petri net (PN), performed using the PNs editor SNOOPY-IOPT in conjunction with the Split tool and the tools to automatically generate the VHDL code from the representations of the PNML models resulting from the partition (these tools were developed under the project FORDESIGN and are available at http://www.uninova.pt/FORDESIGN).

Typical solutions will be analyzed to ensure proper communication between components of the GALS system, as well as characterized and developed an appropriate solution for the interconnection of the components associated with the PN sub-models. The final goal (not attained with this thesis) would be to acquire a tool that allows generation of code for the interconnection solution from the associated components, considering a specific application.

The solution proposed for componentes interconnection was coded in VHDL and the implementation platforms used for testing include the Xilinx FPGA Spartan-3 and Virtex-II.

ACRONYMS

CAD - Computer Aided Design

FIFO – First In First Out

FPGA – Field Programmable Gate Array

GALS - Globally Asynchronous, Locally Synchronous

IOPT – Input Output Place Transition

IP- Intellectual Propriety

LC- Latch Controller

LSI - Locally Synchronous Island

PNML – Petri Net Markup Language

PCC - Pausible Clock Controller

SoC- System on Chip

V_{DD} - positive supply voltage

VHDL – Very High Speed Integration Circuits Hardware Descriptive Language

VLSI – Very Large Scale Integration

INDEX

LIST OF FIGURES

1. INTRODUCTION

1.1. CURRENT SITUATION ON DISTRIBUTED DIGITAL CONTROL CIRCUITS

Currently digital circuits development face several challenges. Improvements in semi-conductor technology lead to continued decreases in size and increase the number of devices that can fit in a single die. It is more and more common that modern digital system is implemented as a System on a Chip (SoCs) or on a reconfigurable SoC (having FPGA and memories as supporting platform). Consequently the design of the chip is decomposed in blocks, often featuring multiple clock domains, and many times running at different frequencies.

Practically, single-clocked digital systems are mostly outdated.

These days the designer as to address three main conjectures.

First, the constrain to implement a global-clock network that can control all the blocks in a chip introduces a greater, undesired, power consumption and a very strong noise for the analog part of the circuit (if any).

Second, to cope with market dispute, shorter times for design leads to the reusing of IP blocks. Such IP blocks are designed and optimized for different clock speeds by their design groups.

Third, inherently each design, for the duty in which it is projected, requires a wide-ranging of clock frequencies.

To deal with such constrains there is, in the industry today, an increasing popular solution which is the Globally Asynchronous Locally Synchronous (GALS) system.

GALS was first introduced by Chapiro in his doctoral dissertation [1] , in which he developed a solution based on pausible-clock circuit. Since then numerous solutions sprung, even thought the GALS design approach has not been widely adopted in the industry. The lack of verification techniques and testing methodologies for asynchronous design, as well the synchronic assumption in the digital circuit industry has proven to be cumbersome for the paradigm shift necessary for the broad acceptance of the GALS solution.

On the other hand, GALS design offers an increased ease of use in functional block reuse, simplified timing closure, and power advantages due to heterogeneous clocking by providing wrapper circuits to handle interblock communication across the clock domain boundaries. Also GALS allows for a smooth transition between the synchronous to asynchronous paradigm due to the elasticity to integrate together in a system synchronous and asynchronous components without worrying with their internal structure.

1.2. OBJECTIVES

The main objective of this work is to study the existing GALS systems and find a viable solution for implementation of interconnections of components obtained through the partition of Petri nets models. At the same time, identify the advantages and disadvantages of each of the known solutions given a global overview of this type of solution.

From the palette of solutions studied and after weighting the advantages and disadvantages of each, one particular solution is selected for implementation.

Because they are general solutions, tuning of a particular one will be implemented to best fit our needs and achieve the goal.

As referred, the solution will be validated in the interconnection of Petri nets sub-models using several application examples from the domain of automation systems (emphasizing the control part of the system).

These examples will be created with the support of some design tools developed inside the FORDESIGN Project:

- The SNOOPY-IOPT editor [2], which is a Graphical Petri net[3-5] Editor for the Input-Output Place-Transition (IOPT) class [6], supporting hierarchical and modular specifications. PNML representation is the preferred output format to assure interoperability among tools.
- The Split tool [7] implements a net splitting operation able to decompose a Petri net model into Petri net sub-models using synchronous communication channels. The generated sub-models are associated with components to be executed concurrently, allowing a distributed execution of the initial model.
- PNML2VHDL tool [8] automatic generates VHDL execution code from a IOPT Petri net model stored in a PNML-compliant file.

With the support of these three tools it is possible to start with the Petri net model of the system and ending with the implementation code, allowing then distributed execution.

So the main goal of the work developed within this dissertation is to be able to generate a solution allowing robust communication among the components in parallel execution, independently of the clock used by each component.

The validation of the results will be achieved through simulation as well as through the real implementation using FPGA devices.

1.3. STRUCTURE OF THE DOCUMENT

This document is organized in following way:

Chapter 2 will present the theoretical foundations as well make an overview of the circuit architectures and basic components to assemble a GALS system.

Chapter 3 will discuss the solution adopted in detail.

Chapter 4 will present an adaption of the solution presented in chapter 3 to a solution that best fit our needs. The introduction of this chapter was with the objective to make a clear distinction from the general solution to the new one, which is adapted to interconnect Petri nets based components.

Chapter 5 introduces application examples to be solved with our devised solution.

Chapter 6 will unfold conclusions and present a closure for this Master Thesis.

2. THEORETICAL FUNDAMENTS

This chapter will address two main issues of this dissertation, in other words everything needed to achieve the final solution.

So, first we will address the tools developed in the scope of the FORDESIGN [6] project and how they will be integrated accordingly to the needs of this thesis.

After that, an overview of the circuit's architecture and of the basic components to assemble a GALS system will be discussed. This includes a discussion of the three main GALS designs.

2.1. PETRI NET BASED COMPONENTS

As referred, the objective of this thesis is to develop a solution for the interconnection of components constituent of a GALS.

The components are executed in parallel obtained as a result of the partition of a model expressed as Petri nets (PN), performed using the PNs editor SNOOPY-IOPT in conjunction with the Split tool and the tools to automatically generate the VHDL code from the representations of the PNML models resulting from the partition.

Figure 1 gives an overview of the work flow and the tools used to assemble the Petri net components. Not all of the FORDESIGN project tools are referred or used here.

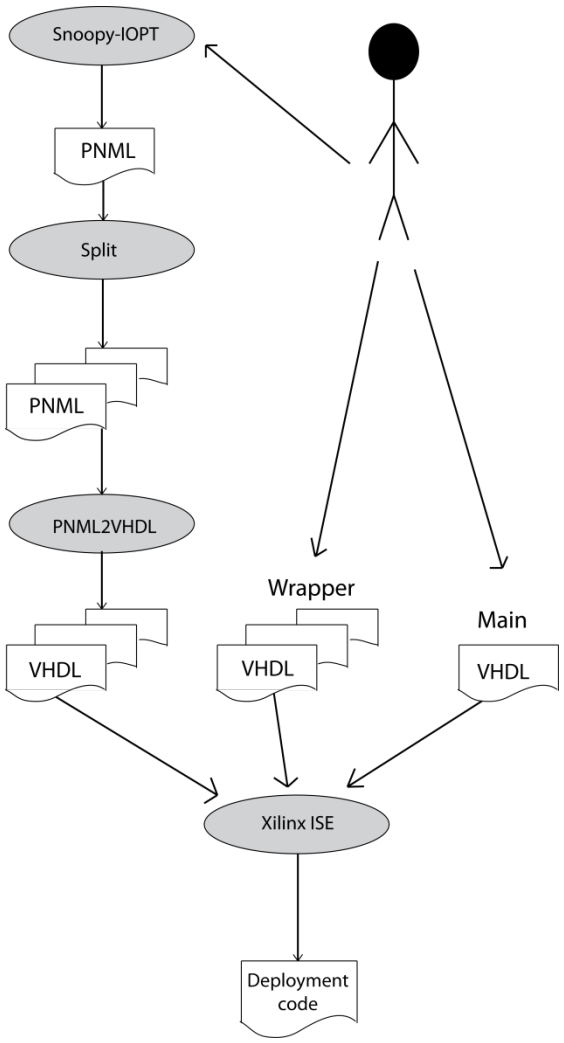


FIGURE 1 – DEVELOPMENT FLOW AND SUPPORTING TOOLS

The gray ovals from Figure 1 represent tools, the rectangles specify files and the arrows stand for information flow.

The editor generates Petri Net models in PNML. Allowing structuring mechanisms like decomposition/composition and refinement/abstraction and the editing of the Input-Output Place-Transition (IOPT) Petri Net Class models, targeted for the modeling of automation systems and embedded systems, the Petri Net Markup Language representation is extensively used in conjunction with the FORDESIGN tools.

After obtaining the PNML file the Split tool implements a net splitting operation able to decompose a Petri net model into Petri net sub-models using synchronous

communication channels. The generated sub-models are associated with components to be executed concurrently, allowing a distributed execution of the initial model.

The net splitting operation is based on the definition of a valid cutting set and specific rules. To define the cutting set the user identifies a set of components to be executed concurrently. The initial model and the cutting set are represented using PNML notation. These can be produced using the graphical editor for Input-Ouput Place-Transition Petri Net Class (SnoopyIOPT editor) or using the line command "split initialModel.pnml cuttingSet.pnml result.txt".

After the validation of the cutting set there are three rules that allow the generation of components, interconnected through synchronous communication channels. At the implementation level, each component can be seen as an autonomous model, however having information about the state of adjacent components.

Here, is the basis of this dissertation. Once the original model is split and each of the sub-models are executed separately, the only way for the model maintaining coherence (preserving execution semantics) is through the usage of synchronizing events. For example: if a place is marked by a token in a sub-model, and this place is common to another sub-model then and synchronization event signal has to be sent to that sub-model, from the sub-model where the place was firstly marked by a token.

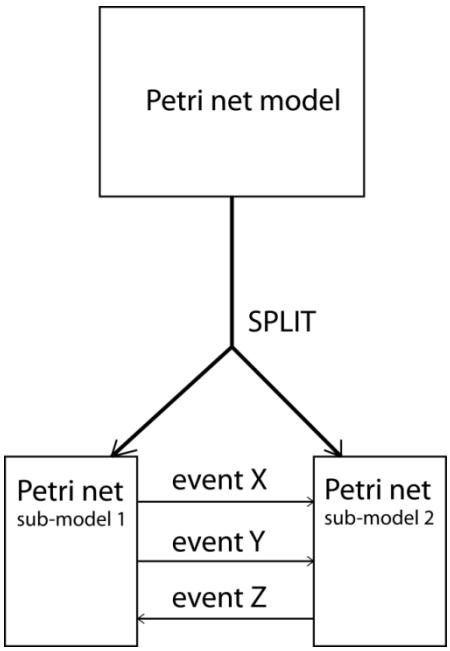


FIGURE 2 – SPLITTING THE ORIGINAL MODEL IN TO SUB-MODELS

Considering that each of the sub-models may run in different time domains then there is a need for each event to be timely synchronized between associated modules. Solving this step is the fundamental question for this work. The way it is going to be accomplished is by introducing wrappers between the events signals (sender and receiver) to achieve time domain synchronization. This will make the system as a whole GALS system. Figure 3 shows the introduction of the wrappers between the event signals.

From the usage of the last FORDESIGN tool (the split tool) we get several PNML files, one per sub-model, allowing distributed execution of the initial model. The next step is to convert these file to VHDL. This is done by applying the PNML2VHDL tool. It automatically generates VHDL execution code from a IOPT Petri net model stored in a PNML-compliant file. The generated VHDL code can be directly deployed on programmable logic devices such as FPGAs, CPLDs, as well for System-on-Chip (SoC) solutions.

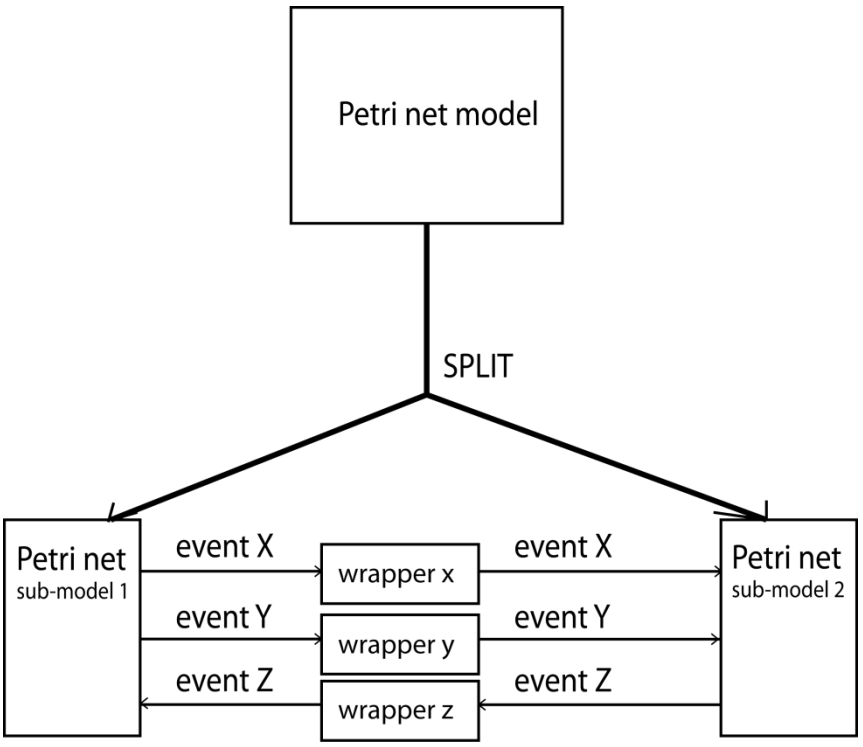


FIGURE 3- SPLITTING THE ORIGINAL MODEL IN TO SUB-MODELS WITH WRAPPERS

Hence, we finally arrive to the bottom of the flow of the figure, where the GALS solution is merged with the synchronous sub-models, using Xilinx ISE, to allow synchronization between the different time domains of the sub-models. Obtaining, in this way, Petri net based components within a GALS system.

2.2. CIRCUIT ARCHITECTURE

The following sub-section present an overview of the circuit's architecture and of the basic components to assemble a GALS system will be discussed.

2.2.1. SYNCHRONOUS SYSTEM

A synchronous circuit is a digital circuit where all of its parts are synchronized with a clock signal. A system without a global clock is called asynchronous, which will be discussed next.

The internal state changes only upon occurrence of a positive (or negative) rising edge of the clock. This leads that all operations in the system need to be carried and completed between two clock pulses. If this criterion is satisfied the system is considered reliable and all circuit behavior can be predicted with accuracy. Each logical operation introduces a delay in the system, which in practice constrains the maximum speed at which the synchronous system can run.

This type of system incurs into two main disadvantages:

- The clock signal as to be distributed to all circuit flip-flops simultaneously. The clock signal is a high frequency signal that potentially consumes great amounts of energy. Even if there is no flip-flops transition (the output remains the same); they still consume some energy, contributing for unnecessary energy expenses and heat accumulation with no apparent reason.
- The maximum clock speed is limited to the longest path in the circuit. This means that both the more complex operation, as the more simple, has to be executed in one clock cycle.

Coming back to the globally synchronous SoC constituted with several IP modules the problem at hand is that each of the modules is projected for determined clock speed, unrelated their reutilization with a multitude of other modules. Finding a clock mode that can feed globally each of the IP core reveals itself as daunting task.

Another problem of this solution is the non guarantee of the same arrival time of the clock signal at all components in the circuit due to delays in wires. If in the past the limiting factor to circuits were the transistors, at present the delay in propagations consumes a larger part of the clock period.

On the other hand, this type of circuits are the most accepted in the industry, given the fact of being utterly studied and have a large support from CAD (Computer Aided Design) tools.

2.2.2. ASYNCHRONOUS SYSTEM

Asynchronous systems are not governed by a local or global clock signal. The circuits use handshaking signals between their components in order to achieve the necessary synchronization [9].This difference gives advantages in each of these subjects:

- Low power consumption, due to fine-grain clocking and zero standby power consumption.
- High operation speed, has it depends in actual local latencies rather than global worst-case latency;
- Less emission of electromagnetic noise. An asynchronous pipeline computes slightly out of phase with the previous one, thus uniformly spreading power consumption over time.
- Robustness towards variations in supply voltage, temperature, and fabrication process parameters.
- No clock distribution and clock skew problems.

The major drawbacks this type of circuit faces are not so much intrinsic to the technology itself but due to an inertia for the community versed in the synchronous interface to shift to a new style. Adding to this, also, comes a lack of developing and testing tools.

2.2.3. SYNCHRONOUS-ASYNCHRONOUS INTERFACE

The basic problem between interactions of different synchronization domains arises from the fundamentally diverse signal transitions. In the asynchronous domain there is not a timing grid coupled to such an event, therefore metastable behavior in peripheral flip-flops of LS (locally Synchronous) modules is most likely to occur. In fact, it will ultimately occur unless some odd random events permits it to be impossible synchronized for a period of time considered to be nominal circuit running time.

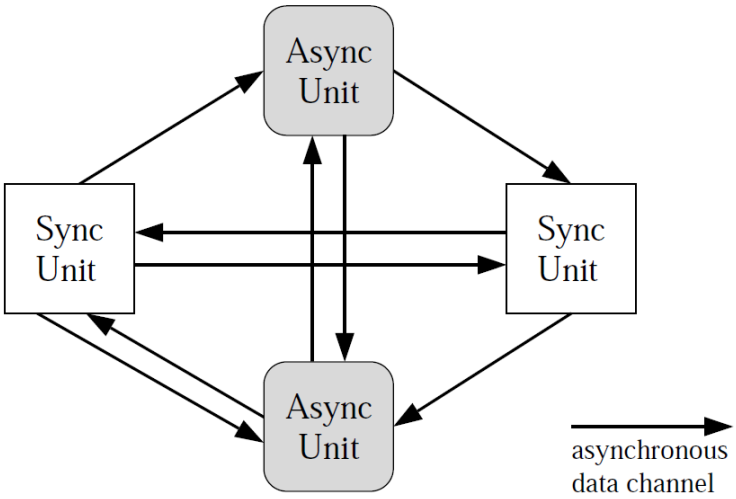


FIGURE 4 – INTERACTION BETWEEN ASYNCHRONOUS AND SYNCHRONOUS MODULE. ***RETRIEVED FROM[10]***

The conventional scheme to address such a problem is the extensive use of synchronizers. This includes the double-latching mechanism and some extension like pipeline synchronization [11].

Although reducing the probability of malfunctioning, this sort of methods, do not exclude it. Also they add undesired latency to each communication. This makes, as a whole, a system prone to failure and undesired to cope with the paradigm that is crossing the boundaries between synchronous and asynchronous.

2.3. GALS

GALS proposals started to appear in 1980s with Chapiro[1], but due to the impracticability of the design at the time, only in mid-1990s and early 2000s started to emerge practical proposals, introducing pausible, or stretching, clocking. These early solutions focused to improve throughput, reduce area overhead and power consumption. Some test cases proved some benefits in operation speed, circuit area and power consumption, but on the overall the added overhead of asynchronous wrapper resulted in performance penalties [12].

More recent studies dwelled on facilitate system integration and reducing Electromagnetic Interference (EMI).

In this section a summary of the components most used when assembling a GALS circuit is presented. On the second part we will do a survey on the GALS architectural techniques.

2.3.1. BASIC GALS COMPONENTS

2.3.1.1. ASYNCHRONOUS WRAPPER

The concept for the asynchronous wrapper is to encapsulate the Locally Synchronous Island (LSI) with an external interface to turn it completely asynchronous [10]. Each of data vectors entering or leaving has to be accompanied by a request-acknowledge pair of handshakes signals (bundled data).

In order to fulfill the synchronous circuit timing constrains, the wrapper has to provide a clock signal. A major advantage is that this local clock frequency can be defined to

fit the needs of the particular module and be stretched (or paused) as necessary. The general wrapper scheme can be depicted in the Figure 5:

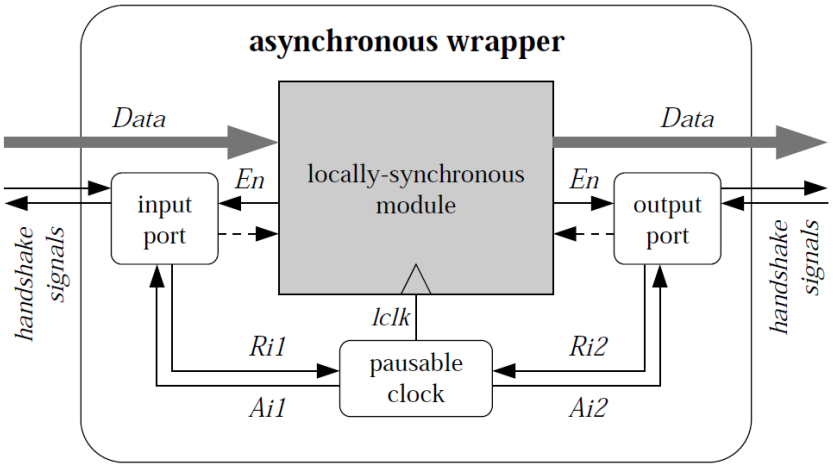


FIGURE 5 – AN ASYNCHRONOUS WRAPPER BASIC SCHEME, *[10]*

2.3.1.2. DATA PORT CONTROLLERS

A globally asynchronous circuit requires asynchronous communication between their locally synchronous islands. This asynchronous communication can use a two-phase or four-phase handshaking protocol. The ports can act in an active or passive manner [9].

There are two kinds of communication ports controllers for the GALS: the demand controller and the poll controller [13]. In the first, it is assumed that the demanded date is required immediately after the request. So the clock in Locally Synchronous (LS) circuit should be immediately stopped and reactivated when the communication is finished. On the former type, the LS clock is not stopped immediately upon request, permitting the circuit to finish its current operation or make other useful operation. This type of controller aims for gain performance but depends [9] on the type of the circuit and the task at hand, so it cannot be used extensively.

In the Figure 6 below a D-type (Demand-type) controller scheme is illustrated.

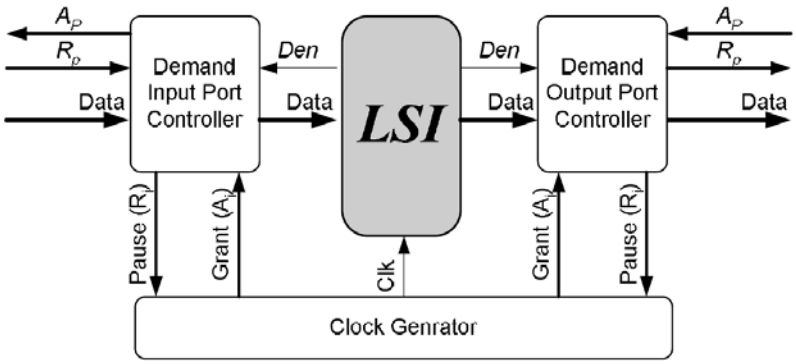


FIGURE 6 – DATA PORT CONTROLLER SCHEME, *RETRIEVED FROM[14]*

A data transfers operation occurs in this fashion:

The LS module toggles Den to indicate it is ready for a new data transfer. Upon receiving that event the D-port actives *Ri*+ (the plus and negative signal refers to when a given signal is at active or inactive, respectively) signal, requesting for a pause in the clock, which in turn receives acknowledgment with *Ai*+. From here the input and output ports have distinct behaviors:

- The Output Port activates *Rp*+ to request a data transfer on the asynchronous channel. Once the other party is ready for the transfer it acknowledges it with *Ap*+. At the completion of the transfer procedure the output port inactivates *Ri* to cancel the clock pause request. The job is considered to be completed upon receiving *Ai*- and *Ap*-, which indicates respectively that the clock is resumed and the data has been successfully latched.
- The Input Port waits to receive *Rp*+ and *Ai*+ to ensure that there is data on the channel and that the request to pause the clock has been successful. It acknowledges the sending party by activating *Ap*+ and latches the received data. The clock is resumed after the completion of the data transfer and the circuit returns to its initial state.

2.3.1.3. PAUSIBLE CLOCK

The term pausible clock was first used by Chapiro[1] in his doctoral dissertation in 1984. He proposed the use of pausible clocks to enable diverse clock domains to communicate avoiding metastability. Each locally synchronous block generates its own clock with a ring oscillator, set according to the requirements of the given block.

The main advantages gained over such clock control are the avoidance of metastability altogether, stretching the clock's sampling edge until the arrival of data from some other domain, which translates in robustness and power. The block awaiting communications does not dissipate dynamic power, has his clock is paused. Also its V_{DD} can be lowered to reduce static power consumption. A feature, that can be, useful for power-critical designs.

Chapiro's assumptions at the time proved impractical in modern design, but provided the foundations for all posterior works.

Later in 1996 Yun et al.[15] invented the pausible clocking control (PCC). The PCC comprises generation of stretchable clocks and processing external handshake. It requires at least two local clock cycles to transfer the data and at most one port per module can be active at the time. The increasing size of the blocks has the inputs and outputs channels raise, render the PPC to small system with circular data flow.

Figure 4 shows a basic clock pausible clock generation scheme.

The main idea of the circuit is to postpone the next positive rising edge of the clock signal until completion of input-output communication actions[16]. Upon receiving a request *Ri*+ from the port controller, the clock suspends the next clock pulse gets delay as long as *Ri*+ is active and acknowledges the requesting port controller by

activating *Ai*. The duration of clock pulses shall not be influenced, therefore only the low phase of the clock pulse shall be stretched upon request.

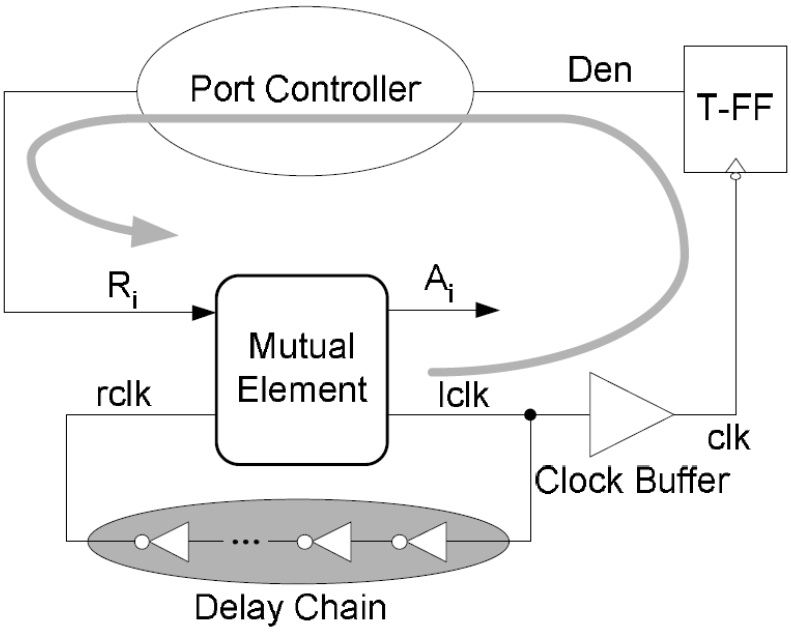


FIGURE 7 – PAUSIBLE CLOCK GENERATION SCHEME

Figure 7 shows an implementation that satisfies these requirements. The ring oscillator consists of an odd number of inverters that generates the clock signal. The Mutual Exclusion (ME) element [17] resolves between possible concurrent events *Ri*+ and *rclk*+. It allows only one of the two inputs to pass at a time. In the situation where two inputs arrive simultaneous it decides randomly which shall pass, but the outputs are guaranteed to remain mutually exclusive.

To ensure the transition on *Ri*+ can stop the next rising of the clock the propagation delay from *lclk* to *Ri*+ has to be smaller than the low phase or *rclk*.

2.3.1.4. ASYNCHRONOUS SYNCHRONIZER

GALS modules can avoid clock arbitration employing a standard asynchronous synchronizer, as depicted in Figure 8.

Considering that a clock cycle is sufficient to ensure metastability resolution, no clock delay verification is assumed, but the data rate is affected drastically because it is simple not possible to transfer data every clock cycle.

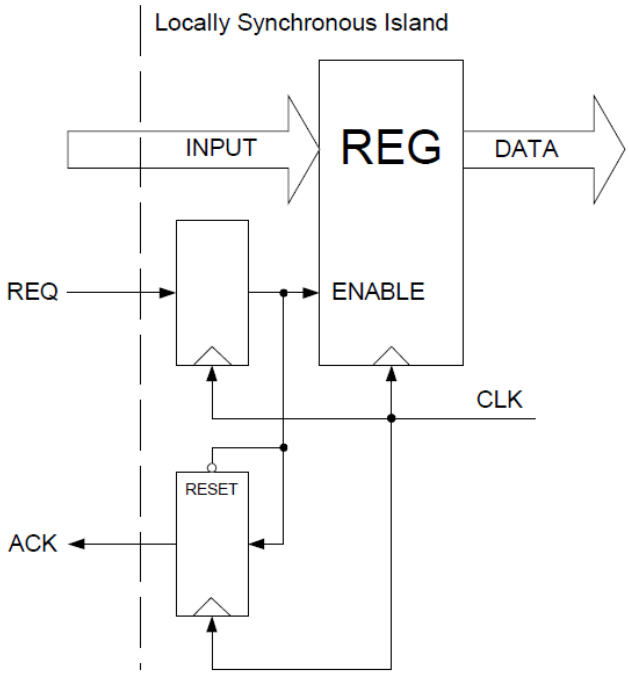


FIGURE 8 - ASYNCHRONOUS SYNCHRONIZER SCHEME

Assuming mesochronous operation (the same clock frequencies between transmitter and receiver) the minimal data cycle time (the time between two Req+) takes seven cycles in worst case (Req+ happens immediately after Clk+ and the transmitter and receiver clocks are in phase) as shown in Figure 9. This data cycle can be reduce if the signals are out of phase (five clock cycles), or by employing a two phase protocol (three cycles).

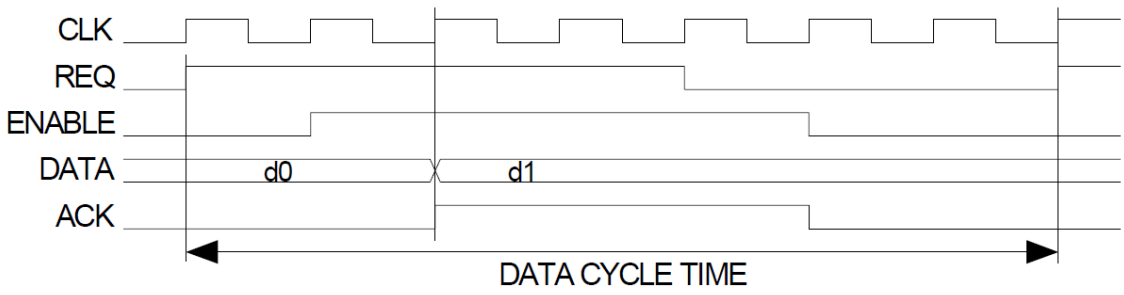


FIGURE 9 – ASYNCHRONOUS SYNCHRONIZER TIMING DIAGRAM

2.4. GALS DESIGNS TYPES

In the work “A Survey and Taxonomy of GALS Design Styles” Teehan et al. came with a description that divided the GALS designs styles into three categories[18].

The *pausible-clock* design style relies on locally generated clocks that can be stretched or paused to prevent metastability or avoid a transmitter or receiver from stalling due to full or empty transmission channel.

The *asynchronous* design, where it is assumed that no timing relationship between the synchronous clocks occurs. Such design have maximum flexibility concerning to timing.

The *loosely synchronous* is used for cases in which there is well know timing relationship between the local clocks. Due to the stability of these clocks one can achieve high efficiency while simultaneous provide tolerance for large amounts of skew inherent in global interconnects. Messerschimitt[19] proposed a taxonomy for these timing relationships:

- *Mesochronous*. The sender and the receiver operate at exactly same frequency but with unknown yet stable phase difference.
- *Plesiochronous.* The sender and receiver operate at the same nominal frequency but may have a slight frequency mismatch, such as a few parts per million, which leads to drifting phase.
- *Heterochronous*. The sender and receiver operate at nominally different frequencies. If it happens that the receiver’s clock frequency is an exact rational multiple of the sender clock frequency, and both are derived from the same clock source, then there is a predictable periodic phase relationship

which is named *Ratiochronous* and is a subset of *Heterochronous* timing relationship. Figure 10 presents the associated taxonomy.

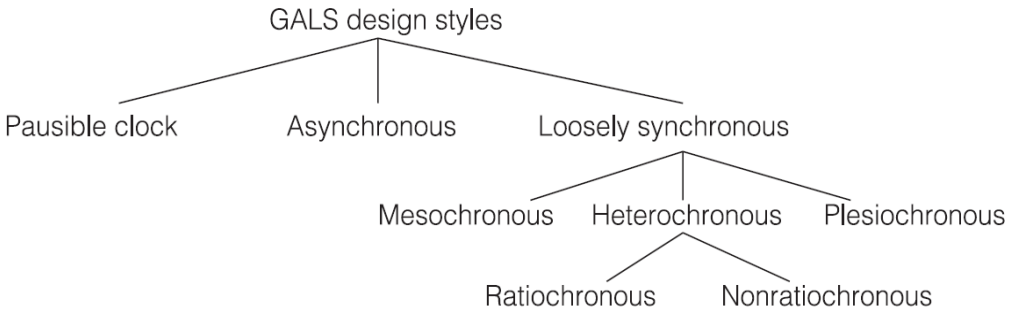


FIGURE 10 – TAXONOMY OF GALS DESIGNS STYLES, *RETRIEVED FROM[18]*

In the following sub-section each design style will be studied in detail.

2.4.1. PAUSIBLE CLOCKS

This is one of the earlier solutions, and has had many proposals in the past few years. The basic idea is to transfer data between the synchronous islands when both data transmitter and data receiver clocks are stopped.

This type of system was first proposed in 2000 [10] and characterized by joining two previous studies into a complete design methodology for GALS. The Pausible Clock Control (PCC) circuits [15] to manage data transfers between independently clocked modules and the Asynchronous Wrapper [13] to provide data flow with greater flexibility and organization.

A more up to date example for pausible clock is shown in Figure 11. Each ring oscillator contains a NAND gate to control clock pausing. The transmitter clock should be allowed to run if it is currently high, if the FIFO buffer can accept a new

value (*ok_to_put* is *high*), or if the transmitter is not attempting to send (*ready_to_put* is low). Likewise, the transmitter clock, should be allowed to run if currently high, if the FIFO buffer has data ready (*ready_to_take* is high), or if the receiver is not attempting to read new data (*ready_to_take* is low).

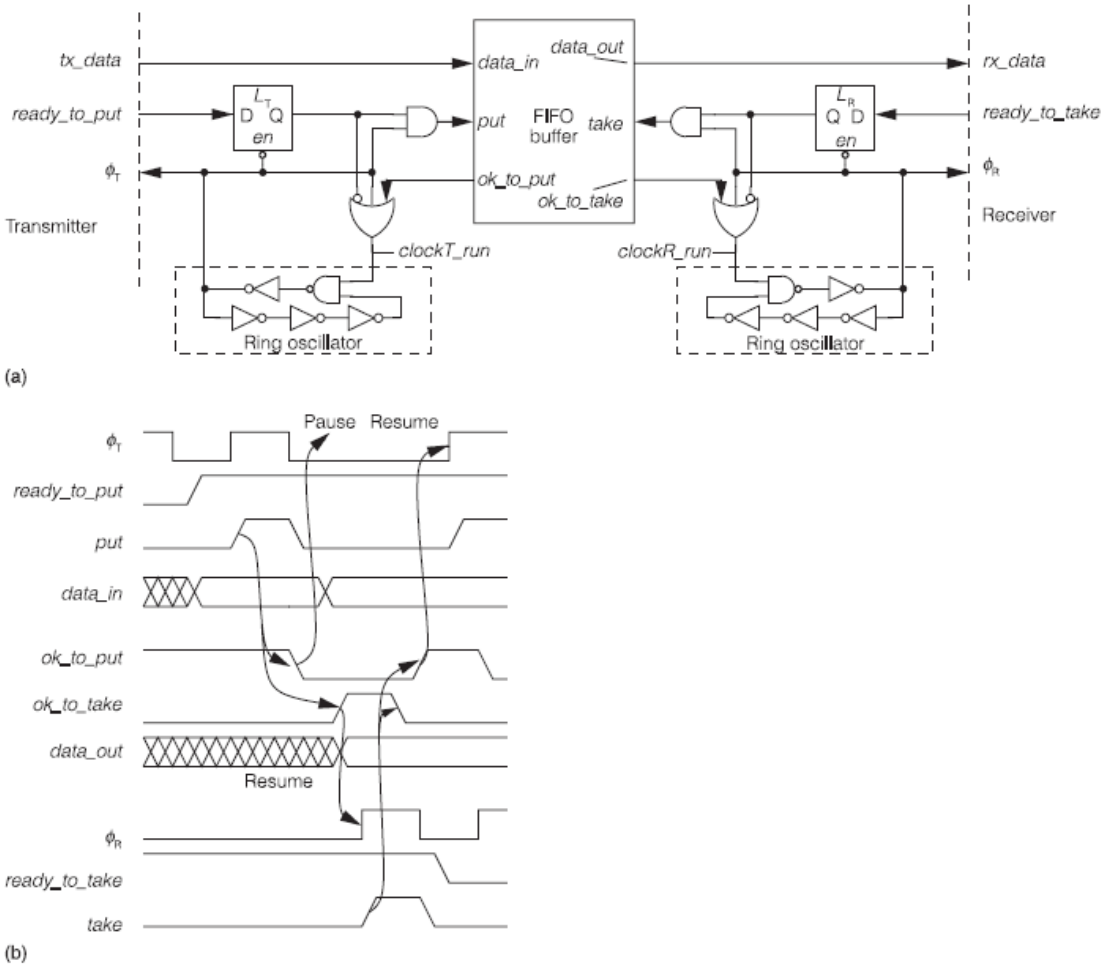


FIGURE 11 – PAUSIBLE-CLOCK GALS DESIGN STYLE CIRCUIT: CIRCUIT (A) AND TIMING DIAGRAM (B), *RETRIEVED FROM[18]*

The timing diagram Figure 11 b shows two consecutive data items being transferred. Assume that the FIFO buffer is initial empty and can hold only one datum. The receiver is ready (*ready_to_take* is high) but the clock is paused because the buffer is empty. The transmitter is ready, *tx_data* is ready and *ready_to_put is high.* While transmitter clock is low the latch is transparent but the AND gate keeps *put* at a low until the FIFO buffer is ready (*ok_to_put* is high), a rising edge of the clock will enable

put and the buffer will fill with the first datum (ok_to_put lowers). From here the transmitter clock pauses because its ready to transmit the second datum (*ready_to_put* high) but the buffer is full. At the receiver end the *ok_to_take* asserted resumes the clock and the data is latched, asserting *take*, which signal that the data has been removed from the FIFO buffer. Because the buffer is no longer full *ok_to_put* goes high, which restarts the transmitter's clock and the process takes place all over again for the second datum transmission.

About design issues, clock tree latency must be considered in GALS design [14]. If it happens to be longer than the latency from the clock distribution, invalid operation can occur after the clock was supposed to be stopped.

Gurkaynak et al. in their study[20] came to the conclusion that designing ring oscillators for robustness and good performance was a major difficulty in their GALS research, concluding that pausible clocking "remains a niche technology at best". The clock period can have high jitter, varying significantly from cycle to cycle as it restarts from pause.

In conclusion, controlling the receiver's clock, this interface ensures that the data arriving at receivers satisfies its timing requirements thus avoiding metastability. Once an interface wrapper has been verified it can be reused successfully for many local blocks without the need for further timing analysis.

Another potential advantage is that variation in the clock period should track variation in logic-gate delays across a range of operation conditions.

2.4.2. ASYNCHRONOUS INTERFACES

The asynchronous interface design style uses circuits known as synchronizers to transfer signals arriving from an outside timing domain to the local timing domain. An example scheme to this kind of interface can be observed in the Figure 12.

The timing diagram shows the transfer of two data values from the transmitter to the receiver, assuming that FIFO the buffer is initial empty. This design uses two flip-flops to synchronize a signal with the local clock and avoid metastability. To account for the synchronizer's delay, the *put_wait* signal prevents the transmitter from sending until the FIFO buffer status following the previous put has propagated through the synchronizer. The *take_wait* signal serves the same function for the receiver. This simplistic design can transfer at most one datum for every three clock cycles of the transmitter or receiver clock, whichever is slower.

Seizovec found a way to increase the throughput of an asynchronous interface by pipelining the synchronization operations trough a FIFO buffer along with the data[11]. This allowed transferring one datum per transmitter or receiver clock cycle, whichever is slower. Later, Boden et al. used this solution in the design of the Myrenet high-speed network hardware[21].

Considering design issues, Asynchronous interface offers the most flexible and probably the easiest integration into existing CAD tool[18]. The main concern is the modeling and validation of the synchronizer circuits and the impact of their delay because real synchronizer have complicated behavior and circuit simulators do not have the numerical accuracy to verify acceptable reliabilitie[22]. Recent simulation methods where developed to address this problem [23, 24].

A conclusion which one can withdraw is the advantage of this interface is that they do not affect the locally synchronous module's operation but the introduction of latency might be significant and unacceptable for high-speed operation.

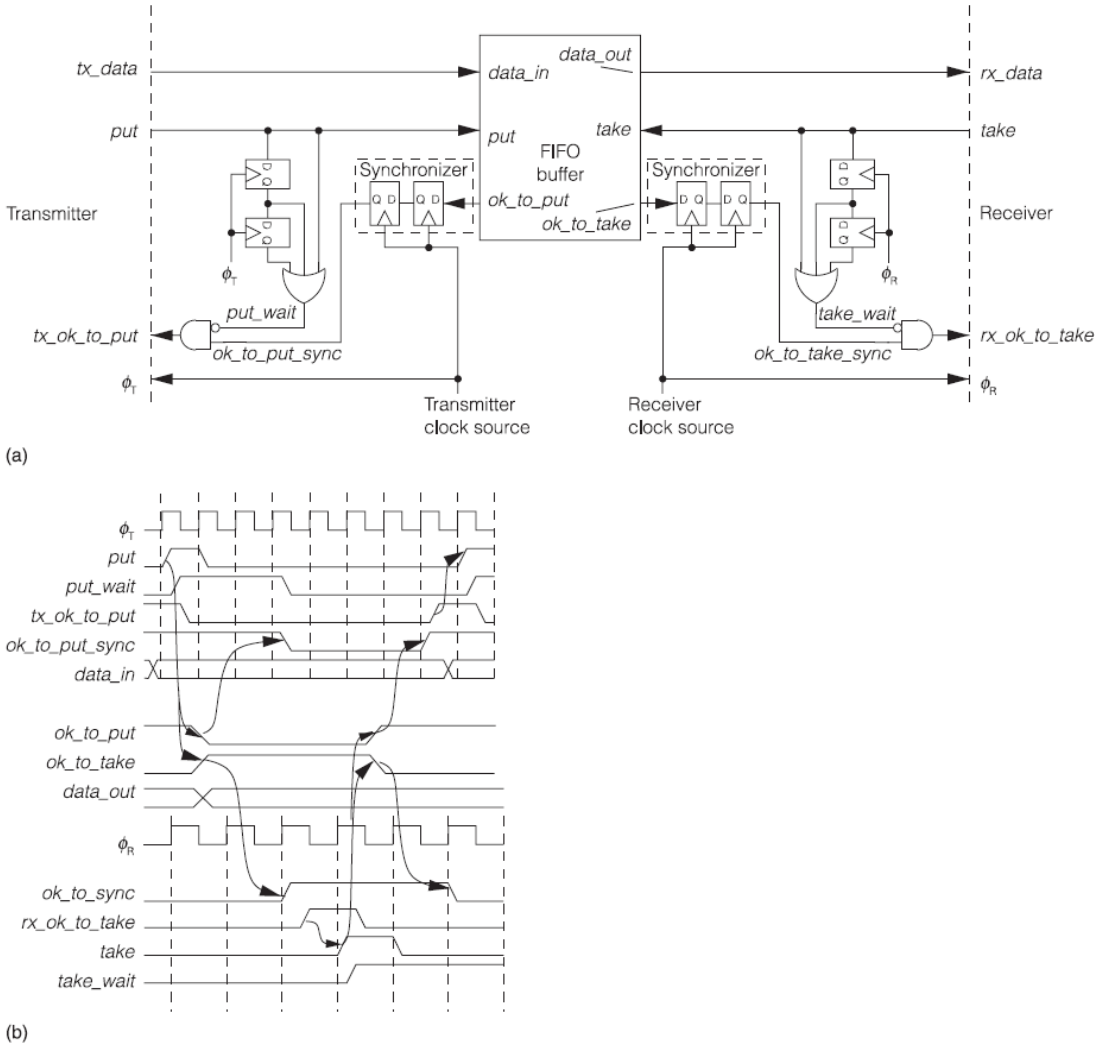


FIGURE 12 – ASYNCHRONOUS INTERFACE GALS DESIGN STYLE: CIRCUIT (A) AND TIMING DIAGRAM (B), *RETRIEVED FROM[18]*

The asynchronous GALS style is expected to find widespread use in SoC designs that can tolerate the extra latency of synchronization.

2.4.3. LOOSELY SYNCHRONOUS INTERFACES

Loosely synchronous interfaces are used for high –performance designs, however some bounds on the communication frequency have to be known and this design is

less prone to dynamic changes in the clock frequency. Handshaking becomes unnecessary during data transfer, so the resulting circuits can achieve higher performance and have higher deterministic latencies than those of the other methods.

A loosely synchronous design exploits one of the known timing relationships described earlier. The simplest case is the mesochronous relationship, in which the frequencies of the local clocks are exactly matched but differ in phase. This situation, normally, occurs when the clocks are derived from the same source but the latency of delivery to each block differs. An example of this scheme can be seen in Figure 13. The sending and receiver clocks are derived from the same source. The solution here is for the receiver to compensate the phase difference through a FIFO buffer. The key for high performance is to initialize the FIFO buffer to be half full. During operation, the transmitter puts one datum into the FIFO buffer every cycle, and the receiver takes one datum. Neither one needs to check on the status of the buffer, it is assumed that it is fast enough. So the buffer will remain + 1 or -1 data item of half full because the frequencies are matched.

To get the FIFO buffer half full, special initialization is required. Initially, a global *reset* signal is asserted, which may need to be synchronized. The TX_INIT block awaits a fixed number of cycles until the reset is guaranteed to have completed everywhere, and then enables the transmitter by asserting *tx_enable*. The transmitter begins to send data. After the first reset data item arrives, empty goes low. Because the transmitter and receiver can have arbitrary skew, this change of *empty* is asynchronous with respect to the asynchronous clock and must be synchronized. After the synchronizer latency, the RX_INIT block receives the signal, awaits any additional cycles necessary for the FIFO buffer to reach-full state, and asserts

rx_enable. The receiver begins to retrieve data items at the same rate the transmitter sends, and no further synchronization is required.

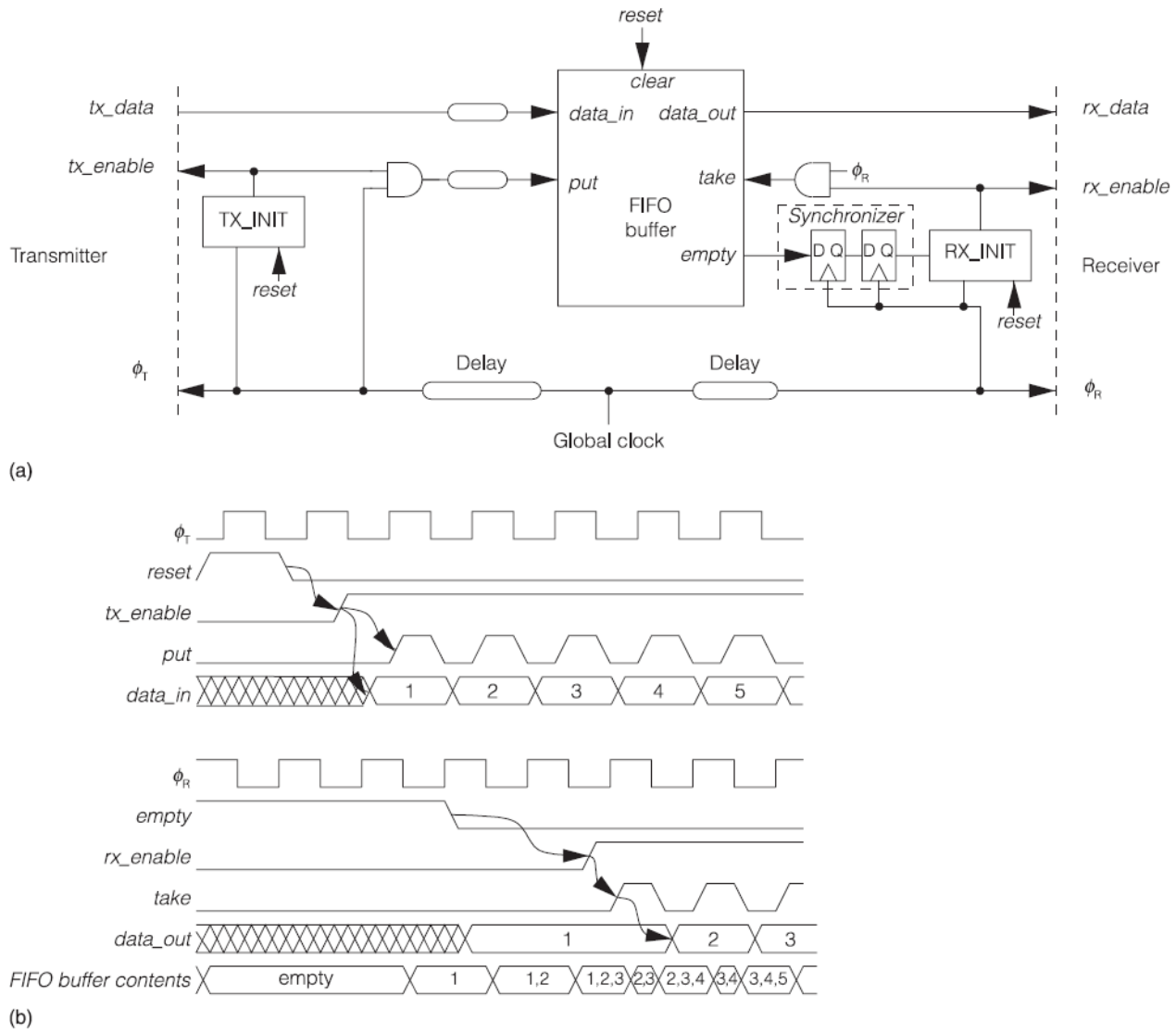


FIGURE 13 – LOOSELY SYNCHRONOUS GALS DESIGN STYLE: CIRCUIT (A) AND TIMING DIAGRAM (B), *RETRIEVED FROM[18]*

Bearing in mind the design issues, determining the optimal size of the FIFO buffers is necessary, as well, timing analysis are required to bound how far the relative phase differs between sender and receiver.

In conclusion, the need for high clock frequencies and low latency in high-performance designs will make them ideal candidates for loosely synchronous techniques.

CAD support is expected to emerge as designers undertake chips with many timing domain[18].

2.5. CONCLUSION

GALS design style build on the extensive infrastructure of synchronous design while avoiding the problems of a distribution of global, low-skew clock and metastability altogether. This methodology also facilitates the integration independently design blocks operation at different frequencies, which become a natural approach for SoC design.

Pausible clocks are appealing for their elimination of metastability failures but do not fit well in CAD flows and do not scale well for designs with high-speed clocks. Therefore it is unlikely that they will find wide spread acceptance, although their ability to completely shutdown makes them attractive for low-power designs.

Fully asynchronous interface offer the greatest flexibility. Commercial tools are already evolving in this direction, with tools that check circuits spanning multiple clock domains for structural and protocol errors. Additional problem with GALS design is that synchronizers and arbiters are nondeterministic, which complicates design validation and test. Some researchers have sought to make the timing of GALS designs deterministic [25]. Validation and test of GALS designs remains an important area for further research.

Loosely synchronous techniques offer the highest performance by removing synchronization delays from latency-critical paths. However, these methods require timing analysis that standard CAD flows do not support.

In conclusion GALS design faces the challenge of acquire support from the CAD developers in order for the designers to accept and develop with this technology. Designs with fully asynchronous interfaces seem to require the least change to the local blocks while avoiding the need for a new global timing analysis tools.

3. ASSEMBLING AN ASYNCHRONOUS INTERFACE

3.1. INTRODUCTION

Now that a global overview of the existing GALS solutions has been made and all the advantages and disadvantages been weighed it is important to select one that will adapt better to the solution we seek. The plausible clock proved to of difficult implementation and the necessity to freeze the system is a major drawback. As for the loosely synchronous is not possible to turn it into a general solution has it is necessary to know beforehand the specificities of each system. So the selection for the Asynchronous Interface was based in the criteria for the adaptability to any system, and, therefore greater reusability.

In order to assemble such an interface we have to put together a series of elements.

The choice criteria for each of the elements were that each of them were already utterly studied and proven. Also, in a global view, all the elements assembled and working together emerged as the least complex, thus, the ones who performed their duty with the least circuit size and speed reduction.

In this chapter these items will be discussed in detail.

3.2. COMPONENTS FOR ASSEMBLING THE ASYNCHRONOUS INTERFACE

3.2.1. THE SYNCHRONIZER

To achieve signal synchronization with a given clock we need no more than two flip-flops in series. With this simple design we not only get the wanted synchronization,

but also, minimize the probability of metastable failure. Figure 14 shows a common two-flip-flop synchronizer. Failure probability drops exponentially with settling time or, equivalently, with the number of flip-flops in the chain. Synchronizers can provide mean times between failures (MTBFs) of millions years or more if properly designed [26].

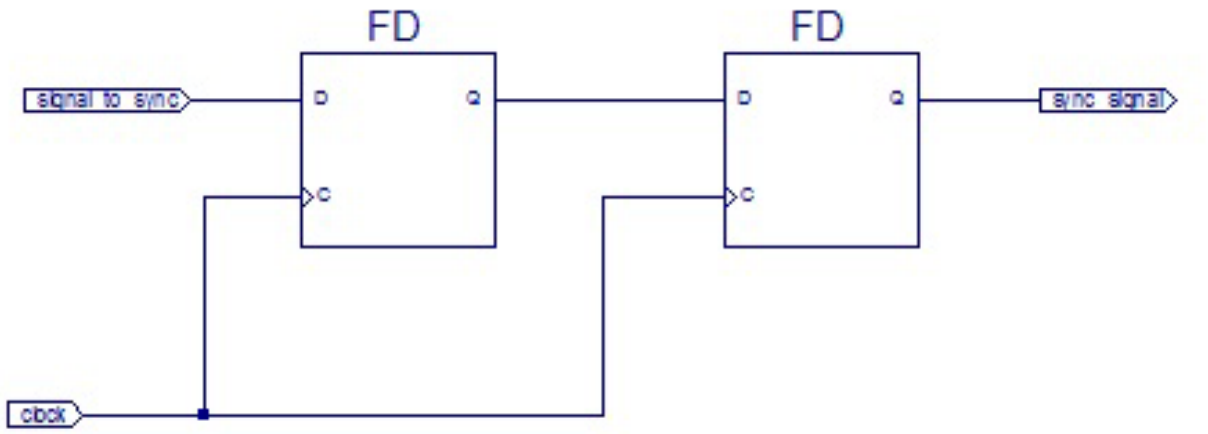


FIGURE 14 – TWO-FLIP-FLOP SCHEMATIC DESIGN.

We can see below the circuit simulation of the two flip-flop design.

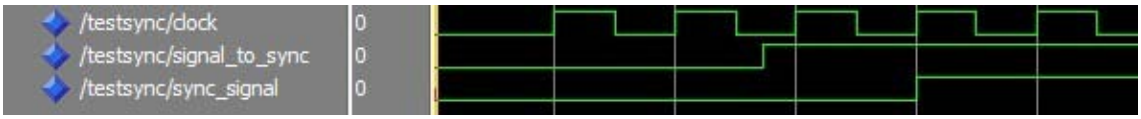


FIGURE 15 – SIMULATION OF THE TWO-FLIP-FLOP

As you can see from Figure 15, it takes at two clock cycles in a two-flop scheme to achieve the synchronization.

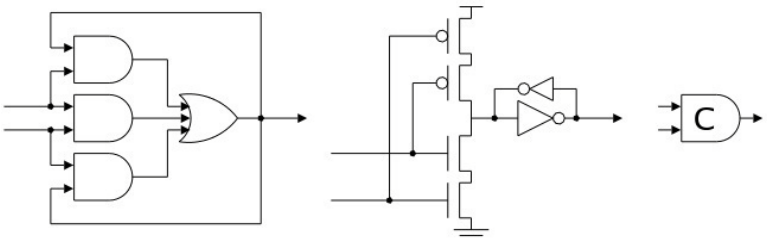
In the case where the first flip-flop becomes metastable, being the probability of $1-e^{-T/\tau}$ [9](which is infinitesimal close to 1) to exit metastability by the next clock, and has

arbitrarily settled to either high or low. If high, the next flop will go high on the next cycle. If low, it will surely go high on the next clock when the signal to synchronize is already stable.

3.2.2. THE MULLER C-ELEMENT

In order to successfully build a FIFO buffer, we will need to add the C-element to our design.

The Muller C-element (often known as C-element or a C-gate), depicted in Figure 16, is commonly encountered in asynchronous VLSI (Very Large Scale Integration) design.



a	b	y
0	0	0
0	1	no change
1	0	no change
1	1	1

FIGURE 16 – THE MULLER C-ELEMENT: POSSIBLE IMPLEMENTATION, SYMBOL AND FUNCTION DEFINITION, *RETRIEVED FROM*[27]

The Muller C-element is a state-holding element much like an asynchronous set-reset latch. When both inputs are 0 the output is set to 0, and when both inputs are 1 the output is set to 1. For other input combinations the output does not change. Consequently, an observer seeing the output change from 0 to 1 may conclude that both inputs are now at 1; and similarly, an observer seeing the output change from 1 to 0 may conclude that both inputs are now at 0.

I also had to include a variation of the C-element called the asymmetric C-element, which is also needed to assemble the latch controller (discussed later). This type of element allows inputs which only effect the operation of the element when transitioning in one of the directions. Asymmetric inputs are attached to either the

minus (-) or plus (+) strips of the symbol. The common inputs which effect both the transitions are connected to the centre of the symbol. When transitioning from zero to one the C-element will take into account the common and the asymmetric plus inputs. All these inputs must be high for the up transition to take place. Similarly when transitioning from one to zero the C-element will take into account the common and the asymmetric minus inputs. All these inputs must be low for the down transition to happen. Figure 17 shows the gate-level and transistor-level implementations and symbol of the asymmetric C-element. In Figure 17 the plus inputs are marked with a 'P', the minus inputs are marked with an 'm' and the common inputs are marked with a 'C'.

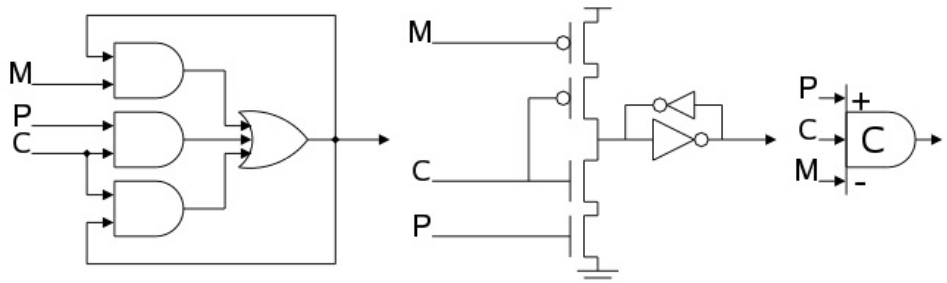


FIGURE 17 - THE ASYMMETRIC C-ELEMENT: POSSIBLE IMPLEMENTATION, SYMBOL.

3.2.3. FIFO BUFFER

A FIFO (First In First Out) buffer is nothing more than a sequence of latches that stores valid data in order, the last latch holds the first data item, as the first holds the last data. Practically speaking, the **first** item **IN** is the **first** item **OUT.**

Bearing in mind that we are in an asynchronous time domain, we have no clock to enable the latches. Therefore, to accomplish this feat, we need, for each latch, a controller that drives the data through each of the latches, from the first to the last, and altogether manages the pace of the transfer flow (the sender cannot transmit

data while buffer is full). To this form of control, in the asynchronous domain, we call handshake.

After we dwell in the realm of the handshake protocols we will first take a look at the Muller pipeline, which is common ground to all protocols.

3.2.3.1. HANDSHAKE PROTOCOLS

The choice of the handshake affects the circuit implementation (area, speed, power, robustness, etc.). We will discuss three protocols that most practical circuits use:

- 4-phase bundled-data;
- 2-phase bundled-data;
- 4-phase dual-rail;

3.2.3.1.1. 4-PHASE BUNDLED-DATA

The term *bundled-data* refers to a situation where the data signals use normal Boolean levels to encode information, and where separate request and acknowledge wires are bundled with the data signal[9].

In the *4-phase* protocol the term 4-phase refers to the number of communication action(see Figure 18): (1) the sender issues data and sets request high, (2) the receiver absorbs the data and sets acknowledge high, (3) the sender responds by taking request low (at which point data is no longer guaranteed to be valid) and (4) the receiver acknowledges this by taking acknowledge low. At this point the sender may initiate the next communication cycle.

3.2.3.1.2. 2-PHASE BUNDLED-DATA

In the standard two-phase protocol an event (rising or falling edge) on Rin signals the availability of input data and the register issues an event on Ain to indicate to the

source of the data that it has been captured and may be removed. The latch also issues an event on Rout to indicate that its output data is now valid and will be held stable until an event on Aout signals that it has been accepted by the next stage in the pipeline[28].

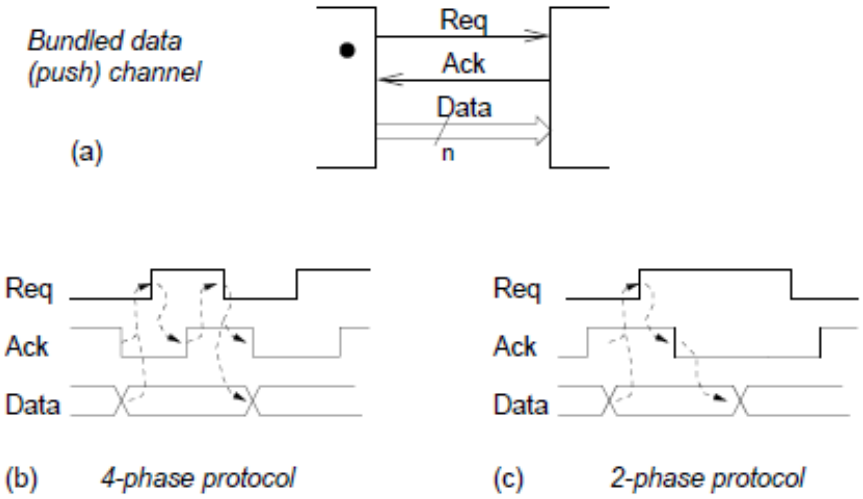


FIGURE 18 - A) A BUNDLED-DATA CHANNEL. B) A 4-PHASE BUNDLED-DATA PROTOCOL. C) A 2-PHASE DATA PROTOCOL, *RETRIEVED FROM* [9]

3.2.3.1.3. 4-PHASE DUAL-RAIL PROTOCOL

The 4-phase dual-rail protocol encodes the request signal into the data signals using two wires per bit of information that has to be communicated.

The core of it is a 4-phase protocol using two request wires per bit of information. One wire is used for signaling a logic 1 (or true), and another wire is used for signaling logic 0 (or false). When observing a 1-bit channel one will see a sequence of 4-phase handshakes where the request signal in any handshake cycle can be either *data true* or *data false.* Two parties can communicate reliably regardless of delays in the wires connecting the two parties, which makes this protocol very robust and delay-insensitive.

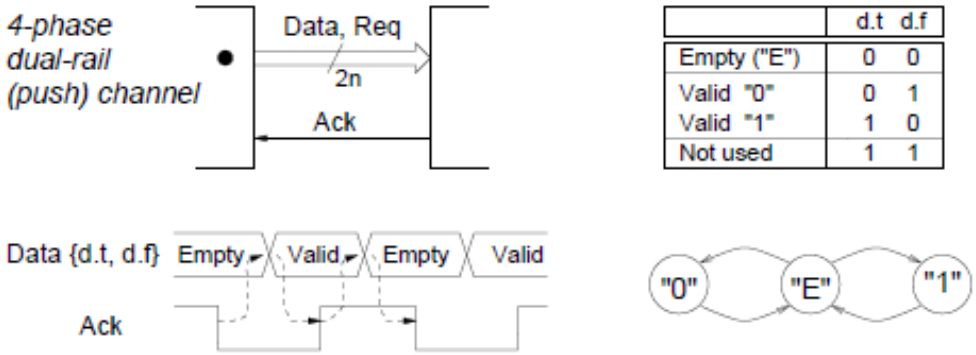


FIGURE 19- A 4-PHASE DUAL-RAIL PROTOCOL, *RETRIEVED FROM*[9]

3.2.3.2. MULLER PIPELINE

The backbone of almost all asynchronous circuits[9] is shown Figure 20. The circuit is built from C-elements and inverters and is known as a Muller pipeline or a Muller distributor.

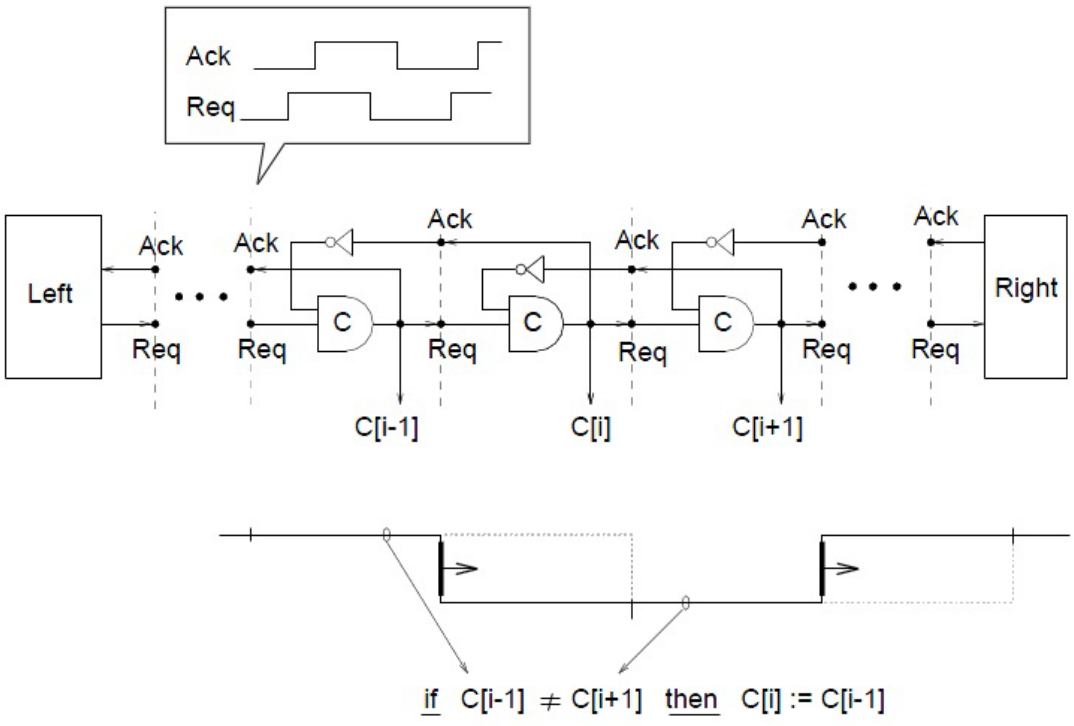


FIGURE 20 - MULLER PIPELINE, *RETRIEVED FROM[9]*

The figure is interpreted as follows: After all of the C-elements have been initialized to 0 the left environment may start handshaking. The i^{th} C-element ($C[i]$) will propagate, input and hold, a 1 from its predecessor, $C[i - 1]$, only if its successor,

C[*i*+1], is 0. In an analogous way it will propagate a 0 from its predecessor if its successor is 1. It is often helpful to think of the signals propagating in an asynchronous circuit as a sequence of waves, as illustrated at the bottom of Figure 20. The elation one can take on the role of a C-element in the pipeline is to propagate peaks of waves in a controlled way that maintains the integrity of each wave.

“On any interface between C-element pipeline stages an observer will see correct handshaking, but the timing may differ from the timing of the handshaking on the left hand environment;”[9] The speed in which a wave will propagate in the circuit is dependable on the circuit itself. The first handshake (request) injected by the left hand environment will reach eventually the right hand environment. If the right hand environment does not react to the handshake the pipeline will eventually fill and the pipeline will stop handshaking with the left hand environment, has the ripple of the wave will propagate back through the FIFO.

Also, regardless of delays in gates and wires the circuit works correctly because of its delay-insensitive property.[9]

3.2.3.3. 4-PHASE BUNDLED-DATA PIPELINE

A 4-phase bundled-data pipeline is particularly simple. A Muller pipeline is used to generate local clock pulses. The clock pulse generated in one stage overlaps with the pulses generated in the neighboring stages in a carefully controlled interlocked manner.

The pipeline implementation is particularly simple but it has some drawbacks: it only accomplishes to fill every odd latch because the state of the C-elements are (0,1 ,0 ,1 , …). Also the throughput depends on the time it takes to complete a handshake

cycle and for the above implementation this involves communications with both neighbors.

We will discuss next a better implementation.

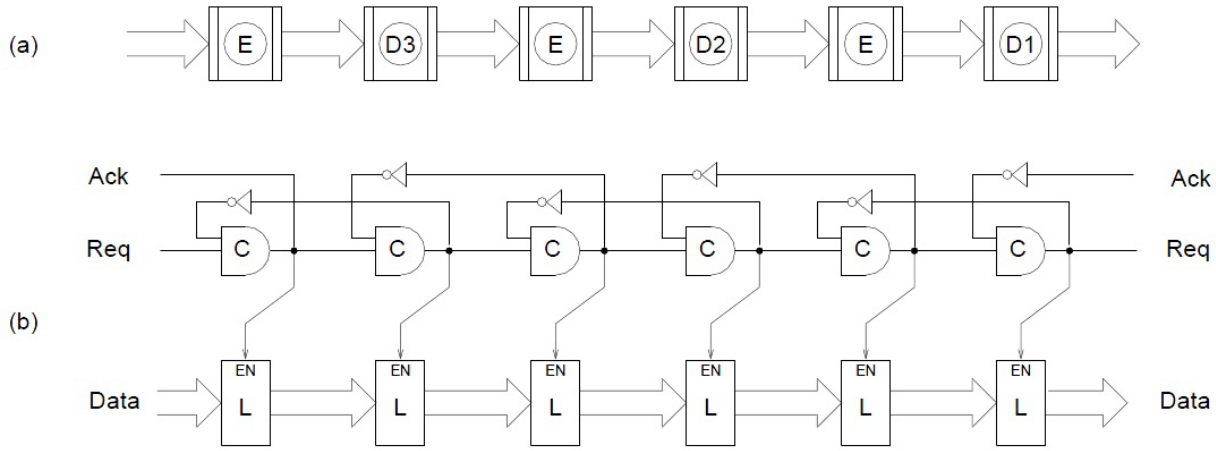


FIGURE 21 – (A) A 4-PHASE BUNDLED-DATA PIPELINE, AND (B) ITS IMPLEMENTATION USING SIMPLE LATCH CONTROLLER AND LEVEL SENSITIVE LATCHES. THE FIFO FILLS EVERY OTHER LATCH, *RETRIEVED FROM[9]*

3.2.3.4. SEMI-DECOUPLED LATCH CONTROLLER

Ideally one would want to fill every latch with valid data. This can be achieved through the implementation of Semi-Decoupled latch controller. The original design is from Furber, et al.[28], and can be seen in Figure 22(a).

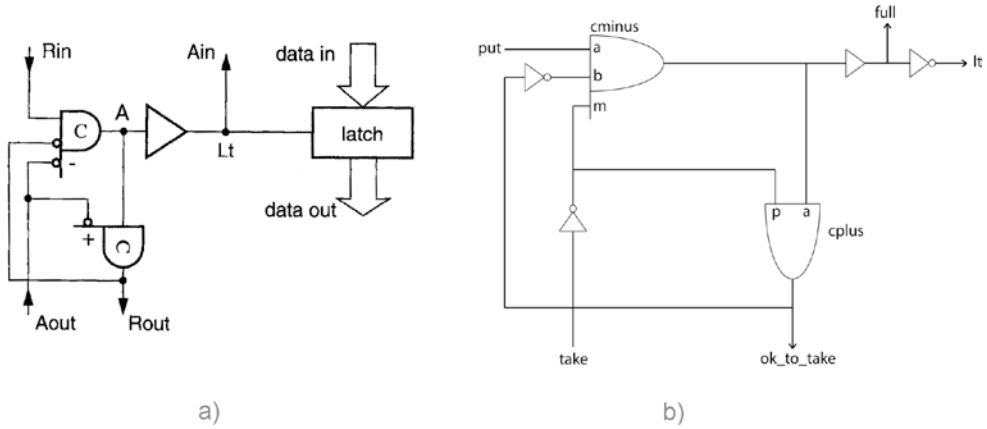


FIGURE 22 - SEMI-DECOUPLED LATCH CONTROLLER (A) ORIGINAL FURBER DESIGN(RETRIEVED FROM [28]) , NORMALLY OPAQUE (B) NORMALLY TRANSPARENT

The method, Furber used, was to add an internal variable A to the four-phase simple latch controller. This variable is used to indicate when the input side of the latch controller is ready to proceed, independently of the output side. This allows the latch to capture new set of data regardless of the next latches condition.

Testing the design it become clear that some minor alterations needed to be performed in order to adapt to the asynchronous interface. The problem, as we can see in the wave simulation depicted in Figure 23, is that if the next latch is not ready to receive the next item, it will remain enabled, thus, losing the current value (because the transmitter was informed by *Ain* that the data was latched with success).

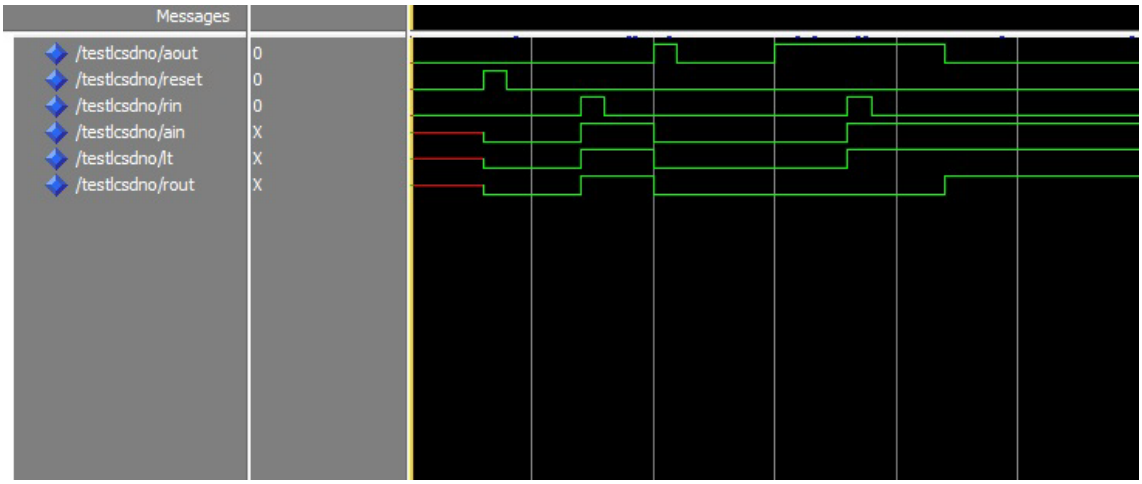


FIGURE 23 - WAVE SIMULATION OF SEMI-DECOUPLED LATCH CONTROLLER FURBER DESIGN

The solution was to apply an inverter to *Lt*. In that way the latch will become normally transparent. When *Rin+* arises the latch will became opaque and hold the value until the next latch is ready to take it (*Aout-*). The wave simulation is in Figure 24.

FIGURE 24 -WAVE SIMULATION OF THE IMPLEMENTED SEMI-DECOUPLED LATCH CONTROLLER

As we can see from Figure 22, the *Lt* (latch toggle) signal is high after the reset. *Lt* goes low after the *put+,* and the flag *full* and *ok_to_take* goes high. The signal names where changed in order to better correlate to the signals of the asynchronous interface.

After *take+*, *ok_to_take* and *full* goes low, and *Lt* goes high.

In the second part of the test (after the vertical yellow line), we consider the next latch is full. This corresponds to the *take* signal high. As we can see, despite the next latch is *full*, when the put signal goes high, *Lt* becomes low, storing the data. *Full* will become high, but *ok_to_take* will remain low until the *take* signal goes low. When this occurs, the latch will know that the next one is ready to take the next data and the *ok_to_take* flag will become high.

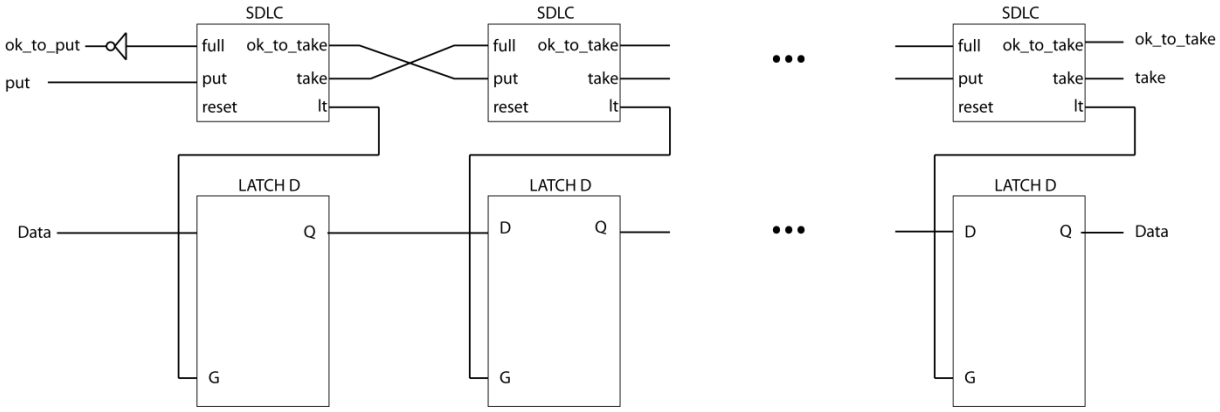


FIGURE 25 – GENERAL DESIGN OF A SEMI-DECOUPLED FIFO BUFFER

I will make mine the word of James B. Johnson in is master thesis [29] the reason for the choice of this type of latch controller:

“The semi-decoupled controller is that the performance/area ratio is better than the fully decoupled four-phase controller also introduced in Furber’s papers [28]. We have already noted the problems with the simplified four-phase latch controller as far as the ability to fill all of the stages of the FIFO. The simplicity of the semi-decoupled

controller design combined with the capability to use transmission gate style latches fits well with our goals of a high-performance, small area data FIFO".

3.3. ASYNCHRONOUS INTERFACE

Now we have all the elements to assemble the asynchronous interface that we discussed earlier.

For reason of simplicity the first test will be with a simple buffer. It will hold only one latch d, which translates in FIFO buffer with depth 1 and width 1.

The design of the interface will be an exact replica of the scheme suggested in the paper by Paul Teehan, et al.[18]

The scheme is presented in Figure 26.

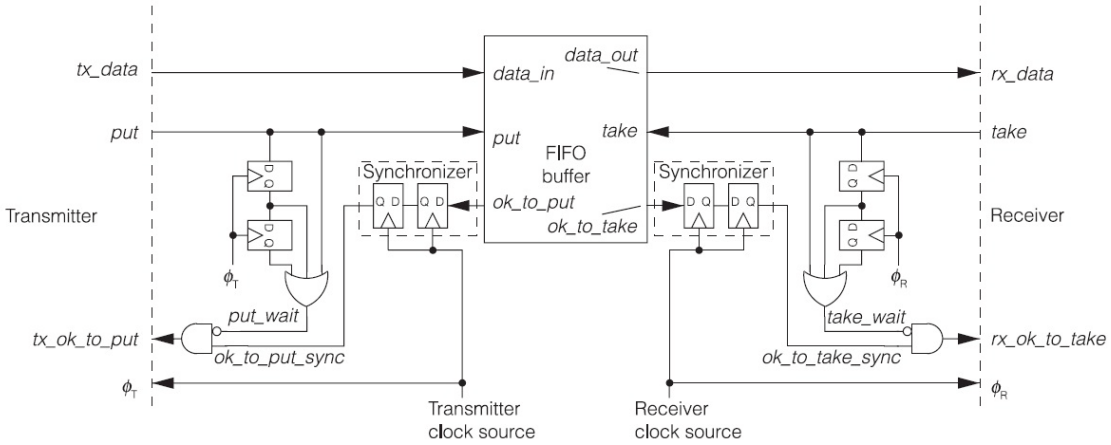


FIGURE 26 - ASYNCHRONOUS INTERFACE, *RETRIEVED FROM*[18]

For the test, the transmitter (rx_clock) and receiver clock (tx_clock) will have the same frequency, but will comprise an offset of 50 ns. We are then before a *mesochronous* case.

The wave simulation can be observed in Figure 27. The transmitter will send two data item in sequence.

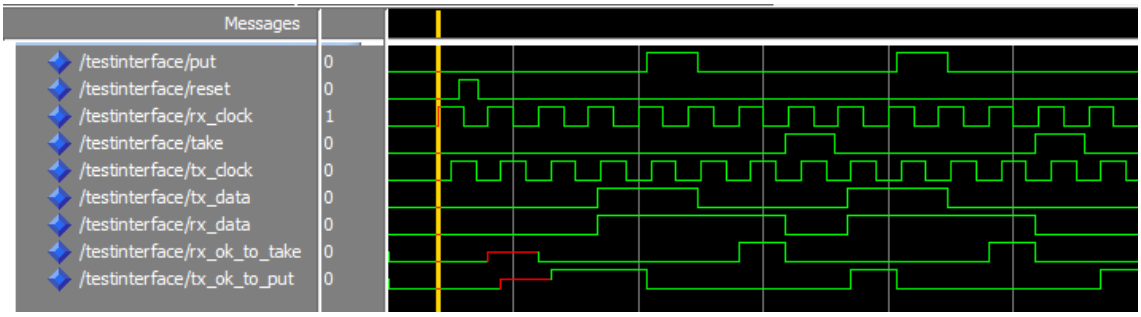


FIGURE 27 - ASYNCHRONOUS INTERFACE WAVE SIMULATION

It can be observed, with the help of the vertical line, the offset between the clocks.

After the *reset ok_to_put* goes high. The *put*+ latches the data and the FIFO puts *ok_to_take* high. it takes three receiver's clocks for *rx_ok_to_take* to go high. This delay is due to the synchronizer. The receiver latches the data and *take* goes. Upon this the buffer acknowledges that the data was successfully transmitted and *tx_ok_to_put* goes high. The buffer is ready to take the next data and the cycle repeats.

3.4. CONCLUSION

In simulation the interface successfully makes the bridge between two different time domains. Observed behavior was in according with textbook.

The greatest difficulty was in the construction of the latch controller. The simple latch controller, the Muller pipeline, lacked the elasticity to achieve the necessary performance. The Fully-Decoupled latch controller was complex and cumbersome and even following several designs from different authors was impossible to achieve the correct behavior. This is probably due to some ingenuity of the author of this

thesis, and the design should, maybe, be assembled in some way the author could not grasp.

The Semi-Decouple latch controller proved to be elegant in design, fitting perfectly the needs of FIFO buffer, without adding unnecessary header and complexity.

The next step (addressing in the next chapter) will be a real world test in an FPGA.

After the simulation success, the expectations are high.

4. A NEW ASYNCHRONOUS INTERFACE

4.1. INTRODUCTION

The need for a new interface rose from the problem that synchronous circuits do not possess handshake signals to commute data.

In order for a LSI communicate to another LSI we need reconfigure the previous studied asynchronous interface. This will lead us to add two new elements: The Input Port and the Output Port.

The new interface to be implemented can be observed in Figure 28. We can see that the FIFO buffer remains the same from chapter 3 and the ports make the signal transition from a LSI to the FIFO buffer and again to another LSI. It is an interface molded to fit below the layers of the FORDESIGN project [8] after the split in the Petri net models and obtaining several sub-models. The new interface will help to transmit the several event signal generated in each sub-models to each other.

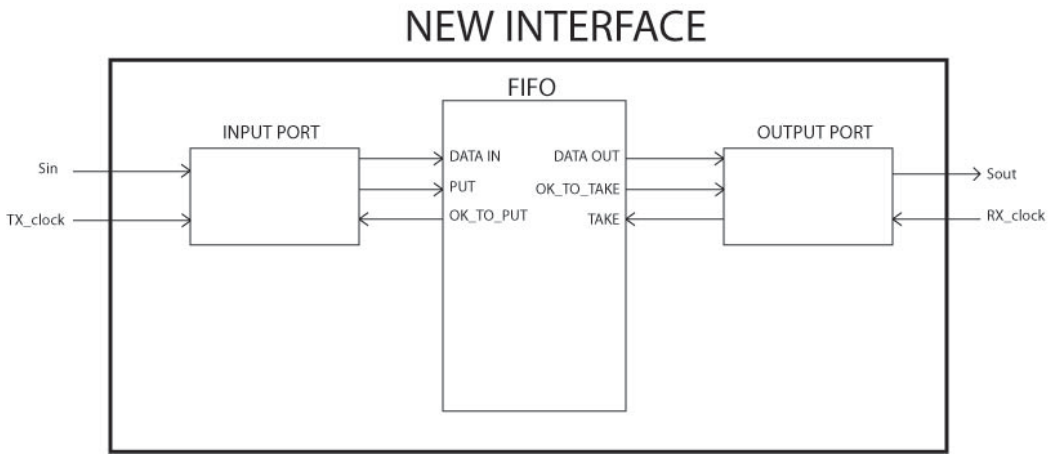


FIGURE 28 - NEW ASYNCHRONOUS INTERFACE

Before dwelling in these new components I will first demonstrate how and why the previous solution is inadequate by introducing the examples in which we will apply the developed solution.

4.1.1. EXAMPLE 1

I will present now one of the examples in which we will apply the solution. The only concern here is to explain the communication characteristics of both examples, for there will be a more detailed explanation in the next chapter.

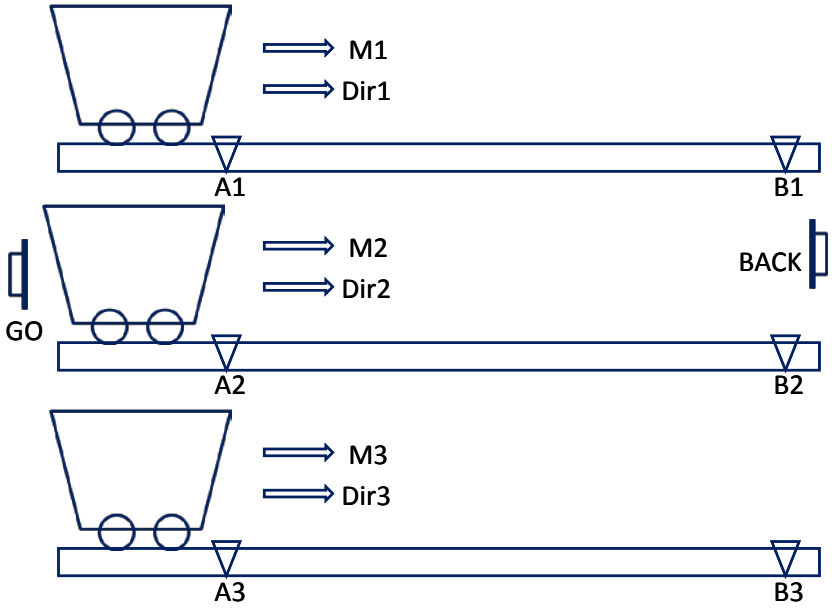


FIGURE 29 – 3 CARS EXAMPLE

On all three cars are in the starting lane they start moving upon pressing the button GO and each stop at the end when Bx (being x the car number) is pressed. Upon pressing BACK button the start moving again to starting lane until Ax is pressed for each one. The cars can only start moving when all of them have finished a course.

Figure 30 is the Petri Net representation of the controller for a 3 car system example. It represents three cars moving back and forth.

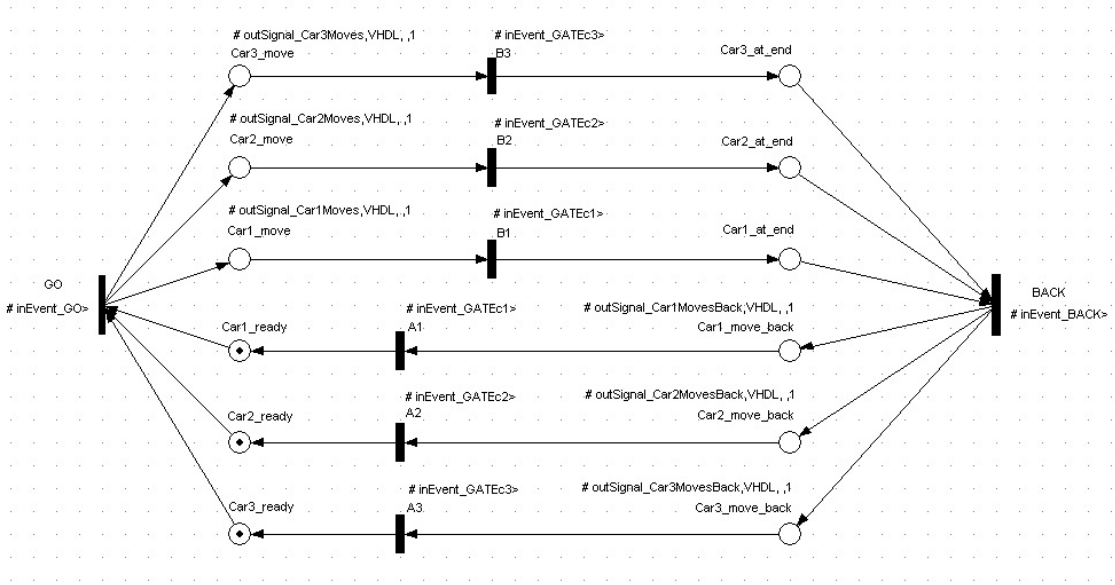


FIGURE 30 – 3 CAR SYSTEM EXEMPLE PETRI NET MODEL OF THE CONTROLLER

The input signals of this controller are:

- GO;
- BACK;
- A1;
- A2;
- A3;
- B1;
- B2;
- B3;

And also, for the implementation, are present the clock and reset signal. .

The output signals are:

- Car 1 moves;
- Car 2 moves;
- Car 3 moves;
- Car 1 moves back;

- Car 2 moves back;
- Car 3 moves back;

In order to get three separate models, one for each car, we apply the SPLIT tool [7].

This tool allows for a net splitting operation able to decompose a Petri net model into Petri net sub-models using synchronous communication channels. The generated sub-models are associated with components to be executed concurrently, allowing a distributed execution of the initial model.

The net splitting operation is hold on the basis of a defining a valid cutting set and following specific rules. To define the cutting set he user starts by identifying a set of components to be executed concurrently. To represent the initial and final model it is used PNML notation. These can be produced using the graphical editor for Input-Output Place-Transition Petri Net Class (SnoopyIOPT editor).

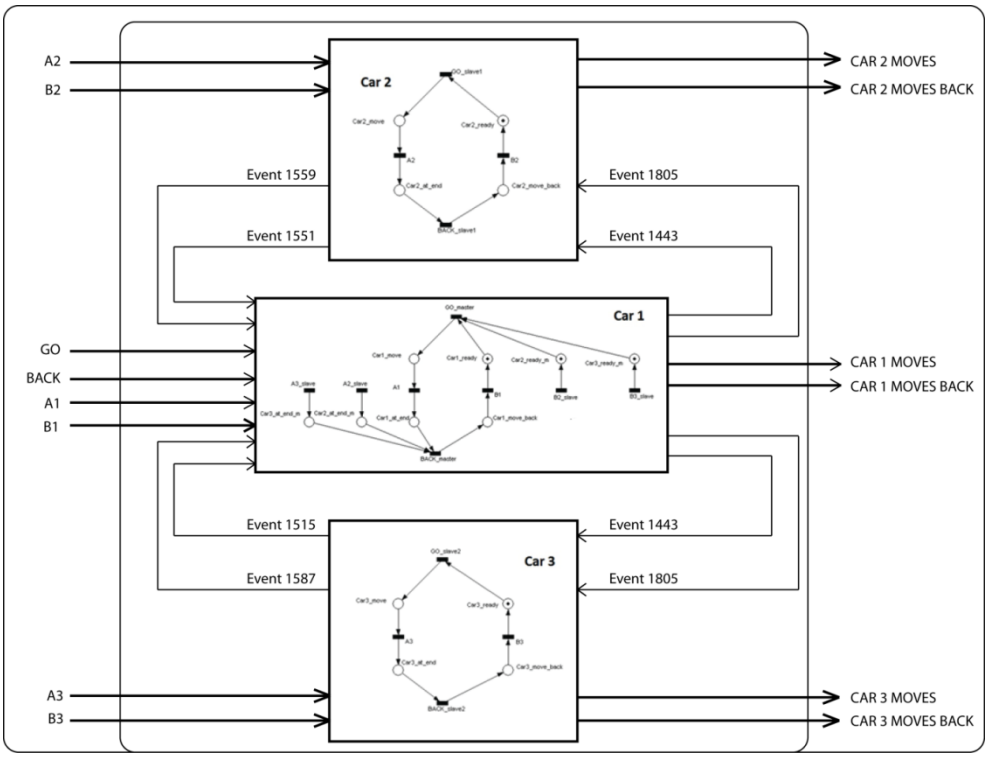


FIGURE 31 – THE 3 CARS SYSTEM SPLIT AND SIGNALS INPUTS AND OUTPUTS

After the validation of the cutting set there are three rules that allow the generation of components, interconnected through synchronous communication channels. At the implementation level, each component can be seen as an autonomous model, however having information about the state of adjacent components.

From the Figure 31 we can observe all the input and output signal of the modules generate by the split operation. In order for the model to maintain coherence event signals are generated between modules to transmit information about affecting states.

So, after the split operation, no handshake signals are present for each of the events in order for module communication to be accomplished. The only signals present are the event signals, the same we wish to transmit.

The Input and Output Ports will have to bypass this problem in order to pass and retrieve the information from the FIFO buffer.

4.2. THE SIGNALS PROBLEM

With no handshake signals the interface was stripped to only the ***input data signal***, the ***transmitter's clock*** and the ***receiver's clock.***

On the other hand the FIFO buffer needs the data signal, the ***put*** and ***take*** signal, in order to successfully copulate the data from one end to the other. It also as the control signals ***ok_to_take*** and ***ok_to_put***.

The problem here is not has trivial has it first seems due to the condition of the latch controller. The data ***in*** must arrive first or at the same time than the ***put*** signal and maintained until the put signal is rises. Failure to accomplish this and the date will not

enter the buffer. This fact is also true if the data to be transmitted is an event, which can occupy only a fraction of the clock signal, which makes matters worse, because the event will have to be extended until it can enter the buffer.

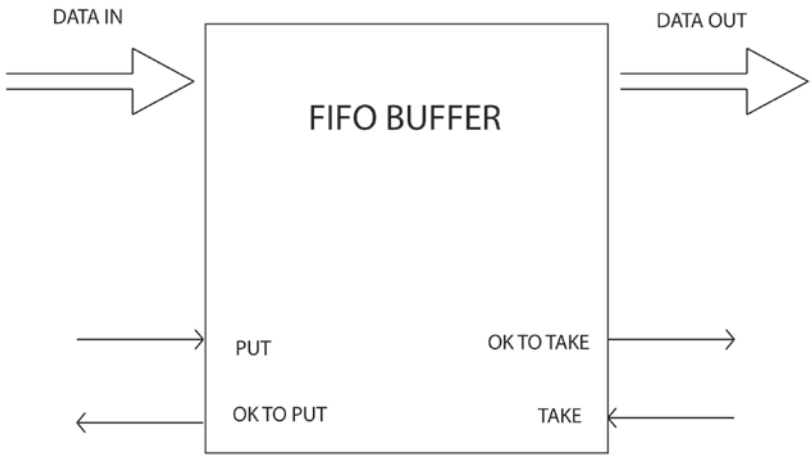


FIGURE 32- FIFO BUFFER INPUT AND OUTPUT SIGNALS

For a deeper analysis on the FIFO buffer please consult the chapter 3 - *Assembling an Asynchronous Interface*.

4.3. THE PORTS

It was clear at this stage that a port controller was needed to interact between the module exchanging the data and the FIFO buffer.

4.3.1. THE INPUT PORT

One of the first solutions achieved in this work is presented in Figure 33. This circuit, although, achieving the desired control did not had a good performance. It only allowed the input of data only every four transmitters clock's.

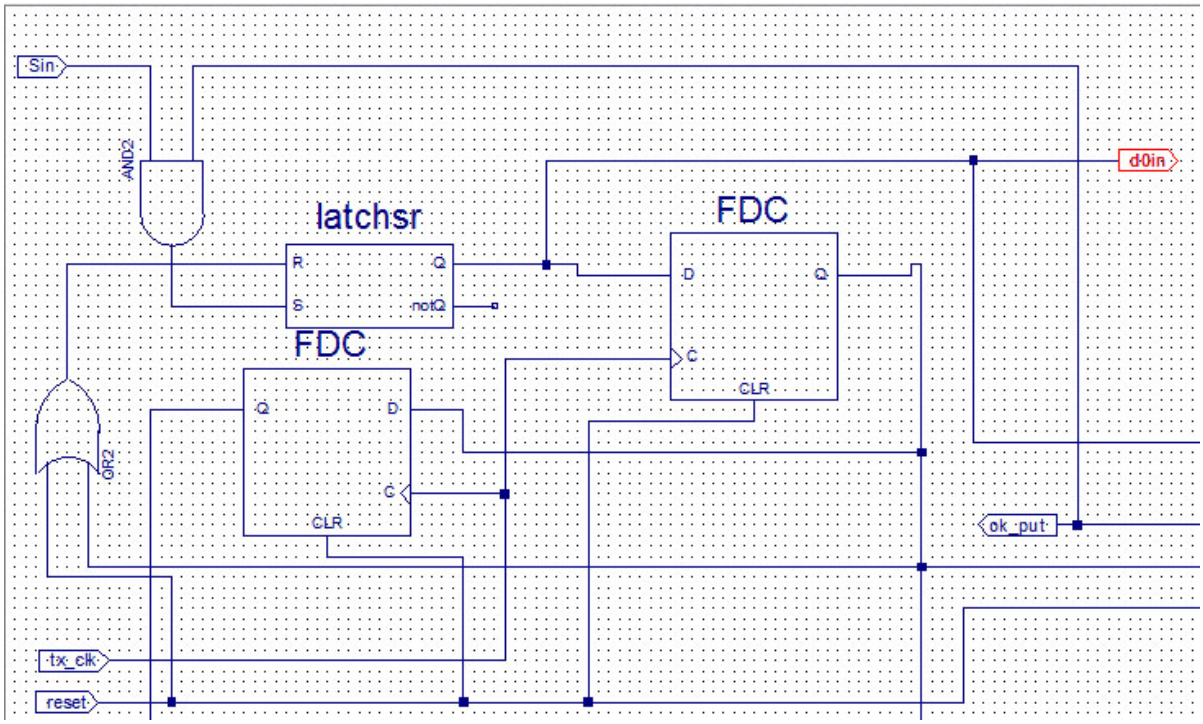


FIGURE 33 – FIRST INPUT PORT

The latch SR would extend the event or signal (Sin) until the put signal come out of the first Flip-Flop D(with one clock delay). After another clock the latch SR would be reset.

This allowed to surpass the restrains of the FIFO buffer, with only two input signals - The Sin and transmitter's clock.

Due to the performance limitations imposed by this system a new approach had to be conceived.

The new input port can be observed in Figure 34.

With this type of port one can achieve a buffer input once every two clocks, until the buffer is full. This delay is introduced by the necessity of the *put* signal to go down before a new input, and, also, because of the latch allowing only for clock synchronized inputs.

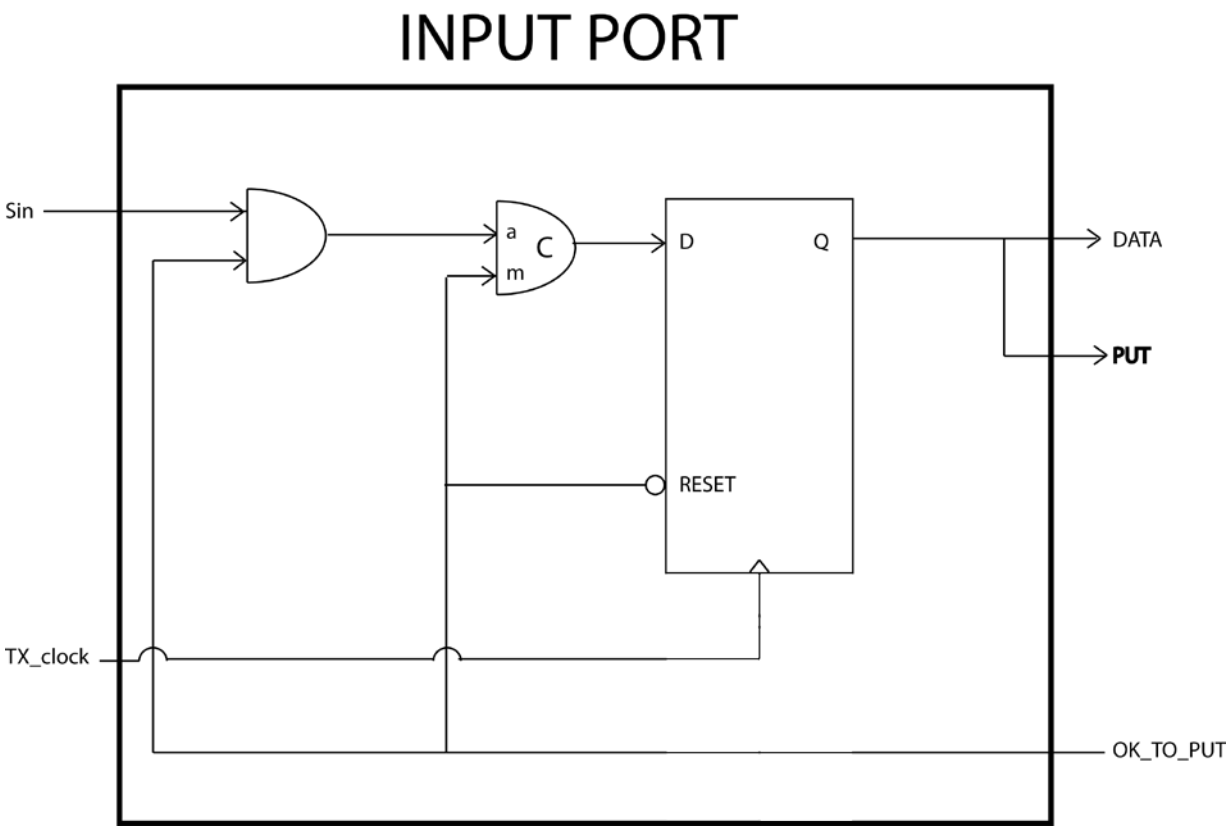


FIGURE 34 – NEW INPUT PORT

The given schematic works as follows:

If the FIFO is in condition to accept new entries then *ok_to_put* will be high. The first AND will be eligible to let a value pass through.

If an event occurs *Sin* will be high for that given period and the C Minus Element will be set high (until both *Sin* and *ok_to_put* are down). The latch is there to extend the signal until there is confirmation that it entered the FIFO buffer (has the signal *data*).

The *put* and *data* signal will be the output of flip-flop D whenever there is a gate input (in this case the output of the C Minus Element) and when *tx_clk* is high. In this way the *put* will be synchronized with the transmitter's clock and naturally with the data to be transmitted. Therefore the data being transmitted will be inserted in FIFO at the same time the clock goes high, with only a certain amount of delay introduced by the flip-flop D.

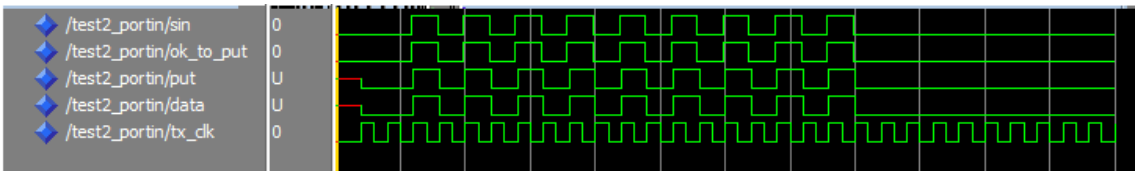


FIGURE 35 – SIMULATION OF THE NEW INPUT PORT

At this time, and for the duration of the *put* signal the *ok_put* wil be low. This implies, analyzing the schematic, that it will reset the C Minus Element and, also, the latch D. This will turn data low and consequently the put signal, when there is the next clock rise. If the FIFO is ready to take a new value the *ok_to_put* will be high and all the elements will be free to take a new value.

4.3.2. THE OUTPUT PORT

The output port is of fairly simple explanation.

The data signal needs to be synchronized with the receiver's clock in order to avoid metastability. This is accomplished using two flip-flop D, as seen in the Figure 14 and now, also, in Figure 36.

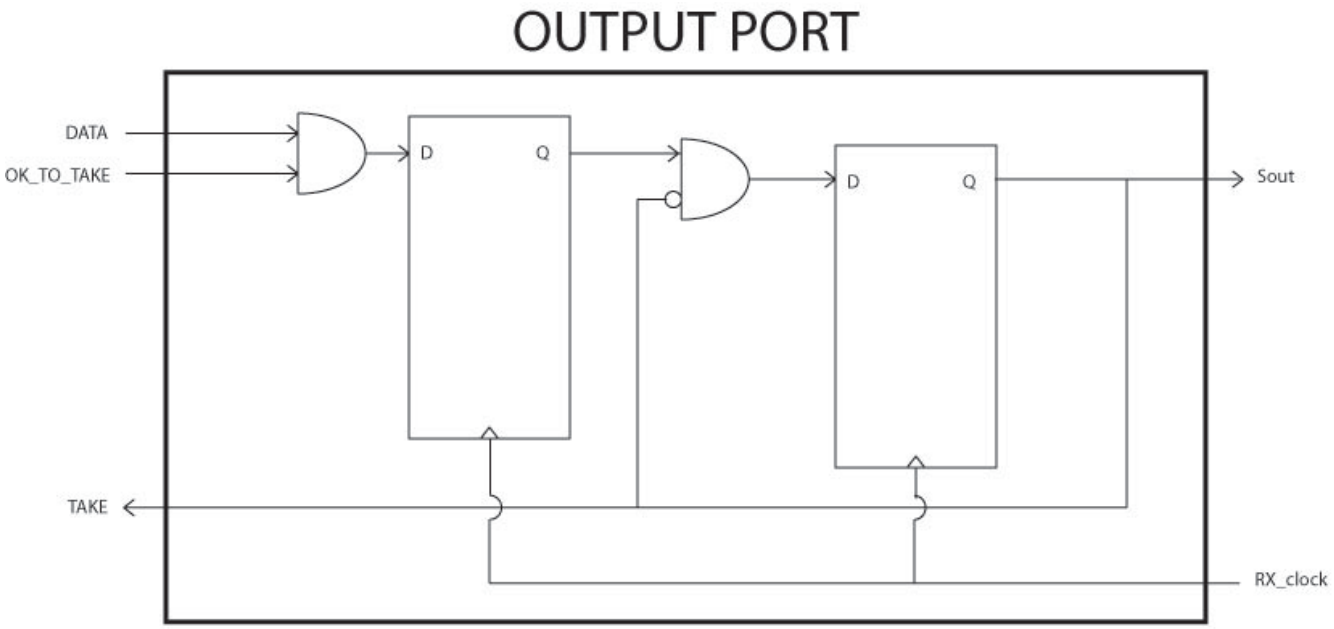


FIGURE 36 – OUTPUT PORT

The *and* port before the synchronizer is there so the output data is present when there is an *ok_to_take* signal present.

Take will be active as soon as *Sout* is high. This also implies that the synchronization was a success and the flow inside the synchronizer can be interrupted, if not, the signal would be extended for one more clock. To accomplish this we introduce an *and* gate between the two flip-flops. The *and* is only active if *Sout* is down.

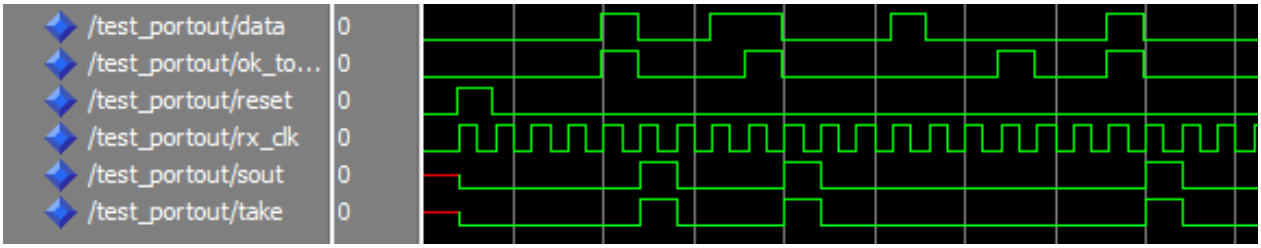


FIGURE 37- SIMULATION OF THE OUTPUT PORT

From the Figure 37 it can be observed that the port, due to the synchronizer element, introduces a delay of one clock.

4.3.3. FIFO BUFFER

The FIFO buffer[9] used in the implementation is depicted in Figure 38. It possesses two semi-decoupled latch controllers, one for each latch. Each of the latches accepts only one bit. The Semi-Decoupled latch controller[28] is the same referred in Figure 22 b), at chapter 3.2.3.4, where it is also analyzed, and can now be observed in greater detail in Figure 39.

The depth and width of the FIFO for the given implementation is more than enough because, for the examples used, the FIFO will never take more than one datum at a time.

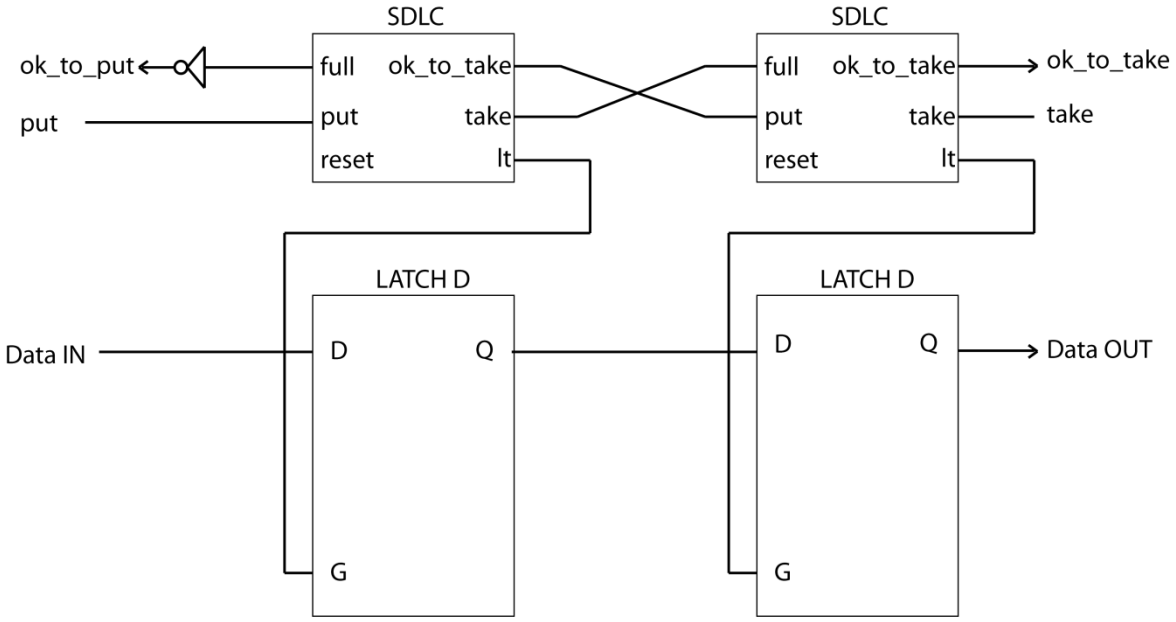


FIGURE 38- FIFO BUFFER WITH WIDTH 1 AND DEPTH 2.

The controller uses two variations of the Muller C-element[9], the cnminus and the cplus[30], presented in Figure 39 b), c) respectively. The cplus output only goes up when signal *p* and *a* is active and continues active while signal *a* is active.

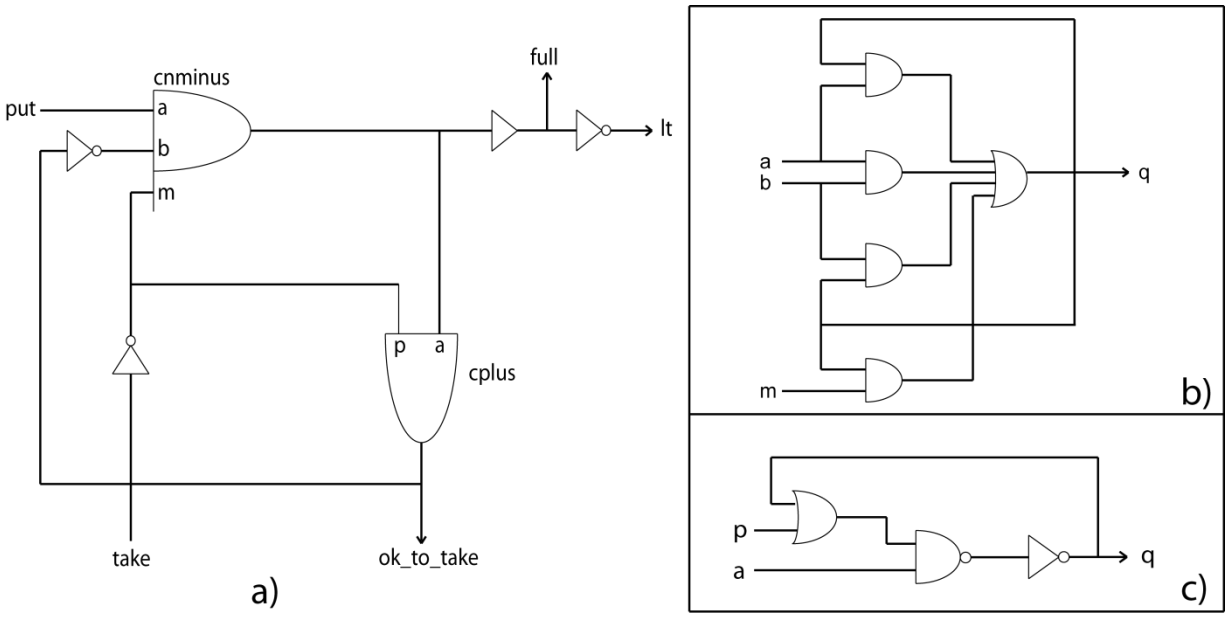


FIGURE 39 –A) NORMALLY TRANSPARENT SEMI-DECOUPLED LATCH CONTROLLER, B) CNMINUS, C) CPLUS

As for the cnminus, the behavior is identical to the C-element[9] except the output only goes down when the signal *m* goes down. To build the element with this behavior we have to draw the corresponding truth table. From the table, and building

the Karnaugh map we obtain the function $Q_{n+1} = QA + QM + QB + AB$, that permits to design the scheme present in Figure 39 b). All this steps are present in Figure 40.

Q	M	A	B	Qn+1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

a)

		A			
		1	1	1	0
B		1	1	1	0
		0	1	1	0
		0	1	0	0
			Q		M

b)

Qn+1 = QA + QM + QB + AB

c)

FIGURE 40 – CNMINUS ELEMENT. A) TRUTH TABLE, B) KARNAUGH MAP, C) FUNCTION

4.4. THROUGHPUT

So, the input signals have been reduced to three: *Sin*, *Tx_Clock* and *Rx_Clock* (not counting with the reset signal). The output is only one: *Sout*.

The behavior can be observed in Figure 41.

For the given simulation the transmitter's clock has a period of 100ns and the receiver's clock has a frequency of 150ns. The offset between the two is of 20ns.

For testing purposes the *Sin* signal is asynchronous and with varied width (it is sometimes bigger than the clock signal). The purpose of this is to determine that the width of the input signal will not influence in the efficiency of the interface.

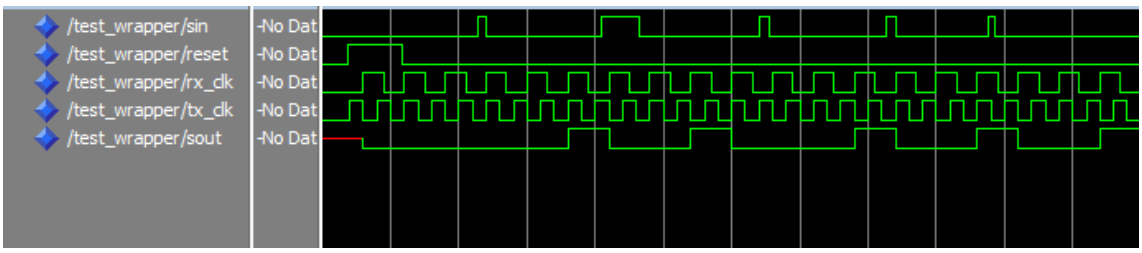


FIGURE 41- BEHAVIOR OF THE NEW ASYNCHRONOUS INTERFACE

The first data input happens before the sixth transmitter's clock edge. At that time the *put* signal goes up, *ok_to_put* goes down and *ok_to_take* goes up. In the next transmitter's clock *ok_to_put* will go up and the FIFO will be ready to a new value.

At the moment the data goes out the other side of the FIFO it take only two receiver's clocks to reach destination. This is due to the delay imposed by synchronization. Sout will go up as well as take, which will empty the FIFO and ok_to_put goes down.

Therefore an input in this Interface can be made every odd clock and the output takes two of the receiver's clock. Thus the time it will take for an event to leave the transmitter until it reaches his destination is as follows:

$$Througput(t1, t2) = 2 * Rx_{clock} + inDelay(t1) + outDelay(t2)$$

$$inDelay(t1) = Tx_{clock}\,'event - Sin(t1)$$

$$outDelay(t2) = Rx_{clock}\,'event - FIFOout(t2)$$

The equation reads as follows: the *throughput* of the signal takes two times the Receiver's clock period plus delay. The delay is composed by two more equations. The *inDelay* is the time difference between the *Sin* event, at time *t1*, and the next Transmitter's clock event. The *outDelay* is the time difference between the time data exits the FIFO buffer, at time *t2*, and the next receiver's clock event.

5. IMPLEMENTATIONS

5.1. INTRODUCTION

This chapter will present two practical problems to be solved through the use of GALS technology.

To aid in the creation of the practical example we will use three tools developed in the FORDESIGN project:

- The SNOOPY-IOPT[2], a graphical IOPT Petri net editor for modeling the example.
- The SPLIT tool [7], which enables us to partition of an IOPT Petri net into autonomous, yet inter-signal dependent, Petri net sub-models.
- The PNML to VHDL tool[8], to convert the PNML to VHDL modules.

As it was previously said the generated sub-model from the partition will generate events in order to maintain model coherence. It is these events that will have to be interconnected with the solution developed in this thesis in order for the example to properly work.

5.2. EXAMPLE 1: THE 3 CARS SYSTEM

To test our GALS interface we need several modules with different clocks.

First we will use a know example and observe its behavior.

The model in Figure 30 (presented in chapter 4) represents the controller of three cars moving back and forth. They can only start moving forward when all of them are

in their ready place (car1_ready, car2_ready, car3_ready). Upon occurrence of the event GO all tokens pass to car_move place (car1_move, car2_move, car3_move) which represents the cars moving forward. They will remain in this state until each of their corresponding transition are activated (B1, B2, B3). At that time each of the tokens makes the transition to the end place (car1_at_end, car2_at_end, car3_at_end).

For the cars to move back and complete the cycle each one as to be at end place before the transition back is activated, which at that time each mark is moves to his corresponding move back place (car1_moves_back, car2_moves_back, car3_moves_back). To reach initial place each of their corresponding transition must be activated. When each of the car completes this cycle the system is in the initial state and can start all over again.

The simulation of this system, after being converted to VHDL, can be observed in Figure 42:

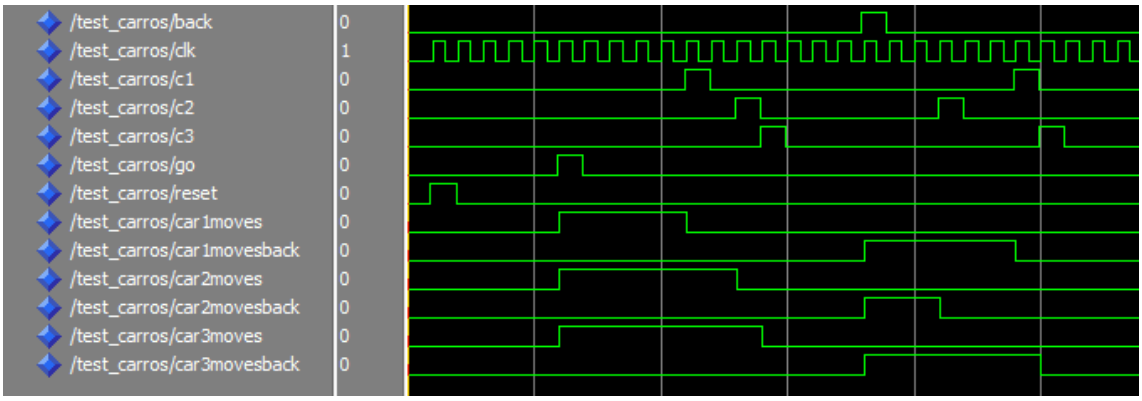


FIGURE 42 – SIMULATION OF THE 3 CARS EXAMPLE

The transition a1 and b1 corresponds to the signal c1. As for the transition a2 and b2 corresponds to the signal c2 and the transition a3 and b3 corresponds to the signal c3.

5.2.1. THE 3 CARS SYSTEM PARTITION

The next step of this exercise is to take the PNML of the 3 cars system and split it in three controllers so that each of the splits contains one model of the cars.

From Figure 31 (present in chapter 4) one can retrieve that car 1 is the main module. It is in the car 1 module that the GO, BACK commands are activated and verification of the ready and at the end places occurs. So in module 1 there are four more places and four more transitions.

In order for the global system to work properly there is a need to pass information between each of the modules. In this case there is only communication between module 1 and module 2 and module 1 and module 3. The communication is made in form of in events and out events:

- GO is active: event 1443 transmitted from module 1 to module 2 and 3;
- A2 is active: event 1515 transmitted from module 2 to module 1;
- A3 is active: event 1551 transmitted from module 3 to module 1;
- BACK is active: event 1805 transmitted from module 1 to module 2 and 3;
- B2 is active (second time): event 1587 transmitted from module 2 to module 1;
- B3 is active (second time): event 1569 transmitted from module 3 to module 1;

This makes a total of eight signals to interconnect between modules.

If all the modules were feed by the same clock, there were no problems in synchronization and it would never occur metastability failure. In Figure 43 is the simulation of this situation.

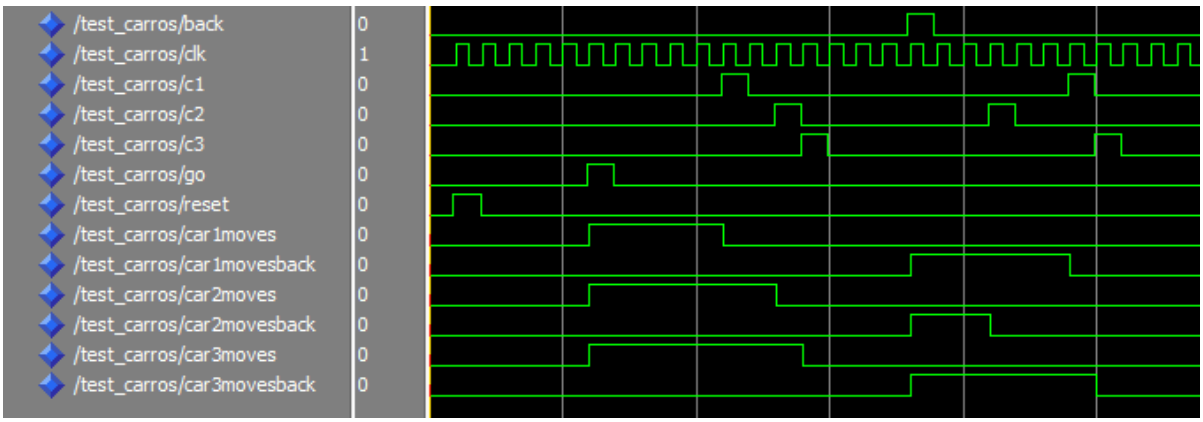


FIGURE 43 –SIMULATION OF THE 3 CARS SYSTEM FED BY THE SAME CLOCK

Now if all the modules are to have different clock speeds the system would become unreliable, as one can observe from the simulation in Figure 44:

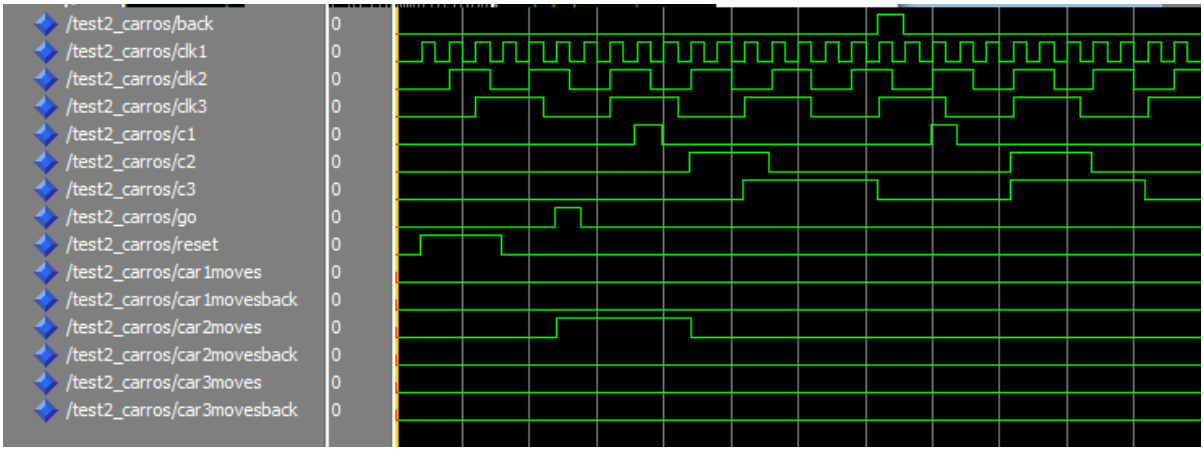


FIGURE 44 - SIMULATION OF THE 3 CARS SYSTEM FED BY THREE DIFFERENT CLOCKS, ONE FOR EACH CAR

In both Figure 43 and Figure 44 there are present signal C1, C2 and C3, which are not represented in the model of 3 cars system. These signals are associated the event signals A1, B1, A2, B2, A3, B3, respectively. So for C1 we have associated A1 and B1. The reason for this change is to be easier to implement in the FPGA, where the buttons and triggers are very limited. This change will not affect in any way the behavior of the model.

Module one (car one) was fed with a clock with 100ns high time and 100ns low time. Module two (car two) was fed with high time clock of 300ns and equal low time. Finally, the third with a 500ns high and low time. All clocks offset are of 100ns.

The first output is completely unpredictable. After the GO signal there should have been, at least, the output signal *car 1 moves,* but instead, there was a rise in the output *signal car 2 moves*. After this there is a complete system failure, which should prove irrecoverable without a system reset. This is because the execution semantic associated with signal channel was not preserved and output events were lost, as for the receiver counterpart is too slow to catch them.

Hence the need for the GALS solution in order to assure asynchronous connectivity between synchronous components.

5.2.2. 3 CARS GALS SYSTEM

To solve the synchronization problem we will apply to the three cars system our solution.

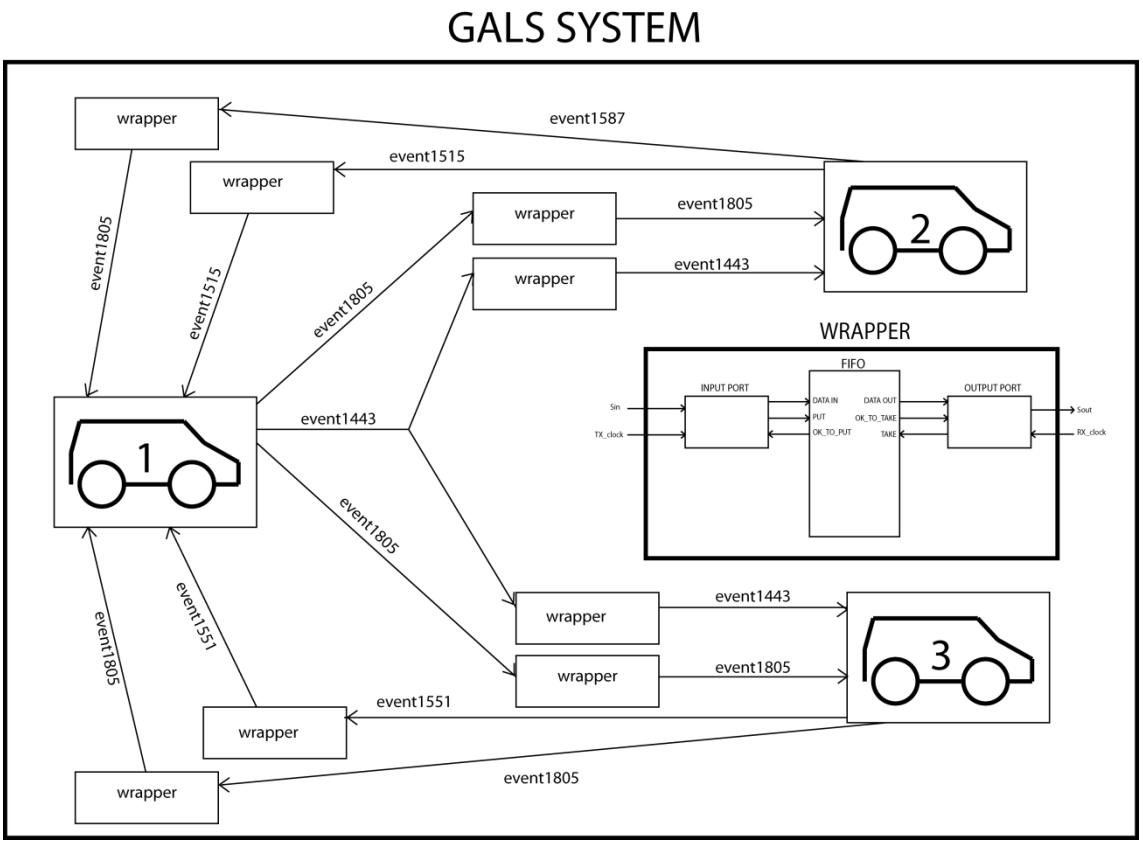


FIGURE 45 – 3 CARS GALS SYSTEM

For each of the modules signal being interchanged by the modules there will be a GALS wrapper (our developed asynchronous interface). As we have eight events being exchanged between modules it will give a total of 8 wrappers, one for each of the events.

Each of the wrapper will have to be fed with the transmitter and receivers clocks respectively for each of the events. For example: After the GO event is triggered, module one will release event *1443*, which as to be synchronized with module two and three clocks. So, for this *event, we* will need two wrappers. Each of the wrappers will be fed with module one clock (the transmitter's clock). And one of them with module's two clocks and the other with module's three clock, which, respectively, the receiver's clock.

Figure 45 shows how the GALS system will integrate in 3 cars example. In the figure only the event signals are represented and each of the events names were assigned by the Split tool[7].

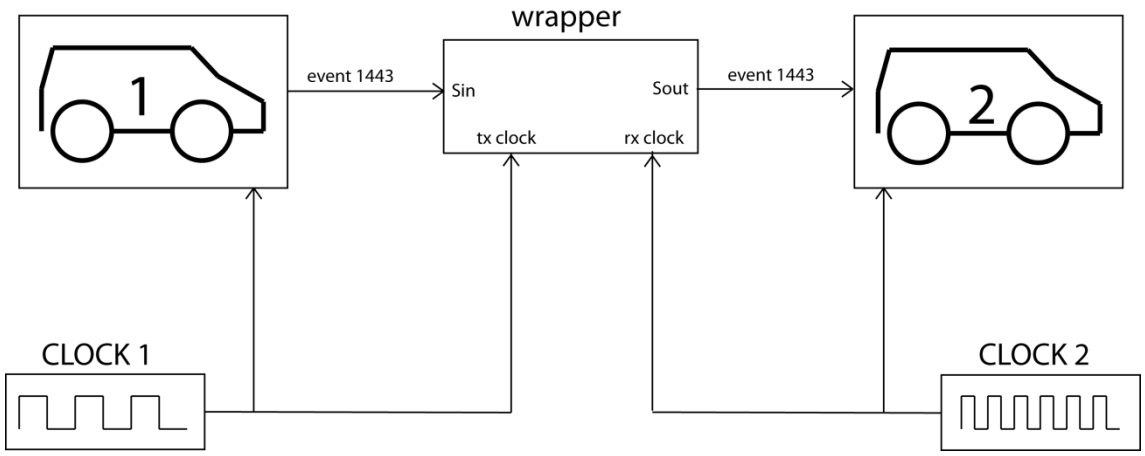


FIGURE 46- EVENT 1443 IN DETAIL

Figure 46 magnifies event 1443 transmission to better understand the wrappers inputs and outputs.

So for event 1443 there will be a total of three inputs into the wrapper and, only, one output:

- Inputs – event 1443, transmitter's clock, receiver's clock.
- Output- event 1443.

Car 1 is the transmitter and is fed by clock 1. The same clock will be used by wrapper 1 to synchronize the signal being sent by car 1.

When event 1443 is transmitted by car 1 it will enter wrapper 1 and exit wrapper 1 duly synchronized with the clock 2, the same clock that feeds car 2.

This situation is analogue to every other event being transmitted between modules.

Extent behavior information on the wrapper is found in chapter 4.

5.2.3. SIMULATION

We feed each car in the GALS system with the same three clocks:

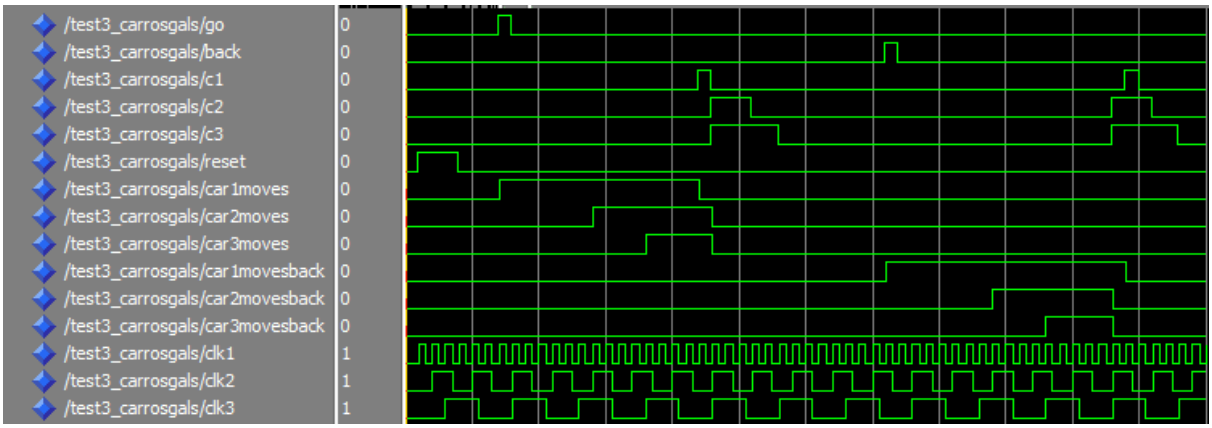


FIGURE 47 – SIMULATION OF THE 3 CARS GALS SYSTEM

Module one (car one) was fed with a clock with 100ns high time and 100ns low time. Module two (car two) was fed with high time clock of 300ns and equal low time. Finally, the third with a 500ns high and low time. All clocks offset are of 100ns.

The output of the system can be observed in Figure 47.

The difference in the outputs is visible noticed. After the GO signal the outputs from car 2 and 3 are not in the same clock due to synchronization delay. For each of the cars it takes two more of their own clocks for the output. The output from car 1 remains the same because the GO signal has its origin in car 1.

These effects are analogous to the BACK signal and C1, C2 and C3 signals.

In overall the system has become somewhat slower but stable, thus, achieving the objective of the GALS system.

5.2.4. IMPLEMENTATION

For a real world test the system was implemented in a Xilinx Spartan-3 FPGA and in the Xilinx Virtex_II pro.

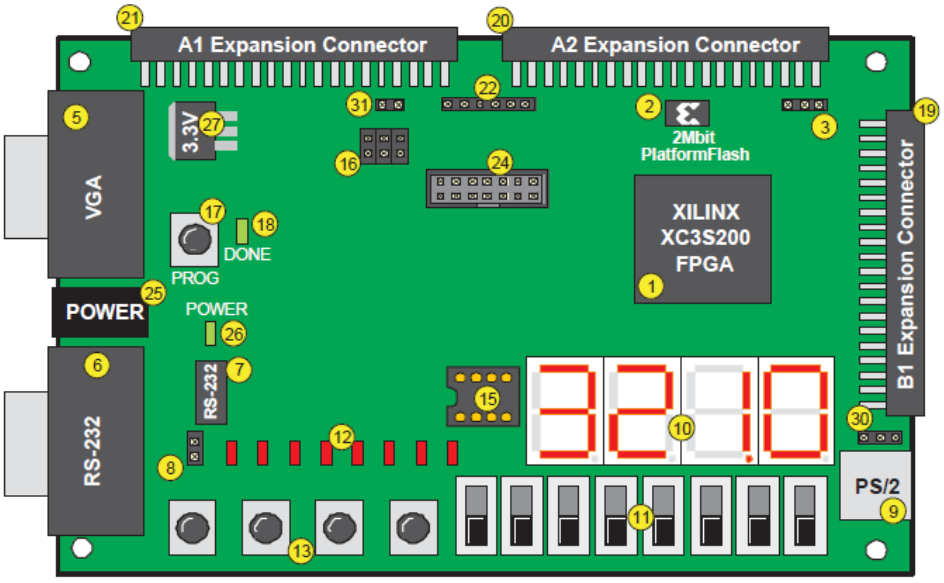


FIGURE 48 – FPGA SCHEME, *RETRIEVED FROM[31]*

For the Spartan-3 the VDHL GALS system was converted to a bit programming file through the Xilinx 6 software. For programming the FPGA the ADEPT software was used in conjunction with the bit file.

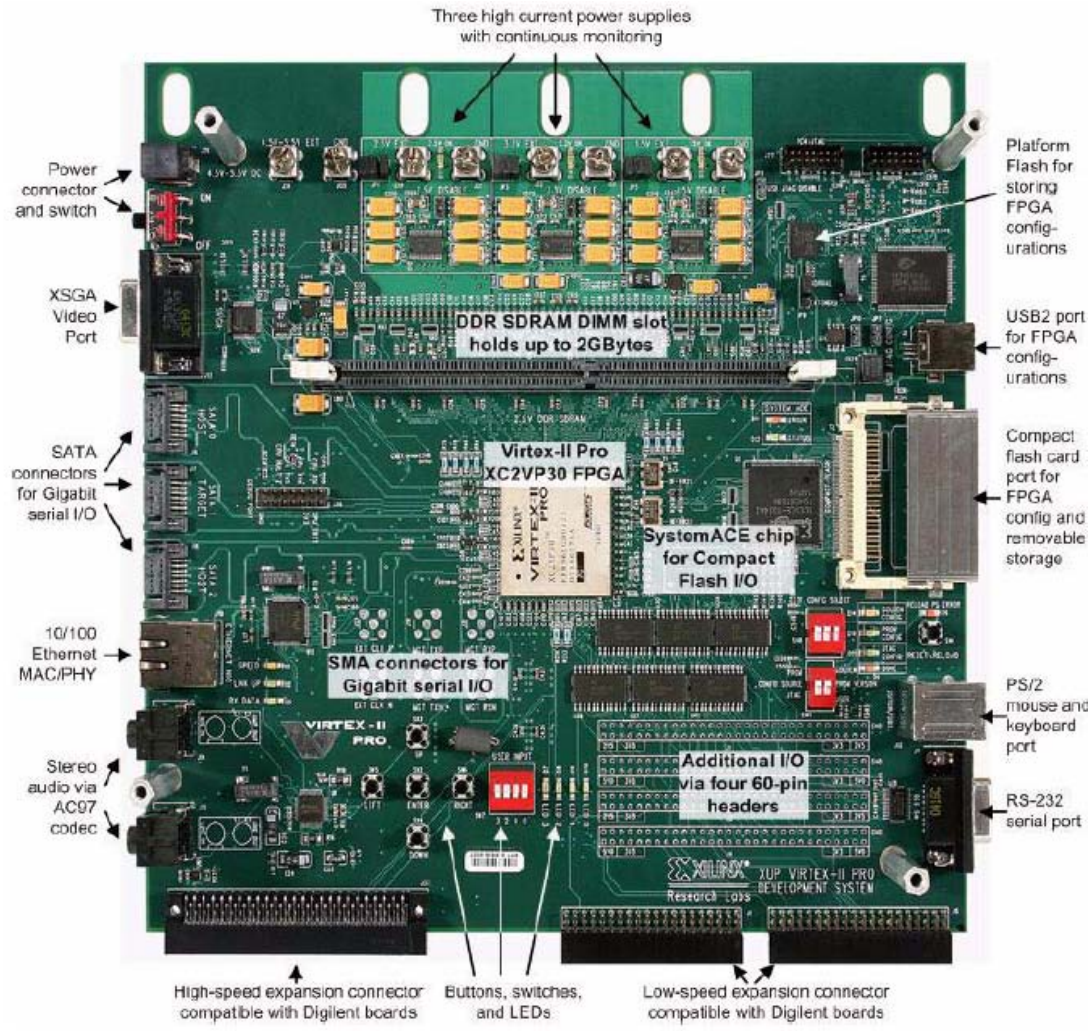


FIGURE 49 – VIRTEX-II PRO DEVELOPMENT SYSTEM BOARD PHOTO[32]

Another problem was that the FPGA only possessed one internal clock generator. For this test we needed two more.

The solution devised was to use the FPGA expansion ports to feed it with two more external clocks. So in total with had three clocks:

1. The 50 MHz FPGA internal oscillator.
2. The 10 MHz external oscillator.
3. The 24 MHz external oscillator.

The FPGA 50 MHz oscillator feed module one. The 10 MHz feed module two, and, 24 MHz feed module three. With this we achieved the necessary conditions to test the GALS system.

The switches on the FPGA were used for the GO, BACK, C1, C2, C3 and reset for the input signals. The LEDs were used to observe the behavior of the system. One LED for each output: Car 1 moves, Car 2 moves, Car 3 moves, Car 1 moves back, Car 2 moves back, Car 3 moves back.

For the Virtex_II Pro the implementation was done through the xilinx ISE 10.1 (it was the only software compatible with this board) with the help iMPATC tool. The Virtex possesses multiple clocks so the was no need for external clock implementation.

The real world test for Spatan 3 and Virtex-II pro proved to be a success. Both of them behave exactly has expected.

5.2.5. POWER, HEAT AND SIZE EVALUATION

Closing this example we are now going to compare the heat generated and power consumption of the implementation with and without the GALS solution, for both platforms.

5.2.5.1. SPARTAN-3

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.2		
Dynamic		0.00	0.00
Quiescent		10.18	12.21
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		10.00	25.00
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		0.00	0.00
Total Power			37.21
Startup Current		0.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.2		
Dynamic		0.00	0.00
Quiescent		75.00	90.00
Vccaux	3.3		
Dynamic		0.00	0.00
Quiescent		100.00	330.00
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		2.00	5.00
Total Power			425.00
Startup Current		500.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00
Confidence Level			Inaccurate

FIGURE 50 – SIDE BY SIDE POWER COMPARISON OF THE SPARTAN-3 3 CARS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

Figure 50 is the side by side power consumption for the example of the 3 cars. This information was taken from the XPower tool from Xilinx.

We can see an increase from 37.21 mW to 425 mW, a difference of 387.79 mW. That represents an increase of power in 1042%.

Ambient Temperature (°C)	25
Junction Temperature (°C)	26.15
Case Temperature (°C)	25.80
Part Type	Commercial
Airflow (LFM)	0
Package	ft256
Total Power (mW)	37.21

Ambient Temperature (°C)	25
Junction Temperature (°C)	39.24
Case Temperature (°C)	36.90
Part Type	Commercial
Airflow (LFM)	0
Package	ft256
Total Power (mW)	425.00

FIGURE 51 – SIDE BY SIDE HEAT COMPARISON OF THE SPARTAN-3 3 CARS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

Has for the heat generated, we see a boost from the case temperature from 25.80ºC to 36.90ºC. An 11.1ºC increase that represents a 43% increase in heat form.

Another aspect is the overhead added by the inclusion of wrappers. Figure 52 shows the device utilization summary for the 3 cars system example with and without GALS.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	16	3,840	1%	
Number of 4 input LUTs	18	3,840	1%	
Logic Distribution				
Number of occupied Slices	9	1,920	1%	
Number of Slices containing only related logic	9	9	100%	
Number of Slices containing unrelated logic	0	9	0%	
Total Number of 4 input LUTs	18	3,840	1%	
Number of bonded IOBs	13	173	7%	
Number of GCLKs	1	8	12%	
Total equivalent gate count for design	239			
Additional JTAG gate count for IOBs	624			

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	56	3,840	1%	
Number used as Flip Flops	40			
Number used as Latches	16			
Number of 4 input LUTs	100	3,840	2%	
Logic Distribution				
Number of occupied Slices	69	1,920	3%	
Number of Slices containing only related logic	69	69	100%	
Number of Slices containing unrelated logic	0	69	0%	
Total Number of 4 input LUTs	100	3,840	2%	
Number of bonded IOBs	15	173	8%	
Number of GCLKs	3	8	37%	
Total equivalent gate count for design	1,069			
Additional JTAG gate count for IOBs	720			

FIGURE 52 - DEVICE UTILIZATION SUMMARY OF THE SPARTAN-3 WITHOUT GALS(ABOVE) AND WITH GALS(BELOW) FOR CARS EXAMPLE

As one can observe from Figure 52 for the example of the 3 cars system the Total Number of Slices increases from 16 to 56.

The 4 input LUTs increase from 18 to 100.

As for Logic Distribution the number of occupied slices increases from 9 to 69.

The number of bonded IOBs goes from 13 to 15.

The number of GCLKs increases from 1 to 3.

And the Total equivalent gate count for design goes from 239 to 1069. That represents a 347% size increase.

Although every device percentage increases in enormous proportions, one should take in consideration that the 3 cars example is very elementary and any change will be felt in a dramatic way.

5.2.5.2. VIRTEX-II PRO

From Figure 53 and Figure 54 we can see that for the Virtex-II pro there are no changes in terms of Power and Heat when the system is operating without GALS and when it is operating with GALS.

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.5		
Dynamic		0.00	0.00
Quiescent		250.00	375.00
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		167.00	417.50
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		2.00	5.00
Total Power			797.50
Startup Current		500.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00
Confidence Level			Inaccurate

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.5		
Dynamic		0.00	0.00
Quiescent		250.00	375.00
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		167.00	417.50
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		2.00	5.00
Total Power			797.50
Startup Current (mA)		500.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00
Confidence Level			Inaccurate

FIGURE 53- SIDE BY SIDE POWER COMPARISON OF THE VIRTEX-II PRO 3 CARS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

The total consumption for both configurations is 797.50 mW. This is due to the power analyzer tool to estimate the leakage current of the gates.

There is change, however, in what concerns device utilization.

Ambient Temperature (°C)	25
Junction Temperature (°C)	25.00
Case Temperature (°C)	25.00
Part Type	Commercial
Airflow (LFM)	0
Package	ff896
Total Power (mW)	797.50

Ambient Temperature (°C)	25
Junction Temperature (°C)	25.00
Case Temperature (°C)	25.00
Part Type	Commercial
Airflow (LFM)	0
Package	ff896
Total Power (mW)	797.50

FIGURE 54 – SIDE BY SIDE HEAT COMPARISON OF THE VIRTEX-II PRO 3 CARS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

Figure 55 presents a detailed analysis of the number of slices, flip-flops, LUTs, IOBs, and GCLKs that are occupied on the device without the GALS system and with the GALS system.

There is an increase from 10 slices to 66 slices.

From 16 flip-flops to 56.

From 18 LUTs to 112.

And a decrease from 12 IOBs to 9.

And 3 GCKLs to 2.

```
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

 Number of Slices:                       10  out of  13696     0%
 Number of Slice Flip Flops:             16  out of  27392     0%
 Number of 4 input LUTs:                 18  out of  27392     0%
 Number of bonded IOBs:                  12  out of    556     2%
 Number of GCLKs:                         3  out of     16    18%


=========================================================================
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

 Number of Slices:                       66  out of  13696     0%
 Number of Slice Flip Flops:             56  out of  27392     0%
 Number of 4 input LUTs:                112  out of  27392     0%
 Number of bonded IOBs:                   9  out of    556     1%
 Number of GCLKs:                         2  out of     16    12%


=========================================================================
```

FIGURE 55 – DEVICE UTILIZATION SUMMARY OF THE VIRTEX-II PRO WITHOUT GALS(ABOVE) AND WITH GALS(BELOW) FOR CARS EXAMPLE

5.3. EXAMPLE 2: MANUFACTURE CELLS

This example is about a manufacturing cell controller, containing robots and movements carpets as illustrated in Figure 56. In the figure is represented a controller with 3 cells. In this practical example we will use 4 cells.

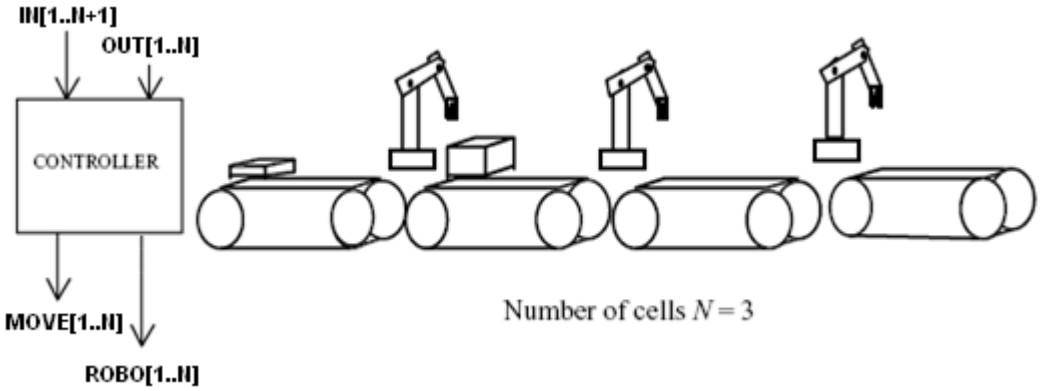


FIGURE 56 – 3 CELLS MANUFACTURE SYSTEM, *RETRIEVED FROM[33]*

The Petri Net model can be observed in Figure 57.

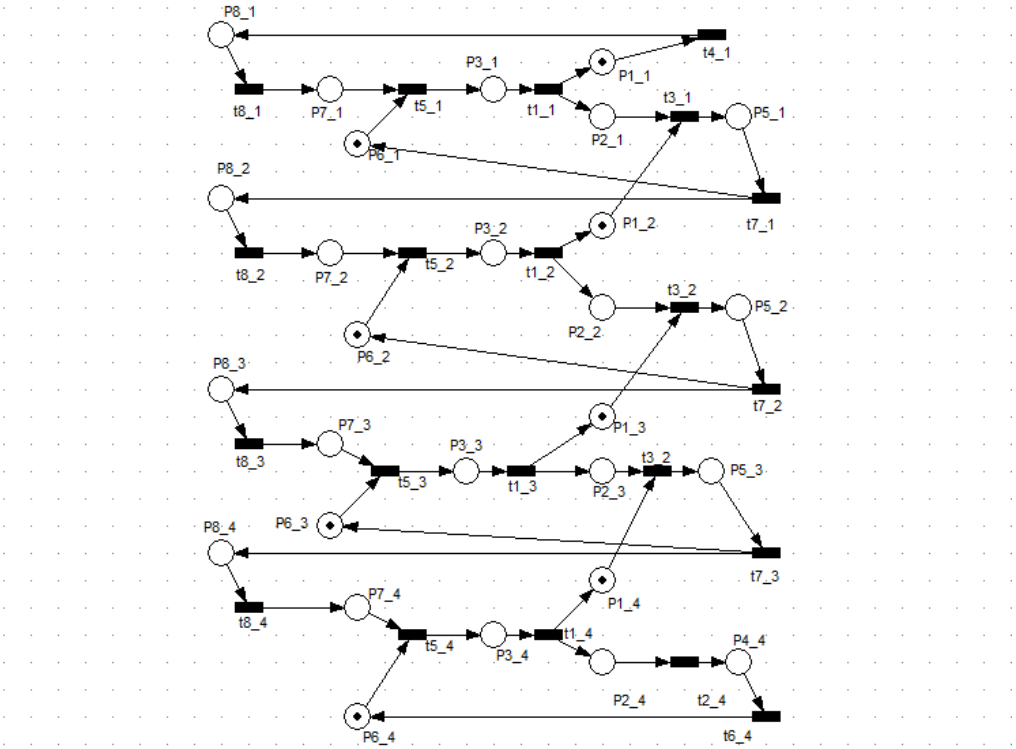


FIGURE 57- 4 CELLS MANUFACTURES SYSTEM PETRI NET MODEL

In order to be scalable, the system is divided into cells and may be added as desired without the need for the system to be adjusted. Each cell has two sensors ("INx" and

"OUTx") and two actuators (Movex "and" ROBOx "), in which the "x" symbolizes the ID of the cell sensor that indicates the end of transformation process and, so, the part can be removed from the assembly line. When INx is active the carpet starts to move (Movex) until sensor OUTx is activated and the carpet stops, at that time ROBOx is active and the piece is changed from one cell to the other.

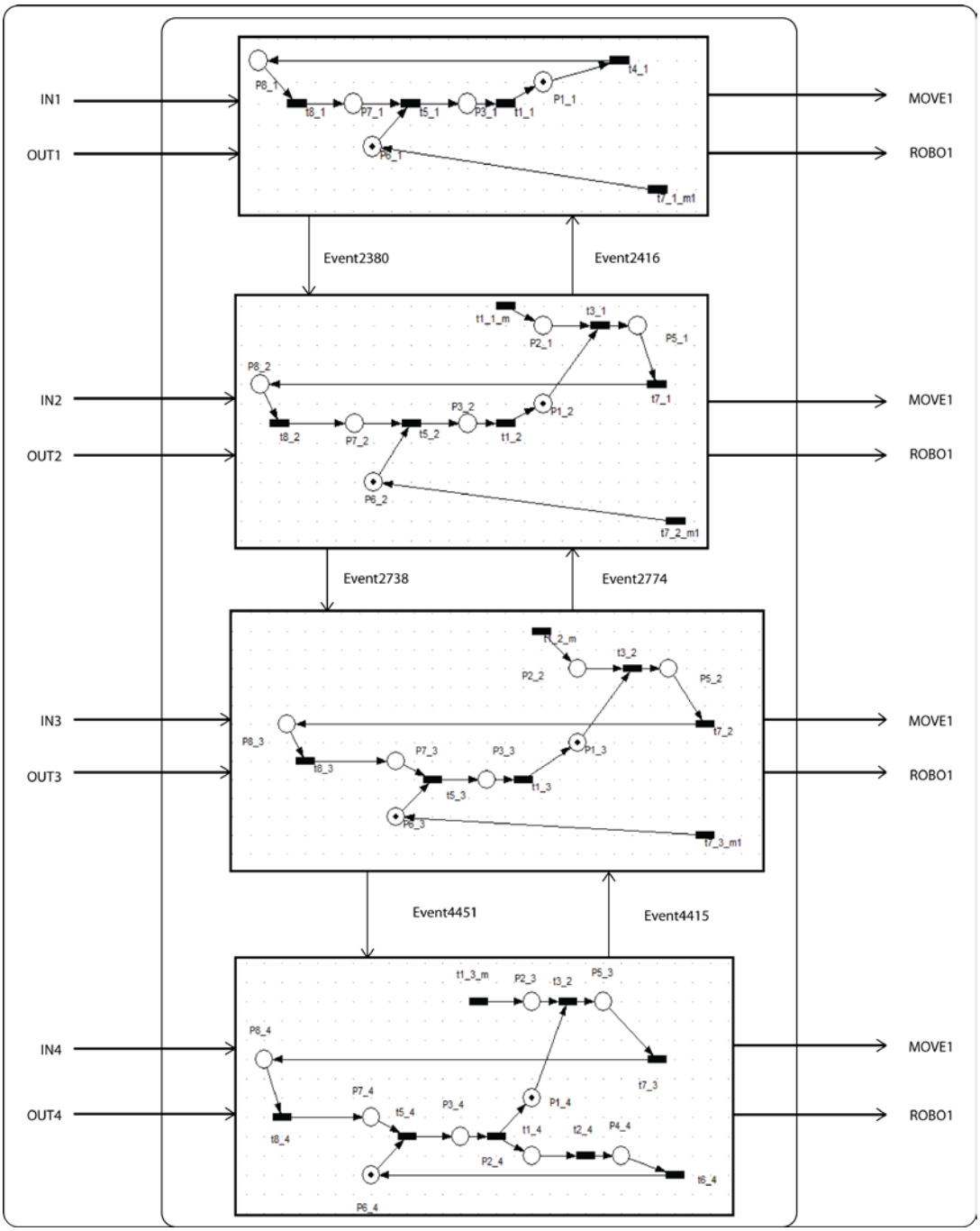


FIGURE 58 – MANUFACTURE CELL 4 MODULES WITH SIGNALS REPRESENTATION

The input signals of this model are:

- IN1;
- IN2;
- IN3;
- IN4;
- OUT1;
- OUT2;
- OUT3;
- OUT4;

The output signals are:

- MOVE1;
- MOVE2;
- MOVE3;
- MOVE4;
- ROBO1;
- ROBO2;
- ROBO3;
- ROBO4;

Figure 58 is the representation of all the signals as the result of applying the SPLIT tool [7] to the Manufacture Cells model.

As a result we have six new event signals:

- event2380;
- event 2416;
- event 2738;
- event 2774;
- event 4451;
- event 4415;

As the first example, this one, also, has no handshake signals to pass the information.

The two examples are in line with what most LSI interconnections issues will be, thus, the solution being developed will be, in all terms, a general solution.

The simulation of the system, after converting the IOPT model to VHDL, can be observed in Figure 59:

The system was fed with only one clock of 100 ns high and low time.

The part will naturally propagate from cell to cell, with new parts being added as soon as one cell becomes free.

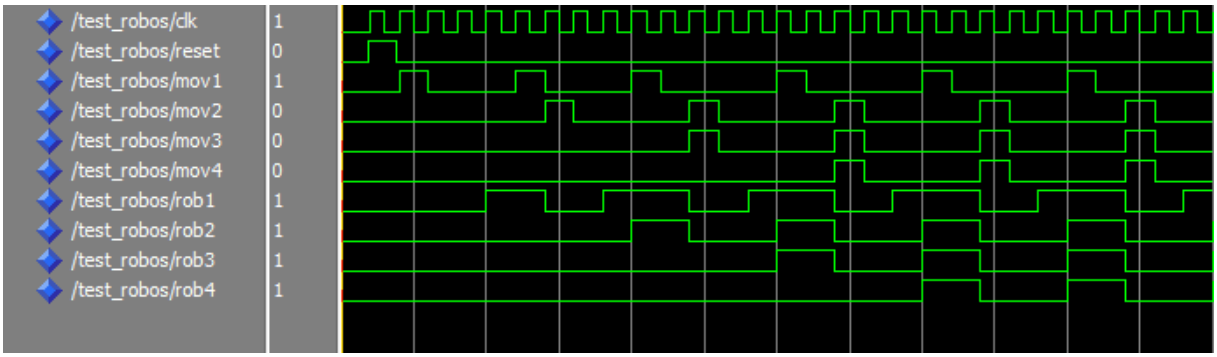


FIGURE 59- SIMULATION OF THE 4 CELLS MANUFACTURES SYSTEM

5.3.1. MANUFACTURE CELL PARTITION

This next step comprises of splitting the Petri Net [3-5] model into four, one for each cell.

For each of sub-model it will be generated a PNML file and, later, converted do VHDL.

The split model can be observed in Figure 58.

The waveform simulation of the system fed by one clock (each of the modules are fed by the same clock, hence no ground for metastability to arise) is shown in Figure 60. Has one can observe there is no difference between the original and the split model. The clock used was also 100 ns high and low time.

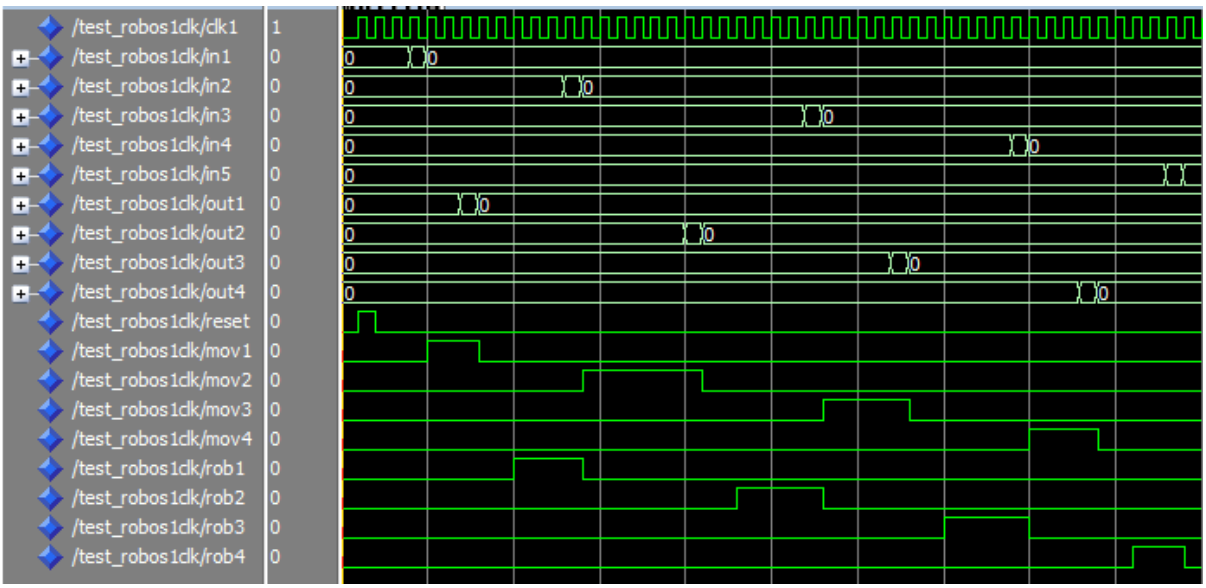


FIGURE 60 - SIMULATION OF THE 4 CELLS MANUFACTURES SYSTEM FED WITH SAME CLOCK

So the next step was to feed each of the modules with a different clock:

- Clock 1 – 100 ns high time, 100 ns low time, 100 ns offset;
- Clock 2 – 200 ns high time, 200 ns low time, 120 ns offset;
- Clock 3 – 300 ns high time, 300 ns low time, 130 ns offset;
- Clock 4 – 400 ns high time, 400 ns low time, 140 ns offset;

The behavior of the system can be observed in the waveform generated in the Modelsim software, presented in Figure 61.

From Figure 61, we can come to conclusion that system encountered synchronization problems. It failed altogether even in the first instants of the simulation. So the solution to be applied is the introduction of Wrappers between signals to achieve the needed synchronization.

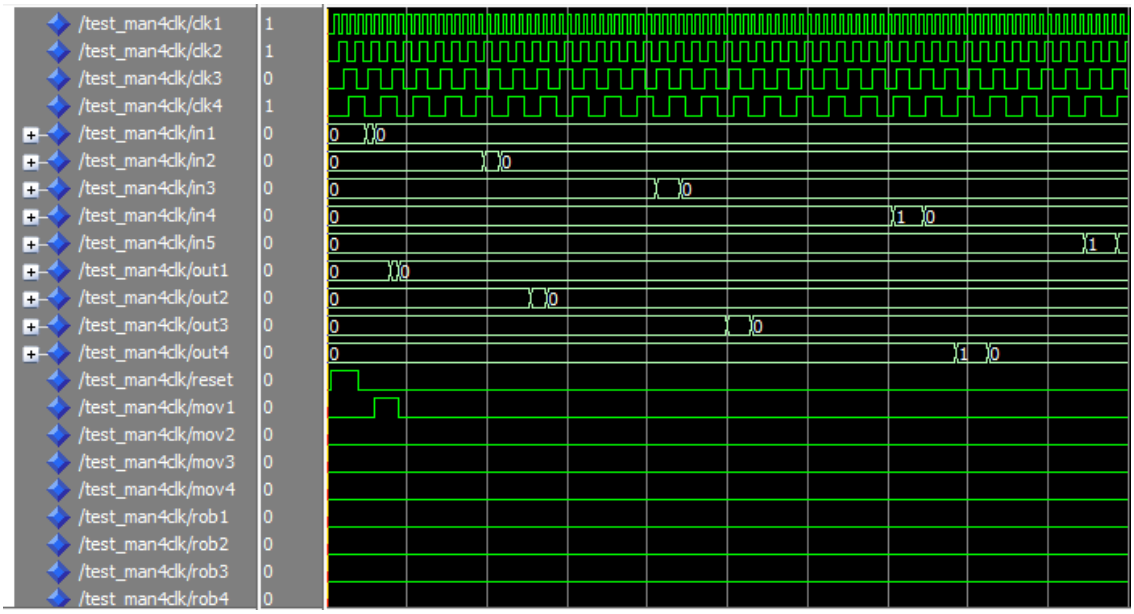


FIGURE 61 - SIMULATION OF THE 4 CELLS MANUFACTURES SYSTEM FED WITH DIFFERENT CLOCKS

5.3.2. MANUFACTURE CELLS GALS SYSTEM

So we have six signals being commuted between the four modules. With the introduction of the Wrapper will have twelve signals:

- Wrapper 1
  - input signal is inevent2380;
  - output signal is outevent2380;
  - receiver's clock is clock 2;
  - transmitter's clock is clock 1;
- Wrapper 2
  - input signal is inevent2416;
  - output signal is outevent2416;
  - receiver's clock is clock 1;
  - transmitter's clock is clock 2;
- Wrapper 3
  - input signal is inevent2738;

- output signal is outevent2738;
- receiver's clock is clock 3;
- transmitter's clock is clock 2;

- Wrapper 4
  - input signal is inevent2774 ;
  - output signal is outevent2774;
  - receiver's clock is clock 2;
  - transmitter's clock is clock 3;

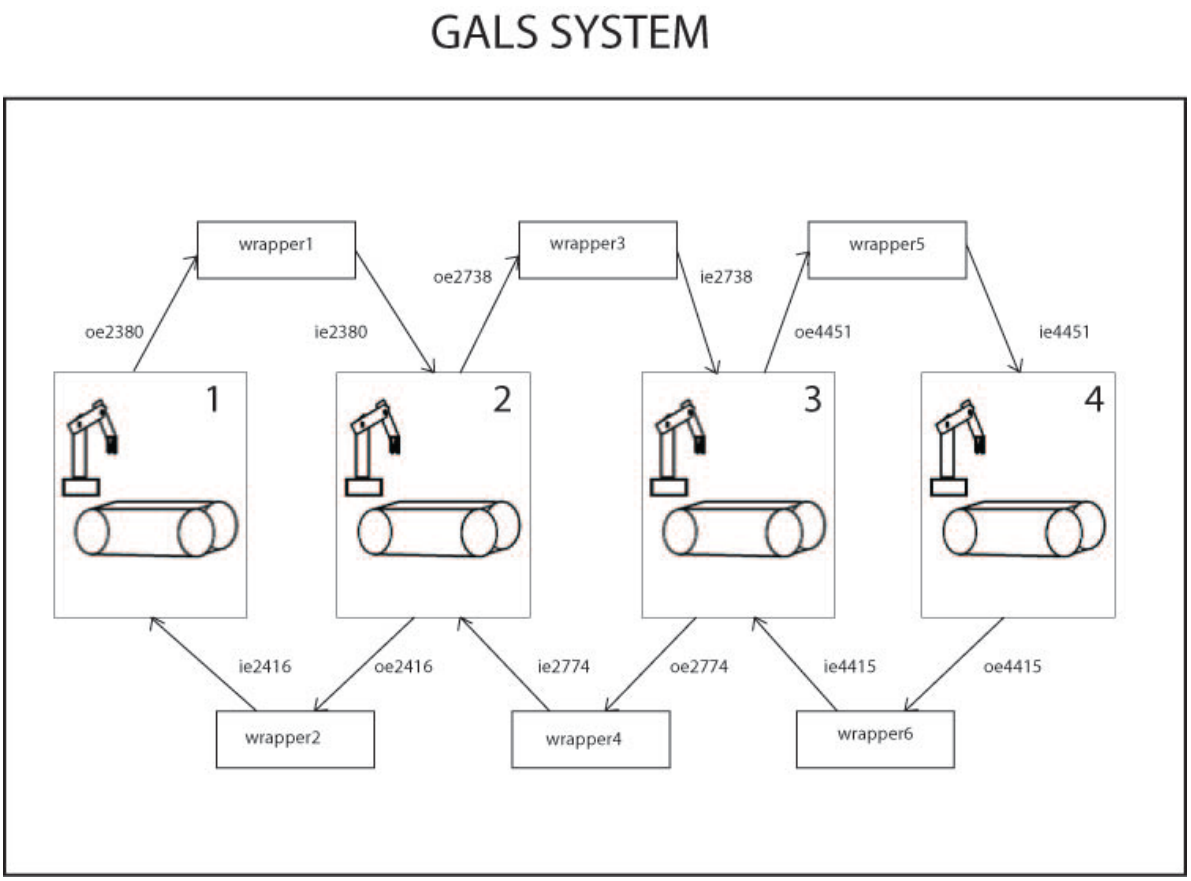


FIGURE 62 – MANUFACTURE CELLS GALS SYSTEM SIGNAL EXCHANGE

- Wrapper 5
  - Input signal is inevent4451;
  - output signal is outevent4451;
  - receiver's clock is clock 4;

- transmitter's clock is clock 3;

- Wrapper 6
  - input signal is inevent4415;
  - output signal is outevent4415;
  - receiver's clock is clock 3;
  - transmitter's clock is clock 4;

5.3.3. SIMULATION

So for the simulation we will have the same four clocks:

- Clock 1 – 100 ns high time, 100 ns low time, 100 ns offset;
- Clock 2 – 200 ns high time, 200 ns low time, 120 ns offset;
- Clock 3 – 300 ns high time, 300 ns low time, 130 ns offset;
- Clock 4 – 400 ns high time, 400 ns low time, 140 ns offset;

Clock 1 will feed module 1, Wrapper 1 and Wrapper 2.

Clock 2 will feed module 2, Wrapper 1, Wrapper 2, Wrapper 3 and Wrapper 4.

Clock 3 will feed module 3, Wrapper 3, Wrapper 4, Wrapper 5 and Wrapper 6.

Clock 4 will feed module 4, Wrapper 5 and Wrapper 6.

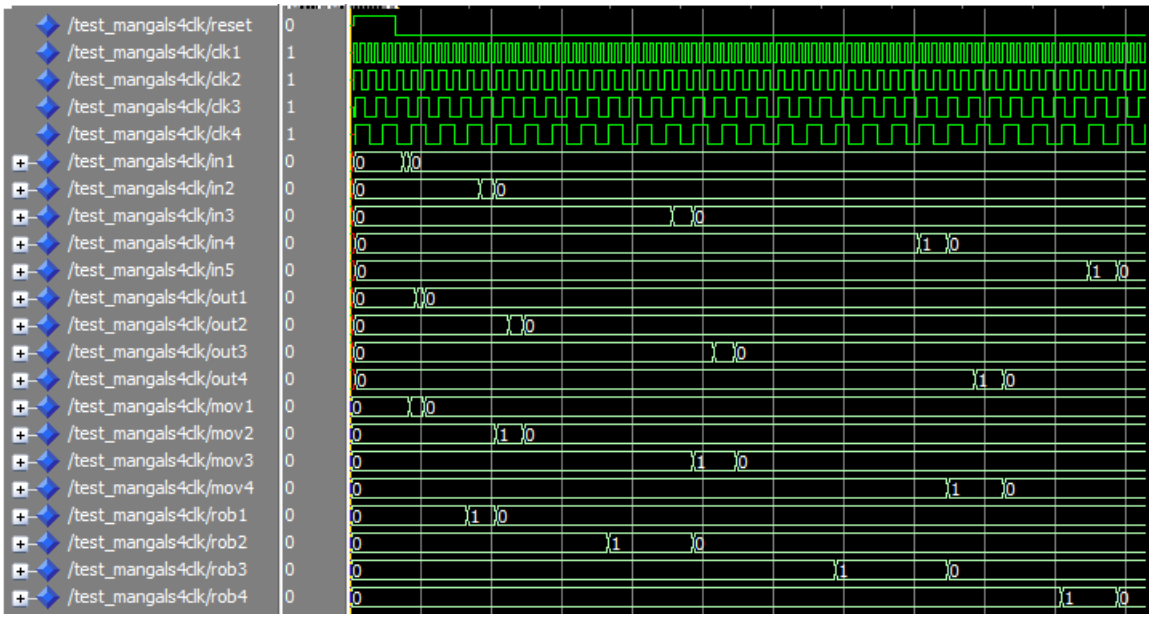


FIGURE 63- SIMULATION FOR MANUFACTURE GALS SYSTEM

The simulation of the system with GALS can be observed in Figure 63.

As observed from the simulation the system although performing with diversity of clocks maintains its coherence, avoiding metastability or the lack of synchronization altogether, and so achieving the finality of the GALS solution.

There is, although, an avoidable drawback in circuit speed as the result from the necessary synchronization between different synchronous islands.

5.3.4. IMPLEMENTATION

Has for example 1, example 2 was also implemented in a Xilinx Spartan-3 FPGA, for real world testing

The Manufacture Cell with GALS systems VDHL was converted to a bit programming file through the Xilinx 6 software. For programming the FPGA the ADEPT software was used in conjunction with the bit file.

The same solution was adopted to solve the problem of the FPGA single clock generator.

So we have the FPGA expansion ports to fed with two more external clocks. So in total with had three clocks:

1. The 50 MHz FPGA internal oscillator.
2. The 10 MHz external oscillator.
3. The 24 MHz external oscillator.

The FPGA 50 MHz oscillator feed module one. The 10 MHz feed module two. The 24 MHz feed module three, and module four was fed with the same 10 MHz of module 2

but with 180º phase difference (Using an inverter). With this we achieved the necessary conditions to test the GALS system.

The system behaved has expected and proved to be a success.

5.3.5. POWER, HEAT AND SIZE EVALUATION

5.3.5.1. SPARTAN-3

Closing this example we are now going to compare the heat generated and power consumption of the implementation with and without the GALS solution.

In Figure 64 is the side by side power consumption for the example of the 3 cars.

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.2		
Dynamic		0.00	0.00
Quiescent		10.18	12.21
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		10.00	25.00
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		0.00	0.00
Total Power			37.21
Startup Current (m		0.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.2		
Dynamic		0.00	0.00
Quiescent		75.00	90.00
Vccaux	3.3		
Dynamic		0.00	0.00
Quiescent		100.00	330.00
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		2.00	5.00
Total Power			425.00
Startup Current		500.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00
Confidence Level			Inaccurate

FIGURE 64 – SIDE BY SIDE POWER COMPARISON OF THE SPARTAN-3 MANUFACTURE CELLS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

We can see an increase from 37.21 mW to 425 mw, a difference of 387.79 mW. That represents an increase of power in 1042%.

Ambient Temperature (°C)	25
Junction Temperature (°C)	26.15
Case Temperature (°C)	25.80
Part Type	Commercial
Airflow (LFM)	0
Package	ft256
Total Power (mW)	37.21

Ambient Temperature (°C)	25
Junction Temperature (°C)	39.24
Case Temperature (°C)	36.90
Part Type	Commercial
Airflow (LFM)	0
Package	ft256
Total Power (mW)	425.00

FIGURE 65 – SIDE BY SIDE HEAT COMPARISON OF THE SPARTAN-3 MANUFACTURE CELLS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

As for the heat generated, we see a boost from the case temperature from 25.80ºC to 36.90ºC. 11.1ºC more, which represents a 43% raise in heat form.

In relation of the size occupied in the FPGA it differs from the example with and without the GALS solution.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	28	3,840	1%	
Number of 4 input LUTs	25	3,840	1%	
Logic Distribution				
Number of occupied Slices	22	1,920	1%	
Number of Slices containing only related logic	22	22	100%	
Number of Slices containing unrelated logic	0	22	0%	
Total Number of 4 input LUTs	25	3,840	1%	
Number of bonded IOBs	10	173	5%	
Number of GCLKs	1	8	12%	
Total equivalent gate count for design	377			
Additional JTAG gate count for IOBs	480			

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	58	3,840	1%	
Number used as Flip Flops	46			
Number used as Latches	12			
Number of 4 input LUTs	97	3,840	2%	
Logic Distribution				
Number of occupied Slices	65	1,920	3%	
Number of Slices containing only related logic	65	65	100%	
Number of Slices containing unrelated logic	0	65	0%	
Total Number of 4 input LUTs	97	3,840	2%	
Number of bonded IOBs	13	173	7%	
Number of GCLKs	4	8	50%	
Total equivalent gate count for design	1,058			
Additional JTAG gate count for IOBs	624			

FIGURE 66 - DEVICE UTILIZATION SUMMARY OF THE SPARTAN-3 WITHOUT GALS(ABOVE) AND WITH GALS(BELOW) FOR MANUFACTURE EXAMPLE

The number of slice flip flop goes from 28 to 58. That is 207% increase.

For the total equivalent gate count for design we see an increase from 377 to 1058, representing a 180% increase in overhead.

5.3.5.2. VIRTEX-II PRO

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.5		
Dynamic		0.00	0.00
Quiescent		250.00	375.00
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		167.00	417.50
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		2.00	5.00
Total Power			797.50
Startup Current		500.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00
Confidence Level			Inaccurate

	Voltage (V)	Current (mA)	Power (mW)
Vccint	1.5		
Dynamic		0.00	0.00
Quiescent		250.00	375.00
Vccaux	2.5		
Dynamic		0.00	0.00
Quiescent		167.00	417.50
Vcco25	2.5		
Dynamic		0.00	0.00
Quiescent		2.00	5.00
Total Power			797.50
Startup Current		500.00	
Battery Capacity (mA Hours)			0.00
Battery Life (Hours)			0.00
Confidence Level			Inaccurate

FIGURE 67 – SIDE BY SIDE POWER COMPARISON OF THE VIRTEX-II PRO MANUFACTURE CELLS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

As with what succeded with the 3 car example we can see that for the Virtex-II pro there are no changes in terms of Power and Heat when the system is operating without GALS and when it is operating with GALS. The total consumption for both configurations is 797.50 mW. The heat produce by both systems is of 25º C. The comparisons for the Power and Heat of the system without and with GALS are in Figure 67 and Figure 68.

Ambient Temperature (°C)	25
Junction Temperature (°C)	25.00
Case Temperature (°C)	25.00
Part Type	Commercial
Airflow (LFM)	0
Package	ff896
Total Power (mW)	797.50

Ambient Temperature (°C)	25
Junction Temperature (°C)	25.00
Case Temperature (°C)	25.00
Part Type	Commercial
Airflow (LFM)	0
Package	ff896
Total Power (mW)	797.50

FIGURE 68 – SIDE BY SIDE HEAT COMPARISON OF THE VIRTEX-II PRO MANUFACTURE CELLS EXAMPLE. LEFT - WITHOUT GALS, RIGHT - WITH GALS

As in what concerns device utilization Figure 69 presents a detailed analysis of the number of slices, flip-flops, LUTs, IOBs, and GCLKs that are occupied on the device without the GALS system and with the GALS system.

There is an increase from 21 slices to 62 slices.

From 37 flip-flops to 67.

From 30 LUTs to 106.

And a decrease from 18 IOBs to 17.

And 4 GCKLs to 2.

```
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

 Number of Slices:                      21  out of  13696     0%
 Number of Slice Flip Flops:            37  out of  27392     0%
 Number of 4 input LUTs:                30  out of  27392     0%
 Number of bonded IOBs:                 18  out of    556     3%
 Number of GCLKs:                        4  out of     16    25%


=========================================================================
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

 Number of Slices:                      62  out of  13696     0%
 Number of Slice Flip Flops:            67  out of  27392     0%
 Number of 4 input LUTs:               106  out of  27392     0%
 Number of bonded IOBs:                 17  out of    556     3%
 Number of GCLKs:                        2  out of     16    12%


=========================================================================
```

FIGURE 69 – DEVICE UTILIZATION SUMMARY OF THE VIRTEX-II PRO WITHOUT GALS(ABOVE) AND WITH GALS(BELOW) FOR MANUFACTURE EXAMPLE

5.4. GENERAL PLATFORM COMPARISON

In this brief sub-chapter a table will be presented comparing the different power, heat and device utilization for both examples with and without GALS system in both platforms.

3 CARS EXAMPLE

	SPARTAN-3				VIRTEX-II pro			
	without GALS	with GALS	variation		without GALS	with GALS	variaton	
TOTAL POWER (mW)	37,21	425	387,79	1042%	797,5	797,5	0	0%
CASE TEMPERATURE (C)	25,8	36,9	11,1	43%	25	25	0	0%
SLICES	9	69	60	667%	10	66	56	560%
FLIP-FLOPS	16	40	24	150%	16	56	40	250%
LUTs	18	100	82	456%	18	112	94	522%
IOBs	13	15	2	15%	12	9	-3	-25%
GCLKs	1	3	2	200%	3	2	-1	-33%

MANUFACTURE CELLS

	SPARTAN-3				VIRTEX-II pro			
	without GALS	with GALS	variation		without GALS	with GALS	variation	
TOTAL POWER (mW)	37,21	425	462,21	1042%	797,5	797,5	0	0%
CASE TEMPERATURE (C)	25,8	36,9	62,7	43%	25	25	0	0%
SLICES	22	65	87	195%	21	62	41	195%
FLIP-FLOPS	28	46	74	64%	37	67	30	81%
LUTs	25	97	122	288%	30	106	76	253%
IOBs	10	13	23	30%	18	17	-1	-6%
GCLKs	1	4	5	300%	4	2	-2	-50%

FIGURE 70 – GENERAL COMPARISON OF BOTH PLATFORMS FOR BOTH EXAMPLES

In overall the Virtex-II pro has a greater general consumption in terms of resources over the Spartan-3.

It consumes 372,5 mW more than the Spartan-3 with GALS system and 760,29 mW more without GALS system. That represents a big gap between both platforms.

Has for the slices, flip-flops, LUTs,IOBs and GCLKs the Virtex-II always has a bigger usage of this elements over the Spartan-3.

The only aspect the Virtex-II wins over the Spartan-3 is in the case temperature. It is 0.80 ºC less without GALS system and 11, 90 º C less with GALS system.

6. CONCLUSION

The first conclusion one can withdraw is that the GALS solution can be implemented with success, even in a commercial FPGA. It can overcome the frontier between asynchronous and synchronous time domains. This, of course, does not come without a price. The inclusion of the GALS Wrapper between each signal that needs synchronization will increase the circuit size and add an unwelcome delay, resulting in a decreased circuit speed.

This work as in some aspect extended the Asynchronous Interface, which served was the basis for the solution implemented. The Asynchronous Interface was dependent of handshake signals to properly work and it was impossible to implement it has a final solution, if the Transmitter and Receiver were both in synchronous time domain. Hence the necessity to adapt the Asynchronous Interface with the inclusion of the input and output ports. With those modifications the need for handshake signal were addressed only for the communication between the ports and the FIFO buffer.

It is now fairly easy to adapt this Interface to whichever time domain. For example: a mix of asynchronous Transmitter and synchronous Receiver, or vice versa.

Another aspect is the overhead added by the inclusion of wrappers

And the Total equivalent gate count for design goes from 239 to 1069. That represents a 347% increase in overhead.

The same pattern can be observed for the manufacture cells example:

Also for the total equivalent gate count for design we see an increase from 377 to 1058, representing a 180% increase in overhead.

Has for the heat and power increments they were same for both examples: A power increment of 1042% and heat rise of 43%.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	5	3,840	1%	
Number used as Flip Flops	3			
Number used as Latches	2			
Number of 4 input LUTs	10	3,840	1%	
Logic Distribution				
Number of occupied Slices	7	1,920	1%	
Number of Slices containing only related logic	7	7	100%	
Number of Slices containing unrelated logic	0	7	0%	
Total Number of 4 input LUTs	10	3,840	1%	
Number of bonded IOBs	5	173	2%	
Number of GCLKs	2	8	25%	
Total equivalent gate count for design	106			
Additional JTAG gate count for IOBs	240			

FIGURE 71 – DEVICE UTILIZATION FOR ONE WRAPPER

It is of the utmost importance to refer that these huge increments in size, power and heat in relation to the examples without GALS is due to the fact that the examples implemented are elementary. By themselves they utilize very little resources from the FPGA. Using a more complex example then this so great impact would not be so visible. The device utilization for only one wrapper can be seen in Figure 71, which is very low indeed.

So in overall if a given project has to be a distributed solution and different time domains have to interact, this solution will accomplish the job perfectly.

There will be a slight reduction in circuit speed and an increase in circuit size, power and heat, but the advantages that one can withdraw from this solution surpass greatly its own flaws.

7. BIBLIOGRAPHY

[1] D. M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems." vol. Ph.D.: Stanford University, 1984.

[2] R. Nunes, L. Gomes, and J. P. Barros, "A graphical editor for the input-output place-transition petri net class," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, 2007, pp. 788-791.

[3] R. Wolfgang, *Petri nets: an introduction*: Springer-Verlag New York, Inc., 1985.

[4] G. Claude and V. Rudiger, *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*: Springer-Verlag New York, Inc., 2001.

[5] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE,* vol. 77, pp. 541-580, 1989.

[6] L. Gomes, J. P. Barros, A. Costa, and R. Nunes, "The Input-Output Place-Transition Petri Net Class and Associated Tools," in *Industrial Informatics, 2007 5th IEEE International Conference on*, 2007, pp. 509-514.

[7] A. Costa and L. Gomes, "Petri net partitioning using net splitting operation," in *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, 2009, pp. 204-209.

[8] L. Gomes, A. Costa, J. P. Barros, and P. Lima, "From Petri net models to VHDL implementation of digital controllers," in *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, 2007, pp. 94-99.

[9] S. F. J. SPARSØ;, *Principles of Asynchronous Circuit*
Design – A System Perspective,: Kluwer Academic Publishers, 2002.

[10] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, 2000, pp. 52-59.

[11] J. N. Seizovic, "Pipeline synchronization," in *Advanced Research in Asynchronous Circuits and Systems, 1994., Proceedings of the International Symposium on*, 1994, pp. 87-96.

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

[13] D. S. Bormann and P. Y. K. Cheung, "Asynchronous wrapper for heterogeneous systems," in *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, 1997, pp. 307-314.

[14] M. Najibi, K. Saleh, M. Naderi, H. Pedram, and M. Sedighi, "Prototyping globally asynchronous locally synchronous circuits on commercial synchronous FPGAs," in *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, 2005, pp. 63-69.

[15] K. Y. Yun and R. P. Donohue, "Pausible clocking: a first step toward heterogeneous systems," in *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, 1996, pp. 118-123.

[16] A. E. D. Kenneth Y. Yun "Pausible Clocking Based Heterogeneous Systems," 1999.

[17] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. M. a. L. A.Conway, Ed.: Addison-Wesley, 1980.

[18] P. Teehan, M. Greenstreet, and G. Lemieux, "A Survey and Taxonomy of GALS Design Styles," *Design & Test of Computers, IEEE,* vol. 24, pp. 418-428, 2007.

[19] D. G. Messerschmitt, "Synchronization in digital system design," *Selected Areas in Communications, IEEE Journal on,* vol. 8, pp. 1404-1419, 1990.

[20] F. K. Gurkaynak, S. Oetiker, H. Kaeslin, N. Felber, and W. Fichtner, "GALS at ETH Zurich: success or failure?," in *Asynchronous Circuits and Systems, 2006. 12th IEEE International Symposium on*, 2006, pp. 10 pp.-159.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and S. Wen-King, "Myrinet: a gigabit-per-second local area network," *Micro, IEEE,* vol. 15, pp. 29-36, 1995.

[22] D. J. Kinniment, C. E. Dike, K. Heron, G. Russell, and A. V. Yakovlev, "Measuring Deep Metastability and Its Effect on Synchronizer Performance," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 15, pp. 1028-1039, 2007.

[23] Y. Suwen and M. Greenstreet, "Computing Synchronizer Failure Probabilities," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, 2007, pp. 1-6.

[24] Y. Suwen and M. R. Greenstreet, "Simulating Improbable Events," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, 2007, pp. 154-157.

[25] M. W. Heath, W. P. Burleson, and I. G. Harris, "Synchro-tokens: a deterministic GALS methodology for chip-level debug and test," *Computers, IEEE Transactions on,* vol. 54, pp. 1532-1546, 2005.

[26] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, 2003, pp. 89-96.

[27] D. E. M. a. W. S. Bartky, "A Theory of Asynchronous Circuits," in *Theory of Switching, Part 1*, H. U. Press, Ed., 1959, pp. 204–243.

[28] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 4, pp. 247-253, 1996.

[29] J. B. Johnson, "APPLICATION OF AN ASYNCHRONOUS FIFO
IN A DRAM DATA PATH," in *Electrical Engineering*. vol. Major in Electrical Engineering Idaho: University of Idaho, 2002.

[30] K. Chen, "Circuit Design for Logic Automata," in *Department of Electrical Engineering and Computer*
Science. vol. Master MASSACHUSETTS: MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2009.

[31] XILINX, "Spartan-3 FPGA Starter Kit Board User Guide," 2008.

[32] XILINX, "Xilinx University Program
Virtex-II Pro Development
System
Hardware Reference Manual," 2005.

[33] J. P. C. Oliveira, Anikó, Gomes, Luís, "Configurador de plataformas específicas em Co-design de Sistemas Embutidos," in *REC'2009 – V Jornadas sobre Sistemas Reconfiguráveis* Monte de Caparica, 2009.