COMPUTER ENGINEERING

Frederico Scherer Butzke

A SystemC Behavioral Model for Delay Variation Analysis in Asynchronous Designs

Santa Cruz do Sul
2015

Frederico Scherer Butzke

A SystemC Behavioral Model for Delay Variation Analysis in Asynchronous Designs

Bachelor's thesis proposed to obtain a degree of Computer Engineering at University of Santa Cruz do Sul.

Advisor: Prof. Dr. Leonel Pablo Tedesco
Reviewer: Prof. Msc. Guilherme Machado de Castilhos
Reviewer: Prof. Msc. Leandro Sehnem Heck

Santa Cruz do Sul
2015

to Clarice, Rogério, Carlos, Manuella and Lela.

ACKNOWLEDGMENTS

There are inumerous people that I am indebted with. Here I address those people who were really important to me while I was in my undergraduate life at Universidade de Santa Cruz do Sul. I won't be able to list all names that cross my mind and I am sure many people were as important as the ones listed here, but unfortunately I forgot. Sorry!

I would like to thank Vitor Righi for helping me to discover computer engineering as a course I would not regret the time I have spent. Also I would like to thank Prof. Dr. Leonel Tedesco for his support, lessons, friendship and the encouragement he transmitted to me during my years in school. Not less important is Prof. Dr. Matheus Trevisan that is a mentor and a great friend. I will be forever thankful for all his support.

In addition I would like to thank AIESEC for all the experiences and people I have met during my three years at the organization. I am thankful for everyone that was part of it.

Finally my family that I am grateful and have no words to describe how much I love them. Thank you for the support and patience. Rogério (father), Clarice (mother), Carlos (brother), Manuella (little sis) and Lela (future wife).

Imagination is more important than knowledge.

Albert Einstein

On Science, 1930s.

Abstract

The evolution of the manufacturing processes in the semiconductor industry are pushing the limits of physics every new technology generation. This growth represents a larger number of transistors per square inch in every generation. Today the predominant design style is synchronous design with complementary metal-oxide-semiconductor being the manufacturing standard. But, within the present decade, some of fundamental limits of physics are going to be reached. Thus, the manufacturing companies are always searching novel ways to overcome the rapid incoming challenges in the manufacturing technologies, *e.g.* transistor leakage in submicron processes and minimum transistor length limit. Even though there are many talented engineers, physicists and chemists, those limits are becoming more challenging every new manufacturing generation. Thus, new design styles appear as possible solutions, such as *asynchronous circuits*. However, this design style is still not mature enough in the market. There are some practical aspects that makes this style not attractive for design houses. For instance, asynchronous circuits have a higher timing requirement for verification and testing since they may have many timing assumptions that must be understood by the designers and well covered by verification tools. Therefore, the present thesis is going to focus in asynchronous circuits verification through the design of an environment for random wire and gate delays. This work will present a simulation framework built on top of SystemC that models asynchronous designs, where components of the design are mapped to a thread and are assigned to a delay value for each round of simulation. The logic functions of the design are verified with a predefined testbench, then the results are compared against the expected behavior to ensure the model remains working though the environment changes its characteristics. To perform the verification, the testbench runs the model many times with different delays for gates and wires, considering some limits in the operating conditions. The case study chosen for this thesis is a pipeline controller based on fundamental mode assumptions where the testbench will apply new input values to the controller just after its internal state has stabilized. The controller output logic functions are decomposed in gate networks that are mapped to SystemC modules. For each module, the inner components, including wire connections, are mapped into SystemC threads. The controller is simulated and a testbench representing a pipeline test is executed generating stimulus for the design. In the end, the testbench reports system measures and whether the design has passed or failed due to variability of the operating conditions.

Keywords: Delay Variation Model, Asynchronous Circuits, Huffman Circuits, Timing Analysis, Verification.

Resumo

A evolução no processo de manufatura de semicondutores vem forçando os limites conhecidos da física ao extremo, a cada nova geração. Essa evolução representa um aumento significativo de transistores em uma única pastilha de silício. Hoje em dia, a técnica mais difundida de manufatura de circuitos usa Óxido-Metal Semicondutor Complementar, porém nas próximas gerações de semicondutores essa técnica irá enfrentar dificuldades pois alguns limites fundamentais da física serão alcançados. Portanto, as empresas de manufatura estão constantemente buscando novas técnicas para vencer os desafios do limite da física que os novos processos vem introduzindo, como o "vazamento" de potência em processos submicrometro e o tamanho mínimo de largura do canal de um transistor. Embora muitos engenheiros brilhantes existem, esses limites fundamentais serão alcançados em breve. Portanto, novos estilos de projeto são considerados como uma possível evolução. Entre eles estão os circuitos assíncronos. Hoje em dia, essa metodologia de circuitos ainda não ganhou adesão entre projetistas e empresas de projeto pois eles possuem maiores requisitos para a verificação funcional pois esses circuitos precisam ser bem planejados para que não tenham pressupostos temporais errados. Nesse contexto a solução apresentada nesse trabalho entra como uma possível ferramenta de suporte, onde o foco será a verificação de circuitos assíncronos com atrasos em portas lógicas e em fios de uma maneira não determinística. Este trabalho apresenta o desenvolvimento de um arcabouço baseado na biblioteca de classes SystemC. Cada componente do projeto será mapeado em *threads* em SystemC e terá um atraso variável. Uma vez que o projeto está verificado, os resultados são comparados com os resultados esperados para garantir que o protocolo é funcional mesmo com atrasos não programados. Para executar essa verificação o ambiente irá rodar o modelo do circuito muitas vezes, atribuindo novos atrasos a cada nova rodada. O caso de estudo escolhido para ser verificado com o ambiente proposto é um controlador de *pipeline* que usa pressupostos temporais em portas lógicas e fios. Uma vez que o circuito é gerado, cada um de seus componentes vai ser mapeado como um processo em SystemC. A partir do modelo em SystemC o controlador é simulado em um ambiente que representa uma pipeline em execução gerando estímulos para o projeto. No final da verificação o ambiente de simulação irá reportar as métricas do sistema e se o controlador passou ou não no teste.

Palavras-Chave: Modelo de Variação de Atrasos, Circuitos Assíncronos, Circuitos de Huffman, Análises Temporais, Verificação.

LIST OF FIGURES

Figure 1 - Problem specification. 15

Figure 2 - Proposed verification environment. 17

Figure 3 - Next state logic in synchronous designs. 19

Figure 4 - Next state logic in asynchronous designs. 20

Figure 5 - Pipeline designs. 21

Figure 6 - Asynchronous pipeline. 22

Figure 7 - Bundled-data handshaking protocol. 23

Figure 8 - 4-phase bundled-data protocol. 23

Figure 9 - 2-phase bundled-data protocol. 24

Figure 10 - Asynchronous pipeline. 25

Figure 11 - Delay insensitive delay model. 26

Figure 12 - Fundamental mode delay model. 27

Figure 13 - Speed independent delay model. 28

Figure 14 - Flow table. 29

Figure 15 - Asynchronous finite state machine. 30

Figure 16 - Burst-mode state machine. 30

Figure 17 - Design implementation. 32

Figure 18 - Static and dynamic hazards. 34

Figure 19 - Static hazard. 35

Figure 20 - Dynamic hazard. 36

Figure 21 - SystemC SC_MODULE example. 38

Figure 22 - Pipeline controller top level design. 43

Figure 23 - Pipeline controller burst-mode specification. 44

Figure 24 - 3D tool - input and output data. 45

Figure 25 - Design implementation. 46

Figure 26 - Mapped technology. 48

Figure 27 - Delays in the LA module. 49

Figure 28 - Proposed verification framework overview. 51

Figure 29 - System implementation. 53

Figure 30 - System implementation C++ classes. 53

Figure 31 - Content of delay.h header file. 54
Figure 32 - SC_MODULES organization LA, en, RR, Z. 55
Figure 33 - SC_MODULES organization top. 56
Figure 34 - INV_RA SC_THREAD. 56
Figure 35 - Top module SC Controller with connections. 57
Figure 36 - Testbench stimulus. 58
Figure 37 - Source thread. 59
Figure 38 - Error detection algorithm. 60
Figure 39 - Algorithm for checking if the output is glitching. 61
Figure 40 - Algorithm for checking the output state. 61
Figure 41 - Makefile script. 62
Figure 42 - Algorithm for generating a latch hold time warning. 62
Figure 43 - Log files. 63
Figure 44 - Delay model for the modules in first simulation. 65
Figure 45 - First simulation output. 66
Figure 46 - Delay model for the modules in second simulation. 66
Figure 47 - Second simulation output. 67
Figure 48 - Second simulation - percentage of errors chart. 67
Figure 49 - Second simulation - simulated gate delays. 69
Figure 50 - Second simulation - pass x fail rounds. 69

LIST OF ABBREVIATIONS

Ack	Acknowledge
AFSM	Asynchronous Finite State Machine
AOI	AND-OR-INV
ASI	Asynchronous System
ASIC	Application Specific Integrated Circuit
BM	Burst Mode
CAD	Computer-Aided Design
clk	Clock
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DI	Delay Insensitive
EDA	Electronic Design Automation
en	Enable
FF	Flip-flop
FM	Fundamental Mode
FSM	Finite State Machine
GALS	Globally Asynchronous Locally Synchronous
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
INV	Inverter
LA	Left Acknowledge
LR	Left Request
LT	Latch
NoC	Network on Chip
POC	Proof of Concept
PVT	Process, Voltage and Temperature
QDI	Quasi-delay insensitive
RA	Right Acknowledge
Req	Request
RR	Right Request
rst	Reset
RTL	Register Transfer Level
SC	SystemC
SDT	Syntax-Directed Translation
SI	Speed Independent
STG	Signal Transition Graph
STL	Standard Template Library
tb	Testbench
VHDL	VHSIC Hardware Description Language

CONTENTS

1 INTRODUCTION 13
1.1 Motivation 14
1.2 Objectives 18
1.3 Organization of Thesis 18
2 LITERATURE REVIEW 19
2.1 Synchronous Background 19
2.2 Asynchronous Circuits Fundamentals 20
2.2.1 Pipelines 20
2.2.2 Communication Protocols 21
2.2.2.1 Bundled-data 22
2.2.2.2 Four-Phase 23
2.2.2.3 Two-Phase 23
2.2.3 Delay Models 24
2.2.3.1 Delay Insensitive 25
2.2.3.2 Funamental Mode 26
2.2.3.3 Speed Independent 27
2.2.4 Graphical Representation 28
2.2.4.1 Flow Table 28
2.2.4.2 AFSM 29
2.2.4.3 Burst-Mode State Machines 30
2.2.5 Design Implementation 31
2.2.5.1 Syntax-Directed Translation 31
2.2.5.2 Flow Table Reduction 31
2.2.6 Verification 32
2.3 Delays and Hazards 33
2.3.1 Static Hazard 34
2.3.2 Dynamic Hazard 35
2.4 SystemC 36
3 RELATED WORKS 39
4 CASE STUDY 42
4.1 Burst Mode Controller Specification 42

4.2 Asynchronous Finite State Machine 44
4.3 Technology Mapping 46
5 TIMING VERIFICATION ENVIRONMENT 51
5.1 Overview 51
5.2 Controller Module Organization 54
5.3 Testbench Module Organization 57
5.4 General Features 59
5.5 Summary 64
6 RESULTS 65
6.1 Simulation 1 65
6.2 Simulation 2 66
6.3 Discussion 70
6.4 Summary 71
7 CONCLUSION 72
7.1 Objectives Check 72
7.2 Future Work 73
8 FINAL THOUGHTS 74
9 REFERENCES 75

1 INTRODUCTION

In 1958, the first integrated circuit (IC) was built at Texas Instruments with only two transistors. Almost six decades past that, the humankind is presenting ICs containing over 10 billion transistors. The increase in transistor count from 2 to 10^9 have been led by the incredible advance in the semiconductor industry that evolved from vacuum tubes to solid cold metal structures that enabled a large miniaturization of the basic building block of electronic designs, named transistor (WESTE et al, 2010).

When analysing the rapid evolution in semiconductors manufacturing process, performance, power and price go within the same direction. Since transistors become smaller, they have less capacitance, which means they become faster, dissipating less power, using less die area and less metal volume that leads to a cheaper production (WESTE et al, 2010).

The constant improvement in ICs helps the people to connect and share the knowledge faster and cheaper in every new technology node. The knowledge and information that are acquired in new technologies are then used to design and produce improvements in the current technology, making even faster applications.

Gordon Moore has observed this behavior and in 1965 pronounced that the number of transistors would double every 18 months. This is known as *Moore's Law*. The ongoing reduction supports the increasing of the number of transistors on a single die, enabling them to integrate complex system for example in system-on-chip (BEEREL et al, 2010).

In recent decades there is a predominant manufacturing technology among different market suppliers such as Intel, TSMC and MOSIS. This technology is called complementary metal-oxide-semiconductor (CMOS). Within CMOS, a design style that is well spread is known as synchronous design the relies on a global clock signal to dictate the next state changes.

Even though, CMOS technology uses little power within a single transistor in the switching process, the large number of transistors switching at high frequencies have made power consumption one of the designer's major concern. Besides, power, delay and performance tradeoffs are arising in early technologies, smaller than 45 nm.

In addition, increasingly process, voltage and temperature (PVT) variations make the power and timing analysis of electronic design a much more complex task (BEEREL et al, 2010).

Therefore, new design styles are being considered as part of a new technology advance for the next generations of application specific integrated circuit (ASIC) designs. One of the design styles that is become popular within the academia is the *asynchronous circuit* design style. Asynchronous circuits change the delay model of the next state logic compared to synchronous designs. Where the design no longer has a global signal controlling the data that flows in the design.

1.1 Motivation

It was previously shown that asynchronous design is an option for dealing with the rapid evolution of the manufacturing technologies and increase of process variations that electronic designers should be aware of.

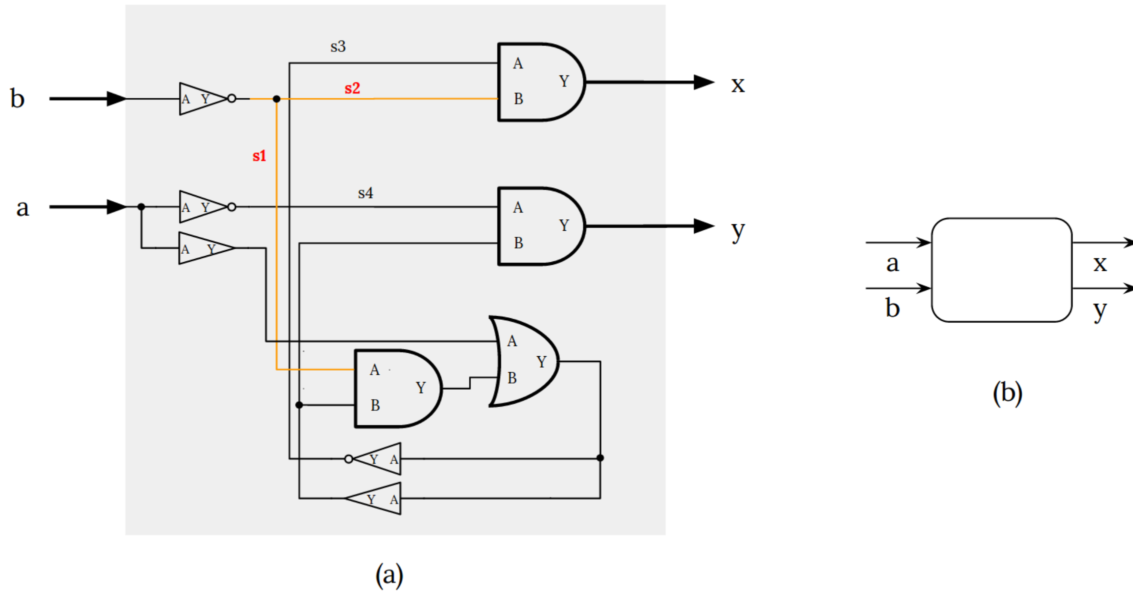
In addition, the design and technology do not need to have any sort of variation concern to chose asynchronous circuits as the design to be adopted. There are many positive aspects of asynchronous that the design team could consider as part of decision process of implementing a system.

However, asynchronous design does not mean that the process variations would no longer be a hazard for the design. Environmental variations may lead to delay variations that may lead to circuit malfunction. It does not depend whether the design is synchronous or asynchronous.

Consider the circuit depicted in Figure 1a. It represents a circuit that implements the behavior of the top module shown in Figure 1b that is a controller.

The controller has only two functions, it tells to the left (right) channel that the right (left) channel is ready. The name active/active means that the controller is responsible for the first event on either channel. The output ports x and y are the channels that the controller sends the requests and the input ports a and b are the response channels.

Figure 1 - Problem specification.



Source: MYERS, C. J.

Suppose the following scenario: $x\uparrow\ a\uparrow x\downarrow\ a\downarrow y\uparrow$. Then, consider $b\uparrow$. After the inverter (INV) that follows *b* there is a wire fork between *s1* and *s2*. Consider that *s1* has a negligible delay compared to *s2*.

What shall happen is that the AND gate connected to *s1* will see a low level being presented at its port *A*. Thus, it is going to output *0* to the OR gate that is responsible for the feedback. At this time the OR gate has both of its input ports at a low level. Therefore, it produces a low level signal at the output port.

Once the feedback is low, the wire *s3* will be at a high logic level at this point *s2* is still high, because it has a greater delay than s1, that behavior would output *1* at the *x* port. Then the signal *s2* finally falls to its expected behavior and *x* goes settles to *0*.

Even though the left channel is again at the state it should be, there was a catastrophic disorder at the system that was caused by the wire delays. The delay in *s2* led to a glitch at the output port *x*. This represents a behavior that was not expected, since the right channel is still accomplishing its communication.

A possible correction for this design is to add a minimum delay requirement for the feedback path that would cause the feedback to happen just after the internal wires have stabilized their states.

The previous example has shown that delays may lead to malfunctions in asynchronous designs. Differently of synchronous designs, where a clock signal is controlling the transitions and when the next state changes would happen, asynchronous circuits having a

design that is not well planned and verified in regards to its internal delays may lead to malfunctions as shown in the previous example.

In asynchronous a glitch is catastrophic, since it means a new request for a new communication that may never happen and may lead to a complete deadlock of the communication protocol.

Therefore, having those glitches and delay assumptions, require a deep understanding of the environment as well as the internal logic of an asynchronous circuit. Adding the delay elements or making the calculations of cell drivers to meet the timing requirement for a specific design might be a hard and tedious task. Nowadays many research groups created different implementations for generating the delays and circuits without hazards (BROWN et al, 1988).

Even though the tools have the ability for generating the circuitry with matching delays and well covered logic, in the Real World the timing may vary according to PVT.

At this point, it would be handful to have a verification environment that would model the asynchronous circuit's gates and wires delays and run a dynamic timing analysis that would comprise a portion of the possible set of timing variations that could happen within a circuit.

This leads to our proposed verification environment that generates random gate and wire delays to simulate variations that could lead to malfunctions.

The timing analysis tool has an instance of the asynchronous design and for each gate and wire a thread would be executed. At the execution time, upper and lower bound delays for each component would be generated.

The idea is to develop a new approach for the verification of asynchronous circuits by creating an environment that allows designers to simulate the system using random delays for each gate in a netlist for a period of time that will cover a good portion of the design. This environment for asynchronous verification is a new approach for optimizing timing constraints verification.

The environment would ensure that the timings in the design would be tested with a great coverage and the gates would behave, regarding delays, randomically.

The current thesis aims to implement this timing verification environment for the varying delays of gates and wires to ensure the asynchronous design has no glitch or any other sort of hazard that could compromise the entire system.

Figure 2 shows the proposed environment. In this work we are going to show and explain how we have generated the case study, Asynchronous Pipeline Controller, and how we have modeled it using SystemC. In addition it is explained how the verification environment works.

In summary, the environment runs on top of a verification framework that consists of different scripts and a SystemC simulator that is used to describe the hardware and produce the timing analysis.

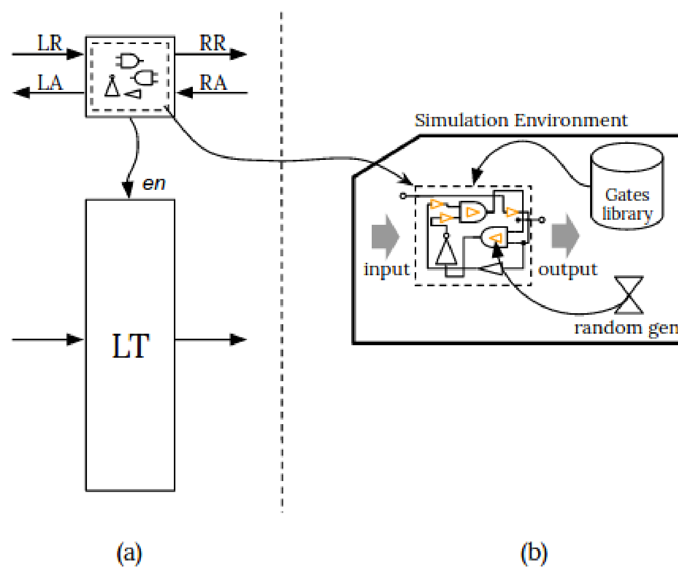
Each design runs for a predefined number of rounds. Within each round new varying delays are generated for wires and gates. The simulations places the controller in a pipelined environment that has two mode of operation, the least concurrent and and the most concurrent. Once the simulation is completed, it prints whether the controller works or not regarding the delays.

If the circuit does not show the expected behavior it prints the error messages to a log file that represents the malfunctions it has experienced in the simulation stage.

Once we have the output information from the simulation we could specify lower and upper bound delays for each gate and wires, in order for preventing glitches with the PVT variations.

The details of how we have designed the case study, the simulation environment and how everything is connected is explained later in this thesis.

Figure 2 - Proposed verification environment.



Source: BUTZKE, F.S.

1.2 Objectives

A. Implement a simulation environment.

This is the main goal of the thesis. We shall design a simulation environment that models asynchronous circuits in a way that each single component can have a unique delay and acts like a thread.

B. Define a case study.

The second goal is to define a case study that will be modeled and verified by the developed environment.

C. Verify a case study using the proposed environment.

This goal refers to the connection of the objectives A and B. Once we have both defined we start the verification phase.

1.3 Organization of Thesis

This thesis is divided into eight sections. Following the introductory section, the Literature Review is shown in Section 2. It presents some basic concepts and fundamentals of Synchronous Design, Asynchronous Design, hazards and a SystemC overview. Section 3, Related Works, presents the related works that the authors have found in the research for the realization of this thesis. Section 4, is the Case Study. It shows the case study development that is to be verified by the developed environment. Section 5 presents the Timing Verification Environment. This section represents the main contribution of the this thesis. Section 6 shows the results and a discussion about them. Section 7 is the conclusion. Finally, the Section 8 are the final thoughts.

2 LITERATURE REVIEW

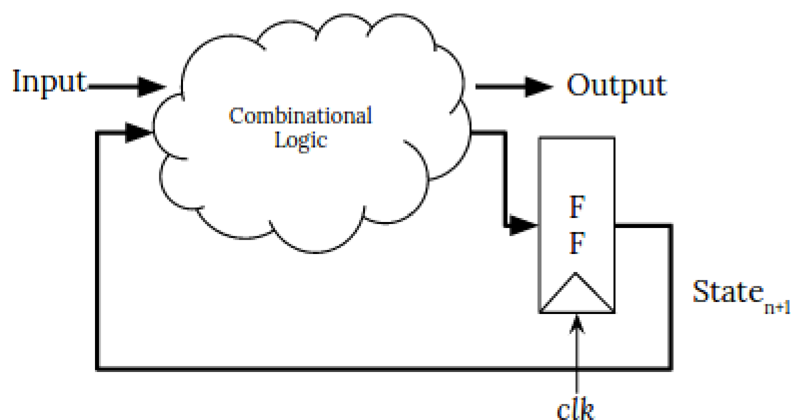
In this section we introduce the fundamentals of asynchronous circuits such as handshake templates, flow tables and asynchronous finite state machines (AFSM). Additionally, we present a review of static and dynamic hazards and in the last subsection we present the basic concepts of SystemC.

2.1 Synchronous Background

We are going to start with a review of synchronous design. Before moving to asynchronous design, we recall what the synchronous design represents. In synchronous design, the FSM is guided by the clock signal. In large designs, it can be distributed using clock trees, or, to be split creating different clock pools inside the design (FURA, 1988)(MELY, 2000).

The clock signal allows the combinational network to be designed between registers, in which are usually FFs (MYERS, 2001). Figure 3 depicts a typical synchronous next state logic design. In this design, the combinational network may compute the data in its input, producing the output a period of time before the clock signal. Thus, the synchronous designs constraints the *clk* pulse in order to ensure that the data produced by the logic network is valid.

Figure 3 - Next state logic in synchronous designs.



Source: BUTZKE, F.S.

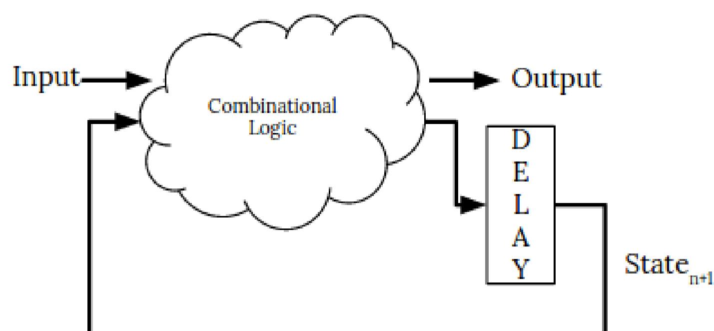
Beerel (2010) says,

> Synchronous design has been the predominant design methodology, largely because of the simplicity and efficiency provided by the global clock. The register decompose the design into blocks of combinational logic between registers which facilitate efficient designs and synthesis. Probably, the time-to-market advantage of std-cell-based ASIC designs being is one of the powerful benefits of synchronous.

2.2 Asynchronous Circuits Fundamentals

Figure 4 presents the next state logic for asynchronous designs. Instead of having state holding elements, *e.g.* FFs, this sort of design has delays in the feedback loop. The feedback creates the state holding requirement for building the Asynchronous Finite State Machine (UNGER, 1969).

Figure 4 - Next state logic in asynchronous designs.



Source: BUTZKE, F.S.

The only synchronism needed is that the feedback should match the delay of the combinational network to not feedback before the internal states have stabilized. AFSMs are input based, as soon as they see an input event, they start changing the current state to a new state.

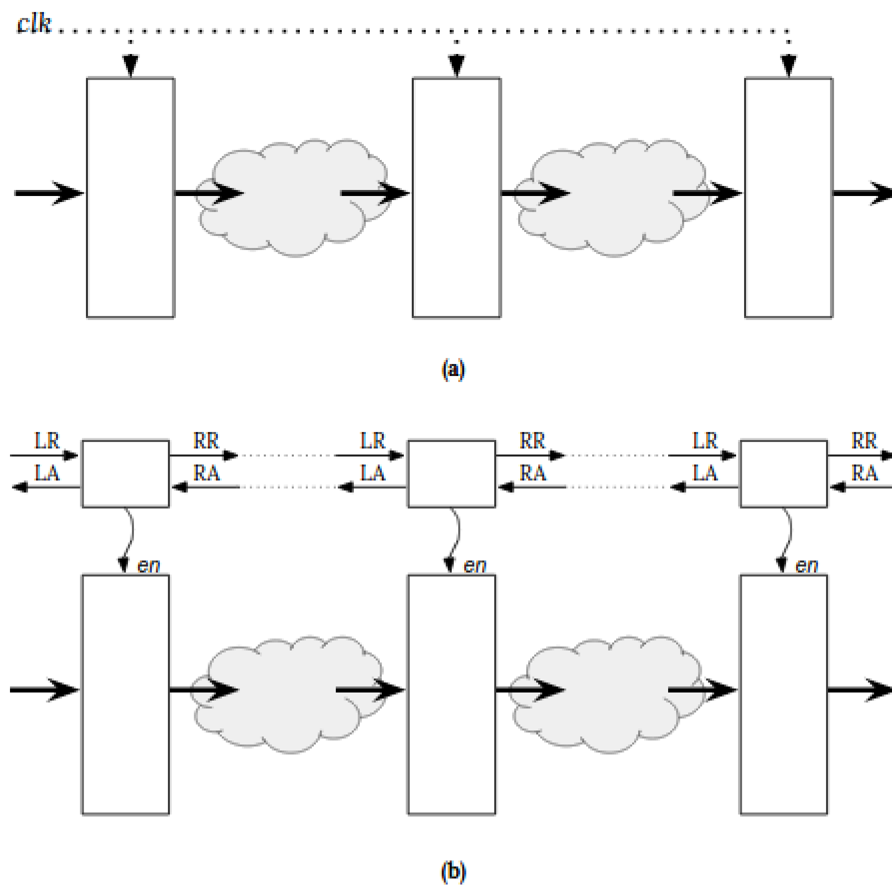
2.2.1 Pipelines

A simple overview of how the replacement of synchronous designs by asynchronous is shown in Figure 5. Figure 5a represents a synchronous design using a *clk* signal. Figure 5b is the asynchronous implementation. Where local handshake modules are replaced where there was previously a *clk* port.

Throughout this thesis, we may refer the local handshaking module as **asynchronous pipeline controller**. When, we consider a **neighbour module**, that module that is direct connected to the a given controller. In addition, the data flow convention is from left to right. In addition, Sparsø (2001) stands that,

> An important message is that the "handshake-channel and data-token view" represents a very useful abstraction that is equivalent to the register transfer level (RTL) used in the design of synchronous circuits.

Figure 5 - Pipeline designs.



Source: BUTZKE, F.S.

2.2.2 Communication Protocols

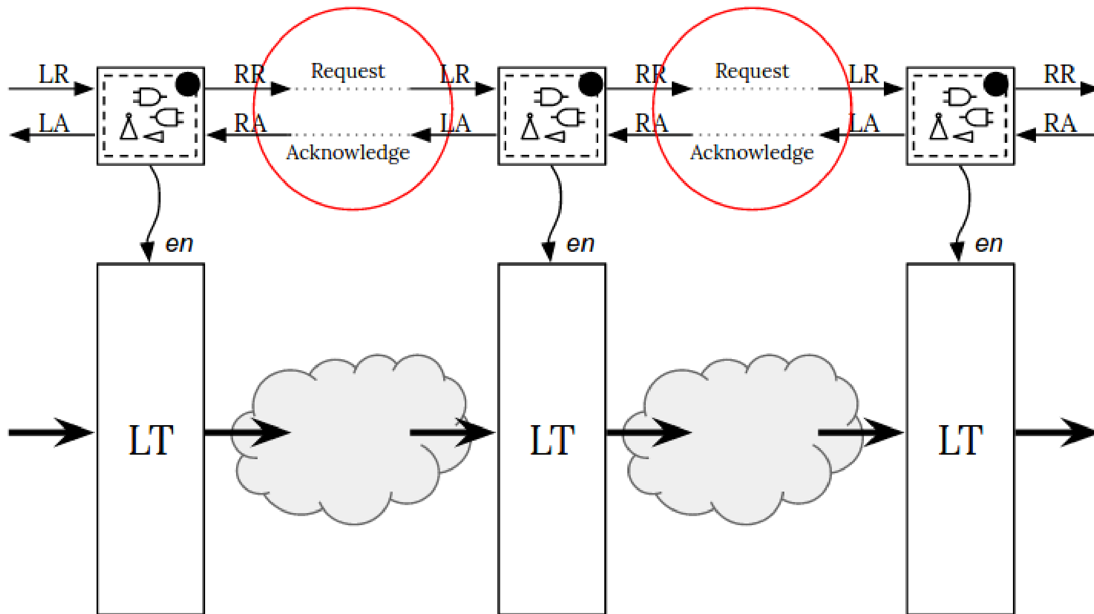
A quick clarification about handshake protocols and asynchronous circuits. Handshake is the process of exchanging communication with other modules by explicit request, *req*, and acknowledge, *ack*, wires. Handshaking is not exclusively implemented with asynchronous designs (KESSELS, 2002).

It happens that in asynchronous circuits the handshake is the method used to implement the communication. The point is that the circuit inside the module that is

implementing the handshake is the truly asynchronous design that happes to implement the handshaking protocol.

Thus, we are going to first introduce the handshaking protocols. In subsection will focus on showing how a handshake protocol may exchange data. Figure 6 represents three asynchronous pipeline stages. Each stage has its own controller that implements the handshake protocol. The both channels are highlighted with red circles.

Figure 6 - Asynchronous pipeline.

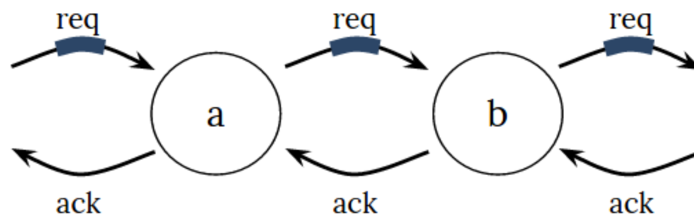


Source: BUTZKE, F.S.

2.2.2.1 Bundled-data

The bundled-data handshaking protocol consists of "bundling" the data into explicit *req* channels. Figure 7 shows an abstraction of a pipeline. Circles represent a pipeline state. The register and the data are omitted. In this representation, when module *a* must request to module *b* it does so by sending a request in the *req* wire. Once module *b* stores the data it acknowledges the communication sending a response in the *ack* channel. There are other types of channel encoding that do not use explicit request lines (CANNIZZARO, 2012), but they are out of the scope of this thesis.

Figure 7 - Bundled-data handshaking protocol.

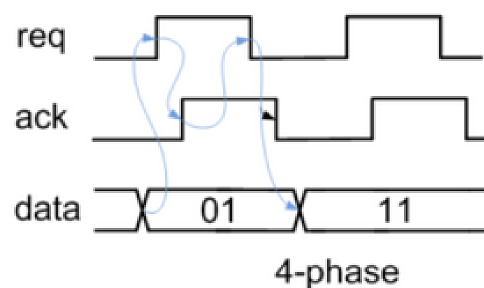


Source: BUTZKE, F.S.

2.2.2.2 Four-Phase

The four-phase handshaking protocol means that the *req* and *ack* must accomplish four transitions to be the exchange to be considered valid. Figure 8 shows a timing diagram for a four-phase bundled-data protocol.

Figure 8 - 4-phase bundled-data protocol.



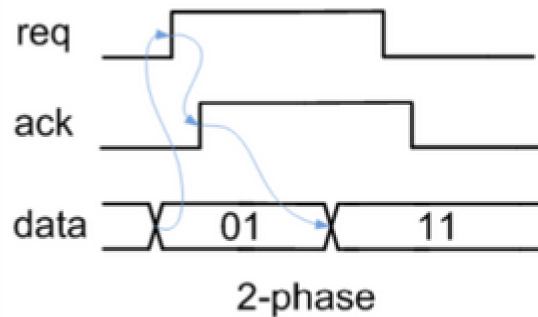
Source: MOREIRA, M.

2.2.2.3 Two-Phase

A variation of the 4-phase bundled data is the 2-phase bundled-data protocol. The protocol behavior is depicted in Figure 9. In this protocol, the transition that was wasted by the 4-phase protocol is now valid.

This protocol may lead to improvements in performance but it has an area tradeoff as shown in (MYERS, 2001).

Figure 9 - 2-phase bundled-data protocol.



Source: MOREIRA, M.

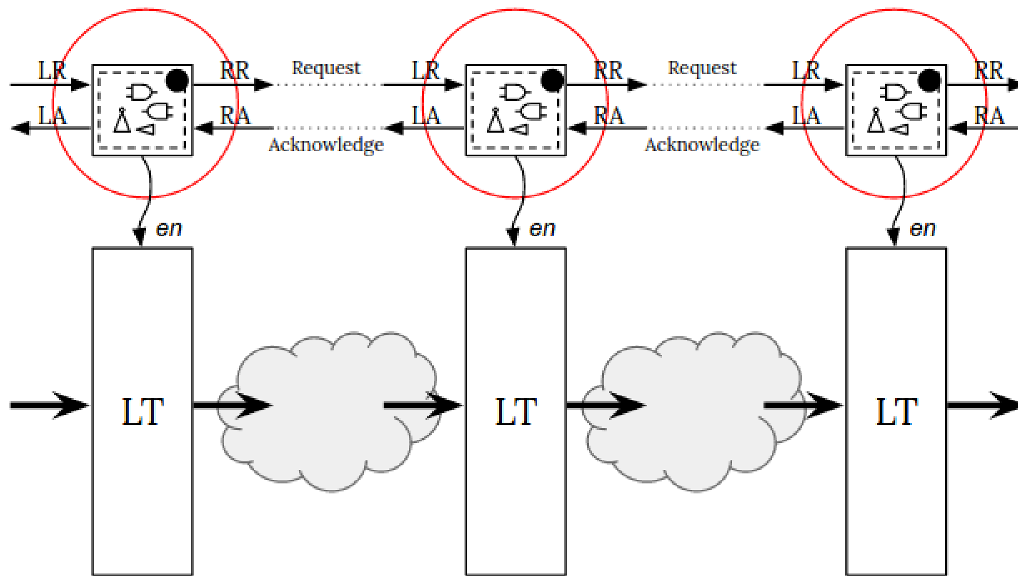
2.2.3 Delay Models

This section presents the difference of delay models in asynchronous circuits. For instance: there are three main delay models for any asynchronous designs: Delay Insensitive (DI), Fundamental Mode (FM) and Speed Independent (SI).

Before moving on in the structure of the designs we introduce the conceit of *active* and *lazy* asynchronous designs. As we have seen previously, AFSM responds to input transition. In asynchronous circuits a channel that waits the input transitions is said to be **lazy**. While the channel that initiates the communication is said to be **active**. In the design schematics, the active channel is presented with a black circle.

Figure 10 shows the same pipeline that was previously introduced. But instead of focusing in the handshake protocol, we move the focus to the circuit inside the pipeline controllers. The asynchronous designs the actually implements the handshaking protocol.

Figure 10 - Asynchronous pipeline.



Source: BUTZKE, F.S.

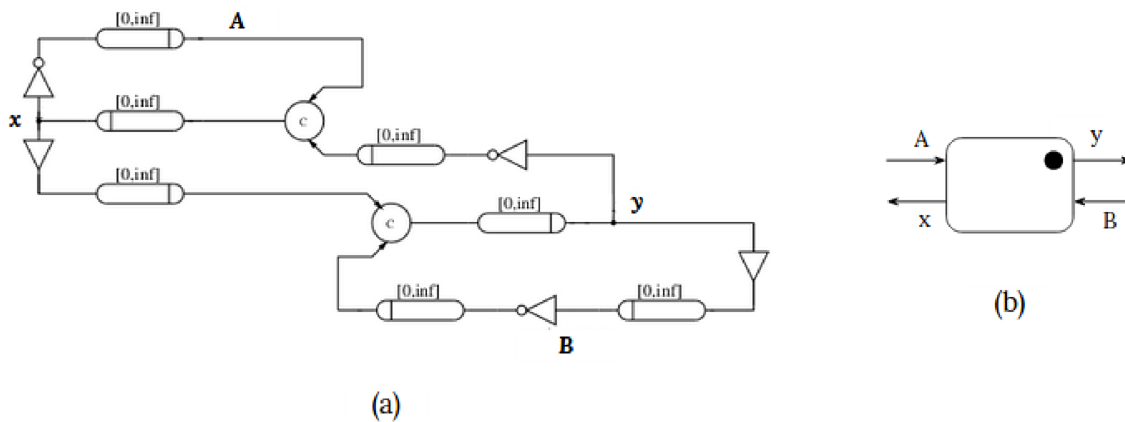
2.2.3.1 Delay Insensitive

This subsection introduces the DI delay model. Figure 11a shows the DI implementation for the design of the module presented in Figure 11b, this module represents a lazy/active module.

The cigar shaped components are wire or environment delays. The have a lower bound of 0 and an upper bound of infinity. The circles with with a centralized C represent C-elements. They one of the basic building blocks for asynchronous circuits that implement DI designs. Their function is to keep the previous output until the point all its input are the complement of the output. It is an analogy for a majoritary gate.

Consider that all internal variables are 0. Then, the only possible outcome is that A may go to 1 between [0, inf] time units. Once A is 1, *x* goes to 1. Here either A can go to 0 and B can go to 1. But for whichever delay values are chosen each time around they output produced by this design will work. This circuit is said to be a delay-insensitive design.

Figure 11 - Delay insensitive delay model.



Source: MYERS, C.J.

At this point we may think that we can design any circuit as a DI design. Unfortunately, this is not the case. In general, if only single output gates are required(buffers, inverters, C-elements), this class of circuits might be valid, but it is severely limited (MYERS, 2001).

2.2.3.2 Funamental Mode

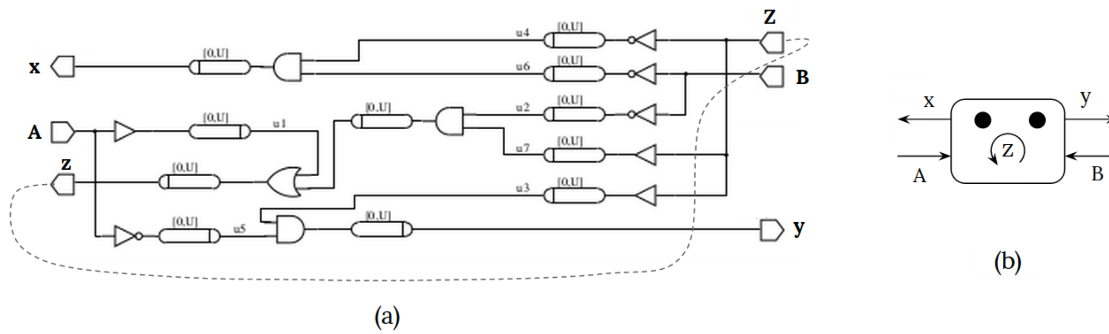
The second class of asynchronous delay model is called Fundamental Mode. This delay model assumes that the circuit is operated in a single-input fundamental mode. The environment will apply only a single input change each time. It considers the time required for the internal states to stabilizes.

To maintain the order and correctness, it may not only be necessary to slow down the environment, but it may also be necessary to delay the state signal change from being fed back too soon by adding a delay between X and x (MYERS, 2001).

Figure 12 presents a design using the fundamental assumption. The circuit implements an active/active protocol where both output port, *x* and *y*, actively send requests. When the output port is active it is represented as a black bullet in the top module. The logic network is composed only by inverters, buffers, AND and OR-gates.

Previously we have seen the DI delay model, where delay had no impact inside the module. Unfortunately this is no longer the case in FM designs. Assume that all signals are 0 except u2, u4, u5, and u6 which are 1. Then trace the following behavior: x+, A+, Z+, z+, x−, A−, y+, B+.

Figure 12 - Fundamental mode delay model.



Source: MYERS, C.J.

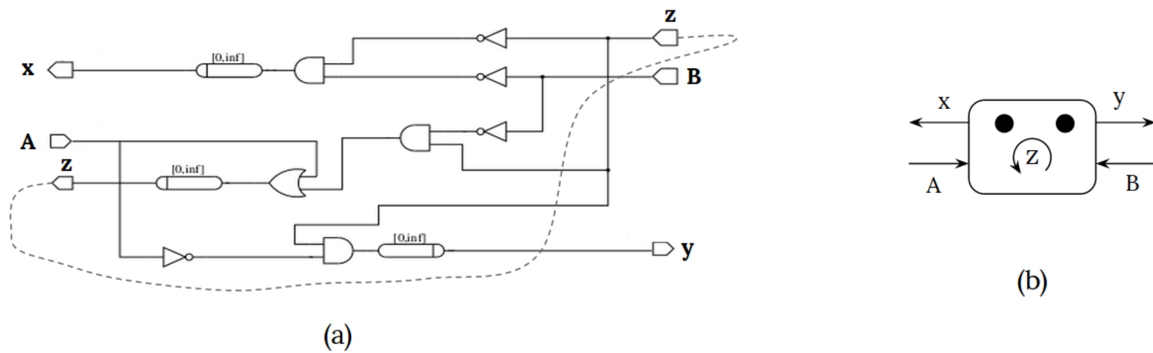
After u2 becomes 0, x can go to 0, and assume that it does so before u6 goes to 0. After z becomes 0, u4 can go to 1. If this happens before u6−, then z is enabled to go to 1. We may have a momentary glitch in x. To avoid the error, we add enough delay in the feedback path that u6 must accomplish its transition before the internal state is fed back.

At this point, u2 and u6 are again enabled to go to 0. Assuming that the delay of u2− is faster than that of u6−. After u2 becomes 0, Z can go to 0 before u6 goes to 0. However, in this case, we have added sufficient delay in the feedback path such that we do not allow the internal state to feedback until we have ensured that the circuit has stabilized. In other words, as long as the delay in the feedback path is greater than the upper bound wire delays, U, the glitch does not occur.

2.2.3.3 Speed Independent

The third and last delay model represents the class of Speed Independent circuits (SI). Figure 13 shows the the logic network and the top module. SI circuits are very similar to FM circuits. However, notice that the delay in wires have no bounds. SI has only a single delay element on each output and next state signal. The upper bound of the delay elements have also changed to infinity. In addition, there is no requirements for delaying the feedback, z. A similar model where only certain forks are isochronic is called quasi-delay insensitive (QDI).

Figure 13 - Speed independent delay model.



Source: MYERS, C.J.

Consider again the sequence: x+, A+, z+, x-, A−, y+, B+. At this point, the gates for z and y see the change in A at the same time, due to the isochronic fork. Therefore, when z goes to 0, the effect of A being 1 has already been felt by y, so it does not glitch to 1.

This means that whenever a signal changes value, all gates it is connected to will see that change immediately. For example, B and A each fork to two other gates while z forks to three. These forks are called isochronic forks, meaning that they have no delay.

Those isochronic forks may be hard to design. Both, SI and FM, models require some special attention to delays. In addition, not all the forks need to be isochronic. For instance each branches of the wire fork for z may have a different delay and the circuit still operates correctly.

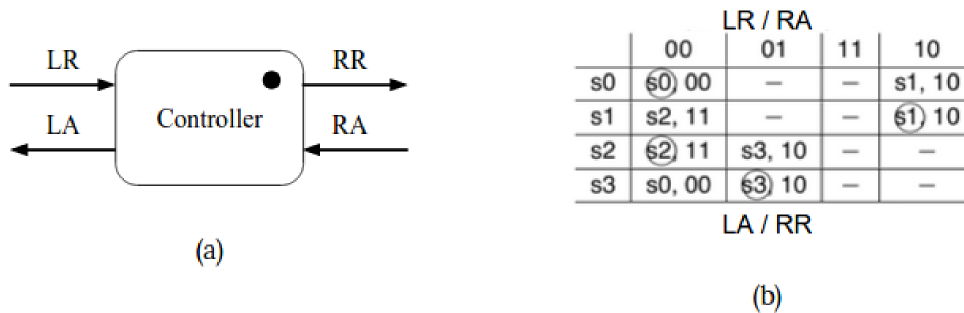
2.2.4 Graphical Representation

The following subsection presents the different graphical representation could be used to demonstrated and design an asynchronous circuit.

2.2.4.1 Flow Table

The most basic form of representing an asynchronous design is with a **flow table**. Flow table are a sort of k-maps for asynchronous circuits. They represents the basic model for any asynchronous design. They relate input transitions, output transitions with an internal state variable. Figure 14a shows a pipeline controller and Figure 14b represents the flow table.

Figure 14 - Flow table.



	00	01	11	10
s0	(s0) 00	–	–	s1, 10
s1	s2, 11	–	–	(s1) 10
s2	(s2) 11	s3, 10	–	–
s3	s0, 00	(s3) 10	–	–

Source: MYERS, C.J.

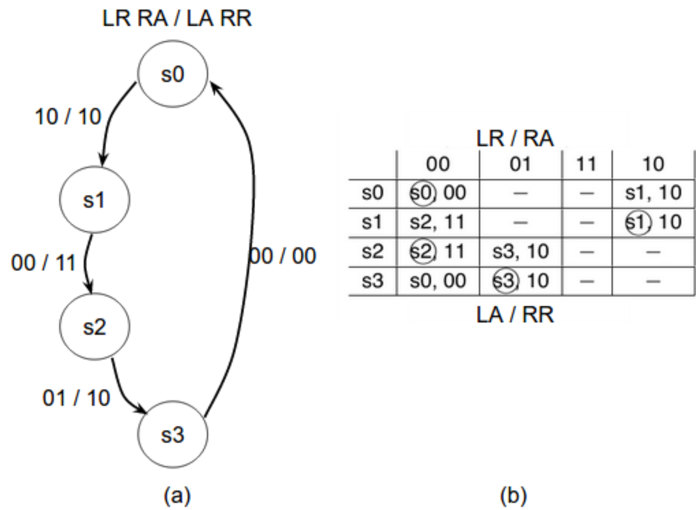
In the flow table the leftmost column represents the internal state. The top labels represent the input ports, the bottom label represents the output ports and the cells represents the **total state** that is the combination of the internal state and the output values.

For example, starting at the state s0 column 00. The output are also 00. If LR goes to 1, then the flow table, “flows” towards the last column, 10, and moves down to the state s1. It ends up with the total state of “s1 10”. Recall that the total state is the internal state and the output combined, thus s1 10 means that the flow table for this controller outputs LA = 1 for this particular total state.

2.2.4.2 AFSM

A finite state machine is usually represented by a graph. Where the set of vertices are the states and the set of edges are the state transitions. Each edge specifies the input that generates the transition to the output state. In asynchronous circuits there are actually a graph called AFSM. Figure 15 shows the AFSM that shapes the behavior of the same designs from Figure 15. As expected the flow table shown in Figure 15b is the same for both. In summary, an AFSM represents a flow table.

Figure 15 - Asynchronous finite state machine.

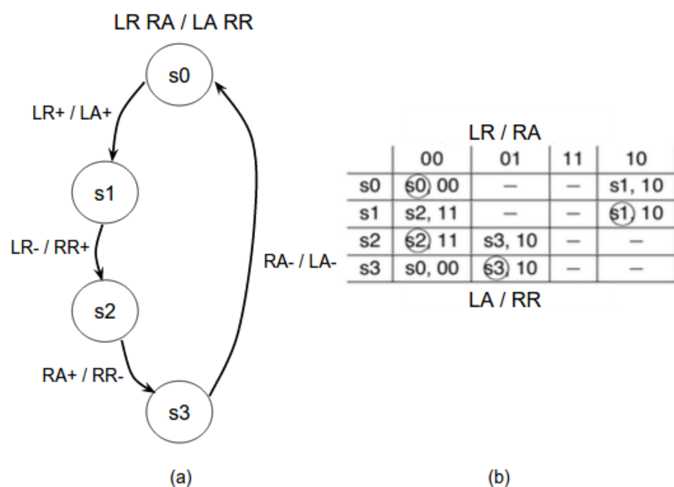


Source: MYERS, C. J.

2.2.4.3 Burst-Mode State Machines

Yet another graphical representation is called Burst-Mode state machines (MYERS, 2001). Figure 16a shows the graph that implements the same flow table from the previous example.

Figure 16 - Burst-mode state machine.



Source: MYERS, C. J.

The difference is that the edges are labeled with the input / output transitions rather than their logic level. They represent whether an input is expected to rise or to fall within a transition. The input set is called *input burst* and the output set is called *output burst*.

Input bursts may not be empty. On the other hand, there is no concern in regards to empty output bursts. In addition, a BM can also be translated to a flow table. The flow table that represents the BM is shown in Figure 16b**.**

2.2.5 Design Implementation

This subsection presents different methodologies that have been proposed in order to map the an asynchronous specification into a working gates network.

2.2.5.1 Syntax-Directed Translation

Syntax-Directed Translation (SDT) expands the handshaking protocol directed to predefined communication hardware structures. The structures implement C-elements, arbitrators and toggles that implement the expected behavior. A deep study of SDT is shown in (MYERS, 2001).

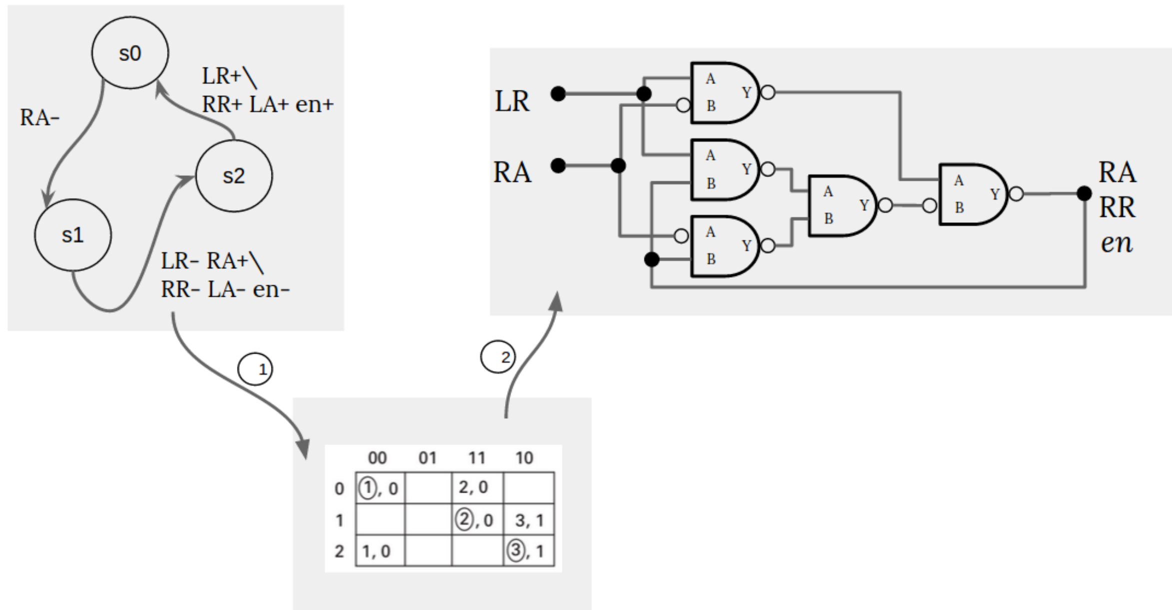
2.2.5.2 Flow Table Reduction

This is the most common way of implementing BM designs, the example of how to generate logic equations from BM state machines is shown in (MYERS, 2001) and (BEEREL, 2010).

It starts with a high level BM specification. Then a set of implementation tools, e.g. 3D and MINIMALIST, convert the BM into a flow table and then realize the state assignment and logic reduction among other optimizations (YUN, 1994)(FUHRER et al, 1999). The algorithms that realize those minimization may be find in (MYERS, 2001).

As a result, the tools produce the set of logic equations that represent the implementation of the design. In the case study section we are going to introduce further steps that might be required in order to produce the valid logic network.

Figure 17 - Design implementation.



Source: BEEREL, P. A.

2.2.6 Verification

When it comes to the commercial exploitation of asynchronous circuits the problem of test comes to the fore (SPARSØ, 2001).

In the synchronous circuits, the performance of a pipelined is defined in terms of its latency and throughput. Throughput represents the number of token that are exchanged per cycle. And Latency is the number of *clk* ticks that are needed in order to produce a result.

The cycle time in an asynchronous system is the period between two results. Because this time can vary between tokens, it is often taken to be the average time between output tokens (BEEREL et al, 2010).

Beerel (2010) also mentions that,

> The latency of an asynchronous system is the time between input tokens being consumed at the primary inputs and output tokens being generated. Latency is measured by the presentation of one set of primary input tokens in isolation, to avoid the possibility of congestion caused by previous tokens impacting the measurement.

When a designer builds an asynchronous specification, he or she, tries to identify whether a deadlock could arise. In order to validate the specification against its original specification, simulation can be used. But this cannot ensure complete testing coverage (MYERS, 2001).

The previous scenario leads to methods of verification that check whether a specification meets the requirements under all permissible delay behaviors or not.

In asynchronous design anything short of exhaustive simulation will not ensure the correctness of the implementation. In this sort of circuits, a hazard could appear in a very narrowed set of input vectors, in a very specific case and delays. Thus, checking the circuit under all possible delays is a must (MYERS, 2010).

2.3 Delays and Hazards

A glitch in asynchronous designs may cause the system to go to an unwanted internal state. Thus, the circuit must be hazard free.

Unger (1969) explains delay elements and stray delays. Here we introduce the difference between *delay elements* and *stray delays*. Delay elements are delays that have been deliberately introduced in the design, while stray delays are those delays that are present in gates and wires because of physical characteristics of the manufacturing process. While delay elements have a minimum and maximum bounded value, stary dalys are modeled with a minimum bound of 0 to some unknown upper bound. A design that is created without delay elements is said to be *delay free*. But that does not mean that the circuit has zero delay, since the stray delays in the logic components and wires.

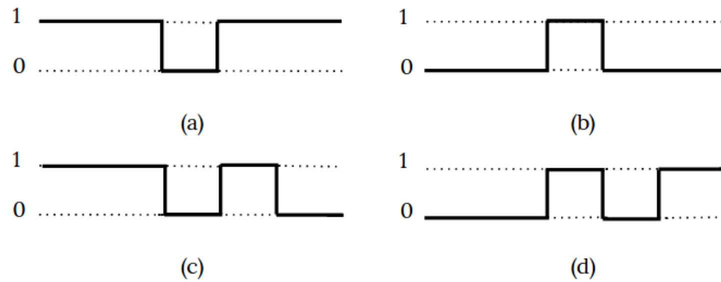
The stray delays play an important role in a combinational design. As demonstrated in Unger (1969), stray delays may cause unwanted pulses in the output function after specific set of input vectors. Thus, a circuit in which such pulses may occur for some distribution of stray delays is said to have a *combinational hazard*.

Unger (1969) also says that this hazards are related to the circuit configuration and not with the physical implementations. A *hazard free* design is one that does not presents combinational hazards regardless of the distribution of stray delays.

In Brown (2008) some example of static and dynamic hazards are presented. In synchronous design, a hazard can be tolerated when it does not harm the function of a circuit. However, in asynchronous circuits, since they are transition based implementations, a tiny glitch may lead the AFSM to go to an undesired state and cause a deadlock.

The combinational hazard if felt in the design through two types of hazards, they are *static hazards* and *dynamic hazards*. Figure 18a and Figure 18b show two static hazards and Figure 18c and Figure 18d shows two dynamic hazards.

Figure 18 - Static and dynamic hazards.



Source: BROWN, S.

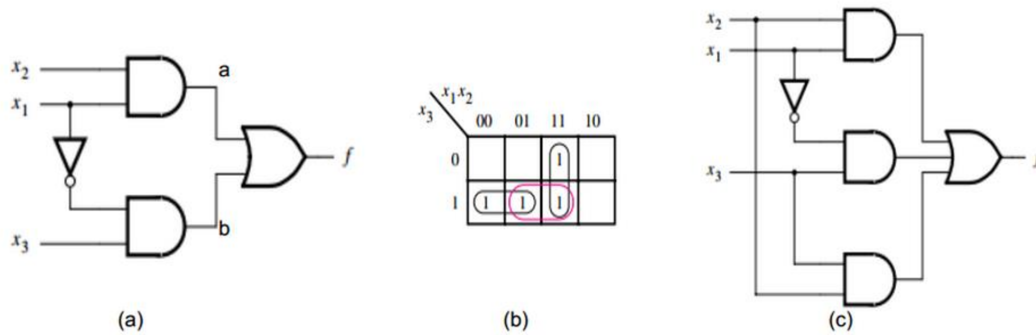
2.3.1 Static Hazard

A static hazard exist if a signal has a momentary pulse when the output should be kept in a particular logic level after one or more input variables have changed (BROWN, 2008).

Figure 16a shows a network with a static hazard. suppose that the circuit has $x_1=x_2=x_3=1$. And x_1 goes low. Now, the circuit is suppose to keep *f*, but consider that each gate has one tiny unit of propagation delay, then the change will appear in *a* first than in *b*. Therefore the signal at *a* will become low before the signal at *b* goes high. Consequently, for a tiny unit, *a* and *b* will be low, causing the function to display a 0 output.

Fortunately this static hazard can be eliminated by combining prime implicants in the K-map. The K-map for the function is depicted in Figure 16b. The original function covered just the two prime implicants that are circled in black. The hazard occurs when there is a transition from the prime implicant $\bar{x}_1 x_3$to $x_1 x_2$, since a potential hazard exists wherever two adjacent 1s in a K-map are not covered by a single product. Thus, a technique for removing hazards is to find the cover that will include both previous prime implicants and avoid the "jump" from one to another. Hence, the inclusion of a third prime implicant, pink circle, in the K-map will cover the both product terms, solving the static hazard and generating a hazard free circuit presented in Figure 16c.

Figure 19 - Static hazard.



Source: BROWN, S.

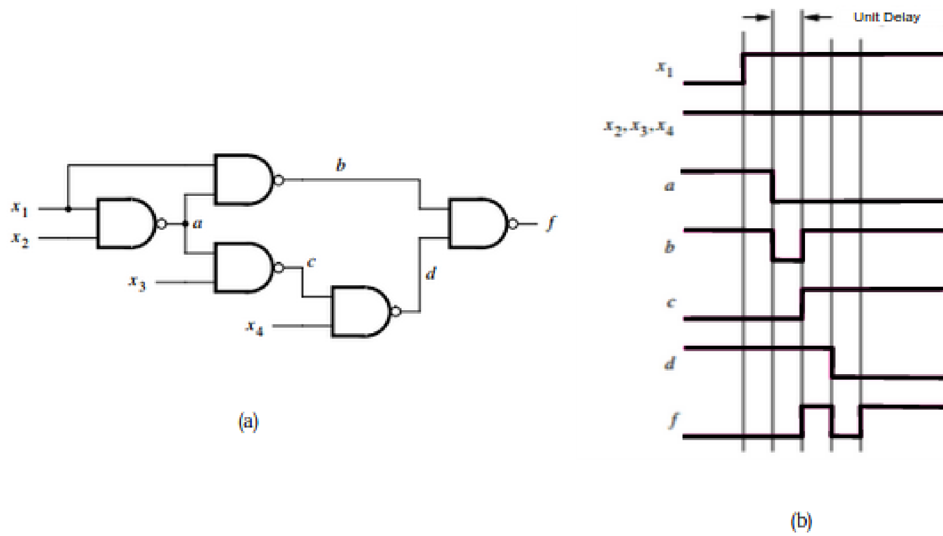
2.3.2 Dynamic Hazard

This hazard arises when the output is supposed to change from 0 to 1 or the other way around and the output experiences an oscillation before the right output value settles in the wanted level, then it is said that a dynamic hazard exists (BROWN, 2008).

The example in Figure 20a shows a circuit that has a dynamic hazard. Assume that all the NAND gates has the same delay, the timing diagram is shown in Figure 20b. Looking at the diagram it shows a glitch that was not suppose to happen. This sort of hazard is caused by the topology of the circuit, where multiple paths exist for a given signal to propagate. If the output signal changes its values three times, then there must be at least three paths along which a change from a primary input can propagate. Also, a circuit that has a dynamic hazard must also have a static hazard.

As Brown (2008) mentions, this hazards are not easy to detect nor easy to deal with. They can be avoided using two-level logic, *e.g.* sum-of-products, and cleaning the static hazards of the implementation.

Figure 20 - Dynamic hazard.



Source: BROWN, S.

2.4 SystemC

SystemC is a set of C++ classes that were written to emulate the behavior of hardware in a way the default C++ cannot. Since SC is implemented using C++ it may be compiled using gcc or g++ or any other C++ compiler available. SystemC can be downloaded at http://www.SystemC.org. The current SystemC version is 2.3.1.

Similar to C++, a SC program start the **sc_main**. The first task performed by the program is to instantiate the top modules and connect the channels. Modules can be defined as SC_MODULE classes while channels are *sc_signals*.

A the object of type SC_MODULE represents a hardware component with some input ports, output ports. It might have even more sub modules and channels.

A SC_CTOR is the constructor method for the SC_MODULE class. This method is called just in the initialization phase and its task is to instantiate sub modules and channels as well as connecting them. Furthermore, a SC_MODULE class may have internal functions, called member functions.

Member functions may model any digital logic. SC has two data types: *sc_logic* and *sc_lv< W >*. Sc_logic represents a single bit, while sc_lv represents W bits. Both data types model logic channels and may assume the states: 1 0 X Z. They are equivalent of *std_logic_vector* in VHDL.

The main role of the simulation is to coordinate the member functions that represent threads that are executed by the simulation kernel. SC_THREAD is a SC method that takes the member function and binds it to the simulation kernel. SC_THREAD is called inside the SC_MODULE SC_CTOR method.

A SC_THREAD is something like an always block in verilog. Before the simulation is actually initiated, the SC_THREAD method is called for all member functions that must be registered. At this point all member functions were registered and the simulation kernel may start executing. A SC_THREAD that has executed all its code and returned, cannot be registered again within the same simulation.

The event listener for a SC_THREAD is the SC function wait(). A SC_THREAD must have at least one wait() statement in order to be valid.

The wait plays an important role within the simulation kernel. Even though we are simulation concurrent systems, they are actually computed in a random sequential order by the CPU. Thus, the SC simulation kernel has a concept called ***delta delay*** that represent a point in time which the things happened concurrently.

SC kernel emulates a concurrent environment by orchestrating and managing the input and output values for each SC_THREAD within delay delay, a delta delay represent a given point in time where things happen concurrently.

When a SC_THREAD is executing it has the control over the simulation, to return the control to simulation kernel in order to advance simulation time the SC_THREADS. Wait() function may also represent a timer event. It accepts an amount of time as argument and after the timeout it return the execution to the SC_THREAD.

When a wait() statement is executed, the simulation kernel saves the previous state of the SC_THREAD and when it resumes, the next piece of code that is going to be executed is the next statement after the last *wait()*.

Figure 21 represents a SC 2-input NOR that is sensitive to a clock edge. Line 4 is the module declaration. Line 5 represents the input port for the clock signal. Lines 6-21 represent the actual member function of NOR that is responsible for the output generation. Lines 23-27 show the SC_CTO and the registration of the process *main_thread* using SC_THREAD method. Line 26 shows the sensitive list that causes the SC_THREAD to execute.

Figure 21 - SystemC SC_MODULE example.

```
1   #include <systemc.h>
2   #include <fstream>
3
4   SC_MODULE(top){
5       sc_in < bool >  clk;
6       void main_thread(){
7           wait();
8
9           cout << ~(SC_LOGIC_0 | SC_LOGIC_0) << " @" << sc_time_stamp() << endl;
10          wait();
11
12          cout << ~(SC_LOGIC_1 | SC_LOGIC_0) << " @" << sc_time_stamp() << endl;
13          wait();
14
15          cout << ~(SC_LOGIC_0 | SC_LOGIC_1) << " @" << sc_time_stamp() << endl;
16          wait();
17
18          cout << ~(SC_LOGIC_1 | SC_LOGIC_1) << " @" << sc_time_stamp() << endl;
19
20          sc_stop();
21      }
22
23      SC_CTOR(top)
24      {
25          SC_THREAD(main_thread);
26              sensitive << clk.pos();
27      }
28  };
29
30  int sc_main(int argc, char* argv[]) {
31      sc_clock  clk_sig("clk_sig", 10.0, SC_NS);
32      top top_i("top_i");
33      top_i.clk(clk_sig);
34      sc_start();
35      return 0;
36  }
```

Source: BUTZKE, F.S.

The *main_thread* works as follows: When the process is registered, SC_THREAD(main_thread), in the constructor, the code of main_thread starts being executed. Thus the first line, line 7, calls wait() and returns the control to simulation kernel while place line 8 as the top of the stack. When the simulation produces a positive *clk*, it knows that the process *main_thread* is waiting for it, thus it starts executing the member function from the top of the stack downwards. Thus, within a positive clock edge the design will execute lines 8-10. Since, the 10th line is another wait(), the last iteration is repeated.

The simulation will execute until the point line 11 is executed. Since there is no other code, the main_thread return to simulation kernel and destroys itself. In this example the simulation kernel has only one thread and thus it stops and return the program. In actual designs, SC_THREDS are commonly designed as infinity loops to prevent them of being canceled.

3 RELATED WORKS

In this section we present four related works that represent some similarity with this thesis. The first related work is “Min–Max Timing Analysis and an Application to Asynchronous Circuits”. It is an IEEE paper published by (CHAKRABORTY et al, 1999). The authors present a timing analyzer tool for modeling the constraints for correct operation in asynchronous circuits. Their work presents a min-max timing simulation algorithm that is employed in timing analysis with bounded component delays. Their algorithm uses an approach that each input to internal gate wire has a unique delay. They also show a technique for simulation accuracy. They design circuits using an extended burst-mode state machine as a case study. Finally their tool analyzes the gate-level design assuming bounded component delays and then set safe timing constraints.

This paper is linked to the current thesis because it proposed a min-max timing analysis for asynchronous circuits regarding their wire and gate delays using FM assumptions. The work presented an extensive analysis of asynchronous designs using a timing verification tool that simulates the circuit with a polynomial-time algorithm that approximates bounded gate delays. The paper represents a subclass of the current thesis because it simulates the environment through bounded gate and wire delays with the proposed algorithm presented by them while this thesis aims to verify the circuit correctness by simulating random delays between upper and lower bounds that go beyond the normal behavior of the components.

The second paper entitled “The Verification of Asynchronous Circuits with Bounded Inertial Gate Delays” was written by (GONG et al, 1998). The work presents a method that cover all deviation delays of gate-level implementation of asynchronous circuits.

The package developed for the paper aimed to simulate all the possible behaviors of the asynchronous circuits composed by simple gates with inertial delays under a series of input signal specified by the Signal Transition Graph. The stimulus generation focus on the input stimulus as key point. The input stimulus set may be configured as the designers want, modeling high concurrent designs that may lead to glitches.

Although these short pulses can be filtered out by the inertial characteristics of the gates, the authors say it is not guaranteed that they will not propagate to the outputs of the actual circuit, so they should be eliminated. Thus they present a technique for error correction by adding delays to the feedback path of those feedbacks that have glitches that will impact the circuit, therefore avoiding hazards. Within the input set, it is possible to specify the transition orders, with the transition order well defined it is possible to find the first wrong

gate transition and assert the reason causing this undesired transition. This information helps to add appropriate delay margin at the right places.

In short, the second work presented a method to verify the correctness of the gate-level implementations that were generated by a STG where the hazards are eliminated by padding delay elements at the appropriate nodes of the circuit.

The second paper relates to the current thesis because they present a flow from generating timing analysis and delay correction for asynchronous circuits generated through STGs that are similar to burst-mode graphs. They correct errors by adding delay constraints to the feedback path, avoiding glitches in the circuit. In this thesis we use the error metrics for detecting glitches in a similar way that was proposed by the second related work.

The third work is entitled "Verification of asynchronous circuits" by (CUNNINGHAM, 2004). The represents the PhD thesis to the University of Cambridge, where Cunningham proposes an extended formal notation to permit the use of signal levels and transitions into previous formal verification methods. The thesis proposes a event-oriented verification program called Veraci.

Within the third related work, the author introduces the concept of event driven circuits where the transitions are important concepts for the circuit to work. The circuit is said to be valid when the input events produce the intended behavior. The author then explains how formal verification should be an essential step of the design for modeling the circuit behavior using formal methods. He says that formal verification is complex and a high costing procedure and because of that, most of the time simulations are used to verify the functionality of the circuit.

There are two methodologies for designing a circuit, they are level-oriented or event-oriented methodologies. These both methodologies are used to specify behavior of circuits and in practice designers use both. This related work proposes a third variation called proposition-oriented behaviour that embraces both of those methodologies. Veraci is a proposition-oriented framework that is used to demonstrate for asynchronous designs.

The third related work relates with this thesis in the verification step, where the proposition-oriented behavior shows a relation with the proposed case study and the event driven nature of his proposed methodology. In addition the PhD thesis presented as the third related work brings many fundamental concepts and ideas about asynchronous circuits and verification that is a great resource of knowledge.

The fourth related work regards to a framework for modeling asynchronous circuits in SystemC. The work entitled“ASC, a SystemC extension for Modeling Asynchronous Systems, and its Application to an Asynchronous NoC”. It was published by (KOCH-HOFER et al, 2007). The main contribution of this paper is the Asynchronous System (ASC), the Asynchronous SystemC framework, that offers the same communication primitives as other tools, e.g. Balsa. The developed library comes with a set of arbiters that are the basic block for building Network on Chips. The aim of the paper is to offer to designers means of modeling and verifying asynchronous circuits. They present a result which shows that a Network on Chip (NoC) that was developed using the ASC library has successfully been integrated into complex Globally Asynchronous Locally Synchronous (GALS) NoC architecture.

The last related work increases the SystemC functionalities and scope by providing extra classes for improving the event driven nature of SystemC library. When the ASC is used it allows the designs to model asynchronous circuits in a way that is easier than implementing all event listeners and members functions for a given asynchronous design. It eases the process of mapping asynchronous designs to SC. This is the point where it links to the current thesis, it presents a way of mapping asynchronous designs to SystemC and how to structure the model and classes of the project. The related work offers the fundamentals for understanding and maximizing the potential advantages that SystemC has to map asynchronous circuits and verify them. Unfortunately, I have discovered this SC extension only after the work had already started, making impossible to restart all the process of design from scratch. But it was a great help because of the resources and ideas I could use within my implementation.

4 CASE STUDY

Designs that implement FSMs usually have two main blocks: control and datapath. When the datapath requires that more than one instruction to execute per cycle, it implements a pipelined datapath. The pipeline breaks the datapath in stages, each stage can be computed at the same clock cycle and store the data to registers. It allows the datapath to be concurrent.

A clocked pipeline has its registers controlled by a period signal generally called *clock* (clk). One of the pipeline requirements is to match the delay of the clock signal for combinational circuit that takes the longest time to compute through buffer insertion. It guarantees that the clock signal will be seen by all stages at the same time or at the same phase, at least. Thus, pipelines with many stages may require extra circuitry for delay matching.

An approach that may eliminate the clock problem and improve performance is to replace the clock signal by local handshake modules that will have channel interfaces that will connected to, and only to, its neighbours. Therefore, we model an asynchronous pipeline as a set of stages. In a top level view, the asynchronous pipeline simply expands by adding a local stage that is connected just to direct neighbour stages.

The simple idea of modularity does not translate the ease of designing the pipeline. To ensure that the pipeline works it is required that all pipeline controller implement the same communication protocol and they must be hazard-free.

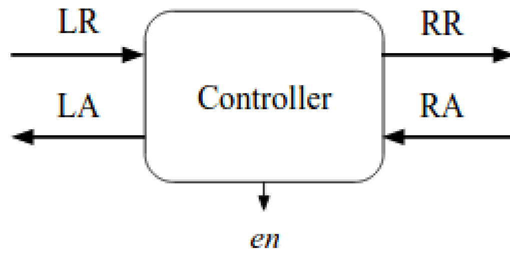
Thus, this thesis will focus in asynchronous pipeline controllers as case study to be verified with the proposed verification environment. The controller implements a 4-phase bundled data handshaking protocol and is designed according FM delay assumptions.

4.1 Burst Mode Controller Specification

Figure 22 shows the top level specification of the pipeline controller use as case study. The pipeline controller has five ports: left request (LR), left acknowledge (LA), right request (RR), right acknowledge (RA) and enable (en).

The horizontal ports represent the interface for handshaking protocol, while the *en* port represents the enable signal that is used to dictate the operation of the latch. When en is 0, the latch is said to be opaque. In the other hand, 1 means that the latch is transparent. When it is opaque, the latch holds the data and when it is transparent it sends its input direct to the output.

Figure 22 - Pipeline controller top level design.



Source: BUTZKE, F.S.

After defining the problem specification, the first step for implementing the proposed design is the BM state machine. Figure 23 depicts the AFSM for the proposed case study. The state machine has eight states.

In the initial state, *s0*, all signals are 0 but en. The en signal is 1 since the latch is transparent in the initial state, meaning it is transparent.

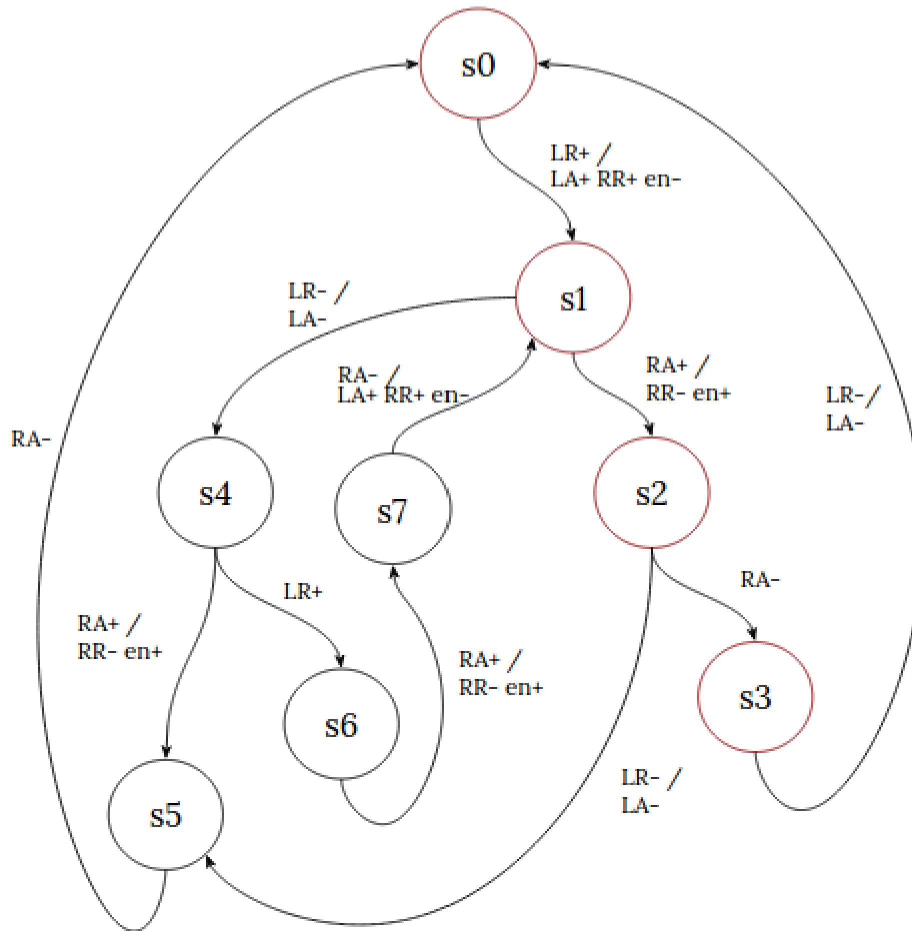
Example: Consider the current state s0, where all input and output ports are 0. Then, an event happens at LR port, where it goes from 0 to 1 represented as LR+. It produces LA+, RR+ and *en*-. Then it moves to state *s1*.

In state *s1* there are two option: RA+ or LR-. Since it assumes a single input change, as soon as one of them transitions, the state machine is going to transition to the respective next state before the other input changes. Consider that the transition RA+ happened. It means the right neighbour of this controller is acknowledging the communication and, thus, it has already saved the value of the data and no longer local storage is required. Therefore, the controller starts completing the 4-phase protocol by RR- and also reopens the latch *en+*. Then the internal state goes to *s2*.

State *s2* has also two possible transition events: LR- and RA-. Since the left channel is still 1, it could go to 0 at any time. In addition, as we have sent RR- in state *s2*, RA- could also go to 0. Suppose RA- does go to 0. Then the state machine goes to state *s3* and does not output any value.

State *s3* has only one possible transition, LR-. At this point the state machine just waits the left channel to reset the 4-phase communication protocol and then, goes back to the initial state, *s0*. The *s0→s1→s2→s3→s0* transition is highlighted with red vertices.

Figure 23 - Pipeline controller burst-mode specification.



Source: BUTZKE, F.S.

In the previous example it has been shown that there are some states that have two possible next state. Actually, there are three states where two events are expected: *s1*, *s2* and *s4*. These states represent a concurrent design. E.g. Following *s0→s1→s4→s6* the left channel accomplished a full 4-phase communication and then it receives another LR+, without even receiving the first RA+. Thus, those forks represent concurrency.

4.2 Asynchronous Finite State Machine

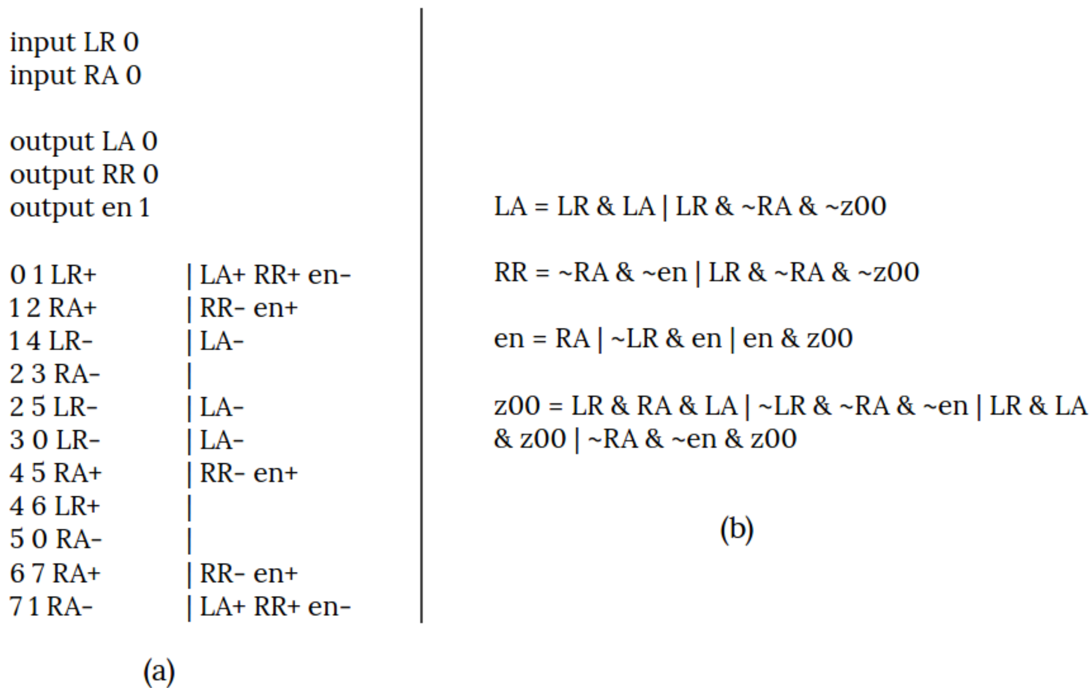
Once a valid BM specification is found, it may be translated into a flow table. As previously seen, the tools 3D ana MINIMALIST implement the requirements for translating BM into flow tables.

The proposed case study chose arbitrarily the 3D tool implement the design. Figure 24a shows the BM state machine translated into a file that represents the input file that 3D uses for generating the logic equations.

The file is composed by three parts: input variables, output variables and state transitions. The initial state for input and output ports is the number located next to them. E.g. *en* starts in 1. In addition, the the tool considers that the initial state is 0.

The state transitions are described as they are seen in the BM state machine. For each edge within the BM we specify a line in the 3D input file. Each line represents: the current state, the next state, the input burst and the output bursts.

Figure 24 - 3D tool - input and output data.



Source: BUTZKE, F.S.

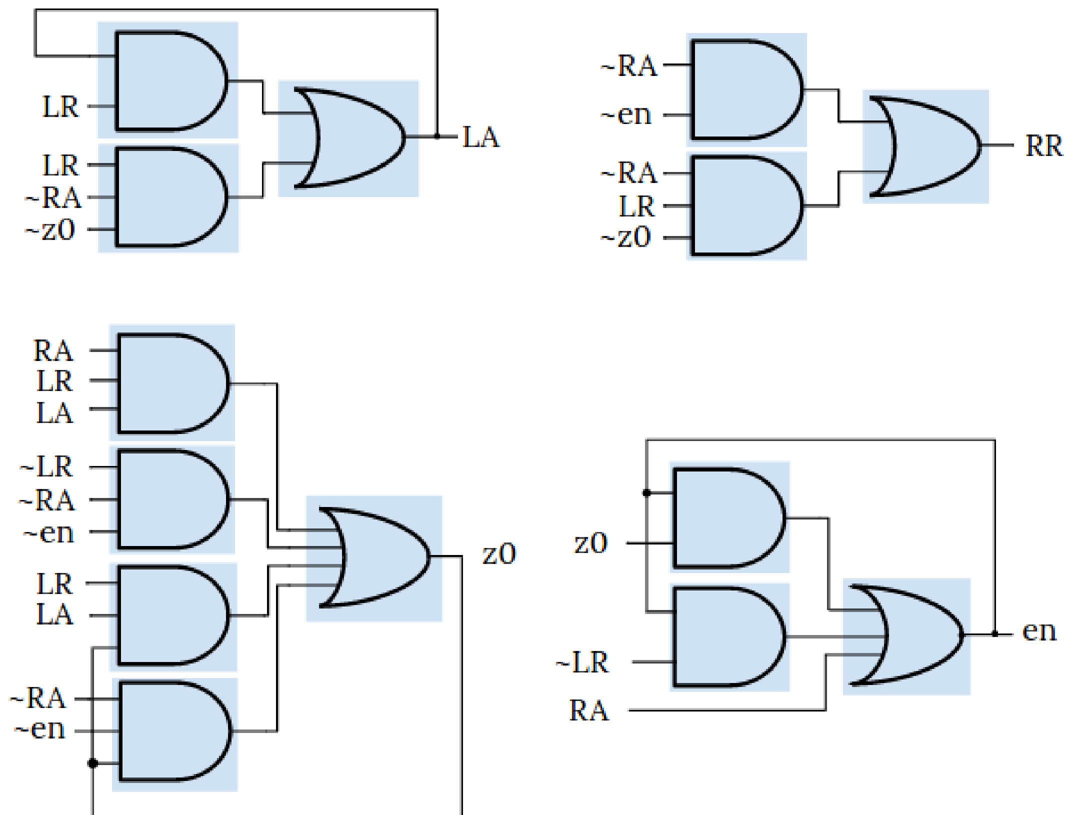
When the tool is executed it parses the data file then it produces the asynchronous finite state machine of the burst mode specification, then use some internal techniques for logic minimization and finally produces the logic equations that represent the asynchronous design.

Figure 24b depicts the output logic equations for the controller specification. It produces a logic equation for each output port as well as for the internal state, *z00*, that is used to keep the current state. The internal state variable was generated since the case study represents a concurrent design. In the other hand, if it had a very specific protocol, a unique set of input transitions occurring in a well known order, the internal state would not be necessary.

4.3 Technology Mapping

The next step generates the network of gates that will represent the controller. This process is accomplished manually in this thesis. Where the logic equations are mapped to complex gates by matching similar sum-of-products functions. Initially, translating the original logic equation of Figure 24b to a network of gates, it would result in the logic network shown in Figure 25**.** In this figure, each blue box represents a logic cell. For demonstration purposes the complement of the input ports are considered to be input ports, as well. Thus, the complemented signals containing a tilde (~) prefix are also input ports.

Figure 25 - Design implementation.



Source: BUTZKE, F.S.

In Figure 25, it is possible to note that the output of the LA, Z0 and en networks are fed back into their own circuitry. This feedback represent that the gates will always consider their past output values in addition to their input.

In addition, the signal Z is not an output port, but it is used by the other modules to ensure that the AFSM is kept in the right state. Thus Z is a “double feedback” where is used by itself and by the other logic networks.

Recall that the gates and wires have propagation delays. PVT variations usually change the capacitance characteristics of the components and thus change their delay bounds. E.g. a component has a better performance when it is cold (WESTE, 2010). Thus the inconsistency of delays from a component to another may lead to timing differences inside the circuit. Therefore, having many logic levels for each output is not a good practice, since the delays in the wires from a gate to another may lead to static or even dynamic hazards.

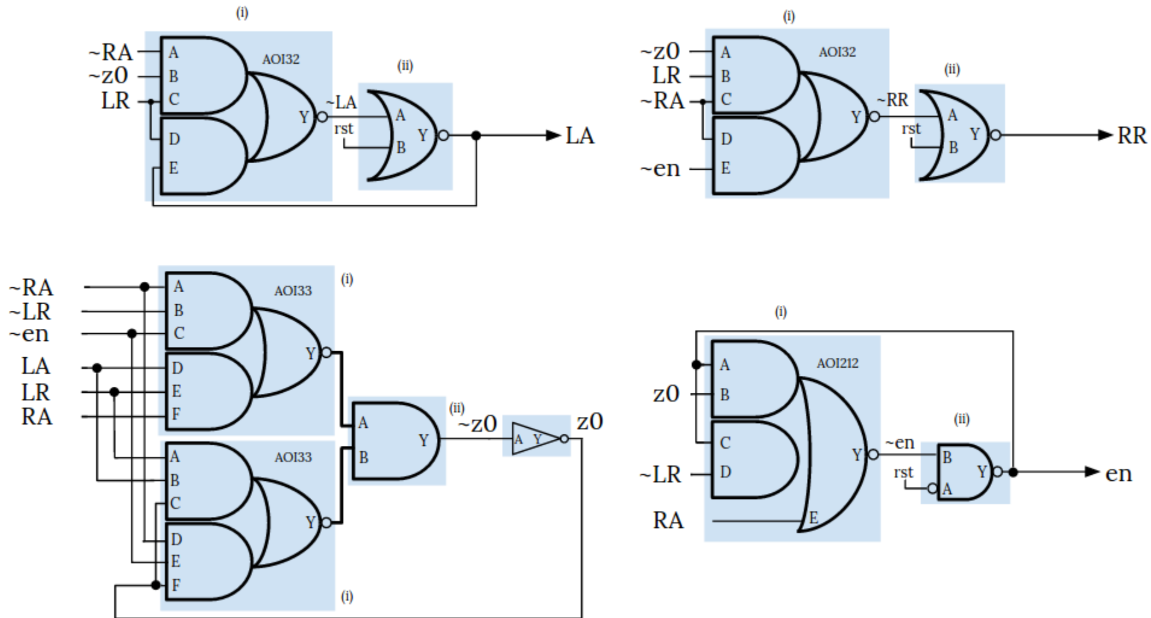
For instance, in Figure 25 z0 has two logic levels. It has four 3-input AND gates connected to a single OR gate. In this case we have four wires that need to be connected to the 4-input OR that might or might not have the same length in silicon. Thus, their capacitance may vary, varying their propagation delays. Even though the order of magnitude may be very low we cannot predict whether there will be a timing violation.

A good practice is always to try to map those logic functions to a given technology of complex gates. In this thesis a particular design kit was used to map the logic equations to complex gates. The design kit has many complex gates that implement complex logic functions and are very useful for mapping the logic equations.

Figure 26 represents the logic equations from the previous example mapped to complex gates of a given third-party design kit. In this figure, there are two main changes: i) The single gates from Figure 25 were mapped to more complex functions. ii) There are some gates that ensure the initial state or reset state of the controller.

The first part is the analysis of the complex gates. The logic network for LA had two AND gates connected to an OR. After the technology mapping, it became an entire AND-OR-INV(AOI) complex gate. It is called AOI32. This means that the top AND is a 3-input AND and the bottom is a 2-input AND.

Figure 26 - Mapped technology.



Source: BUTZKE, F.S.

The logic network for RR is also mapped with an AOI32 gate. In the other hand, the logic network for the *en* is an AOI212, where there are two 2-input AND and a third connection straight to the OR.

Finally, the internal state Z needs four different AND gates that should be connected by an OR gate. Its original logic equation has no similar complex gate. Thus, the mapping of the original functions cannot be realized directly. The solution to use a pair of AOI33 complex gates. Two 3-input AND gates connected by a NOR.

The second topic relates to those NOR, NAND and AND gates that are connected to the output ports.

Initially, suppose there exists a complex gate, g, that is an AOI. Its output function is inverted, $\bar{o}$. There is a trick using Boolean Algebra that enables complementing the output at the same time a *rst* signal resets the value. Thus, it is not required to add an explicit inverter.

When *rst* is active high, the internal states and the output ports go to their initial states when *rst = 1*. The way the reset state is accomplish within the controller design is as follows: When feedback and output ports must go to 0 a NOR gate is used: $\overline{(\bar{o}) \vee rst} \rightarrow o \wedge \overline{rst}$. In the other hand when the feedback must be 1, a NAND gate called NANDA is used. The A comes from an inverted input and is the input port *rst* is connected to: $\overline{(\bar{o}) \wedge \overline{\overline{rst}}} \rightarrow o \vee rst$.

For the LA network, LA must be 0 in the initial state, thus Thus, a NOR that inverts ~LA at the same time resets it is connected to the fanout. This design pattern is copied for RR.

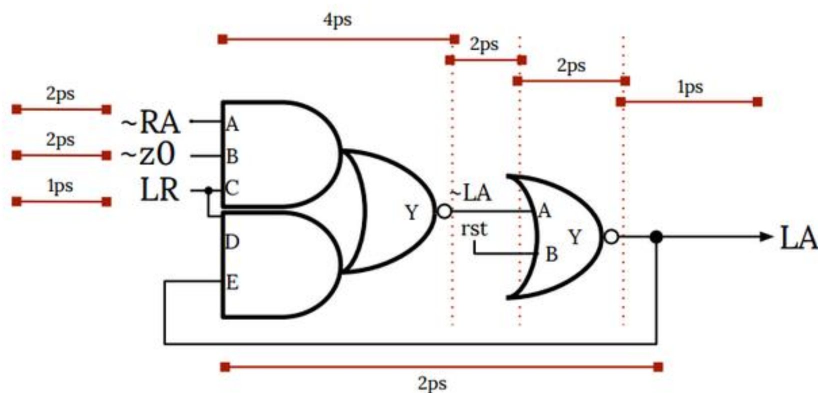
Enable output must be 1 in the reset state. Thus, the NANDA gate is used to invert *~en* and reset it. In this example, another advantage of using AOI for building the logic network is seen. Since the AOI outputs *~en* there is no need for placing an inverter in the controller for inverting the enable signal. Thus, the logic network responsible for generating *en* also generates *~en* that can be connected over the entire controller.

Finally, the internal states Z has an AND gate connecting both AOI complex gates. It represents the following function: $(\overline{aoi_A}) \wedge (\overline{aoi_B}) \rightarrow \bar{Z}$. As mentioned previously, ~en, is an useful signal because it is used in other logic networks, the same happens with ~Z. We then use a single inverter to complement the variable and generate Z.

The mapping into complex gates reduces the risk of dynamic hazards and static hazards since it reduces the number of logic levels and the timing requirements to compute the output after a given input transition.

Even though the risk is reduced, the circuit still contains many timing assumptions and path delays that must be verified. Figure 27 shows an example of the delays for computing the output LA. The delays are drawn as red lines with small squares on the edges. It is possible to notice that each part of the circuit may have one delay. For example, from the moment a signal arrives at the input port of the controller until the moment the complex gate of the LA module fells it may take 1 ps for LR and 2 ps for the inverted RA.

Figure 27 - Delays in the LA module.



Source: BUTZKE, F.S.

Finally, it has been shown that even though the controller is mapped with complex gates that perform complex logic functions, the functions still have delays. Figure 27 has the ultimate demonstration of what should be taken in account when the design is verified. Those delays for each one of the controller modules and wires may represent a great effort

requirement for verifying the controller. Thus, in the next section a verification environment for those delays is presented.

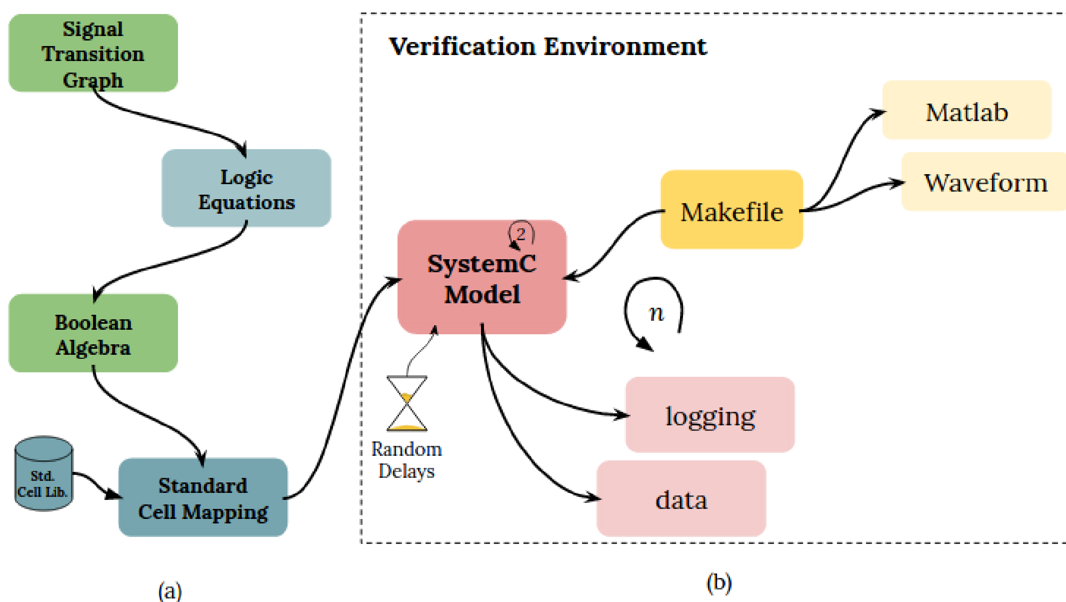
5 TIMING VERIFICATION ENVIRONMENT

This section presents the main contribution of this thesis. It presents the development steps and how they are organized for the development of the verification framework.

5.1 Overview

Figure 28 presents the different pieces that are connected to form the verification environment framework that is going to verify the pipeline controller case study.

Figure 28 - Proposed verification framework overview.



Source: BUTZKE, F.S.

The figure presents two main parts. Figure 28a shows the steps that have been followed in the case study development. Figure 28b, is the proposed verification framework. It starts with the **SystemC Model** of the asynchronous design that is the case study proposed. It represents the complex gates and the connections that have been set up in the logic network. At this point both, wires and gates, are modeled as SystemC processes. Once the model is structured and all input and output ports are connected, the system is then ready to start simulating.

The central coordinator of the framework, the Makefile script, calls the SystemC and the other support analysis tools such as MATLAB that will display the charts when the simulation is completed and generating the waveforms for visual analysis.

Within the proposed environment, a **simulation** has *n* **rounds**. Within each round new random delays are generated for the wires and gates of the System Model. Once the setup phase is ready the environment runs a testbench that places the controller in a virtual pipeline environment and starts producing stimulus for it. The testbench executes the testbench until it reaches a predefined number of interaction. An **interaction** represent a communication where the AFSM is at a given initial state and after receiving the input stimulus goes back to some initial state.

The testbench executes two **runs**. Each run accomplishes three interactions. Within each, the testbench runs two scenarios: the most concurrent communication environment and the least concurrent simulation environment.

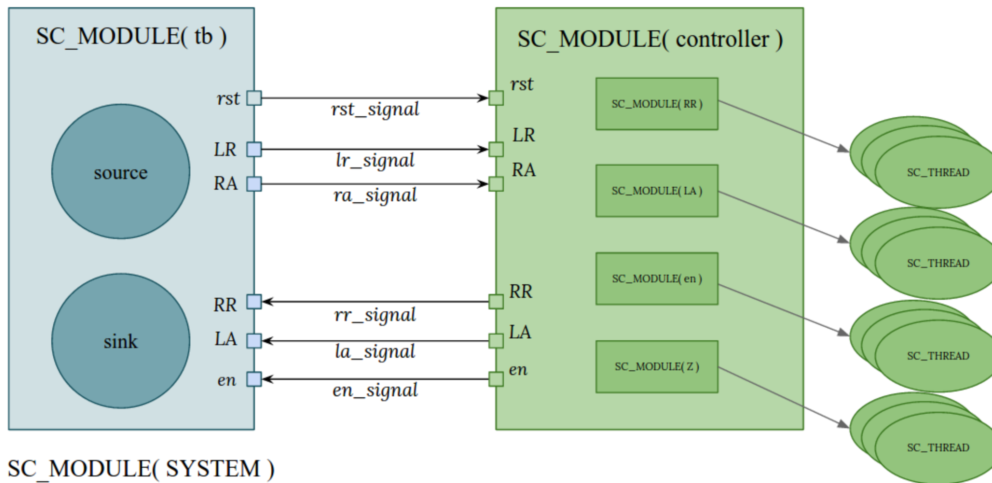
In the end of the simulation, report files and data are generated. They are used by other tools of the framework to display the results of the execution.

Once the general overview of the framework has been shown, the following sections are going to explain the specific details about the implementation. Figure 29 shows the architecture of the SystemC Model. It is composed by the top level module called **System**. The System instantiates two internal modules: **tb**, and **controller**. Both internal modules are connected through the same set of channels. The arrows indicate the data flow, from who produces to who consumes the data.

The ellipses and circles represent registered processes in SystemC. They are implemented as SystemC threads. The square and rectangular shapes are modules and are declared as SC_MODULE classes in the System Model. The inner modules in the SC_MODULE(controller) have internal processes that are also registered processes using SC_THREADS.

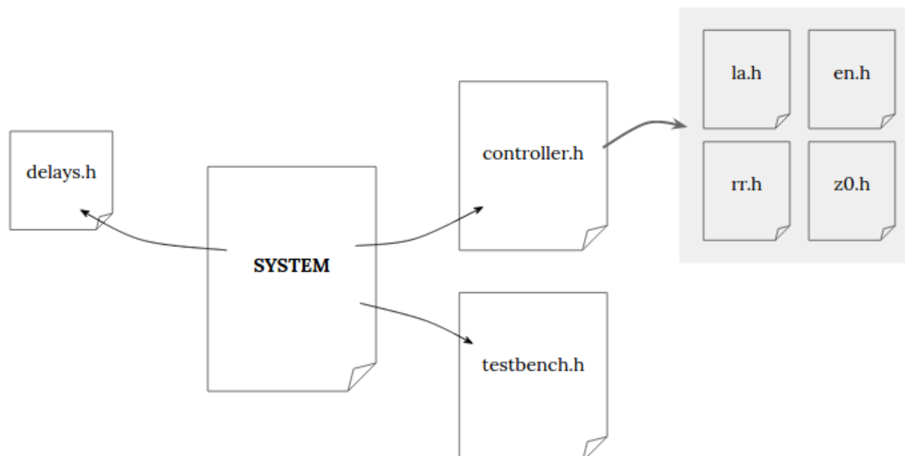
With the top most structure of the SystemC Model it is possible to show the organization of the SystemC files for the framework. Figure 30 depicts the C++ files and their hierarchy within the SystemC Model. For each logic equation and output port we have a instantiated SC_MODULE. Each module represents a new C++ header file represented with a shaded square.

Figure 29 - System implementation.



Source: BUTZKE, F.S.

Figure 30 - System implementation C++ classes.



Source: BUTZKE, F.S.

In Figure 30 we have seen that there is a specific header for dealing with delays. It is the *delays.h* file. This header represents the bound between the maximum and minimum values that the gate delays may have as well as the delay of the reset stage. For instance, Figure 31 shows the content of the header file. For example, at line 10 it is shown the maximum delay for the AOI33 gate.

For each process within each module, the processes delays will be generated by a random function that will return a value between the _MAX_ and _MIN_ for every delay constraint. Those delays represent arbitrary delays, but they could represent the actual delays caused by the gate capacitances in the designs.

Figure 31 - Content of delay.h header file.

```
3   #ifndef DELAYS_H
4   #define DELAYS_H
5
6   //Time Variables
7   const double RST_DELAY = 100.0;
8
9   //Gate Delays
10  const double AOI33_MAX_DELAY =  12.0;
11  const double AOI33_MIN_DELAY =  8.0;
12
13  const double AOI32_MAX_DELAY =  12.0;
14  const double AOI32_MIN_DELAY =  8.0;
15
16  const double AOI212_MAX_DELAY = 12.0;
17  const double AOI212_MIN_DELAY = 8.0;
18
19  const double AND2_MAX_DELAY = 8.0;
20  const double AND2_MIN_DELAY = 4.0;
21
22  const double NOR2_MAX_DELAY = 8.0;
23  const double NOR2_MIN_DELAY = 4.0;
24
25  const double NAND2A_MAX_DELAY = 8.0;
26  const double NAND2A_MIN_DELAY = 4.0;
27
28  const double INV_MAX_DELAY = 4.0;
29  const double INV_MIN_DELAY = 1.0;
30
31  const double BUF_MAX_DELAY = 4.0;
32  const double BUF_MIN_DELAY = 1.0;
33
34  #endif
```

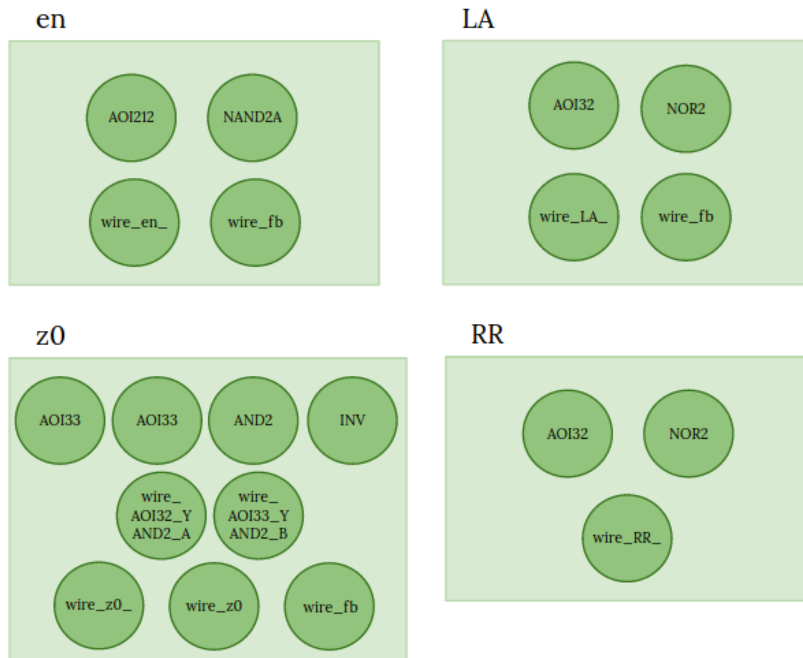
Source: BUTZKE, F.S.

5.2 Controller Module Organization

This subsections starts by presenting the controller internal modules. Figure 32 shows the four top modules that compose the controller module. In the implementation there are exactly one SC_MODULE for each one of the boxes.

Within each, there are the processes, implemented as SC_THREADS that simulate their internal behavior. For example, the module LA has four processes: *AOI32*, *NOR2*, *wire_LA_* and *wire_fb*. There is exactly one process for each one of the internal components in LA. AOI32 represents the AOI32 complex gate, NOR2 represents the output/reset gate. wire_LA_ is the output from the AOI32 and wire_fb is the LA that is feedback to the circuit. Each one of this elements will have a assigned and random delay in every round.

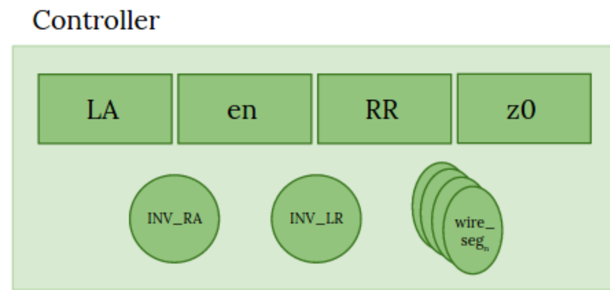
Figure 32 - SC_MODULES organization LA, en, RR, Z.



Source: BUTZKE, F.S.

Figure 33 depicts the controller top module. It shows the boxes, SC_MODULES, for each one of the output logic equations, as well as two processes that simulate inverters and a set of processes that simulate wires. The inverters, INV_LR and INV_RA, are also processes running on the controller top module. The wire processes *wire_seg* model the many wire connections between the INV_LR, INV_RA and the SC_MODULES LA, RR, en and Z. When there is a forking in some wire, there will be a wire process for each one of them, meaning that the same signal may have different timings within the controller because of its forks. Figure 34 represents the implementation of the SC_THREAD that implements the INV_RA thread. Inside the controller and the inner SC_MODULES, the delay in wires is modeled as a buffers gate, thus is very similar to the INV_RA thread but does not model the inversion behavior.

Figure 33 - SC_MODULES organization top.



Source: BUTZKE, F.S.

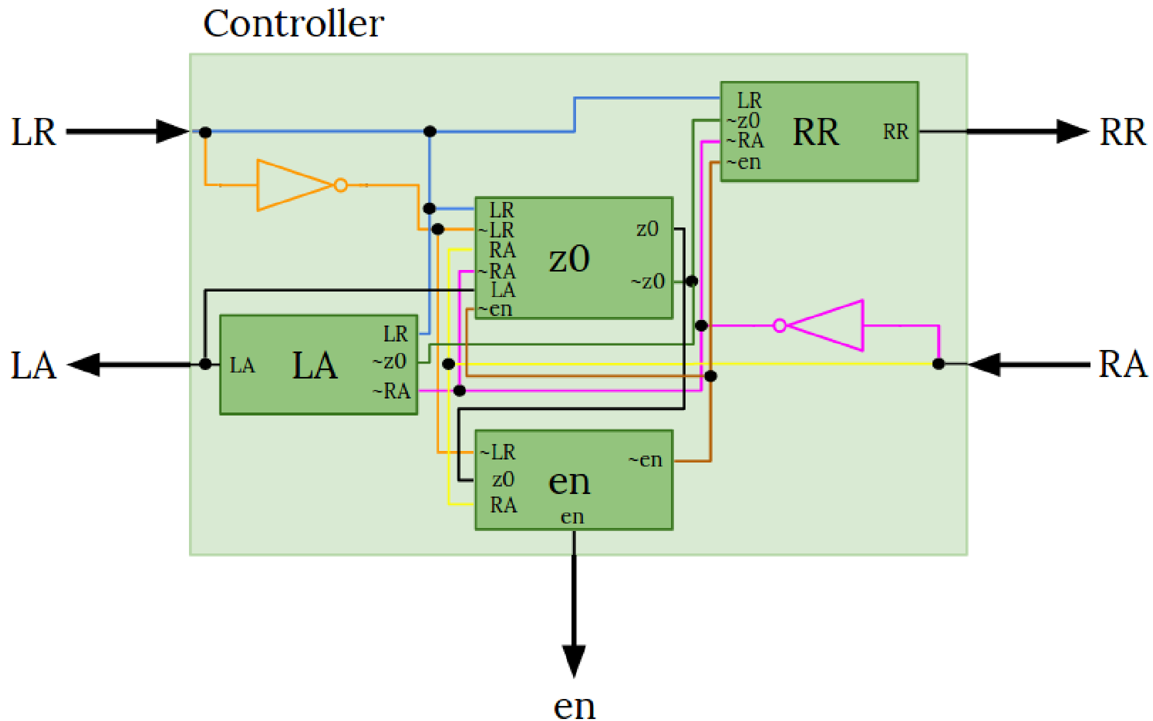
Figure 34 - INV_RA SC_THREAD.

```
1   sc_signal < sc_logic > ra_not_sig;
2
3   SC_THREAD(INV_RA);
4       sensitive << RA;
5
6   void controller::INV_RA(){
7       while(true){
8           wait(INV_RA_delay); //Gate Delay
9
10          ra_not_sig.write( ~RA.read() );
11          wait(); //Wait Sensitive Event
12      }
13  }
```

Source: BUTZKE, F.S.

Finally, the controller top level is shown with its inner channels, inverters and modules all connected together. The top module is shown in Figure 35. Within the figure is possible to analyse different things: i) With exception of LA and z0 channels, all the others have some sort of wire forks. ii) ~RA and LR both have three wire forks. iii) Modules RR and LA do not produce their complement signal. iv) Even though the output from RR, LA and *en* are next to their respective output ports, the wire connections between them are also registred processes in SystemC.

Figure 35 - Top module SC Controller with connections.



Source: BUTZKE, F.S.

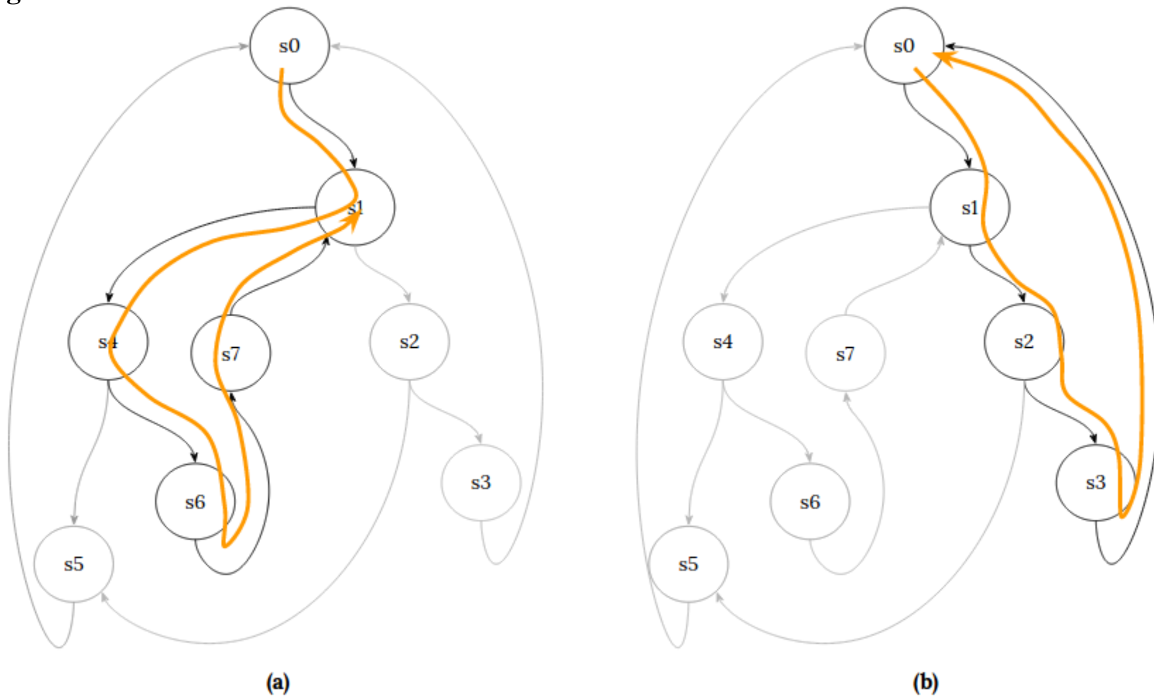
Figure 35 has shown the connections between modules inside SC_MODULE(controller) class. It gives a great overview and idea of the magnitude and complexity of wire connections and delays that are involved within a design.

5.3 Testbench Module Organization

The testbench has two processes: *sink* and *source*. The stimulus testbench is composed by two **runs**. Each run represents a different pipeline behavior. Figure 36 represents the BM state diagram presented in the case study development. Here the transitions were omitted to ease the reading. Figure 36a and 36b represents both behaviors. In the first the BM state machine is going to test the least concurrent flow: $LR \preccurlyeq LA \preccurlyeq RR \preccurlyeq RA$.

The second run, places the controller in the most concurrent environment where the left channel is at a different phase compared to the right channel, meaning that it requests a new communication without the right channel being ready.

Figure 36 - Testbench stimulus.



Source: BUTZKE, F.S.

The sink process is responsible for consuming the data that the output channels of the controller models are sending: RR, LA and *en*. While the source thread represents the stimulus that the controller will receive: LR, RA and *rst*.

All ports have the same name as their respective channels and the channels represent the controller ports. Thus, the port naming convention for the testbench is actually the name of the port of the controller.

The sink thread actually is divided in two processes. The testbench implements a sink thread for the RR port and a thread for the LA port. Both threads are programed in a way that they act independently from each other.

The source thread is presented in the Figure 37. The source thread algorithm is executed only once at the start of the simulation. The only attribute of this thread is to ensure that the controller is in its initial state.

The sink thread that represents the right channel, *sink_rr*, is the responsible for stopping the current simulation. It checks when the simulation has reached the limit of tokens that were transmitted along the pipeline. In addition, it signals to *sink_la* that RA have been lowered and thus allowing the sink_la to produce new requests while in the least concurrent environment.

Figure 37 - Source thread.

```
1  void testbench::source(){
2      wait(SC_ZERO_TIME);
3
4      rst.write(SC_LOGIC_1);
5      wait(RST_DELAY, SC_PS);
6
7      rst.write(SC_LOGIC_0);
8      rst_evt.notify();
9  }
```

Source: BUTZKE, F.S.

Sink_la is also important. It controls the both runs within a round. It starts with the least concurrent environment and then executes a predefined number of communication exchanges with the controller, half of the number of tokens exchanged by the round, and then changes the run setup to the most concurrent.

5.4 General Features

In this subsection we describe how the verification environment generates the output reports and data as well as the metrics it uses for detecting errors and warnings.

When the design is first created, there is a limit of number of communication exchanges that are going to be completed by the testbench. The limit represents how many data exchanges will be simulated within the controller. Therefore, the first error metric is whether the number of communication realized is the same as the limit. In case it presents a different value the current round of simulation is stopped and a new round starts, reporting the error. This may be caused by a deadlock in the design, where no transitions happened since the controller state machine is in a state it cannot change.

The secondary verification metrics are the comparison of some event counter:$LR\downarrow == RA\uparrow \,\&\, RR\; == LA$.

If any of them assert false, then the current round has an error. The error may be a glitch that occurred making the AFSM to lose its internal state and send an output more than once. Other possible cause is that the communication is too fast. Figure 38 depicts the algorithm that is responsible for returning whether the round had the expected execution metrics or not. Line 16 shows the logic expression representing the number of tokens exchanged and the comparison between the number of iteration over each channel. If any one of them asserts false, the current controller setup is considered invalid.

Figure 38 - Error detection algorithm.

```
1  //Return
2  //  true, 1, if controller had expected behavior
3  //  false, 0, otherwise
4  bool isAValidController(){
5      if (num_of_LR_down != num_of_RA_up)
6      {
7          errorlog << "[Error. #LR down != #RA up]" << endl;
8      }
9      if (num_of_RR != num_of_LA)
10     {
11         errorlog << "[Error. #RR != #LA]" << endl;
12     }
13     if(num_of_tokens_exchanged != 6){
14         errorlog << "[Error. Incorret Number of Tokens Have been Exchanged.]" << endl;
15     }
16     return ((num_of_tokens_exchanged == 6) && (num_of_LR_down == num_of_RA_up) && (num_of_RR == num_of_LA) );
17 }
```

Source: BUTZKE, F.S.

In addition, the simulation environment registers two SC_THREADS within each module to verify whether the output is undefined, X or glitching. The threads are: *check_output_x* and *check_output_glitching*. The former checks the output state after every input signal has changed. It is like an always block with all signals that represent the module being the sensitive list. This represent an error since the output function cannot assume undefined state after the reset. The check_output_gltiching thread checks the changes in the output value after the input ports have changed. If the output changes more than once after an input change, there is a hazard.

Figure 39 shows the code for detecting whether the output is glitching or not. Lines 1-5 shows the member function that is registered as a SC_THREAD. And lines 7-26 represent the actual verification function. Basically we check if the output changes more than once when the module had only one input transition.

Figure 40 shows the piece of code for detecting the output in an undefined state. Lines 1-3 represent the member function registration processes. The actual code is from lines 5 through 15. The code checks the output state after each transition on it. If it asserts an undefined state after the reset period then an error is flagged.

Figure 39 - Algorithm for checking if the output is glitching.

```
1   SC_THREAD(check_output_glitching);
2       sensitive   << en
3                   << lr_
4                   << z0
5                   << ra;
6
7   void EN_MODULE::check_output_glitching(){
8       //Wait for sensitive list
9       check_output_glitching_counter = 0;
10      wait();
11
12      while(true){
13          wait();
14
15          if(en.event()){
16              //If the output has changed
17              if(check_output_glitching_counter++ == 2){
18                  sc_stop();
19              }
20          }
21          else{
22              //If an input has changed meanwhile
23              check_output_glitching_counter = 0;
24          }
25      }
26  }
```

Source: BUTZKE, F.S.

Figure 40 - Algorithm for checking the output state.

```
1   SC_THREAD(check_output_x);
2       sensitive   << en
3                   << rst;
4
5   void EN_MODULE::check_output_x(){
6       while(true){
7           wait();
8
9           if(rst.read() == SC_LOGIC_0){
10              if(en.read() == SC_LOGIC_X){
11                  sc_stop();
12              }
13          }
14      }
15  }
```

Source: BUTZKE, F.S.

After every round, the testbench will return the value of 1 whether there was a malfunction or 0, otherwise. Then, the Makefile script is responsible for the sum of all return values for all simulation rounds. In the end of the simulation the script will produce an output with the total number of runs and the total errors.

Makefile script is the responsible for coordinating the execution of the rounds and organizing the data reports. The piece of code that the script executes while in simulation is presented in Figure 41. Line 5 controls the loop and breaks it when the round counter, *$$i*, is greater than the number of rounds. Then lines 6-10 represent the iterations for each round. Line 8 executes the program that is the SystemC Module and output the round log to a log file. Line 9 represents the error counter. It sums the return value of the previous command and add to the total_errors variable. Then the iteration counter is incremented in line 10.

Figure 41 - Makefile script.

```
1 ...
2 all:
3         total_errors=0; \
4         i=1; \
5         while [ $$i -le $(NUMBER_OF_ROUNDS) ]; do \
6                 echo "\n\tRunning Round: $$i!\n"; \
7                 sleep 1.0; \
8                 ./$(NAME) $$i > $(LOG_FILE_ROUND); \
9                 total_errors=$$(($$?+total_errors)); \
10                i=$$(($$i+1)); \
11        done;
12 ...
```

Source: BUTZKE, F.S.

The set of warning messages verifies whether the LA happens before *en* port closes the latch. In the framework environment it represents a warning since the designer may add delays to LA to ensure it happens at the same time or after *en*. Figure 42 shows the piece of code that detects if LA happens before en closes the latch. Line 6 shows when that verification is accomplished, if en is not 0 it means it is still high even though LA has acknowledged the communication.

Figure 42 - Algorithm for generating a latch hold time warning.

```
1   ...
2   while(true){
3       LR.write(SC_LOGIC_1);
4       wait(LA.value_changed_event());
5
6       if(en.read() != SC_LOGIC_0){
7           errorlog << "[Error. LA happened Before EN @ " <<
                sc_time_stamp() << ", buffers are required at LA
                output]" << endl;
8       }
9   ...
```

Source: BUTZKE, F.S.

The metrics that we have adopted in order to verify the design for every round are the **Total Delay** and **Protocol Latency**.

The total delay represent the sum of the delays for modules inside the controller. The protocol latency in the other hand, measures the timing average of the transitions $lr \downarrow la \downarrow$ and $ra \uparrow rr \downarrow$. Those transitions are used since when $lr \downarrow$there is only one possible outcome from the controller that is $la \downarrow$. While when $ra \uparrow$the only possible output is $rr \downarrow$. The average of this delay represents the real delay of the controller, since they represent the communication speed in both channels independently.

Once the simulation is completed there exists some report files that are generated by SystemC. The main files are *error.log* and *simulation.log*. Error.log keeps track of the errors, warnings and unexpected behaviors. Simulation.log displays the delays assigned for each wire/gate within the round as well as the metrics and general information for validation purposes. Figure 43 show the syntax of the simulation.log (a) and error.log (b) files.

Figure 43 - Log files.

```
########
ROUND 1

Randomizing Modules' Gate Delays...
LA AOI32_thread_delay: 14 ps
LA NOR2_thread_delay: 11 ps
...
Z wire_AOI33b_AND__thread_delay: 2 ps
Z wire_fb_thread_delay: 2 ps

Randomizing Controller INV and BUF
delays...
INV_LR_delay: 4 ps
INV_RA_delay: 8 ps
...
en_not_buffer_rr_delay: 3 ps
rr_buffer_RR_delay: 4 ps

Standard Deviation for the Delays in Gates:
2.72213
Tokens exchanged: 6
#LR down: 6 #RA up: 6
#RR total: 13 #LA total: 13

########
```

(a)

```
########
ROUND 1
[Error. LA happened Before EN @ 218 ps,
buffers are required at LA output]

########
```

(b)

Source: BUTZKE, F.S.

In addition, besides textual reports, the framework provides a set of waveforms. It traces all channels for every module in the controller hierarchy. Thus, we can verify visually glitches and whether the protocol is working or not.

5.5 Summary

Finally, this section has presented the proposed environment. We have seen the overview of the framework as well as how controller and testbench modules are organized. In the last subsection it was presented the concept of error and warning as well as the metrics the proposed work implements.

6 RESULTS

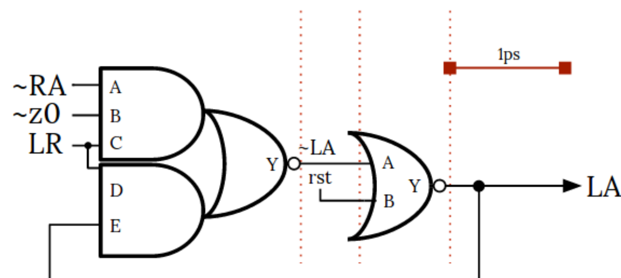
This section presents the results this thesis has obtained. Two main simulations have been setup. For all simulations that use random gate delays, the following parameters are default:

- Number of Runs per Round: 2.
- Number of Tokens per Run: 3.

6.1 Simulation 1

The first simulation verifies the BM design specifying a bounded gate delay only for the fanout of each network. For instance, the NOR gate in the LA network is the only element of module LA to have an assigned delay. This represents that all the wires and internal gates have zero delay. Figure 44 depicts the setup for the delays in the controller modules. Here only the fanout delay is considered while the other gate and wire delays are considered to be null.

Figure 44 - Delay model for the modules in first simulation.



Source: BUTZKE, F.S.

Since the output gate delay assume an unique and arbitrary value, there is the need for only one round. The simulator output is presented in Figure 45. The simulation output presents the phases it is currently running and by the end it presents the error counter compared to the total number of rounds.

Figure 45 - First simulation output.

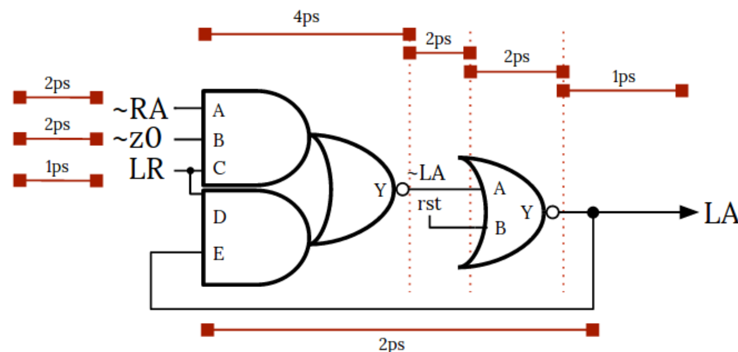
```
************************************
 COMPILATION SUCCESSFUL
************************************

************************************
SIMULATION

Running Round: 1!

************************************


************************************
SIMULATION COMPLETED
Errors: 0 out of 1 rounds.

Check:
Simulation Log: simulation.log.
Error Log: error.log.
Compiler Log: round.log.
Waveform: make wave.
Data Analysis: make plot.
************************************
```

Source: BUTZKE, F.S.

6.2 Simulation 2

This simulation implemented the random gate delay generator for all wire and gate threads. The range of delays used was presented in Figure 31. The presented delays are just a Proof of Concept (POC) that any bounded values could be "plugged in" and simulated within the framework. Figure 46 shows all sort of timing variables the second simulation is going to consider while running the simulation. The figure shows the logic network that represents the LA module. All sort of delays are considered to exists in some way. For example the delay from an input signal from the time it arrives to the controller until the point it is presented at the LA module input ports.

Figure 46 - Delay model for the modules in second simulation.



Source: BUTZKE, F.S.

In this simulation the framework will simulate 1000 rounds. When the simulation is completed the framework displays the content message shown in Figure 47. It shows that the simulator has run 1000 rounds and 81 of them presented some sort of anomaly. Those anomalies may be glitches, deadlocks or represents that the AFSM has lost its internal state and have produces more RR or LA than LR and RA transitions.

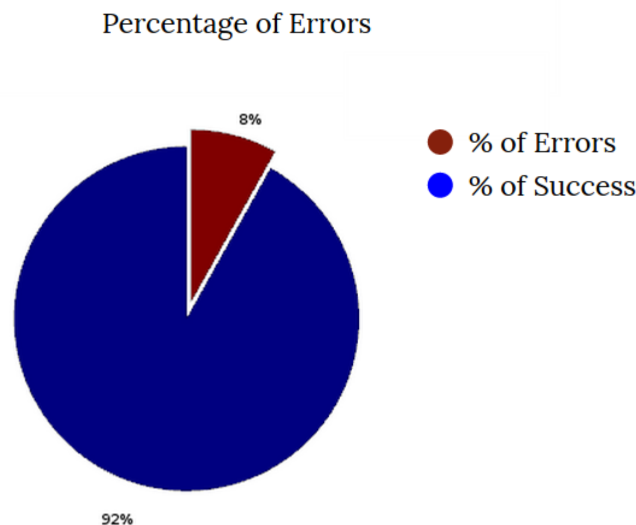
Figure 47 - Second simulation output.

```
************************************
SIMULATION COMPLETED
Errors: 81 out of 1000 rounds.

Check:
Simulation Log: simulation.log.
Error Log: error.log.
Compiler Log: round.log.
Waveform: make wave.
Data Analysis: make plot.

************************************
```

Source: BUTZKE, F.S.

The pie chart presented in Figure 48 depicts the amount of errors compared to the total number of rounds.

Figure 48 - Second simulation - percentage of errors chart.



Source: BUTZKE, F.S.

The next output data provided by the tool is shown in Figure 49. It presents the spectre of the total gate delays per round of simulations. Each black dot represents a simulation. The y-axis represents the total delay for the controller design. The red dash lines represent the

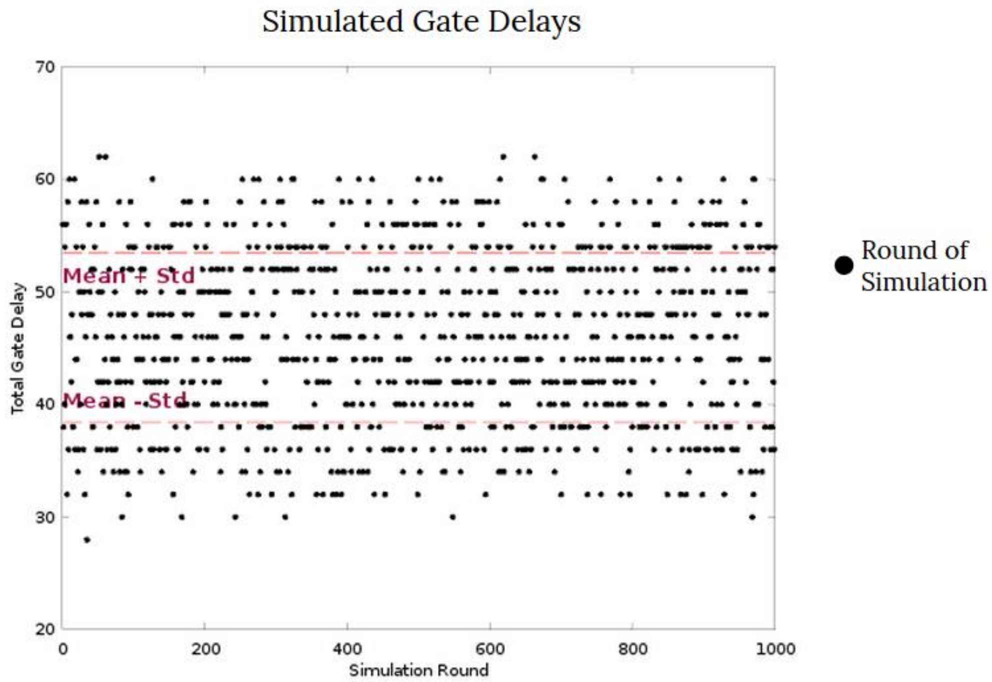
standard deviation for the mean of the total delay. The main reason we have presented this chart is because it shows that the simulation framework has stimulated a considerable set of different controller delays and did not focus in a small set of possible values.

The last information displayed from the simulation is the chart shown if Figure 50. Each plus and times signs represent a given simulation. The plus sign means the simulation was a valid. While the times sign represents a fail simulation. In the x-axis we have the standard deviation for the delays inside the modules of a controller within a round. It tells whether the random generated delays represented a well spaced delays (higher standard deviation) or whether it generated only close delays (lower standard deviation). In y-axis we present the protocol latency.

The last chart was intended to show a visual proof that the higher standard deviation of the inner gate delays the higher are the chances of the controller to be fail. The set of rounds that contain errors are concentrated in the top right corner, meaning that the controllers with a high standard deviation within their internal gate delay are likely to have more errors than the others that have a average timing for the internal gates and wires.

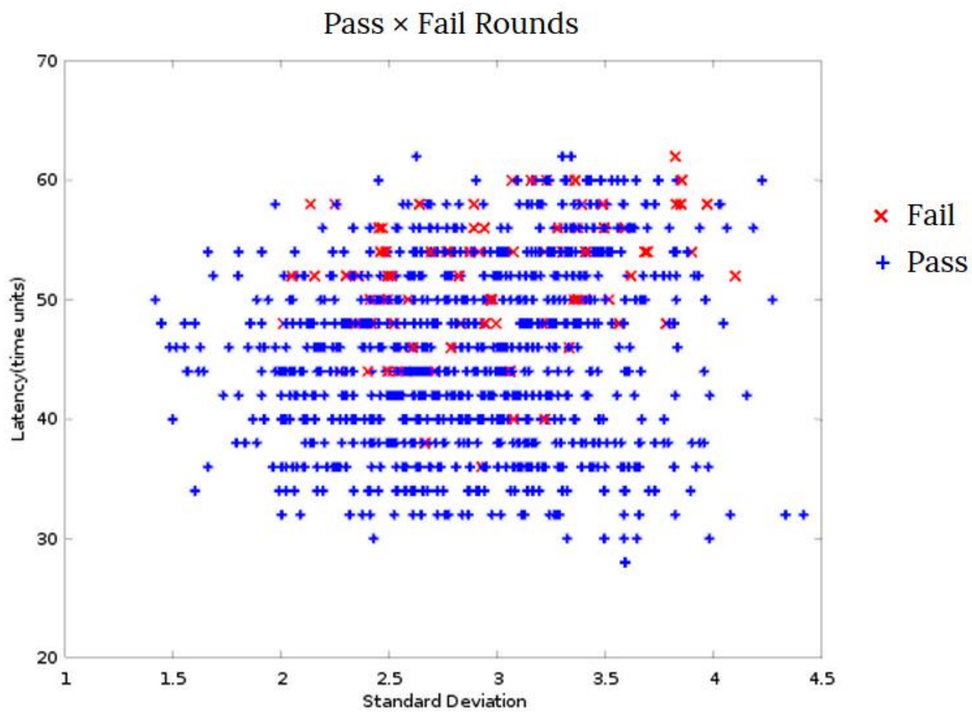
Being on the top means that the average protocol latency is high. The reason for this to happen is that the modules that are affected by the high delays are the modules that are directly connected to the channels. It means that the right and left channels are affected by the delays of that controller. Thus the top right means that the inner gate delays have a very spaced delays assigned to them and the highest delays are the delays that are from the modules responsible for communicating with the neighbor controllers.

Figure 49 - Second simulation - simulated gate delays.



Source: BUTZKE, F.S.

Figure 50 - Second simulation - pass x fail rounds.



Source: BUTZKE, F.S.

6.3 Discussion

In this subsection we analyze the obtained results. There were two main simulations that have been run for this thesis.

The first simulation regards to setting a static delay to the end of each SC_MODULE that represents a specific function, e.g. LA, where the module only has a fanout delay. Thus, for this simulation it is obtained only one result. The result presented in Figure 45 has shown a pass simulation. The behavior was expected since only the fanout ports have delays. Hence, the wires and inverters that are the fanins of have zero delay and are seen at all gates at the same time.

The second simulation represents the verification environment working at its full capacity. For this simulation we prepared a set of 1000 rounds and specified a range of possible gate and wire delays that would be specified at random by the simulation kernel to the processes.

The result of this simulation has thrown 81 errors. It means that the number of tokens, *reqs* and *acks* have not matched. This behavior represents a protocol violation. Thus an error is flagged for that particular round and the error counter is increased by 1.

The chart presented in Figure 49 shows the different set of rounds the controller has being simulated. Looking at the graph it is possible to verify that the controllers that are assigned with well spaced delays, higher standard deviation, have a concentration of fail simulations. This is because having delays with many time units of difference may lead to glitches and erroneous state transitions. Also the simulation applies different propagation delays for the wires forks. Thus the same signal may be felt by two modules at different times causing unexpected behaviors in the design.

In addition, looking at Figure 50 is possible to trace a trend line or pattern that is the higher the delay variation the higher are the chances of errors. Furthemore, the errors are concentrated to the top right, if the delay variation increases, increasing the standard deviation, the errors are likely to ¨walk" to the top right.

It is worth to mention that the both simulations were ran to simply ensure the simulation environment would work properly. The second simulation had a MIN to MAX delay variation more spaced than usual, to simulate extreme variation delays. Since the environment does not apply an exhaustive verification errors might occur even though the environment might end the simulation with a valid controller. The odds of errors of the controller after the verification rely on the probability of errors and cases it did not cover

while simulating. The designer should keep in mind that the more rounds he simulates the less is the uncovered error probabilities. Thus the 1000 simulations chosen for the second simulation cover more cases than it would be covered by 100 simulations. In the other hand 10000 simulation rounds would cover more cases than 1000 rounds.

Finally, for the second simulation and its purpose 1000 was the most suitable number of rounds since the purpose was to simulate extreme variation environments to check if errors would be caught while covering a good portion of the possible variations.

6.4 Summary

Even though we have no mechanisms for error correction and detection within the proposed environment, the framework has the ability of generating log informations that may help the designers in the verification process. It tells which rounds errors have happened as well as the exact gate and propagation delays setup, enabling the recreation of the environment. In addition, the warning messages aim to be a feedback that the designer receive from the simulation telling that some propagation delays are faster than others.

Finally, the main contribution of this thesis shown that the framework has presented the intended behavior and achieved successfully its purpose that was showing whether a controller is or is not a valid design. At the end the tool tells whether we had or not an error. It also provides the log that help debugging the design.

7 CONCLUSION

The purpose of this thesis is to implement a verification environment for asynchronous circuits that enables the verification with random gate and wire delays. The original idea came from the feeling that the market is moving towards new design methodologies in a near future. This work tries to fit in this growing field of study. It attempts to be a contribution for inspiring future designers and to help finding novel testing methodologies.

This thesis has presented two main sections, Section 4 and 5. They represent the two different pieces that were connected in order to make this work a reality. Section 4 presented an Asynchronous Pipeline Controller as the case study for this thesis. It represents a design methodology starting with an BM state machine down to the technology mapping. Section 5 presented the verification environment composed of SystemC classes, scripts and support tools that enable the verification of the case study proposed in the Section 4.

The fundamentals and related work were presented in Sections 2 and 3. Section 2 presents a general introduction to asynchronous circuits and compared this sort of design against the current design standard, the synchronous design. Also general guidelines for building an asynchronous systems starting with a state machine definition is introduced. In addition, many terms and concepts were also introduced. Section 3 presents some related work that attempt to accomplish similar objectives in the field of asynchronous circuits.

Section 5 presented the results of the simulation of the case study within the proposed framework. We have seen that the set of output informations (logs, warnings, errors and charts) show whether the controller design is valid or not. Thus, they represent the ultimate validation and simulation status report. If there is an error within those informations, the the controller must be analyzed or restructured to avoid the hazards and errors that occurred.

7.1 Objectives Check

- **Implement a simulation environment.**
  Objective accomplished. Section 5 shows the steps and the proposed framework.
- **Define a case study**.
  Objective accomplished. Section 4 presents the specification and design of the asynchronous pipeline controller..
- **Verify a case study using the proposed environment**.
  Objective accomplished. We have verified the case study within the framework and we have run a simulation containing a thousand rounds. Section 6 presents the results of the simulations.

7.2 Future Work

In this subsection we present some topics that would be great case studies or features in the framework.

A possible case study might be the implementation of an array of pipeline controllers. They would represent a pipeline design and the focus would be the generation of random delays for the individual controllers to verify whether the pipeline presents faults when we introduce random gate delays to it. In addition this framework is extensible for other asynchronous designs. Since it creates random delays for the inner module through SystemC classes, we would have just to model new processes and modules to represent other designs.

There are some features that were not the focus of the present thesis but would be great tool for the proposed framework. The first feature would read a given standard cell library and would generate automatically the delays for gates. In addition we could use automatic tools for generating the technology mapping that would ease the generation of case studies and models that are verified by the environment.

8 FINAL THOUGHTS

Asynchronous circuits represent a design methodology that the author believes is going to be in the future designs. This sort of circuits are not becoming popular alone, research groups around the World are also researching topics as complex system-on-chips and 3D circuits. However, if those designs do become popular, they would also need novel ways to overcome their problems. Thus asynchronous is again a solution for the technologies that are growing at this very instant.

Asynchronous circuits represent a personal interest of the author. He has decided to use his bachelor’s thesis to improve his knowledge in the subject as well as trying to contribute with ideas and tools.

In addition, choosing SystemC as the description language of the framework represented a huge challenge for the author. In the undergraduate courses we focus especially in VHDL and Verilog hardware description languages. Thus, learning a third syntax gives even more academic experience for the author.

The author is very pleased with the developed framework and the subjects he has developed. He thinks the proposed implementation was a great choice for being developed and presented as his bachelor's thesis.

9 REFERENCES

BEEREL, P. A.; DIMOU, G. D.; LINES, A. M. Proteus: An ASIC Flow for GHz Asynchronous Designs. In: IEEE DESIGN & TEST OF COMPUTERS, [S.l.], 2011. p. 36-51.

BEEREL, P.; OZDAG, R.; FERRETTI, M. A Designer's Guide to Asynchronous VLSI. 1ª Ed. New York: Cambridge, 2010.

BLACK. D.; DONOVAN, J.; BUNTON, B.; KEIST, A. System C: From the Ground Up. 2ª Ed. New York Springer, 2010.

BROWN, S.; VRANESIC, Z. Fundamentals of Digital Logic with Verilog Design. 2ª Ed. New Delhi: Tata McGraw-Hill Publishing Company Ltda., 2008.

CANNIZZARO, M.; Lavagno, L., "PID (Partial Inversion Data): An M-of-N Level-Encoded Transition Signaling Protocol for Asynchronous Global Communication," Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on , vol., no., pp.134,141, 7-9 May 2012.

CHAKRABORTY, S.; DILL, D.; KENNETH, Y. Min–Max Timing Analysis and an Application to Asynchronous Circuits. Proceedings of the IEEE, vol. 87, no. 2, February 1999.

CUNNINGHAM, P. A. Verification of asynchronous circuits. Ph.D. Dissertation. University of Cambridge, Cambridge, United Kingdom, 2004.

FUHRER R. M. , NOWICK S. M. , THEOBALD M., N. K. JHA, B. LIN, and L. PLANA, “Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines.” Technical Report, CUCS-020-99, Columbia University, NY, 1999.

FURA, D.; HILL, T.; RAFTERY, M., "Design and validation of a fault-tolerant distributed clock," Aerospace and Electronics Conference, 1988. NAECON 1988., Proceedings of the IEEE 1988 National , vol., no., pp.495,502 vol.2, 23-27 May 1988

FURBER, S.B.; DAY, P; GARSIDE, J.D; PAVER, N.C.;TEMPLE, S.; WOODS, J.V. The design and evaluation of an asynchronous microprocessor. In Proc. Int’l. Conf. Computer Design, pages 217–220, October 1994.

FURBER, S.B.; GARSIDE, J.D.;TEMPLE, S.; LIU, J.; DAY, P.; PAVER, N.C. Amulet2e: An asynchronous embedded controller. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 290–299. IEEE Computer Society Press, 1997.

GONG J.; WONG M.C. E. Verification of Asynchronous Circuits with Bounded Inertial Gate Delays.

KESSELS, J.; PEETERS, A.; WIELAGE, P.; SUK-JIN K., "Clock synchronization through handshake signalling," Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on , vol., no., pp.59,68, 8-11 April 2002.

KOCH-HOFER, Cedric at al. ASC, a SystemC extension for Modeling Asynchronous Systems, and its application to an Asynchronous NoC. Proceedings of the International Symposium on Networks-on-Chip vol. 1. 2007.

MARTIN, A. J.; NYSTRöM, M.; WONG, C. G. Three Generations of Asynchronous Microprocessors In: IEEE DESIGN & TEST OF COMPUTERS, [S.l.], 2003. p. 9-17.

MARTIN, A. J; NYSTRöM, M. Asynchronous Techniques for System-on-Chip Design. In Proceedings of the IEEE. vol. 94. 2006. p. 1089-1120.

MARTIN, A.J. Compiling communicating processes into delay-insensitive VLSI circuits. Distributed Computing, 1(4):226–234, 1986.

MARTIN, A.J.; BURNS, S.M.; LEE, T.K.; BORKOVIC, D.; HAZEWINDUS, P.J. The first asynchronous microprocessor: The test results. Computer Architecture News, 17(4):95–98, 1989.

MELY, C.; SHIH-HSU HUANG, "A reliable clock tree design methodology for ASIC designs," Quality Electronic Design, 2000. ISQED 2000. Proceedings. IEEE 2000 First International Symposium on , vol., no., pp.269,274, 2000.

MOREIRA, M.; OLIVEIRA, B.; PONTES, J.; MORAES, F.; CALAZANS, N., "Adapting a C-element design flow for low power," Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on , vol., no., pp.45,48, 11-14 Dec. 2011

MULLER, D.E. Asynchronous logics and application to information processing. In H. Aiken and W. F. Main, editors, Proc. Symp. on Application of Switching Theory in Space Technology, pages 289–297. Stanford University Press, 1963.

MYERS, C. Asynchronous Circuit Design. 1ª Ed. New York: John Wiley & Sons, Inc, 2001.

MYERS, C. J.; BEEREL, P. A. Technology Mapping of Timed Circuits. In: PROCEEDINGS SECOND WORKING CONFERENCE ON ASYNCHRONOUS DESIGN METHODOLOGIES, [S.l.], 1995, p.138-147.

NIELSEN, L.S. Low-power Asynchronous VLSI Design. PhD Thesis. Department of Information Technology, Technical University of Denmark, 1997. IT-TR:1997-12.

NIELSEN, L.S.; NIESSEN, C.; SPARSØ; J.; VAN BERKEL, C.H. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. IEEE Transactions on VLSI Systems, 2(4):391–397, 1994.

NIELSEN, L.S.; SPARSØ, J. An 85 μW asynchronous filter-bank for a digital hearing aid. In Proc. IEEE International Solid State Circuits Conference, pages 108–109, 1998.

NOWICK, S.M.; SINGH, M. High-Performance Asynchronous Pipelines: An Overview. In: IEEE DESIGN & TEST OF COMPUTERS, [S.l.], 2011. p. 8-22.

PAVER, N.C.; DAY, P.; FARNSWORTH, C.; JACKSON, D.L.; LIEN, W.A.; LIU, J. A low-power, low-noise configurable self-timed DSP. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 32–42, 1998.

SPARSØ, J. Asynchronous circuit design - A tutorial. Boston: Kluwer Academic Publishers, 2001.

SPARSØ, J.; STAUNSTRUP, J. Delay-insensitive multi-ring structures. INTEGRATION, the VLSI Journal, 15(3):313–340, October 1993.

SPARSØ, J.; STEVE, F. Principles of Asynchronous Circuit Design – A Systems Perspective. Springer, 2001.

STEVENS, K.; GINOSAR, R.; ROTEM, S. Relative Timing. In IEEE Transaction On VLSI Systems. vol. 11. no. 1. 2003. p 129-140.

STEVENS, Ken at al. The Future of Formal Methods and GALS Design. In Electronic Notes in Theoretical Computer Science. vol. 245. 2009. p 115-134.

STEVENS, Ken at all. An Asynchronous Instruction Length Decoder. In IEEE Journal of Solid-State Circuits. vol. 36. no. 2. 2011. p 217-228.

UNGER, S. Asynchronous Sequential Switching Circuits. 1ª Ed. New York: John Wiley & Sons, Inc, 1969.

VAN BERKEL, C.H.; BURGESS, R.; KESSELS, J.; PEETERS, A.; ROCKEN, M.; SCHALIJ, F.; VAN DE WIEL, R. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In 2nd Working Conference on Asynchronous Design Methodologies, London, May 30-31, 1995, pages 72–79, 1995.

VAN BERKEL, C.H.; BURGESS, R.; KESSELS, J.; PEETERS, A.; RONCKEN, M.; SCHALIJ, F. Asynchronous circuits for low power: a DCC error corrector. IEEE Design & Test, 11(2):22–32, 1994.

WEST, N. H. E.; HARRIS, D. M. CMOS VLSI Design: A Circuits and Systems Perspective. 4ª Ed. Boston: Addison-Wesley, 2011.

WESTE, N., HARRIS, D. 2010. CMOS VLSI Design: A Circuits and Systems Perspective (4th ed.). Addison-Wesley Publishing Company, , USA.

WESTE, N.; HARRIS, D. CMOS VLSI Design: A Circuits and Systems Perspective (4th ed.). Addison-Wesley Publishing Company, USA., 2010.

WILLIAMS, T.E.; HOROWITZ, M.A. A zero-overhead self-timed 160ns. 54 bit CMOS divider. IEEE Journal of Solid State Circuits, 26(11):1651–1661, 1991.

WILLIAMS, T.E.; PATKAR, N.; SHEN, G. SPARC64: A 64-b 64-active instruction out-of-order-execution MCM processor. IEEE Journal of Solid-State Circuits, 30(11):1215–1226, November 1995.

YUN, K. Synthesis of asynchronous controllers for heterogeneous systems. Ph.D. dissertation, Stanford University, Stanford, CA, 1994.