School of Electrical, Electronic & Computer Engineering

Formal Design and Synthesis of GALS Architectures

Sohini Dasgupta

Technical Report Series

NCL-EECE-MSD-TR-2008-131

March 2008

Contact:

Sohini.Dasgupta@ncl.ac.uk

Supported by EPSRC grant GR/S12036

NCL-EECE-MSD-TR-2008-131

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

`http://async.org.uk/`

Formal Design and Synthesis of GALS Architectures

by Sohini Dasgupta

School of Electrical, Electronic & Computer Engineering

Newcastle University

PhD Thesis

January 2008

To My Parents

Contents

Bibliography **184**

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AFSM	Asynchronous Finite State Machine
AMS	Austrian Micro Systems
CSC	Complete State Coding
DAG	Directed Acyclic Graph
DSM	Deep Sub Micron
DVFS	Dynamic Volatge and Frequency Scaling
FF	Flip Flop
FIFO	First-In-First-Out
GALS	Globally Asynchronous and Locally Synchronous
IC	Integrated Circuits
IP	Intellectual Property
IP	Input Port
ITRS	International Technology Roadmap for Semiconductors
KPN	Kahn Process Netweork
LSTS	Labelled Synchronous Transition System
LTTA	Loosely Time-Triggered Architecture

ME	Mutual exclusion Element
MUTEX	Mutual Exclusion
NRZ	Non Return to Zero
OP	Output Port
PLL	Phase Locked Loop
PN	Petri Net
RG	Reachability Graph
RTZ	Return to Zero
SoC	System on Chip
STG	Signal Transition Graph
STS	State Transition System
USC	Unique State Coding
WE	Weak Endochrony

List of Publications

Journal Publications

1. S. Dasgupta, A.Yakovlev. Comparative Analysis of GALS Architectures, In IET Computer and Digital Techniques, March 2007, Vol. 2(2), pp 59-69.

2. S. Dasgupta, D. Potop-Butucaru, B. Caillaud, A. Yakovlev. Moving from Weakly Endochronous Systems to Delay Insensitive Circuits, In Electronic Notes in Theoretical Computer Science, January 2006, Vol. 146 (2), pp 81-103.

Conference publications

1. S. Dasgupta, A. Yakovlev. "Modeling and Performance Analysis of GALS architectures". In proceedings of *IEEE International Symposium on System-on-Chip (SOC),* Finland, November 2006 pp 1-4.

2. S. Dasgupta, D. Potop-Butucaru, B. Caillaud, A. Yakovlev. "Moving from Weakly Endochronous Systems to Delay Insensitive Circuits". In proceedings of *International Workshop on Formal Methods of GALS (FMGALS)*, Italy, July 2005.

3. S. Dasgupta, A. Yakovlev. "Modeling and Verification of GALS Ring Architectures". In proceedings of *IEEE International conference of Design, Automation and Test in Europe (DATE)*, Germany, March 2005, Vol. 1, pp 568-569.

4. S.Dasgupta, A. Yakovlev. "GALS implementation of CAN transfer protocol Engine". In proceedings of *Postgraduate Conference*, Newcastle, January 2006.

5. S.Dasgupta, A. Yakovlev. "Globally Asynchronous and Locally Synchronous Ring Architectures-Performance Issues". In proceedings of *Postgraduate Conference*, Newcastle, January 2005.

Technical Reports

1. S. Dasgupta, A.Yakovlev, V.Khomenko, "Desynchronisation Technique using Petri Nets". Technical Report Series NCL-EECE-MSD-TR-2006-114, Microelectronics System Design Group, School of EECE, University of Newcastle, April 2006.

2. S. Dasgupta, A. Yakovlev. "Modeling and Performance Analysis of GALS Architecture". Technical Report Series NCL-EECE-MSD-TR-2007-124, Microelectronics System Design Group, School of EECE, University of Newcastle, November 2007.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Alex Yakovlev, for introducing me to the area of Asynchronous systems. He was patient and gave me a lot of freedom when I was stumbling around looking for a research problem in the early years and kept my spirits high with his motivation whenever anything went wrong. This thesis would not have been possible without his valuable support and continuous guidance throughout this research. I would also like to thank Dr. Dumiru Potop Butucaru, at INRIA France, and Dr. Victor Khomenko for their invaluable technical inputs.

I would also like to thank Dr. Danil Sokolov, Dr. Deepali Koppad and Crescenzo D'Alessandro for their numerous technical discussions.

I would also like to express my gratitude to my family for their whole hearted support during the course of this research. I am thankful to my husband, Praneet, for his support during my research.

Finally, I would like to acknowledge that this work was partly sponsored by the School of EECE, Newcastle University and UK Engineering and Physical Sciences Research Council (grant number $EP/C007298/1$).

Abstract

A Globally Asynchronous and Locally Synchronous (GALS) system can be obtained by: (1) integrating independently clocked domains via an asynchronous communication link or, (2) desynchronising a synchronous system into a number of synchronous compartments whose interface is seamlessly refined to handle asynchronous communication.

In the area of system integration, a number of schemes have been proposed to handle the synchronisation problem. There have been no comparative performance analysis to aid the designer to choose one scheme over another. Therefore, we classify these schemes into three generic categories so that they can be brought to a common platform for comparison and show how they can be applied to an existing partitioned synchronous architecture to obtain a reliable, low latency and efficient clock control architecture. We present circuit solutions and comparative analysis results for the generic classes in terms of circuit implementation, performance and relative power consumption.

The various system desynchronisation methodologies proposed are targeted for single clock synchronous systems where all components operate on the same

clock or known clock ratios. Weak Endochrony (WE), to the best of our knowledge, is the only methodology that can handle unknown clock ratios. Therefore, in this area, we specifically address: (a) the problem of synthesising the asynchronous wrappers needed for GALS implementation based on the WE model, which defines correct desynchronisation conditions for a synchronous system and, (b) the problem of complexity posed by (a), by proposing a new methodology to desynchronise modular synchronous specifications, with unrelated clock ratios, into independent synchronous compartments for their realisation into GALS architectures and obtain simple wrappers that are efficiently synthesisable using existing synthesis tools.

The wrapper synthesis phase in (a) involves the interface refinement and translation of the WE models into Petri net models, while preserving the WE properties, to obtain latency insensitive circuits. In the system desynchronisation phase (b) we present a formal framework that besides bridging the semantic gap between the synchronous and asynchronous models also addresses the issues posed by previous desynchronisation methods, while allowing us to reason and present conditions for the correctness of system partitioning. The results of (a) and (b) have been demonstrated by applying the respective algorithms on simple processor models.

Chapter 1

Introduction

The complexity of digital systems grows enormously, as can be seen from the technology roadmap [1, 2] in Table 1.1. Globally Asynchronous and Locally Synchronous architectures (GALS) are designed to facilitate the integration of multi-million transistor on a single chip. Clocking circuits are becoming increasingly hard to design with larger chip sizes, higher clock rates and larger wire delays. To reduce the design time of a system, it is becoming essential to reuse verified and tested intellectual property (*IP*) blocks. The integration of various *IP* cores on complex systems on chip requires a multitude of clock frequencies on a single die. Such integrations are enabled by modern deep sub-micron fabrication technologies in the form of chips with more than a billion transistors [3]. GALS architectures aid such integration by allowing the synchronous blocks to operate independently with other synchronous blocks through asynchronous communication channels. Therefore, instead of using a fixed period clock as used in globally

Property	Year 1999	Year 2001	Year 2005	Year 2011
CMOS process [μm]	0.18	0.15	0.1	0.05
Transistor on chip [$Mtrans/cm2$]	7	14	41	247
On-chip clock[GHz]	1.25	1.77	3.5	10
Off-chip clock[GHz]	0.48	0.722	1.035	1.54
Power dissipation (handheld systems) [W]	1.4	1.7	2.2	2.4
Vdd[V]	1.5	1.2	0.9	0.5

Table 1.1: Technology roadmap from 1999 to 2011

synchronous systems, GALS systems have a locally generated clock whose period is specific for the local (synchronous) block. The application of locally generated clocking schemes enables integration of *IP* cores with unrelated clock ratios. The components of a heterogeneous system can be clocked using different clock generation schemes, to obtain a GALS system.

This thesis amalgamates two areas of research that addresses two prevalent issues in the design of GALS architectures. GALS systems have been extensively studied and many approaches have been proposed for the design of such systems. There is a plethora of researchers who have proposed various design methodologies, presented in Chapter 3, and analysed their particular designs for different system parameters. Hence, a classification of the proposed systems becomes important. Once classified, a generic model of the scheme can be built for each class and a comparative study between them carried out. This first part of the research classifies the model and presents novel comparative analysis results based on performance and power for each generic model.

Since GALS design is seen as a replacement for conventional synchronous design, it is becoming increasingly important to devise ways of translating existing synchronous systems into a GALS architecture. There are two ways to do this. A GALS architecture could be built from individually clocked blocks. These blocks are integrated using different communication primitives. FIFO is one such primitive and is extensively discussed in the subsequent chapters. On the other hand, a globally synchronous system can be *desynchronised* into small independent islands that are made to interact with each other using similar communication primitives, as mentioned above.

The desynchronisation problem can be defined as the property by which a synchronous system can be deployed over asynchronous architecture in a correct by construction mechanism. The main objective for desynchronisation is the ability to use existing synchronous design tools and practises for the specification and optimisation of a system, while still obtaining an efficient final distributed architecture. All the approaches to achieve the above can be classed under two categories, which combine the theoretical properties of synchronous designs with efficient implementation where the synchronous constraints are relaxed.

1. *Desynchronisation:* desynchronisation was motivated by the problem of achieving a correct by construction modular deployment of embedded software on distributed architecture [4, 5, 6]. In such an implementation large system processes are designed synchronously and are made to communicate via asynchronous communication lines. This allows the different parts of the system to operate at different speeds.

2. *Latency Insensitive Design:* Latency insensitive design was motivated by the problem of long interconnect delays between the different components of the system. In deep submicron (DSM) technologies this is more profound because the long paths between the components introduce delays that force the overall clock to slow down in order to maintain synchronous behaviour. By introducing automatic pipelining and insertion of *relay stations* [7], latency insensitive design allows the implementation to avoid slowing down the clock. This aids the recovery of a part of the throughput that could have been achieved without the interconnection delays.

1.1 Motivation and contribution

The motivation and the main contribution of this research is presented below:

GALS design and comparative analysis

The GALS literature lacks the comparative study between the different GALS designs which enable design exploration of these applications in terms of which communication interfaces or architectures are more suitable for deploying. Since power is an important metric for SoC applications, this work also estimates power overheads, in addition to the performance overheads, introduced by various on-chip communication and the clock generators used in each scheme. This thesis gives a comprehensive overview of the GALS design methodology. Three generic models of GALS clocking schemes are presented. These form the basis of most

design approaches proposed so far. The contribution of this work lies in the introduction of a system level methodology which is amenable to analysing the power and performance characteristics of the different clocking schemes. The clocking schemes presented here allow the different synchronous blocks to be connected asynchronously without the need to interface them with additional synchronising elements.

Synthesis of desynchronised components

We consider the problem of synthesising the asynchronous wrappers and glue logic needed for the correct GALS implementation of a modular synchronous system. Our approach is based on the weakly endochronous synchronous model, which defines high-level, implementation-independent conditions guaranteeing correct desynchronisation at the level of the abstract synchronous model. It is shown that *Weakly Endochronous (WE) System* provides solutions to many issues posed by previous *correct-by-construction* methodologies. The previous methodologies were applied to independent synchronous blocks which had related clock ratios. WE systems can handle unrelated clock ratios, which is the most important characteristic that leads to the choice of WE systems as the basis of our desynchronisation methodology. This methodology is composable, a factor that deterred the formation of GALS systems consisting of more than two synchronous blocks in some of the forerunners. The main contribution of the work is to develop a methodology to translate the WE models into an implementable one by:

- adding extra signalisation to cope with asynchronous communication which

supports the notion of *data driven clocking scheme,* introduced in chapter 4.

- attaching the FIFO information to each signal explicitly, which was earlier implicit in the model.

The above two aid the implementation of WE system into delay insensitive circuits.

New desynchronisation methodology

The above work led to the identification of some issues while carrying out the synthesis process on practical applications. This led to the introduction to our own desynchronisation methodology. Our approach to this synthesis problem consists of partitioning the synchronous blocks and assigning them to *Localities.* Each of these localities emulate synchronous behaviour and hence can be treated as independently clocked domains. The interfaces of these blocks are modified to handle asynchronous communication which results in a correct GALS implementation. The blocks are made to communicate with each other using asynchronous FIFOs. The localities are modeled using the Petri net language, which is highly expressive in modeling concurrency. This is because concurrency is one of the main properties of a distributed architecture. Our contribution has been the formalisms presented on the notions of correctness for the partitioning of the globally synchronous systems into localities and preservation of semantics in the GALS deployment of the synchronous system. The approach is easy and efficient and can be applied to general systems.

1.2 Thesis Outline

The thesis is organised as follows:

Chapter 2 presents the idea of GALS systems. It briefly describes the different components that comprise the GALS architecture. Then it goes on to introduce the modeling languages used in different parts of the research. These include Petri nets and State Transition systems. Both these languages play an integral part in specifying our systems at various levels of abstraction.

Chapter 3 highlights and reviews the existing methodologies related to the work presented in this thesis. The main categories of methodology are system integration and system desynchronisation techniques. This chapter also introduces the concept of Endochronous and Weakly Endochronous systems which form the basis of our synthesis and desynchronisation methodologies.

Chapter 4 categorises the different clocking schemes proposed to date, into three main classes. A generic model is then built for the three classes and each of them synthesised in the Cadence framework. Important metrics are identified for these circuits to compare them on the basis of performance and power. Results presented include both model level and system level analysis taking these metrics into account.

Chapter 5 introduces the idea of weak endochrony and the criteria that equip the models to be deployed in an asynchronous environment from a synchronous one. It presents an algorithm to translate this model to an implementable level, while preserving the semantics of the initial WE system and the correctness of the

GALS implementation.

Chapter 6 proposes a new desynchronisation methodology which addresses the problems encountered in the previous method, presented in Chapter 5. This method introduces the idea of Localities and shows how it can be applied to synchronous systems to obtain a distributed architecture. It presents an algorithm for locality allocation and gives correctness notions for such partitions.

Chapter 7 summarises the results achieved during the course of the research and also presents areas of potential future work.

Chapter 2

Background

2.1 Introduction

The GALS design methodology has been developed to address several issues that have been posed by synchronous and asynchronous design methodologies. The goal is to combine the advantages of both synchronous and asynchronous design styles and avoid their shortcomings. This chapter presents a brief description of the three design styles, namely, synchronous, asynchronous and GALS. It also presents the underlying models for the specification of these design styles. These models include Petri nets and State Transition Systems.

2.1.1 Synchronous Design

The synchronous design style is the most established design style of today. The main characteristic of a synchronous system is the global clock that governs all

the activities in the system. All the events in the circuit are ordered by the clock signal. In a synchronous circuit this global clock samples all data signals of the circuit. The circuit operates correctly as long as all the signals have their data valid at the time of the clock event.

Since the advent of the digital design era, this scheme has been very successful. But as with all engineering solutions there are several disadvantages of this approach. Recent developments in IC manufacturing technologies, besides increasing the performance of the ICs by several magnitudes, has also aggravated several design problems with this design style. The main challenge arising from this design style is to incorporate the distribution of the global clock across a single die.

Since the timing of a synchronous system is of utmost importance, it is imperative to distribute the clock signal to all the clocked elements in the circuit at the same time. The modern IC technologies significantly reduce the circuit size and increase the clock rate. Therefore, the precision with which the clock is required to be distributed to the different elements increases. Hence, in modern design techniques a significant amount of time is spent in distributing the clock and achieving timing closure.

2.1.2 Asynchronous Design

Contrary to the synchronous design styles, asynchronous circuits operate without a clock signal. They consist of several blocks that communicate via handshake signals. These handshake signals request data from connected blocks and on the

arrival of the relevant data acknowledge its receipt. The handshake signals are generated locally within each block. Since they do not depend on a global clock signal, they are also known as "self-timed" circuits.

The self-timed circuits do not have any problems associated with the clock distribution, therefore, eliminating all synchronisation problems. Asynchronous designs exhibit average case performance as opposed to worst case performance. There is an improvement in the critical timing of a circuit when comparing average case scenarios to worst case scenarios. In a synchronous system, the clock period is chosen in such a way that it accommodates the slowest synchronous block and therefore it exhibits worst case performance. On the other hand asynchronous designs can sense the completion of a computation with the help of completion detection signals. Moreover, a synchronous circuit operates, even if it has no function to perform and hence consumes dynamic power in idle states. A self-timed circuit in such a situation is inactive and would simply wait. Self-timed designs are based on reliable data transfer independent of absolute timing information. Therefore, different modules designed by the same approach can be easily integrated with each other.

The self-timed approach is fundamentally different from synchronous designs. Most engineers are unfamiliar with this design style and hence reluctant to adopt such methodologies. Moreover, the lack of industrial EDA tool support has hindered the viability of asynchronous circuits.

2.1.2.1 Classes of Asynchronous Circuits

Asynchronous circuits can be divided into different delay classes and operation modes which are defined below: [8, 9].

Delay models

1. **Bounded delay:** The delays in both gates and wires are bounded. Bounded delay means that the delay is known or at least it is limited. This delay model is also used for synchronous circuits.

2. **Speed-independent:** This model assumes that the gate delays are unbounded and that the delays in wires are negligible.

3. **Delay-insensitive:** Delays in both wires and gates are unbounded, hence a signal can be delayed without circuit failure.

4. **Quasi-delay-insensitive:** Unbounded gate and wire delays, where some wire forks are isochronic, i.e., the delay to the wire ends are the same for all wires in the fork or the delay difference is negligible.

Operation modes

1. **Fundamental mode:** No input signals from the environment are allowed to change until the circuit and feedback signals have stabilised.

2. **Input/Output mode:** The environment is allowed to change the input signals as soon as the circuit has produced the corresponding output signals.

Hence, the feedback signals do not have to be stable.

3. **Single-Input Change:** Only one input signal can change at a time.

4. **Multiple-Input Change:** Multiple-input signals are allowed to change at the same time.

2.1.2.2 Handshake Protocols

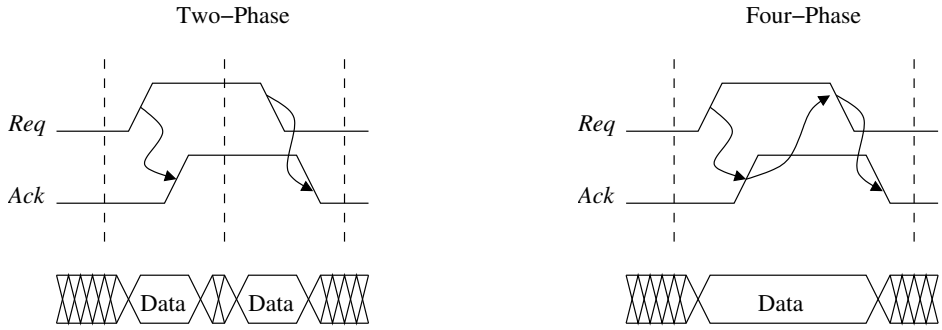
In order to transfer data between blocks it is necessary to use a protocol, so that both the transmitter and the receiver can agree when data is valid. For the synchronous domain, the clock is used to decide when the data is valid.

Two-phase and four-phase handshake protocol

Two-phase and four-phase handshake protocols are briefly described. The asynchronous processor, Amulet 1 uses mostly the two-phase handshake protocol, but also includes circuits using the four-phase handshake protocol [10]. Figure 2.1 shows example of both two-phase and four-phase handshake protocols.

In the case of the two-phase handshake protocol, the changes in a signal are important and while for the four-phase handshake protocol, the signal level is important. For two-phase handshake protocol, one event occurs on both request and acknowledgement, and for four-phase handshake protocol, two events occur on the request and acknowledge signals. In Figure 2.1, two transfers are shown for the two-phase protocol, and one transfer for the four-phase protocol.

A transfer is always started with an event on the request signal and is al-



(a) 2-phase handshake protocol (b) 4-phase handshake protocol

Figure 2.1: Two and Four- phase handshake protocols

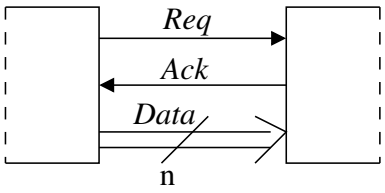


Figure 2.2: Bundled data and dual rail channels

ways followed by an event on the acknowledge signal. The two-phase handshake protocol is also known as transition or non-return-to-zero (NRZ) signalling and four-phase handshake protocol is also known as level or return-to-zero (RTZ) signalling. Both the two-phase and four-phase protocols can be bundled with the data and is then called bundled data protocol as shown in Figure 2.2.

2.1.3 GALS Design

The globally synchronous paradigm suffers from issues such as clock skew and power consumption caused by the global clock distribution. In globally synchronous environments, synchronisers or asynchronous FIFOs are used between

different clock domains to reduce the probability of failure due to metastability. This leads to a widespread control overhead, which in turn increases the size of the die. For many years, advocates of the self timed circuits have predicted the demise of synchronous design.

There has been a gradual transition from the synchronous world to the mixed synchronous-asynchronous world. The traditional methodology for clock generation is the synchronisation of a slow off-chip clock with a faster on-chip clock using a phase-locked loop (PLL). The output of the PLL is distributed across the chip by a balanced $H - tree$ which balances the propagation delay between the roots and the leaves. However, a time varying phase displacement called jitter is introduced by the PLL itself and capacitively and inductively coupled into the clock tree. Process variations in the interconnects and its repeaters introduce phase differences called skew between the leaves of the tree. These variations are detrimental because they limit the portion of the clock cycle which is available for computation. These effects are further exacerbated by shrinking die size and increased clock frequencies.

Seitz [11] defined synchronous designs as those in which the sequence and time are connected by a system wide clock and self timed designs as those in which sequence and time are connected inside elements whose terminal behaviour only satisfies a sequence domain representation insensitive to elements and wiring delays. Therefore, Seitz suggested that to avoid the complexity of distributing a single global clock across the entire chip area and the varying power requirements for different blocks, the synchronous blocks can employ independent in-

ternal clocks. The frequency can be scaled up or down depending on the performance requirements of the system. The asynchronous channels over which the synchronous blocks communicate may use delay insensitive styles such as dual rail for single bits or 1-of-4 encoding for two bits [12]. They may also use bundled data convention in which a delay matched control bit is transmitted in parallel with the data bus.

Chapiro's PhD thesis [13] laid the foundation for subsequent work in standard GALS system design. Although the circuitry described in this work cannot be successfully applied to modern high speed digital systems, it still forms the fundamental basis of GALS systems. Hemani et. al. [14] investigated the potential of GALS architectures to save the clock power as compared to synchronous designs. In their work they showed that in a system consisting of N synchronous blocks, the clock power is reduced by a factor of the square root of N. They also modeled the power consumption in GALS overhead logic and showed that GALS still offers a net power savings over the fully synchronous architectures.

In addition to local clock generator circuitry, GALS systems require synchronisation between clock domains for reliable data transfer. This synchronisation is necessary to avoid or remove metastability. Metastability occurs when a flip flop fails to arrive at a known state in a specific amount of time. In such a situation it is not possible to predict the element's output voltage and the time when the output would settle to a correct voltage level. During this state the flip flop's output is at some intermediate volatge level.

A flip-flop may enter a metastable state if the input is not stable when it is

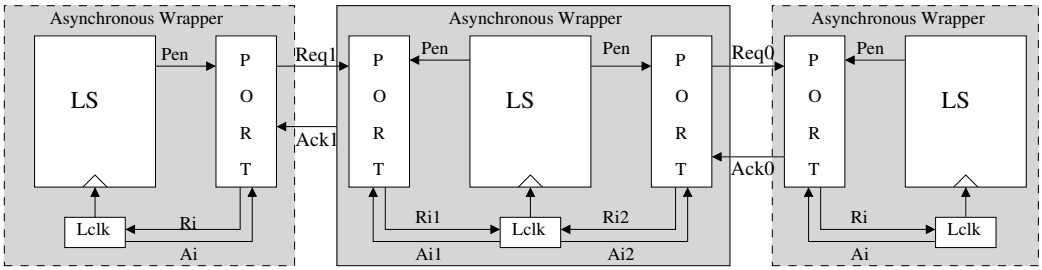


Figure 2.3: A single port GALS System

sampled, i.e., the setup and hold times for the flip-flop are violated. Metastability may occur in the receiving data register of a GALS system if the input is unstable during the small window around the active edge of the clock. A register typically contains several flip-flops, and problems may arise if one or several of the flip-flops in a register enter a metastable state. To avoid metastability, the synchronisation strategy stops the clock when the data transfer takes place.

2.1.3.1 GALS architecture

A typical GALS architecture is depicted in Figure 2.3. The GALS wrapper consists of the following:

- Local Clock Generators
- Handshake Unit

2.1.4 Local clock Generator

The asynchronous wrapper also includes a local clock generator. This generator controls the granting of the clock pulse to avoid metastability from affecting the overal working of the system by using Mutual Exclusion (MUTEX) elements. They basically consist of ring oscillators and mutual exclusion elements. The locally synchronous modules can utilise the local clock signals of frequencies suitable to their needs. Whenever an asynchronous data transfer takes place, the associated port controller sends a request to stretch the clock, R_i. In response to the request the clock generation circuit stretches or pauses the clock and acknowledges the stretch by setting A_i. The clock generation circuits can either receive stretch signals from just one port controller, or they could receive the request signals from more than one port controller contained within the wrapper.

Mutual Exclusion Element

The clock generator circuit shown in Figure 2.4 consists of a ring oscillator and a MUTEX. The ring oscillator generates a continuous clock signal, whose frequency can be varied by varying the delay in the feedback loop, depending on the system requirements. The MUTEX is used to arbitrate between the clock grant request R_i and data transfer request $Rclk$.

The rising edge that appears first at the input of the MUTEX will go on to pass through the MUTEX to its respective output, and the rising edge of the other signal will only pass through after the first granted signal goes low. If both the signals appear simultaneously at the input of the MUTEX, then the signals run into a metastable state inside the MUTEX. At this stage, the MUTEX randomly chooses

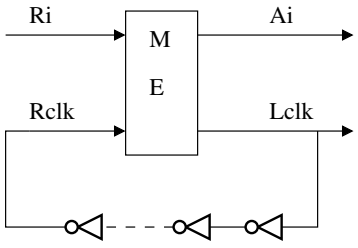


Figure 2.4: Clock generator circuit

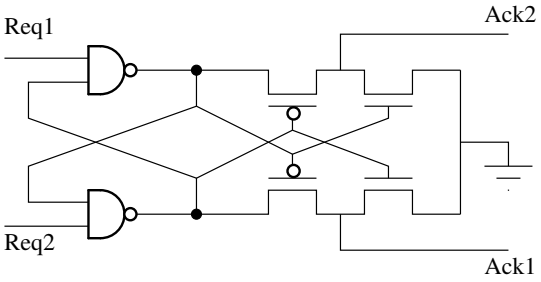


Figure 2.5: Mutual Exclusion Element

between the requests and grants one of them. Figure 2.5 shows the structure of a MUTEX.

An asynchronous wrapper can have more than one port. The port-controllers of each of these ports have the provision to request for clock stretch whenever required. In order to facilitate this, the wrapper requires to have a clock generator which is equipped to accept multiple clock stretch request from all the port controllers in the wrapper. Such a circuit is depicted in Figure 2.6.

The multi-request circuit safely arbitrates between the incoming request signals, $R_i, .., R_{in}$ and the rising edge of $Rclk$. It can be seen that even if one clock stretch request is present the $Rclk$ is not granted due to the presence of the AND

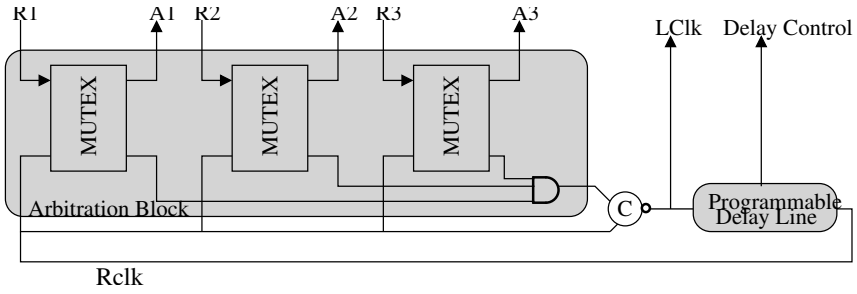


Figure 2.6: ME with multiple port controllers

gate.

2.1.5 Handshake Unit

The handshake unit mainly consists of the GALS ports. A GALS port comprises a port controller defined by an Asynchronous Finite State Machines (AFSM) and a flip flop for signalling the completion of a transfer. These GALS ports from different modules communicate with each other using a simple handshake protocol. The sending port "requests" a data transfer and the receiving port "acknowledges" as soon as it is ready. The GALS port can pause the local clock to ensure synchronisation.

GALS Port Controllers There are two families of port controllers:

- Demand: A *demand-type* (D-type) port also ensures data integrity but incorporates a feature similar to clock gating [15]. This type of port stops the local clock as soon as it is enabled ("sleep while waiting"). All ports work with the push principle where a sender always initiates a data transfer and

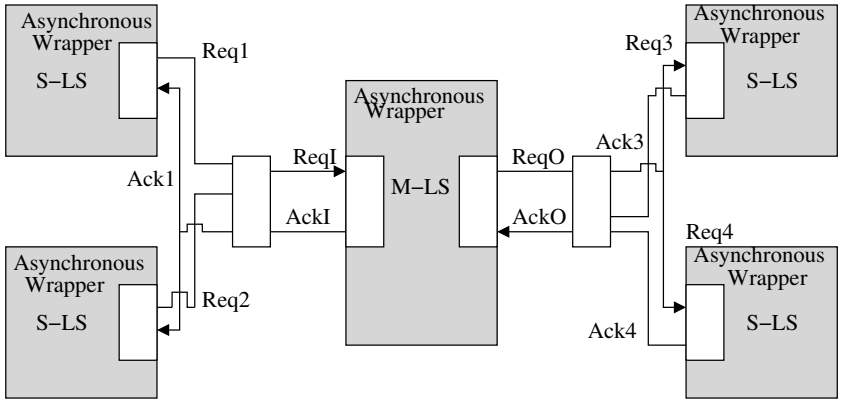


Figure 2.7: A multi-port GALS system

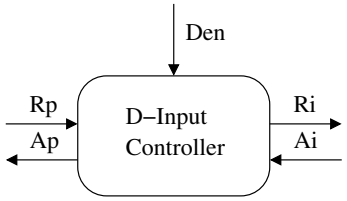


Figure 2.8: D-port Controller

the receiver answers with an acknowledge. When a D-type port is enabled, notified by a switching event on Den, it immediately issues a clock stretch request R_i, which gets acknowleged by A_i, and does not release it until the data transfer has taken place, denoted by the external hanshake cycle on R_p/A_p. A D-type port is used whenever data transfer is required immediately - when no further computations can be performed by the locally synchronous island without the data being transferred. Figure 2.8 shows a D-type port controller.

- Poll: A *poll-type* (P-type) port asks for clock stretching only to prevent metastability and ensures data correctness ("proceed while waiting"). These port controllers influence the clock as infrequently as possible [15]. A P-type port is used whenever a data transfer is possible but is not necessarily required to happen immediately. The locally synchronous module continues to work normally, while the P-type port controls the data transfer. Figure 2.9 shows a P-type port controller. After the activation of the port by signal P_{en} , the port polls the handshake lines connected to it. In contrast to the D-ports, there is no predetermined cycle of the local clock during which the data transfer accurs. Therefore an extra signal T_i is generated to denote that the transfer has taken place. In order to feed this signal to the locally synchronous domain a separate finite state machine is used to synchronise this signal to the local clock. Since during the event on $T_i$the local clock is kept low, a synchronised signal T_s is generated that is high during the clock pulse following a transition on T_i. In this way, T_s can be safely sampled with the local clock's rising edge.

All data transfers on a particular port in a GALS system are managed by port controllers. Figure 2.10 shows the above port controllers between two locally synchronous islands. The enable signal triggered by the locally synchronous module uses 2-phase protocol, while all the signal links between the port controller and the clock generator as well as between the two communicating port controllers employ 4-phase handshaking. A GALS module with multiple port controller is shown in Figure 2.7.

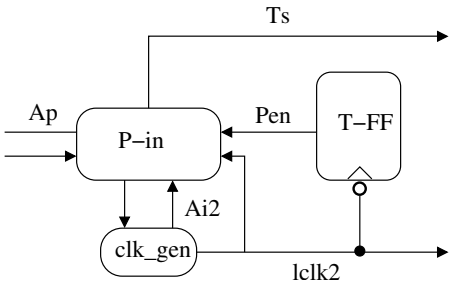


Figure 2.9: P-port Controller

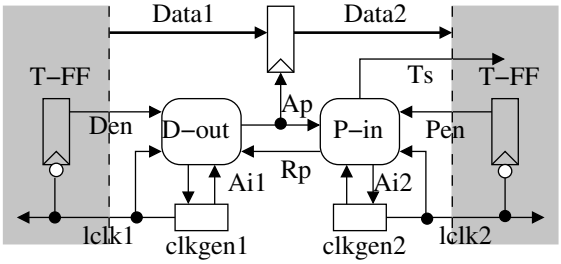


Figure 2.10: D-type and P-Type port controllers

2.2 GALS Behavioural Modeling Schemes

This section presents the modeling schemes used to specify the systems used in this thesis.

2.2.1 Synchronous Transition Systems

The components and systems interact with each other and with the environment through *variables*. The domain of variable v is denoted by D_v. Given V a set of variables, a label over V is a partial valuation of its variables. The set of all the labels over V is :

$$L_v \quad = \quad \prod_{v \in V} (D_v \cup \{\bot\})$$

Every system, component and communication lines are modeled using *Generalised Concurrent Transition Systems* (GCTS). An $GCTS$ is a tuple $\Sigma = (S, s_0, V, T)$, where, S is the set of states, $s_0 \in S$ is the initial state, V is a set of communication variables, $T \subseteq S \times L_v \times S$ is a set of transitions satisfying the following axioms:

Axiom 1. *Void transition:* $\forall s \in S : (s, \bot_V, s) \in T$.

Axiom 2. *Prefix closure: If* $(s, l, s') \in T$ *and* $l' \leq l$*, then there exists* $s'' \in S$*, such that* $(s, l', s''), (s'', l \backslash l', s') \in T$.

A *trace* ϕ over the set of variables V is a finite sequence of labels over V. The *support* of a label $l \in L_v$ is $supp(l) = \{v \in V \mid l(v) \neq \bot\}$. The interaction between a synchronous component and its environment is a sequence of *reactions*, which are mappings $r \in Reactions = V \rightarrow D'$. The *signature* of a reaction r is $sig(r) = \{v \mid r(v) \neq *\}$. $Reactions(U)$ denotes the set of reactions of signature $U \subseteq V$. The *reachable state space* of a $GCTS$ Σ is a set

$$RSS(\Sigma) \quad = \quad \{s \in S \mid \exists \phi : s_0 \xrightarrow{\phi} s\}$$

To take into account directed communication, *directed variables* are used. These variables are denoted by $!c$ (to depict the emission on channel c) and $?c$

(to depict reception on channel c). The variable $!c$ and $?c$ have the same domain denoted by D_c. $C(V)$ denotes the set of channels associated with a set of variables V. The communication lines consist of directed FIFO channels, each channel depicted by a pair of directed variables. A value on channel c is emitted by assigning the variable $!c$, and a value is received on the channel c by reading variable $?c$. The variables $!c$ and $?c$ must have the same domain D_c. To represent synchronous and GALS systems, the clock signals are represented by special clock variables that carry no data and are denoted by τ.

Below are some definitions of some relational properties that are followed by the traces obtained from each of the system components.

Definition 2.1. *Asynchronous Equivalence Relation ($\sim$):* For all $v \in Directed(V)$, if $\phi_1 \leq \phi_2$ and $\phi_2 \leq \phi_1$, then ϕ_1 and ϕ_2 are *asynchronous equivalent.*

The definition states that if for every communication channel c of two traces ϕ_1 and ϕ_2, the order of activities is the same, then the two traces are asynchronously equivalent. For example, if ϕ_1 $:!a!b?a$ and ϕ_2 $:!a?a!b$, then $\phi_1$and $\phi_2$are *asynchronously equivalent*. This is because the order of $!a$ and $?a$ is the same.

Definition 2.2. *Prefix Relation* ($\preceq$)*:* For all $v \in Directed(V)$, if $\phi_2 = \phi_1\phi_3$, then ϕ_1 is a prefix of ϕ_2.

If there exists a trace ϕ_3 such that a sequential execution of $\phi_1$and ϕ_3, denoted by $(\phi_1; \phi_3)$, is asynchronously equivalent to ϕ_2, then ϕ_1 is a prefix of ϕ_2. Therefore, a prefix ϕ_1 of a trace ϕ_2, maintaining the condition that the order

of ϕ_1 is equivalent to ϕ_2, can be enhanced by ϕ_3. For example, if $\phi_1 = !a?a!b$, $\phi_2 = !a!b?b?a$ and $\phi_3 = ?b$, then ϕ_1 is a prefix of ϕ_2.

Definition 2.3. *Asynchronously Non-contradictory Relation* ($\bowtie$): For all $v \in Directed(V)$, if $\phi_1|_{\{v\}} \preceq \phi_2|_{\{v\}}$ or $\phi_2|_{\{v\}} \preceq \phi_1|_{\{v\}}$, then ϕ_1 and ϕ_2 are *asynchronously non-contradictory*.

The above definition states that if traces ϕ_3 and ϕ_4 exist such that $\phi_1;\phi_3 \sim \phi_2;\phi_4$ and if it is possible to complete ϕ_1 and ϕ_2 to a trace ϕ, such that $\phi \sim \phi_1;\phi_3 \sim \phi_2;\phi_4$, then ϕ_1 is asynchronously non-contradictory to ϕ_2. For example, if $\phi = !a?a!b?b$, $\phi_1 = !a?a!b$, $\phi_2 = !a!b?b$, $\phi_3 = ?b$ and $\phi_4 = ?a$, then it can be observed that $!a?a!b?b \sim !a?a!b\,;\, ?b \sim !a!b?b\,;\, ?a$, therefore making ϕ_1 non-contradictory to ϕ_2.

2.2.2 Petri Nets

Petri nets are widely used to model concurrent systems because they have simple and intuitive semantics. A Petri net (PN) is a model used to represent systems with concurrency. It is a quadruple $PN = \{P, T, F, \mu_0\}$, where P is a set of *places*, T is a set of *transitions*, F is an *arc* denoting the flow relation $F \subseteq \{(P \times T) \cup (T \times P)\}$ and μ_0 is the *initial marking*. A labelled PN is a PN with a labelling function $L : T \rightarrow A$ associating each transition of the net with a name. A labelled Petri net can have a combination of implicit places, where the input and output transitions are named using symbols from the alphabets, connected by arcs and transitions which are labelled with signal transitions $(a+, a-)$ or events

(a_req, a_ack).

There exists an arc from $x \in P \cup T$ to $y \in P \cup T$ iff $(x, y) \in F$. The *preset* of a node $x \in P \cup T$ is defined as $\bullet x = \{y \mid (y, x) \in F\}$ and the *postset* as $x\bullet = \{y \mid (x, y) \in F\}$. A *marking* is a mapping $\mu : P \rightarrow N$ denoting the number of tokens in each place, $N = \{1, 0\}$ for $1-$*safe* PNs. The number of token assigned to a place p by a marking μ is written as $\mu(p)$. A transition $t \in T$ is enabled at a marking μ, denoted by $\mu[t >$, if for every $p \in \bullet t$, $\mu(p) > 0$. For a $1-$*safe* PN, the firing of the transition t modifies the marking by consuming one token from each of the predecessor places and producing one token to each of the successor places. A marking μ' is reachable from marking μ if there exists a *firing sequence* $\sigma = t_0 ... t_n$ that transforms μ to $\mu'$and is denoted by $\mu[\sigma > \mu'$. For any $a \in A$, by $\mu[a >$ (or, $\mu[a > \mu'$), it is meant that $\mu[t >$ (or, $\mu[t > \mu'$) for some t with $L(t) = a$. Let $S\mu_0$ be a set of reachable markings from the initial marking μ_0.

Given a Petri net N, the *pre-* and *post-multiset* of a transition t are respectively the multiset $pre_N(t)$ and the multiset $post_N(t)$, such that for all $p \in P$, $|p|_{pre_N(t)} = F(p, t)$ and $|p|_{post_N(t)} = F(t, p)$, where $|p|$ denotes the number of tokens present in the place p.

There are several structural and behavioural properties that are followed by Petri net models. Some of these properties, that are used in this research are elaborated below.

A Petri net marking is live if for each marking $\mu \in S\mu_0$ and for each transition t there exists a marking $\mu' \in S\mu$ that enables t. A marked Petri net is live if its

initial marking is live. A marked Petri net is $k-$bounded (or simply bounded) if there exists an integer k such that for each place p, for each reachable marking μ, $\mu(p) \leq k$. A marked net is safe if it is $1 - bounded$.

A transition t_1 disables another transition t_2 at a marking $\mu \in S\mu_0$ if both t_1 and t_2 are enabled at μ and t_2 is not enabled in any $\mu' \in S\mu$. A marked Petri net is *persistent* if no transition can ever be disabled at any reachable marking. Therefore, if for every place $p \in P$, $|\bullet p| = 1$ and $|p \bullet| = 1$, then we can say that the net is *persistent*. Two transitions t_1 and t_2 in a marked Petri net are concurrent if there exists a reachable marking $\mu \in S\mu_0$ where both t_1 and $t_2$are enabled and neither t_1 disables t_2 nor vice versa. Two transitions t_1 and t_2 are in conflict if there exists and reachable marking μ where both t_1 and t_2 are enabled and firing of t_1 disables t_2 and vice versa. A choice place is a place for which $|p \bullet| > 1$. A choice place is said to be unique choice if at most one of the successor transitions $|p \bullet|$ ever becomes enabled. A Petri net is free-choice if for any two transitions $t_1$and t_2 that share a predecessor place, both $t_1$and t_2 have only one predecessor. A Petri net is extended free choice if any two transitions that share one or more predecessor places have exactly the same set of predecessor places.

The places and transitions of the net can be interpreted in different ways depending on the requirements of the target specification. Examples of such interpretations are presented in Table 2.1.

Input Places	Transition	Output Places
Preconditions	Event	Postconditions
Input Data	Computation	Output Data
Input Signals	Signal Processor	Output Signals
Buffer	Processor	Buffer

Table 2.1: Some typical interpretations of transitions and places

2.2.3 Signal Transition Graphs

Signal transition graphs, or STGs, are a widely used representation of asynchronous digital circuits [16, 17]. STGs are Petri nets whose transitions are interpreted as signal transitions of a circuit. The transitions are interpreted as value changes on input, output or internal signals of the circuit [18]. Positive transitions (labelled with a " $+$ ") represent $0 \rightarrow 1$ changes and Negative transitions (labelled with a " $-$ ") represent $1 \rightarrow 0$ changes. A Signal Transition Graph N is a tuple $N = \{S, A, L\}$, where $S = \{N, \mu_0\}$ is a net system $N = \{P, T, F, \mu_0\}$, μ_0 is the initial marking of N, A is a set of signals and $L = T \rightarrow A \times \{+, -\}$ is a function that assigns a signal change to each transition of the net. Moreover, $A = X \cup Z$, where X and Z are disjoint sets of input and output signals, respectively.

It is graphically represented as a directed graph with transitions labelled with signal names and places denoted by circles. Usually places with only one input and output transition are omitted. A Signal transition graph is binary if its underlying net system is binary. Each marking is assigned a binary code which is a string of $0's$ and $1's$ denoting the value of the signal, at a particular marking.

This encoding of the STG states should be *consistent* i.e. no transition $t+(t-)$

should be enabled at a marking if the binary code of that marking gives the value of the signal t as $"1"("0")$. This can be intuitively seen as a signal whose value is already $"1"$ cannot rise further and similarly a signal with value $"0"$ cannot fall further.

An STG is said to satisfy *Complete State Coding* (*CSC*) property such that if there exists two markings with the same binary code then the output signals enabled at those markings should be the same. But if no two markings can have the same binary code then it is said to satisfy the *Unique State Coding (USC)* property.

An STG is output persistent if all the output signal events are persistent in all reachable markings and input signals cannot be disabled by outputs. This allows only inputs to be in direct conflict with each other and therefore, allowing the designer to model non-deterministic choice in the environment.

Several definitions have been given in the literature to express the idea of consistency of a STG. Examples of these definitions are: *live* STGs [16, 19, 20, 18, 21, 22], *correct* STGs [23, 24, 25, 26], *implementable* STGs [27, 28, 16] and *well-formed* STGs [16, 17].

From a Signal Transition Graph it is straightforward to obtain a state based model of the behaviour of the modelled system known as the State Graph or the Reachability Graph. It is obtained by starting from the initial state μ_0 of the STG and then by exhaustively simulating it by firing feasible transition sequences until all the states have been visited [18]. Each node of the SG is in one-to-one correspondence with the markings of the STG reachable from μ_0 and is labelled with

the binary code corresponding to that marking. An edge joins state s' with the state s if the marking μ (corresponding to s) can be reached from m'(corresponding to s') through the firing of a single transition. This transition labels the edge. The SG is thus known as the *Reachability Graph* of the STG. It is a finite state machine like description of the same behaviour as the STG.

The persistency property [18], described earlier can now be formally defined in terms of a SG. A state graph is said to be persistent if $\forall s \in S$ and $\forall a, b \in A$, $a \neq b$, if a and b are allowed in s, then ab is also allowed in S. This is also known as *output semi-modularity* [26].

The following statement was proved in [16]: *"An STG can be implemented by a Speed Independent circuit if it is consistent and output persistent."* In this work, we will be dealing with STGs that are *live, output-persistent, safe* and *consistent.*

Chapter 3

GALS Design Technique

3.1 Introduction

GALS system design has been extensively studied and several approaches have been presented over the years that address the problem of block partitioning and data synchronisation between independent blocks. This chapter reviews some of the work presented in the areas of system integration and desynchronisation techniques.

3.2 System Integration Strategies

In this section we address the strategies of integrating independent synchronous blocks. For a particular system architecture, some strategies are more desirable than the others depending on the requirements of the systems. These require-

ments include low power consumption, reliable data transfer and low latency between the input and the output requests. There is a long history of approaches that guarantee safe communication between blocks that do not share the same clock. These approaches are based on synchronous operations in the local blocks and asynchronous handshakes between them.

The first classification is based on the frequency-phase relationship between the clocks of the synchronous blocks that are required to be integrated. For example, with two clock signals, the following classifications can be made:

Synchronous: Both clocks share the same frequency, and there is no phase difference between the two clocks. In this case, no synchronisation is required.

Mesochronous: Both clocks share the same clock frequency, but there is a constant phase difference between the clocks. This can be solved by phase compensation [29].

Plesiochronous: Both clocks have nearly the same frequency, but there is a small difference. As a result of this, the phase difference between the two clocks can accumulate to an unbounded value. Adaptive synchronisation helps solve the synchronisation problem [30].

Periodic: There is a fixed ratio between the frequencies of two clocks. In this case predictive synchronisers are used for synchronisation [31].

Asynchronous: There is no frequency (or phase) relation between two clocks. For synchronising the two clocks, 2-flop synchronisers, FIFO or pausible, stretchable or data driven clock techniques can be used.

A further classification can be done on the basis of data and clock synchro-

nisation strategies. Some schemes resolve metastability, while others avoid the occurrence of metastability. The scheme that resolves metastability is:

- standard synchronisers.

The schemes that avoid metastability are :

- adaptive synchronisers.
- FIFOs.

The following subsections give an overview of the above mentioned schemes.

3.2.1 Standard Synchronisers

Asynchronous interfaces are characterised by the presence of a synchronisation mechanism. One such mechanism is the synchroniser shown in Figure 3.1. A synchroniser is a circuit which attempts to solve one of the two equivalent problems: (1) given a transition on the data signal and a transition on the clock signal, determine which occurred first; or (2) given a voltage on a data signal, determine whether it is above or below some threshold value at a given instant in time. Therefore to address the above problems and perform safe data transfer between asynchronously communicating blocks, standard synchronisers in the form of a cascade of registers can be used. A standard solution is the use of two flip-flop synchronisers, shown in Figure 3.2. The incorporation of such synchronisers between receiver and sender blocks is shown in Figure 3.2. The purpose of synchronising signals is to protect downstream logic from the metastable state of the first

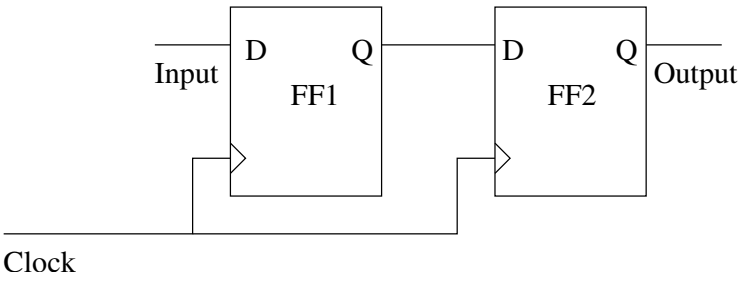


Figure 3.1: Standard synchronisers

flip-flop in a new clock domain. A simple synchroniser comprises two flip-flops in series without any combinational circuitry between them. This design most likely ensures that the first flip-flop exits its metastable state and its output settles before the second flip-flop samples it. It is also required to place the flip-flops close to each other to ensure the smallest possible clock skew between them.

Pechoucek [32] conducted a statistical analysis of the response times of a variety of synchronisers and found them to have an exponential distribution. Since, the response time was unbounded, it was concluded that the probability of failure of any synchroniser is non zero. In the work he noted that the failure rate could be reduced either by decreasing the clock frequency or by increasing the number of flip flops in the cascade, acknowledging the impact on system performance. Stucki and Cox [33] developed analytical models for synchroniser response times based on application parameters, circuit parameters and design parameters.

Unfortunately, the standard synchronisation scheme adds a latency of several clock cycles which could be undesirable for high speed data communication. Therefore, the application of standard synchronisers is advantageous for low

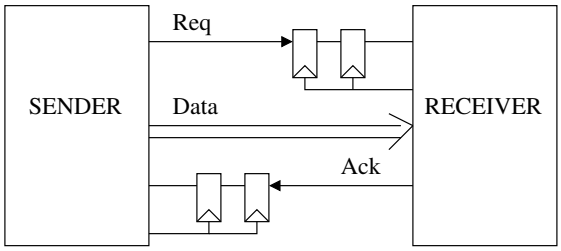


Figure 3.2: Standard two-flop synchroniser

speed data communication between the independent synchronous blocks.

3.2.2 Adaptive Synchronisers

Data adaptive synchronisation [34] adjusts the delays on the data lines instead of adjusting the local clock phase. Since the communication channels are connected point to point, the delays on them can be changed so that they do not conflict with the local clock, without affecting the other channels. When a conflict is detected, the data delay is adjusted to prevent conflicts in future communications. Therefore, in this technique the data lines are delayed as much as it is needed at a particular moment, in order to avoid metastability. Figure 3.3 shows a statistical phase detector which estimates the delay margin needed. When the most suitable delay value for a data line is found, a tuneable delay circuit connected to the corresponding data line is programmed. As a result the probability of metastability is reduced to some degree which is sufficient for most practical applications. This scheme can be used for mesochronous systems.

However, this approach does not aid reduction of power consumption and in-

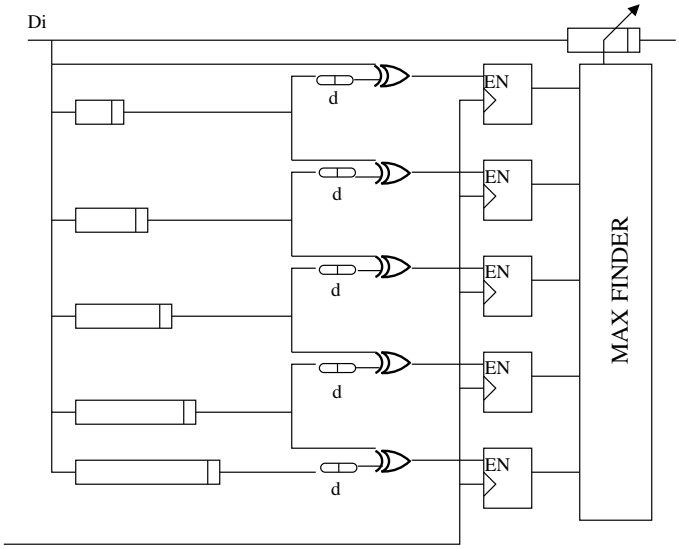


Figure 3.3: Adaptive synchroniser

troduces a large hardware overhead owing to the introduction of a separate delay line for every single data line. Moreover, it cannot be used for data transfer between blocks operating at unrelated clock frequencies.

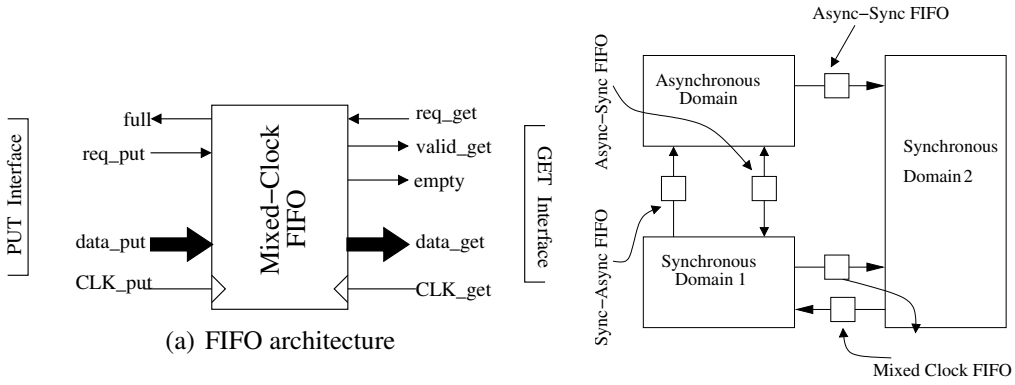
3.2.3 FIFO Synchronisation

Another synchronisation approach is interfacing the synchronous blocks with asynchronous FIFOs. Such buffers are sometimes called elastic FIFOs because their sequential depth dynamically expands or contracts depending on the amount of data they are holding. Such a system can tolerate very large interconnection delays and is quite robust to metastability. If the transmitter and the receiver are not ready at the same time or if the receiver is a little slower than the transmitter, a FIFO can be used to speed up the transfer of data. Using a FIFO, the transmitter

can transfer all data before the receiver can receive it. The FIFO acts as a buffer that temporarily stores data. There exists many ways of implementing a FIFO. Elastic FIFOs were used by Kim and Sridhar in [35] where Muller-C elements perform the handshaking between the receiver's clock and the incoming data request signal. Metastability can occur since the clock signal is not persistent, i.e., it is de-asserted after a fixed period of time even if it hasn't been acknowledged.

To avoid the bandwidth loss due to data waiting for synchronisers to resolve themselves, Seizovic [36] proposed pipeline synchronisation. One stage of the elastic FIFO is connected to the data bus in parallel with each stage of the synchroniser on the request control line. Seizovic defines a metric called asynchronicity which describes how partially synchronised the data is at each stage of the pipeline.

Adequate data throughput can be achieved via such interfaces as presented in [37]. The FIFO architecture presented in this work is shown in Figure 3.4(a). Figure 3.4(b) shows the design of an architecture when such FIFOs are integrated in the system. Such a FIFO when integrated in the system gives rise to high latency. [37] hides some synchronisation latency by inter-module FIFO buffers. The main drawback is the latency required. The STARI protocol also employs asynchronous FIFOs to achieve synchronisation at the cost of large latency as presented in [38]. Synchronisation is achieved on the first data transfer, and is automatically maintained thereafter. The FIFO must be kept about half full, and each insertion and removal operation must complete within one cycle. If these requirements are violated (e.g., on FIFO underflow), synchronisation is lost, and the system has to be



(a) FIFO architecture

(b) System integraton using mixed clock FIFO

Figure 3.4: Mixed async-sync FIFOs

restarted. Under these conditions each end of the FIFO appears to be synchronised to the local clock, so there is no chance of metastability.

3.3 Local Clock Control Scheme

Another reliable synchronisation scheme is *local clock control* scheme. This scheme is briefly introduced in Chapter 2. This section gives an overview of different schemes proposed to control the local clock.

In this scheme the independently clocked blocks are enveloped by an asynchronous wrapper. The receiver's clock can be stopped to allow reading of asynchronous data. The pausable clock is controlled by the port controllers, described in Chapter 2. The basic design of a GALS module is shown in Figure 3.5. Interconnected GALS modules combine to form a GALS system.

The local clock generator consists of an oscillator constructed from a tunable

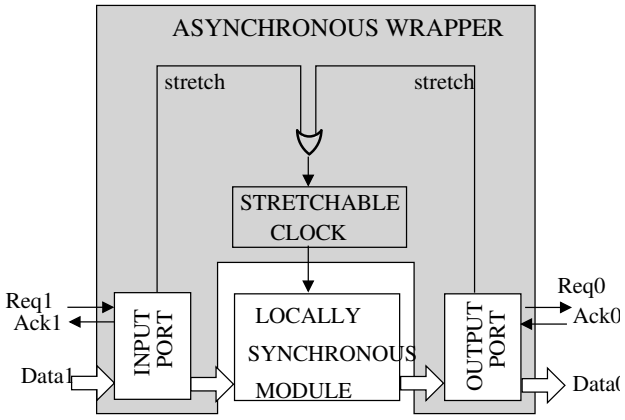


Figure 3.5: Locally clocked module

delay and an inverter. The frequency of such a clock can be periodically calibrated to an off chip reference clock. To stretch the local clock an arbitration block is placed in parallel to the delay line. The mutual exclusion elements resolve possible concurrent events between the Req. Clock signal and output from the NOR gate. Mutual exclusion (ME) allows only one of the two incoming requests to pass at a time. The element decides whether the request is granted or the next clock pulse is permitted. If all the ME agree to grant the request for clock pause then the clock signal is set. Figure 3.6 was presented in [39] and shows the scheme described above. The two incoming requests to the ME are $R1$ and $rclk$. If $R1$ is granted, asynchronous transfer is facilitated. While granting of $rclk$ produces the next clock pulse.

Another approach was presented in [40], where the scheme was demonstrated in the context of processors and memories. In this method the memory is updated through asynchronous handshake, in each clock cycle. Therefore, in each cycle

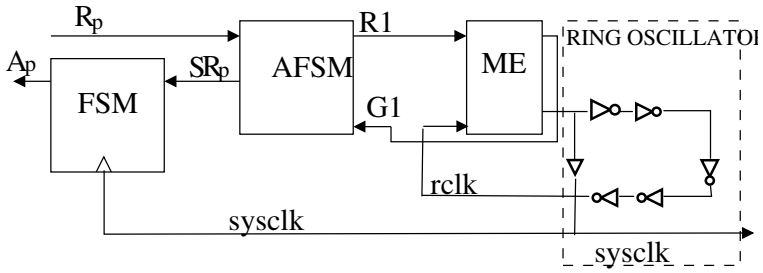


Figure 3.6: Pausible clock controller

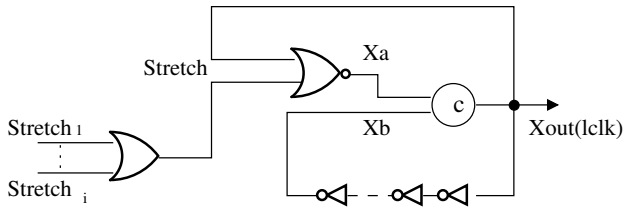


Figure 3.7: Stretchable clock controller

the clock is stretched to allow asynchronous data transfer. Such a scheme is shown in Figure 3.7.

Rosenberger et. al developed a technique in [41], to build delay insensitive modules by exploiting input registers with asynchronous handshake interfaces. These modules are referred to as $Q - modules$. The $Q - modules$ are internally clocked and are used to specify delay insensitive specifications. These modules operate in two distinct phases initiated by the falling and the rising edges of the clock. On the falling edge each input register samples its inputs and stores their present value. When the values are stored in all the input registers, the clock is released and allowed to generate the next clock pulse. The rising edge of the clock causes all the registers to update their output to equal the values stored in the inputs

of the registers. Once the update output stage is completed, an acknowledgement transition is asserted which transits to the actual computation stage.

Vanscheik et. al [42] proposed a similar scheme to $Q-modules$. In their work the state elements were called $DFLOPs$ and were implemented exclusively with digital logic. Their clock generator does not include a self timed delay. Therefore, the longest path through the combinational logic must be less than the delay incurred by the handshake. In a suggested optimisation, the amount of time allowed for metastability resolution is also reduced to the minimum handshaking delay. In such a situation, some of the $DFLOPs$ may not update their outputs until the next cycle if their inputs had switched too close to the previous clock edge.

A finer grain scheme is the elastic pipeline, similar to the elastic FIFOs but with logic between the pipeline stages and bundled delays added to he handshaking signals. Their throughput may be limited by forward data propagation, handshake control overhead or backward bubble propagation depending on how full the pipeline is.

Pechoucek outlines in [32], a clock control scheme where the generation of a fixed number of clock cycles is triggered by the availability of input data. This type of clocking scheme was recently employed to create an on chip clock generator for a DSP [43] and a data driven GALS clocking scheme for a low power reconfigurable processor [44]. Lim describes the use of a stoppable clock generator in [45]. In this scheme a single input to the clock generator prevents the clock from generating the next clock pulse until data is available. This work also describes the use of Mutex to provide an arbitrated input behaviour.

Chapter 4 presents a generic classification with the primary aim to obtain generic models for the proposed schemes in order to compare them on the basis of performance and power related system parameters. These categories are given by:

- Pausible clocking schemes
- Stretchable clocking schemes
- Data driven clocking schemes

Based on the data and clock synchronisation classification, presented in the previous section, the clock control schemes that resolve metastability are *pausible clocking scheme* and *stretchable clocking scheme*, while the *data driven clocking scheme* is able to avoid metastability. The above will be discussed in detail, in Chapter 4.

3.4 System Desynchronisation Strategies

Asynchrony can be introduced in a globally synchronous system by either removing (asynchrony) or relaxing (GALS) the synchronous constraints. Therefore, a synchronous system can be either translated into an asynchronous system by completely removing the notion of clocks or into a GALS architecture, by locally retaining the notion of clock. This section highlights various approaches proposed to achieve the above.

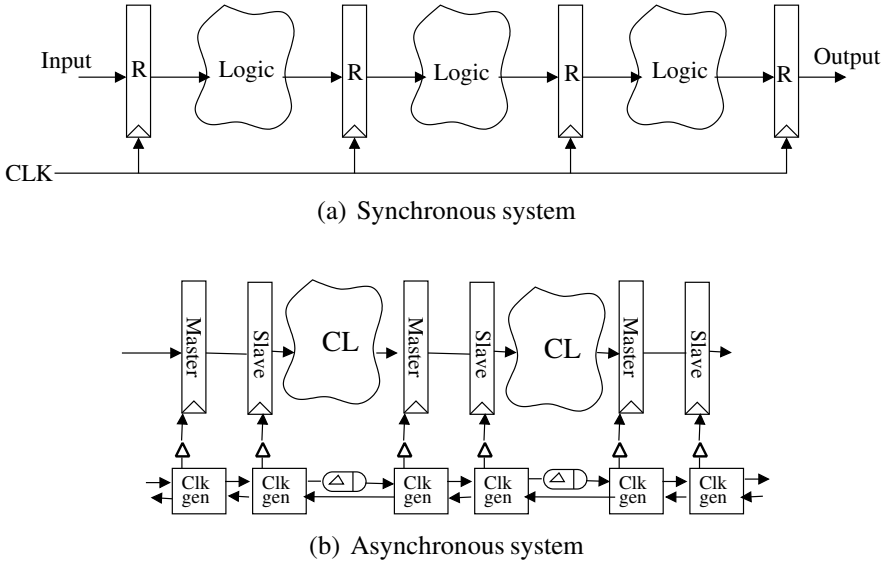
3.4.1 Asynchronous Deployment

Several researchers have proposed different approaches to automate the design of asynchronous circuits [46, 47] instead of using asynchronous design tools. Instead a synchronous system is converted into an asynchronous system, in order to use standard synchronous design tools in the system development. In [48] Blunno presented a de-synchronisation model which substituted the clock network with a set of asynchronous controllers. They investigated different concurrency degrees in different handshake schemes and proposed a controller with maximum de-synchronisation. They compared a synchronous and de-synchronised version of the DLX microprocessor. They did not report any large differences between the synchronous and the de-synchronised microprocessor, when comparing area, speed and power consumption. Figure 3.8 shows an example of de-synchronisation. The system shown in Figure 3.8(a) is desynchronised into the one shown in Figure 3.8(b).

Desynchronisation approaches targeting hardware design have been presented both by Jacobson et al. [49] and Cortadella et al. [50]. Their basic idea is to start from a fully synchronous synthesised integrated circuit, and then replace the global clock network with a set of local handshaking circuits.

3.4.2 GALS Deployment

To construct an asynchronous wrapper that controls the input, output and clock generations from each synchronous module is non-trivial and has been exten-



(a) Synchronous system

(b) Asynchronous system

Figure 3.8: De-synchronisation technique

sively studied. There are different approaches that implement synchronous specifications over GALS architectures. They are mainly based on latency-insensitive protocol, endochrony and Kahn process Networks (KPN). All these approaches follow the same pattern of transforming the components of a system based on initial specification into equivalent synchronous components whose interface is modified in such a way that they can be considered self-timed.

Latency-insensitive protocols were proposed in [51] and, then, applied to synchronous hardware design in [52, 53]. A complete presentation of latency-insensitive design is given in [54], which includes a detailed discussion of the analysis and optimisation of latency insensitive systems. The application of latency-insensitive design to integrated circuits provides two main advantages [52]:

(a) automatic pipelining of long wires is enabled by the insertion of patient

processes (a module is a patient process if its behaviour does not depend on the latency of the communication channel because it is compliant with the *latency insensitive communication protocol*) called relay stations [55]. ;

(b) it eases the assembly of different components that are pre-designed which can be interfaced to the communication protocol without changing their internal structure as long as these components are stallable.

Casu and Macchiarulo have proposed an alternative implementation for the building blocks of latency insensitive systems which applies to the particular case when the computation of each core module can be scheduled statistically [56]. This implementation can be used only with closed systems. In latency insensitive protocol, each synchronous component reads each input and writes every output in each reaction. It simplifies implementation. The main disadvantage of this protocol is that it simulates a single clocked system and hence is proposed for single clock architectures only.

Benveniste [57, 4] formally defined the desynchronisation problem in reference to embedded system applications. Their main motivation is to address the issue of compositionality of synchronous languages and enable modular code generation. Informally, endochronous systems are characterised by a condition that the presence and absence of all variables can be inferred incrementally during each reaction from already known values of the present input variables and state variables of the Synchronous Transition System (STS) under consideration. In particular, they advocate a methodology centred on the use of the synchronous paradigm for system specification and validation followed by a provably correct desynchroni-

sation step to derive a distributed implementation (e.g. on GALS architectures). Endochrony can be both model checked and synthesised. Unlike latency insensitive protocol, the endochronous approach takes into account execution modes and independence between components in order to minimise communication and allow multi-rate computation.

Unlike latency insensitive systems, endochronous systems can take into account different execution modes and independence between components in order to minimise communication and allows multi-rate computations. The main drawback is poor handling of concurrency and hence endochronous systems are not compositional. This leads to inefficient synthesis of systems formed of more than two components. Though KPN [58] is the only approach formulated in causal framework, the main disadvantage is its strong determinism criteria, since non-determinism is often useful in the specification and analysis of concurrent systems.

A mathematical framework to support the composition of heterogeneous reactive systems is presented in [5] together with a set of theorems supporting the automatic generation of correct-by-construction adaptors between heterogeneous designs. The idea is applied to the deployment of synchronous design on GALS architectures and Loosely Time-Triggered Architectures (LTTA) [59]. The Polychrony project aims to support design refinement from the early stages of requirement specification to the later stages of synthesis and deployment [60, 61, 62, 63]. The term polychrony denotes the capability of describing circuits and systems using the synchronous assumption together with multiple clocks. This can be

applied to abstracting the key properties of a system as well as to describing the characteristics of the components that can be used to implement it. The concept of polychrony is used in [61, 64] to address the formal validation of the refinement of synchronous multi-clocked designs into GALS architectures. In this area Talpin and Le Guernic presented a process algebraic theory of behavioural type systems and applied it to the synthesis of latency insensitive protocols. They showed that the synthesis of component wrappers can be optimised using the behavioural information carried by the interface type descriptions to yield minimised stalls and maximised throughput.

The work of Berry and Sentovich in [65] studies the issue of asynchronous interaction between synchronous Esterel programs. The main issue addressed in this work was to prevent over writing messages due to asynchrony by blocking the sender when the single place buffer is full. Although in this way, the size of the buffer is restricted to 1 and hence reducing latency, the parallelism and pipelining is decreased. In [66], distribution of synchronous sequential programs is discussed. In this approach, the asynchronous interaction between the components is encapsulated in send and receive commands and the work mainly concentrates on finding the appropriate places for sends and receives in order to minimise communication and maximise parallelism.

Implementing asynchronous systems using synchronous languages is also studied in [67]. This work presents a general semantic model for the synchronous and asynchronous computation. The main attention is given to the implementation of communication mechanisms such as mutual exclusion elements and rendezvous

scheme. It must be noted that though these mechanisms form an integral part of asynchronous design, no methodology was presented to obtain components instrumented with the structures from synchronous components. The work presented in [68] models asynchrony (interleaving semantics) in the I/O automata model using synchronous communication scheme. However, due to differences between the models of computation for asynchronous and synchronous systems e.g., the input enabling constrains in I/O automata, the notion of buffer is implicitly addressed in the semantics.

These methods have evolved over the past few years and there exists a methodology that improves on the previous work in terms of scheduling independence and compositionality, integral properties for modular designs. This leads to introduction of *weakly endochronous system*s [69]. They support signalisation schemes that are simpler and more efficient than latency insensitive and endochronous counterparts. The main disadvantages of WE systems is the absence of efficient methodologies to implement the wrappers.

3.4.3 Endochronous and Weakly Endochronous Systems

A solution to desynchronisation consists of recreating non-strict synchrony by adding extra signals that act as clocks for the desynchronised signals. The technique of associating a Boolean clock with a signal is called *Booleanisation.* At each tick of a given clock C, it is checked whether a signal s defined in that clock domain is present or not. If it is present, then the clock signal of s is written as $C\_s = 1$ and if not then $C\_s = 0$.

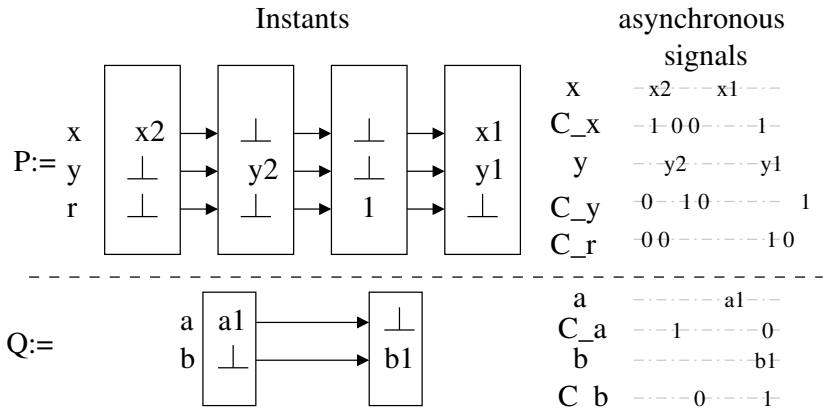


Figure 3.9: System synchronisation

As discussed in the previous section that in an endochronous system the presence or absence of a signal can be inferred incrementally from already known values and variables present in the system. Input clocks in an endochronous system have sufficient relations to infer the presence or absence status of all the signals of the system at all times. The main problem with such an approach is that it does not extend to composition of two or more endochronous systems. This is because if the modules are ruled by different clocks, it is impossible to build a global process, reading their respective asynchronous signals and clocks. This is due to lack of information to resynchronise the flow of both the signals. Therefore, composition is only possible if the two processes are governed by the same clock or one is a fraction of the other. In such a case, the processes require a protocol to first synchronise their clocks and then keep them synchronised.

Figure 3.9 illustrates a counter example where the processes P and Q have unrelated clock signals. It can be seen that synchronous or asynchronous signals are

not composable as a common process. This is due to the lack of synchronisation information between the signals from the two processes since they have unrelated clocks. The signals from the different processes can have different synchrony relation [70]:

- The processes communicate together and share a clock
- Processes do not communicate but share the same clock from a third process
- Processes do not communicate and the signals are not related.

For the first two cases, an external scheduler can reconstruct the instants. But endochrony fails in the third case.

To address the issue of compositionality in concurrent execution of synchronous modules, Potop et.al. [69] defined a more relaxed assumption than endochrony called *Weak Endochrony.* Informally, weak endochrony is a speed-independence property which characterises synchronous components whose behaviour does not depend on the order in which various inputs are read. This approach allows the asynchronous composition of the modules to meet determinism. With this approach, minimal reactions need to be constructed and synchronisation clocks defined and associated to reactions. The reactions are then scheduled to use to available data efficiently. For maximised concurrency within a synchronous block the data should be used as soon it is available and this entails decomposition of the synchronous system into small blocks. Each of these blocks are activated on the arrival of input data. Each of these blocks are equipped with a single clock and

Module M1
input R1, R2, K
output A1, A2
relation R1#K R2#K
abort
  loop await R1
    emit A1 end
||
  loop await R2
    emit A2 end
when K
end module

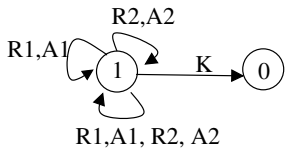


Figure 3.10: Mapping of a synchronous program to $LSTS$

shares the inputs from interface of the main process and the outputs from the other blocks. The idea of *Weak Endochrony* is described in more details in Chapter 5.

3.4.3.1 Transition System model

This section introduces the modeling framework used to define weakly endochronous systems. The synchronous programming has been proposed as an effient approach for the design of reactive and real-time systems. Some examples of such languages are *ESTEREL, LUSTRE* and *SIGNAL.* Mapping SIGNAL programs to distributed architectures was proposed in [71]. Mapping of LUSTRE programs onto a network of automata communicating asynchronously via unbounded FIFOs were presented in [72]. [65] presented a technique, using ESTEREL synchronous language, for modeling synchronous system which can be desynchronised, although the asynchronous model was not fully stated.

These languages provide statements that emulate the actions of a synchronous

system. These programs can be mapped to a mealy machine using behavioural semantics of [73]. The mapping entails corresponding each state with a program term and the behavioural transitions are translated directly. One such Mealy machine is the Labelled Synchronous Transition System ($LSTS$). The underlying model of computation of these programs is easy to understand and efficient in describing embedded applications where accuracy and correctness are of paramount importance. But due to compilation complexity and difficult to follow excution models of these languages, $LSTS$s are used instead, which are obtained by mapping the synchronous programs onto states and transitions of $LSTS$s, to model synchronous and GALS systems.

Implementations of synchronous programs loop continually executing the following three actions 1) reading inputs; 2) computing and emitting outputs; and 3) computing and storing the next state. The mapping of such programs onto $LSTS$s is shown in the following example:

Figure 3.10 shows the $LSTS$ of the program fragment presented. The system executes $R1$ and $R2$ producing $A1$ and $A2$ respectively. The process is aborted on the reception of K.

Chapter 4

Comparative Analysis of GALS Clocking Schemes

4.1 Introduction

This chapter presents the comparison between three different GALS approaches. This comparison highlights the advantages and disadvantages of the three design solutions based on circuit implementations, power and performance analysis. The implementation of synchronous computational blocks are not cycle accurate, while the communication blocks are modelled in a cycle accurate manner. Petri nets excel in their usefulness to model systems at higher levels of abstraction and tools like Petrify aid their translation into a gate level implementation. Petri net modeling provides the designer with fast verification and implementation of the system. This chapter presents the Petri net models of the three GALS architec-

tures. These models are verified for correctness using in-house verification tools PUNF/CLP [74]. The verified models are fed into Petrify to produce logic equations for gate level implementation. We use two pre-synthesised blocks, namely, Mutual Exclusion Element (ME) [75] and FIFO [76] and these are plugged into the circuit implementation of each of the three schemes. The gates were implemented within the Cadence toolkit using the AMS CMOS 0.35 technology library. The flow of the chapter is depicted in Figure 4.1.

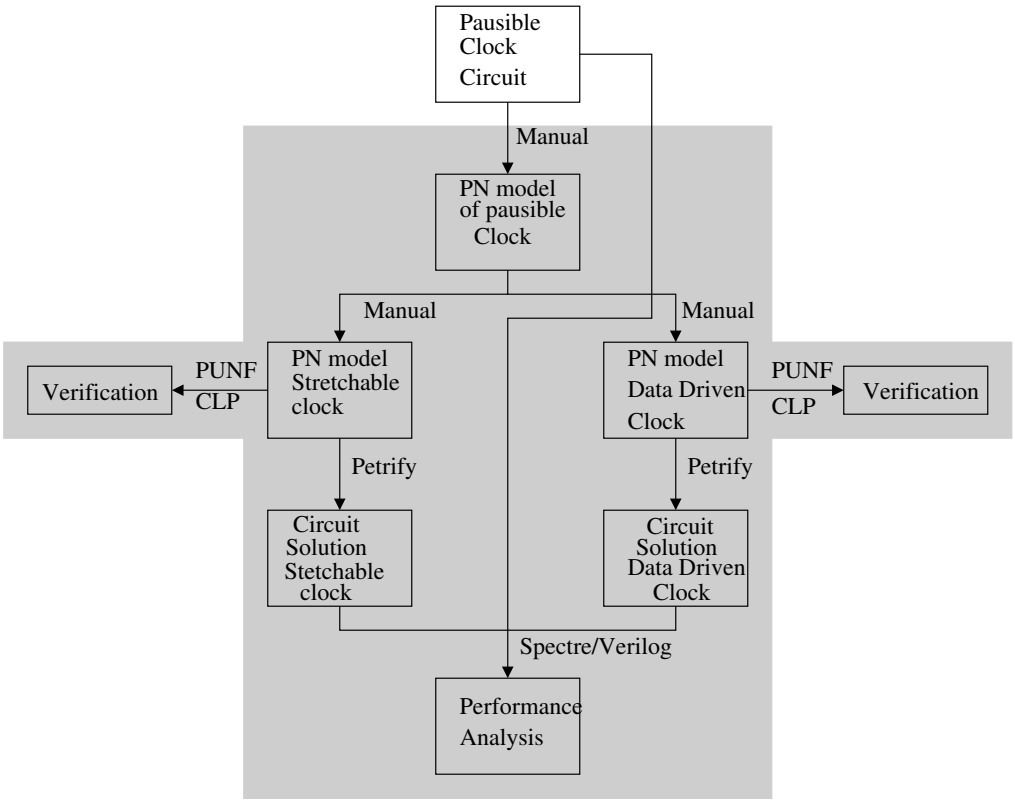


Figure 4.1: Design flow

A novel design solution is presented for stretchable and data driven clocking

scheme from prevalent conceptual models. The GALS architecture with pausible clocking scheme is obtained from [77] and compared for efficiency and power consumption with the above mentioned approaches.

4.2 Overview of the GALS system

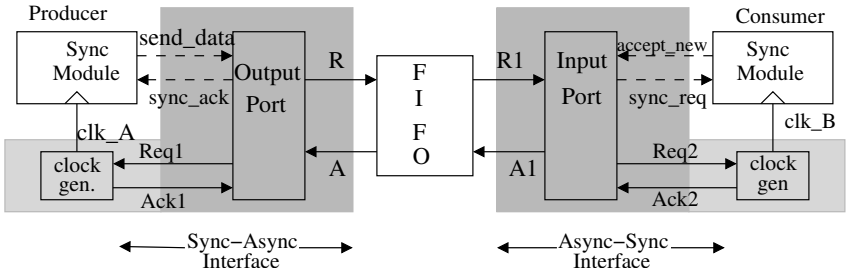


Figure 4.2: Overall system architecture for producer-consumer interface

In the domain of locally clocked control schemes, introduced in Chapter 3, the GALS implementation for a given multiprocessor system can be broadly divided into three clock control architectures depending on their type of control. These are pausible, stretchable and data driven clocking schemes. These schemes are extended to a system with two clocked domains, one producer and the other receiver, to replicate communication between two synchronous islands. The two clocked domains communicate via a two stage asynchronous FIFO. Such a system architecture is shown in Figure 4.2. The signals $Req1$ and $Req2$ will be replaced by a pause clock, stretch clock or start clock request depending on the type of clocking scheme.

The clock generator block is controlled by asynchronous port controllers, namely, the Input Port (IP) and the Output Port (OP). Such a system is scalable, with as many IPs and OPs as there are inputs and outputs from a particular synchronous island. A request-acknowledge pair of handshake signals accompanies each data entering or leaving the synchronous module. The validity of data is signalled by a four phase protocol depicted by $R+$, $A+$ $R-$ $A-$ and data is guaranteed to be valid between $R+$ and $A-$. The clock generator sends clock pulses to the synchronous module to carry out synchronous computations, while the communication is inactive and vice versa.

4.2.1 Models of the clocking schemes

The Petri net (PN) models help in formalising the behaviour of asynchronous circuits. For that, PN models require special interpretation (see also Signal Transition Graph in Chapter 2). Here we recall some PN notations. In PNs the events of the signals are labelled as $+$ and $-$ symbols. A transition of a label named $R+$ indicates the event of the rising edge of the signal R, while $R-$ denotes the falling edge of R. A transition can only fire if there is at least one token (in our case, exactly one token) in its pre-place. Firing of the transition consumes a token from each of its pre-places and assigns a token in each of its post places. The adaptability and usefulness of PN modeling are demonstrated by two following features: (a) Extraction of PN model from a circuit level solution for verification and (b) Use of a PN model to form the specification of a new control system for synthesis. This work exploits both the features of PN models.

Figures 4.5, 4.6 and 4.7 depict the models of the consumer block, i.e. the async-sync interface, of each of the three GALS clocking schemes. We start with the circuit solution of the pausible clocking scheme, presented in [77], shown in Figure 4.8(a). We extract a PN model from this circuit solution, as depicted in Figure 4.5. From the specification of stretchable and data driven clocking schemes and using this PN as a reference model, we then obtain the PN models for the stretchable and data driven clocking schemes.

Figure 4.3 shows the PN fragments derived from the causal relationship between signals for different parts of the circuit. The setting or resetting of a signal in the circuit is manifested in the PN model by the arrival of a token in the pre-place of the rising or the falling signal event, respectively. Figure 4.3(a) and (b) show the Mutual Exclusion element (ME) block and its corresponding PN, respectively. The interface signals of the ME block are $r1$, $r2$, $g1$ and $g2$. If signal $r1$ arrives before signal $r2$, then $g1$ is granted. When $r2$ goes low, $g1$ is de-asserted. While the signal $g1$ is asserted, $g2$ cannot be asserted. Such a causal relationship is depicted in the PN fragment. Similarly, a causal relationship can be derived from the other parts of the circuit. Let us consider the example shown in Figure 4.3 (c) . This part of the circuit consists of a XOR gate, a mutual exclusion element, shown as a black box in this example, as we have already considered it in the above example, and a latch. The signals $R1$ and b are the inputs to and $r1$ is the output from the XOR gate. Signal $r1$ is the input to the ME. As we treat it as a black box there is no conflict with another request. The signal $g1$ acts as a clock to the latch, which has $R1$ as its input and b as its output. The Petri net fragment in Figure 4.3 (d)

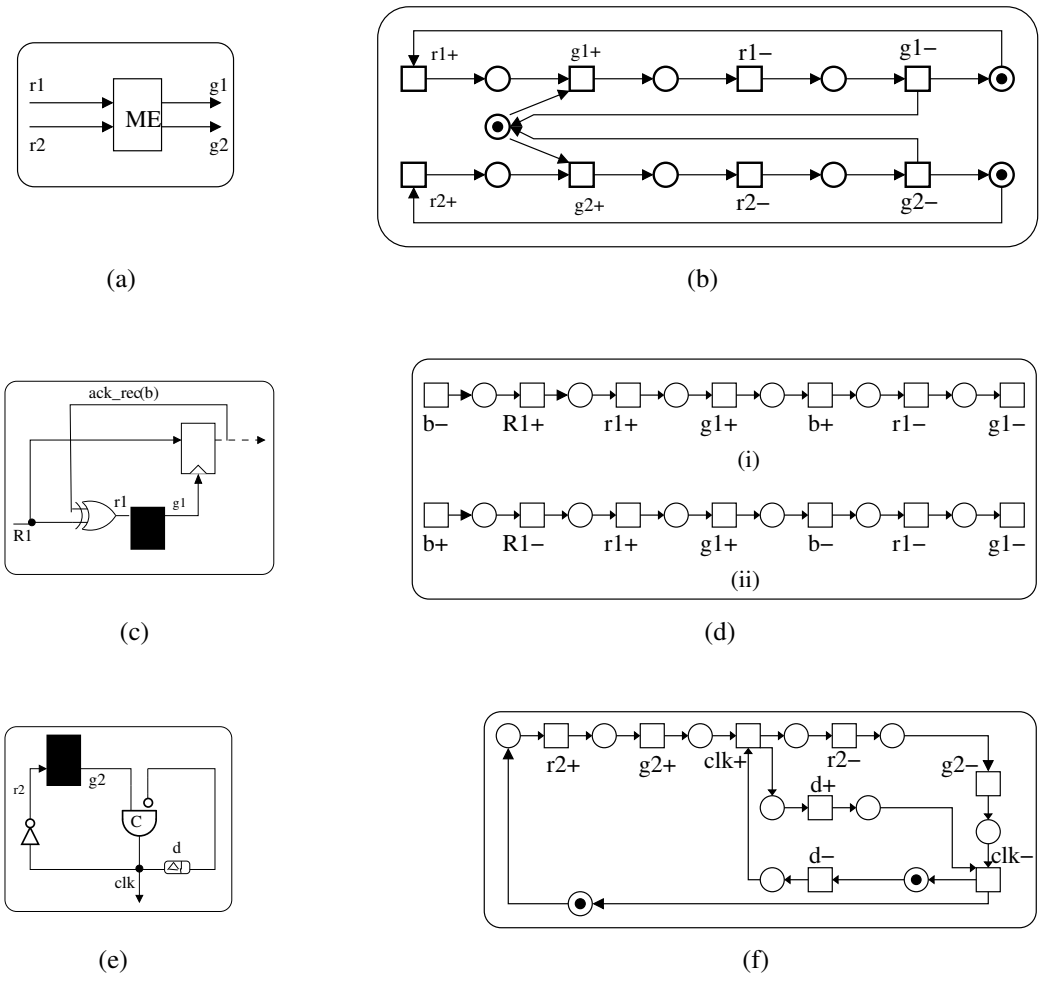


Figure 4.3: Circuit blocks and PN fragments

shows two situations that lead to the setting and eventually, resetting of the input to the black box, i.e $r1$. The dashed lines in the figure correspond to eventual occurrence of a signal. For example, the dashed lines between $b-$ and $R1+$ denotes that $b-$ eventually causes $R1+$ and there are transitions in between which, for simplicity, have not been shown. A similar causal relationship can be derived for

the locally generated clock circuit shown in Figure 4.3 (e) and its corresponding PN model depicted in Figure 4.3 (f).

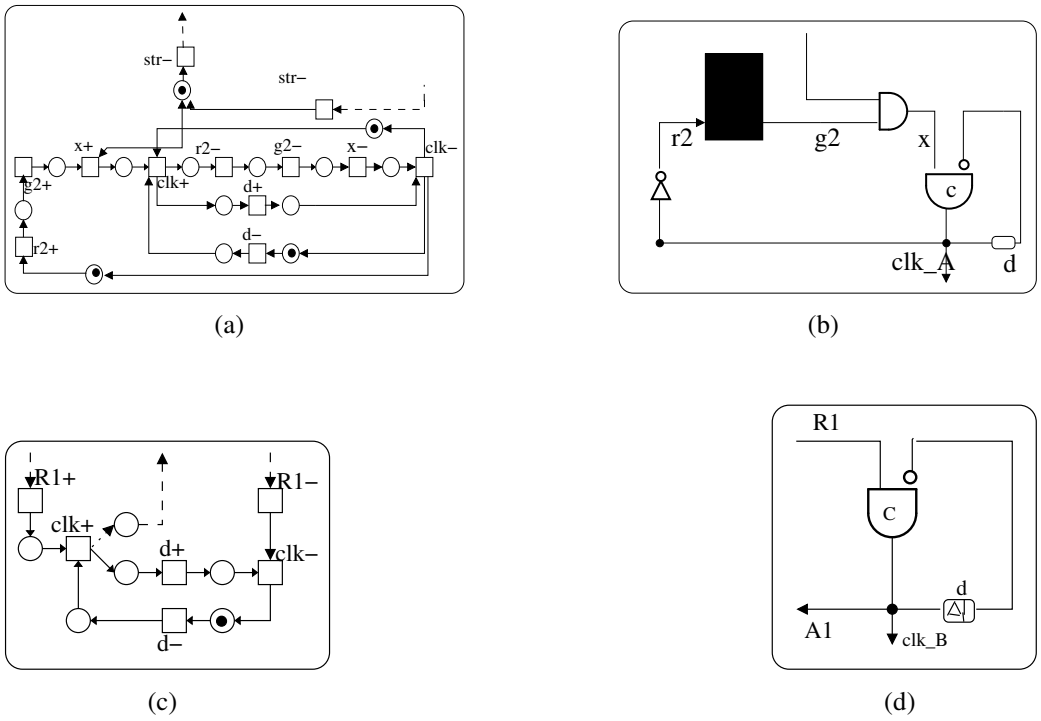


Figure 4.4: Clock control circuits and their corresponding PN models

Based on the PN model of the pausible clock and the specification requirements of stretchable and data driven clocking schemes, we obtain the PN models of the schemes. The clock control architectures for the above mentioned schemes are depicted in Figure 4.4. The PN models, shown in Figure 4.4(a) and (c), are similar to the pausible scheme and only differ in the way the clocks are started and stopped. Figure 4.4 (b) and (d) depict the corresponding circuits for these models. Once started, they follow the same pattern as the pausible scheme. Figure 4.4(b)

and (d) show the clock control architectures of the stretchable and data driven schemes obtained from the PN models. The port controller models, shown in Figure 4.6 and 4.7, are derived from the basic specification of stretchable and data driven clocking schemes. The overall functions of the three clock architectures, when the PN fragments are put together, are discussed below. The PN models are subsequently used for model level analysis and the corresponding circuits are used for the circuit level analysis of the three systems.

It is noted that, for the sake of simplicity, the interaction of the communication interface with the synchronous module, as shown in Figure 4.2, is not shown in the PN models presented in Figure 4.5, 4.6 and 4.7. These signals include the signal $sync\_ack$ (input to the synchronous module) and $send\_data$ (output from the synchronous module) on the producer side and signal $sync\_req$ (input to the synchronous module) and $accept\_new$ (output from the synchronous module) on the consumer side. When the synchronous module, on the producer side, receives signal $sync\_ack$, it releases an enable signal $send\_data$ which denotes the availability of data to be sent to the consumer block. Similarly, on the consumer side, the synchronous request($sync\_req$) is sent to the synchronous module after the reception of enable signal $accept\_new$ from it. Therefore, the dotted lines in three models denote that signal transition $b+ \rightarrow A1+$and $b- \rightarrow A1-$ take place in the presence of the enable signal $accept\_new$, produced by the consumer synchronous module, as discussed above.

4.2.2 Discussions of the models

Pausible clock: The pausible clocking scheme offers an elegant solution to metastability issue which comes into play when there is a cross domain communication. Pausible clocks are characterised by a free running clock. A Mutual Exclusion (ME) element is inserted in the circuit to allow the clock to be interrupted when an item of data is ready to be transferred. The interruption of the clock enables safe transfer of asynchronous data. The Petri net model of the async-sync interface (consumer side) of this system is shown in Figure 4.5. The signal $r1$, produced by signals $R1$ and b, requests for a clock pause, while signal $r2$ requests a clock grant. Signals $g1$ and $g2$ are mutually exclusive and granting of $g1$ interrupts the clock. This leads to an asynchronous data transfer. This request is acknowledged on reception of the positive edge of the clock signal. Once the clock goes low, it is triggered again after a tunable delay d.

Stretchable clock: A stretchable clock can also be viewed as a free running clock, like the pausible clock. The difference between the two is that a stretchable clock knows in advance that the next clock cycle should wait for an asynchronous input. Therefore, only in the absence of input request signals, the clock would be free running. This architecture leads to an increased throughput, since the request does not have to compete with the clock for an asynchronous data transfer. The async-sync interface of this system is depicted in Figure 4.6. The signal $clk+$ can only proceed if signal str is low, which denotes that there is no data to transfer from the FIFO. When the signal str is high, the data is transfered from the FIFO to the consumer block. The request received is acknowledged when the clock

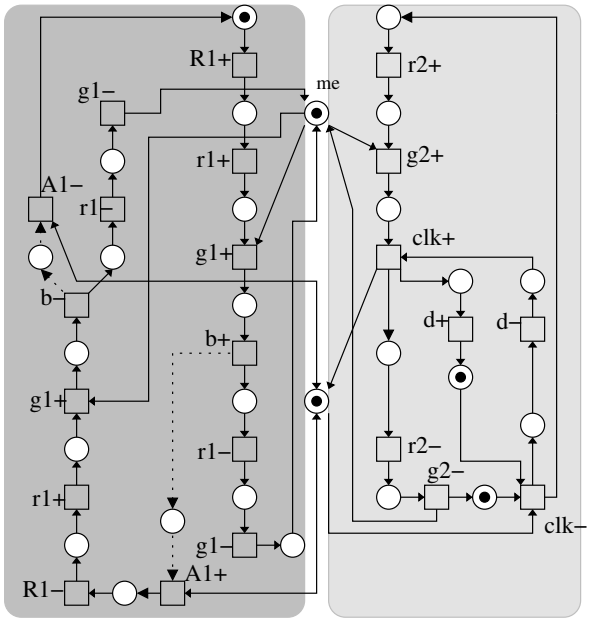


Figure 4.5: PN model of pausible clocking scheme

goes low. The delay line, like the pausible clocking schemes delays the rising and falling of the clock signal clk. The delay line is parallel to the arbitration block in the local clock generator block denoted by the light grey shaded portion.

Data driven clock: In data driven clock scheme clock edges are produced in response to the presence of data at the input ports of the *IP* block. Therefore, the clock is not free running, unlike pausible and stretchable clocking schemes. The Petri net model of the async-sync interface of this system is shown in Figure 4.7. Signal clk is asserted on the reception of the positive edge of the signal $R1$. Signal clk is de-asserted on the reception of the negative edge of $R1$ and after a tunable delay d. The clock is inactive in the absence of signal $R1$.

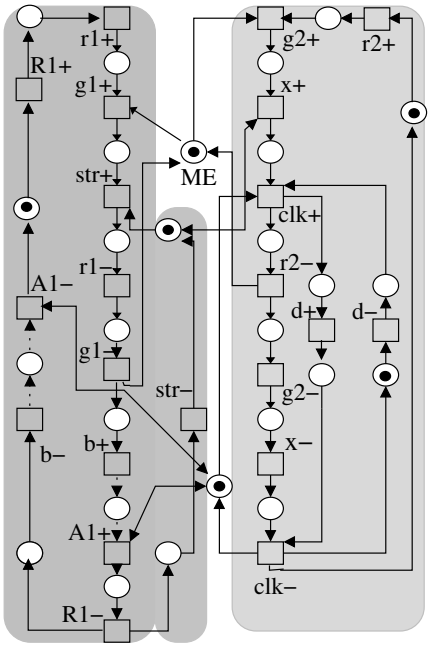


Figure 4.6: PN model of stretchable clock scheme

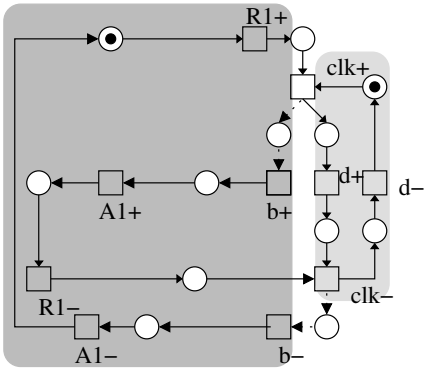


Figure 4.7: PN model of data driven clock scheme

4.3 Verification and Logic Synthesis

The PN models have been constructed in the PEP tool and verified for functional properties like safeness and deadlock freedom using in-house tools PUNF/CLP.

These models effectively present a special class of formalism called Signal Transition Graph (STG). Synthesis based on STGs involve the following steps [78]: (i) checking sufficient conditions that are required for the implementation of an STG in a hazard-free logic circuit, (ii) if (i) is not satisfied, then modifying the model to make it implementable, and (iii) finding the appropriate next state function for non-input signals. The tool Petrify performs all the above tasks automatically. On successful completion of these tasks it can proceed to generate logic equations for the circuit's gates, to implement the STG.

The circuit implementation for stretchable and data driven clocking schemes have been obtained from the logic synthesis of their respective STG descriptions, using the above tool. In order to logically synthesise a given STG, using Petrify, it is necessary to check that it satisfies safeness and liveness properties.

The tools PUNF and CLP read a PN and perform its verification on the finite and complete unfolding prefix of the PN [74]. For each new firing a new transition, called event, is generated and for each newly produced token a place, called condition, is generated. The unfolding prefix is therefore a finite acyclic Petri net graph on which it is computationally easier to carry out various model checking procedures. The statistics obtained after verification are listed in Table 1. This table presents the number of conditions and events generated from each of the models. These numbers are an indication of the size and complexity of the circuits obtained from these PN models. It also shows that each of the models satisfy safeness and liveness properties. We also present the statistics for pausible clocking scheme for the sake of comparison with the other models.

The verified models of the sync-async interface (producer side) and async-sync interface (consumer side), depicted in Figure 4.6 and 4.7 together with the FIFO were composed together to form a closed system as depicted in Figure 4.9 and 4.10.

Model name	|B|	|E|	Liveness	Safeness
Pausible clock	115	81	√	√
Stretchable clock	37	30	√	√
Data driven clock	20	15	√	√

|B| = Number of Conditions

|E| = Number of Events

Table 4.1: Verification statistics for Petri net models of GALS architectures

4.4 Circuit Implementation

In this section, we present the circuit implementation of the three clocking schemes. The systems consist of producer and consumer synchronous modules communicating via a two stage FIFO. The leftmost block and the FIFO block constitute the interface between synchronous producer and asynchronous receiver, while the block on the right and the FIFO denote the interface between asynchronous producer and synchronous consumer.

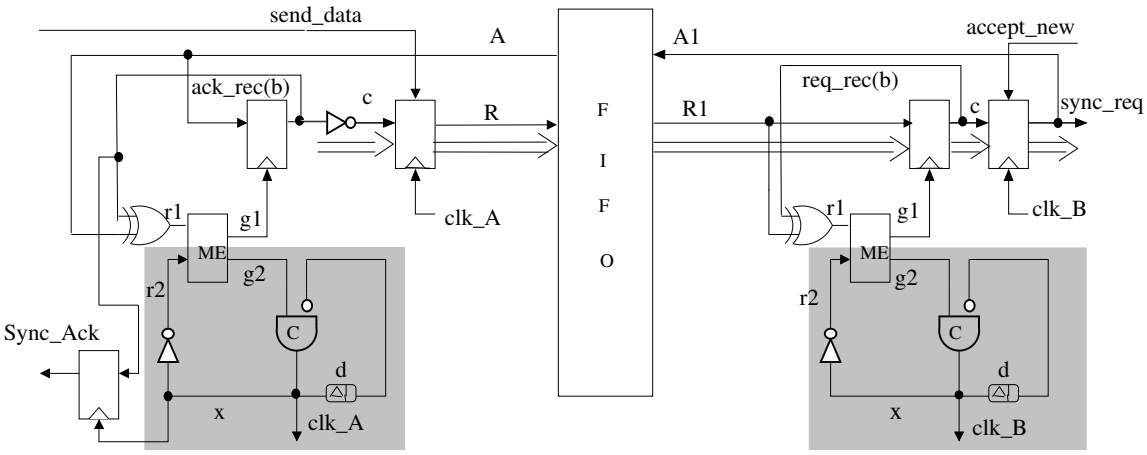


Figure 4.8: Pausible clock circuit

Pausible clock: The implementation of a producer-consumer block over an asynchronous interface is depicted in Figure 4.8. The asynchronous interface arbitrates between granting in favour of the $r1$ signal, to transfer data to subsequent synchronous blocks or a clock request to generate clock (clk_A) for its locally synchronous module. If signal $r1$ is granted, the data is latched in the first latch and the hold is released from the ME. This allows clock request to win over the ME. Therefore, data is stable before the clock arrives at the next stage of latch avoiding metastability at the second edge triggered latch. Figure 4.11(a) shows the phase relation between signals clk_A, ack, ack_rec, $sync\_ack$. The shaded portion denotes the window when asynchronous data is received. The synchronous module always waits for a synchronous signal syn_ack. On its reception, the module releases an enable signal for new data transfer. This type of design methodology is also explored in [39].

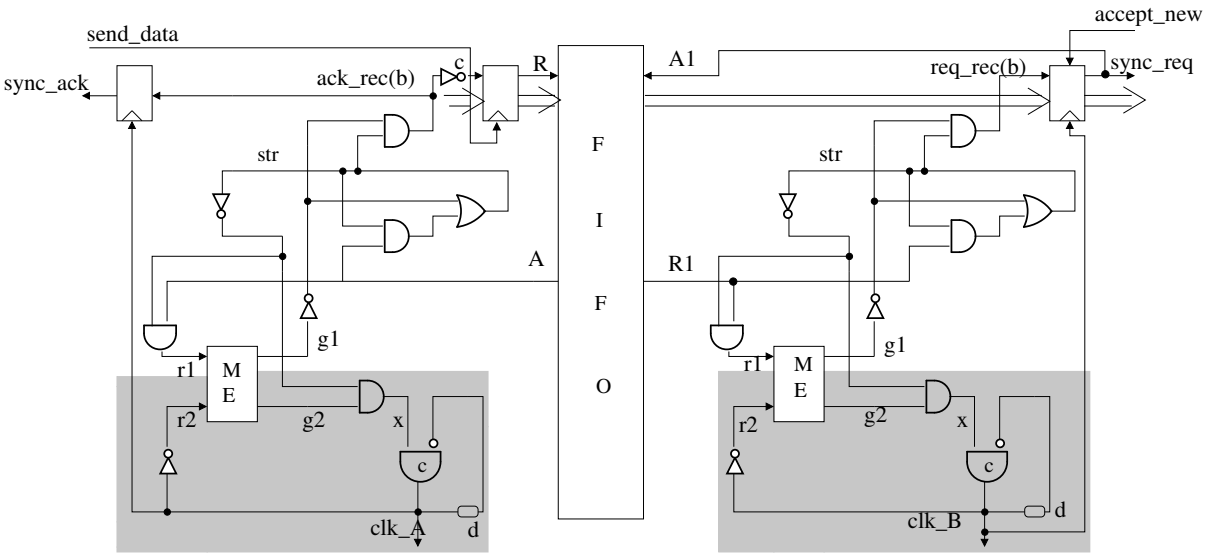


Figure 4.9: Stretchable clock circuit

Stretchable clock: The system architecture of this scheme is depicted in Figure 4.9. The assertion of the stretch signal (str) prevents the clock from going high before the assertion of signal ack_rec (in producer block) or req_rec (in consumer block). The assertion of signals ack_req and req_rec lead to the de-assertion of R (in producer block) or assertion of $A1$ (in consumer block), respectively, which in turn de-asserts signal str. On the producer side, the synchronous module waits for a synchronous $sync\_ack$, in a manner similar to the pausible clocking scheme. Hence, signal ack_rec has to be synchronised to the clock to produce $sync\_ack$. As can be seen from the stretchable clock architecture in Figure 4.9, signals ack_rec+ and $clk+$ are mutually exclusive due to signal str (this can also be seen on the consumer side (async-sync interface) of the system, in the Petri net models shown in Figure 4.6, where req_rec+ is mutually exclusive to $clk+$).

Therefore, the positive edge of signal ack_rec cannot be synchronised on the positive edge of signal clk. If the signal ack_rec is synchronised to the negative edge of the clock cycle with a flip-flop, the system could run into a deadlock. This is due to the fact that if signal clk has already gone low before the triggering of signal $str+$, and then if $str+$ occurs stopping signal $x+$(which causes $clk+$) from firing, signal $ack+$ would wait for the falling edge of signal clock, which would not be triggered till $str-$ occurs . Hence, ack_rec+ will never meet the set up and hold time of the falling edge of clock signal. Therefore, the only solution is to use a latch, instead of a flip flop. The latch is made to sample the signal ack_rec when the clock is low. This synchronised ack_rec is then sent to the synchronous module, which in turn sends an enable signal to indicate a data-ready-to-send status. This enable signal latches the $ack\_received'(c)$ in the final set of latches to assert the request signal for sending new available data. A similar scheme is presented in [40] and [41]. A phase relation between signals *clk_A, ack, ack_rec, Sync_ack* at the sync-async interface for stretchable clock, similar to pausible clocking scheme, is depicted in Figure 4.11(b).

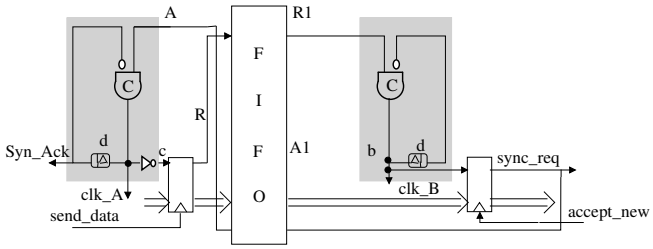
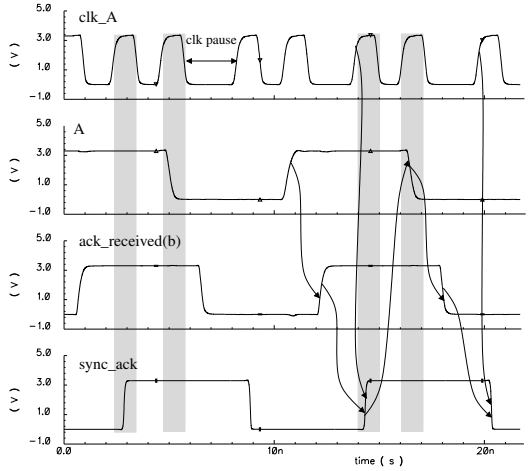


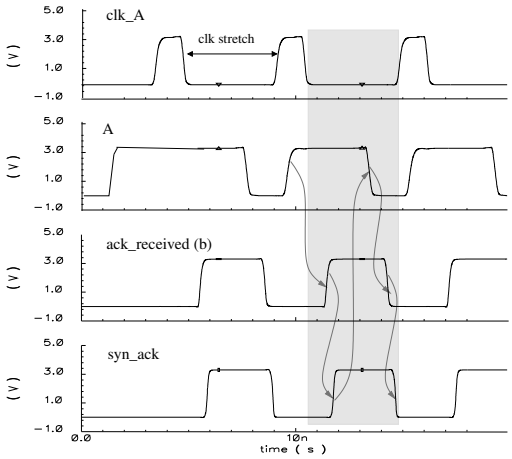
Figure 4.10: Data driven clock circuit

Data Driven clock: Such an architecture is depicted in Figure 4.10. Since,

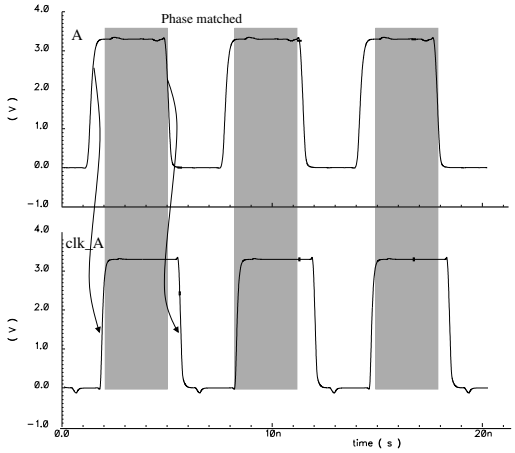
power is an important issue in SoC applications, design methodologies which provide circuit solutions with reduced power consumption becomes highly attractive. In this scheme, the local clock oscillates at a frequency determined by the availability of data signalled by the request signal. Therefore the circuit is switched off when there is no data to send. This scheme significantly reduces power consumption as clock is only started when enough inputs have been received to carry out a particular computation. Unlike the previous two clocking schemes, there is no added synchronisation required for the ack_rec/req_rec, since the signals are already synchronised to the clock and can be directly sent to the synchronous module on the reception of enable signals, as denoted in the figure. An extensive design solution for this approach can be found in [79] and [80]. The simple phase relation between $a1$ and clk_A, on the sync-async interface for this scheme is shown in Figure 4.11(c).



(a) ack+->ack- phase relation for pausible clock



(b) ack+->ack- phase relation for stretchable clock



(c) ack+->ack- phase relation for Data driven clock

Figure 4.11: Asynchronous communication-phase relationship at the producer block

4.5 Performance Analysis

It is assumed that the system is partitioned logically into synchronous islands and that they communicate with other synchronous blocks through an asynchronous interface. The asynchronous interface interacts with the clock generator circuit of these synchronous blocks for cross domain data transfer. The Petri net models of the asynchronous interface and clock control circuit developed have been fed to Petrify for the generation of logic equations to build the gate level implementations of the architectures. The analogue and digital partitions of the circuits have been simulated using the SPECTRE and Verilog simulators within the Cadence framework. We used mixed signal simulations to aid the monitoring of several signals using digital specification, while leaving other parts of the circuit to run analog simulations. The design has been incorporated with various digital blocks to reduce the time taken for analog simulation.

4.5.1 GALS system characterisation parameters

To characterise any design based on SoC applications, we need to define some metrics that are applicable to power and performance of a system. Similarly, we define such metrics for the GALS systems. These metrics have been evaluated to analyse an architecture for studying the effects of different system parameters on the performance of the system.

The metrics that are relevant for the analysis of pausible clock circuitry of GALS architecture are the number of times a clock is paused for a given simu-

lation time and the average latency incurred due to such clock pauses. Another important system analysis metric for efficiency comparison is the throughput of the system, i.e., the average production/processing capacity of a system.

Owing to increasing clock frequencies and smaller device sizes, it is becoming particularly important to consider the total power consumption metric in deciding on a particular design methodology. GALS based architectures reduce power consumption due to the ability to shift to an asynchronous mode when the local clock of the synchronous system is paused. Hence, a comparison of energy consumption in different GALS architectures would help choose between the different asynchronous communication circuitry. Therefore, an analysis of these metrics is useful for the designers to estimate the performance penalties in using one clocking scheme over the other.

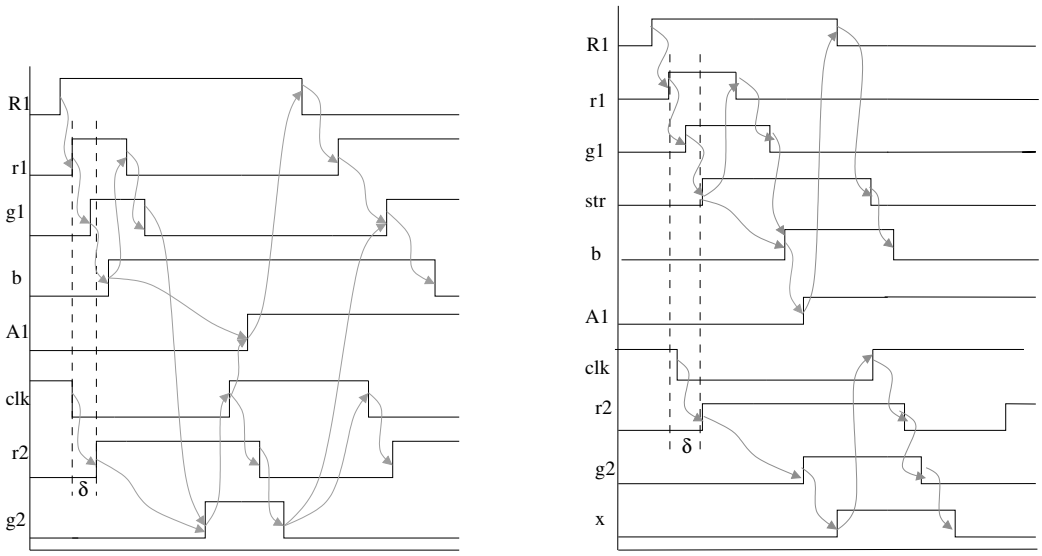
The following sub sections will present the model-level and circuit level analysis performed on the three GALS architectures.

4.5.2 Model Level Analysis

We present here the analysis of the best and worst case delay between the pausible and stretchable clocking schemes. These delays can help us estimate the usefulness of using one scheme over another. We take into consideration the latency between sending a request $R1$ from the FIFO to the consumer module and receiving an acknowledge $A1$ at the FIFO input from the consumer module, to be sent to the producer module, denoting a complete transfer of an item of data sent by the producer. Since the delay of the logic circuit (i.e. the logic gates)

for asynchronous data transfer and clock generation is comparable and can vary with different implementations of the same logic, we mainly take into account the number of clock cycles needed to obtain the desired output. Here, we assume that signals $r1+$ and $r2+$ for both pausible and stretchable clocks arrive with a time delay of δ between them, such that δ is greater than the time under which metastability may occur within the mutual exclusion element. This avoids the possibility of the resolution leading to a random selection of outputs from the element. In Figure 4.12 and 4.13, we present the timing diagrams for the best case and the worst case delay, respectively, scenarios for both the clocking schemes. For the best case delay, we assume that $r1+$ for both the clocking schemes arrives with time delay δ unit of time before signal $r2+$. As shown in Figure 4.12(a) , signal $A1+$ occurs after at least one clock cycle, following $R1+$, for pausible clocking scheme. In stretchable clocking scheme, shown in Figure 4.12(b), signal $A1+$ occurs in less than half a clock cycle. Hence, the best case delay analysis showed that stretchable clock architectures demonstrated faster data transfer compared to pausible scheme.

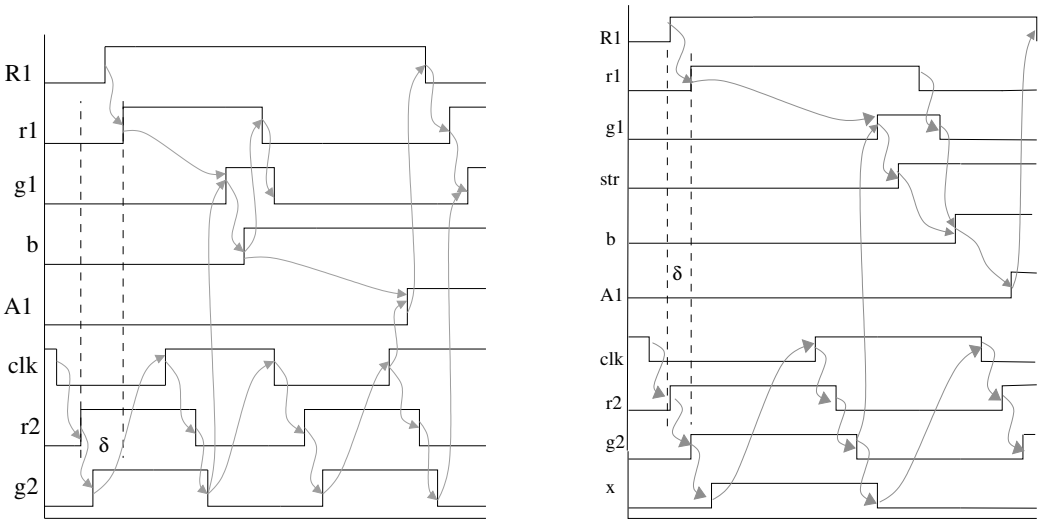
Such an observation is due to the fact that in pausible clocking scheme, the final set of latches, shown in Figure 4.8 waits for a positive clock edge before sending the signal to the next clock domain. It is easy to observe that the arrival of signal b misses the first clock edge and has to wait for the next clock edge to appear. The latch is enabled by a signal sent from the producer module which indicates when it is ready to receive new item of data. In stretchable clocking scheme, as shown in Figure 4.9, the latches are triggered when clock goes low.



(a) Pausible clock (b) Stretchable clock

Figure 4.12: Best case req-ack latency in producer block

This latch also waits for the enable signal sent by the consumer module, similar to the enable signal used in pausible clocking scheme, when it is ready to receive new data.



(a) Pausible clock (b) Stretchable clock

Figure 4.13: Worst case req-ack latency in producer block

Similarly, in the worst case scenario for the pausible clock (shown in Figure 4.13(a)) and stretchable clock (shown in Figure 4.13(b)), the signal $r2+$ arrives with time delay δ before $r1+$ for both the schemes. The delay, between the reception of request $R1$ and the emission of acknowledge $A1$, is over one and a half clock cycles for pausible clock. For stretchable clocking scheme, the delay is just over a clock cycle. Therefore, it is observed that we are able to save half a clock cycle on every data transfer for stretchable clocking scheme.

4.6 Circuit Level: Experimental Results

This section presents the results of power and performance analysis of GALS architecture with the three clocking schemes.

In the experimental setup, a 2 stage FIFO inter-module communication scheme has been used. In the experiments an input parameter, namely, the producer clock frequency, has been varied. It is varied from 125 MHz to 1.75 GHz to observe the behaviour. The frequency of the consumer clock is maintained at 500 MHz. Higher frequencies are possible depending upon the complexity of the producer and consumer blocks. The frequency of the clock has been varied by varying the delay d, in the three clocking schemes. This delay extends the clock period, thus changing the frequency of the clock. The ratio between the producer clock and consumer clock is called clock ratio. The clock ratio has been varied from 0.25 to 3.5 in steps of 0.5. This allows us to study the different phase relationship between the consumer and producer clocks.

Figure 4.14 and 4.15 show the number of clock pauses in the producer for pausible and stretchable clocking schemes, respectively, as the clock ratio is increased. We see that as the frequency of the producer clock increases, the number of pauses increases. The asynchronous data transfer logic operates at a particular frequency. This frequency depends on the rate of production of signal R from the producer block and rate of reception of signal A from the consumer block. The transfer frequency becomes smaller than the frequency of the producer clock as the producer clock frequency increases and becomes higher than the consumer clock frequency. Hence, it takes longer to finish the cycle that de-asserts the grant on the arbiter. Due to this we observe more clock pauses as the period of the clock is too small to mask this delay. At lower frequencies, the time period is large enough to mask the pause during its lower half of the clock cycle.

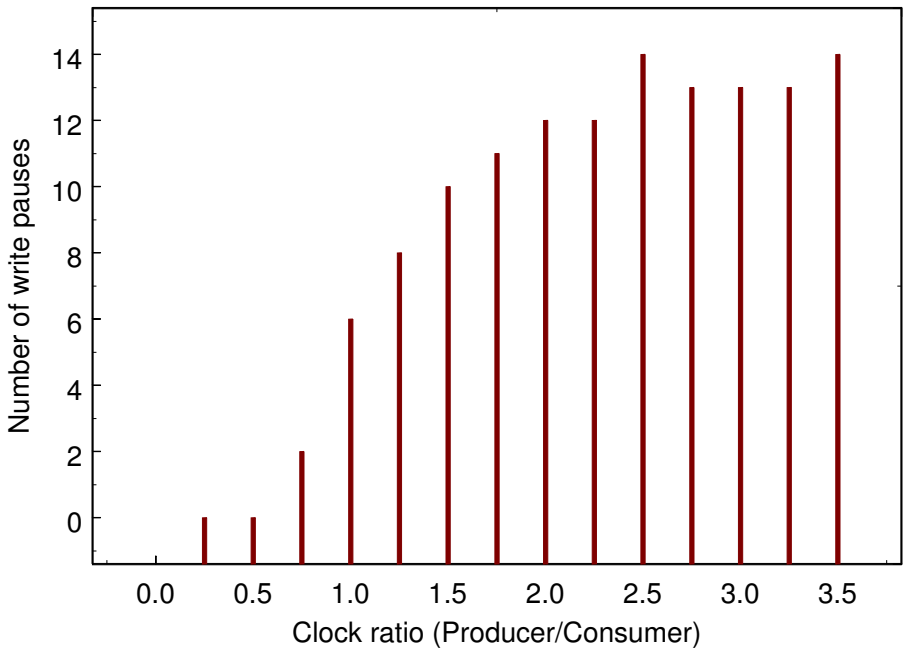


Figure 4.14: Number of pauses for pausible clocking scheme

The number of clock pauses in pausible and stretchable clocking scheme are comparable due to the scenario described above. But it can be observed from the graphs depicting total time incurred by these latencies, shown in Figures 4.16 and 4.17, that they are no longer comparable. The stretchable clocking scheme incurs longer latencies than pausible clock. This is because the clock is only asserted when signal str is low.

The arrival of signal A on the producer side or signal $R1$ on the consumer

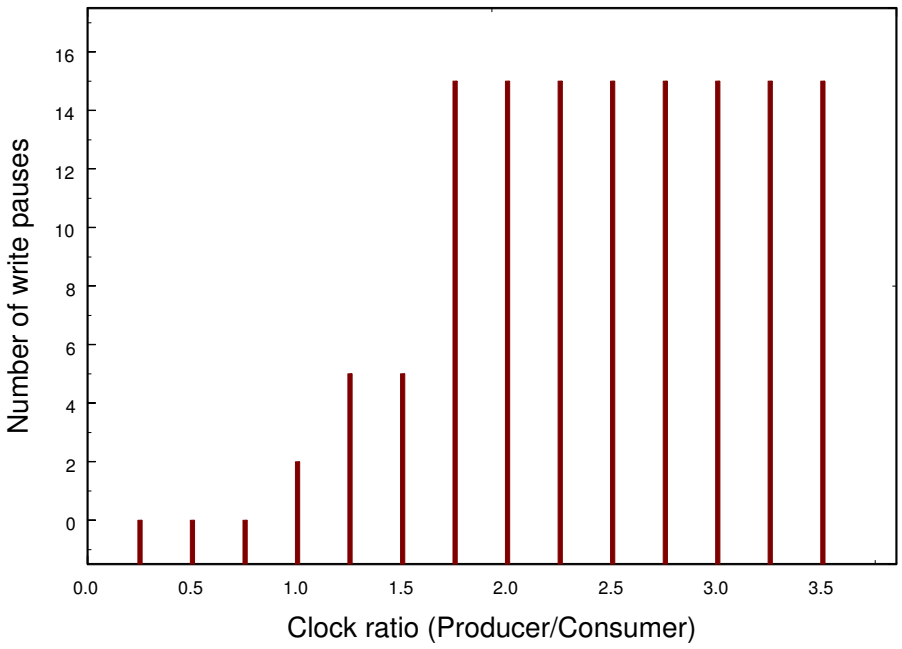


Figure 4.15: Number of clock pauses for stretchable clock

side, asserts signal str. When the producer frequency increases and becomes more than the consumer frequency, the FIFO gets filled up as more requests are produced than can be consumed by the consumer module. Hence, the de-assertion of signal A is delayed. This phenomenon is exemplified in Figure 4.18. The FIFO is made up of a set of C-elements [81]. The bold lines depict the signals that are asserted, while the non-shaded lines depict de-asserted signals. It can be observed that when the FIFO is full $Write\,Ack$ (which denotes signal A in Figure 4.9)

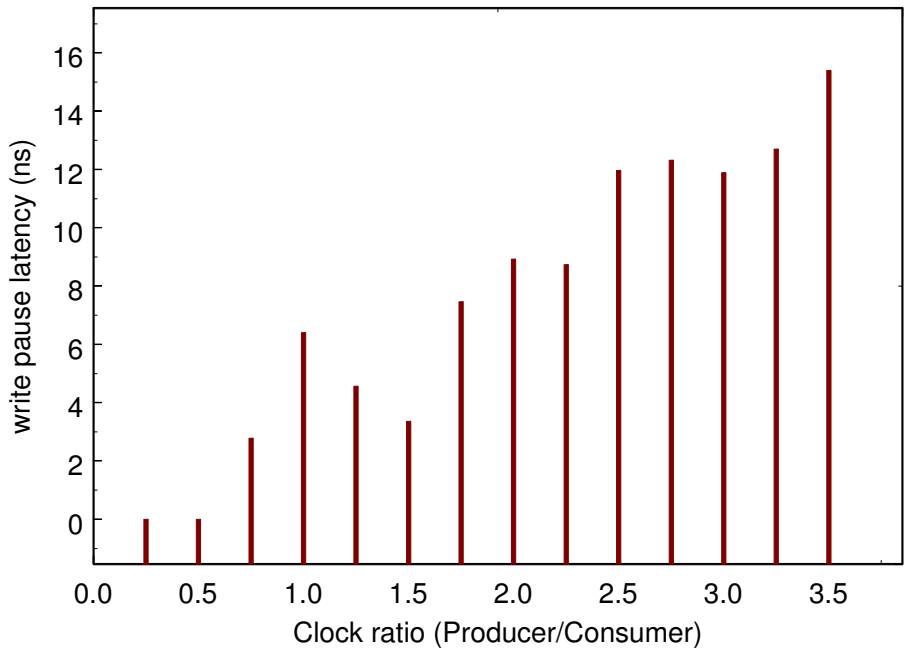


Figure 4.16: Pause latency in pausible clock

remains asserted and is only de-asserted when an item of data is read from the FIFO, i.e. $Read\,Ack$ (which denotes signal $A1$ in Figure 4.9) is asserted. The delay in the de-assertion of A, delays the de-assertion of signal str, which in turn delays the assertion of signal clk. This leads to a prolonged clock stretch. Such an occurrence is not observed in pausible clocking scheme.

This is because the reception of $b+$ immediately releases the grant on the arbiter and at this stage, the clock can arbitrarily win the grant to assert itself.

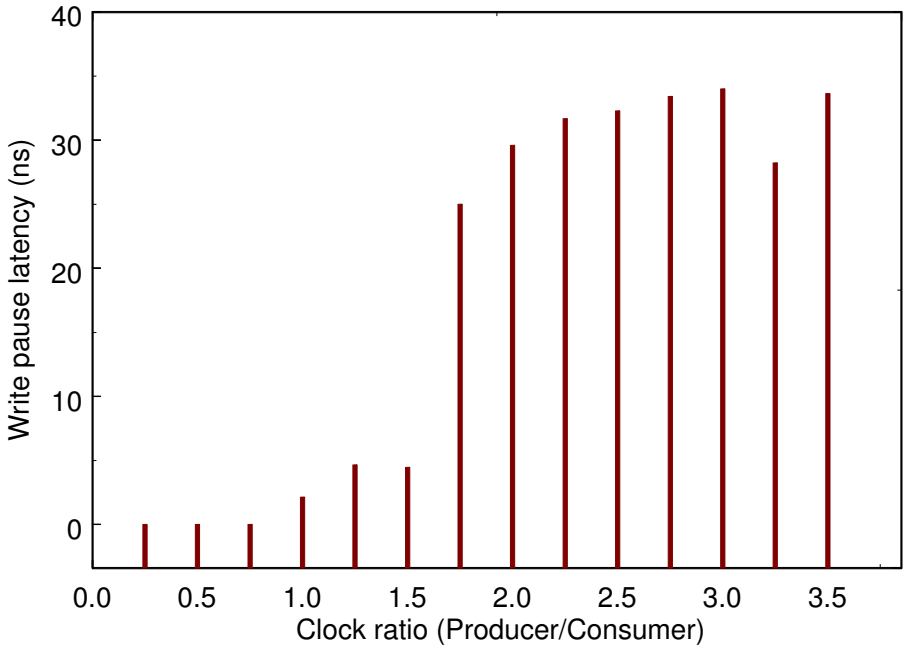


Figure 4.17: Pause latency in stretchable clock

This justifies the graphs shown in Figure 4.16 and 4.17.

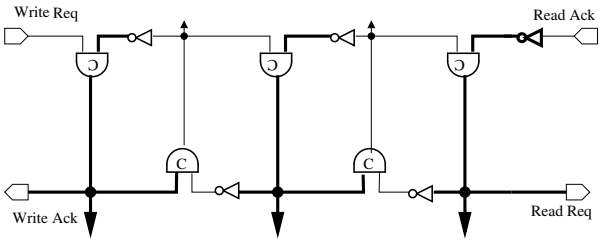


Figure 4.18: FIFO design

Figure 4.19 shows the average dynamic power consumption, at an operating voltage of $3.3V$, for transferring a burst size of 1 over a two stage FIFO implementation, with varying clock ratios. The power analysis includes the following blocks: input and output ports of the wrapper, clock control circuit and the FIFO structure. It is observed that as the clock ratio increases power consumption increases. This is because, as clock ratio increases, the throughput and operating frequencies of the synchronous islands increases, leading to an increased power consumption.

It can be seen that the pausible and stretchable clocking schemes consume more power than the data driven clocking scheme because of their complex asynchronous circuitry. Since the implementation of the FIFO is same for all the protocols, complexity of port controller implementation of pausible and stretchable clocking schemes accounts for such observations. It must also be noted that the absolute power value for all the three clocking schemes include the current drawn by the local ring oscillator clock (the block $'d'$ in Figure 4.8, 4.9 and 4.10).

Figure 4.20 shows the impact of changing clock ratio on the throughput of the communication channel. It can be observed that as the frequency of the consumer clock increases the throughput increases linearly up to clock ratio 1. This is because more data is being read by the consumer in the same period of time. After this time, the throughput reaches a saturation point. This is because the consumer clock operates at a lower clock frequency compared to the producer clock. Hence, there is no additional increase in throughput.

The throughput values obtained for stretchable and data driven clock are higher

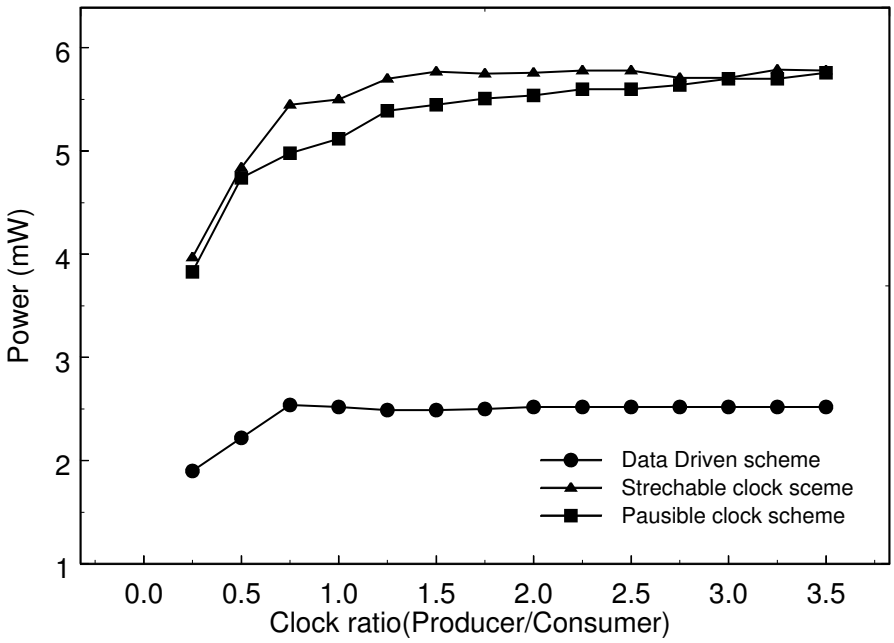


Figure 4.19: Power analysis

than pausible clock. This is due to the delay between two consecutive rising edges of the request signal ($R+$). Detailed phase relationships between signals that cause this delay is shown in Figure 4.21 and 4.22. It is observed that this delay is $12ns$ for pausible clock and $8ns$ for stretchable clocking scheme.

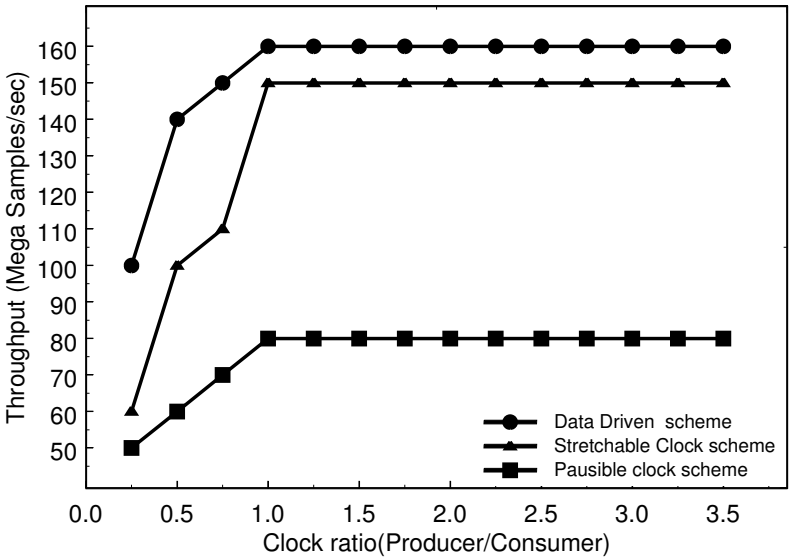


Figure 4.20: Throughput analysis

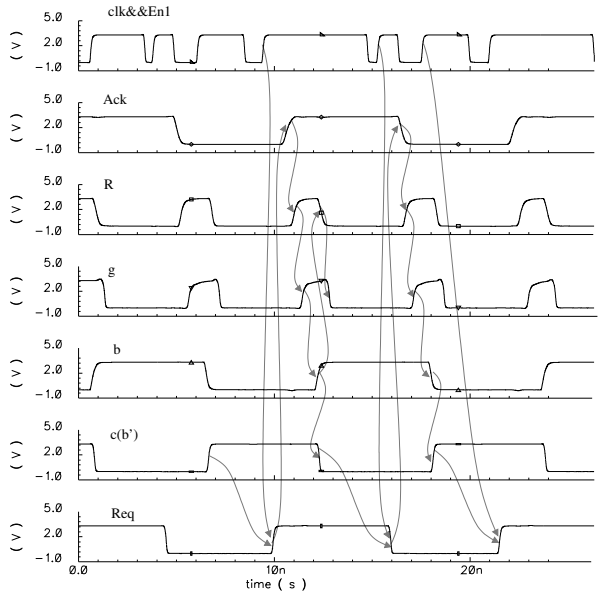


Figure 4.21: Request delay analysis for pausible clock

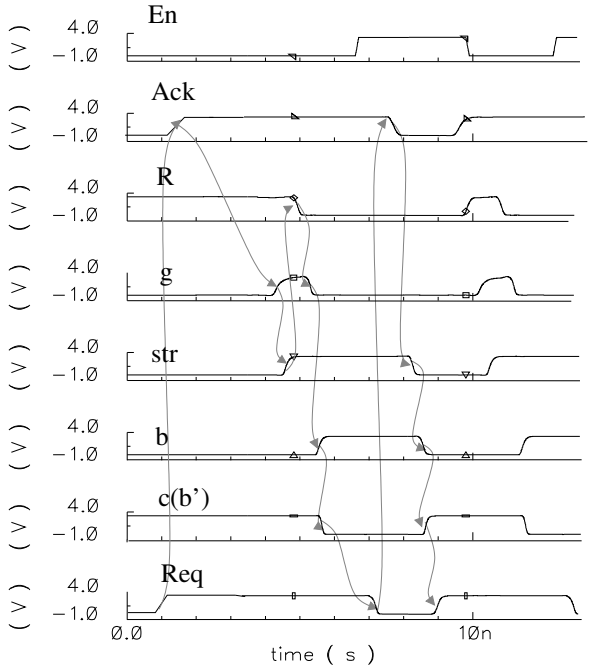


Figure 4.22: Request delay analysis for stretchable clock

The throughput is maximum for data driven clock. It is higher than the stretchable scheme since the signal A in the stretchable clocking scheme waits for synchronisation for crossing over to synchronous domain to produce $Sync\_ack$. On the contrary, no such synchronisation is needed for data driven clock as the clock starts when there is data to transfer and hence the signal A thus produced is already synchronised to the clock. This explains the trend of the curves in the graph that depicts the throughput of the different clocking schemes.

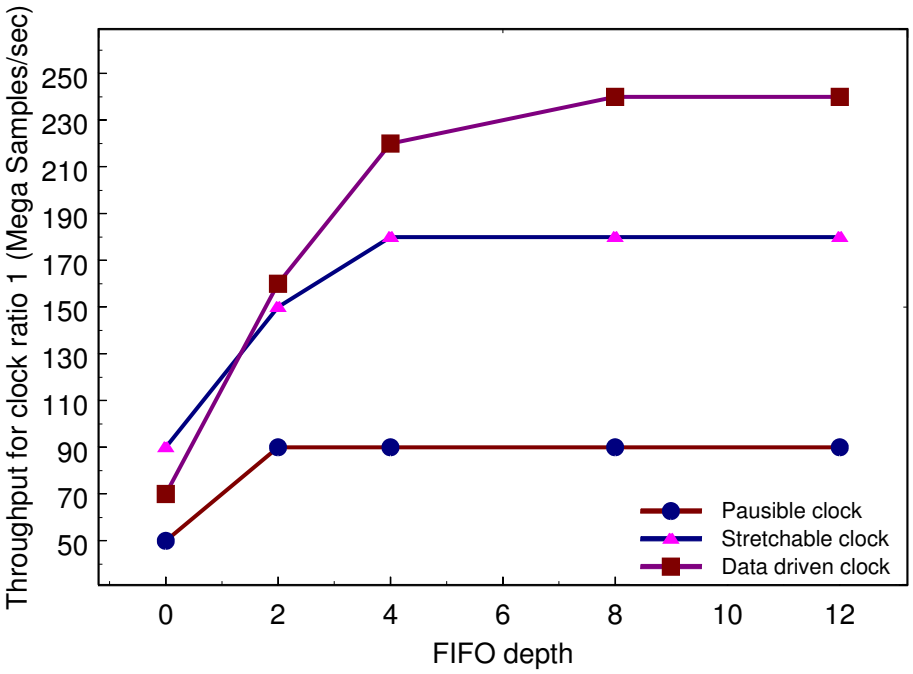


Figure 4.23: Throughput Analysis with varying FIFO sizes

The control schemes are tested with varying FIFO depths. The FIFO depth is varied from 0 to 12. It is observed that for pausible clock the performance does not improve by making the depth more than 2 slots. For stretchable clock, throughput increases till 4 slots, after which there is no improvement in performance. Data driven clocking scheme shows increase in performance up to 8 slots and then does not improve any further. These observations are depicted in Figure 4.23. The readings for the different architectures with varying FIFO depths are taken at

clock ratio 1 for comparison.

Table 4.2 presents the maximum throughput (in percentage) obtained for the three clocking schemes over a 2-stage FIFO at clock ratio 1. It also presents the power (in percentage) consumed by the same architectures at the same clock ratio.

Schemes	Pausible clock(%)	Stretchable clock(%)	data driven clock(%)
Throughput	50	93	100
Power	218	203	100

Table 4.2: Comparative Throughput and Power analysis results

It can be seen from Figure 4.24 that on removal of the FIFO and by connecting the producer and consumer blocks directly, the system demonstrates low performance and the throughput saturates at 50 Mega Samples/sec. It is also observed that this saturation is reached earlier in time for stretchable and data driven clocking schemes. The rise occurs below the clock ratio of 0.25. The region of throughput rise for the two clocking schemes is also depicted in Figure 4.24.

It can also be observed that in the absence of FIFO structures, performance of stretchable clock is better than the performance of the data driven clock. Requests are generated every $10ns$ for stretchable clock, in contrary to data driven clocking scheme, where requests are generated every $13ns$ at clock ratio 1.

4.7 Summary

This chapter presented the classification of different clocking schemes for Globally Asynchronous and Locally Synchronous architectures. Petri nets were used

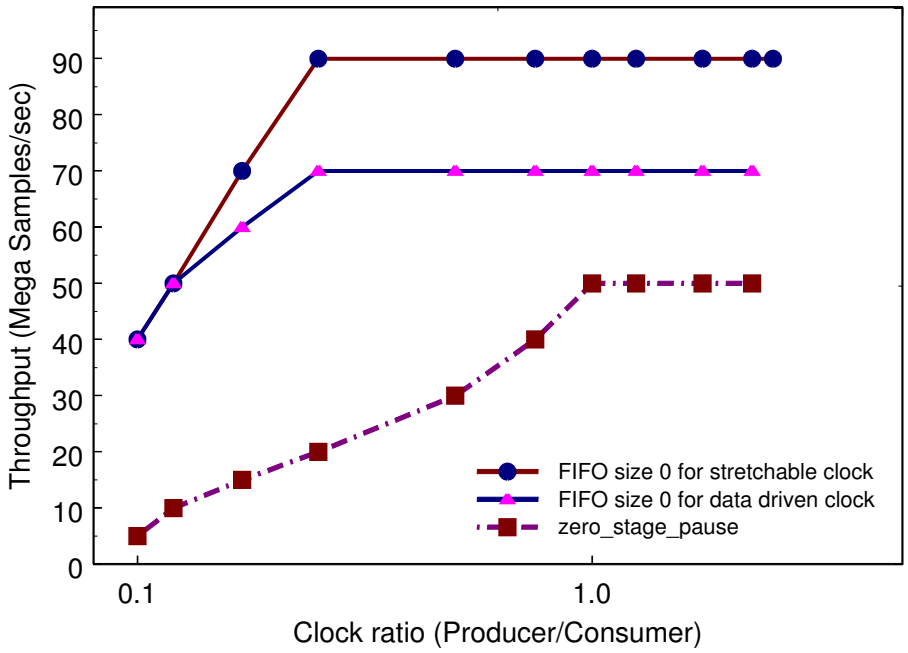


Figure 4.24: Throughput with FIFO size 0

to model the different schemes. It also presented an analysis of the three systems on performance and power consumption criteria. Data driven approach guarantees faster operation if making further progress in computation requires a constant stream of data and the computation can be completed in accurately timed clock periods. This is because if there is a constant flow of data in pausible and stretchable clocking schemes, there is a probability that the clock will be unnecessarily interrupted at every instant of data transfer without completing any useful compu-

tation, resulting in its delayed completion. Data driven clocking scheme requires matched data and hence constant stream of data is ideally suited for this type of scheme. If data is sent in bursts and its progress cannot be done in a number of specified timed periods then pausible and stretchable clocks are more viable options. In data driven clocking scheme, the local clock is not free running and hence, extra circuitry needs to be introduced that can dynamically count the number of clock cycles needed for each computation, which would further increase the complexity of the system. An increase in system complexity would give rise to increased dynamic power consumption. Pausible and stretchable clocking schemes have free running clock which is only stopped in the region of possible occurrence of metastability and hence no such circuitry is required. When a required burst of data arrives, the clock is paused to transfer the entire burst. This also reduces the number of clock pauses.

Stretchable clocking scheme demonstrated better performance compared to pausible clock in terms of throughput. But, it also exhibited longer clock pause time in the producer block compared to pausible clocking scheme, at higher clock ratios (Producer frequency/Consumer frequency). This results in increased computation time in the synchronous island at higher clock ratios. Therefore, if it is possible to operate the system at a lower clock ratio, then stretchable clock is more viable than pausible clocking scheme. For higher clock ratios pausible clock is more advantageous. Therefore, based on the requirements of the system, stretchable clocking scheme could be more desirable over pausible clocking scheme, or vice versa.

Chapter 5

GALS Implementation of Weakly Endochronous Systems

5.1 Introduction

Previous chapters have already presented the advantages of GALS architectures. Therefore, it seems practical to move towards a paradigm that amalgamates the advantages of both synchronous and asynchronous designs for implementing complex system on chip to form Globally Asynchronous and Locally Synchronous (GALS) systems. Thus, the system can exploit the existing synchronous tools for the design of the IP blocks with local clocks while the wrappers and communication channels between the different modules are handled by asynchronous methodologies. This chapter addresses the problem of correctly and efficiently implementing a modular synchronous specification as a GALS architecture where

each locally synchronous module communicates with other modules via asynchronous communication lines.

The exact problem that is considered in this chapter is that of synthesising the asynchronous wrappers starting from the specification of the synchronous modules. The approach is based on weakly endochronous synchronous model, which defines high level implementation conditions guaranteeing correct desynchronisation at the level of the abstract synchronous module. The synthesis problem involves the weakly endochronous module construction phase and the actual wrapper synthesis phase. The focus is on the synthesis of delay insensitive asynchronous wrappers from weakly endochronous modules. The choice of delay insensitive logic as implementation domain is determined by its excellent modularity properties, its ability to support concurrency and by the existence of state of the art tools allowing the specification and synthesis of delay insensitive circuits.

The main contribution is the introduction of the synthesis methodology, involving synchronous and asynchronous formalisms, for implementing correct by construction GALS models, while ensuring the properties of Weakly Endochronous (WE) systems (introduced in Chapter 3). We recall here, the basic idea behind Weak Endochronous Systems. WE is a speed-independence property which characterises synchronous components whose behaviour does not depend on the order in which various inputs are read. Hence, this property enables the application of asynchrony at the synchronous system interfaces. The properties of a WE systems is presented in Subsection 5.4.1.

5.1.1 Overview of the methodology

The main contribution of the work is to translate the WE components of a synchronous system into delay insensitive circuits, while preserving the criteria imposed by WE systems for correct GALS deployment.

This approach is primarily based on asynchronous handshake protocols. As shown by the shaded block in Figure 5.1, the distributed synchronous system is encapsulated by an asynchronous wrapper. This asynchronous wrapper consists of communication channels and a clock generator. The communication channels consists of a set of input and output FIFOs, shown in Figure 5.1. We consider that each signal is transmitted from one synchronous island to the other in a FIFO dedicated to itself. Therefore, we have as many FIFOs as there are signals in the system. When data is available at the input FIFOs, they are read by the synchronous module and the clock generator triggers the local clock for computation. On completion of the computation, the clock is stopped and the outputs are written on the output FIFOs. The release of the the clock after the completion of the computation allows the synchronous module to read the next set of inputs from the input FIFOs.

Similar to the *Data Driven Clocking scheme* presented in Chapter 4, the clock is triggered when the data required for a particular computation is read and is waiting for some operation to be done on it. The clock is released after the completion of the computation. This leads to a signification reduction in power consumption. Unlike the prevalent clock gating schemes, the synchronous module is not unnecessarily stalled by the unavailability of an input not required for a particular

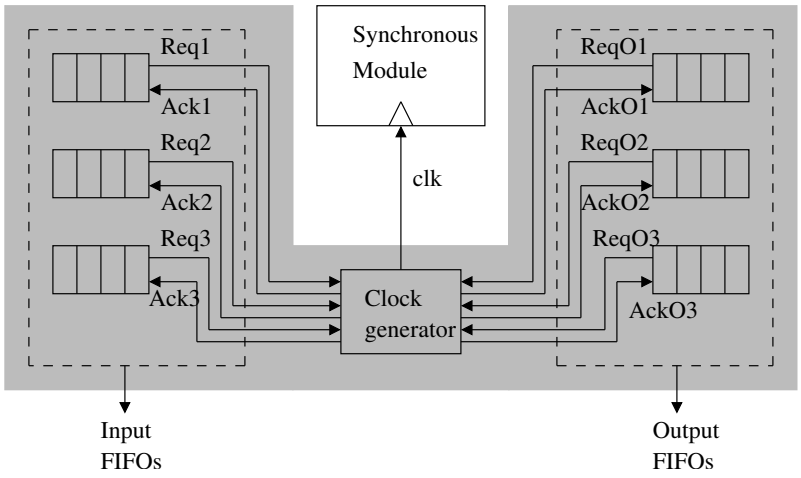


Figure 5.1: Delay Insensitive System

computation. This leads to an increased efficiency.

The WE models of the individual system blocks exhibit synchronous behaviour. Therefore, it is required to transform these models before they can be brought to an implementable level.This involves:

1. Refining the model to handle handshake operations suitable for asynchronous communication.

2. Removal of the global clock and retention of local clocks, where the clock signals are used for computation.

3. Fragmenting the individual synchronous modules into smaller segments to introduce relevant local clock control signals.

Since each synchronous module is weakly endochronous, the removal of the global clock preserves the I/O behaviour and correctness. The final step involves addi-

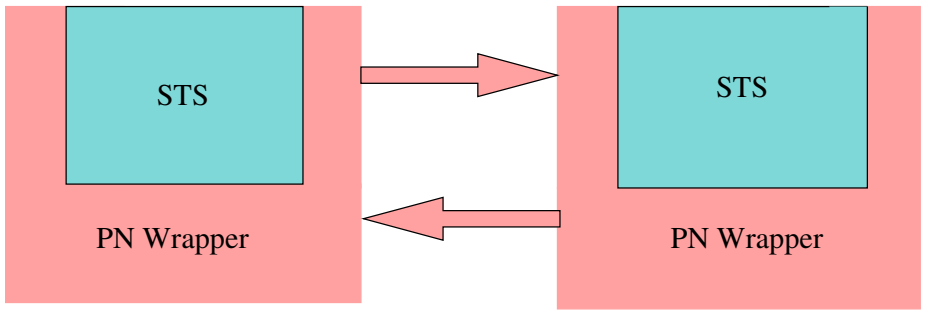


Figure 5.2: GALS wrappers constructed around the synchronous WE blocks

tion of extra signalisation and signal reordering, ensuring that the WE criteria is satisfied at every stage of refinement. The top view of the model with the asynchronous wrapper is depicted in Figure 5.2. The model of the wrapper obtained can then be fed to existing synthesis tools to obtain a gate level implementation.

Therefore, the wrapper, in general, has the following two functions:

- reconstruct, for each synchronous module, the input synchronisation points, which in turn would control the clock
- preserve the semantics of the synchronous specification in GALS implementation.

5.2 Preliminaries for modeling

Here we introduce the models of concurrency which allows to capture the essential features of synchronous and asynchronous systems. The underlying model of computation, *state transition systems (STS),* has already been introduced in Chap-

ter 2. This section presents a special class of STSs, known as *Microstep Transition Systems,* suitable for the specification of synchronous system to be desynchronised for GALS deployment [69].

5.2.1 Microstep Transition System model

To represent a synchronous system, finite state machines are used having exactly one clock variable, consisting of only clock and directed variables and satisfying a number of axioms. Such a system is called a *Microstep transition system.* The weakly endochronous systems are modeled using *Microstep transition systems.*

Definition 5.1. *Microstep:* the tuple $\Sigma = (S, s_0, V, \tau, T)$ is a microstep synchronous transition system (STS), if $(S, s_0, V \cup \tau, T)$ is a concurrent transition system, where all the variables of V are directed, where τ is a clock variable, where

Axiom 3. *clock transitions: if* $s \overset{<\tau>}{\rightarrow} s'$ *and* $l(\tau) = \perp$*, then* $l|_V = \perp_V$

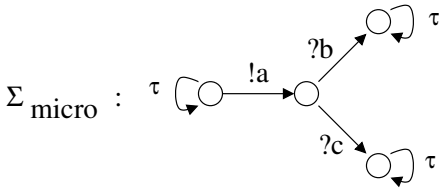
The above axiom identifies the clock symbol with label $< \tau >$ which are the only transitions where the clock transitions are present. These transitions separate synchronous reactions during which a variable can be assigned only once. Therefore, each reaction starts and ends with a clock transition(s).

Axiom 4. *Stuttering invariance:* $s_0 \overset{<\tau>}{\rightarrow} s_0$ and $s \overset{<\tau>}{\rightarrow} s' \Rightarrow s' \overset{<\tau>}{\rightarrow} s'$

For a pair of states, (s_1, s_2), if $s_1 = s_2$, then it is called a stuttering state. A process is a set of behaviours σ that is invariant under stuttering iff it contains every behaviour obtained from σ by adding/removing stuttering steps.

Axiom 5. *unique assignment: if* $s_0 \xrightarrow{l_1} s_2 \xrightarrow{l_2} ..... \xrightarrow{l_n} s_n$ *and* $\forall i : l_i \neq \tau$, *then* $l_1, l_2....l_n$ *are non-overlapping.*

two assignments of the same variable must be separated by a clock transition.

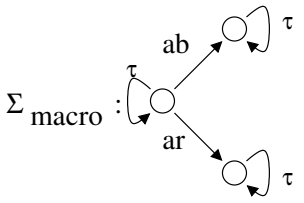


In a similar way, the classical macrostep transition system can be defined.

Definition 5.2. *Macrostep:* A tuple $\Sigma = (S, s_0, V, \tau, T)$ is a macrostep transition system, if,

$$s \xrightarrow{l} s' \leftrightarrow \exists \phi : \begin{cases} s \xrightarrow{\phi} s' \\ \phi = Step_0(\phi) < \tau > \\ l =< Step_0(\phi) > \end{cases}$$

The Macrostep representation of the Σ_{micro}, is denoted by Σ_{macro}.



This compact form hides both the I/O and computational causality which are essential features of asynchronous implementation. Therefore, the rest of the chapter deals with microstep transition systems to handle the transition from synchronous to GALS architecture.

5.2.2 Theory of Regions

In the previous section, the microstep transition system has been proved to be equivalent to reachability graphs. This system model is required to be translated to PN for synthesis. This is done by using Theory of Regions [82]. Some of the important notations are recalled in this chapter. Subsets of states in a *Transition System (TS)* or a RG denoted by $\{S, A, l, s_0\}$, that correspond to a set of places in a *Petri net* denoted by $PN = \{P, T, F, M_0\}$ are called *Regions*. Let S_1 be a subset of states S of a *TS*. A transition $s_1 \rightarrow s_2$ enters S_1 if $s_1 \notin S_1$ and $s_2 \in S_1$. Transition $s_1 \rightarrow s_2$ exits S_1 if $s_1 \in S_1$ and $s_2 \notin S_1$. If neither of the conditions hold true then the transition does not cross the region. If both the conditions hold true then the transition is internal to the region. A subset R is a region if ,

$\forall t \in A$ where $l(t) = a$(where a is a lable of transition t), one of the conditions hold true:

1. enter R
2. exit R, or
3. does not cross R.

If R_1 and R_2 are regions of a TS, such that $R_2 \subset R_1$, then R_2 is a subregion of R_1. Region R_2 is a *minimal* region if it contains no *sub-regions* of the TS. A region R is a pre-region of an event a if transition labelled a exits R. A region R is a *post-region* of an event a if the transition labelled a enters R. Figure 5.3 (a) shows an elementary transition system. This TS is divided into regions as shown

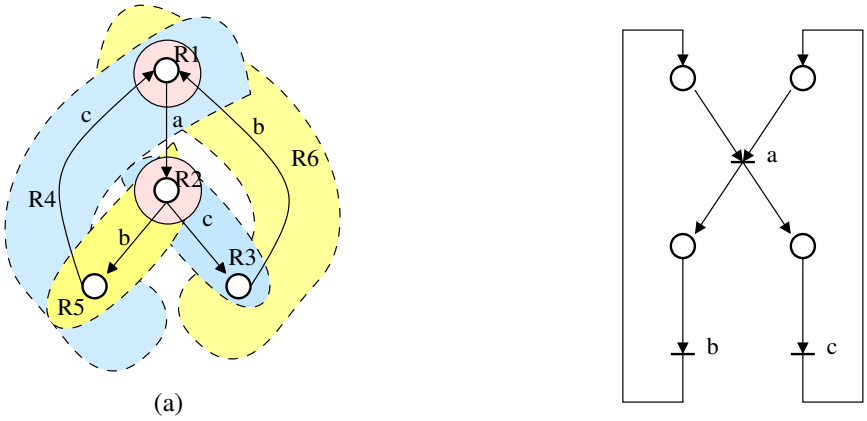


Figure 5.3: Translation of a TS into a PN

in the figure. Figure 5.3 (b) shows the PN that is obtained by mapping the regions onto places as depicted in [83].

5.3 Composition-GALS idea

Modular synchronous systems and GALS implementations are built from microstep synchronous automata by using two types of compositions, namely synchronous and asynchronous, as presented in [69]. Both of these approaches are based on point-to-point communication scheme through FIFOs as described in [84].

Definition 5.3. Composable Transition System

I/O transition systems $\Sigma_i,\ i = 1, 2, ..n$ are composable if their variable sets are mutually disjoint.

The above definition has the following two requirements:

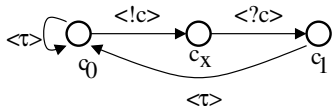
1. Point-to-point communication (no directed variable is shared by two or more systems. But, broadcast can be simulated by replicating and renaming the variables.)

2. Non-overlapping clock sets.

5.3.0.1 *Synchronous Composition:*

To represent synchronous communication, 1-place synchronous FIFO models are used which are microstep synchronous transition systems themselves. The FIFO model associated with channel c is illustrated below.

$$SFIFO(c,\tau) \;=\; (\{c_0, c_1\} \cup \bigotimes_{x \in D_c} \{c_x\}, c_0, \{\tau\} \cup \bigotimes_{x \in D_c} \{!c = x, ?c = x\}, \rightarrow_S)$$



Synchronous parallel composition of $STSs\ \Sigma_i = (S_i, s_{0_i}, V_i, \tau_i, T_i), i = 1, 2$, denoted by $\Sigma_1 | \Sigma_2$, is given by:

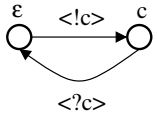
$$\Sigma_1[\tau_1/\tau] \otimes \Sigma_2[\tau_2/\tau] \otimes \bigotimes_{c \in C(V_1) \cap C(V_2)} SFIFO(c,\tau)$$

5.3.0.2 *Asynchronous Composition:*

Asynchronous communication is represented by an infinite asynchronous FIFO whose transition relation is given by:

$$AFIFO(c) \;=\; (D_c^*, \epsilon, \bigotimes_{x \in D_c} \{!c = x, ?c = x\}, \rightarrow_A\}$$

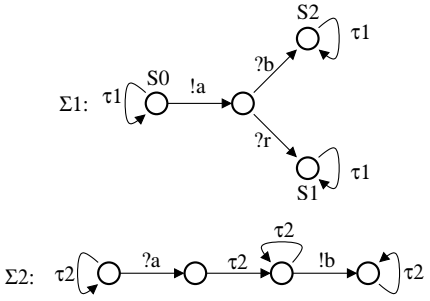
A more specific case of extending the infinite stage FIFO to a $1-$place FIFO is denoted by:



Asynchronous composition of $\Sigma_i = (S_i, s_{0_i}, V_i, \tau_i, T_i), i = 1, 2$, denoted by $\Sigma_1 || \Sigma_2$, is given by:

$$\Sigma_1 \otimes \Sigma_2 \otimes \bigotimes_{c \in C(V_1) \cap C(V_2)} AFIFO(c)$$

Below we show synchronous and asynchronous compositions of two STSs, Σ_1 and Σ_2



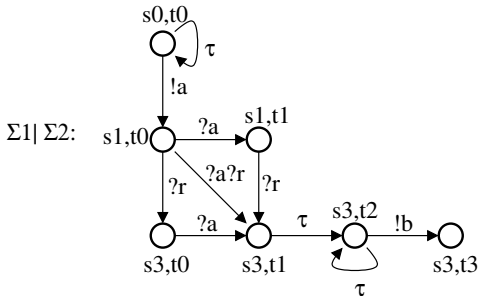
The two requirements that need to be satisfied while composing systems are, which arise from the definition of the FIFOs:

1. A signal cannot be received before it is emitted.

2. For synchronous composition, the system cannot take a clock transition after a signal is emitted and before it is read.

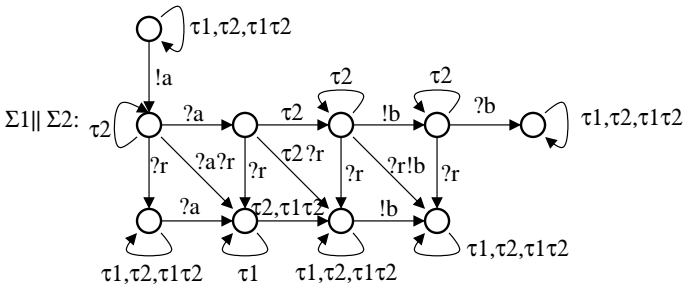
Based on these two requirements we exemplify the two compositions.

The synchronous composition is given by:



The representation is simplified by not showing the state of the two FIFOs $SFIFO(a, \tau)$ and $SFIFO(b, \tau)$. It can be seen that the composed system is blocked in state (s_3, t_3) because $SFIFO(b, \tau)$ cannot take a clock transition since data has been written but not read. Therefore, the system $\Sigma_1 | \Sigma_2$ can deadlock.

The asynchronous composition is given by:



It is to be noted that $\Sigma_1 || \Sigma_2$ has traces like $!a;\ ?a;\ \tau_2;\ !b;\ ?b$, that are not asynchronously equivalent to any of the synchronous traces of $\Sigma_1 | \Sigma_2$. In such a case, the GALS implementation does not preserve the semantics of the specification. Therefore, a good correctness criterion for desynchronisation is the preservation of asynchronous traces. This criterion is computationally infeasible even for finite systems. Therefore, the next section gives sufficient conditions which are decidable.

5.4 Weak endochrony

In this section weak endochrony is discussed, that was introduced in [69], to present the construction of modular synchronous architecture. To do so, endochronous systems [57] are first defined which are extended to form weakly endochronous systems. Endochronous systems are characterised by a condition that the presence and absence of all variables can be inferred incrementally during each reaction from already known values of the present input variables and state variables of the Synchronous Transition System (STS) under consideration. The main advantages of WE systems are that it can take into account different execution modes and independence between components in order to minimize communication and allows multi-rate computations, unlike latency insensitive systems [54]. But due to poor handling of concurrency, endochronous systems are not compositional. This leads to inefficient synthesis of systems formed of more than two components.

The above disadvantages lead to the introduction of weakly endochronous systems. These systems take into account the issues with the previous approaches and presents an approach that efficiently handles internal concurrency and both the properties together to form a correct desynchronisation criterion that is decidable on finite synchronous systems. They support signalling schemes that are simpler and more efficient than latency insensitive and endo/isochronous counterparts.

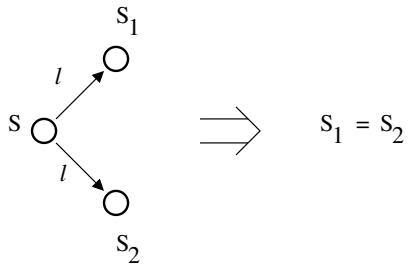
5.4.1 Weakly Endochronous Criteria

Weak endochrony generalizes over latency-insensitivity and endochrony, being able to represent concurrency between different operations of a synchronous component. Thus, it potentially supports lighter communication protocols than the existing approaches. Both latency insensitive systems and endochronous systems satisfy the axioms of weak endochrony.

Definition 5.4. A system $LSTS\ \Sigma = (U, S, \rightarrow, s')$ is weakly endochronous if the following properties are satisfied for all $s, s_1, s_2 \in RSS(\Sigma)$, and for all $r, r_1, r_2 \in Reactions(U)$:

Criterion 5.1. *Determinism* - $s \xrightarrow{l} s_i$, $i = 1.2 \Rightarrow s_1 = s_2$

Example 5.1. If s is a state and ϕ is a trace, then $s.\phi$ is a unique state reached from s with ϕ.



Criterion 5.2. *Independence - If the labels l_1 and l_2 are disjoint and if $l_1, l_2 \neq \tau$ and if $s \xrightarrow{l_1} s_1$ and $s \xrightarrow{l_2} s_2$ then*

$$\exists \quad s', such\, that \quad s \xrightarrow{l_1 \cup l_2} s'$$

Criterion 5.3. *Clock Properties - assuming that $s_0 \xrightarrow{<\tau>} s_1$ and $\phi \in Traces_\Sigma(s_0)$ with $\tau \notin supp(\phi)$,*

1. $\phi \in Traces_\Sigma(s_1)$

This criterion states that if it is possible to either perform a clock operation or carry out a directed variable communication, then it is also possible to first clock and then do the communication. Therefore, in one clock cycle it is possible to have varying number of communications (signal transitions) without affecting the communication behaviour of the circuit. Therefore, if it is either possible to perform τ or $?a; ?b$, then $\tau; ?a; ?b$ is also a permissible trace.

2. if $\phi < \tau > \in Traces_\Sigma(s_0)$, then $\phi < \tau > \in Traces_\Sigma(s_1)$ and $s_0.\phi < \tau > = s_1.\phi < \tau >$

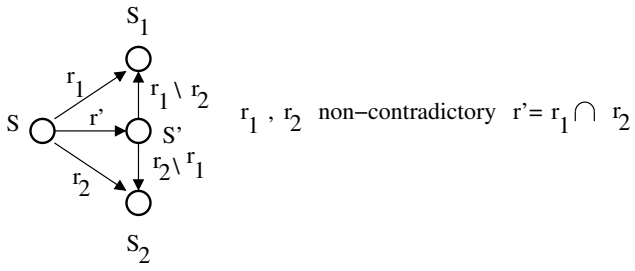
The above criterion states that in a circuit, the same state is reached by either doing $a;\tau$ or $\tau;a;\tau$. The same conclusion can be drawn as the previous property, that the communication behaviour does not change by increasing or decreasing the number of clock cycles.

3. if $\phi\psi < \tau > \in Traces_{\Sigma}(s_1)$, then there exists $\psi' \leq \psi$, such that $\phi\psi' < \tau > \in Traces_{\Sigma}(s_0)$.

It is already known from property 2, that $\tau;\phi;\psi;\tau \;\; \phi;\psi;\tau$. It is possible to do a prefix of the communication in one clock cycle because there exists a trace p, such that $\psi';p \;\; \psi$, i.e. it is still possible to complete the trace ψ after the clock tick. Using this property, it is possible to to divide the signal sequences into smaller fragments in the synchronous circuit.

4. if $\phi < \tau >, \theta < \tau > Traces_{\Sigma}(s_0)$ and $\phi \bowtie \theta$, then $\phi(\theta \backslash \phi) < \tau > \in Traces_{\Sigma}(s_0)$.

Let either of the communication traces ϕ or θ be permitted to execute, where $\phi \bowtie \theta$, i.e. prefixes of another trace t. The above property denotes that in such a situation it is possible to execute another non-contradictory prefix of t. This prefix say, $\phi;p$ is bigger than ϕ and θ. Therefore, in the synchronous system if it is possible to do two prefixes of a communication sequence, then it must also be possible to do a bigger prefix of this sequence.

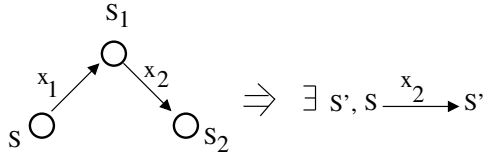


Criterion 5.4. *Choice: if* $\phi < v = x_i > \in Traces_\Sigma(s)$, $i = 1, 2$ *and* $\phi_1 \bowtie \phi_2$, *then* $\phi < v = x_2 > \in Traces_\Sigma(s)$

If it is possible to execute either ϕ_1 or ϕ_2 (prefixes of a trace t) each followed by the execution of x_1 and x_2, respectively, it is also possible to execute $\phi_1; x_2$ or $\phi_2; x_1$. Therefore, if it is possible to interrupt a trace t, by first doing communication sequence ϕ_1 followed by x_1 then it is also possible to do x_1 after a the execution of a non-contradictory trace ϕ_2. It is an important criterion for desynchronisation. This is because in the asynchronous circuit the order of $!a$ and $?a$ is guaranteed but the order of a and b are not guaranteed. Therefore, it is important that the circuit is able to do both x_1 and x_2, otherwise deadlocks are possible or circuit can reach different states.

Let t :$!a!b?a?b$ be a signal sequence. Two runs of t can be ϕ_1 :$!a?a!b?b$ and ϕ_2 :$!b?b!a?a$. Further let x_1 :$!c = 2$ and x_2 =$!c = 3$. The synchronous circuit must now allow the following runs: (1) $\phi_1; x_1$, (2) $\phi_2; x_2$, (3) $\phi_1; x_2$, (4) $\phi_2; x_1$.

Therefore, there is a choice of doing $x_1$or x_2.

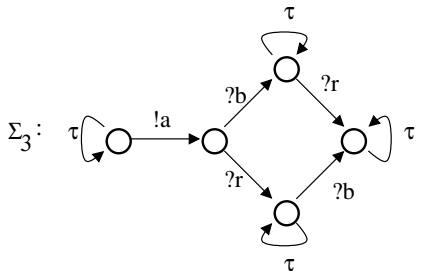


5.4.2 Correctness results

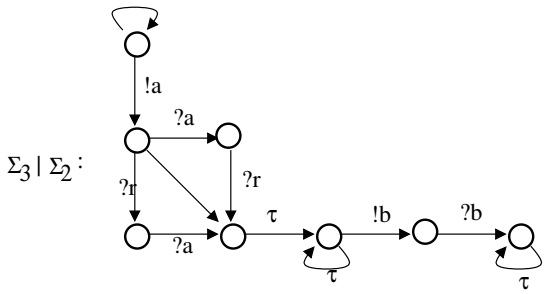
The following theorems, presented in [69], give the basis for the correctness and synthesis of GALS architecture from synchronous specification:

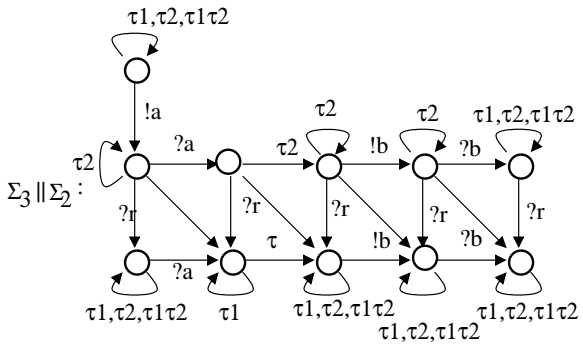
Theorem 5.5. *Let Σ_i, $i = 1, ..n$ be composable weakly endochronous $\mu STSs$, then, $|_{i=1}^{n}\ \Sigma_i$ is weakly endochronous.*

The previous section presented the weakly endochronous criteria. From the above criteria, it can be seen that Σ_2 is weakly endochronous and Σ_1 is not. There is a choice between reading b and reading r at the state s_1 which is not visible at the exterior. Therefore, if the environment provides both b and r, the input reading will be non-deterministic. On the other hand, if $?b$ and $?r$ are concurrent, then the system is weakly endochronous. This is depicted in system Σ_3.



Therefore, it can be seen that the $GALS$ implementation model $\Sigma_3 || \Sigma_2$ preserves the semantics of $\Sigma_3 | \Sigma_2$.





It can be seen that any trace of $\Sigma_3 || \Sigma_2$ is asynchronously equivalent to $\Sigma_3 | \Sigma_2$. Such a GALS implementation is correct because it does not introduce any new behaviour.

Theorem 5.6. *Let Σ_i, $i = 1, ..n$ be composable weakly endochronous $\mu STSs$, and if, $|_{i=1}^{n}\ \Sigma_i$is non-blocking, then $\|_{i=1}^{n}\Sigma_i$ is correct with respect to the synchronous specification.*

The theorem states that if the components of a deadlock-free synchronous specification are weakly endochronous, then the synthesis of the GALS wrappers can be done locally for each module, without knowledge about the global system.

The implementation can be derived by connecting the resulting modules with asynchronous FIFOs of arbitrary length. In the following section we define a model for the representation of asynchronous implementation of component-wise synchronous specification. The model removes the global clock and preserves global synchronization by means of signalling.

5.5 Synthesis of the weakly endochronous modules

The methodology for synthesis of the GALS implementation of synchronous systems requires the transformation of the microstep transition systems of the WE components to Petri nets. This is required in order to use existing synthesis tools to obtain asynchronous controllers and be able to use existing PN modules to extend the type of clock control schemes. There are existing methods for the translation of a State Transition Graph into a PN (whose Reachability graph (RG) is bisimilar to the transition system), based on the Theory of Regions, presented in Section 5.2.2.

This section presents the translation methodology proposed on the original system, to bring the specification of the WE system to an implementable level. We start by presenting some restrictions and assumptions on the models for synthesis.

5.5.1 Requirements of the specification for synthesis

Before we proceed to the synthesis algorithm we specify some requirements of the system for synthesis.

1. Weakly endochronous systems exhibit true concurrency (represented by the diagonal in the diamond as shown in Figure 5.4). We replace true concurrency with interleaving concurrency in the synchronous model. This does not restrict the class of systems because in practical examples no two signals are truly concurrent. Therefore, the model chooses from any execution

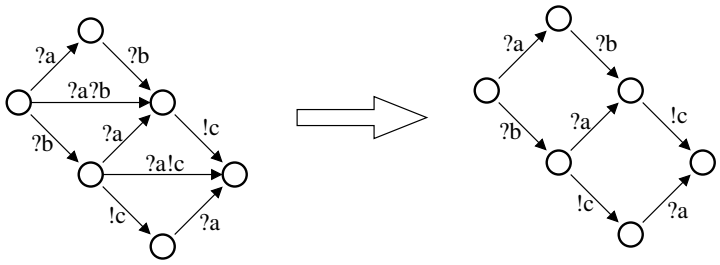


Figure 5.4: True vs interleaved concurrency

sequence and progresses with it.

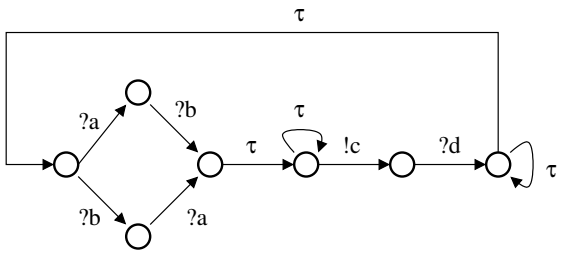
Such a phenomenon is depicted in Figure 5.4.

2. We consider burst-mode systems, where all emissions take place before all the receptions. Therefore, no input burst can be a subset of another input burst leaving the same state. Once the input burst is complete, the circuit activates the specified output burst and enters the specified next state. A new input change is allowed only after the circuit has completely reacted to the previous input bursts.

3. The state that is a destination of a clock transition is not a destination of a non-clock transition.

In Figure 5.5 it can be seen that the destination of $?d$ is also a destination of the clock transition τ. Such a situation is not allowed in our methodology.

4. Finally, a reaction that is atomic in a given state cannot be refined at a later state.



(a) counter example

Figure 5.5: clock assumptions

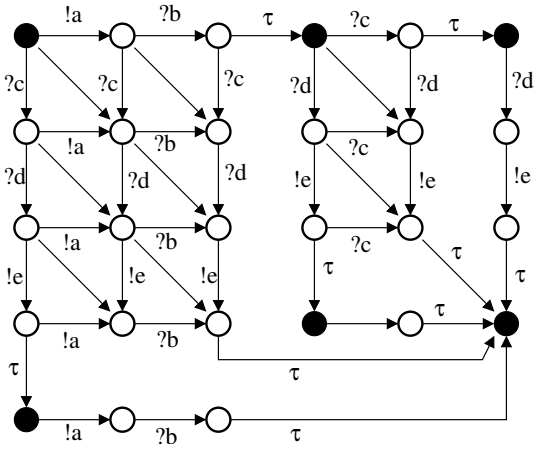


Figure 5.6: Variable flow consistency

In figure 5.6, the reaction $?c; ?d!e$ is refined to $?d; !e$ after the first clock tick. Therefore, it can be seen that the trigger conditions for $!e$ were $?c$ and $?d$ in the first instant and changes to $?d$ at the next instant which follows after the next clock tick. Such a situation is not covered by the synthesis algorithm.

5.5.2 Pre-requisites for transformation

Before presenting the algorithm for the synthesis of weakly endochronous μSTSs, it is required to define some equivalences between the original and the transformed models after the application of actions like signal deletion and insertion on μSTS.

Proposition 5.7. *Behavioural Equivalence for signal deletion on $\mu STS(\approx)$:*

Let $s. \rightarrow$ be a signal deletion transformation of a weakly endochronous system from $\Sigma_1 = (S, V, \rightarrow_{\Sigma_1})$ to $\Sigma_2 = (S', V', \rightarrow_{\Sigma_2})$. If $\Sigma_1 R \Sigma_2$, where R is the symbol for bisimilarity relation, then $\Sigma_1 \approx \Sigma_2$.

Let $s. \rightarrow$ be a sequence of silent events deleted transforming $\Sigma_1 = (S, V, \rightarrow_{\Sigma_1}$) to $\Sigma_2 = (S', V', \rightarrow_{\Sigma_2})$. In order to prove relation R between the original and the transformed graph, we define the relation $\Leftrightarrow$. If the two graphs are related by $\Leftrightarrow$, then we can say that the two graphs are bisimilar up to the relation $\Leftrightarrow$. Let the *injective* function be defined by γ. Then, the relation $\Sigma_1 R \Sigma_2$ exists, iff there exists γ between the elements of Σ_2 and the $non - a$ and $non- \rightarrow_a$ and $non - v$ elements of Σ_1, where a is an element of the deleted set of states, $\rightarrow_a$ is an element of the deleted set of transition relations and v is an element of the the deleted set of variables, such that for every flow relation $\rightarrow$ and state s of Σ_2,

- $l(v) = l(\gamma(v))$ (γ preserves the labels of the variables)

- $\rightarrow \in \bullet s$, then $\gamma(\rightarrow) \in \bullet\gamma(s)$ and $s \in \rightarrow \bullet$, then $\gamma(s) = \gamma(\rightarrow)\bullet$ (γ preserves the pre-set of the states and the post-sets of the transition relations).

Intuitively, we can say that $\Sigma_1 \Leftrightarrow \Sigma_2$ iff Σ_2 can be obtained from Σ_1 by deleting a sequence of silent events, consisting of a set of states, variables and

transition relations. Th silent events are events that cause a change of state but are not observable by the external observer. If $\Sigma_1 \Leftrightarrow \Sigma_2$ holds then $\Sigma_1 R \Sigma_2$ holds. Hence, the relation $\approx$ is satisfied and we can say that the Σ_1 and Σ_2 are behaviourally equivalent.

The notion of validity for signal insertion by transition splitting μSTS is straightforward and the transformation can also be justified in terms of bisimulation, similar to the one presented for signal deletion.

5.5.3 Synthesis algorithm steps

The synthesis methodology derives an asynchronous wrapper to enable the deployment of GALS architecture. The automaton requires to go through some transformations to incorporate handshake and clock control signals for the implementation of GALS architecture. This automaton must follow the assumptions stated in Subsection 5.5.1, before it is finally implemented.

1. We consider an automaton, denoted by $\Sigma = (S, s_0, V, \tau, T)$. This automaton satisfies weakly endochronous criteria and hence it can be synthesised as an independent module without the information of the global system it is integrated into.

2. A set of states $Sync = s_1, s_2, ..., s_n$ is determined that are destinations of clock transitions, denoted by τ. Such states are called *Synchronizing states* where synchronous reactions begin.

For the example in Figure 5.7, $Sync = (s_1, s_2, s_3, s_{4,})$.

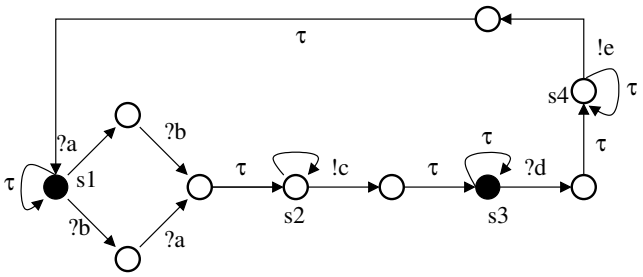


Figure 5.7: Original model

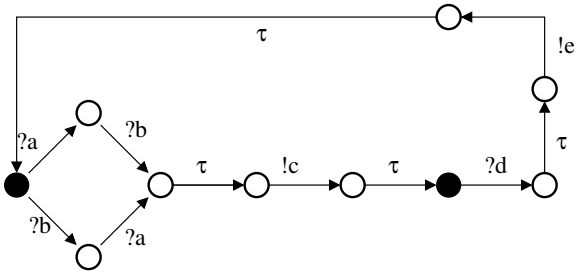


Figure 5.8: Model without stuttering steps

3. The stuttering steps (Axiom 4) are removed from the original model.

This is in accordance with the functional correctness because addition or removal of stuttering steps does not effect the behavior of a system. The net obtained after the removal of the stuttering steps is shown in Figure 5.8.

4. From the Weakly Endochronous criterion 3, we can divide the signal sequences into smaller parts in the synchronous component. Hence, $\forall s \in Sync$, determine the set $Sync_{small}$ of all smallest reactions, in terms of inclusions of sets of operations (*emissions* and *receptions*) executed along the reaction.

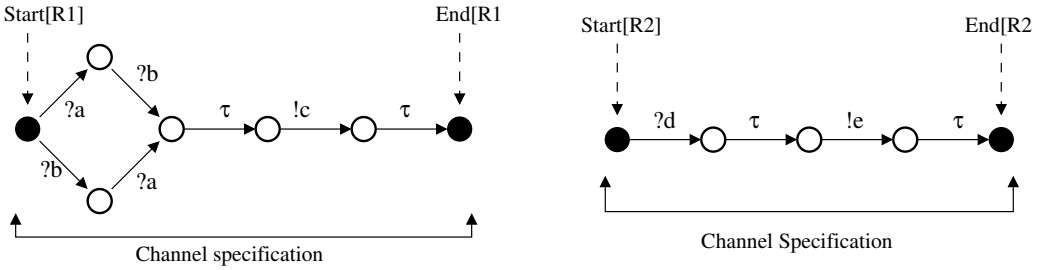


Figure 5.9: Model partitioned into reactions

5. $\forall r \in Sync_{small}$, such that $r \in Reactions$, we check that all emissions are after all receptions. From the determinism of WE systems, we can assume that $Sync_{small}$ contains equivalence classes of smallest reactions that have a set of *reception* operations followed by a set of *emission* operations.

Figure 5.9 shows the partition of the model into reactions. These smallest reactions have all the emissions taking place after all the receptions.

6. To every equivalence class $Sync_{eqiv}$ in $Sync_{small}$ we can associate:

   $Start[R]$: the unique initial state of all reactions.

   $End[R]$: The unique destination τ for each reaction that appear

   - after the set of emission transitions for equivalence classes consisting of sets of reception transitions followed by sets of emission transitions.
   - after the set of reception transitions for equivalence classes consisting of sets of reception transitions.

7. For all the equivalence classes with sets of receptions followed by emissions, delete the sequence $s \stackrel{<\tau>}{\rightarrow}$, where, $\stackrel{<\tau>}{\rightarrow} \in \bullet End[R]$ and $s \in \bullet \stackrel{<\tau>}{\rightarrow}$.

This step is not applied to those classes that only contain reception transitions.

Since the signals deleted/removed from the original model are *silent events,* the modified model can be shown behaviourally equivalent (Proposition 5.7) to the original model.

Each equivalence class consists of sets of transitions that have all their reception transitions before the emission transitions. Therefore, reconstruction of the final model will lead to the connection of the emission transitions of one class with the reception transitions of the next class that appear along the same path. In a GALS environment clock transitions are only required after the reception of data to aid computation. The clock transition should be absent at other times during which the system operates in a causal mode to avoid re-synchronisation issues. Therefore, the clock transitions between emission and reception transitions are removed while preserving the behavioural correctness of the model.

8. Handshake expansion is applied to the partitioned models. The channel specification of the models only contains the active transitions. The handshake expansion equips the model with both active and passive transitions which are essential requirements for synthesising a net.

This step is elaborated to show the DAG refinement using *handshake expansion* [85]. The channel representation, denoted by $!a$ (emission of signal a) and $?a$ (reception of signal a) cannot be directly implemented by existing STG based

synthesis tools, since the falling transitions are not specified. Therefore, to implement the above STG, with channel specification, handshake expansion is applied to obtain a refined specification of the circuit.

This expansion is brought about by substituting each channel with two wires. The channels are re-labelled by $request$ and $acknowledge$ signals for passive and active ports of a module. For instance, channel a is specified as a_{in}_req and a_{in}_ack for passive port and a_{out}_req and a_{out}_ack for active port. The event τ is refined into $clk+$ and $clk-$ transitions.

An additional ordering constraint is introduced at this stage to satisfy the GALS criterion. This constraint is in accordance with the correctness criteria. The ordering constraint is defined by a protocol which is characteristic of *stoppable clocks* presented in [79]:

- The clock is raised $(clk+)$ after all the input requests have been received.
- The synchronous computation is triggered by the $clk+$ signal. The completion of the computation is reported by the assertion of the *Completion Detection* (cd) signal.
- On reception of cd, the clock signal clk is lowered $(clk-)$.
- The output request is sent to the next module.
- The module then waits for the output acknowledgement signal.
- On the reception of the acknowledgement signal, all the input request signals are acknowledged.

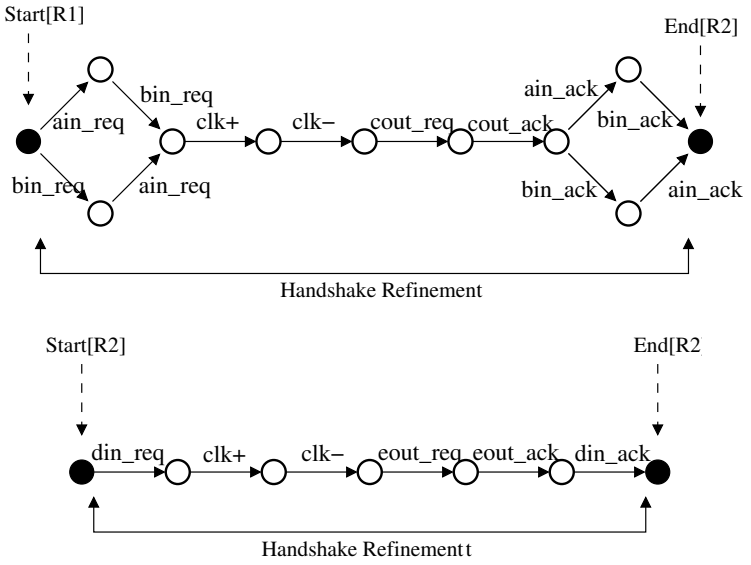


Figure 5.10: Handshake refinement

- When all the input requests are acknowledged, the module is ready to receive new input request signals.

The clock is stopped when the module is idle. After the reception of the required subset of data, the clock is started to perform the relevant computations. On completion of the computations the clock is again lowered, waiting for new input data to arrive.

The refined model after the application of the handshake refinements is depicted in Figure 5.10. The correctness of refinement is discussed in Section 5.7.

9. The clock is controlled by the input signals, i.e., the arrival of all the input signals notify the clock control signals that the rising edge of the clock can be allowed for computation. The completion of the computation, denoted

by a signal cd (completion detection signal), allows the clock signal to be lowered to generate the outputs obtained from the computational block. The silent event cd is a newly inserted signal and complies with the notion of validity for signal insertion.

Such a transformation is depicted in Figure 5.11. The reception of all the input signals, namely, a_req and b_req triggers the clk signal to high. On the reception of the completion detection signal, cd the clock goes low denoting the availability of outputs to be written. Here, we would like to introduce the concept of Request driven clocking scheme. This scheme is a direct outcome of the synthesis steps 5 and 8. In 5 we consider sequence of transitions where all the emissions take place after all the receptions. In step 8, we raise the clock when all these inputs (receptions) have arrived and lower the clock when the computation is completed and outputs are written.

10. To every refined equivalence class $Sync_{eqiv}$ in $Sync_{small}$ we can associate:

    $Mid[R]$: the unique state that separates all receptions from emissions in each reaction, i.e, the active reception transitions before the clock is triggered.

    $Clk[R]$: = the sequence of transitions corresponding to the critical region where the clock is raised, the computation is performed,the clock lowered and the communication completed. This is also the transition sequence between the states $Mid[R]$ and $End[R]$.

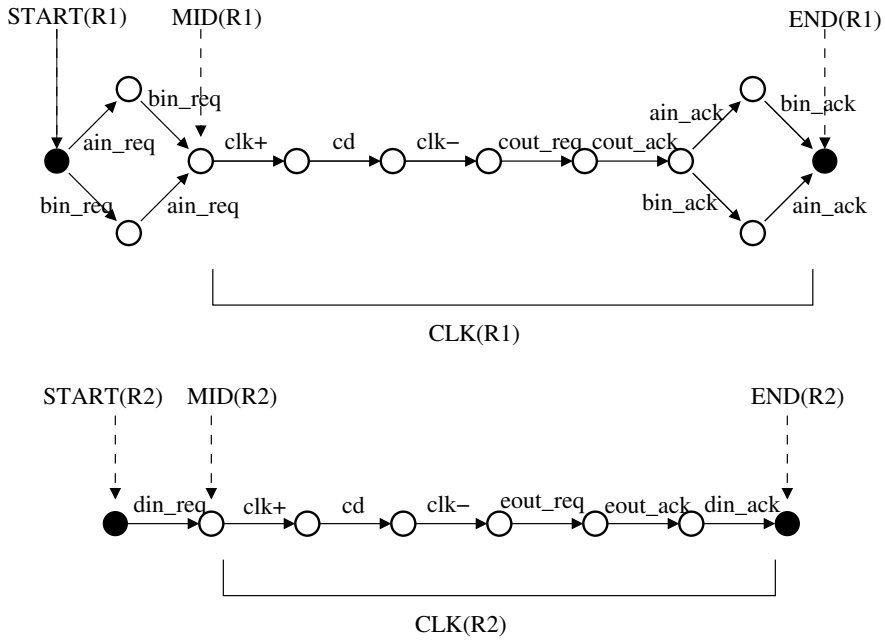


Figure 5.11: Partitioned model with states assigned to sets

Figure 5.11 identifies the states and assigns them to the relevant states as discussed above.

11. $\forall s$ in $Sync$:

    $\forall Sync_{equiv}$, we consider all the states and sequences of transitions from s to $End[R]$ and all their interleavings. This automaton is denoted by S. It is to be noted that S is contained in Σ.

12. As a final step, we need to construct the final implementable model. We presume that all the states in the original model have identities, then

    $\forall Sync_{equiv}$ in $Sync_{small}$:

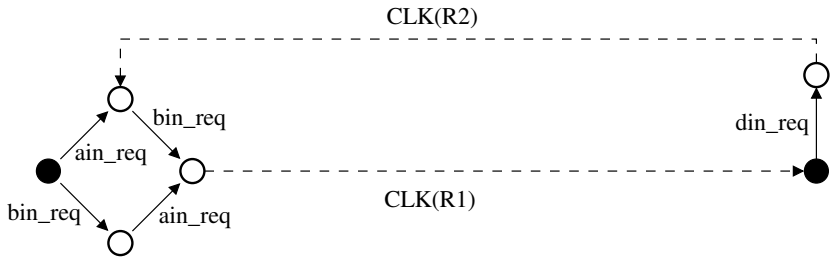


Figure 5.12: Final transformed model

all the states with the same identity are merged to obtain the final model of the component. For example, if $End[R1]$ is a source of another reaction with initial state $Start[R2]$ appearing along the same path, then merge $End[R1]$ and $Start[R2]$. The states $End[R1]$ and $Start[R2]$, would have the same identity since they are the same state in the original model, before the partitioning step.

Applying the above rules the final net is shown in Figure 5.12. This net is implementable and can be fed to Petrify for further automated refinements.

The above steps are straightforward when there is a clear distinction between the receptions and emissions. If an emission is concurrent to a reception operation, then there would be a reaction with a sequence which has emission before reception. In such a situation, the reaction with such a situation is omitted from the final transformed model. This does not affect the behaviour of the system, because the system is forcibly made to choose the sequence where the reception is before the emission.

13. The transformed automaton of each of the components that comprise the

synchronous system are fed to Petrify to translate them to Signal Transition Graphs (STG).

This is done automatically by using the Theory of Regions presented in Subsection 5.2.2.

14. The STGs thus obtained can be logically synthesized using Petrify. At this stage we have the circuits of the individual components. These components are then made to communicate with each other using arbitrary length asynchronous FIFOs.

If the states of the FIFO are ignored, the final net $N = \{P, T, W, M_{N_0}\}$ of a system composed of two components depicted by nets $N_1 = \{P_1, T_1, W_1, M_{N1_0}\}$ and $N_2 = \{P_2, T_2, W_2, M_{N2_0}\}$, obtained by directly connecting the outputs of N_1 to the inputs of N_2 can be defined by:

$$P = P_1 \cup P_2$$

$$T = T_1 \cup T_2$$

$$W(p,t) = \begin{cases} W_1(p_1, t_1)\, if\, p_1 \in P_1, t_1 \in T_1 \\ W_2(p_2, t_2)\, if\, p_2 \in P_2, t_2 \in T_2 \end{cases}$$

$$W(t,p) = \begin{cases} W_1(t_1, p_1)\, if\, p_1 \in P_1, t_1 \in T_1 \\ W_2(t_2, p_2)\, if\, p_2 \in P_2, t_2 \in T_2 \end{cases}$$

$$l(t) = \begin{cases} l_1(t_1)\, if\, t_1 \in T_1 \\ l_2(t_2)\, if\, t_2 \in T_2 \end{cases}$$

$$M_N = \begin{cases} M_{N_1}(p_1)\, if\, p_1 \in P_1 \\ M_{N_2}(p_2)\, if\, p_2 \in P_2 \end{cases}$$

15. The refined model of each of the synchronous module is fed to Petrify to obtain the logic equations for gate level implementation. A detailed description of obtaining such implementations from Petri nets can be found in [50]. Each of the component circuit thus obtained communicate with other modules of the system via *asynchronous FIFOs.*

5.6 Case Study:DLX architecture

We de-synchronize the *DLX* datapath architecture [86] to exemplify the proposed traversal from weakly endochronous systems to latency insensitive circuits. Here, we consider a simple unpipelined *DLX* architecture. Our approach can be directly extended to pipelined *DLX* architecture. The Figure 5.13 shows a simplified and abstract view of the overall partitioned *DLX* architecture. The globally synchronous system is partitioned into five main synchronous islands, Instruction Fetch(*IF*), Instruction Decode*(ID*), Execution(*EX*) and Write Back(*WB*). In [86] the *ID* and the *WB* stages are merged. The instructions from the *MEM* block synchronise with the instructions coming from the *IF* block. These islands operate at different clock speeds.

Desynchronisation of the *DLX* architecture into the above partitions would allow each island/block to retain a local clock whose frequency can be scaled independently of the other blocks. Moreover the blocks operate with increased concurrency which cannot be achieved in a globally synchronous environment. The dotted lines shows the synchronous island that will be used as a running

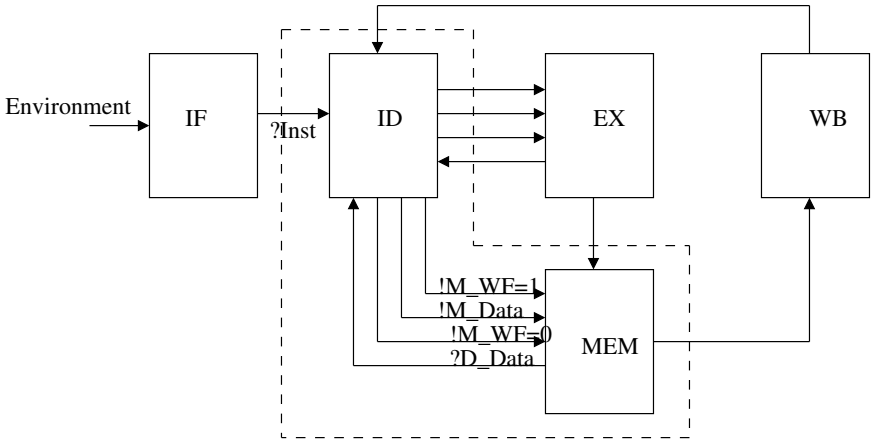


Figure 5.13: DLX architecture

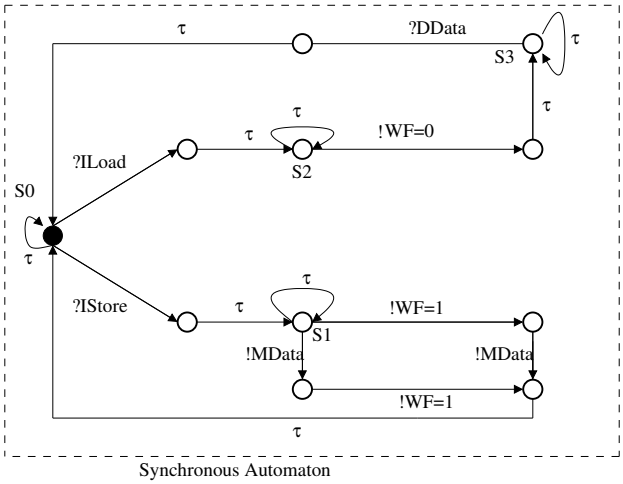


Figure 5.14: DLX-ID automaton

example for demonstrating the synthesis methodology. The block chosen is the ID block and its interaction with EX and the MEM blocks. This block receives data from Instruction Fetch(*IF*) block and communicates with the *MEM* block, with exchange of data between them.

From the theory presented in this chapter, if each of these blocks are modeled satisfying the WE criteria, then they can be composed correctly by making each block communicate with the other blocks through asynchronous FIFOs, to form a correct-by-construction GALS architecture. Therefore, we proceed by modeling the ID to present the results of our synthesis methodology.

Figure 5.14 depicts the automaton of the *Instruction Decode* component of the DLX architecture. This automaton waits for signals $?Load$ and $?Store$ to perform two different computations. If the automaton receives $?Load$, it produces $!WF = 0$ (*write flag=0)* signal. After this operation, it waits for $?D\_Data$. The signals $!WF = 0$ and D_Data is separated by τ transitions. This is because, a system cannot write and afterwards perform another read in the same clock cycle. After the operation $?D\_Data$ the automaton returns to its initial state s_0.

For the example in Figure.5.14, $Sync = \{s_0, s_1, s_2, s_3\}$.

For the above example the allowable equivalences classes are,

$Sync_{small}(s_0) = \{\{?Load\,!WF = 0\,\tau\}, \{?Store\,!WF = 1\,!M\_Data\,\tau\},$
$\{?Store\,!M\_Data\,!WF = 1\,\tau\}\}$

$Sync_{small}(s_3) = \{\{?D\_Data\,\tau\}\}$

The transformed reactions are shown in Figure 5.15.

Construction of the final model following the algorithm steps gives rise to a model depicted in Figure 5.16.

In a similar way, the models for the other blocks of the DLX architecture are obtained. Since each of the blocks thus obtained after the application of the different steps of transformation are weakly endochronous, the composition of these

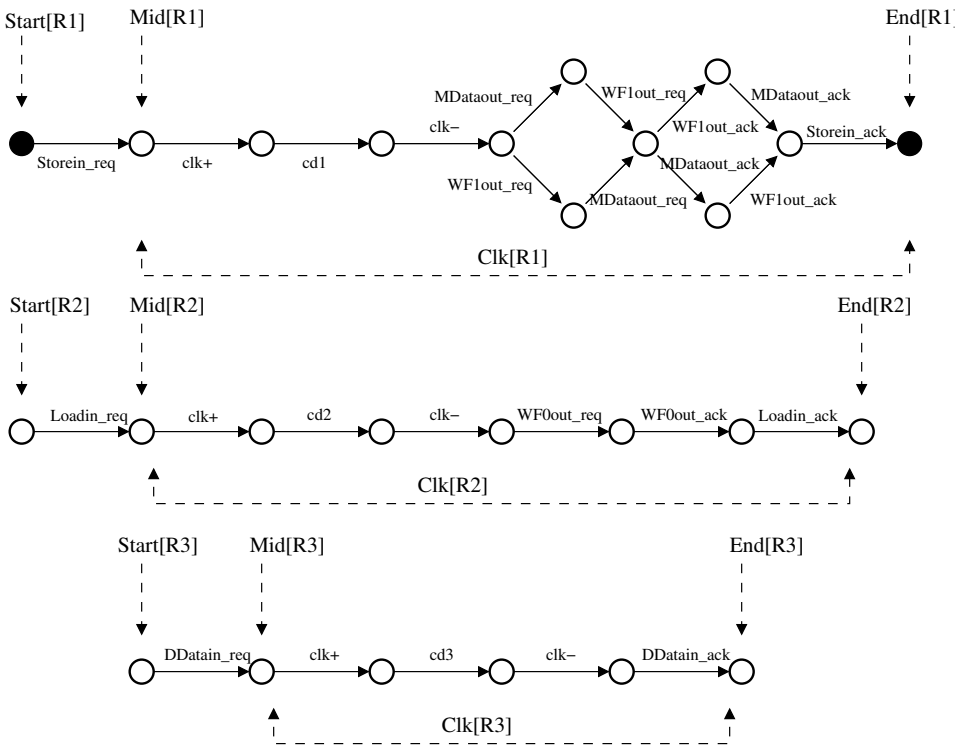


Figure 5.15: Reaction refinement

blocks via FIFOs will be weakly endochronous. Then from theorem 2 we can say that the GALS implementation is correct w.r.t the synchronous specification.

5.7 Correctness of Handshake Refinement

The handshake refinement applied during the transformation steps should be correct with respect to the original net. Correctness of refinement can be shown with the notion of *observational equivalence* [85] between the original channel specification semantic and the two wire refinement model. We define *observational*

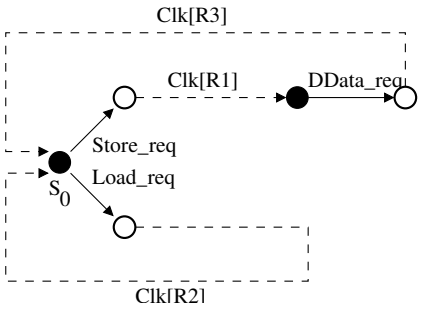


Figure 5.16: The final model

equivalence between the original and refined PNs as an existence of a mapping between a set of original events and refined events. For every original action such a mapping selects a ***Critical event*** (E) from a set of all events (E_1) of the refined model. For example, the event of reception of an input signal $(!Load)$ at the passive port is refined in the two wire model as $[Load_{in}\_req, Load_{in}\_ack]$. Here, the critical event is $Load_{in}\_req$ which denotes the availability of an item of data to be read at the passive port, while $Load_{in}\_ack$ is regarded as a silent event. The events (E_1/E) are regards as *Auxiliary events.* These *Auxiliary events* may be placed in the specification according to the designer's choice. This choice depends on the ordering constraint imposed by the protocol the circuit follows. Therefore for the above example, $Load_{in}\_ack$ only occurs on the reception of output acknowledgement signals. This protocol followed by the refinement methodology is presented in Section 6.

If only the critical signals are considered, then it can be observed that the original and the refined PN models give rise to the following traces:

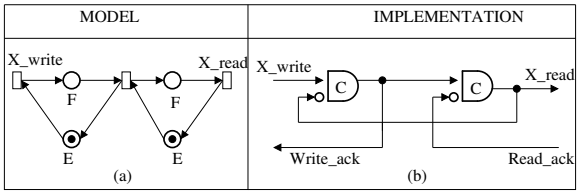
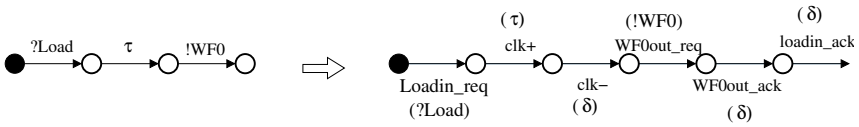


Figure 5.17: Single rail FIFO model and implementation



Therefore, the two are observationally equivalent. A similar equivalence can be showed between the original and the final refined model of the DLX ID block.

5.8 Implementation

The final model obtained in Section 5.6 is translated into PN using the theory of Regions. This translation is done by the logic synthesis tool *Petrify.*

Asynchronous FIFOs are chosen as the communication primitives to connect the different clocked domains, working at different speeds. In the model we use a very straightforward design of a standard FIFO which is a basic requirement of the system. The model and implementation of such a single rail FIFO is shown in Figure 5.17. Therefore, each signal is communicated via dedicated FIFO channels. Figure 5.17(a) and (b) show the Petri net representation of a $2-$depth FIFO and the C-element circuit implementation of the FIFO, respectively. Such structures have been explicitly studied in [84].

5.9 Summary

This chapter presented a method to model and synthesise delay insensitive modules using a well established concurrency modeling language, Petri nets. This makes it possible to use existing asynchronous tools for model checking and logic synthesis of such modules. A set of assumptions have been presented that are required to be followed by the synchronous specification of the synchronous components for the application of methodology for the transltaion of the *WE* components into delay insensitive circuits. The method has utilised the structural and functional similarities between the underlying models of computations, namely, *State Transition Systems (STSs)* and the *Reachability Graphs (RGs)* of the Weakly Endochronous components. The models have undergone steps of transformation to bring the component description to an implementable level. One of the transformation steps includes the translation of the implicit FIFO communication scheme incorporated in *WE* system specification to an asynchronous handshake protocol to aid the implementation of the components. This protocol reuses the *data driven clocking* scheme, presented in Chapter 4. Therefore, the locally synchronous blocks only consume power when all items of data have arrived for a computation process to begin. When the computation is completed, the module goes into a *"sleep mode"* during which asynchronous transfers take place.

A generic synthesis algorithm has been defined, that incorporates all the transformation steps, and ensures the properties for correct GALS deployment is satisfied. This algorithm applies the theoretical results obtained in [69] to obtain a

final distributed delay insensitive circuit from a synchronous specification. The wrappers thus obtained uses asynchronous FIFOs as a communication primitive to communicate with other components of the system.

Chapter 6

Desynchronisation Technique using Petri nets

6.1 Introduction

This chapter introduces a new methodology for the desynchronisation of synchronous systems into globally asynchronous and locally synchronous (GALS) architectures. In the previous chapter Transition Systems (TS) were used as the specification model to describe the synchronous systems for the purpose of desynchronising the system into GALS architecture. The models obtained for each synchronous module can be very large and complex due to the weak handling of concurrency posed by the previous desynchronisation methodology. Concurrency is a prerequisite for the specification of synchronous systems which are required to be equipped to handle asynchronous communication for their GALS deployment.

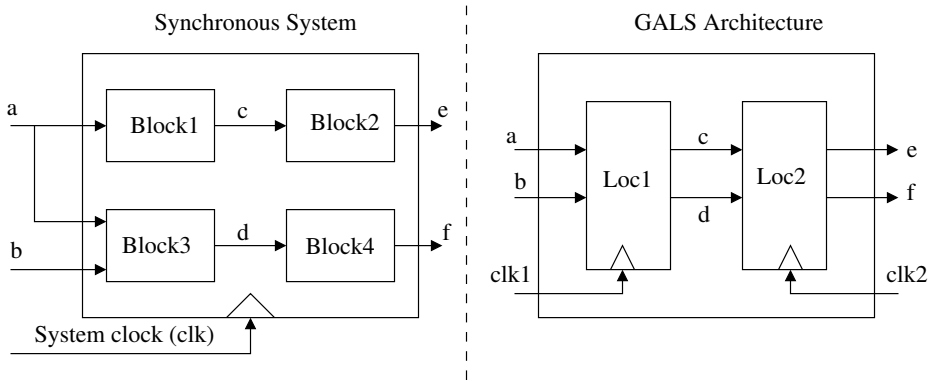


Figure 6.1: Synchronous system transformation into distributed architecture

Moreover, these models are translated into Petri nets in order to use existing asynchronous tools for logic synthesis.

Therefore, the complexity of the transition system, obtained from the previous methodology, and the computational complexity of the PN synthesis of these models form the main motivations for this work. The new technique uses PN as the specification model whose efficient concurrency handling technique makes it one of the most viable models to describe systems for desynchronisation. Moreover, the theory behind the new technique uses the concept of *Localities,* inspired by [87], which helps in describing the distribution of a synchronous system over asynchronous architectures owing to its strong structural and functional correspondence with GALS architectures.

A synchronous system consists of components which are associated with a set of input and output signals. Such a system is depicted in Figure 6.1. Each of these components are governed by activities like sending and receiving con-

trol signals as well as exchanging data signals across different components. The actions performed by all the components are controlled by a single global clock.

The notion of *localities* inspired by [87] in application to biological membrane systems, introduces the idea of localising these above mentioned components and hence their associated actions into individual blocks. These blocks are incorporated with some additional ordering constraints on their input and output signals. These constraints enable the individual blocks or localities to behave like independent synchronous systems. Therefore, the global clock can be eliminated and each locality can be employed with local clocks which govern the actions assigned to them. Such a transformed system is depicted in Figure 6.1. As is evident from the figure, more than one component can be mapped onto each locality depending on some rules and optimisation criteria, discussed later. These individual localities created would then communicate with each other asynchronously due to the absence of a clock signal in between the localities owing to the removal of the global clock and application of the local clocks. The technique to obtain a distributed architecture from a globally synchronous system must satisfy two essential correctness properties, namely,

- *semantics preservation of the original synchronous system*: During the execution of each synchronous sequence, components of the synchronous system compute events for the output signals based on the internal signals and the values of the input signals. Within each unit of time, the system is transformed by a maximally concurrent execution of input and output signals. The deployment of such a system into GALS architecture entails un-

bundling of input signals to aid out-of-order reception of these signals in an asynchronous environment. Therefore, this transformation to form a distributed architecture should preserve the semantics of the original system.

- *prevention of deadlocks*: When the synchronous system is transformed into a GALS architecture, the input transitions that were bound in the original system are unbundled, as previously discussed. This out-of-order reception of signals should not cause the system to enter into a deadlock state. Therefore, there should be additional constraints in the transformed model to avoid such occurrences.

Both the properties have been dealt with in Section 6.6.

The desynchronisation methodology proposed in this chapter can be summarised in the flow diagram shown in Figure 6.2. The rest of the chapter will deal with the various steps presented in the shaded sections of the flow.

6.2 Preliminaries

This work uses Petri net models to describe the synchronous systems. This is because all the components and actions carried out by synchronous systems can be directly mapped onto the different elements of a Petri net. For instance, synchronous events are represented on the *transitions* and the trigger conditions are denoted on the *places.* In order to show that a trigger condition is true, the place is equipped with a token. To make a synchronous component transit from one

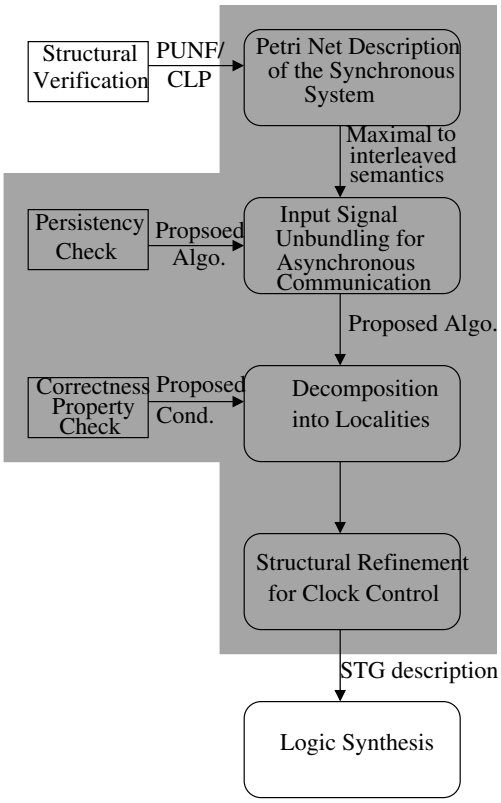


Figure 6.2: Flow diagram of the proposed methodology

configuration to another is denoted by the firing of transition(s). Therefore, PN models are sufficiently expressive in describing a synchronous system. A detailed description of such models is presented in Section 6.4.

6.2.1 Petri nets

We recall some important notations concerning Petri nets (see Chapter 2), that will be reused in this Chapter.

Given a Petri net N, the *pre-* and *post-multiset* of a transition t are respectively

the multiset $pre_N(t)$ and the multiset $post_N(t)$, such that for all $p \in P$, $|p|_{pre_N(t)} = F(p,t)$ and $|p|_{post_N(t)} = F(t,p)$, where $|p|$ denotes the number of tokens present in the place p. Since, all the systems defined in this work are *safe*, $|p| = 1$.

Definition 6.1. Step

A step is a multiset of transitions $U : T \rightarrow N$, where N is a set of natural numbers.

The steps can be executed in various modes depending on the system that the PN models describe, discussed in Sections 6.4 and 6.5.

For a PN to be synthesisable, it is required to satisfy some *Behavioural* and *Structural* properties, discussed in Chapter 2. We recall an important behavioural property, namely, *Persistency* because it plays an integral role in our desynchronisation methodology.

Definition 6.2. Persistency

A Petri net (N, μ_0) is persistent if for any two different transitions t_1, t_2 of N and any reachable marking μ, if t_1 and t_2 are enabled at μ, then the occurrence of one cannot disable the other.

We can generalize the notion of persistency to apply to steps.

6.3 Motivation for using Localities

Our initial model starts with the description of a globally synchronous system. In such a system, execution and communication progress along a sequence of events

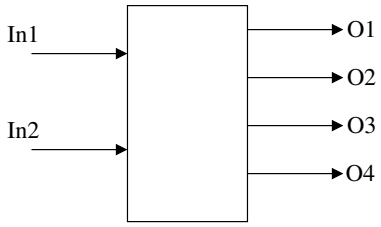


Figure 6.3: Simple synchronous block

which are tagged by a global logical clock, i.e., they are active only at certain logical instants. During the execution of such sequences, each of the synchronous system components compute events for the output signals based on the internal signals and the values of the input signals. This global clock paradigm is associated with max firing semantics and transition binding. In order to exemplify the different firing semantics, a simple example is considered in Figure 6.3.

As stated earlier, Petri net models are used to demonstrate the globally synchronous system. In this model the synchronous events are represented on the *transitions* and the trigger conditions are denoted on the *places*. In our PN model of the synchronous system, we presume that every event on the transition is implicitly tagged by the global clock and hence, for simplicity, we do not show the actual clock transition on this model. In such a net there are disjoint sets of inputs I and outputs O and a function l which maps the transitions of the Petri net to the set $I \cup O \cup \{t_{int}\}$, where $t_{int} \notin I \cup O$ is a silent event not observable by the environment. Let the inputs $In1$ and $In2$ be concurrent to each other. Let Figure 6.4 denote the Petri net representation of the input output dependencies of the system which is shown in Figure 6.3.

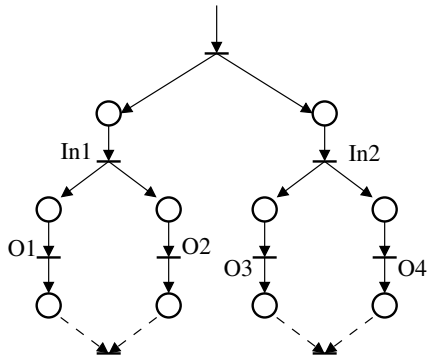


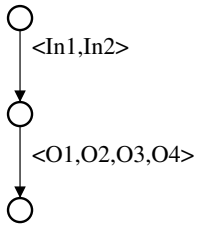
Figure 6.4: PN model of the synchronous block

As mentioned before, a globally synchronous paradigm is associated with maximal firing semantics. A state graph depicting such a semantic is presented in Figure 6.5(a). In order to desynchronise the synchronous system into a GALS architecture, the input steps are required to be unbundled to enable out of order communication. In a synchronous system when two inputs are unbundled and received at deterministic instances of time, Figure 6.5(b) is obtained.

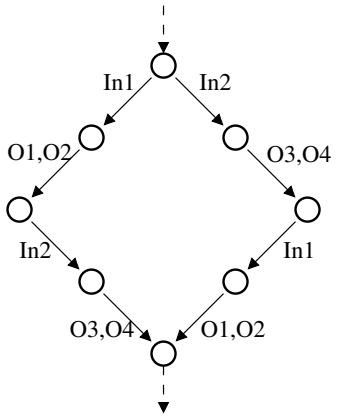
If a system is globally clocked, the inputs, outputs and the internal signals can be scheduled to fire in persistent (Definition 6.2) steps since they can be made to be generated at known instances of time.

Such a schedule cannot be maintained in a GALS environment, since the inputs do not arrive at known instances of time. Therefore, in order to realise this system in an asynchronous or GALS environment, additional conditions are required to be added to:

- Prevent the system from entering deadlock states arising from the arrival of out of order inputs



(a) Bundled step model



(b) ununddled input ordered state machine

Figure 6.5: System model

- Exploit the advantages of asynchrony, by allowing the inputs to arrive as and when available leading to increased concurrency.

In order to incorporate the idea of asynchrony, the inputs must be allowed to arrive in any order and at any instant of time. This results in unbundling of inputs as shown in Figure 6.5(b). Since the input signals cannot be scheduled to arrive at known instants, persistency property cannot be guaranteed. After applying this feature the state machine of the same system takes the form shown in Figure 6.6. This results in an unknown delay between the inputs. Therefore, from the figure it can be seen that the model has non-persistent steps at state s_1 and s_2.

To exemplify the following example is considered. Let$< In1 >$ arrive first, which causes $< O1, O2 >$ to execute in a maximal step. But before the execution of the maximal step $< O1, O2 >$ is completed, if $In2$ arrives then the system attempts to execute the maximal step $< O1, O2, O3, O4 >$. Therefore, the arrival

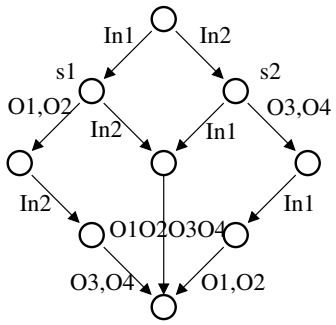


Figure 6.6: Unbundled out-of-order inputs system model

of $< In2 >$ disables the step $< O1, O2 >$leading to violation of persistency property between the steps $< In2 >$ and $< O1, O2 >$at s_1. Non-persistent steps at the state s_2 can be easily shown in a similar way.

In order to avoid this situation, the system is not made to follow Max-O semantics globally. If it is possible to partition the system in such a way that none of the input transitions and output steps are non-persistent in each partition, then the Max-O semantics can be restricted to each partition leading to a correct realisation of a concurrent system. This gives the motivation for the use of *localities.* In order to obtain a correct implementation of a GALS system from a synchronous specification, the synchronous system is required to be partitioned into localities, which are analogous to partitioned blocks. The importance of persistency for output steps is associated with the stable conditions in which the local synchronisation (bundling of outputs) is performed later in the implementation. Therefore, the partitioning of the global synchronous system into localities and application of Max-O semantic in each locality aid the removal of the global clock by guaranteeing

the absence of deadlock and the realisation of correct input output dependencies.

6.3.1 Max-O semantics and validity criteria using Processes

The previous section presented the idea about Max-O semantics used to describe distributed architectures. This notion is required to be handled by the specification models, in this case PN models, used to describe such systems. The standard interleaving semantics for PN does not associate any notion of maximal firing by which a set of transitions are always fired concurrently. Therefore, *maximal output semantics* is introduced which binds sets of output transitions in order to fire them concurrently.

In this section we draw some equivalences between models of PN with *maximal output semantic* and *standard semantics*. The reason for obtaining such equivalences is to use PNs that are behaviourally equivalent under both the semantics due to the feasibility of verification and synthesis. Hence, the models used to represent our system are those that are equivalent under standard and Max-O semantics. Here, we require to define the restrictions that support the above equivalence. This is done with the help of theory of Processes, which was introduced in [88].

A process can be represented as a labelled acyclic graph, with places having at most one incoming and one outgoing arc. The processes can be viewed as subnets of unfolding. Let $\sqsubseteq$ be the prefix relation on processes. The nodes of the processes have identities, i.e. they are not anonymous. Therefore, if $\pi \sqsubseteq \pi^{'}$, then $\pi^{'}$ is a continuation of π rather than some unrelated to π process whose initial part is isomorphic to π.

Definition 6.3. Behavioural Equivalence

Let $\Sigma = \{P, T, M, \mu\}$ be a Petri net model. Let PN_{STD} be the reachability graph of Σ under standard semantics and PN_{max} be the reachability graph of Σ under the Max-O semantics. Let π and $\pi^{'}$ be the sets of all finite processes of PN_{STD} and PN_{max}, respectively, then

1. $\pi^{'} \leq \pi$
2. $\gamma \sqsubseteq \gamma^{'}$

where, $\gamma \in \pi$ and $\gamma^{'} \in \pi^{'}$.

The standard semantics have interleaved output steps and the Max-O semantics have maximal output steps. Hence, the interleaved semantics will have more permissive steps as compared to Max-O semantics. Therefore, intuitively we can say that the processes of standard semantics are greater than the processes of Max-O semantics.

To prevent the Max-O semantic from having additional events which are not permitted by the standard semantic, our second condition comes into play. Therefore, by enforcing the processes of PN_{max} semantics to be a prefix of some process of PN_{STD}, we address the above issue.

Figure 6.7 shows an example of a net that is equivalent under both semantics. In a similar way, the PN models used to describe the synchronous or the distributed systems should be equivalent under both standard and Max-O semantics.

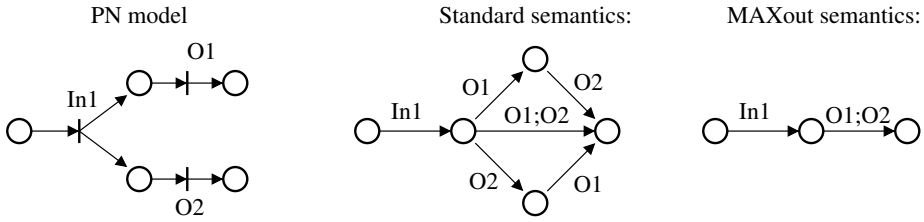


Figure 6.7: A PN equivalent under the two semantics

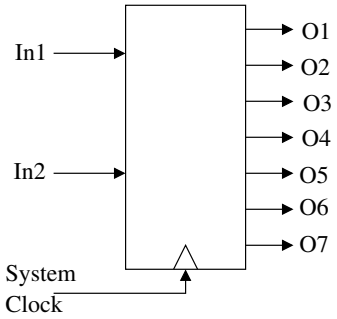
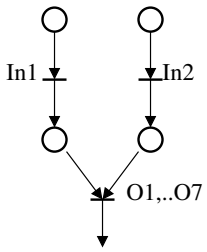


Figure 6.8: Synchronous Block

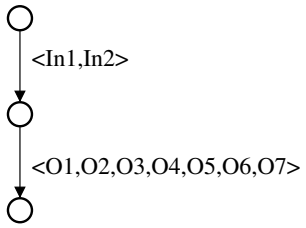
6.4 Synchronous model description

A more complex example is now considered to highlight the main aspects of our desynchronisation methodology. This will be running example for exemplifying the process of GALSification.

Figure. 6.8 shows a typical synchronous system. There are two inputs $In1$ and $In2$ to the block and seven outputs, namely, $O1, O2, O3, O4, O5, O6$ and $O7$ from the block. The system clock is used to clock the whole system globally. The PN model specification of such a system is shown in Figure 6.9(a). The state representation of the maximal firing semantics in a globally synchronous



(a) PN representation



(b) Maximal firing semantics

Figure 6.9: Synchronous model

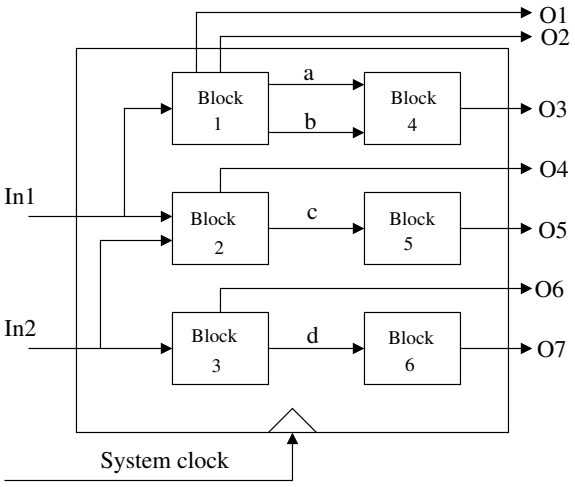
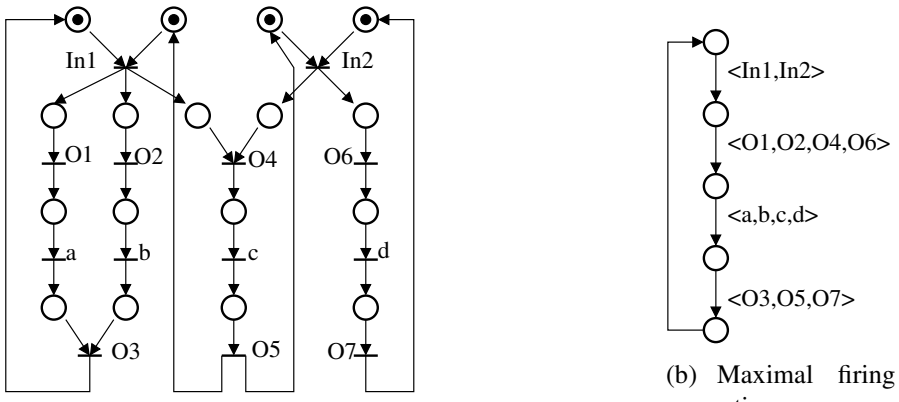


Figure 6.10: The synchronous system

environment is shown in Figure 6.9(b).

Each synchronous system can be further divided into smaller computational blocks. These blocks have their own input signals coming from and outputs going to other similar internal blocks. These signals, when seen from the top level of the single synchronous block, form the internal signals of the circuit. These smaller blocks have their own sets of internal signals. Such a system is exemplified in



(a) PN representation

(b) Maximal firing semantics

Figure 6.11: PN model of the synchronous block

Figure 6.10. The signals a, b, c and d form internal signals to the overall synchronous block. A PN representation of such a system is shown in Figure 6.11(a). The reachability graph of such a system is shown in Figure 6.11(b). Therefore, the system can be partitioned into small computational blocks as shown in Figure 6.10. If the system is required to be further partitioned into smaller blocks, then the inputs $In1, In2, a, b, c$ and d can be split to aid the process of locality formation, as discussed earlier.

For the formation of localities and to aid asynchronous communication between the localities some transformations are applied at the PN model level. At the granularity of the individual blocks that compose the synchronous system, these internal signals form inputs to and outputs from the internal blocks, i.e a acts as an output from block 1, but behaves as an input for block 4. Since the internal signals are now interpreted as output from one block and input into the next block, transformations are applied on the net to incorporate this communica-

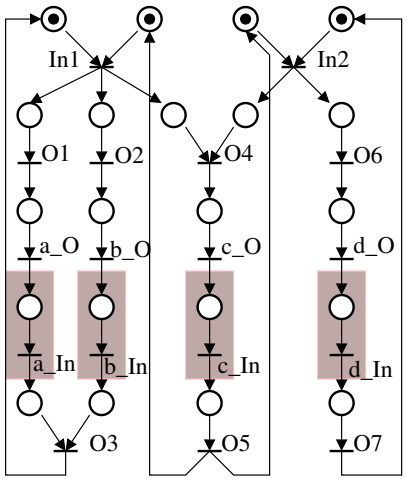


Figure 6.12: Modified model

tion on the channel, in order to distinguish the outputs from the input signals for desynchronisation. This is necessary to incorporate the idea of localities which have sets of input and output transitions allocated to each locality. Therefore, output from one locality forms the input to another. To do so, we partition the signal into output and input signals. For example, signal a is partitioned into a_O and a_In. To do this, the model needs to be transformed by inserting new internal signals, for which the transition insertion technique defined in Section 6.4.1 is used. This refinement leads to a modified PN model of the original system and is depicted in Figure. 6.12. The shaded blocks denote the insertion of signals in the original system.

6.4.1 Net Transformations and notion of validity

In order to obtain a distributed PN model of a system, some transformations are required on the model to aid the compartmentisation process. One such transformation is *Signal Insertion.* In this section, signal insertion by transition partitioning is formally defined. The type of insertion is restricted to *sequential post insertion* because the insertion is to aid the partition of a signal into its output and input counterparts and hence eliminates concurrent insertions.

Definition 6.4. Transition partitioning

Given a *labelled Petri net* $\Upsilon = (\Sigma, I, O, l)$ where $\Sigma = (P, T, F, \mu_0)$, I is a set of inputs, O is a set of outputs, such that $I \cap O = 0$, l is a function that maps the transitions of the Petri net to the set $I \cup O \cup \{t_{int}\}$, where, $t_{int} \notin I \cup O$, the partition of the transition $t \in T$ yields an LPN $\Upsilon' = (\Sigma', I, O, l)$ with $\Sigma' = (P', T', F', \mu_0)$, where,

- $T' = T \cup \{u\}$, where $u \notin P \cup T$ is a new transition
- $P' = P \cup \{p\}$, where $p \notin P \cup T$ is a new place
- $F' = F \cup (\{t, p\} \cup \{p, u\} \cup \{(u, q) \mid q \in t\bullet\}) \backslash \{(t, q) \mid q \in t\bullet\}$

The notion of validity for signal insertion is straightforward and the transformation can be justified in terms of weak bisimulation which is well studied. Such a notion is presented in [[89], Proposition 5.3].

(a) Conflicts

(b) Output fan-outs

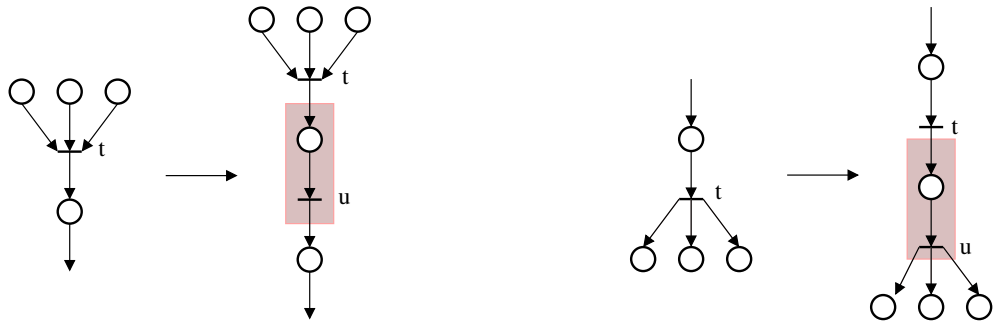
Figure 6.13: Restrictions on transition splitting

Conditions of valid transformations There are some restrictions that are required to be followed while inserting the signals.

- The newly inserted places form the interface places between the different localities. Therefore, these places cannot have the token stolen by another transition in conflict. To avoid a transition from stealing the token and resulting in running one locality into a deadlock, situation depicted in Figure. 6.13(a), should not be allowed. Hence, interface places cannot be choice places.

- If the signal has fan-outs, the buffer should be inserted before the fanout, instead of one buffer in each branch. The later can lead to formation of unnecessary localities due to numerous signal insertions. This is exemplified in Figure. 6.13(b).

Therefore, the allowable examples are shown in Figure 6.14.



(a) insert after fan-ins (b) insert before fan-outs

Figure 6.14: Transition partitioning

Transition re-labelling The transitions t (the transition which is split) and u (the newly inserted transition) are labelled by adding a post-fix $\_O$ to the label of t and $\_In$ to the label of u. This is done to associate meaning to the inserted signals which signify channel communication. Therefore, for the example shown in Figure 6.11, the transition labelled a is split into a_O, denoting output from block 1 and a_In, denoting input to block 4.

The newly inserted place $t\bullet$ can be regarded as a unit of storage, for instance a finite FIFO. This FIFO stores item of data before transferring it across to the next block.

For a synchronous system, the *Input transitions* should be able to fire as and when the tokens are available. On the reception of the inputs, all the outputs that are dependant on this input are generated together.

These steps thus defined can have different modes of execution. To define the synchronous execution modes we can define the following:

- Free-execution - This means that the events can be executed in any order

when the trigger conditions for those events are available at deterministic instances of time, to transit the synchronous component from one configuration or marking to another.

- Seq-execution - This means only one trigger condition is sufficient to execute an event to transit the synchronous component from one configuration or marking to another.

- Max-execution - This means that in each step a maximal multiset of events, denoting the availability all the trigger conditions for the events, must be executed to transit the synchronous component from one configuration or marking to another.

Therefore, from the restrictions on the input/output transitions of a synchronous system, it can be derived that the input and output transitions can follow any of the above execution rules.

The individual compartments, depicted in Figure 6.15, can now be viewed as a modular synchronous block with its own input and output signals.Therefore, each of these compartments will have to follow the synchronous behaviour.

Therefore, the original synchronous system can be now defined as a collection of these compartments, modelled at the PN level by a standard operation of a union of PNs, merged on places [90]:

$$\Sigma \quad = \quad (P_1, T_1, F_1, \mu_1) \cup ...(P_n, T_n, F_n, \mu_n)$$

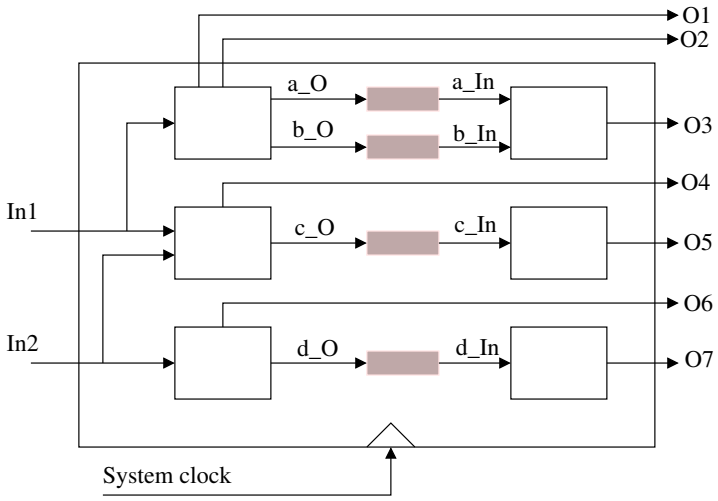


Figure 6.15: System architecture with storage units

Since each of these compartments are viewed as synchronous blocks, the input and the output transitions from these blocks also follow the same execution rules as discussed above.

6.5 Petri nets with localities

In order to model a distributed architecture from a synchronous system model, we apply the theory of Petri net with Localities which was originally introduced in [87]. In the previous work, the co-located transitions executed maximally and in persistent steps only. We extend this by making a distinction between input and output transitions and allowing the input transitions to execute as and when they arrive and restricting the output transitions to execute maximally. This extension is in direct relation to the synchronous behaviour, discussed in the previous sections.

The transitions in the PT-net belong to a fixed unique locality. The allocation of localities to the transitions is achieved by partitioning the PT net using a locality mapping function γ. This means if two transitions return the same value for γ they will be co-located.

A PN with *localities* is a tuple denoted by $NL = (P, T, F, \mu_0, \gamma)$, where the underlying PN is denoted by ${}_{UND}(NL) = (P, T, F, \mu_0)$ and $\gamma : T \rightarrow N$ is the location mapping for the transition set T. $\gamma(t)$ returns an integer value which denotes the locality of the transition t. Initially, for all $t \in T$, $\gamma(t)$ is set to 0, which denotes that the transition is unallocated.

A net can be partitioned into localities giving rise to the formation of smaller nets that constitute the original graph.

Definition 6.5. Let $\Sigma = \{P, T, F, \mu_0\}$ be an elementary net system. Then, the localisation leads to the division of the net into n smaller nets, denoted by,

$$\Sigma_i = (P_i, T_i, F \cap (P_i \times T_i \cup T_i \times P_i), P_i \lhd \mu_0),$$

for $i = 1$ to n, where n is a set of integers, each $T_i \subseteq T$ so that $(T_1 \cap T_2 \cap ....T_n) \neq$ Ø and each $P_i \subseteq P$ so that $(P_1 \cap P_2 \cap ....P_n) \neq$ Ø, $P_i \lhd \mu_0$ is defined by the following:

If $\mu_0 : P \rightarrow \{0, 1\}$, then $\forall p \in P_i$, $\mu_{0i} : P_i \rightarrow \{0, 1\} | \mu_{0i}(p) = \mu_0(p)$.

Synchronous components versus localities In order to map the synchronous components into localities, the execution modes of the synchronous counterpart

should be supported by the localities thus formed. Therefore, the three execution modes are redefined which are incorporated to support the specification of synchronous behaviour.

Definition 6.6. Free-enabled

A multiset of transitions U is free-enabled at a marking μ, if $\mu \geq pre_N(U)$. This is denoted by $\mu[U >$.

Definition 6.7. Seq-enabled

A multiset of transitions U is seq-enabled at a marking μ, if U is free-enabled at marking μ and $|U| = 1$.

Definition 6.8. Max-enabled

A multiset of transitions U is max-enabled at a marking μ, if U is free-enabled at marking μ and there is no transition t such that $\mu[U + \{t\} >$.

Therefore, it can be seen that each locality is able to emulate the behaviour of the synchronous components, giving rise to semantic preservation when synchronous components are mapped onto the localities.

The next section presents some rules for the allocation of localities in a synchronous system.

6.6 Notion of partitioning correctness

As discussed above, a synchronous system can be desynchronised into a distributed architecture by unbundling the inputs and forming localities. The forma-

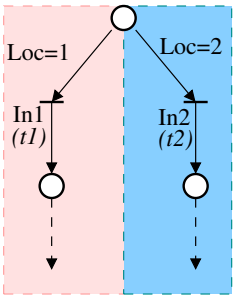
tion of these localities should satisfy some correctness properties to ensure correct desynchronisation. The partitioning of the GALS deployment of a synchronous system is correct w.r.t the original synchronous system if there is a behavioural equivalence between the GALS system and the initial synchronous specification.

This is formally defined in the following way:

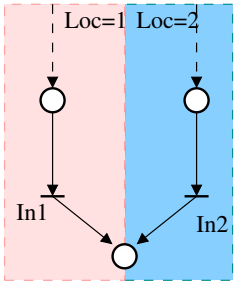
Definition 6.9. Let $\Sigma = \{P, T, F, \mu_0\}$ be an elementary net system. The partitioning $\Sigma = \Sigma_1 \cup \Sigma_2 \cup ....\Sigma_n$, each belong to localities $L_1, L_2...L_n$, respectively, is correct at a marking μ iff for all steps of transitions $U_1 \subseteq T_1$,..$U_n \subseteq T_n$, where $U_1, ...U_n$ are enabled in $\Sigma_1, ...\Sigma_n$, respectively, the combined step $U_1 \cup U_2 \cup ...U_n$ is enabled in Σ. This denoted as,

$$(\mu \lhd P_1)[U_1 >_{\Sigma_1} \wedge (\mu \lhd P_2)[U_2 >_{\Sigma_2} \wedge ...(\mu \lhd P_n)[U_n >_{\Sigma_n} \Rightarrow \mu[U_1 \cup U_2 \cup ...U_n >_{\Sigma},$$

for all $U_1 \subseteq T_1, U_2 \subseteq T_2..., U_n \subseteq T_n$.



(a) Conflict-choice place



(b) conflict-merge place

Figure 6.16: Conflict between transitions

Conflict Resolution In order to adhere to the above criterion, the locality allocation should satisfy correctness properties for *conflict resolution.* For example, an incorrect partition is shown in Figure 6.16(a). The net Σ is partitioned into Σ_1 and Σ_2 belonging to localities L_1 and L_2, respectively, so that the transition t_1 is allocated to locality L_1 and t_2 is allocated to L_2 and therefore, $T_1 = t_1$ and $T_2 = t_2$. Now, substituting p for μ, leads to markings $\{p\}[\{t_1\} >_{\Sigma_1}$ and $\{p\}[\{t_2\} >_{\Sigma_2}$ in each of the localities but $\{p\}[\{t_1, t_2\} >_{\Sigma}$ is not true. Hence, the partitioning is incorrect. Such an occurrence that leads to an incorrect partition can be similarly shown for Figure 6.16(b).

The notion imposes the transitions in conflict to be placed in the same locality. The locality optimisation technique can lead to occurrence of such a situation. Hence, care must be taken while inserting the input/output bridges in the partitions. Therefore, the correctness can be guaranteed if the following criteria is satisfied:

Criterion 6.1. *Let $\Sigma = \{P, T, F, \mu_0\}$ be an elementary net system that has been partitioned into $\Sigma_1, \Sigma_2, ..., \Sigma_n$. If transitions from the partitions do not share preconditions or postconditions , or*

$$\bullet T_1 \cap \bullet T_2 \cap ... \bullet T_n = T_1 \bullet \cap T_2 \bullet \cap ... T_n \bullet = \emptyset$$

then the partitioning is correct.

Step Persistency Another correctness property that the partitioned blocks must satisfy is *Step-persistency.* The reason for identifying and handling non-persistency

is already presented in Section 6.3. The non persistent transitions can be identified and made persistent by executing the following steps.

1. For each output transition in the net, identify the set Out of output transitions that are dependant on more than one input transition.
2. for each output transition in Out, return the set In of input transitions, on which the output depends.
3. For each input in In, check if it causes more than one output transition.
4. Return the set $persist$ of input signals for which (3) is true.
5. Return the output transition $O1$, such that $O1 \in Out$ and the input that causes it belongs to the set $persist$.
6. Connect the output obtained in (5) with each of the input signals in the set $persist$ through an output-input transition pair as exemplified in Figure 6.18.

The signals that are inserted are sets of output-input transition pairs, denoted by Ox and Ix, which behave as internal or silent events for the overall system. These signals satisfy the notion of validity of signal insertion discussed in Section 6.2. Adding extra signalisation does not introduce any new behaviour in the system. This is because the signals are added as pairs of input/output transitions and emulate a buffer which only introduces some extra delays in the system without affecting the consistency of the signals. These signals aid the formation of localities which have all inputs arriving before the emission of all the outputs.

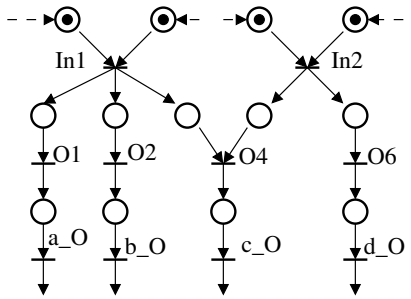


Figure 6.17: Persistency check

The above is exemplified by taking Figure 6.17 into consideration. Since $In1$ and $In2$ can arrive with unknown delays, the model needs to be modified to remove non-persistency as described in this section. From (1) we obtain the set $Out := O4$ as it is the output transition that is dependant on more than one input transition. From (2) set In returns $\{In1, In2\}$ which cause the output transition $O4$. For each input $In1$ and $In2$, step (3) returns $true$, since both the inputs cause more than one output signals. Therefore, the set $persist$ in step (4) returns $\{In1, In2\}$. Step (5) returns $O4$. The additional signals, in the form of output-input transition pair, are inserted between $In1$ and $O4$ and between $In2$ and $O4$.

Therefore, at the model level, the system depicted in Figure 6.12 is transformed into the system depicted in Fig. 6.18. The block level representation of the part of the circuit, under consideration, is depicted in Figure 6.19.

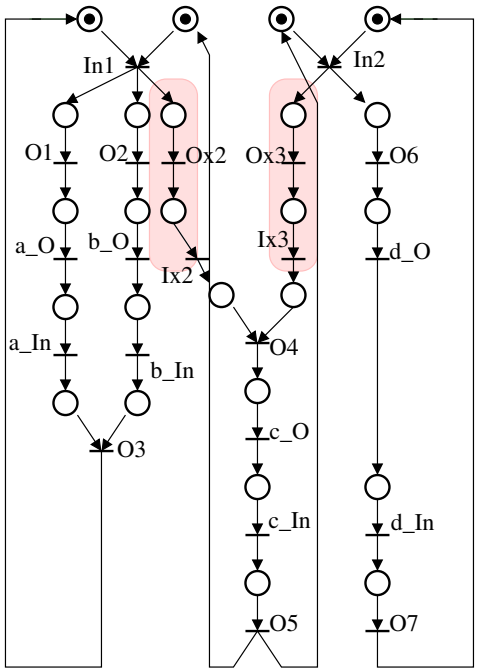


Figure 6.18: Bridge formation

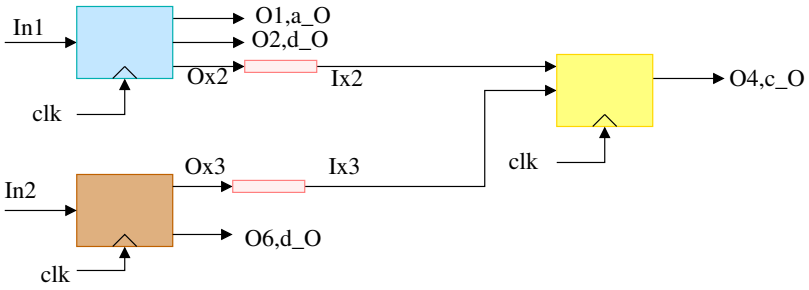


Figure 6.19: Block level representation

6.7 Rules for Locality Allocation

Taking the partitioning constraints, obtained from the correctness properties in the previous section into consideration, we present some rules for the allocation

of localities. This leads to correct localisation of transitions. The method of deployment of synchronous systems over localities requires the adherence to the following rules:

1. In order to obtain the partition sets of transitions, the information about the locations of each input and output transition of the PT-net is required. We derive the localisation of each input and output transition of the synchronous circuit from the input output dependencies. For example, if output transition y is computed in locality L, then so does the input signal, x in this case, that are required for the execution of y. Therefore, the input x must also be located in L. Such a localisation will directly influence the localisation of internal signals.

2. When allocating localities, all the branches of the choice, i.e. all the transitions of the choice place should be placed in the same locality. Therefore, two transitions in conflict should not be placed in two different localities. Violation of this property leads to an erroneous locality allocation for transitions, discussed earlier in Section 6.6.

3. In each locality, all the transitions should satisfy the persistency constraint. If the constraint is violated, additional signals are required to be inserted to eliminate non-persistency. This is elaborated in Section 6.6.

Formally, the above rules lead to the formation of a system that can be defined by the following definition:

Definition 6.10. Let $\Sigma = \{P, T, F, \mu_0\}$ be an elementary net system. Then, the partitioning leads to the division of the net into n smaller nets, denoted by

$$\Sigma_i = (P_i, T_i, F \cap (P_i \times T_i \cup T_i \times P_i), \mu_0 \lhd P_i),$$

for $i = 1$ to n, where n is a set of integers, each $T_i \subseteq T$ so that $(T_1 \cap T_2 \cap ....T_n) =$ Ø and each $P_i \subseteq P$ so that $(P_1 \cap P_2 \cap ....P_n) \neq$ Ø.

These rules lead to a correct localisation of input/output transitions.

6.8 Allocation of Localities

This work does not address the problem of finding the optimum localisation of the computations w.r.t the performances of the resulting distributed system. The localisation of all the actions of the synchronous system is derived directly from the localisation of the input and output signals. This section also presents an optimisation for the locality allocation methodology by redistributing transitions over localities to avoid locality overloading arising from large input fan outs.

6.8.1 Algorithm for locality allocation

In order to allocate localities to the transitions of a system, we require to define some methods which are presented in Algorithm 1. The algorithm implements a locality allocation scheme that is in accordance with the rules presented in the previous section 6.7.

The algorithm described in this section incorporates a bi-directional subnet traversal in order to allocate localities to the transitions it visits. It takes as input a Petri net model of a synchronous system denoted by Σ. The output of the algorithm is a Petri net model of the synchronous system Σ, with locality information added to each transition in the model.

This algorithm defines the following methods and the functionality of each is described below:

Tr_f: This function denotes the forward subnet traversal. This function traverses the net and adds the transitions to sets T_v and $Tail$, depending on certain conditions. Once the sources have been identified, for each element of the set the net is traversed forward, assigning each transition visited to the set T_v. If during the traversal a transition is reached, which already belongs to the the set T_v due to a previous net traversal, the traversal is terminated at this node and this transition is added to the set $Tail$. $Tail$ denotes the node where a traversal stops and marks the boundary for a locality. The $Tail$, in contrary to $Source$, forms the output interface for a particular locality. While traversing the net forward, if a transition is reached that belongs to $Source$, then the predecessor transition(s) is added to the set $Tail$.

Alloc_Tail: Once the set of tail transitions have been identified. This function takes the set $Tail$, as a parameter and assigns localities to all the tails of a given source.

Tr_b: This function defines a backward net traversal starting at the tail and terminating at the source. This function finds a set of unallocated transitions on

the backward path to the source(s). If a node that belongs to T_v or $Source$ is reached which is already allocated, the locality value is not changed. Therefore, the node retains its original locality, if such a situation arises.

Assign_Loc: This function takes a set of transitions and allocates localities to them depending on the locality values of the *Tail* transitions.

Method:

1. A set of $Source$ is identified which consists of all input transitions of the net.

2. For all the transitions belonging to $Source$, the following steps are executed:

   - Traverse the net forward assigning each transition to the sets T_v or $Tail$.
   - Allocate localities to all the tails.
   - Traverse backward starting from the tail and terminating at the source assigning all the unallocated transitions to the set $LocAssign$, in its path.
   - Finally allocate the transitions in the set $LocAssign$ with the locality of the $tail$ transition chosen from the set $Tail_{new}$, which is a set of assigned $tail$ transitions.

Finally, we obtain a set of all transitions of the net, each of which is allocated to at most one locality in the system.

Algorithm 1 Allocation of Localities

function: Allocate_Localities
input: $\Sigma = \{P, T, F, \mu_0\}$
output: $\Sigma_{out} = \{P, T, F, \mu_0, L\}$
$T_v := \emptyset$; $Tail := \emptyset$; $Tail_{new} := \emptyset$; $LocAssign := \emptyset$
for each $t \in T$
  $\gamma(t) = 0$
$Source \leftarrow$ set of all inputs of the system
$n \in N \leftarrow$set of natural numbers
$n = 0$
for each $t_{in} \in Source$ **do**
  if $t_{in} \notin T_v$
    $tr\_f(t_{in})$
      for each $t_{tail} \in Tail$ **do**
        $AllocTail(t_{tail})$
      for each $t_{tail} \in Tail$ **do**
        $tr\_b(t_{tail})$
      choose $t \in Tail_{new}$
      for each $t_{ua} \in LocAssign$
        $\gamma(t_{ua}) = \gamma(t)$

Forward net traversal Once the set of sources is identified, these form the nodes of forward net traversal. In this method, the transitions are assigned to sets T_v (a set of visited transitions) and $Tail$ (a set of tail transitions) depending on the following conditions:

T_v: if the transition passed as a parameter, it is added to the set of visited transitions.

$Tail$: when the transition does not belong to the set $Sources$ and the successor transition belongs to the set $Sources$, then it is added to the set of tail transitions.

The traversal is terminated for a given path when the tail is identified. This method is repeated until all the the tail transitions, for a given source, are identi-

Algorithm 2 Forward Net Traversal

function: tr_f(t)
input: $\Sigma = (P, T, F, \mu_0)$, $Sources$
output: $\Sigma = (P, T, F, \mu_0)$, $T_v \subseteq T$, $Tail \subseteq T$
for each $p \in t\bullet$ **do**
  for each $t_{succ} \in p\bullet$ **do**
    $T_v := T_v \cup t$
    if ($t_{succ} \notin T_v$) **then**
      $T_v := T_v \cup t_{succ}$
    else $Tail := Tail \cup t_{succ}$
    if $(t \notin Sources \,\&\&\, t_{succ} \in Sources)$
      $Tail := Tail \cup t$
    if ($t \notin Tail$)
      if($t_{succ} \notin Tail$)
        tr_f(t_{succ})

fied.

Allocation of localities to tail transitions The previous method identifies a set of tail transitions that could be allocated or unallocated. In this method, all the allocated and unallocated transitions are identified and assigned to sets $Alloc$ and $Nalloc$, respectively. If the $Alloc$ set is null, denoting that all the tail transitions are unallocated, we assign an integer value to all the tail transitions. If $Alloc$ is not null, a tail is randomly chosen (this is because all the tail transitions in the set will have the same locality) from the set and its locality is assigned to all the other tail transitions in the set $Nalloc$. This methodology assigns the same locality values to all the sources that form presets of a set of output transitions.

Backward net traversal In this method, for each of the tail transitions obtained from Algorithm 3 which are an element of the set $Tail_{new}$, the net is traversed in

Algorithm 3 Locality Allocation for Tail Transitions

function: AllocTail($Tail$)
input: $Tail \subseteq T$
output: $Tail_{new} \subseteq T$
$Alloc = \emptyset$
$Nalloc = \emptyset$
for each $t \in Tail$
  if ($\gamma(t) \neq \emptyset$) **then**
    $Alloc := Alloc \cup t$
    else $Nalloc := Nalloc \cup t$
if ($Alloc == \emptyset$) **then**
  $n = n + 1$
  for each $t \in Nalloc$
    $\gamma(t) := n$
    $Tail_{new} := Tail_{new} \cup t$
else
  choose $t_{alloc} \in Alloc$
  for each $t \in Nalloc$
    $\gamma(t) := \gamma(t_{alloc})$
    $Tail_{new} := Tail_{new} \cup t$

the backward direction to all the unallocated sources that are reached on the backward traversal path. In the path, it assigns all the transitions that are unallocated to a set called $LocAssign$. When the traversal reaches a source that does not belong to the set T_v, the transition is assigned to the set T_v, so that this transition is not need to be dealt with anymore since it has already been allocated a locality owing to another backward traversal.

6.8.2 A simple Example

1. Consider a complete model of a synchronous system Σ in the form shown in Figure 6.12. This system is partitioned into localities by following the

Algorithm 4 Backward Net Traversal

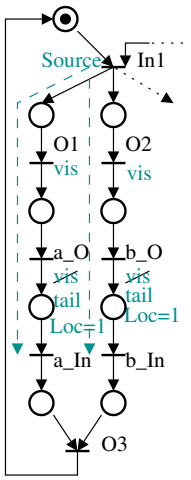
function tr_b(t)
input: $\Sigma = (P, T, F, \mu_0)$
output: $\Sigma = (P, T, F, \mu_0)$**,** $LocAssign \subseteq T$
for each $p \in \bullet t$ **do**
  for each $t_{pred} \in \bullet p$ **do**
    if ($\gamma(t_{pred}) == 0$)
      $LocAssign := LocAssign \cup t_{pred}$
    if ($t_{pred} \in Source$)
      $T_v := T_v \cup t_{pred}$
    else $tr\_b(t_{pred})$

steps of the algorithm. The steps are elaborated by applying the rules on the system model.

2. Let $Source$ be a set of all input signals that are sent to each internal computational block in the system Σ. It is represented by $Source = \{In1, In2, In3, In4, In5, In6\}$.

3. For each input transition $t_{in} \in Source$, the net is traversed forward, adding each transition visited, to the set T_v, including the source transition t_{in}. For example, in the forward traversal along the first branch for input $In1$, the transitions that are assigned to the set T_v are $In1$, $O1$ and a_O.

4. The forward traversal assigns transition t to a set $Tail$ if the transition is such that its successor transition, denoted by $t_{succ} \in p\bullet$, where $p \in t\bullet$, belongs to the set $Source$. This is exemplified in Figure 6.20.



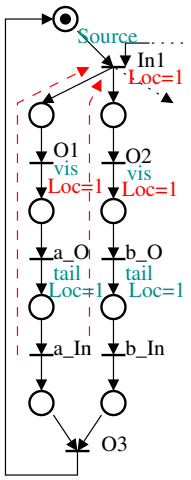
(a) Forward Traversal

Figure 6.20: Net Traversal

5. This procedure is repeated for each branch of the initial input source. This is depicted in Figure 6.20.

6. Once all the tails have been identified for a given root, we assign the same locality value to all the tails. For example in Figure 6.20, transitions labelled a_O and b_O are assigned to the same locality 1.

8. For each tail, we traverse backward and assign the same locality value to all the transitions till all the possible backward paths are assigned a locality, each path terminating at the transition that belong to the set $Source$. This is shown in Figure 6.21. On the backward traversal path allocation continues until all the sources in its path have been allocated.



(a) Backward Traversal

Figure 6.21: Backward net traversal

10. The above steps are repeated for all the inputs in $Source$.

The above steps lead to the partition of the system into localities. The partitioned system is shown in Figure 6.22. The places shared by the localities depict the storage units that form the interface between two localities. Table 6.1 shows the different localities and the input and the output signals assigned to each locality.

To guarantee partitioning correctness as discussed in Section 6.6, the net is checked for *step-persistency* before the locality allocation algorithm is applied. Relevant signals are inserted if there is any persistency violation.

The partitioning algorithm presented, also satisfies the correctness criterion, namely, *conflict-resolution.* Adherence to *conflict-resolution* is exemplified in Figure 6.23. As can be seen from the figure, the transitions in conflict, namely

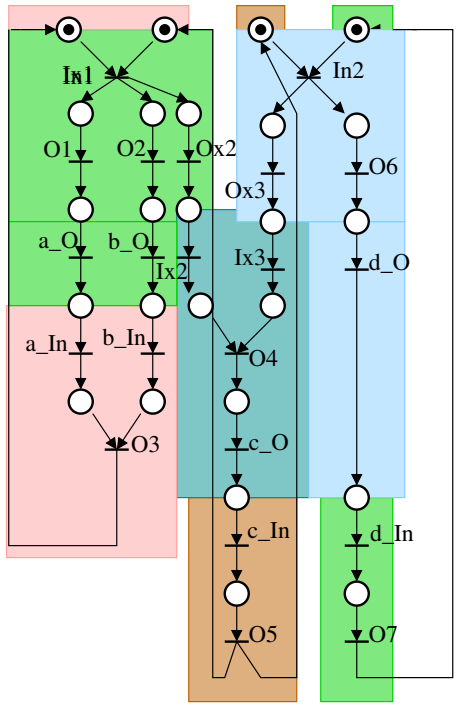


Figure 6.22: Partition Optimisation

a_O and b_O are placed in the same locality. Since *choice places* are contained inside each locality, it can be easily seen that the algorithm also contains the *merge places* inside the localities, owing to the input/output dependency notion used to derive the locality formation.

6.9 Locality Optimisation

Let us consider the system shown in Figure 6.24. The rules of locality allocation allocates localities based on input output dependencies. Therefore, large input fanouts could lead to the formation of large localities. In order to avoid the

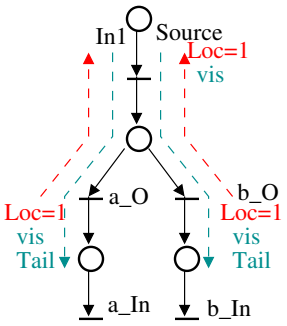


Figure 6.23: Locality allocation for conflicts

Locality	Inputs	Outputs
$L1$	$In1$	$O1, O2, a\_O, b\_O, Ox2$
$L2$	a_In, b_In	$O3$
$L3$	$In2$	$Ox3, O6, d\_O$
$L4$	$Ix2, Ix3$	$O4, c\_O$
$L5$	c_In	$O5$
$L6$	d_In	$O7$

Table 6.1: I/O allocation to localities

overloading, the fanouts can be partitioned so that they are allocated to different localities. We do not present an optimum criterion of obtaining as many or as few as possible localities, because it is system dependant. Therefore, depending on a system requirement, large localities can be further partitioned into smaller localities to avoid overloading.

This can be done by introducing extra signalisation in the path of the fan-out that requires segregation. A manual signal introduction technique is introduced in this section which is applied to the original un-partitioned system. The signals that are inserted are sets of output-input transition pairs, denoted by Ox and Ix,

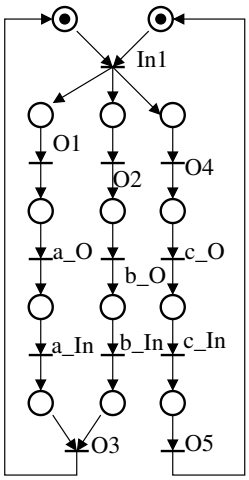


Figure 6.24: Input fan-outs

which behave as internal or silent events for the overall system. Such an insertion is depicted in Figure 6.25. These signals abide by the notion of validity of signal insertion discussed in Section 6.4.1. The output signal Ox can only be enabled by $In1$, which requires one or more of its fan-out transition to be relocated to another locality. The transition Ox is followed by Ix which in turn enables the outputs, which were originally activated by $In1$. Hence, these signals enforces the formation of extra localities, in turn reducing the load on any one locality. The algorithm presented in the previous section can now be applied to this transformed system. When the algorithm is applied the following is obtained:

- The transition Ox belongs to the same locality as $In1$.
- The newly inserted input signal Ix and the output transitions that follow along the path are assigned to a new locality, say X.

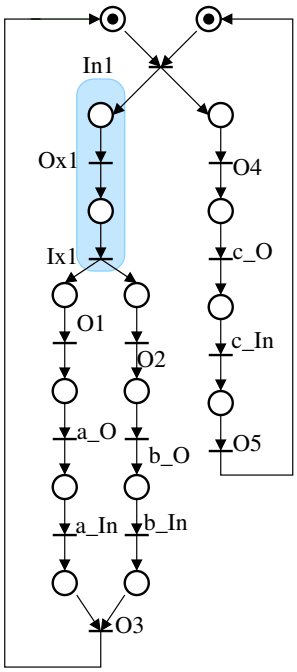


Figure 6.25: Bridge formation for fanouts

- If these output signals require other source signals, besides $In1$, for their activation, then these sources are also located in the same locality and so are the output transitions that are dependant on these sources.

Adding extra signalisation does not introduce any new behaviour in the system. This is because the signals are added in pairs and act as buffers which only introduce extra delays in the system without affecting the consistency of the signals. While inserting the signal it has to be made sure that the choice places are not split. The choice branches should be contained in the same locality, as discussed earlier.

6.10 GALSification

Each of the localities formed have a clock signal that activates that locality. In a synchronous system this clock is the global clock which is sent to each of the localities. At every clock edge new data is read from the input signal ports. These inputs can arrive from other localities or the environment. Due to the delays in the wire, the inputs may arrive at different times.

Due to this phenomenon the order of inputs is not guaranteed. On the contrary, a clock edge triggers all the active input and output transitions maximally.

If the input maximal steps are unbundled and the restrictions on the input and the output transitions, based on the correctness properties, are guaranteed then the global clock which samples these input and output transitions, is no longer a requirement at the locality interfaces. Therefore, we can eliminate the global clock and substitute each locality with local clocks. Therefore, each locality behaves like an independent synchronous system which communicates asynchronously with other localities.

The signals are communicated from one locality to another through an asynchronous domain. Hence, the actions performed in this domain are causal. Hence, the localities have to synchronise with each other while sending and receiving data with the receiver and sender blocks, respectively. This leads to the formation of a GALS architecture.

Therefore, we can deduce that if the I/O conditions are satisfied the globally synchronous system can be translated into a GALS system. Hence, to enforce

the I/O conditions in each locality, we impose that the inputs are interleaved to guarantee correctness by allowing them to be received as and when they arrive and outputs are maximal, i.e, if they are active they will fire concurrently. Hence, we implement the principle of maximal output or Max-O semantics which is obtained by substituting max-enabled transitions of each locality to the outputs generated from them.

6.11 Implementation of clock control

Until now, the clock signal in the net has been represented implicitly. The transitions were coupled with the notion of clock. Therefore, the activation of a transition signified that the clock's positive edge for positive logic or negative edge for negative logic, is also active. Firing of the transition activates the negative edge for positive logic and positive edge for negative logic. On the contrary, a clock control circuit is required when local clocks are deployed to the individual localities. This clock control circuit synchronises the signals while crossing the domain from one locality to another.

Hence, the clock transitions need to be shown explicitly to enable this control. Therefore, instead of coupling transitions with the clock information, we introduce explicit clock transitions to signify the positive/negative edges. The formation of the localities enables the treatment of each locality as a black box. Therefore, the process of reading the inputs and the producing the outputs is not dealt with. It is considered that the inputs are read and the outputs produced in one

clock cycle. For the sake of simplicity, in the examples the operations are shown to be completed in one clock cycle.

To show the clock control, the signals which denote the availability of inputs and outputs are explicitly depicted. Since the signals travel from one locality to another, which belong to different clock domains, the transfer from the clocked domain to the unclocked domain should occur when the clock is inactive to avoid metastability. The clock should not be triggered until this transfer is completed. When the transfer is completed the clock can be triggered again to process another set of data. A similar process occurs when a signal is transfered from an unclocked to a clocked domain in the form of inputs. When all the input signals have been received which signifies the availability of input data, the clock is allowed to go high to process the data and produce relevant outputs. The above phenomenon elaborates the working of the clock control architecture in the deployment of a synchronous system into a GALS system. This is exemplified in Fig. 6.26. Note that this net uses double headed arcs (sometimes called read-arcs).

The green arrows depict the operations of the net that triggers the positive edge of the clock. For the input $In1$ to be received, the clock requires to be inactive or low to avoid metastability (see Chapter 2). On the reception of the signal, the clock is triggered to process the data. The completion of the process is denoted by the *completion detection* signal CD. When signal CD is received, the clock is lowered for the transfer of outputs.

Such transformations are applied to all the localities to handle local clocks. The insertion of the signal CD does not affect the behaviour of the system as it

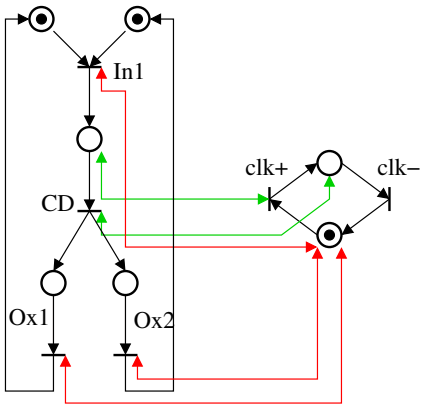


Figure 6.26: Clock Control

is an *internal* signal and it adheres to the notion of validity of signal insertions discussed in Section 6.2. It is inserted after the reception of the input signal and therefore does not delay the input signals.

In a similar way, such implementations are obtained for all the localities. The final model has inputs going in and outputs coming out of each locality to be communicated to the other localities which is done through asynchronous FIFOs. The FIFOs are the interface places that act as storage units for the system. Finally, a system is obtained with localised clocks which communicate with each other using asynchronous FIFOs.

6.12 Summary

This chapter addressed the problem of synthesising delay insensitive wrappers for GALS implementation. The methodology used Petri nets to model the syn-

chronous system interfaces, thus reducing the complexity of the component models posed by the previous methodology presented in Chapter 5. This work overcomes the concurrency problems through the use of Petri nets. It presented a formalism for desynchronising globally synchronous system while preserving the behavioural correctness and semantics of the original system. The method incorporates the idea of *Localities,* to formally define the partitioning of the synchronous systems. The partitioning technique takes into account the *persistency* of signals which need to be satisfied in each partitioned block or locality. This also aids the use of a data driven clocking scheme as the chosen local clock control scheme. This is because, in order to satisfy the persistency property, each individual block waits for all the inputs to arrive before starting the clock. Therefore, like the previous, power is only consumed when all the data for a particular computation has been received.

This work has improved on the previous methods of converting existing synchronous circuits into a GALS architecture. This chapter presented a notion of correctness of the partitiong technique. This has led to the definition of the necessary conditions for correctness. The algorithm for system partitioning presented, offered a faster and more efficient route to the synthesis of the asynchronous wrappers while preserving the IO behaviour of the synchronous systems.

Chapter 7

Conclusion

This thesis has drawn together two relatively disjoint areas of GALS research in order to address important issues in the field of GALS design methodologies. These include GALS integration techniques on the one hand, and system desynchronisation techniques for GALS deployment, on the other. In this chapter we summarise the main results obtained from the research. In the area of system integration, this thesis presents a model and system level design technique that aids performance and power analysis. We studied the interplay among effects of communication latency, clock pause and FIFO insertions. In the area of desynchronisation, we proposed a new synthesis framework for weakly endochronous system and gave a new formalization for the desynchronisation of a globally synchronous system into a GALS architecture and we studied the interplay among the concepts of event absence, event sampling, and communication latency in modeling distributed concurrent systems.

7.1 Summary of contribution

The intermediate models used for the design and synthesis of GALS architecture are Petri nets and state transition systems. Due to their ability to exhibit advanced concurrency, Petri nets are ideal for modeling distributed concurrent systems. On the other hand state transition systems are well suited for specifying clocked systems. Therefore, chapter 2 gave the necessary background for GALS systems, Petri net and state transition system modeling.

Chapter 3 reviewed a set of existing methodologies in the relevant areas of GALS design. This chapter also presented the concept of endochrony and weak endochrony. This forms the basis of the proposed synthesis methodology and motivation for the introduction of the new desynchronisation formalism.

Chapter 4 presented the classification of different clocking schemes for Globally Asynchronous and Locally Synchronous architectures. These schemes have been modelled using Petri nets. A Petri net model of these interconnect architectures, allows the designer to use existing logic synthesis tools, like Petrify to obtain gate level design solutions. Such solutions for GALS systems with stretchable and data driven clocking schemes have been presented in this thesis. All the three clocking schemes exhibited reliable data transfer between the synchronous domains. A complex SoC can exploit any of the above given architectures depending on the requirements of the target system. These models can be plugged into existing partitioned synchronous blocks. These schemes can be extended to employ various power reduction methodologies in the wrapper without affecting

the synchronous *IP* blocks.

In addition to the classification and design solutions for the three clocking schemes this thesis also analysed the three systems on performance and power consumption criteria. Stretchable and data driven clocking schemes demonstrated higher throughput and lower power consumption characteristics, respectively, compared to the prevalent pausible clocking scheme. The stretchable and pausible clocking schemes were further compared on two other metrics, namely, the number of times the clock is paused or stretched and the total latency incurred by these pauses. Such an analysis aids the designer to make different design decisions based on power and performance.

Chapter 5 sets the guidelines for a new methodology for the synthesis of the delay-insensitive asynchronous wrappers needed for the correct-by-construction GALS implementation of a modular synchronous system. The approach is based on the results presented in [69], which define high-level, decidable criteria for the correct GALS implementation of modular synchronous specification, namely the weak endochrony of the modules and the absence of deadlocks in the global synchronous specification. The synthesis problem is thus reduced to that of synthesising the asynchronous wrappers for weakly endochronous synchronous modules. This problem can be solved on a local basis, without knowledge about the properties of the global system.

We use as an example a model of a DLX-like processor to intuitively present and give implementation steps on the different phases of the proposed methodology. This work has been presented in [91].

Chapter 6 addressed the problem of synthesising a GALS system by a desynchronisation methodology which employed PN as its model of abstraction. The granularity of desynchronised systems thus constructed using PNs is smaller than the ones obtained from the previous method [91] and thus is easier to automate and apply even for large complex circuits. The GALS system is obtained by applying the theory of localities to a synchronous system model preserving the synchronous properties of the input output signals. This chapter also defined two behaviour preserving transformations, namely, signal insertion and localisation of transitions used at different stages of the desynchronisation process. As a result this chapter presented a desynchronisation methodology with a relatively clear route to automated synthesis, preserving the IO behaviour of the synchronous systems.

7.2 Future Work

Chapter 4 presented a comparative study of of the three classes of clocking schemes based on performance and power consumption with static frequency variation. Dynamically changing the frequency and the supply voltage for sub-blocks to reduce power consumption has been successfully implemented for high performance micro-processors[92]. They are called Dynamic Voltage and Frequency Scaling (DVFS) methods. Since microprocessors are sources of increased power consumption, these methods are very desirable for the reduction of power consumption. GALS design offers many advantages that could be exploited to realise

DVFS systems. The main characteristic being that it allows the individual blocks to be clocked independently. Therefore, it is possible to extend this idea to support different voltages as well. This method could be applied to each of the clocking schemes and study the change of the analysis results.

In a GALS system like the ones presented in chapter 4, which pause the clock until data transfer is completed, both the *Req-Ack* signal pairs can be used to determine how active a module is. If the environment is faster than the module, then by increasing the operating frequency and voltage of the module could result in an increased throughput. On the other hand, if the module is faster than the environment, the supply voltage of the module and the operating frequency of the local clock can be reduced to save energy.

Chapters 5 and 6 presented sound formalisms for the translation of WE systems into synthesisable Petri net models and desynchronisation of globally synchronous systems for GALS deployment, respectively. These algorithms need to be automated to reduce design time and designer intervention.

The state transition models obtained by applying the idea of WE are complex for practical examples due to the weak handling of concurrency. Therefore, one possible extension is to extend the underlying theory in order to simplify the generated logic by taking into consideration:

- closed-system assumptions, for instance under the form of sequential care sets
- the fact that synchronous specifications are often meant to run in asyn-

chronous environments, under specific input arrival hypothesis

The locality allocation techniques presented in chapter 6 can be further optimised to minimise interconnection between localities, yet still containing the choices between signals in the same locality to guarantee correctness, as previously discussed. The protocol used in the proposed algorithm, can be optimised to increase the component speed. We would also take into consideration that the system may require more than one clock cycle for a particular computation. Hence, a counter can be introduced to count the number of clock cycles required for a computation and allow the clock to tick for the required number of cycles while preserving the behaviour of the system.

Bibliography

[1] International technology roadmap for semiconductors. 2003.

[2] Medea+, design automation roadmap. March 2002.

[3] S. Naffziger. The implementation of a 2-core multi-threaded itanium family processor. In *Proceedings of ISSCC*, 2005.

[4] A. Benveniste, B. Caillaud, and P. L. Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125–171.

[5] A. Benveniste, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. *Lecture Notes in Computer Science*, 2855:35–50, 2003.

[6] J. P. Talpin, A. Benveniste, B. Caillaud, and P. L. Guernic. *Hierachical normal form for desynchronization*. Technical Report 3822, IRISA, 1999.

[7] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in soc design. *IEEE Micro*, 22(5):24–35.

[8] J. Sparsø and S.B. Furber. *Principles of asynchronous circuit design - A systems perspective*. Kluwer Academic Publishers, 2001.

[9] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[10] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple. Amulet1: an asynchronous arm microprocessor. *IEEE Transactions on Computers*, 46(4):385–398, April 1997.

[11] C. Seitz. *System Timing, Chapter 7 of Introduction to VLSI Systems by C. Mead and L. Conway*. 1980.

[12] W. J. Bainbridge and S. B. Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proceedings of 7th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 118–126, 2001.

[13] D. Chapiro. *Globally-Asynchronous and Locally-Synchronous Systems*. Stanford University, Report No. STAN-84-1026, 1984.

[14] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous and locally synchronous design style. In *Proceedings of Design Automation Conference*, pages 873–878.

[15] T. Villiger, H. Kaslin, F. K. Gurkaynak, S. Oetiker, and W. Fichtner. Self-timed ring for globally-asynchronous locally-synchronous systems. In *Pro-*

ceedings of the Ninth International Symposium on Asynchronous Circuits and Systems (ASYNC), 2003.

[16] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, 1987.

[17] L. Y. Rosenblum and A. Yakovlev. Signal graphs: From self-timed to timed ones. In *Proceedings of the International Workshop on Timed Petri Nets*, pages 199–206, 1985.

[18] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactionson Computer-Aided Design*, 14(1):61–86, January 1995.

[19] L. Lavagno, K. Keutzer, and A . Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 302–308.

[20] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 568–572, 1992.

[21] C. W. Moon. *Synthesis and Verification of Asynchronous Circuits from Graphical Specications*. PhD thesis, U. C. Berkeley, 1992.

[22] P. Vanbekbergen, G. Goossens, F. Catthoor, and H. J. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifica-

tions. *IEEE Transactions on Computer-Aided Design*, 11(11):1426–1438, November 1992.

[23] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identication of self-timed circuits. *Proceedings of Formal Methods in System Design*, 4(1):33–75, 1994.

[24] A. Kondratyev, J. Cortadela, M. Kishinevsky, E. Pastor, O. Roigand, and A. Yakovlev. Checking signal transition graph implementability by symbolic bdd transversal. In *Proceedings of The European Design and Test Conference*, pages 325–332, 1995.

[25] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, U. C. Berkeley, 1992.

[26] E. Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs*. PhD Thesis, Universidad Politecnica de Cataluna, 1996.

[27] A. Kondratyev and A. Taubin. Verification of speed-independent circuits by stg unfoldings. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75, 1994.

[28] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. An efficient heuristic procedure for solving the state assignment problem for event-based specification. *Proceedings of IEEE Transactions on Computer-Aided Design*, 14(1):45–60, January 1995.

[29] A. Chakraborty and M.R. Greenstreet. Efficient self-timed interface for crossing clock domains. In *Proceedings of 9th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 78–88, 2003.

[30] L.R. Dennison, W.J. Dally, and D. Xanthopoulos. Low-latency plesiochronous data retiming. In *Proceedings of 16th Conference on Advanced Research in VLSI*, pages 304–315, 1995.

[31] U. Frank and R. Ginosar. A predictive synchronizer for periodic clock domains. *Electronic Notes in Computer Science*, 3254, 2004.

[32] M. Pechoucek. Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, 25(2):133–139, February 1976.

[33] M. J. Stucki and J. R. Cox. Synchronization strategies. In *Proceedings of Caltech Conference on VLSI*, pages 375–393.

[34] R. Kol and R. Ginosar. Adaptive synchronisation for multi synchronous systems. In *Proceedings of ICCD*, pages 188–198, 1998.

[35] S. Kim and R. Sridhar. Hierarchical synchronization schemes using self-timed mesochronous interconnections. In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages 1824–1827.

[36] J. Seizovic. Pipeline synchronization. In *Proceedings of International symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 87–96, 1994.

[37] T. Chelcea and S. M. Nowick. The design of low-latency interfaces for mixed-timing systems. In *IEEE Workshop on Complexity-Effective Design (ISCA)*, 2002.

[38] M. Greenstreet. Implementing the stari chip. In *Proceedings of IEEE International Conference on Computer Design*, pages 38–43, 1995.

[39] K. Yun and R. P. Donhue. Pausible clocking: A first step towards heterogeneous systems. In *Proceedings of International Conference on Computer Design*, 1996.

[40] J. Kessels, A. Peeters, P. Wielage, and S. Kim. Clock synchronization through handshake signalling. In *International Symposium on Asynchronous Circuits and Systems*, 2002.

[41] F. Rosenberger, C. Molnar, T. Chancey, and T. P. Fang. Q-modules: Internally-clocked delay insensitive modules. *IEEE Transactions on Computers*, 37(9), September 1988.

[42] W. S. VanScheik and R. F. Tinder. High speed externally asynchronous/internally clocked systems. *IEEE Transactions on Computers*, 46(7):824–829, July 1997.

[43] P. Nilsson and M. Torkelson. A monolithic digital clock-generator for on-chip clocking of custom dsps. *IEEE Journal of Solid-State Circuits*, 31(5), May 1996.

[44] Zhang et al. A 1-v heterogeneous reconfigurable dsp ic for wireless base band digital signal processing. *IEEE Journal of Solid-State Circuits*, 35(11), November 2000.

[45] W. Lim. Design methodology for stoppable clock systems. *IEE Proceedings*, 133(1):65–69, January 1986.

[46] R. Kol and R. Ginosar. A doubly-latched asynchronous pipeline. In *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 706–711, 1997.

[47] D.H. Linder and J.H. Harden. Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry. *IEEE Transactions on Computers*, 45(9):1031–1044, September 1996.

[48] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Proceeding of 10th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 149–158, 2004.

[49] H. Jacobson, P. Kudva, P. Bose, P. Cook, S. Schuster, E. Mercer, and C. Myers. Synchronous interlocked pipelines. In *Proceedings of the 8th International Symposium on Asynchronous Circuits and Systems*, page 2003.

[50] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. A concurrent model for de-synchronization. In *Proceedings of International Workshop on Logic Synthesis*, pages 294–301, 2003.

[51] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. *Proceedings of the 11th International Conference on Computer-Aided Verification*, 1633:123–133, 1999.

[52] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by- construction latency insensitive design. In *Proceedings of International Conference on Computer-Aided Design*, pages 309–315, 1999.

[53] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[54] L. P. Carloni. *Latency-insensitive design*. PhD Thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, Berkeley, 2004.

[55] L. P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Electronic Notes in Theoretical Computer Science*, 146(2):61–80, 2006.

[56] M. R. Casu and L. Macchiarrulo. A new approach to latency insensitive design. In *Proceedings of the Design Automation Conference*, pages 576–581, 2004.

[57] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. *Lecture Notes in Computer Science*, 1664:162–177, 1999.

[58] N. Lynch and E. Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, 1989.

[59] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J. P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. *Lecture Notes in Computer Science*, 2491:32–49, 2002.

[60] P. L. Guernic, J. P. Talpin, and J. C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–303, 2003.

[61] S. K. Shukla R. Gupta F. Doucet J. P. Talpin, P. L. Guernic. Formal refinement-checking in a system-level design methodology. *Fundamenta Informaticae*, pages 243–273, 2004.

[62] J. P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. Polychrony for refinement based design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1172–1173, 2003.

[63] The POLYCHRONY Toolset. Developed at irisa. available at www.irisa.fr/espresso/polychrony/.

[64] M. Mousavi, P. L. Guernic, J. Talpin, S. K. Shukla, and T. Basten. Modeling and validating globally asynchronous design in synchronous frameworks. In *Proceedings of European Design and Test Conference*, pages 384–389, 2004.

[65] G. Berry and E. M. Sentovich. An implementation of constructive synchronous programs in polis. *Formal Methods in System Design*, 17(2):135–161, 2000.

[66] A. Girault and C. Mt'enier. Automatic production of globally asynchronous locally synchronous systems. *Lecture Notes in Computer Science*, 2491:266–281, 2002.

[67] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. *Lecture Notes in Computer Science*, 2491:240–251, 2002.

[68] R. Kurshan, M. Merritt, A. Orda, and S. Sachs. Modelling asynchrony with a synchronous model. *Formal Methods in System Design*, 15(3):175–199, 1999.

[69] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Proceedings of the Fourth International Conference on Application of Concurrency to System Design*, 2004.

[70] J. Ouy. A survey of desynchronization in a polychronous model of computation. *Electronic Notes Theoretical Computer Science*, 146(2):151–167, 2006.

[71] P. Caspi B. Caillaud. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[72] O. Maffeis and P. Le Guernic. Distributed implementation of signal: schedulling and graph clustering. *Lecture notes in Computer Science*, 863.

[73] G. Berry. The constructive semantics of pure esterel(ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps).

[74] V. Khomenko. *Model checking based on prefixes of petri net unfoldings, publisher =*.

[75] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley Publication, 1980.

[76] I. Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32(6):720–738, June 1989.

[77] S. W. Moore, G. S. Taylor, R. D. Mullins, and P. Robinson. Point-to-point gals interconnect. In *proceedings of Eighth International Symposium om Advanced Research in Asynchronous Circuits and Systems*, 2002.

[78] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Synthesis of Asynchronous Controllers and Interfaces*. Springer, Berlin, 2002.

[79] M. Krstic, E. Grass, and C. Stahl. Request driven gals technique for wireless communication systems. In *proceedings of 11th International Symposium om Advanced Research in Asynchronous Circuits and Systems*, 2005.

[80] D. S. Bormann and P.Y.K. Cheung. Asnchronous wrapper for heterogeneous systems. In *Proceedings of ICCD*, pages 307–314, 1997.

[81] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design - A System's Perspective*. Kluwer Academic Publishers, 2001.

[82] M. Nielsen, G. Rozenberg, and P. Thiagarajan. Elementary transitions systems. *Theoretical Computer Science*, 96(1):3–33, April 1992.

[83] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving petri nets from finite transition systems. *IEEE Transactions on Computers*, 47.

[84] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[85] A. Yakovlev and A. Koelmans. Petri nets and digital hardware design. *Lecture Notes in Computer Science: Lectures on Petri Nets II: Applications*, 1492, 1998.

[86] T. L. Floyd. *Digital Fundamentals*. Prentice Hall Publishers, 1997.

[87] M. Koutny and M. Pietkiewicz-koutny. Transition systems of elementary net systems with localities. In *proceedings of CONCUR*, pages 173–187, 2006.

[88] V. Khomenko, A. Madalinski, and A. Yakovlev. Resolution of encoding conflicts by single insertion and concurrency reduction based on stg unfoldings. In *proceedings of ACSD*, pages 57–66, 2006.

[89] A. Madalinski. *Interactive Synthesis of Asynchronous Systems based on Partial Order Semantics*. PhD thesis, University of Newcastle, 2006.

[90] G. Berthelot. Checking properties of nets using transformations. *Lecture Notes in Computer Science: Advances in Petri Nets 1985*, 222, 1940.

[91] S. Dasgupta, D. Potop-Butucaru, B. Caillaud, and A. Yakovlev. Moving from weakly endochronous systems to delay-insensitive circuits. *Electronic Notes in Theoretical Computer Science*, 146(2):81–103, January 2006.

[92] K. J. Nowka, G. D. Carpenter, E. W. MacDonald, H. C. Ngo, B. C. Brock, K. I. Ishii, T. Y. Nguyen, and J. L. Burns. A 32-bit powerpc system-on-a- chip with support for dynamic voltage scaling and dynamic frequency scaling. *IEEE Journal of Solid-State Circuits*, 37(11):1441–2447, 2002.