Latency Insensitive Design for Multiple Clock SoC

A thesis submitted in partial fulfillment of
requirements for the third semester of degree of

Master of Technology
in
Computer Science & Engineering

Submitted by

Shirshendu Das
08410112

Under the guidance of

Dr. Hemangee K. Kapoor

Department of Computer Science and Engineering

Indian Institute of Technology, Guwahati

Abstract

Latency insensitive (LI) design has been proposed as the correct by construction methodology for coping with latency arising due to long interconnects in a single clock SoC. GALS systems are also a robust framework for solving several issues in the design of complex hardware systems. Thus, in order to build a general system, we need both the approaches from LI design and GALS architecture. Our work builds the mathematical foundations for such type of systems.

We extend the single clock LI theory to handle latency issues in connecting multiple clocked modules. Illustrations are given to show why the single clock LI design cannot be applied to this new domain. The theory given can be used to connect modules with rationally related clock frequencies and also for frequencies having phase differences.

A multiple clock patient process is defined and actions needed to handle stalls as well as back-pressure are provided. A definition for an appropriate relay station is given, which can be plugged along with the relay stations of the single clock domain. The new relay station is thus acting as a converter/link between the two timing domains. Proofs for composability of such systems are given and the limiting case of a single clock relay station is derived from the multiple clock relay station.

Contents

List of Figures

Chapter 1

Introduction

As the commercial demand of System-on-Chip (SoC) based products are increasing it become very necessary to design the chips in short time and also with a minimum cost. For achieving these two constraints the designer should consider the followings:

- The design process of the SoC must be very fast. They have to choose some design procedure so the design cycle need not repeat multiple times.
- They have to reuse the existing intellectual-property modules (known as IP cores) to design the SoC's.

But the main problem in designing a SoC in quick time is to handle the communication delays between the different modules of SoC. Even the number of layers and aspect ratios are increasing; the resistance-capacitance delay of an average metal line having a constant length is getting worse with each process generation [**?**, **?**]. The operating frequency, die size and average interconnect length are also increasing. Combination of all these effects making the interconnect delay larger than a clock cycle. According to [**?**] a signal requires more than ten clock cycles to traverse the entire chip area. Also [**?**] estimated that in one clock cycle a signal can travel only 0.4% to 1.4% of the total chip area. Hence the key property to increase the performance of the overall design is to minimize the on-chip distance traveled by a signal.
It is believed that the recent CAD tools containing logic synthesis and physical design will suffer from the impracticality of accuracy estimating the latency of global wires [**?**]. A small variation in the input specification (hardware description language) can

lead to major variations in the final design. The main assumption in synchronous design is that the delay of each combinational path is less than one clock cycle [**?**]. Combinational path means the path traveled by a signal starting from one latch and only going through the combinational logic and wires to reach another latch. The slowest combinational path dictates the maximum operating frequency for the system. Once the final layout have been derived by the designer, every path having delay more than one clock cycle simply represents an exception and must be fixed. To fix these exceptions designers use techniques like wire buffering, transistor resizing to reroute the wires, replacing the modules and if nothing works then they have to redesign the entire system. So the design process repeats multiple times to minimize the on chip delays. But even there is no guarantee to get a minimum *on-chip delay* design in finite time.

Existing IP cores can be used to design new SoC. Using these IP cores the designers can save both the time and cost. IP cores were actually functional blocks designed for previous generation within the same vendor. Now IP cores are also available as a standard functional unit designed by some specialized vendors. An IP core must have the capability to communicate with other modules in different environment. Designing a new SoC requires assembling the existing IP cores with some modifications on them. The main challenges in using these IP cores is the synchronization issues that will arise naturally while assembling predesigned component. Researchers proposed the solution that there should be a orthogonalization of the factors effecting the design in order to increase the complexity of these systems [**?**]. System on Chip design essentially consists of two parts. The first one is the computation performed by each of the IP cores and the second one is the communication between them. It is believed that the design paradigm will now shift from computational bound to communication bound designs. Under these circumstances, the Latency Insensitive design will automatically gain attention as it serves the need of the hour. It uses the existing IP cores to design the new SoC's without going through system rerouting or redesigning. Also, latency is influencing the design stages of state at microprocessors. Few stages of pipelining called the "drive" stages" of Netburst architecture of Pentium 4 are initiated by latency [**?**].

1.1 Latency Insensitive Design

The Latency Insensitive (LI) design actually based on latency insensitive protocol proposed by Carloni et el.[2]. This protocol controls the communication among the different modules of a patient system. Here patient system means a synchronous system whose functionality depends only on the order of each signaling events and not there exact timing. For example consider that there are two signals entering in a system as input and another signal is coming out from the system as output which carrying the result of some functional computation performed with the input signals. Now if the system is patient then for any arbitrary number of delays in the input signals the output signal will always consist of same ordering sequence of events. The timing of events in a sequence may not be similar with another sequence but the ordering of there events will remain same.
Designer can model a synchronous system as a set of modules. These modules are communicating with each other by exchanging signals on a set of point-to-point channels. The protocol guarantees that a system, if composed of functionally correct modules, behaves correctly, independent from the delays in the channels connecting the modules. An hardware implementation of the system is possible to synthesize automatically in such a way that the functional behavior of the system is robust with respect to large variation in communication latency. This methodology orthogonalizes computation and communication because it separates the module design from the communication architecture options, while enabling automatic synthesis of the interface logic. The separation is usefull in two ways:

- Module design becomes simple because designer can assume the synchronous hypothesis. That is the intermodule communication will take no time (or in other words, it's completed within one virtual clock cycle.)
- It's permits the exploration of tradeoffs in deriving the communication architecture up to the design process late stage, because the protocol gurantees that the interface logic can absorb arbitrary latency variations.

A brief description of latency insensitive design flow is given below:

- Collecting some synchronous components. These components can be custome modules or IP cores.

- Encapsulate each component to an automatically generated shell to make them patient. A shell is a collection of buffering queues one for each port and a control logic that interfaces the component with the LI protocol.

- Physical design. In this step the designer design the system by using standard CAD tools. This process considers the delay of each channel as zero time of a virtual clock.

- Inserting relay stations: designers segment every wire whose latency is grater than the clock period of real clock by distributing the necessary relay stations. The key idea is borrowed from pipelining: partition the long wires into segments whose length satisfy the timing requirement imposed by the real clock. We will explain relay station later in this chapter.

The only necessary precondition in this method is that the encapsulated components are stallable, that is they can freeze their operation for an arbitrary time without loosing there internal state.

Let us consider that we want to design a system with four modules m_1, m_2, m_3 and m_4. Consider that m_1 will send information to m_2, m_3 and m_4. Also m_3 needs to send information to m_2 and m_4. The design process start by taking four IP cores one for each module. After this, each IP core must have to encapsulate with an automatically generated shell to make the core patient. Let us called these combinations (IP core encapsulated with shell) as patient modules. Now with these four patient modules the design tool design the system layout without considering the delays in the communicating wires. Figure 1.1(a) is showing a sample layout design. After designing this layout we have to calculate the communication delays of each wires. In our figure these are shown by numbering each wires. Now we can see from the figure that there are two wires having delay more than one clock period. To solve these problems we now need to insert relay stations in the corresponding wires (as shown in fig 1.1(b)) to divide them into segments having delay of one clock period. Now since each modules are patient and even the relay stations are also patient [2] (we will explain later), the entire system must be a patient system [2] (explanation is in next chapter).

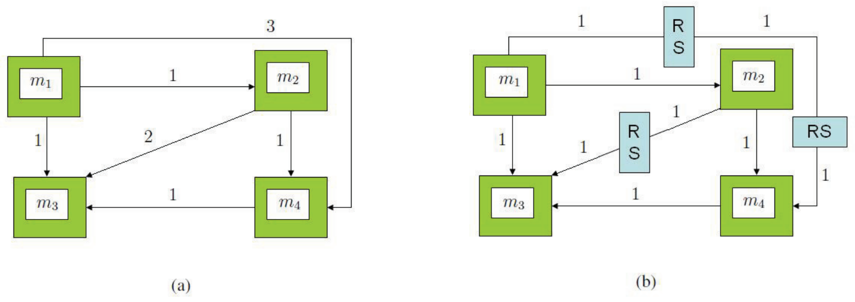

Figure 1.1: *An example of Latency Insensitive(LI) design.*

1.1.1 Channels and back-pressure

Channels are point-to-point unidirectional links between a source and a sink module. Data are transmitted on a channel by means of packets that consist of variable number of fields. We consider only two fields: payload, which contains data; and void, a one bit flag that, if set to 1 denotes that no data are present in the packet. If a packet contains meaningful payload then we call it as true packet.

A channel is a combination of wires and relay stations. There should be always a finite number of relay stations in a channel. These relay stations are actually representing the buffering capability of a channel. At each clock tick the source module will either put a new true packet into the channel or put a void packet on it if there is no true data to send. From now onwards we represent the event of putting a void data into a channel as inserting a delay into the channel. Conversly, the sink module will receive the new incoming packet from the channel. The receiving packet may be true or void.

Every module maintains a separate queue for its each input and output channels. A source might not be ready to send a true packet, and a sink module might not be ready to receive it if, for instance, its input channel is full. Since the latency insensitive protocol demands fully reliable communication among the modules, it requires the proper delivery of all packets. To achieve this, channel definitions are slightly modified to allow a bit of information to move in the opposite direction. Figure 1.2 shows the signals as dotted lines. It is similar to the NACK signal in the request/acknowledge protocols of asynchronous design. This back pressure mechanism controls the flow of

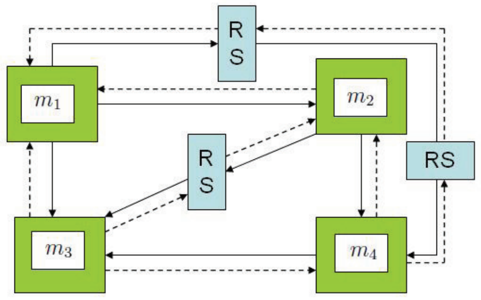

Figure 1.2: *An example showing backpressure in Latency Insensitive design.*

information on a channel while guaranteeing that no packets are lost.

1.1.2 Shell encapsulation

It is possible to devise a method to automatically synthesize an instant of a shell as a wrapper to encapsulate a module. We can also interface the shell with the channels so that the module becomes a patient system. The only necessary pre-condition for doing this is that the module must be stallable. At each clock cycle, the module's internal computation must fire only if all inputs required for that computations have arrived. The module shell's first task is to gurantee this input synchronization. Its second task is to output propagation. At each clock cycle, if the module has produced new output value and no output channenl has previously raised a stop flag, then the shell can transmit these output values by generating new true packets. If the shell does not verify either of these conditions, then it must transmit a void packet. So in summary, a shell for a module cyclically performs the following tasks:

- Filtering away the void packets from the incoming packets and extracting the true packets
- When all he input values required for the next operation are become available, it allows the module to perform the operation.
- It obtains the computation results from the module.

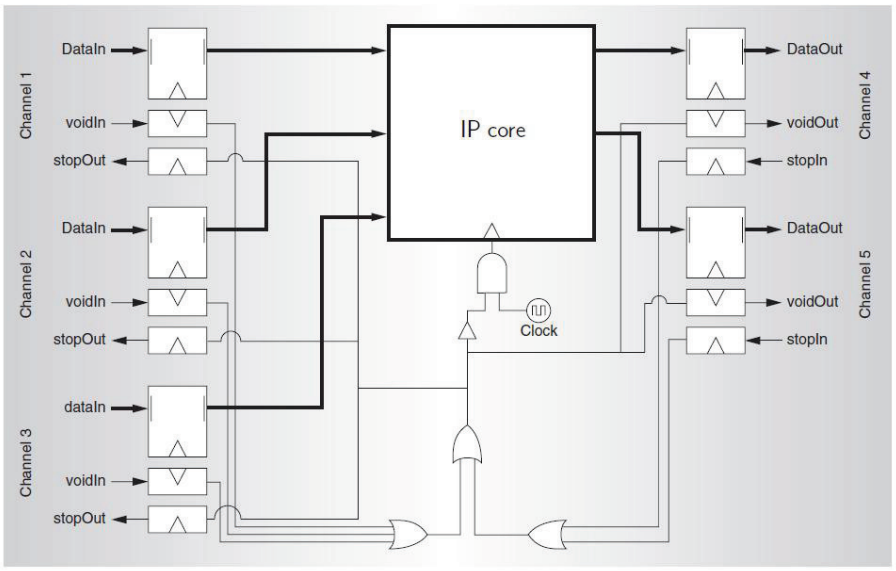

Figure 1.3: *Shell encapsulation: Making an IP core patient. (taken from [**?**])*

- It routs the result into the output channels if no output channel has previously raised a stop flag; otherwise it stores the result in the queue and sends a void packet. Note that in every clock cycle it will send a packet either true or void.

Figure 1.3 taken from [**?**] illustrates a simple unoptimized implementation of a shell wrapping a module having three input and two output channels.

1.1.3 Relay stations

As we mentioned before, relay stations are inserted within the wires to divide them into segments. Now if we consider each segment as a separate channel then relay station is a process connecting two channels. In the next chapter we will explain the properties and the proof of the theorem given in [2], which says that relay station is a patient process.

1.2 Motivation

Recent advances in VLSI technology has made it possible for integration of complex systems into SoC design. Higher power integration and deep pipelining has raised concerns regarding the power consumption and clock skew over in a synchronous SoC. This has lead to the new paradigm of Globally Asynchronous Locally Synchronous (GALS) design. This design divides the SoC into various domains called Synchronous Blocks (SBs) and they interact with each other using asynchronous communication protocols. The modules in a block are synchronous with each other and can communicate with each other using either synchronous or asynchronous protocols according to the design requirement. Here we have two paradigms which propose solutions to the design problems today. Attempts have been made by researchers to apply LI Design for GALS systems. In this work, we attempt to provide a denotational semantics for a particular design of multiple clock latency insensitive (MCLI) design and thus prove the correctness of the LI design for multiple clock systems. We call the systems, designed with "MCLI design" as MCLI systems. The existing denotational model of the LI system only provides semantics for single clocked relay stations (i.e. both the sender and the receiver have the same clock frequency with no jitter and skew). Thus we did this work to extends the relay station to support multiple clock domain. Lastly we derive a process algebraic model for our proposed MCLI system. For this we use the process algebraic language "Timed CSP" [**?**], which is an extension of "CSP" [**?**].

1.3 Organization of the report

Our work is mainly divided into two parts:

1. **Proposing a solution for MCLI design and a denotational framework for expressing it.**
2. **Presenting a multiple-clock process-algebraic model for our proposed MCLI system.**

This chapter introduces the basic features and the need of Latency Insensitive (LI) design. Later we examine the current state of art and view at the drawbacks of the existing LI design methodology and semantics of LI design. We then propose a solution for MCLI design and also a denotational framework for expressing it. After this we build a "Timed CSP" [**?**] model (process algebraic) for our proposed solution of MCLI design. The rest of the chapters are organized as follows: in chapter 2 we provide the literature review and the detail description of the existing denotational framework for expressing Single Clock LI design. We used many notations, definitions and theorems from this existing framework to propose and prove the correctness of our own framework. In chapter **??** we propose our whole work of part1 as defined above. Chapter 4 is dedicated for the literature review of process algebra. There has been many languages (process algebraic) for modeling interactive systems depending on their nature (clock-less, single-clock or multiple-clocks). In this chapter we discuss different existing languages of processes algebra and their pros and cons. Also we explain the existing process-algebraic model for single-clock LI systems proposed in [1]. Our proposed model for MCLI system is actually an extension of this model [1]. In Chapter 5, we propose a process algebraic model for our MCLI system using "Timed CSP". And finally in Chapter 6, we conclude with future works.

Chapter 2

Related Works-1

In this chapter we are going to discuss some important works about both single clock and multi clock LI design. Single clock LI design is now a fully successful chip design procedure. Many authors contributed for single clock LI design after it has been proposed by Carloni et el. [2, **?**]. Multiclock LI design is still in experimental phase and several peoples are working on it. Many useful concepts about multiclok LI designed has been proposed, but no one has given any denotational semantics for expressing and proving its correctness. In the first section of this chapter we discuss some important and interesting works about single clock LI design. Second section discuss about the current status of multiclock LI design. The remaining sections describe the denotational semantics used by Carloni et el. for the theory of single clock LI design [2].

2.1 Single Clock Domain

Latency-insensitive system were originally proposed by Carloni et el. [2, **?**, **?**, **?**] for the design of single clock SOC's. If an IP block is not latency-insensitive then they proposed to make it latency-insensitive by encapsulating the IP by a simple wrapper circuit proposed by them. The main concepts of this work are already discussed in chapter 1. In this method, in order to manage interconnect latency, relay stations need to be inserted along the interconnect. Later many interesting implementations of the LI design protocol came into existence [**?**]. The performance analysis of LI designed system has been analyzed in [?]. But this analysis has not considered the "back-pressure", which is important for fully reliable communication in LI designed systems. The performance analysis done at [?] using Max Plus algebra formally proved the performance upper bound, achievable by LI design. They also proved that their proposed implementation of LI protocol provides robust communication through back-pressure. A model of the relay stations and shell wrappers in SyncCharts was described in [?]. There has been a Process Algebra model of the LI Design [1]. In this paper the author has given the definitions for LI computational blocks and LI connectors. Some conditions are also given to check the liveness and deadlock freedom of LI systems. The paper was actually a step toward the high-level specification and verification of LI systems.

2.2 Multiple Clock Domain

In order to extend the single clock latency-insensitive design proposed by Carloni et el. [2] to multiclock many authors have done many useful works. In [?] and [?] the authors propose four new FIFO designs. Each design is for various combinations of synchronous and asynchronous systems. The basic implementation of LI methodology for the GALS architecture is proposed in [?]. In [?] the authors proposed a method of addressing the combined problem of multiple clock frequencies and implementation-induced latencies and clock skews. In this method different clock frequencies are managed using the rate multipliers proposed by [?]. In [?] and [?], the authors examine systematically the problem of designing interface circuits for rationally clocked components in GALS systems.

2.2.1 GALS design style

In [?] the authors categorize GALS design into three different classes:

- plausible clocks.
- asynchronous interface.
- loosely synchronous interfaces

Plausible clock is used to enable separate clock domains to communicate without metastability. Each locally synchronous blocks generates its own clock with a ring oscillator. Each ring oscillator period is set according to the speed requirements of the block it derives. One way of pausing is to pause temporarily [**?**, **?**] or stretch [**?**] the receiver's clock. Such designs require "wrapper logic" at the receiver. Its main drawback is, the receiver's clock has to be started again and again. A design by Myers et el. [**?**] which interfaces asynchronous to synchronous domain uses this approach of pausing the clocks. Pausing delays a clock sampling edge until after the arrival of data from the other domain, thus avoids metastability altogether.

The second GALS design style is the asynchronous interface. This methods uses circuits known as synchronizer to transfer signals arriving from an outside timing domain to the local timing domain. Although simple asynchronous interfaces suffer from low throughput, this limitation can be overcome with careful designs. Asynchronous interfaces offer the most flexibility and probably the easiest integration into the exiting CAD flows.

The third style, loosely synchronous interface, arises when some bounds on the frequencies of communicating blocks are known. In this style, the designer exploits these bounds to ensure that timing requirements are met. This style requires timing analysis on the paths between the sender and receiver and is less amenable to dynamic changes in the clock frequency. This analysis however, makes handshaking unnecessary during data transfer, so the resulting circuits can archive higher performance and have more deterministic latencies than those of the other methods.

2.2.2 Multi-clock interconnects

There are three major interconnect designs proposed in [**?**], [**?**], [**?**] and [**?**]. We give a brief description of them as follows:

2.2.2.1 Self-Timed Interfaces for Crossing Clock Domains

The full details of this design can be found in [**?**]. The advantage of this design is that it not only deals with multi-clock interconnect, but also handles the jitter and clock delay induced. The basic model is the use of a synchronizer which takes the input from both the clocks of receiver and sender and designs a new clock pulse that does not have a synchronization problem with either the sender or the receiver. The paper starts with a design where the modules are running on same clock frequency but having jitter and then extends the same design to modules where the clock frequencies are rationally related and unrelated clock cycles. The disadvantage is that the design is complex and the scope of this design is restricted.

2.2.2.2 Interface for rationally clocked GALS systems

The full details of this design can be found in [**?**]. The main idea of this paper is to examine two different aspects of multi-clock interconnect, namely flow control and synchronization. They explained that these two phenomenon are orthogonal but elimination of one does not necessary means the elimination of the second. This paper then proceeds to enplane each of these aspects in detail and thus comes up with a design which is free from both the problems of flow control and synchronization. A method to implement this design for rationally related frequencies is also given. The design is simple and robust is only applicable to a restricted designs.

2.2.2.3 Robust interfaces

The full details of design can be found in [**?**]. This paper is somewhat similar to [**?**] and gives a description of a FIFO which acts as multi-clock interconnect. The FIFO has input at one clock frequency and output at a different clock frequency. The FIFO maintains a *begin* and *end* flag which define the beginning and ending of the FIFO. Two signals empty and full are connected to receiver and sender respectively. It is a general interconnect and can be used to connect domains of unrelated and rationally related clock frequencies. This design is simple and robust, but there is a requirement of use of synchronizer in some cases.

2.3 Existing Theory

As we mentioned previously, theory of single clock LI design was proposed by Carloni et el. [2]. Now we are giving a brief explanation about there proposed theory. This explanation is necessary because we are using similar framework for our proposed theory (for multiple clock LI design). We used many notations, definitions and theorems from this existing theory to extend it for multiple clocks. Carloni et el. [2] used a previous framework called *Tagged Signal Model*, which has been proposed by Lee and Sangiovanni Vincentelli, to represent complex system as a collections of signals and processes [**?**]. In the remaining two sections of this chapter we will explain both *Tagged Signal Model* as well as the theory proposed by [2].

2.4 Tagged Signal Model

The formal theory behind the LI design is based upon a *tagged signal model* [**?**]. which can be used to represent complex systems as a collections of signals and processes. Below we are explaining some concepts of *Tagged Signal Model* which are useful for both existing and our proposed theory.

The basic element of the *tagged signal model* is the object called *event*. An event has a tag and a value. Tag is used to model time, precedence relationship and synchronization point etc. where the value is used to represent the operands and results of computation. If $\mathcal{V}$ represents the set of values and $\mathcal{T}$ represents the set of tags then an *event* e is defined as a member of $\mathcal{V} \times \mathcal{T}$.

A signal s is defined as $s \subset (\mathcal{V} \times \mathcal{T})$. It can also defined as a set of events. If two events have the same tag then the events are called *synchronous* events. Similarly two signals s_1 and s_2 are said to be *synchronous* if their corresponding events have same tag. So all synchronous signals contains same set of tags.
The set of all signals is denoted by S where S is a power set of $\mathcal{V} \times \mathcal{T}$. An *N tuple* of signals is defined as the set of *N* signals. The set of all such N tuples is denoted by $\mathcal{S}^N$. A process $\mathcal{P}$ is a subset of $\mathcal{S}^N$. A particular *N tuple* $nt \in \mathcal{S}^N$ is said to satisfy a process P if $nt \in P$. The *N tuple*, which satisfies the process is called a *behavior* of the process. So the process can also be defined as a set of behaviors. If $\{P_1, \ldots, P_M\}$ be a set of processes of same sort [**?**] then the *composition* of these processes is a new process P, defined as $P = \bigcap_{i=1}^{M} P_i$.

If $J = (j_1, \ldots, j_m)$ be an ordered set of indexes in the range $1 \leq j \leq N$ then the $projection$ $\pi_I(b)$ of a behavior $b = (s_1, \ldots, s_N) \in \mathcal{S}^N$ onto $\mathcal{S}^m$ is defined as $\pi_J(b) = (s_{j_1}, \ldots, s_{j_m})$. Given a process $P \subset \mathcal{S}^N$, the projection $\pi_I(P)$ can be defined as the set $(s' | \exists s \in P \wedge \pi_J(s) = s')$. A simple process where two (or more) of the signals are constrained to be identical is called a $connection$ and is denoted as C. For example $C(i,j) \subset \mathcal{S}^N : (s_1, \ldots, s_N) \in C(i,j) \Leftrightarrow s_i = s_j$, with $i, j \in [1, N]$.

Inputs are events or signals that are defined outside the process. Many processes have the notion of inputs. More formally an *input* to a process $P \subset \mathcal{S}^N$ is an externally imposed constraints $A \subset \mathcal{S}^N$ such that $A \cap P$ is the set of total acceptable behaviors. The set of all possible inputs $B \subset \mathcal{S}^N$ is a further characterization of a process. If P be a process and b represents its inputs then P is said to be *closed* if $B = \left\{\mathcal{S}^N\right\}$, a set with only one element $A = \mathcal{S}^N$. Since the set of behaviors is $A \cap P = P$, there are no input constraints in a closed process. The set of signals of a process P can be partitioned into three disjoint subsets by partitioning the index set as $\{1, \ldots, N\} = I \cup O \cup R$, where I is the ordered set of indexes for the input signals of P, O is the ordered set of indexes for output signals of P and R is the ordered set of indexes of the remaining signals of P. A process is $functional$ with respect to (I, O) if for every behaviors $b \in P$ and $b' \in P$ where $\pi_I(b) = \pi_I(b')$, it follows that $\pi_O(b) = \pi_O(b')$. Hence we can completely characterize a functional process P by the tuple (F, I, O) where F is a function a function $F : \mathcal{S}^{|I|} \rightarrow \mathcal{S}^{|O|}$. A process is determinate if for any input $I \in B$, it must have exactly one behavior or exactly no behaviors. In other cases the process is considered as *nondeterminate*.

If a system (set of processes) is synchronous then all the signals belonging to this system will be synchronous with each other. A totally ordered set *timestamps* of tags $\mathcal{T}$ is used in timed systems. Given a signal s, the natural ordering of its events is represented by the *timestamps* ordering of s. A functional process is (strictly) causal if two outputs can only differ at the timestamps that (strictly) follow the timestamps when the inputs producing these outputs show a difference. A functional process P is said to be causal if $\forall s_i, s_j \in \mathcal{S}^{|I|} (d(F(s_i), F(s_j)) \leq d(s_i, s_j))$. Similarly P is strictly causal if $\forall s_i, s_j \in \mathcal{S}^{|I|} (d(F(s_i), F(s_j)) < d(s_i, s_j))$. Here d represents a metric on the set $\mathcal{S}^N$ of N-tuples of signals.

2.5 Theory of LI design

2.5.1 Important definitions about LI design

Definition 2.5.1. *Function $\sigma : \mathcal{S} \times \mathcal{T}^2 \rightarrow \sum_{lat}$ represents the sequence of values of a signal. This function takes a signal $s = \{(v_0, t_0), (v_1, t_1), \ldots\}$ and an ordered pair of timestamps $(t_i, t_j), i \leq j$ and returns a sequence $\sigma_{[t_i,t_j]} \in \Sigma_{lat}$ s.t. $\sigma_{[t_i,t_j]}(s) = v_i, v_{i+1}, \ldots, v_j$.*

Example 2.1. Consider the signal s_2 of behavior b in the figure 2.1. We have

$$\sigma_{[t_0,t_6]}(s_3) = i_1, i_2, \tau, \tau, i_3, \tau, i_4$$

Definition 2.5.2. *The function $\mathcal{F}_\iota : \Sigma_{lat} \rightarrow \Sigma^*$ returns only the informative events from a given sequence of above function. Mathematically the function returns a sequence $\sigma' = \mathcal{F}_\iota[\sigma]$ s.t*

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s), & \textit{if } \sigma_{[t_i,t_i]}(s) \in \Sigma \\ \epsilon, & \textit{otherwise} \end{cases}$$

Example 2.2. If we take the signal s_2 of behavior b from the figure 2.1, then we have

$$\mathcal{F}_\iota[\sigma_{[t_0,t_6]}(s_3)] = i_1, i_2, i_3, i_4$$

Definition 2.5.3. *The function $\mathcal{F}_\tau : \Sigma_{lat} \rightarrow \{\tau\}^*$ returns only the stalling events from a given sequence of above function σ. Mathematically the function returns a sequence $\sigma' = \mathcal{F}_\tau[\sigma]$ s.t*

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s), & \textit{if } \sigma_{[t_i,t_i]}(s) \in \tau \\ \epsilon, & \textit{otherwise} \end{cases}$$

Example 2.3. If we take the signal s_2 of behavior b from the figure 2.1, then we have

$$\mathcal{F}_\tau[\sigma_{[t_0,t_6]}(s_3)] = \tau, \tau, \tau$$

$$b = \begin{cases} s_1 = & i_1, i_2, i_3, i_4, i_5, i_6, i_7, \tau, \tau, \tau, \tau, \tau, \dots \\ s_2 = & i_1, i_2, \tau, i_3, i_4, i_5, \tau, i_6, i_7, \tau, \tau, \tau, \dots \\ s_3 = & i_1, i_2, \tau, \tau, i_3, \tau, i_4, i_5, \tau, i_6, \tau, i_7 \dots \end{cases}$$

Figure 2.1: *An example of a behavior with three signals.*

Definition 2.5.4. *Given a signal s is said to be strict if and only if all informative events precede all stalling events. The signals which are not strict are called delayed signal.*

Example 2.4. In figure 2.1, the signal s_1 of behavior b is a strict signal where the remaining two signals are delayed signals.

Definition 2.5.5. *Two signals are said to be latency equivalent if they have the same sequence of informative events. It means that the two signals are identical except for different delays between two successive informative events. Mathematically two signals s_i and s_j are latency equivalent $s_i \equiv_\tau s_j$ iff $\mathcal{F}_\iota[\sigma(s_i)] = \mathcal{F}_\iota[\sigma(s_j)]$.*

Example 2.5. In figure 2.1, all three signals of behavior b are latency equivalent with each other.

Definition 2.5.6. *Given a class of latency equivalent signals a strict signal which is latency equivalent with them is called a reference signal of that class of signals.*

Example 2.6. In figure 2.1, the signal s_1 is the reference signal of the remaining two signals of behavior b.

Definition 2.5.7. *The ordinal of an informative event coincides with its position in the reference signal. Mathematically the ordinal of an informative event $e_k = (v_k, t_k) \in \mathcal{E}_\iota(s)$ is defined as $ord(e_k) = \mathcal{F}_\iota[\sigma_{[t_0,t_k]}](s) - 1$.*

Example 2.7. In figure 2.1 $ord(e_0(s_2) = 0), ord(e_4(s_2)) = 3$ and $ord(e_4(s_3)) = 2$.

Definition 2.5.8. *If $b_1 = \{s_1, \dots, s_N\}$ and $b_2 = \{s'_1, \dots, s'_N\}$ two behaviors then they are said to be latency equivalent iff $\forall i, (s_i \equiv_\tau s'_i)$. If all the signals in a behavior are strict signals then the behavior is called strict behavior. Every class of latency equivalent behaviors contains only one strict behavior. This is called the reference behavior.*

Definition 2.5.9. *Two processes P and P' are said to be latency equivalent $P \equiv_\tau P'$, if every behavior of P is latency equivalent to some behavior of P'. A process is said to be a strict process iff every behavior b is strict. Every class of latency equivalent processes contains only one strict process and it is called the reference process.*

Definition 2.5.10. *Given a behavior $b = (s_1, , s_N)$, symbol $\leq_c$ denotes a well-formed order on its set of signals. The well-founded order induces a lexicographic order $\leq_{lo}$ over the set of informative events of b, s.t. for all pairs of events (e_1, e_2) with $e_1 \in \mathcal{E}_\iota(s_i)$ and $e_2 \in \mathcal{E}_\iota(s_j)$*

$$e_1 \leq_{lo} e_2 \Leftrightarrow [(ord(e_1) < ord(e_2)) \vee ((ord(e_1) = ord(e_2)) \wedge (s_i \leq_c s_j))] \tag{2.1}$$

Example 2.8. If we consider that the three signals of behavior b in the figure 2.1 have the relationship $s_1 \leq_c s_2 \leq_c s_3$ then the ordering of the informative events in this behavior will be:

$$e_0(s_1), e_0(s_2), e_0(s_3), e_1(s_1), e_1(s_2), e_1(s_3), e_2(s_1), e_3(s_2), e_4(s_3), e_3(s_1), e_4(s_2), \ldots$$

Definition 2.5.11. *Given a behavior $b = (s_1, , s_N)$ and an informative event $e(s_i) \in \mathcal{E}_\iota(s_i)$, the function nextEvent is defined as*

$$nextEvent(s_j, e(s_i)) = \underbrace{min}_{e_k(s_j) \in \mathcal{E}_\iota(s_j)} \{e(s_i) \leq_{lo} e_k(s_j)\} \tag{2.2}$$

Example 2.9. In the figure 2.1 considering $s_1 \leq_c s_2 \leq_c s_3$, the next event of signal s_2 after the $2^n d$ informative event of s_3 will be:

$$nextEvent(s_2, e_1(s_3)) = e_3(s_2)$$

but the next event of s_3 after the second informative event of s_2 will be:

$$nextEvent(s_3, e_1(s_2)) = e_1(s_3)$$

Definition 2.5.12. *Given a behavior $b = \{s_1, \ldots, s_j, \ldots, s_N\}$ and an event $e_k(s_j) = (v_k, t_k)$, a stall move returns a behavior $b' = stall(e_k(s_j)) = \{s_1, \ldots, s'_j, \ldots, s_N\}$, s.t for all $l \in N$*

$$\sigma_{[t_0, t_{k-1}]}(s'_j) = \sigma_{[t_0, t_{k-1}]}(s_j)$$
$$\sigma_{[t_k, t_k]}(s'_j) = \tau$$
$$\sigma_{[t_{k+l+1}, t_{k+l+1}]}(s'_j) = \sigma_{[t_{k+l}, t_{k+l}]}(s_j).$$

Example 2.10. In figure 2.2(a), b is a behavior with three signals s_1, s_2 and s_3. Now if we insert a stall at $e_1(s_1)$ then the corresponding $b' = stall(e_1(s_1))$ is shown if figure 2.2(b).

$$b = \left\{ \begin{array}{ll} s_1 = & i_1,\ i_2,\ \tau,\ i_3,\ i_4,\ i_5,\ i_6,\ i_7,\ \tau,\ \tau,\ \tau,\ \tau, \ldots \\ s_2 = & i_1,\ i_2,\ \tau,\ i_3,\ i_4,\ i_5,\ \tau,\ i_6,\ i_7,\ \tau,\ \tau,\ \tau, \ldots \\ s_3 = & i_1,\ i_2,\ \tau,\ i_3,\ \tau,\ i_4,\ i_5,\ \tau,\ i_6,\ \tau,\ i_7,\ \tau, \ldots \end{array} \right.$$

(a)

$$b' = \left\{ \begin{array}{ll} s_1 = & i_1,\ \tau,\ i_2,\ \tau,\ i_3,\ i_4,\ i_5,\ i_6,\ i_7,\ \tau,\ \tau,\ \tau, \ldots \\ s_2 = & i_1,\ i_2,\ \tau,\ i_3,\ i_4,\ i_5,\ \tau,\ i_6,\ i_7,\ \tau,\ \tau,\ \tau, \ldots \\ s_3 = & i_1,\ i_2,\ \tau,\ i_3,\ \tau,\ i_4,\ i_5,\ \tau,\ i_6,\ \tau,\ i_7,\ \tau, \ldots \end{array} \right.$$

(b)

Figure 2.2: *An example of $stall$ function*

2.5.2 Procrastination effect

A *procrastination effect* represent the"effect" of a stall move $stall(e_k(s_j))$ on other signals of behavior b in correspondence of events following $e_k(s_j)$ in the lexicographic order. The process will "respond" to the insertion of stalls in some of their signals by "delaying" other signals that are causally related to the stalled signals. Given a behavior b for each stall move on event of b, we have a corresponding set of behaviors (the procrastination effect set)

Definition 2.5.13. *Procrastination in is a point-to-set map that takes a behavior $b' = (s'_1, , s'_N) = stall(e_k(s_j))$ resulting from the application of a stall move on event $e_k(s_j)$ of behavior $b = (s_1, , s_N)$ and returns a set of behaviors $\mathcal{PE}[stall(e_k(s_j))]$ s.t. $b'' = (s''_1, , s''_N) \in \mathcal{PE}[b']$ iff the following three conditions hold*

1. $s''_j = s'_j$;

2. *$\forall i \in [1, N], i \neq j, s''_i \equiv_\tau s'_i$ and $\sigma_{[t_0,t_{l-1}]}(s''_i) = \sigma_{[t_0,t_{l-1}]}(s'_i)$ where t_l is the time stamp of event $e_l(s_i) = nextEvent(s_i, e_k(s_j))$.*

3. *$\exists K$ finite $s.t. \forall i \in [1, N], i \neq j, \exists k_i \leq K, \sigma_{[t_{l+k_i},\infty]}(s''_i) = \sigma_{[t_l,\infty]}(s'_i)$*

Each behavior in $\mathcal{PE}[b']$ is obtained from b' by possibly inserting other stalling events in any signal of b', but only at "later timestamps. Procrastination effect returns a behavior that latency dominates the original behavior.

$$b = \begin{cases} s_1 = i_1, i_2, \tau, \overset{\downarrow}{i_3}, \tau \ldots\ldots\ldots \\ s_2 = i_4, i_5, \tau, i_6, \tau, \tau, i_7 \ldots\ldots\ldots \\ s_3 = i_8, i_9, \tau, i_{10}, \tau, \tau, i_{11} \ldots\ldots\ldots \end{cases} \qquad b' = \begin{cases} s'_1 = i_1, i_2, \,\vert\, \tau, \tau, i_3, \tau \ldots\ldots\ldots \\ s'_2 = i_4, i_5, \,\vert\, \tau, i_6, \tau, \tau, i_7 \ldots\ldots\ldots \\ s'_3 = i_8, i_9, \,\vert\, \tau, i_{10}, \tau, \tau, i_{11} \ldots\ldots\ldots \end{cases}$$

$$b''_1 = \begin{cases} s''_1 = i_1, i_2, \tau, \tau, i_3, \tau \ldots\ldots\ldots \\ s''_2 = i_4, i_5, \tau, \tau, i_6, \tau, \tau, i_7 \ldots\ldots\ldots \\ s''_3 = i_8, i_9, \tau, \tau, i_{10}, \tau, \tau, i_{11} \ldots\ldots\ldots \end{cases} \qquad b''_2 = \begin{cases} s''_1 = i_1, i_2, \tau, \tau, i_3, \tau \ldots\ldots\ldots \\ s''_2 = i_4, i_5, \tau, \tau, i_6, \tau, \tau, i_7 \ldots\ldots\ldots \\ s''_3 = i_8, i_9, \tau, \tau, \tau, i_{10}, \tau, \tau, i_{11} \ldots\ldots\ldots \end{cases}$$

Figure 2.3: *An example of procrastination effect.*

Example 2.11. Let us consider the b given in figure 3.1 as a behavior of a process P. Now if we want to insert a delay at $e_3(s_1)$ as shown by an arrow in the figure then according to definition of $stall(e_3(s_1))$, a new behavior b' will produce. This new behavior is shown in the figure as b'. We can see that the information i_3 which was in the 3^{rd} event (starting from zero) of s_1 in b is now shifted to the 4^{th} event of signal s'_1 in b'. The remaining two signals of b' are same as their corresponding signals in b. Consider that $s_1 \leq_c s_2 \leq_c s_3$. In b, i_1 is happening before i_4 and i_4 is happening before i_8. The same condition is true for i_2, i_5, i_9 and i_3, i_6, i_{10} also. If we observe b' carefully we can see that the causality relation among the events of all the signals is satisfying up to the red line. But after the red line causality relation is not satisfying (i_3 is not happening before i_6 and i_{10}). Due to this reason procrastination effect is necessary to produce a behavior b'' from b' in such a way that it should satisfy the causality relation of P. We can do it by inserting some delays in s'_2 and s'_3 as shown in the figure by b''_1 and b''_2. Both b''_1 and b''_2 are satisfying the above three condition of procrastination effect.

2.5.3 Patient processes and their Compositionality

The key condition in the LI design method is that the intellectual property (IP) blocks must be patient. A patient process can take stall moves on any signal of its behaviors by reacting with the appropriate procrastination effects.

Definition 2.5.14. *A process P is a patient process iff*

$$\begin{aligned} \forall b = (s_1, \ldots, s_N) \in P, \forall j \in [1, N] \\ \forall e_k(s_j) \in \mathcal{E}_\iota(s_j) \\ ((\mathcal{PE}[stall(e_k(s_j))] \cap P \neq \emptyset). \end{aligned}$$

Hence a process is called patient process if it is possible to insert any number of delays in any of its signal without changing any property of the process. The explanation of the above mathematical definition of patient process is as follows: If we apply a delay in a signal which belongs to a behavior b of the process P then according to the definition 2.5.12 and the definition of procrastination effect a set of N-tuples will produce which will satisfy the three conditions of procrastination effect. This set is represented by $\mathcal{PE}[stall(...)]$. All the elements in this set will maintain the causality relation of P but it does not mean that all the elements will also satisfy the properties of P. And we know that if any N-tuple satisfies the properties of a process P then the tuple is called a behavior of P. If any N-tuple from the give set of N-tuples, is a behavior of P then it can be say that after inserting the delay it is possible to maintain the original properties of the process. If these is possible for inserting delays in any signal belongs to any behavior of P, then P is said to be a patient process.

Each module in the LI design must be patient. And the procedure to convert each IP core as a patient process is already discussed in chapter 1. But we have to clear some confusion yet. If all the modules in a SoC are patient then what is the guarantee that the whole system will be patient? This is the most important question because LI design always consider that the system must be patient. Carloni et el. [2] proposed some theorems to prove that the composition of patient processes is also a patient process. The theorems are explained below:

Lemma 2.5.1. *Let P_1 and P_2 be two patent processes. Let $b_1 \in P_1, b_2 \in P_2$ be two behaviors with the same lexicographic order s.t. $b_1 \equiv_\tau b_2$. Then, there exist a behavior $b' \in (P_1 \cap P_2), b_1 \equiv_\tau b' \equiv_\tau b_2$.*

Explanation 2.5.1. The proof of this lemma is given in [2]. Given that b_1 is latency equivalent with b_2 but this does not mean that b_1 and b_2 are always equal. When b_1 and b_2 are equal, b' will just same as b_1 and b_2 because $b' \in (P_1 \cap P_2)$. So definitely $b_1 \equiv_\tau b' \equiv_\tau b_2$. But when b_1 and b_2 are just latency equivalent, it is not possible to say directly that b' is latency equivalent with both b_1 and b_2. Fortunately there is a procedure to produce a behavior $b_1^* \in P_1$ from b_1 and a behavior $b_2^* \in P_2$ from b_2, s.t. $b_1^* \equiv_\tau b_1$ and $b_1^* \equiv_\tau b_1$ and $b_1^* = b_2^*$. In this case we can say that $b' = b_1^* = b_2^*$ and hence we can also say that $b_1 \equiv_\tau b' \equiv_\tau b_2$.
The figure 2.4 shows an example for explaining the procedure by which b_1^* and b_2^* are producing. Let b_1 and b_2 shown in the figure 2.4(a) be the behavior of P_1 and P_2 respectively. Now to produce b_1^* and b_2^* from b_1 and b_2, s.t. $b_1^* = b_2^*$, we have to proceed step by step. In each step we align the first unaligned pair of the corresponding events.

In step 1 the first unaligned pair of corresponding events is $pir_1 = [e_1(s_1 \in b_1), e_3(s_2 \in b_2)]$. These two events are shown by blue arrows in the figure 2.4(a). To align these two events we have to shift the event $[e_1(s_1 \in b_1)]$ by two position. The only way to do this is to insert delays at $[e_1(s_1 \in b_1)]$. After inserting one delay at $[e_1(s_1 \in b_1)]$ we got a behavior $b_1'' \in (\mathcal{PE}(stall(e_1(s_1 \in b_1))) \cap P_1)$. Figure 2.4(b) showing the newly created b_1'' with the original b_2. Now comparing b_1'' and b_2 we can see that the two events of pir_1 are still not aligned. So we insert another delay at $[e_2(s_1'' \in b_1'')]$. As a result we got a behavior $b''''' \in (\mathcal{PE}(stall(e_2(s_1'' \in b_1''))) \cap P_1)$ as shown in figure 2.4(c). Now we can see that the two events of pir_1 are aligned with each other. So step 1 finish.

In step 2, the first unaligned pair from figure 2.4(c) is $pir_2 = [e_4(s_2 \in b''''_1), e_3(s_2 \in b_2)]$. For aligning them we have to insert one delay at $[e_4(s_2 \in b''''_1)]$. The resulting behavior $b_2'' \in (\mathcal{PE}(stall(e_3(s_2 \in b_2''))) \cap P_2)$ is shown in the figure 2.4(d). Comparing the two behavior b_1'''' and b_2'' we can see that the two unaligned events of pair pir_2

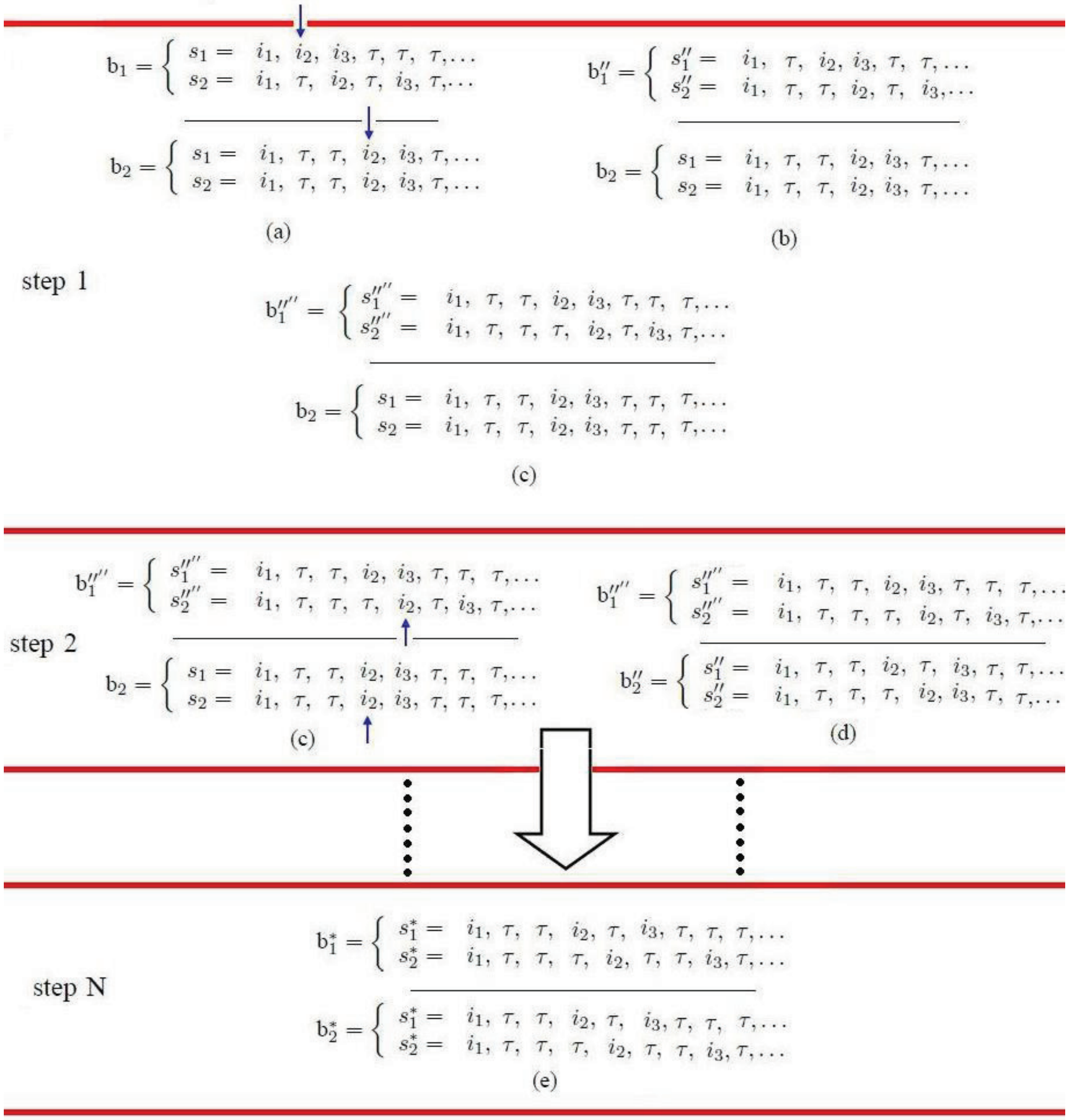
$\mathrm{b}_1 = \begin{cases} s_1 = & i_1,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2 = & i_1,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ldots \end{cases}$
$\mathrm{b}_2 = \begin{cases} s_1 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ldots \\ s_2 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ldots \end{cases}$
(a)
$\mathrm{b}_1'' = \begin{cases} s_1'' = & i_1,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ldots \\ s_2'' = & i_1,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ldots \end{cases}$
$\mathrm{b}_2 = \begin{cases} s_1 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ldots \\ s_2 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ldots \end{cases}$
(b)
step 1
$\mathrm{b}_1''' = \begin{cases} s_1''' = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2''' = & i_1,\ \tau,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ldots \end{cases}$
$\mathrm{b}_2 = \begin{cases} s_1 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \end{cases}$
(c)
step 2
$\mathrm{b}_1''' = \begin{cases} s_1''' = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2''' = & i_1,\ \tau,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ldots \end{cases}$
$\mathrm{b}_2 = \begin{cases} s_1 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2 = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \end{cases}$
(c)
$\mathrm{b}_1''' = \begin{cases} s_1''' = & i_1,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2''' = & i_1,\ \tau,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ldots \end{cases}$
$\mathrm{b}_2'' = \begin{cases} s_1'' = & i_1,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ \tau,\ldots \\ s_2'' = & i_1,\ \tau,\ \tau,\ \tau,\ i_2,\ i_3,\ \tau,\ \tau,\ldots \end{cases}$
(d)
step N
$\mathrm{b}_1^* = \begin{cases} s_1^* = & i_1,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2^* = & i_1,\ \tau,\ \tau,\ \tau,\ i_2,\ \tau,\ \tau,\ i_3,\ \tau,\ldots \end{cases}$
$\mathrm{b}_2^* = \begin{cases} s_1^* = & i_1,\ \tau,\ \tau,\ i_2,\ \tau,\ i_3,\ \tau,\ \tau,\ \tau,\ldots \\ s_2^* = & i_1,\ \tau,\ \tau,\ \tau,\ i_2,\ \tau,\ \tau,\ i_3,\ \tau,\ldots \end{cases}$
(e)

Figure 2.4: *An example to explain Lemma 2.5.1*

has been aligned. So step 2 finish.

In this way within a finite number of steps, it is possible to align all the corresponding events of b_1 and b_2. But since we are aligning by inserting delays, hence according to procrastination effect and the properties of patient process finally we get two behaviors b_1^* and b_2^* s.t. $b_1^* = b_2^*$ and $b_1^* \equiv_\tau b_1$, $b_2^* \equiv_\tau b_2$ and $b_1^* \in P_1$, $b_2^* \in P_2$.

Theorem 2.5.2. *If P_1 and P_2 are two patient processes, then $(P_1 \cap P_2)$ is also a patient process.*

Explanation 2.5.2. The proof of this theorem is given in [], here we are giving a explanation of the proof. Let $P = (P_1 \cap P_2)$, now to proof that P is a patient process, we have show the following:

- *"for any behavior $b \in P$, if we insert a delay in any of its signal (say z) then we have to got a behavior $\hat{b}$ s.t. $\hat{b} \in \mathcal{PE}(stall(z))$ and $\hat{b} \in P$."*

Let $b \in P, b_1 \in P_1$ and $b_2 \in P_2$ are the behaviors of P, P_1 and P_2 respectively. Consider also that $b_1 = b_2$. Now according to the lemma 2.5.1, $b \equiv_\tau b_1 \equiv_\tau b_2$. Let the signal where we want to inset the delay is s_j and the event is $e_k(s_j)$. Note that if $s_j \in P$ then it must be true that $s_j \in P_1$ and $s_j \in P_2$.

Now let us consider that after inserting a delay at $e_k(s_j)$ we get a behavior b'' s.t. $b'' \in \mathcal{PE}(stall(e_k(P[s_j])))$. Similarly let $b_1'' \in \mathcal{PE}(stall(e_k(P_1[s_j])))$ and $b_2'' \in \mathcal{PE}(stall(e_k(P_2[s_j])))$. Also consider that $b_1'' \in P_1$ and $b_2'' \in P_2$. It is not always true that $b'' = b_1'' = b_2''$. But applying the procedure given in the explanation of lemma 2.5.1, it is possible to find three behaviors b^*, b_1^* and b_2^* from b'', b_1'' and b_2'' respectively, s.t. $b^* = b_1^* = b_2^*$. Now since $b_1^* \in P_1$ and $b_2^* \in P_2$ it can be proof that $b^* \in P$. But to prove P as a patient process we still need to show that $b^* \in \mathcal{PE}(stall(e_k(P[s_j])))$. If we observe the procedure of getting b^* from b'' we can see that b^* is obtained by just inserting delays after the $nextEvent(P(s_i''), e_k(P[s_j'']))$, where s_i'' stands for all the signals in b''. So it means that $b^* \in \mathcal{PE}(stall(e_k(P[s_j])))$. Hence $P = (P_1 \cap P_2)$ is a patient process.

Theorem 2.5.3. *For all process P_1, P_2, P_1', P_2', if $P_1 \equiv_\tau P_1'$ and $P_2 \equiv_\tau P_2'$ then $P_1 \cap P_2) \equiv_\tau (P_1' \cap P_2')$.*

Theorem 2.5.4. *For all strict processes P_1, P_2 and all patient processes P_1', P_2', if $P_1 \equiv_\tau P_1'$ and $P_2 \equiv_\tau P_2'$, then $(P_1 \cap P_2) \equiv_\tau (P_1' \cap P_2')$.*

The proof of the above two theorem are given in [2].

From the above theorems we can say that it is possible to replace all the processes in a system of strict processes by corresponding patient process and the resulting system will be patient process and latency equivalent to the original one. This is the core idea of LI design: take a design based on the assumption that computation in one functional module and communication among modules "take no time" (synchronous hypothesis), i.e., the process corresponding to the functional modules and their composition are strict and replace it with a design where communication does take time and as a result, signals are delayed, but without changing the sequence of informative events observed at the system level.

2.5.4 Patient Process and Relay Stations

We already mentioned that LI design is based on combining the IP cores. Each IP core must make patient and if all the modules (patient IP cores) in the system are patient then the entire system will also patient. But actually in LI design relay station are inserted into the channels to divide them into segments. So we cannot just say that a system designed by LI design contains only patient IP cores but it also contains relay stations. On other words a system designed by LI design is a combination of patient IP cores and relay stations. Now it clear that to be the system patient the relay stations must also be patient. In this section we will describe relay station's and there properties.

2.5.4.1 Channel and Buffers

The tagged signal model provides the notion of channel to formalize the composition of processes [**?**]. A channel is a connection constraining two signals to be identical.

Definition 2.5.15. *A channel* $\mathcal{C}(i,j) \subset S^N, i, j \in [1, N]$ *is a process, s.t.*

$$b = (s_1, \ldots, s_N) \in \mathcal{C}(i,j) \Leftrightarrow s_i = s_j.$$

A channel is not a patient process [2] because it lacks the capacity of storing an event and delaying its communication between two processes. Hence to formally