F E R M A T

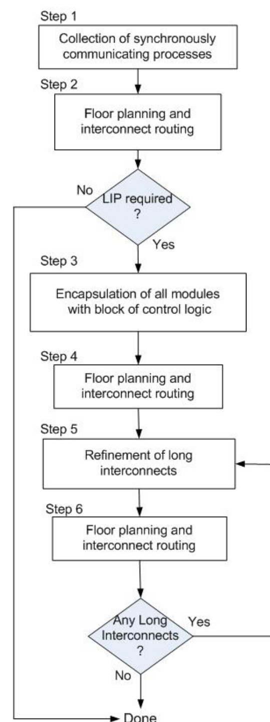
Formal Engineering Research using Methods, Abstractions and Transformations

Technical Report No: 2005-02

Presentation and Formal Verification of a Family of Protocols for Latency Insensitive Design

Syed Suhaib, David Berner, Deepak Mathaikutty,
Jean-Pierre Talpin, Sandeep Shukla

{ssuhaib,damathai,shukla}@vt.edu
{david.berner,talpin}@irisa.fr



Abstract -- With increasing clock frequencies in the gigahertz ranges, the distance signals travel during one clock cycle decreases, making a synchronous clock routing throughout the chip impossible. Therefore, interconnect latency needs to be tolerated, and in IP based design various synchronous logic blocks need to be connected via long interconnects that have communication delays of multiple clock cycles. Sometimes two communicating IPs also belong to different clock domains, making their interaction difficult to implement. Latency insensitive protocols were introduced for cases when the interconnect delay was the main concern but the IPs were under same clock domain. Here we generalize the issue of solving the problem of long interconnects in between IPs for a single-clock domain as well as for a multi-clock domain. We also use a formal verification strategy for making sure that as we gradually refine our protocol, we do not become functionally inequivalent to the original synchronous reference design.

Virginia Tech 1872

Presentation and Formal Verification of a Family of Protocols for Latency Insensitive Design

Syed Suhaib, David Berner, Deepak Mathaikutty,
Jean-Pierre Talpin, Sandeep Kumar Shukla

Virgina Tech, USA
INRIA, France

{ssuhaib,damathai,shukla}@vt.edu
{dberner,talpin}@irisa.fr

Contents

1 Introduction **2**
1.1 Main Contributions 3

2 Related Work **3**

3 Preliminary Definitions & Notations **5**
3.1 SPIN 6

4 Protocol Details and Implementation **7**
4.1 Carloni's Latency Insensitive Protocol 7

5 Our Approach **10**
5.1 Eliminating Relay Stations 10
5.2 Multi-Clock 15

6 Verification of Protocols **18**

7 Conclusion and Future Work **19**

List of Figures

1 Equalizer Example 8
2 LI system as proposed by Carloni 9
3 Refinement 12
4 LI system with Bridge 13
5 LI and Synchronous model 14
6 Multi-clock LI Implementation 17
7 Latency Equivalence Checking of Synchronous and LI System . . 18

List of Tables

Abstract

With increasing clock frequencies in the gigahertz ranges, the distance signals travel during one clock cycle decreases, making a synchronous clock routing throughout the chip impossible. Therefore, interconnect latency needs to be tolerated, and in IP based design various synchronous logic blocks need to be connected via long interconnects that have communication delays of multiple clock cycles. Sometimes two communicating IPs also belong to different clock domains, making their interaction difficult to implement. Latency insensitive protocols were introduced for cases when the interconnect delay was the main concern but the IPs were under same clock domain. Here we generalize the issue of solving the problem of long interconnects in between IPs for a single-clock domain as well as for a multi-clock domain. We also use a formal verification strategy for making sure that as we gradually refine our protocol, we do not become functionally inequivalent to the original synchronous reference design.

1 Introduction

In the current System-on-chip(SoC) based design, reduced time-to-market demands efficient reuse of complex components. This has led to the idea of developing libraries of Intellectual Properties (IP)s or reusable components. The integration of such complex IPs on SoC and communication between them has shifted the performance bottleneck of the system from computation to communication. With clock frequencies of these IPs in the multiple gigahertz range, and the interconnection distances staying almost constant with the chip size, we have hit the limit, where the signal propagation distance during one clock cycle is shorter than the longest wire. The solution is to distinguish short interconnects from long interconnects, and use the long interconnects preferably on low traffic inter-component communication. Intelligent repeaters [1, 2, 3] on their interconnects make sure that no value sent through them are lost. This approach is known as *Latency Insensitive design*. The idea is to create a design from a specification without assuming any latency in the interconnects, such that the resulting design is latency equivalent to the specification. Informally, latency equivalence means that given same input signals, the output signals have the same ordering of events except for interspersed absent events. Formal definition of this follows in the later sections.

While this is solving the fundamental problem, there are still issues that make the application unsatisfactory or at least inconvenient. (i) these repeaters or relay stations are additional components that have to be placed on the floor plan, and consequently require changes in the original placement and routing. Several iterations of a lengthy process may be necessary in order to reach a state where all timing constraints are met. (ii) in such a completely synchronous design all components have to run the same clock frequency. Reusing IPs from different origins however often imply that these IPs are designed to run at varying frequency ranges. Furthermore, only a few components have to run with the fastest clock, others could run at a much slower clock, while still meeting all their performance

constraints.

In this paper, we illustrate a technique with an example to show how to successively eliminate these two constraints, reducing overhead caused to the designer in terms of design time, while at the same time leaving them more room in their power budget. In the first step, we show how to get rid of the relay stations, by putting some small extra intelligence into the component interface. In the next step, we show an extension of these interfaces that allow for components to have different clocks, but are defined rational clock relations. Both of these improvements can be done systematically, while minimizing changes in the actual component and therefore eliminating additional sources of errors. We employ formal verification as our strategy for ensuring that as we gradually refine our protocols, we maintain functional equivalence to the original intent of the latency insensitive design.

1.1 Main Contributions

The main contributions of this paper are as follows:

- Formal modeling and validation of existing latency insensitive protocols [3].
- New refinement-based approach to single-clock latency insensitive systems.
- Formal modeling and validation of our new approach.
- New approach to latency insensitive systems for multi-clock systems.
- Formal modeling and validation of the multi-clock latency insensitive system.

2 Related Work

Latency insensitive protocols (LIP) for systems with long interconnection delays (i.e. greater than one clock cycle) were proposed by Carloni et al [1, 2, 3] for single-clock SoCs. All processes with long interconnects are encapsulated in a

wrapper to derive a process that is latency equivalent to the actual process, without having to modify the internals of the original IP. Relay stations are added along the long interconnection wires like repeaters for successful data transfer. They contain at least two registers and a small control logic. The insertion of these relay stations increases the number of elements to route and requires additional space on the chip for placement. Once it is determined where relay stations have to be added based on the length of the wires, placement and routing of the entire chip design now including the relay stations have to be redone. Several iterations for placement and routing are needed in order to get a configuration that satisfies all interconnection constraints.

All components of such latency insensitive (LI) designs are assumed to operate with the same clock. Singh and Theobald generalize the LI theory for GALS systems [4]. In their approach, complex FSMs control all input and output signals. The communication network is implemented as an asynchronous system to connect modules with different clocks. Overall this approach is associated with heavy penalties in terms of implementation costs and performance.

Casu and Macchiarulo show how to reduce chip area compared to Carloni's approach [5]. They use a smart scheduling algorithm for the functional block activation and substitute the relay stations with simple flip-flops. One disadvantage of this approach is that the schedule has to be computed a priori and depends on the computation in the process. If any change is made in any process, it may result in change of flow of tokens. In this case, the schedule has to be recalculated which is an expensive procedure.

In our approach, we do not face the problem of implementing complex FSMs, asynchronous communication protocols, or scheduling computations. Instead we propose to solve the latency problem without using relay stations along the wires. We also generalize the solution for multi-clock systems where communication is done based on a global/communication clock and the process interfaces bridge the global and the local clock to ensure correct functionality of the processes.

We formally model the family of protocols that we propose and verify them for latency equivalence with a corresponding synchronous system.

3 Preliminary Definitions & Notations

Let V be the set of data values and, T be a countable set of time stamps. Unless otherwise specified, in this paper, we assume T = N = set of natural numbers. An event $e \in V \times T$ is an occurrence of a data value with a particular time stamp. However in the systems, we consider, a special event called *absent event* denoted by τ may occur[1]. Therefore the set of all events is denoted by E, where $\tau \in$ E and for all other e $\in$ E e $\in$ V $\times$ T. When e $\in$ V $\times$ T it e is called an *informative event*. A *signal* s is defined to be a sequence of events, often denoted as $e_1e_2e_3\cdots e_n\cdots$ (a signal is possibly infinite) where $e_i \in E$.

For the preliminary definitions, if s is a signal, s[i] denotes the ith event, hence either s[i] $\in V \times T$ or s[i] = τ. The set of all signals is denoted by S. We also distinguish from all signals in the system, some special kind of signals, like the *Stall signals* and *Control signals*. A stall signal s_t is a sequence of boolean events, i.e., $s_t[i] \in Bool \times T$ The set of all stall signals is denoted by S_T. A control signal s_{rw} is a sequence of events from $\{R,W\} \times T$ set, i.e., $s_{rw} \in \{R,W\} \times T$. The set of all control signals is denoted by S_{RW}. In our system, IPs are hardware modules that map input signals to output signals, therefore in this paper we refer to them as processes. A process p is a function $S^n \longrightarrow S^m$ where n, m are natural numbers.

In the rest of this section, we define a few terms and notations that are used throughout the paper.

Definition 1 *Given two tuples of m and n elements,* $\bigodot$ *creates a tuple of m+n elements.*

$< a_1,a_2,\ldots,a_n > \bigodot < b_1,b_2,\ldots,b_m > = < a_1,a_2,\ldots,a_n,b_1,b_2,\ldots,b_m >$

Definition 2 *Given two tuples of n events and n signals respectively,* $\bigoplus$ *creates a*

[1] It may be caused due to lack of valid data in the producer or the consumer's request to delay a transmission

tuple of n signals with the n events appended to the n signals.

$< e_1, e_2, \ldots, e_n > \bigoplus < s_1, s_2, \ldots, s_n > \; = \; < e_1 \oplus s_1, e_2 \oplus s_2, \ldots, e_n \oplus s_n >$ *where* $e_1 \oplus s_1 = s''$ *is a signal such that* $s''_1 = e_1 s_1$

Definition 3 *Let* p_1 *and* p_2 *be two processes where,* $p_1 : S \rightarrow S$ *and* $p_2 : S \rightarrow S$ *then* $p_2 \circ p_1(s) = s'$ *is called a composition.*

Definition 4 *$Vect_{i=1}^{n}$(exp(i)) = <exp(1),exp(2),$\cdots$,exp(n)> where, exp(i) is an expression s.t. exp(k) is a textual replacement of i by k.*

Definition 5 *Let function* $\mathcal{F} : S \rightarrow S$ *be defined as,* $\mathcal{F}(s)$ = *f(s,1,n) where,*

$$f(s,i,n) = \begin{cases} f(s,i+1,n), & if\ s[i] = \tau \\ s[i], & if\ (i = n) \\ s[i] \oplus f(s,i+1,n), & otherwise \end{cases}$$

$\mathcal{F}$ *takes a signal as input and outputs a signal dropping all* τ *events, but preserving all informative events. If s is infinite, then the above definition changes slightly.*

Definition 6 Latency Equivalence: *Two signals* s_1 *and* s_2 *are said to be latency equivalent,* $s_1 \equiv_e s_2 \Leftrightarrow \mathcal{F}(s_1) = \mathcal{F}(s_2)$. *If the output signals of two systems are latency equivalent given the same input signals, then these systems are said to be latency equivalent.*

3.1 SPIN

SPIN [6] is a model checker used extensively for formal verification of systems. SPIN is used to trace logical design errors and to check the consistency of the specification. Its basic building blocks include asynchronous processes, message channels, synchronizing statements, and structured data. We use these basic blocks to write synchronous models. The communication is done through global variables. Since the processes run asynchronously in SPIN, we synchronize the execution of all processes with a central clock controller in order to make our

model behave like a synchronous system. The execution of each component depends on a flag set by the clock controller, which is reset by the component at the end of its execution. After which the clock controller waits for all components to finish execution before it starts the next cycle. As in a real synchronous system, the duration of a cycle (i.e. the maximum system frequency) is determined by the slowest component.

4 Protocol Details and Implementation

4.1 Carloni's Latency Insensitive Protocol

In the LI approach of Carloni et al each process is encapsulated in a wrapper (called "shell") that is forming the interface to other shells. Logic blocks called relay stations are placed along the wires that are longer than the signal propagation distance during one clock cycle. These relay stations work as pipeline blocks to send data from one shell to another over a long wire. The shell reads all incoming values, filters out empty events, and feeds the process with valid input events. If any input signal does not have a data available, the process is stalled until all input signals are present. When the process completes computation, the shell writes events to the output wires, but only if the respective connected component is accepting events.

We present our LI systems based on this approach. So first we made a formal model of the Carloni LIP (Figure 2). In this implementation we put all functionality of the wrapper into an *equalizer* process. An equalizer is modeled such that it reads all the input events from the incoming signals. An input is considered only when it is associated with an informative event. The equalizer is only forwarding data when all its input signals provide informative events. If an input signal sends an absent event in one cycle, the equalizer sends a stall signal via a feedback to all processes that had sent an informative event and it suspends the execution of other processes. When the process itself is disabled by its equalizer, it is producing τ events at its outputs, therefore causing the components before to wait until it can

resume operation. Formal definition of the equalizer is given below:

Definition 7 *An Equalizer function* $\mathcal{E} : S^n \longrightarrow (S^n \times {S_T}^n)$ *is defined as:*

$$\mathcal{E}(s_1, \cdots, s_n) = f(s_1, \cdots, s_n, 1, 1, \cdots, 1) \text{ where,}$$

$$f(s_1, \cdots, s_n, i_1, i_2, \cdots, i_n) =$$
$$if(\exists_{j=1}^{n} (s_j[i_j]) = \tau)$$
$$then$$
$$< \tau, \tau, \ldots, \tau > \odot Vect_{j=1}^{n}(exp_1(j)) \bigoplus f(s_1, \cdots, s_n, Vect_{j=1}^{n}(exp_2(j)))$$
$$else$$
$$Vect_{j=1}^{n} (s_j[i_j]) \odot < F, \ldots, F > \bigoplus f(s_1, \cdots, s_n, Vect_{j=1}^{n}(i_j + 1))$$

$$exp_1(j) : if\ (s_j[i_j]) = \tau \text{ then } F \text{ else } T$$
$$exp_2(j) : if\ (s_j[i_j]) = \tau \text{ then } i_j + 1 \text{ else } i_j$$

The function f takes arguments $i_1, i_2, \cdots, i_n$ *to index* $s_1, s_2, \cdots, s_n$ *to access the respective events.*

Let us consider an example of an equalizer and three processes: A, B, C (shown in Figure 1). Each of the processes connect to the equalizer with signals $\{s_1, s_2, s_3\}$. Also the equalizer outputs stall signals $stall_A$, $stall_B$, $stall_C$ to the respective processes. Let us assume that processes B and C produce informative events whereas process A outputs an absent event. In this scenario, the equalizer will set $stall_B$ and $stall_C$ signals to stall the processes B and C in the next cycle and output absent

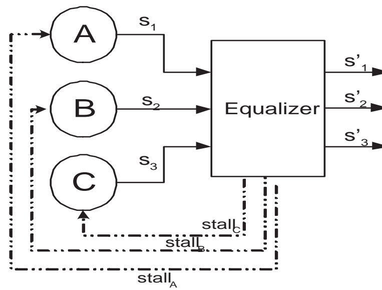


Figure 1. Equalizer Example

events on s'_1, s'_2, s'_3. Once it receives an informative event on s_1, it will remove the stalls that were set and produce the corresponding outputs. The equalizer modeled in the LI implementation also consists of a stall signal generator that connects to the main process. An example of this LI approach is shown in Figure 2. In the example, there are two processes P and Q that are encapsulated in wrappers. A random generator produces random informative events and sends them to the shell of process P. The shell reads the data from its inputs and performs some computation and sends an output to Q via a relay station. In this example, the process Q simply prints the data it reads and is forwarding it to its output. The setup is as follows: EqP and EqQ are the equalizers of the corresponding processes. The global clock is connected to all the processes. There is a delay of two cycles on the interconnection between shell P and shell Q. Therefore, one relay station is placed on the interconnect to permit successful transfer of data from process P to process Q. The solid arrows denote the transfer of values where as the dotted lines denote the transmission of events. Each process has enable signals set at the next process station because if a process is not ready to accept data, it can disable the previous processes to stop them from producing more tokens. This feedback mechanism is preventing the usage of infinite buffers. Note that we assume that processes considered here are functions and hence deterministic and they are monotonic as well [7]

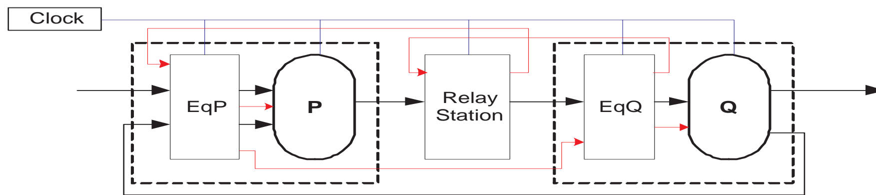


Figure 2. LI system as proposed by Carloni

5 Our Approach

5.1 Eliminating Relay Stations

First step to generalize the original LI approach is to eliminate the need for relay stations. There are several advantages to this. First of all, we reduce the number of elements in the system, which simplifies routing and placement. Also, we reduce the process of re-iterating over the routing and placement after estimating the actual delays since we do not have to insert new components in between wires for long interconnections.

The basic idea of our approach is that if a event between two components takes two clock cycles, we only send one value every second clock cycle thereby adapting the communication speed between components to their distance. An outgoing interface takes care of this restriction, and stalls the sending component whenever it delivers values too fast.

While this eliminates all relay stations, it however slows down the system significantly due to an increasing number of stalls. We eliminate this slowdown by additional communication lines. The number of interconnects needed depend on the interconnect delay. For example, if there is an interconnect delay of n cycles, then n interconnects are placed in-between the processes. Note that in most cases n is in the range of 2 or 3. These additional wires an extra cost, but in modern processes it is not very expensive to add parallel wires. Only in rare cases of very sparse communication does it make sense to use a LI implementation for more than three cycle distances, since stalling is bound to slow down the rest of the system otherwise.

To ensure that the events are correctly transfers from one process to another through long interconnects, we add extra interconnects that are bounded by a *splitter* on the source and a *merger* on the sink. The splitter is implemented at the output of a process, and it transfers events on the corresponding interconnect. The splitter only puts one event on one of the output interconnects and τs are placed

on the rest of the signals at a particular time stamp.

Definition 8 *n-Splitter $\mathcal{H}$ is a process, s.t., $\mathcal{H}(s) = (s'_1, s'_2, \ldots, s'_n)$ where*

$$(s'_1, s'_2, \ldots, s'_n) = h(x :: y, n, 1) \text{ and,}$$

$$g(n) = \begin{cases} \tau, & if\ n = 1 \\ \tau \bigodot g(n-1), & otherwise \end{cases}$$

$$f(x,n,i,j) = \begin{cases} x \bigodot g(n-j), & if\ i = 1 \\ \tau \bigodot f(x,n,i-1,j+1), & otherwise \end{cases}$$

$$h(x :: y,n,i) = \begin{cases} f(x,n,i,1) \bigoplus h(y,n,1), & if\ i = n \\ f(x,n,i,1) \bigoplus h(y,n,i+1), & otherwise \end{cases}$$

Here, $s = x :: y$ means that x is the first event of the signal s and y is rest of the signal.

On contrary to the splitter, we implement a *merger* that reads the corresponding events from these interconnects based on the placement of events by the splitter. We fomalize the splitter below and also state the lemma about the composition of the splitter and the merger. There is a splitter-merger combination for each long interconnect in our LI implementation and we call this composition a bridge.

Definition 9 *Merger $\mathcal{M}$ is a process, s.t. $\mathcal{M}(s_1, s_2, \ldots, s_n) = s'$ where*

$$s' = g(\ (x_1 :: y_1, x_2 :: y_2, \ldots, x_n :: y_n),\ n,\ 1) \text{ and,}$$

$$f(x :: y,n,i) = \begin{cases} x, & if\ i = n \\ f(y,n,i+1), & otherwise \end{cases}$$

$$g((x_1 :: y_1, x_2 :: y_2, \ldots, x_n :: y_n),n,i) = \begin{cases} f((x_1,x_2,\ldots,x_n),n,i) \oplus & \\ g((y_1,y_2,\ldots,y_n),n,1), & i = n \\ f((x_1,x_2,\ldots,x_n),n,i) \oplus & \\ g((y_1,y_2,\ldots,y_n),n,i+1), & otherwise \end{cases}$$

$$s_1 = x_1 :: y_1,\ s_2 = x_2 :: y_2,\ \cdots\ s_n = x_n :: y_n,$$

Definition 10 *If $\mathcal{M} \circ \mathcal{H}(s) = s''$, then $\mathcal{F}(s) = \mathcal{F}(s'')$*

proof sketch: *From Definition 8, when the n-splitter is given an input signal s, the n output signals are as follows:*

$$\begin{aligned}
s'_1 &= e_1\tau^{n-1}e_{n+1}\tau^{n-1}e_{2n+1}\tau^{n-1}\ldots \\
s'_2 &= \tau e_2\tau^{n-1}e_{n+2}\tau^{n-1}e_{2n+2}\tau^{n-1}\ldots \\
s'_3 &= \tau\tau e_3\tau^{n-1}e_{n+3}\tau^{n-1}e_{2n+3}\tau^{n-1}\ldots \\
&\vdots \\
s'_n &= \tau^{n-1}e_n\tau^{n-1}e_{2n}\tau^{n-1}e_{3n}\tau^{n-1}\ldots
\end{aligned}$$

From Definition 9, when the merger is given the n output signals of the splitter $(s'_1, s'_2, \ldots, s'_n)$ as input. If the resultant output signal is s'' then $\mathcal{F}(s) = \mathcal{F}(s'')$.

Definition 11 *A Bridge $\mathcal{B}$ is single-input, single-output process p, built by sequentially composing a Splitter and a Merger ($\mathcal{H}$-$\mathcal{M}$), where $p(s) = s'$ and $s' = M \circ H(s)$.*

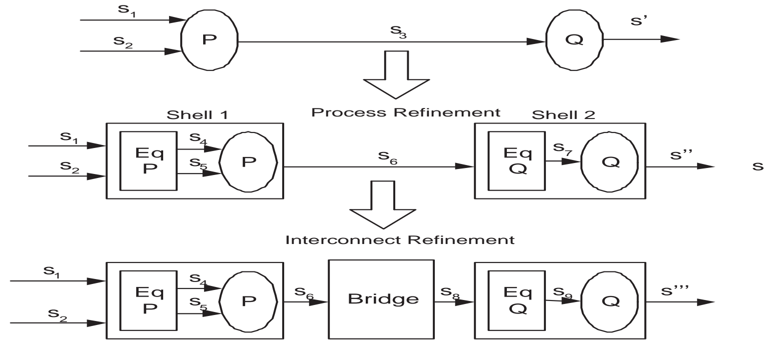


Figure 3. Refinement

In definition 12, we define our two-step refinement process for a single-clock LI composition.

Definition 12 **Single Clock LI Composition:** *Given a single clock system Π_s with N synchronous processes, m input signals and n outputs signals, we define a k-delayed LI composition with the following steps:*

Step 1 (Process Refinement): Every process $p \in \Pi$ is encapsulated into a shell by sequentially composing them with an equalizer $\mathcal{E}$.

Step 2 (Interconnect Refinement): Every k-delayed interconnect is refined by introducing a bridge between the source and the sink of the interconnect.

The refinement process is shown in Figure 3 with two processes P and Q. s_1, s_2 are inputs to the process P. s_3 is a long interconnect that connects process P and Q and s' is the output of Q. At the first step, the processes are encapsulated in a shell which is done by composing each process with an Equalizer. In the next step, the long interconnect s_3 is refined with the bridge as described in definition 11.

Definition 13 *The shell encapsulation of process P is latency equivalent to the output of Process P for any given input signal.*

Proof sketch: Follows from [1] and Definition 7.

Figure 4 shows the structure of this implementation based on the previous example in section 4.1. Each component still has an equalizer for the inputs, however on the output side, long interconnects are connected by the bridge. The throughput of the system is exactly the same as the throughput of a system with relay stations. Let us

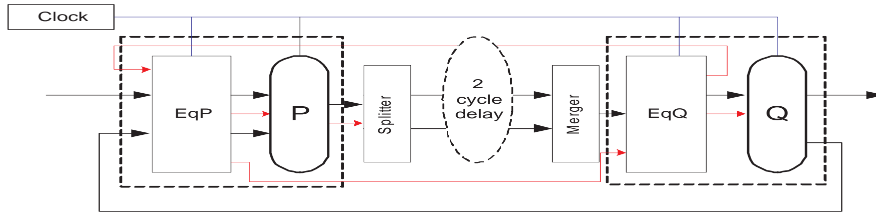


Figure 4. LI system with Bridge

consider two systems: Π and Π' as shown in Figure 5. Π is a synchronous system with two processes P and Q. P has two input signals, s_1 and s_4 and an output signal s_2. In Π, the interconnect has no latency and hence, s_2 and s_3 are same signals input to process Q. Q gives an output s and another signal s_4 to process P. Π' is the LI implementation of the synchronous model Π with processes P' and Q'

which are encapsulations of P and Q of Π [1]. Π' has a bridge in-between process P' and Q' as defined in definition 11. The signals of Π' are connected similar to system Π. $s_1 = s'_1$ are the input signals to both the systems.

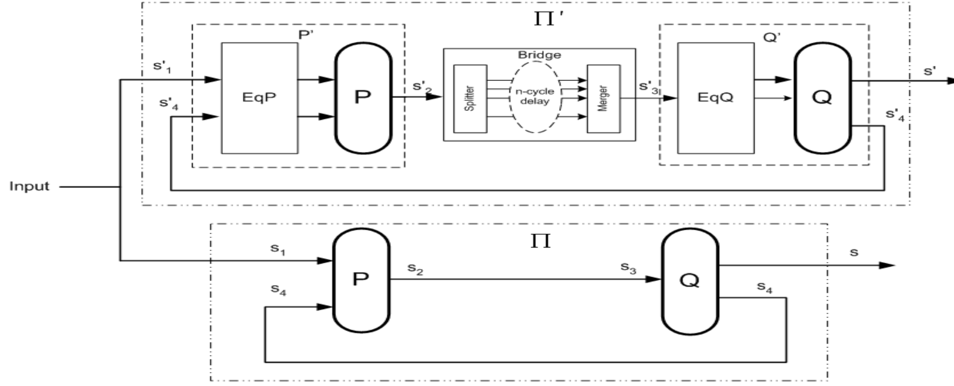


Figure 5. LI and Synchronous model

Theorem 14 *Two systems* Π, Π' *are latency equivalent.*

Proof sketch: We use circular reasoning [8] to show the prove sketch. Initially, all signals are latency equivalent. When the first informative event enters the system at s_1, s'_1, the corresponding event at the output of P and P' are same assuming that a valid event is present at s_4 and s'_4. Therefore, from Lemma 13, $\mathcal{F}(s_2) = \mathcal{F}(s'_2)$ after the first informative event is processed by both P, P'. Also $s_2 = s_3$ in system Π since they are the same signal, and we know from definition 11 and lemma 10 that $\mathcal{F}(s'_2) = \mathcal{F}(s'_3)$. Similarly, for processes Q, Q', $\mathcal{F}(s_3) = \mathcal{F}(s'_3)$ at the first informative event. Therefore, at the output of Q,Q', $\mathcal{F}(s_4) = \mathcal{F}(s'_4)$ at the first event. At every t^{th} informative event on s_2, s'_2, the value will depend on the $t-1^{th}$ event of s_4, s'_4 which will be same. Therefore, event on s_2, s'_2 will always be the same at t^{th} event and hence, $\mathcal{F}(s_2) = \mathcal{F}(s'_2)$ will always be true after t^{th} informative events from the input. The same can be said for the output of Q, Q'. Therefore, all the signals will be latency equivalent after t informative events are processed. Hence, the two systems are latency equivalent.

5.2 Multi-Clock

An LI system as defined by Carloni is still a synchronous system where all components are connected to the same clock and work with the same speed. We generalize this definition towards a multi-clock implementation where we allow components with different clocks connected via arbitrarily long wires. At this time, however, we are only permitting the use of components with defined, rational clock relations. This approach therefore makes it possible to connect components where one is for example three times faster than the other or two components that have for example a clock ratio of 11:38 compared to the communication/-global clock. The global clock is the fastest clock in the system. We assume that each process has a read phase and a write phase. In the read phase, a process reads the value on its input signals and in the write phase, a process writes the value on its output signal.

We modify two components of our LI system to refine it for a multi-clock system. We extend our original Equalizer process to an Extended Equalizer as shown in definition 15. The Extended Equalizer has the information of the clock cycle of the actual process. Therefore, in the multi-clock system, at most one value is provided to the computational process during its read phase by the Extended Equalizer. The process does its processing and in the write phase writes a value. This value is read by the Extended Splitter which is an extended version of the Splitter from the previous section. The Extended Splitter reads the value from the main process during the write phase of the process. Since the Extended Splitter works on the global clock, τ are written at every write phase of the global clock except when a value is read from the compuational process. The Extended Equalizer will give the main process only one value during a single read phase and similarly, the Extended splitter will read only once during a single write phase of the computational process. The formal description of the Extended Equalizer and the Extended Splitter are given below:

Definition 15 *An Extended Equalizer function* $\mathcal{E}_e : (S^n \times S_{RW}) \longrightarrow (S^n \times {S_T}^n)$ *is defined as:*

$\mathcal{E}_e(s_1,\cdots,s_n,s_{rw}) = f(s_1,\cdots,s_n,1,\cdots,1,s_{rw},1,ne)$ *where,*

$f(s_1,\cdots,s_n,i_1,\cdots,i_n,s_{rw},k,m) =$

if $(\exists_{j=1}^{n}\ (s_j[i_j]) = \tau) \wedge (s_{rw}[k] = R)$ *then*

$<\tau,\ldots,\tau> \bigodot Vect_{j=1}^{n}(exp_1(j))$

$\bigoplus \mathcal{E}_e(s_1,\cdots,s_n,Vect_{j=1}^{n}(exp_2(j)),s_{rw},k+1,\ ne)$

else if $(\forall_{j=1}^{n}\ (s_j[i_j]) \neq \tau) \wedge (s_{rw}[k] = R) \wedge (m = ne)$ *then*

$Vect_{j=1}^{n}\ (s_j[i_j]) \bigodot < F,\ldots,F > \bigoplus f(s_1,\cdots,s_n,Vect_{j=1}^{n}(i_j+1),s_{rw},k+1,\ e)$

else if $(\forall_{j=1}^{n}\ (s_j[i_j]) \neq \tau) \wedge (s_{rw}[k] = R) \wedge (m = e)$ *then*

$<\tau,\ldots,\tau> \bigodot < F,\ldots,F > \bigoplus f(s_1,\cdots,s_n,Vect_{j=1}^{n}(i_j),s_{rw},k+1,\ e)$

else if $(\forall_{j=1}^{n}\ (s_j[i_j]) \neq \tau) \wedge (s_{rw}[k] = W)$ *then*

$f(s_1,\cdots,s_n,Vect_{j=1}^{n}(i_j),s_{rw},k+1,ne)$

The function f takes arguments $i_1,i_2,\cdots,i_n$ to index $s_1,s_2,\cdots,s_n$ and argument k to index signal s_{rw}. $m \in M$ where M = {e,ne} is a set of 2 elements, where 'e' indicates an events has been processed and 'ne' indicates that no event has been processed. The formulation of exp_1,exp_2 are given in Definition 7.

Definition 16 *Extended n-Splitter* $\mathcal{H}_e$ *is a process p:* $(S \times S_{RW}) \longrightarrow S^n$ *defined as*

$\mathcal{H}_e(s,s_{rw}) = h(x :: y,n,1,x_{rw} :: y_{rw},\ ne)$ *where,*

$h(x: : y,\ n,\ i,\ x_{rw} :: y_{rw},\ m) =$

if $((i = n) \wedge (x_{rw} = W) \wedge (m = ne))$ *then* $f(x,\ n,\ i,\ 1) \bigoplus h(y,\ n,\ 1,\ y_{rw},\ e)$

$$else\ if\ ((i < n) \wedge (x_{rw} = W) \wedge (m = ne))\ then\ f(x, n, i, 1) \bigoplus h(y, n, i{+}1, y_{rw}, e)$$

$$else\ if\ ((i = n \vee i < n) \wedge (x_{rw} = W) \wedge (m = e))\ then\ h(y, n, i, y_{rw}, e)$$

$$else\ if\ (x_{rw} = R)\ then\ h(y, n, i, y_{rw}, ne)$$

Here, $s = x :: y$ and $s_{rw} = x_{rw} :: y_{rw}$. Function h takes an arguments i to index the n output signals and m to keep track of whether an event was processed. The formulation of $g(n)$, $f(x,n,i,j)$ are given in Definition 8.

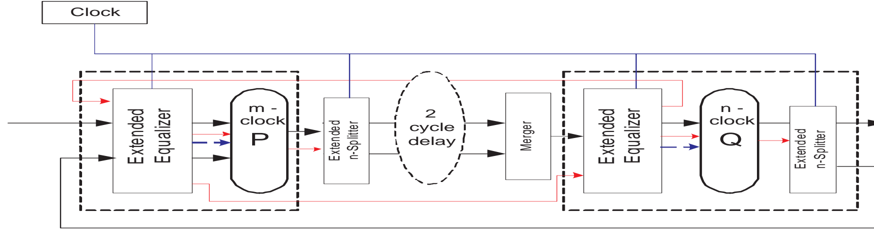


Figure 6. Multi-clock LI Implementation

Figure 6 shows the multi-clock LI implementation of the system. The main processes run based on the local clocks, whereas, the interfaces of the processes run based on the local clocks. The basic blocks of the module are same as shown in the previous implementations. Each process has an Extended Equalizer and a Extended n-Splitter. In the case of no long interconnects on the output, the Extended Splitter outputs the values on a single signal. In the read phase of each process, the data is read from Extended Equalizer. In the writing phase of the process, the data is written on the signals read by the Extended Splitter. On long interconnects, the Extended Splitter controls the writing of the values on its signals as described by Definition 16. Due to lack of space, we omit any formal proofs for latency equivalence in this case.

6 Verification of Protocols

To ensure that all members of this family of LIPs are functionally correct, we check for their latency equivalence with a corresponding synchronous system. By latency equivalence, we mean that all the values corresponding to the same set of informative events must be equal. We model the two systems in SPIN and feed the same sequence of tokens to the synchronous model as well as the LI model, and compare the output tokens to be equal. The setup for verification of the two systems is shown in Figure 7.

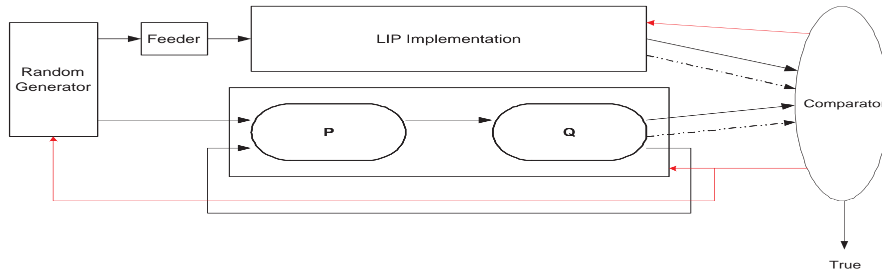


Figure 7. Latency Equivalence Checking of Synchronous and LI System

The figure illustrates a random generator giving the same sequence of values to both the systems. The corresponding processes of both the system perform the same computation. The output of the systems is fed to the Comparator. The Comparator is a process that embeds an equalizer that takes the inputs from the signals of the two implementations and has an assertion that checks that the values read are equal at every clock tick. This assertion ensures that the values received at the output are equal. Verification to check for the latency equivalence was done for LIP with relay station and synchronous model, LIP without relay station and components with same clock, and finally LIP without relay station that permit the use of components with different clocks but with a known clock ratio. For verification of the latency equivalence for this LIP, various different ratios for the clocks were taken into account.

7 Conclusion and Future Work

We propose refinements over existing LIP implementations. We simplify existing LIPs by eliminating the need for relay stations on multi clock cycle interconnections. With our protocol designers can reuse IP components with different clocks, therefore making it possible to mix IPs of different speeds, thus extending the repertoire of easily accessible components without the need of making costly adaptations. While a LIP with different clock ratios still does not represent a real GALS system, it might be, however, a very efficient refinement. We are working on relaxing the assumption of known ratios of the clocks.

For cases where clock ratios are variable or not known a priori the currently presented options will, however, not work. In order to accurately represent these systems we need to handle real GALS systems, with asynchronous communications. We are aware of this limitation and are currently working on an extension of this work to deal with these issues. In order to validate the presented protocols we build examples as formal models and verify them with the SPIN model checker. We illustrate the theory for the protocols with formal definitions and proof sketches.

References

[1] Carloni, L., McMillan, K., Saldanha, A., Sangiovanni-Vincentelli, A.: A methodology for correct-by-construction latency insensitive design. In: In Proc. International Conf. Computer Aided Verification. (1999) 309–315

[2] Carloni, L., McMillan, K., Sangiovanni-Vincentelli, A.: The theory of lantency insensitive design. IEEE Transactions on Computer Aided Design of Integrated Circuits and System **20** (2001) 1059–1076

[3] Carloni, L., McMillan, K.L., Sangiovanni-Vincentelli, A.L.: Latency insensitive protocols. In: 11th International Conference on Computer-Aided Verification. Volume 1633., Trento, Italy, Springer Verlag (1999) 123–133

[4] Singh, M., Theobald, M.: Generalized latency-insensitive systems for single-clock and multi-clock architectures. In: Design, Automation and Test in Europe (DATE'04). (2004)

[5] Casu, M., Macchiarulo, L.: A new approach to latency insensitive design. In: Design Automation Conference. (2004)

[6] Holzmann, G.: The SPIN Model Checker. Addison Wesley (2004)

[7] Jantsch, A.: Modeling Embedded Systems and SOCs - Concurrency and Time in Models of Computation. Morgan Kaufmann (2001)

[8] Rushby, J.: Formal verification of mcmillan's compositional assume-guarantee rule. Technical report, SRI International (2001)