Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures[*]

Montek Singh
Univ. of North Carolina
Chapel Hill, NC, USA
montek@cs.unc.edu

Michael Theobald
Carnegie Mellon University
Pittsburgh, PA, USA
theobald@cs.cmu.edu

Abstract

Latency-insensitive systems *were recently proposed by Carloni et al. as a correct-by-construction methodology for single-clock system-on-a-chip (SoC) design using predesigned IP blocks. Their approach overcomes the problem of long latencies of global interconnects in deep-submicron technologies, while still maintaining much of the inherent simplicity of synchronous design. In particular, wires whose latency is greater than a clock cycle are segmented using "relay stations," and IP blocks are made robust to arbitrary communication delays.*

This paper shows, however, that significant extensions are needed to make latency-insensitive systems useful for the practical design of large-scale SoC's. In particular, this paper proposes three extensions. The first extension allows each synchronous module to treat its input and output channels in a much more flexible manner, i.e., *with greater decoupling. The second extension generalizes inter-module communication from point-to-point channels to more complex networks of arbitrary topologies. Finally, the third extension is to target multi-clock SoC's. The net impact of our extensions is the potential for improved throughput, reduced power consumption, and greater flexibility in design.*

1. Introduction

Latency-insensitive systems [5, 6, 3] were originally proposed for the design of single-clock SoC's. A synchronous module is said to be latency-insensitive if it can operate correctly in the presence of arbitrary delays on its input and output channels. If any of the input or output channels is not available, the module is stalled via clock gating until all of its channels become available. As a result, latency-insensitive IP blocks are easily composable, offering ease of design reuse.

While Carloni et al.'s latency-insensitive approach has been successfully used in the design of certain single-clock SoC's, significant extensions are needed before it can be successfully used for practical design of more general single-clock systems, as well as for multi-clock systems. In particular, one limitation of their approach is the assumption that the data rates on all input and output channels of a synchronous module are identical. Thus, all input channels must be read, and all output channels must be written on every clock cycle when the module is not stalled. As a result, the unavailability of any input or output channel causes the synchronous module to stall, even when that channel is actually not needed for the next operation, thereby limiting system throughput. In addition, since a module is expected to always produce all outputs, some of the outputs may contain "garbage" or invalid data, which is unnecessarily transported. A second limitation of their approach is that it only considers point-to-point interconnects. As this paper shows, this limitation could also cause a loss of performance. Finally, their approach only considers single-clock systems, and if it were directly used to implement a multi-clock system, then the system's throughput would be limited by the throughput of its slowest synchronous module.

Contributions

This paper extends and generalizes the basic latency-insensitive approach as follows:

1. *More Flexible Synchronous Modules:*
   We provide a formal approach to specifying and synthesizing wrappers that read and write only those channels that are actually needed.
2. *Arbitrary Communication Network Topologies:*
   We provide a communication network that breaks away from the simple point-to-point paradigm. We provide both synchronous and asynchronous (*i.e., clockless*) implementations, which only transport useful (*i.e.*, valid) data.
3. *Handling Multiple Clocks:*
   We show that the new approach can be naturally adapted for multi-clock systems using well-known techniques for interfacing distinct clock domains.

The net impact of our extensions is the potential for improved throughput, reduced power consumption, and greater flexibility in design.

It is interesting to note that the communication strategy used in our approach lies in-between that of the basic latency-insensitive approach and the recently proposed network-on-a-chip (NoC) approach [1]. In particular, NoC's

[*] This research was supported by NSF under CCR-0121547 and CCR-0098072, by ARO under DAAD19-01-1-0485, by ONR/NRL under N00014-01-1-0796, by SRC under 99-TJ-684, and by a UNC Faculty Development Award and a UNC University Research Council Award. The views in this document are those of the authors and do not necessarily represent the official policies of the funding sources.

provide a higher level of communication abstraction, where data is organized as packets, and sophisticated protocols handle error recovery and flow control. The basic latency-insensitive approach, however, is at a lower level than ours, with the entire communication medium operating at a single clock rate, and employing only point-to-point channels. Our approach, in contrast, provides more general communication topologies and can handle variable clock rates, but does not have the overhead of packet-based communication.

This paper is organized as follows. First, Section 2 gives background on the basic latency-insensitive approach. Then, Section 3 presents our new approach to designing more flexible latency-insensitive modules. Section 4 presents our extension of the communication network to arbitrary topologies, for both single-clock as well as multi-clock systems. Finally, Section 5 gives conclusions and directions for future research.

2. Background: Latency-Insensitive Systems

Latency-insensitive systems were introduced by Carloni et al. [5, 6, 3]. The key idea of this approach is to use clock gating to stall a module whenever any of its communication channels is unavailable, thereby making the module's operation tolerant of arbitrary communication delays. Thus, the computation performed by synchronous modules is effectively decoupled from inter-module communication. This decoupling is achieved by encapsulating the synchronous modules inside specially-designed "wrapper" circuits. As a result of this encapsulation, the synchronous blocks become more modular, thereby facilitating design reuse.

Figure 1 shows how a module ("pearl") is encapsulated using a wrapper circuit ("shell"). On each clock tick, assuming all input and output channels are available, the synchronous module consumes a data item from each input channel, and generates a data item for each output channel. If any input channel is not ready with valid data, or if any output channel is not ready to accept data, the synchronous module is stalled by gating its clock.

Communication between the modules is achieved using point-to-point channels. Those wires whose latency is greater than the target clock cycle time are segmented into smaller parts using "relay stations," so that the communication latency does not limit the clock rate. Unlike traditional buffering repeaters, relay stations are storage elements which are clocked. Thus, the communication latency between any pair of modules is always an integral number of clock periods.

The complete design flow consists of four basic steps:

1. Specification of synchronous components.
2. Encapsulation.
3. Physical layout, placement and routing.
4. Relay station insertion.

2.1. Alternative Approach: Pausible Clocking

An alternative approach to achieving latency-insensitivity is to employ clock pausing (*i.e.*, stretching the inactive clock phase), instead of clock gating [12, 11].

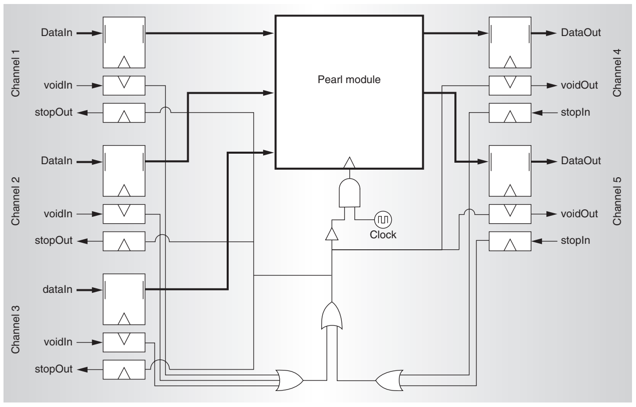


Figure 1. Carloni et al.'s approach to latency-insensitive design (from [4])

A significant disadvantage of the pausible clocking approach is the use of ring oscillators for generation of stoppable clocks, as opposed to crystal oscillators. Ring oscillators are prone to significant amounts of jitter—caused by stopping and restarting of the ring oscillator—as well as clock frequency variation due to temperature and voltage variations [7]. As a result, the performance of synchronous IP blocks may be severely degraded, since stable low-jitter clocks are key to modern high-performance synchronous design.

Due to the above disadvantages, we feel that pausible clocking currently only has a limited potential for use in high-performance SoC design. Therefore, our approach currently focuses on clock gating instead.

3. New Approach – Part I: More Flexible Synchronous Modules

This section presents a generalized encapsulation method to enable the design of more flexible synchronous modules. The new approach is motivated by an example in Section 3.1. An intuitive explanation is given in Section 3.2. Finally, a formal presentation is given in Section 3.3.

3.1. Motivation

Carloni et al.'s approach uses a simplifying assumption: every input channel, as well as every output channel, is exercised by a module on every clock tick. This assumption about a module's communication is overly restrictive: *e.g.*, a synchronous DSP core may actually need only a subset of its inputs and may generate data for only a subset of its output channels. Thus, their approach may cause a significant loss of throughput by generating more stalls than necessary. The following example illustrates this limitation.

Consider the program fragment of Figure 2, which describes a simple system. Figure 3 shows one possible hardware implementation of this specification, using a decomposition of the system into three distinct modules: *M1*, *M2* and *M3*. Module *M1* corresponds to the `while` loop and

```
while (true) do
  input k;
  for (i=0; i<10; i++) do
    input x;
    compute y=f(x,k);
    output y;
  end for
end while
```

Figure 2. Example 1

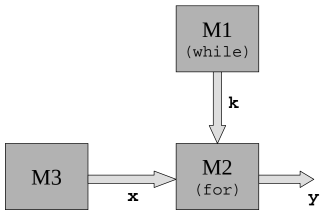


Figure 3. A possible implementation of the nested loops of Example 1.

provides the input k to *M2*. *M2* corresponds to the for loop, and *M3* provides the input x to *M2*.

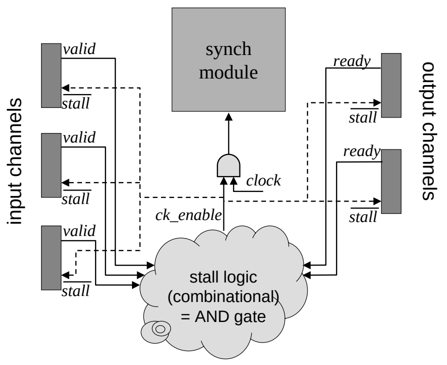
There is a key feature of this system that makes it difficult to derive an implementation using Carloni et al.'s approach. In particular, for every input item that *M2* receives from *M1*, ten data items are read by *M2* from *M3*. Thus, after a data item is received by *M2* from *M1*, no further communication is needed between those two modules for the next nine clock cycles. Carloni et al.'s latency-insensitive modules cannot be directly used here because they exercise every input and output channel on every clock tick.[1] Therefore, it becomes necessary to modify their approach in order to allow the handling of situations where only a subset of input channels must be read. Likewise, writing to only a subset of output channels must be handled.

3.2. Generalized Latency-Insensitive Modules

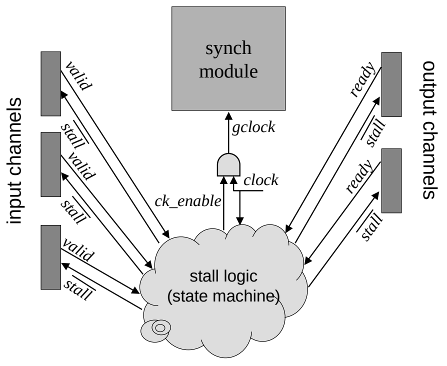
We now propose a modification of the wrappers that make synchronous modules latency-insensitive, in order to handle more general input–output behavior. This modification directly addresses the issues raised in Example 1 (Figure 3) above, *i.e.,* avoids unnecessary stalling when unavailable inputs are not actually needed.

Our modification applies to the stall generation circuitry inside the wrapper circuits. The inputs to the stall generation circuitry are the *valid* and *ready* signals from input and

1 Carloni et al.'s approach can be made to work in this scenario provided *M2* sends nine "garbage" data values to *M1*. However, this approach may introduce additional critical paths into the system, thereby potentially causing loss of performance. Moreover, transmitting unnecessary data values is wasteful of power.



(a)



(b)

Figure 4. Generalization of the stall logic from (a) simple combinational gate, to (b) FSM.

output channels, respectively. The outputs of the stall circuitry are: (i) the *ck_enable* signal, used to gate the module's clock input, and (ii) $\overline{stall}$ signals for the channels.[2]

Figure 4 shows the wrapper circuits before and after our modifications. In particular, Figure 4(a) shows a simplified view of Carloni et al.'s wrapper circuit of Figure 1. The stall logic consists of a single AND gate, which generates the *ck_enable* signal, which in turn is used to gate the clock of the synchronous module. The *ck_enable* signal is de-asserted if any of the input or output channels is unavailable. Figure 4(b) shows the stall circuitry after our modification. In particular, the combinational logic that generates stalls is now replaced by a more sophisticated finite-state machine (FSM). In addition, instead of only one stall signal, several decoupled signals are generated: a *ck_enable* for the synchronous module, and a $\overline{stall}$ for each channel.

The generalization of stall logic to a state machine allows us to treat input and output channels in a more flexi-

2 Note that our *valid, ready, ck_enable* and $\overline{stall}$ signals are actually negations of the signals of Carloni et al. (see Figure 1).

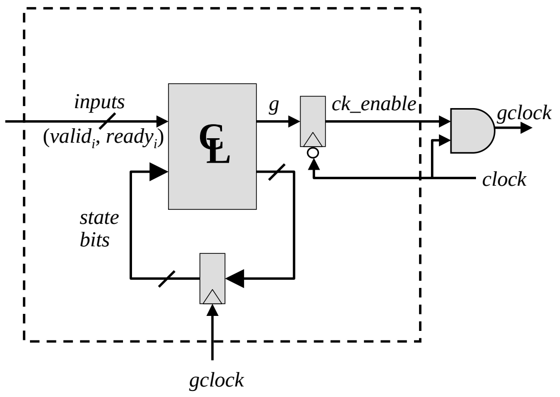


Figure 5. Wrapper FSM Implementation.

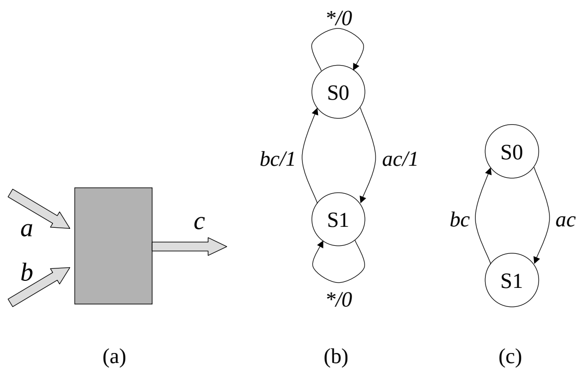


Figure 6. Wrapper FSM Example.

ble manner. For example, for a given state, only a particular subset of the input channels may need to be read, and only a subset of the output channels may need to be written. The remaining channels, which are not read or written, are ignored for the next clock tick; unavailability of these channels does not generate stalls.

The generalization presented here has two key benefits. First, a significant reduction in unnecessary stalls may be obtained, since stalls are no longer caused by the unavailability of those channels that are not currently needed. Second, modules that are not currently producing needed outputs can be safely stalled, without fear of stalling their neighbors. As a result, significant savings in power consumption may be obtained.

3.3. Wrapper Specification and Synthesis

This section presents in detail the implementation of the FSM for the wrapper circuit stall logic.

The stall logic is implemented as a synchronous Mealy machine, as shown in Figure 5. It consists of three components: (i) a block of combinational logic, which produces the output signals and the next state values, (ii) a register for latching the state bits, and (iii) a register for latching the outputs. The output signals generated are actually the negations of the stall signals, *i.e.,* they indicate when the module's clock should be enabled, and which of its channels should be enabled. To simplify the discussion, we only consider the enable signal for the synchronous module's clock; the enable signals for the channels are similarly generated. In the figure, *clock* is the original ungated clock signal, and *gclock* is its gated version which controls the operation of the associated module.

The key idea of the FSM's operation is as follows. In a given state, if the inputs represent availability of all required channels, then the combinational logic sets the signal *g* to "1," which implies that at the next tick of the *clock* signal, *gclock* is asserted so that the module can compute. On the other hand, if any required channel is unavailable, *g* is set to "0," which suppresses the generation of *gclock*, effectively putting the module to sleep.

There are two interesting features of Figure 5 that need further explanation. First, the machine's output *g* is latched by a register on the negative clock edge before being used to gate the module's clock. This latching action is needed to ensure that the gated clock, *gclock*, will be free of glitches; latching is performed on the negative clock edge so that the gating signal itself is available in time for the next rising clock edge. The second feature is that the register that stores the state bits is controlled by *gclock*, not by the original clock (*clock*). This has the effect of stalling the state machine whenever the associated module is stalled. As a result, "busy waiting" is eliminated from the state machine, thereby simplifying its specification and implementation.

We now present an example to illustrate the specification and synthesis of the wrapper state machine. Consider the synchronous module of Figure 6(a), with two input channels, *a* and *b*, and one output channel, *c*. Suppose that the module reads data from only one of the input channels, and passes it out onto the output channel. Further suppose that the module alternates between the input channels, starting with channel *a*. Then, Figure 6(b) represents an FSM specification for the stall logic, where *S0* and *S1* represent the two states of the stall logic, and *ac* and *bc* represent the conditions corresponding to the availability of channels *a* and *c*, and that of channels *b* and *c*, respectively. The arcs labeled *xx/1* represent those conditions when the module's clock is enabled. Similarly, the arcs labeled **/0* represent the remaining conditions, *i.e.,* when the module is stalled.

The use of the gated clock (*i.e., gclock*) to latch the state bits, instead of using the original clock allows us to simplify the FSM specification and implementation. In particular, since the clock is gated for the arcs labeled **/0* in Figure 6(b), those arcs simply represent a stalling of the FSM itself; hence the **/0* arcs can be eliminated. Further, since all of the remaining arcs have the FSM output value of "1," the FSM can be simply represented by Figure 6(c).

The FSM can be implemented using one state variable, *y.* Let state *S0* be encoded as *y=0,* and *S1* as *y=1.* If *g* represents the FSM output, and *Y* is the next-state value, then the following logic equations implement the FSM:

$$\begin{aligned} g &= \overline{y}ac + ybc \\ Y &= \overline{y}ac \end{aligned}$$

It is interesting to note a similarity between the proposed generalization and a particular style of asynchronous control synthesis: *burst-mode* synthesis [10]. In a burst-mode controller, each state is associated with one or more *input bursts,* where each input burst represents events on a subset of the controller inputs. When an input burst is received, the controller changes state, and produces an *output burst, i.e.,* events on a subset of the controller outputs. Signals that do not belong to a burst do not change their values when that burst occurs.

Using this terminology, the subset of channels that are read by a synchronous module constitute an input burst, and the subset of channels written constitute an output burst. At any given time, the availability of any one of the expected input bursts allows the synchronous module to operate on the next clock tick, and produce an output burst. If no input burst is available, the module is stalled. Note that, if determinism is required, more than one input burst should never be available simultaneously; as a corollary, no input burst should be a proper subset of another input burst.

It must be borne in mind that the similarity between the wrapper FSM notation and the burst-mode notation is interesting only for the purpose of modeling. The FSM is actually synchronous, so its implementation is much easier than that of asynchronous controllers.

It is crucial to formally verify the system for certain properties, *e.g.,* the absence of deadlocks. In particular, the specifications of the wrappers and that of the communication network together yield a system specification, which can be model-checked.

4. New Approach – Part II: Arbitrary Communication Network Topologies

The basic approach to latency-insensitive design assumes that all channels in the system are point-to-point channels. The aim of this section is to augment the basic approach with arbitrary communication network topologies. First, an example is presented in Section 4.1 to show the limitations of existing approaches, and provide motivation for the new approach. Section 4.2 then presents our proposed approach.

The approach of this section can be applied both to single-clock as well as multi-clock systems. The specification styles for both cases are the same, and only the underlying low-level implementations are somewhat different.

4.1. Motivation

Let us consider an example from the multi-clock domain.

Example 2. Consider the architecture of a three-stage stream processor, shown in Figure 7. Assume that fast modules (*i.e.,* IP blocks) are available for the first and third stages, but only half-rate modules are available for the second processing stage. In order to achieve full throughput, two modules must be used for the second stage, necessitating a split-merge type of interconnection. In this case, the split distributes one high-speed stream of data into two half-rate ones, and the merge combines them back into a single stream. If only simple point-to-point communication channels were allowed, the designer may be forced to use a single module for the second processing stage, resulting in a 50% loss of throughput.

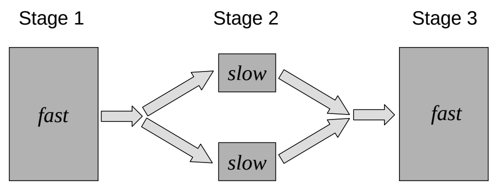


Figure 7. Illustrating the need for more sophisticated communication networks, *e.g.,* those involving splits and merges.

In general, we can state the following for any system composed of Carloni et al.-style synchronous modules connected by point-to-point communication channels:

> *System throughput will be limited by the rate of the slowest synchronous module.*

The validity of the claim is obvious if one notes that the synchronous modules of Carloni et al. [6] are required to stall if even a single input channel is not ready with valid new data, or if even a single output channel is not ready to receive new data. Note, however, that the actual throughput obtained may even be less than the rate of the slowest module because, in addition, the slowest module may be stalled at times.

Due to the limitation of point-to-point communication as pointed out above, a more general approach is needed for the design of the communication network.

4.2. Generalized Communication Network

We propose a generalization of the point-to-point communication channels to networks with arbitrary topologies. Due to space limitations, we only provide an outline of the approach here.

Our approach consists of using a number of specialized blocks to implement the communication network. Examples of such specialized blocks include: (i) *forks,* which replicate one input data stream onto multiple output channels; (ii) *splits,* which distribute data from one input channel onto multiple output channels; and (iii) *merges,* which combine (*i.e.,* interleave) multiple input data streams onto one output channel. These specialized blocks can be arbitrarily connected to form a rich set of communication topologies.

The communication network is implemented using synchronous or asynchronous circuit techniques. In particular, if the SoC has multiple clocks, then the communication network is implemented as an asynchronous subsystem to "glue" together the different clock domains. On the other hand, if the SoC has only one clock, then the communication network could be built either asynchronously, or synchronously using that same clock. An asynchronous network implementation effectively results in a *globally asynchronous locally synchronous* (GALS) system architecture.

If the communication network is implemented synchronously, then the network is implemented by composing specialized synchronous modules (*e.g.,* forks, joins, etc.), each of which is specified and implemented using our approach to designing stallable synchronous modules (Section 3).

On the other hand, if an asynchronous implementation style is chosen, the specialized blocks are available as handshake circuits from several well-known asynchronous component libraries (*e.g.,* [2, 9]). In order to ensure robustness against metastability, the circuits immediately interfacing with the synchronous IP blocks are based on ideas from the mixed-timing approaches of Chelcea and Nowick [8], and Chakraborty and Greenstreet [7].

Instead of specifying the network using the specialized blocks, higher level specifications can be used, followed by direct translation into a network of specialized blocks. For example, two derivatives of the CSP language, Tangram [2] and Balsa [9], are especially suitable for specifying complex communication networks. In addition, they provide automated translation into an asynchronous circuit-level implementation.

Once the communication network is implemented, a final step is needed to handle those long wires whose latency is greater than the target cycle time of the system. In particular, such long interconnects are divided into multiple segments. If the implementation of the communication network is synchronous, "relay stations" are inserted between segments [5, 3]. If the network is implemented asynchronously, our approach inserts asynchronous FIFO cells.

More specifically, our approach consists of three steps:

1. *Specify the communication network topology,* either using the specialized blocks, or using a high-level CSP-like language such as Tangram or Balsa.

2. *Choose between a synchronous and an asynchronous implementation.* For single-clock systems, either could be used; for multi-clock systems, an asynchronous implementation is used. If synchronous, implement as stallable finite-state machines. If asynchronous, implement using predesigned handshake circuits available in Tangram and Balsa.

3. *Identify wires with long latencies.* Segment them, and insert relay stations (synchronous) or FIFO handshake cells (asynchronous).

The net impact of the proposed generalization of the communication network is two-fold. First, a significantly greater degree of expressivity is offered for the specification of inter-module communication. Second, the designer is offered much greater freedom to "mix-'n-match" modules of different speeds and different types of interfaces. For example, the designer is able to use multiple instances of a slower module to interface with a faster module, using split-merge structures. As a result, better overall hardware utilization is achieved, thereby obtaining higher system throughput.

Finally, we would like to mention that some alternatives to point-to-point channels have already been reported. These approaches are based on simple topologies such as shared buses and rings [11], or simple forks and joins [13]. While these are steps in the right direction, they are nevertheless point solutions. Our proposed approach represents an important first step towards providing a formal framework for the specification and synthesis of arbitrary communication networks.

5. Conclusions and Future Work

This paper presented three generalizations to extend the notion of latency-insensitive systems. The first extension allows much greater flexibility in interfacing a synchronous module with its I/O channels, thereby allowing higher system throughput through elimination of unnecessary stalls. The second extension proposes more general communication network topologies than the currently popular point-to-point interconnects. The third extension allows the handling of multiple clock domains.

The suggested extensions have been introduced using motivating examples which demonstrate their benefit. In future work, we intend to apply our methodology to large-scale SoC designs.

References

[1] L. Benini and G. DeMicheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[2] K. v. Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[3] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. The theory of latency insensitive design. *IEEE Transactions on Computer-Aided Design*, 20(9), September 2001.

[4] L. Carloni and A. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5), Sep/Oct 2002.

[5] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 309–315, Nov. 1999.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In *Computer Aided Verification*, pages 123–133, 1999.

[7] A. Chakraborty and M. R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. In *Proc. International Symposium on Asynchronous Circuits and Systems*, May 2003.

[8] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proc. ACM/IEEE Design Automation Conference*, June 2001.

[9] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[10] S. M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, Oct. 1994.

[11] T. Villiger, H. Käslin, F. K. Gürkaynak, S. Oetiker, and W. Fichtner. Self-timed ring for globally-asynchronous locally-synchronous systems. In *Proc. International Symposium on Asynchronous Circuits and Systems*, May 2003.

[12] K. Y. Yun and R. P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proc. International Conf. Computer Design (ICCD)*, Oct. 1996.

[13] S. Zhuang, W. Li, J. Carlsson, K. Palmkvist, and L. Wanhammar. Asynchronous data communication with low power for gals systems. In *IEEE International Conference on Electronics, Circuits and Systems*, 2002.