Design, Implementation, and Validation of a New Class of Interface Circuits for Latency-Insensitive Design

Cheng-Hong Li, Rebecca Collins, Sampada Sonalkar, and Luca P. Carloni
Department of Computer Science - Columbia University in the City of New York

***Abstract*—With the arrival of nanometer technologies wire delays are no longer negligible with respect to gate delays, and timing-closure becomes a major challenge to System-on-Chip designers. Latency-insensitive design (LID) has been proposed as a "correct-by-construction" design methodology to cope with this problem. In this paper we present the design and implementation of a new and more efficient class of interface circuits to support LID. Our design offers substantial improvements in terms of logic delay over the design originally proposed by Carloni et al. [1] as well as in terms of both logic delay and processing throughput over the synchronous elastic architecture (SELF) recently proposed by Cortadella et al. [2]. These claims are supported by the experimental results that we obtained completing semi-custom implementations of the three designs with a $90nm$ industrial standard-cell library. We also report on the formal verification of our design: using the NuSMV model checker we verified that the RTL synthesizable implementations of our LID interface circuits (relay stations and shells) are correct refinements of the corresponding abstract specifications according to the theory of LID [3].**

I. Introduction

One of the most critical issues in designing Systems-on-Chip (SOC) with nanometer technology processes is the increasing impact of global wire delays: as more and smaller processing cores are accommodated on a chip, global (inter-core) wires do not scale in delay as local (intra-core) wires do because they need to span physical distances that represent significant proportions of the die [4], [5]. As the delays of global wires are no longer negligible compared to gate delays, the chip becomes a distributed system, thereby posing a serious challenge to the traditional CAD flows that are based on the synchronous design paradigm [6]. Furthermore, since wire delays are hard to predict at early stages of the design process, an increasing number of design exceptions in terms of post-layout timing violations forces costly design re-iterations (*timing-closure problem*).

Latency-insensitive design (LID) [1], [3], has been proposed as a "correct-by-construction" design methodology to handle the increasing impact of global communication latency in nanometer integrated circuit design without forcing major departures from traditional and well-established design flows. Given a synchronous system specification, e.g. a register-transfer level (RTL) netlist of logic blocks specified and validated using a hardware-description language, a functionally-equivalent latency-insensitive system can be automatically derived by encapsulating each sequential logic block (referred as a *pearl* or *core*) within an automatically generated interface process (a *shell*). The advantage of this transformation is that any communication channel connecting two core/shell pairs can now present a varying latency in terms of number of clock cycles without affecting the functional correctness of the original design. In practice the latency of a channel is changed through the insertion of *relay stations*, that are clocked buffers with capacity of at least two and simple flow-control logic. Hence, LID provides a sound way to address the problem of interconnect delay in nanometer design by simplifying the application of wire pipelining for global communication channels at any stage of the design process and without requiring any re-design of the cores. Furthermore, it simplifies the assembly and reuse of pre-designed cores for building complex SOCs because these can be arbitrarily complex sequential logic blocks as long as they are *stallable*: this is the only prerequisite for LID and it can be easily implemented with *clock gating* mechanisms [1], [3].

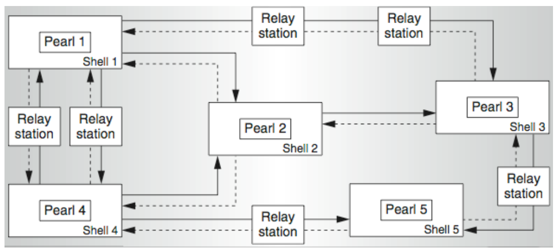


Fig. 1. Shell encapsulation, relay station insertion, and channel back-pressure.

In practice, the LID methodology calls for three steps: (1) a strictly synchronous (or *strict*) system is originally designed and validated as a netlist of stallable cores; (2) a *patient* system is automatically derived from the strict system by encapsulating each core within a shell; (3) any number of relay stations can be inserted on any channel between any pair of shells. Fig. 1 (taken from [6]) shows a latency-insensitive system with five core-pearl pairs connected by point-to-point, unidirectional channels. The shell logic and relay stations together implement a *latency-insensitive protocol* that is designed to accommodate arbitrary variations of wire delays while guaranteeing that the functional behavior of the original strict system is preserved (*semantics preservation*).

A formal definition of the properties of relay stations and shells is given in a denotational framework as part of the theory of LID [3]. At the core of LID lies the notion of latency-equivalence: two signals are latency equivalent if they present the same ordered streams of data items but possibly with different timing. In a synchronous model of computation the existence of a clock guarantees a common time reference among signals and, therefore, a signal must presents an event at each clock cycle [7], [8]. LID distinguishes between the occurrence of an informative event (a valid data item or *valid* token) and a stalling event (*void* token). Any class of latency-equivalent signals contains a single reference signal that does not present stalling events (a strict signal) while all the other

		1	2	3	4	5
LID-2ss	data	A	B	C	C	…
	void	0	0	0	0	…
	stop	0	1	1	1	…
receiver stalled		0	1	1	1	…
sender stalled		0	0	0	1	…
LID-1ss	data	A	B	B	…	…
	void	0	0	0	…	…
	stop	0	1	0	…	…
receiver stalled		0	1	1	…	…
sender stalled		0	0	1	…	…

Fig. 2. Simulations of the two latency-insensitive protocols with different back-pressure mechanisms.

members of the equivalence class (stalling signals) contain the same sequence of informative events interleaved by one or more stalling events. Following the tagged-signal model [8], the notions of latency-equivalence signals, strict signals, and stalling signals are extended to sets of signals (behaviors) and sets of behaviors (processes) [3].

In a nutshell, LID allows to derive from the original reference strict system specification, which contains only strict processes, any possible latency-equivalent implementation, which contains only patient processes. Each strict process abstracts the core in the original specification while the corresponding latency-equivalent patient process is obtained by composing the core with a shell. While the original cores are not designed to process void tokens, a shell-core pair is a patient process, i.e. it can tolerate the arrival of a void token at any of its I/O channel ports at any given clock cycle and be able to eventually continue with its correct operations.

In a practical implementation, void tokens are used to capture latency variations on communication channels and are processed by the shells in a way that makes them transparent to the cores. In particular, relay stations, which are not present in the original strict design, are initialized with void tokens when introduced in the patient design to pipeline a given channel. Void tokens are then processed by the shell while remaining transparent to the cores. Informally, any shell acts according to an *AND-firing policy*, thereby it stalls its core whenever at least a valid token is missing on one of its input channels. As a shell stalls its core, potential valid tokens that may be present on other input channels are stored locally in input queues within the shell for future processing by the core. In this way each shell dynamically absorbs the latency variations across the channels by realigning the valid tokens before presenting them to the core. Whenever it is not stalled, the core processes valid tokens on its inputs according to the functional behavior of the original strict system.

Since in practice a queue can only have a finite size, a downlink shell must be able to inform an uplink shell that is necessary to postpone the production of valid token for some cycles (*backpressure*). In the denotational framework of theory of LID, a backpressure event at a given clock cycle is also abstracted as the occurrence of a void token on the channel between the two shells [3]. While the theory of LID defines the general properties that any latency-insensitive protocol must obey, many possible implementations are conceivable in practice. A protocol implementation that relies on just two control bits, a $void$ bit to identify invalid data and $stop$ bit to implement backpressure, was first presented in [1] and discussed in more detail together with the supporting interface circuits in [6], [9].

Contribution. The latency-insensitive protocol that is discussed in [1], [6], [9] stipulates that a shell or relay station is stalled whenever the $stop$ bit is kept high for *two consecutive* clock cycles. In this paper we refer to this protocol as LID-2ss, which stands for *two-stop-to-stall*. The top of Figure 2 reports a simulation trace of a channel according to LID-2ss where the receiver is being stalled at cycle 2. Because the receiver is stalled, valid token A is not processed and thus is buffered by the receiver's shell. To avoid buffer overflow and possible loss of the data, the receiver stalls the sender by asserts the $stop$ bit both at cycle 2 and 3. Notice that the sender only stalls at cycle 4 holding on its output port the valid token C after receiving two $stop$ signals. This means token B needs to be buffered by a queue in the receiving shell together with token A. In fact, both the shell queues and the relay stations have storage capacity equal to two according to the library of interface circuits that were proposed to support LID-2ss.

In this paper we describe a simpler latency-insensitive protocol labeled as LID-1ss, which stands for *one-stop-to-stall*, that is based on a different back-pressure convention. In the new protocol, a shell or a relay station stalls whenever it receives a single $stop$ signal, as reported by the simulation trace in the bottom part of Figure 2: here, the receiver asserts the stop bit only at cycle 2, and the sender begins to stall immediately at cycle 3. In our design a queue of capacity equal to one in the receiver's shell is sufficient since only data token A must be buffered there during stalling while B is preserved uplink in the channel for future processing. Notice that our new protocol LID-1ss does not allow us to reduce the storage capacity of a relay station to one because this would reduce by half the performance of a latency-insensitive system as explained in the theory of LID [3]. On the other hand, it does allow us to reduce the storage capacity of a shell input queue to one with respect to the original protocol LID-2ss because we can take advantage of the storage capacity within the core [1].

We contribute a new set of interface circuits (i.e. shells and relay stations) that support the LID-1ss protocol and offer substantial improvements with respect to previous works in the literature. In particular,

- they are better in terms of area overhead and logic delay than the circuits supporting the original latency-insensitive protocol LID-2ss as discussed in [1], [6], [9];
- they are better in terms of both logic delay and processing throughput than the synchronous elastic architecture (SELF) that were recently proposed in [2].

We also report on our work to validate both our design and the original design: using the NuSMV model checker we formally verified that the RTL synthesizable implementations of the key LID building blocks (relay stations and shells) is a correct refinement of the corresponding abstract specifications

[1]To discuss how the performance of a latency-insensitive system can be optimized through relay-station insertion and the sizing of shell input queues goes beyond the scope of this paper and we refer to [10].

according to the theory of LID [3].

The paper is organized as follows. In Sec. II we briefly overview the related work on latency-insensitive design. The RTL logic of the interface circuits supporting our LID-1ss protocol is described in detail in Sec. III. We then discuss the formal verification of these circuits in Sec. IV. Finally, in Sec. V. we present a comprehensive set of experimental results that provide a comparative analysis of LID-1ss, LID-2ss, and SELF in terms of logic delay, effect on system's processing throughput, and area overhead.

II. Related Work

The LID methodology has gained certain interests in recent years, and several extensions and related approaches have been proposed [2], [11]–[15]. Indeed, while it specifies the fundamental properties of any latency-insensitive protocol, the denotational framework used to develop the theory of LID [3] leaves open the possibility of developing various protocol specifications with different implementation characteristics.

The simpler protocol specification that we discuss in this paper was already assumed in [13], [14]. The authors of [13] presented a mixed-timing relay station that stalls for one clock cycle if a stop signal is received. As they focus on describing several low-latency mixed-timing FIFO interfaces, they do not discuss the design of shell blocks to support LID. The authors of [14] use max-plus algebra to analyze the performance of a latency-insensitive system with back-pressure. The model of the protocol that they adopt assumes that a sender is stalled when one or more of its receivers asserts the stop bit. However, neither the design of the shell nor the design of a relay station is provided. Conversely, in this paper we contribute the complete interface logic for a single-clock synchronous system at the RTL level.

Cortadella et al. recently proposed synchronous elastic architectures based on synchronous elastic flow (SELF) as a new approach to LID that "combines the modularity of asynchronous design with the efficiency of synchronous implementations" [2]. The SELF protocol is another example of a latency-insensitive protocol implementation that relies on $valid$ and $stop$ bit like the LID-2ss protocol that was originally proposed by Carloni et al. [1] and the LID-1ss one that we discuss in the present paper. However, despite the protocol resemblance, SELF does not use input queues in shells to store valid tokens during stalling. Instead, a valid but unused token is held by its immediate sender. Similarly to LID relay stations, SELF uses sequential buffers, called *elastic buffers (EB)*, to pipeline long channel wires. On the other hand, the notion of a shell interface is simply not present in SELF. Instead, in SELF it is possible to have elastic buffers with multiple input/output channels thanks to special elastic *fork* and *join* control structures [2]. Robustness with respect to latency variations is achieved in SELF by combining elastic buffers, fork and join structures while performing an *elasticization* transformation on the original circuit. This step consists essentially of replacing each flip-flop in the core with two transparent latches of different polarity, similar to a master-slave structure, but with the independent enable signals

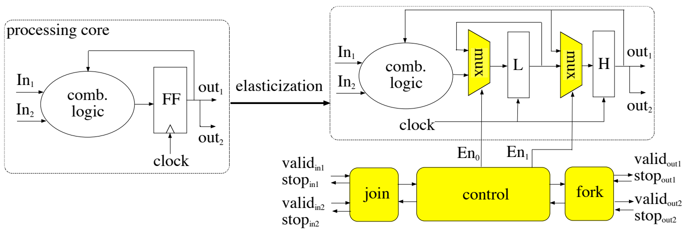


Fig. 3. Elasticizing a processing core according to the SELF scheme.

for the two latches so that "a mechanism for double-pumping in one cycle" can be realized. The elasticization of a processing core is illustrated in Fig. 3, where the shaded boxes represent the additional logic that must be added to support SELF. This logic design structure makes the SELF protocol differ subtly from the one that we propose with respect to the timing of sending a $stop$ bit to a sender. In our LID-1ss protocol this is sent whenever a queue is full. In SELF, the interface logic of a processing core with multiple input channels requests all valid tokens to be resent (by asserting the corresponding $stop$ bits) whenever at least one invalid tokens arrives at the *same* clock cycle. This has negative impacts on the performance of a system based on SELF because: (a) it degrades the overall system throughput and (b) it limits the maximum clock frequency at which the final circuit can run due to long combinational paths spanning two interconnect channels. In Section V we present a detailed discussion of these issues in the context of a comparative analysis of the three protocols and we show that our proposed solution does not present any of these drawbacks.

Suhaib *et al.* [16] propose a framework for validating families of latency-insensitive protocols where a new protocol is verified by taking a system, transforming it into a latency-insensitive system that obeys the new protocol, and then comparing the output behavior of the original and transformed systems on a subset of possible inputs. This technique is good for the development and debugging phase of new protocols because it can uncover many bugs quickly without requiring an exhaustive verification. As described in Sec. IV, our approach is more applicable to a later phase of the design of a latency-insensitive protocol implementation. In particular, we formally verify the RTL implementation of relay station and shell *modularly* in a modular fashion so that a previously verified synchronous system does not need to be reverified after it has been transformed into a latency-insensitive system. This approach has several advantages. New systems can be verified independently of the architecture they will operate on. In addition, formally verifying the shell is quite demanding in terms of computational memory. Verifying an entire system implementation with numerous cores, each encapsulated in its own shell would be prohibitively expensive at the same level of rigor.

III. A Simplified Latency-insensitive Protocol and Its Implementations

In this section we discuss in detail the implementation of the simplified latency-insensitive protocol LID-1ss that we

		1	2	3	4	5	6	7	8	9	10	11
	$dataIn_1$	A_1	A_1	A_2	A_3	A_4	A_5	A_6	A_6	A_6	A_8	A_9
In 1	$voidIn_1$	0	1	0	0	0	0	0	1	1	0	0
	$stopOut_1$	0	0	0	0	0	0	0	0	0	0	0
	$dataIn_2$	B_1	B_2	B_3	B_4	B_5	B_6	B_6	B_6	B_6	B_8	B_9
In 2	$voidIn_2$	0	0	0	0	0	0	0	1	1	0	0
	$stopOut_2$	0	0	0	0	0	1	0	0	0	0	0
	$dataOut_1$	C_1	C_2	C_2	C_3	C_4	C_4	C_5	C_6	C_7	C_7	C_8
Out 1	$voidOut_1$	0	0	1	0	0	1	0	0	0	1	0
	$stopIn_1$	0	0	0	0	0	0	0	0	0	1	0
	$dataOut_2$	D_1	D_2	D_2	D_3	D_4	D_4	D_5	D_6	D_7	D_7	D_8
Out 2	$voidOut_2$	0	0	1	0	0	0	0	0	0	1	0
	$stopIn_2$	0	0	0	0	1	0	0	0	0	0	0

Fig. 4. Sample I/O behavior of the new shell. Shaded data tokens are bubbles.

introduced in Section I. Briefly, the new protocol differs from the original LID-1ss protocol discussed in [1] in the back-pressure mechanism: the LID-1ss protocol uses a single $stop$ bit to stall a sender. For both the shell and the relay station, we first present sample simulations of their I/O behaviors and then explain the details of the RTL designs.

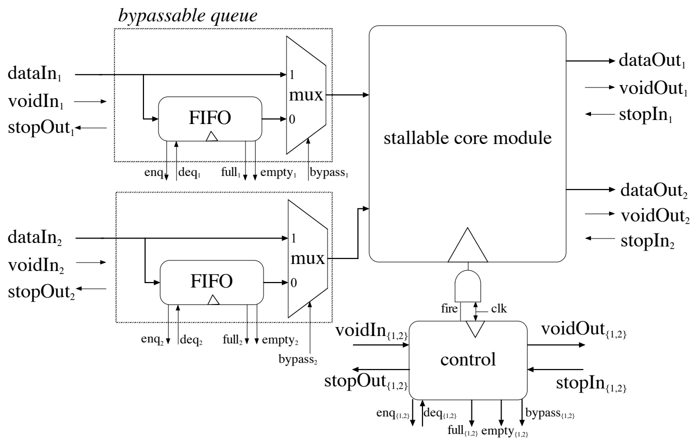
Shell. Fig. 4 shows a sample simulation trace of a two-input-two-output shell with the assumption that both input queues have a capacity of two. Several scenarios are illustrated in this trace. In cycle 1 both input channels (channel 1 and 2) present valid data tokens, and, therefore, the core can be fired to produce valid output tokens (C_1 and D_1) at cycle 2. At cycle 2 the void input token of channel 1 (void bit is high) causes the shell to stall the core at cycle 3. Therefore, both the output tokens at cycle 3 are marked as void with their $voidOut$ bits being asserted by the shell.

The scenario in which the shell receives back-pressure from its downlinks happens at cycle 5, when the downlink receiver of channel 4 asserts the $stopIn_D$ bit. Thus the output token D_4 is regarded as void at cycle 5(see Sec. I), the core is stalled at cycle 6, and both C_4 and D_4 are repeated at cycle 6. However, since the downlink receiver in channel 3 has already sampled C_4, the void bit is set for the repeated C_4 so the downlink of channel 3 will not sample the same token twice. The accompanying $void$ bit of D_4, on the other hand, is not set because the downlink of channel 4 has yet to sample D_4. In this case D_4 is sampled at the end of cycle 6 (when the clock edges arrives to start cycle 7).

What follows from cycle 6 shows the case when an input queue is full. The stop request from the downlink of channel 4 causes the input queue of channel 2 to be filled up at cycle 6 (two valid tokens are stored in channel 2's queue at the end of cycle 5, due to the stalls at cycle 3 and 6), thus a stop request is raised to the upstream sender in channel 2. Note that at cycle 6 the shell is not able to store token B_6. The same token is thus repeated by the uplink of channel 2 and is sampled by the shell at cycle 7.

Next we present the details of the shell RTL logic design. Fig. 5(a) reports a block diagram of a two-input-two-output shell, and the logic functions of the controller is listed in Fig. 5(b). The control logic is general and can be easily scaled to handle an arbitrary number of inputs and/or outputs. All the logic functions are quite simple and can be implemented with few logic gates.

The clock gating signal $fire$ decides whether the core



(a)

$$\begin{aligned}
\mathit{fire} &= \textstyle\bigwedge_{i\in\mathcal{I}}(\overline{voidIn_i} + \overline{empty_i}) \cdot \overline{\bigvee_{i\in\mathcal{I}}(stopIn_i \cdot \overline{voidOut_i})} \\
\forall j \in \mathcal{O}\ voidOut_j{}^+ &= \begin{cases} 0 & \text{if } stopIn_j \cdot \overline{voidOut_j} \text{ is true} \\ \overline{\mathit{fire}} & \text{otherwise} \end{cases} \\
\forall i \in \mathcal{I}\ stopOut_i &= full_i \\
\forall i \in \mathcal{I}\ enq_i &= \overline{voidIn_i} \cdot (\overline{\mathit{fire}} + \overline{empty_i}) \cdot \overline{full_i} \\
\forall i \in \mathcal{I}\ deq_i &= \overline{empty_i} \cdot \mathit{fire} \\
\forall i \in \mathcal{I}\ bypass_i &= empty_i
\end{aligned}$$

(b)

Fig. 5. (a) A block diagram of a two-input-two-output shell and a stallable core module. (b) Logic functions of the shell controller.

module is fired or stalled. It is asserted when each channel presents a valid token either directly from the channel input or from its input queue, and no stop request has arrived on any output channel. The second condition can be detected by checking the current $stopIn$ and $voidOut$ bits for each output channel. If the $voidOut_j$ bit is high for some channel j, the downlink receiver of channel j has received the latest valid token. In this case the core module can proceed even the receiver requests to stop.

The $voidOut$ bit informs to the downlink module on channel j whether the current token is a valid token or not. It is a sequential signal buffered by an edge-triggered flip-flop. The condition $stopIn_i \cdot \overline{voidOut_i} = true$ means that the downlink module on channel j is not able to process the current (also the latest) data token. In this case the core module will be stalled, the current token will be repeated, and $voidOut_j$ will be set low. In all other cases the value of the $voidOut$ bit depends on whether the core module will be fired.

The major data-path components in a shell are the by-passable queues storing unused valid tokens from input channels. Its minimum forward latency is zero. The by-passable queue is implemented as a standard FIFO whose output is multiplexed with the incoming data of the channel. If the queue is empty, the controller selects the data token from the input channel and passes it to the core module. The internal queue is a *sequential* element: all of the operations (i.e. enqueue and dequeue) and the update of its status (i.e. full or empty) take place at each clock edge. Hence all of the $stopOut$ signals, which are the full signals from the queue, are sequential signals.

Relay station. Fig. 6 reports sample I/O behaviors of a relay station. From cycle 1 to 4, the relay station simply relays the received data, void or not, from its input channel to its output

	1	2	3	4	5	6	7	8	9	10	11
$dataIn$	A_1	A_1	A_2	A_2	A_3	A_4	A_4	A_5	A_6	A_7	A_7
$voidIn$	0	1	0	1	0	0	1	0	0	0	0
$stopOut$	0	0	0	0	0	0	0	0	0	1	0
$dataOut$	$*$	A_1	A_1	A_2	A_2	A_3	A_4	A_4	A_5	A_5	A_6
$voidOut$	0	0	1	0	1	0	0	0	0	0	0
$stopIn$	0	0	0	0	1	0	1	0	1	0	0

Fig. 6. Sample I/O behavior of the new relay station.

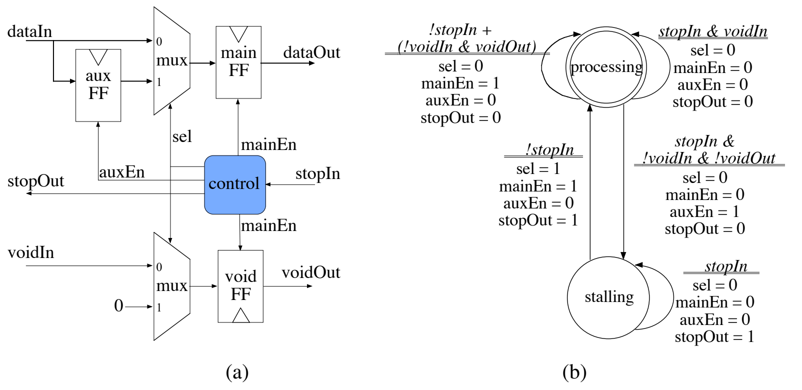


Fig. 7. (a) Block diagram of the new relay station; (b) The state transition diagram of its controller.

channel.

At cycle 9, the relay station receives a stop request from its downlink receiver. It then stalls (and repeats its output token) for one cycle to avoid overflow its downlink receiver. Meanwhile, the incoming data token at cycle 9 is buffered in the relay station's internal storage, and the stop request is sent to its uplink sender at next clock cycle.

Sometimes, an optimization can be applied to avoid stalling the relay station when the downlink receiver asserts the $stopIn$ bit. This is shown at cycle 5 to 6. At cycle 5 the relay station receives the stop request and sends out a void token at the same time. Because the void token will not be sampled by its downlink receiver, the relay station can safely continue to relay data tokens at cycle 6 without being stalled.

Another optimization occurs when the relay station absorbs a stop request instead of relaying it to its uplink sender. For instance, at cycle 7 the relay station receives a void token from its uplink and a stop request from its downlink. It can actually discard the void token received at cycle 7, instead of buffering it, and simply repeat its current output at cycle 8. In this way, it avoids propagating the stop request.

Fig. 7(a) shows an implementation of the relay station for the proposed latency-insensitive protocol; Fig. 7(b) reports the state transition diagram of its controller. The new relay station uses two edge-triggered flip-flops to store incoming data tokens, and one flip-flop to buffer the $voidOut$ bit. The two flip-flop storing data tokens provide the necessary twofold storage capacity. The output of the main flip-flop is the data output of the relay station. The controller decides when to update the three flip-flops and sets $stopOut$ and $voidOut$ bits according to the protocol. The control logic is discussed next.

The controller is a two-state Mealy finite state machine with three input and four output signals. The initial state is the *processing* state, which enables the main flip-flop and sets the $stopOut$ bit low. In the stalling state, instead, the relay station uses both the main and the auxiliary flip-flops to store data tokens, and request the uplink sender to stop sending more data tokens by asserting its $stopOut$ bit. Note that the value of the $stopOut$ bit depends only on the current state of the controller, and thus no combinational path exists between $stopIn$ and $stopOut$.

The switching from the *processing* state to the *stalling* state is triggered by the condition that the $stopIn$ bit is high, and both the $voidIn$ and $voidOut$ bits are low. The asserted $stopIn$ bit indicates that the receiver is not able to process the output data taken of the relay station. Hence the relay station has to maintain its output token by keeping the same data in the main flip-flop. On the other hand, the relay station must save the incoming valid token (indicated by low values of $voidIn$ and $stopOut$) to the auxiliary flip-flop, and enter the *stalling* state. Note that the incoming $voidIn$ bit is not saved in the void flip-flop, because in this case it is always low (this is part of the condition to switch from the *processing* to the *stalling* state) and thus can be easily recovered.

The relay station goes back from the *stalling* to the *processing* state when its downlink receiver deasserts the $stopIn$ bit, indicating that it is ready to receive more valid data tokens. Then, the relay station moves the token saved in the auxiliary flip-flop to the main flip-flop. It also updates the void flip-flop with a constant low value because the accompanying void bit of the data token in the auxiliary flip-flop must be deasserted.

IV. Formal Verification of the Protocol Implementation

An important compositional result is proven as part of the theory of latency-insensitive design [3]: if all modules in a strict system are replaced by corresponding latency-equivalent patient modules, then the resulting system is patient and latency equivalent to the original one. Naturally, this theoretical result is not enough to guarantee that a particular implementation of a latency-insensitive system is correct. The theory tells us that we can build a patient system out of patient parts, but we must also verify that the parts (the actual implementations of the shells and relay stations) are patient. On the other hand, we can verify the implementations of shells and relay stations in isolation because according to the compositionality rule for latency equivalence of patient processes, a system composed of shell-core pairs and relay stations is also latency equivalent to the original strict system.

We first translated by hand the synthesizable VERILOG code implementing the logic of the shell and relay station described in Section III into the NuSMV language [17]. Then we used the NuSMV model checker to verify that they are correct refinements of the specifications given in the LID theory.

In particular we verified the design for properties related to latency equivalence, liveness, and storage capacity. For a relay station this is sufficient to prove that it is a patient process. The shell is a little trickier. For the shell, patience also depends on the functionality of the core that the shell encapsulates and the shell implementation varies slightly depending on the number of input and output channels of its core.

Verification approach. Figure 8 and Figure 9 illustrate our verification approach for the relay station and the shell respec-

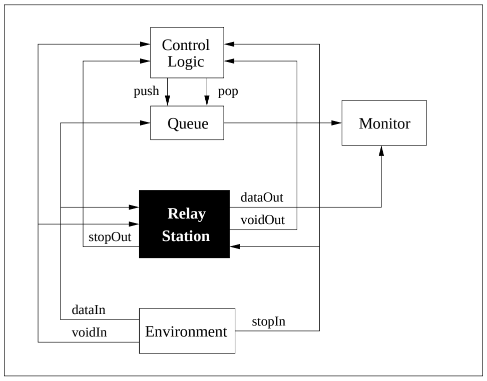


Fig. 8. Verification Framework for a Relay Station

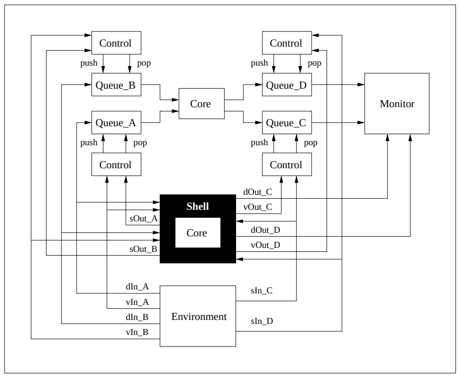


Fig. 9. Verification Framework for a Shell

Property	Module name	Time	Memory
Latency Equivalence	Relay station	0.2 sec	7.2 MB
	Shell	15.5 min	2.4 GB
Liveness	Relay station	5.5 sec	14.3 MB
	Shell	1.4 hours	2.4 GB

TABLE I
MEMORY AND TIME STATISTICS FOR THE VERIFICTION TASKS.

tively. The verification framework consists of the component-under-verification (CUV) together with the environment, queue, and monitor modules.

The environment generates data items, the valid bits, and the stop bits in an unconstrained manner: at each clock cycle, the environment may non-deterministically choose a value for *dataIn*, and non-deterministically set *voidIn* and *stopIn* to either *true* or *false* values. This enables verification under all possible input sequences; if any possible input sequence fails, a counterexample is generated. The monitor checks the correctness of the property to be verified by comparing the stream(s) of valid data produced by the CUV versus the stream(s) of data that passed through the queue. The correct functioning of a latency-insensitive component is checked under the assumption that its environment obeys the latency-insensitive protocol i.e. the environment holds a data token until it is sampled by the component. We do not impose this assumption on the environment and instead track the sampling of data tokens according to the latency-insensitive protocol.

The queue is a FIFO used to store the valid data tokens sampled by the monitor until they are matched with the output tokens. It has standard push and pop operations for adding new valid tokens to the tail of the queue and popping valid tokens off the head of the queue. A valid data token is pushed in the queue whenever the CUV latches in the token. Similarly a valid data token is popped off the queue whenever the CUV outputs a data token. These decisions are made by the queue control logic based on the values of the *stop* and *void* bits. The queue's pop signal is forwarded to the monitor, and when a pop occurs the monitor compares the queue's output to the CUV's output.

For the verification of the relay station a simple FIFO is sufficient because the relay station itself has simple store-and-forward behavior. For the verification of the shell, we also need a core module to perform computation on the given inputs and produce output data. We chose a 2-input, 2-output core that computes in parallel the two-input NAND and NOR logic operations and stores the results in two internal flip-flops. Separate queues are maintained for each incoming channel, and a second core module is instantiated outside the shell. When both input queues have valid data tokens, these are passed to the core and the results are stored in an output queue. The monitor compares the output of the shell with the data in the output queue.

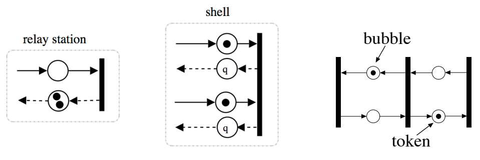
Formal Properties. We checked the properties of latency equivalence, liveness, and storage capacity. The latency equivalence property expresses that there is no loss, duplication or reordering of valid tokens in a data stream. To test latency equivalence of the relay station, we checked that the relay station's outgoing data stream is latency equivalent to its incoming data stream. To verify latency equivalence of the two-input two-output shell, we compared the data tokens produced by the core alone and those produced by the core/shell pair.

The liveness property expresses progress in the system. A component is live if it produces meaningful data provided the environment allows it. We imposed a fairness constraint on the environment for the *void* and *stop* bits so that the environment generates valid data items infinitely often and enables the downlink stream infinitely often. The liveness property states that the component generates valid data tokens infinitely often and enables the uplink stream infinitely often.

The storage capacity property checks that the number of data items in the monitor queue never exceeds the storage capacity of the component. The relay station capacity is equal to two. The storage capacity of the shell depends on the size of its internal queue, which is at least equal to one.

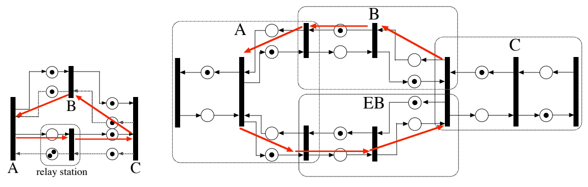
The above properties were verified individually for the shell and relay station Verilog implementations. All of the properties passed verification. The latency equivalence property was also tested on known erroneous implementations of both the shell and relay station. The verification failed and generated counterexamples as expected.

The verification was performed on a machine with 2 AMD Opteron $^{\text{TM}}$ processors and 3.5 GB memory over Redhat Linux with the Fedora Core 6, and NuSMV version 2.4.1. Time and memory usage from the verification experiments are summarized in Table I.



(a) (b)

Fig. 10. Marked graph models for (a) LID-2ss and LID-1ss; (b) SELF.



(a) LID-1ss (b) SELF

Fig. 11. Marked graph models of the example in Fig. 12(a).

V. Comparisons of Latency-insensitive Protocols and Implementations

In this section we present a comparative analysis of the proposed latency-insensitive protocol implementation of LID-1ss versus the implementation of both the original LID-2ss protocol and the SELF protocol in terms of system throughput, logic delay, and area overhead. In Section II we provided a brief overview of SELF [2] and we clarified that SELF does not use the concept of shell interfaces but relies instead on elastic fork and join structures. In the sequel, however, whenever it is convenient we will use the term "shell" to refer to the SELF interface logic for a processing core and, in particular, to the control of the substitute elastic buffer and the structures joining input and forking output channels.

System Throughput. To make a system robust with respect to communication latency through the application of either LID or SELF may generally have a negative impact on the its performance measured as processing throughput. This is defined as the ratio of the number of valid tokens over the number of valid tokens plus void tokens that the system processes over time. Since both a relay station (RS) and an elastic buffer (EB) are initialized with a void token and since void tokens may create more void tokens whenever they stall a computation, the placement of RS's or EBs on channels that belong to feedback loops and/or re-convergent paths may induce permanent degradation of the system throughput. The system throughput can be computed exactly by using either *marked graph* models [2], [18], or equivalently max-plus algebra [14]. Fig. 10 shows the marked graph models for LID (LID-2ss and LID-1ss [10]) and SELF [2]. Note that in the LID shell model in Fig. 10(a), the sizes of the shell queues are represented by a variable q (in a shell input channels can be statically sized differently for performance purposes [10]).

Both the LID and the SELF models are compositional as they inherit their topological structure from the modeled system. Fig. 11 reports the LID model and the SELF model for the system shown in Fig. 12(a). Note that in the LID model each transition takes a single time unit to fire while in the SELF model a transition takes half a time unit to fire because SELF is a latch-based design.

The throughput of a LI or SELF system is equal to the inverse of the *maximum cycle mean* [19] of its corresponding marked graph model.[2] The mean of a cycle is the ratio between the sum of each transition's firing time and the number of tokens on the cycle (an invariant number in a marked graph). For both models in Fig. 11 we highlighted the critical cycles, i.e. cycles having the highest cycle mean. The LID-1ss-based implementation has a maximum cycle mean of $4/3$, and thus has a throughput of $3/4 = 0.75$, assuming all input queues in a shell have a capacity of one [10], [14]. The throughput of the SELF version, on the other hand, is lower: $2/3 = 0.67$.

[2]The maximum cycle mean of a graph can be computed by a number of efficient algorithms [20], [21].

In this particular example, the ideal system throughput, equal to 1, can still be achieved for both implementations. For the LID-1ss version it is necessary either to insert an additional relay station between cores B and C (or A and B) or to raise to two the size of the input queue in the C shell for the channel $B \rightarrow C$. The second approach is called *optimal channel queue sizing* [10], [14]. Since SELF does not use queues, the only solution to improve the throughput is to insert an additional elastic buffer between cores B and C (or A and B).

For certain systems, however, a SELF-based implementation cannot achieve the same system throughput of an implementation based on either LID-1ss or LID-2ss due to their particular structures, typically characterized by a combination of reconvergent paths and/or feedback loops. For example, for the system shown Fig. 12(b) LID-1ss and LID-2ss can achieve higher system throughput than SELF. Note that the system has a similar reconvergent path from A to C as the example in Fig. 12(a), but it has two additional *cycles*: (A, B, E, A) and (B, C, D, B). In a LID-1ss implementation, to achieve the ideal throughput equal to 1 one must increase the input queue size of channel $B \rightarrow C$ in C's shell to 2. In this case, however, it is impossible for a SELF implementation to achieve such an ideal throughput. The best one can do is to insert an additional elastic buffer between B and C (or A and B), which brings the throughput up to $3/4$ (because the cycle with the inserted EB becomes the new critical cycle).

The basic reason why the SELF design cannot match the throughput of LID-2ss and LID-1ss is because it lacks shell input queues to temporarily store valid tokens. Instead, whenever an elastic core has two or more input channels, the arrival of an invalid token in any input channel causes re-transmissions of all other valid tokens arrived in the same clock cycle. As the above examples show, this re-transmission can be more frequent in a SELF system than in a LID system, where unprocessed good tokens can be stored in the shell queues as long as space permits.

Interface Logic Delay. The delay of LID and SELF's interface logic affects the overall system performance in two ways. First, the longest combinational logic path within an interface or across two communicating interfaces might become the new critical path of the system, and thus determine the maximum clock frequency at which the system can run. Second, when pipelining a wire using RS/EBs, interface with shorter cross-

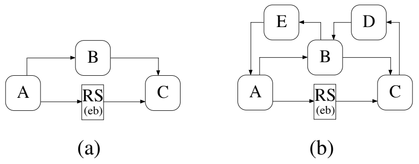
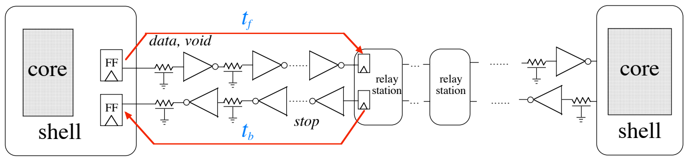
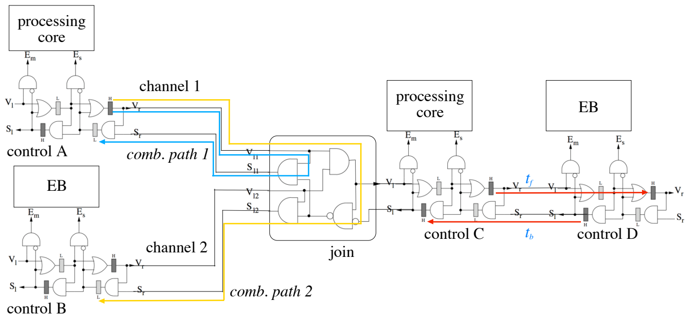


Fig. 12. Examples of systems with unbalanced reconvergent paths.



(a) Long wires are optimally buffered by repeaters.



(b) SELF slack computation and the combinational paths introduced by the join structure.

Fig. 13. Wire buffering and slack computations.

interface logic delays can be stretched farther away, and thus less RS/EBs are needed to span the same distance. Because each inserted RS or EB introduces an additional void token into the system and will potentially reduce system throughput, it is desirable to design interfaces with minimal cross-interface logic delay.

In order to analyze the logic delays of LID-2ss, LID-1ss, and SELF's interface logic, we synthesized their RTL implementations[3] and mapped with a 90nm industrial standard cell library using Synopsys Design Compiler [22]. As shown in Fig. 13(a), the interface logic is assumed to drive optimally buffered wires [4], [23]. The critical logic delays within each individual component and across the logic of communicating interface are then extracted using Design Compiler's static timing analyzer.

For LID-2ss and LID-1ss designs based on edge-triggered flip-flops (FFs), the slack is derived by deducting the maximum logic delay between two flip-flops and the flip-flop setup time from the clock period. For SELF design based on level-sensitive latches, the slack is calculated by subtracting the maximum logic delay between two active-high (or active-low) latches and latch setup time from the clock period.[4] When calculating cross-interface slacks, as shown in Fig. 13(a) (LID-2ss and LID-1ss) and Fig. 13(b) (SELF), the delays of forward paths (data and $void/valid$) t_f and of backward paths ($stop$) t_b are both considered (without counting delays of buffered wires across the channel).

Fig. 14(a) and Fig. 14(b)-14(e) summarize the results of our analysis of the impacts of logic delay on system performance in terms of the minimum slacks and the maximum physical lengths of interconnects as allowed by the three sets of interface logic respectively. Fig. 14(a) reports the minimum slacks left in each interface logic and four combinations of communicating interface logic at 500 MHz clock rate, while ignoring the delays of buffered interconnects. The channel width is assumed to be 64-bit wide, and each core has two input channels. The more slack an interface logic has, the faster clock rate can be applied. LID-1ss has more slack in all but one scenarios, and thus enjoys faster clock rates than LID-2ss and SELF. Conversely, the slack of the shell-shell pair in SELF is significantly low. This may either limit the system clock frequency, or require the insertion of an additional elastic buffer between the two shells to increase available slack. But inserting an EB introduces a void token and, therefore, it may possibly lower the system throughput.

Fig. 14(b)-14(e) report maximum allowable wire lengths between four different pairs of communicating interface components at various clock frequencies. LID-1ss presents the maximum interconnect lengths in all four possible scenarios. Note that the "X" marks indicate that at the given clock frequency the timing constraint is not met in the corresponding pair of communicating interface logic, so additional relay stations/elastic buffers must be inserted between them or the pair must be physically close to avoid long interconnect wires. The former solution might decrease system throughput; the latter might constrain physical design tools.

The maximum physical lengths of interconnects allowed between the RS-shell or shell-shell pairs in SELF are shorter than what the corresponding slacks imply. This is because the join structure used in the two-input "shell" in SELF creates multiple combinational paths running across a single channel twice or spanning across two channels, as indicated in Fig. 13(b). Therefore the slack available between the two-input shell and its uplink counterparts are shared among the interface logic and the corresponding forward path and backward path between them. As a result, the join structure allows a much shorter physical length for the interconnects, and physical design tools must be very careful to "balance" the lengths of the "joined" wires to avoid timing violations. Note that these combinational paths are introduced by the interface logic with multiple input channels (here the two-input shell), regardless of whether the senders are elastic buffers or other processing cores.

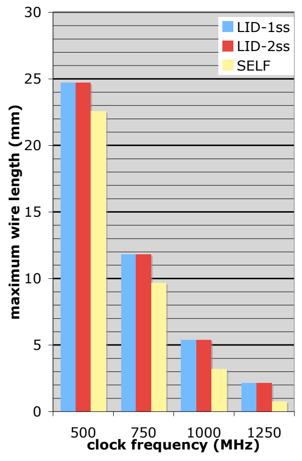
It should be noted that the combinational paths created by the join structure are *inherent* to SELF, and cannot be avoided by redesigning the join structure. SELF chooses to buffer unused valid data at the immediate sender's end to avoid using input queues at the receiver's end. Hence a multi-input core receiving an invalid token must request the re-transmission of

[3]We implemented the proposed LID-1ss design, and obtained the RTL implementations from the authors of the LID-2ss and SELF.

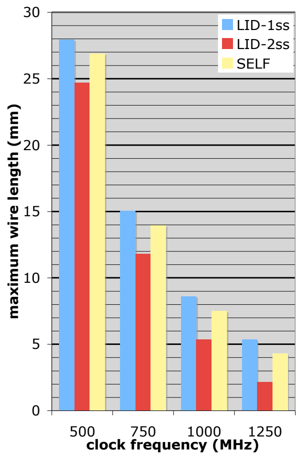
[4]Although a latch-based design like SELF allows *time borrowing*, the total delays over a path spanning a chain of active-high and -low latches must stay within a fixed number of clock periods determined by the number of high-low latch pairs. To simplify the analysis without sacrificing accuracy, we assumed that the path between two active-high (or -low) latches must be within one clock period.

	shell	RS	shell-RS	RS-RS	RS-shell	shell-shell
LID-1ss	1.23	1.28	1.32	1.5	1.33	1.24
LID-2ss	1.14	1.23	1.32	1.32	1.1	1.27
SELF	1.24	1.00	1.21	1.44	1.31	0.92

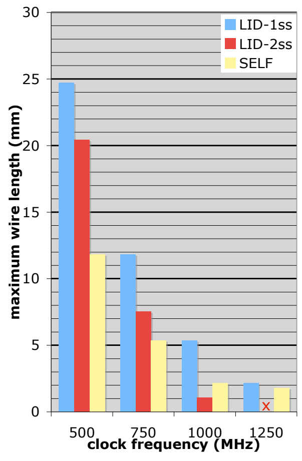
(a) Slacks (in nanoseconds) of interface logic at 500 MHz clock rate.



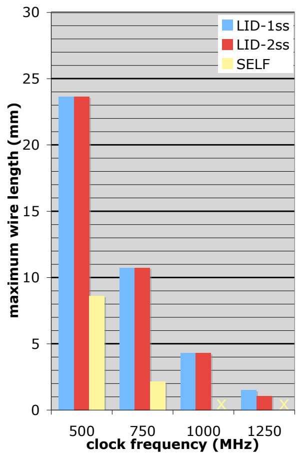
(b) 2-in-2-out shell ⟶ RS



(c) relay station ⟶ relay station



(d) RS ⟶ 2-in-2-out shell



(e) 2-in-2-out shell ⟶ 2-in-2-out shell

Fig. 14. Minimum slacks and maximum physical lengths of interconnects allowed by interface logic.

all of the valid tokens received at the *same* clock cycle as they arrive. This means that combinational paths between the communicating interface logic are required.

The above analysis of logic delay shows that the proposed LID-1ss interface logic can support higher system clock rate and throughput than LID-2ss and SELF. The reason is that the interface logic of LID-1ss has more slack, and requires a smaller number of wire pipelining elements (relay stations) because it allows longer interconnect between its interface logic. Latch-based SELF design does provide additional flexibility to the physical design tools because time borrowing allows an EB to tolerate varying wire delays and thus to be placed in a wider range of area. But this flexibility comes at the price of introducing new combinational paths whenever join structures are used for processing cores with multiple input channels.

Area Overhead Comparisons. Shell interfaces, relay stations and elastic buffers do occupy active silicon area and therefore represent a necessary area overhead of any latency-insensitive design approach. We analyzed and compared area overhead figures for the three approaches discussed in this paper after performing logic synthesis and technology mapping.

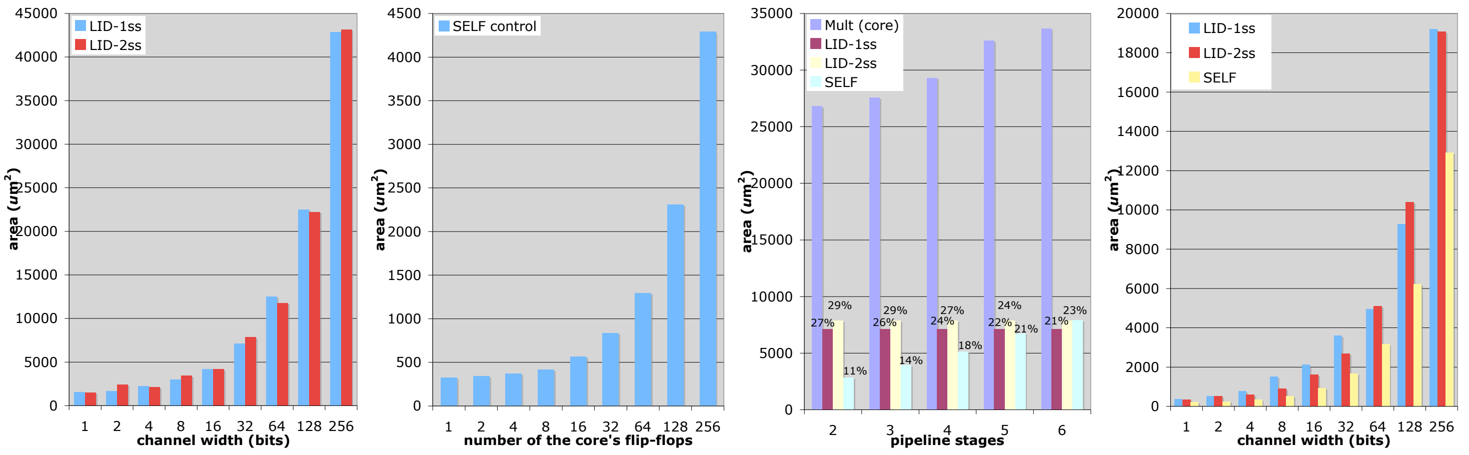
Fig. 15(a) reports the area overhead of the shell designs in LID-2ss and LID-1ss over a range of different channel widths; Fig. 15(b) shows the corresponding overhead incurred in elasticization of processing cores with different number of FFs. The area overhead of shells in both LI designs is dominated by input queues, whose area depends on the widths of input channels. This also means that the area of the two LID shells is roughly the same as shown in Fig 15(a). For SELF, the area overhead of elasticizing a processing core grows with the number of flip-flops contained in the core. This is because the substitute latches require additional input steering logic as illustrated in Fig. 3.

Fig. 15(c) compares the area overhead of the three shells applied to 32×32 pipelined multipliers from the Synopsys DesignWare [24] IP core library. For a number of pipeline stages varying from 2 to 6 the bar diagram reports the absolute area of the synthesized multipliers as well as the area of the corresponding shell interfaces for LID-2ss, LID-1ss, and SELF. The overhead ratios between each shell's area and the multiplier's area is labeled on top of each corresponding bar. As expected, the absolute area of the shells in LID-2ss and LID-1ss are constant regardless the number of pipeline stages, but the area overhead ratio of the LID-1ss shell's area drop from 27% to 21% as the multiplier's logic grows (the same trend applies to LID-2ss). In contrast, SELF shell's area grows with the number of pipeline stages, and is higher than LID-2ss and LID-1ss in the case of the 6-stage pipelined multiplier.Note that the area overhead ratios of SELF shells to the multipliers actually *increases* as the multiplier's logic increases. Overall the shell design of LID-2ss and LID-1ss scale better with the internal complexity of the logic core than their SELF counterparts. Although in this example the area overhead of LID-1ss and LID-2ss are significant when compared to the cores, we expect this to become fairly moderate when LID is applied to more complex IP cores than a pipelined multiplier.

Fig. 15(d) reports the area of relay stations and elastic buffers over a range of different channel widths. The area overhead of the latch-based SELF EBs is $2/3$ of their counterparts in LID-2ss and LID-1ss. This is due to SELF clever use of two latches to provide the necessary twofold capacity. Due to the more complex steering logic between the two flip-flops the LID-1ss's relay stations are slightly larger than LID-2ss's ones.

VI. Concluding Remarks

We proposed a new class of interface circuits to support latency-insensitive design based on LID-1ss, a simpler latency-insensitive protocol. We presented a detailed experimental analysis comparing LID-1ss to the original protocol discussed in [1], [9], that we called LID-2ss, as well as to the synchronous elastic architecture (SELF) proposed in [2]. We showed that LID-1ss offers clear improvements in terms of



(a) 2-in-2-out shells in LID-2ss and LID-1ss

(b) 2-in-2-out EB control

(c) Area overhead of encapsulating a pipelined multiplier.

(d) RS and EB

Fig. 15. Area of synthesized interface logic components.

area overhead and logic delay with respect to LID-2ss and in terms of both logic delay and processing throughput with respect to SELF.

Both LID-2ss and LID-1ss are aimed at a faithful application of the latency-insensitive design methodology [1], while SELF partially departures from it. Basically SELF embraces the LID idea of automatic wire pipelining via the insertion of elastic buffers, but replaces the notion of shell encapsulation of a processing core with an elasticization step that requires the replacement of all flip-flops in a core with a pair of transparent latches of different polarity (plus some necessary control logic). Therefore, SELF is arguably an approach that is both more intrusive and of more limited applicability. In fact, while LID-2ss and LID-1ss do not make any assumption on how a processing core is implemented (this can be either a soft IP core or a hard IP core since no modification to its logic is required), SELF is not applicable to hard IP cores and in order to manipulate a synthesizable soft IP core which is described using a hardware-description language (HDL) it requires direct support from the HDL compiler with respect to memory elements inference and replacement.

References

[1] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, “A methodology for “correct-by-construction” latency insensitive design,” in *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, Nov. 1999, pp. 309–315.

[2] J. Cortadella, M. Kishinevsky, and B. Grundmann, “Synthesis of synchronous elastic architectures,” in *Proc. of the Design Automation Conf. (DAC)*, 2006, pp. 657–662.

[3] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, “Theory of latency-insensitive design,” *IEEE Tran. on Computer-Aided Design of Integr. Circuits and Syst.*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[4] R. Ho, K. W. Mai, and M. A. Horowitz, “The future of wires,” *Proc. of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.

[5] D. Matzke, “Will physical scalability sabotage performance gains?” *IEEE Computer*, vol. 30, pp. 37–39, Sep. 1997.

[6] L. P. Carloni and A. L. Sangiovanni-Vincentelli, “Coping with latency in SOC design,” *IEEE Micro*, vol. 22, no. 5, pp. 24–35, Sep-Oct 2002.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, “The synchronous language twelve years later,” *Proc. of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[8] E. A. Lee and A. Sangiovanni-Vincentelli, “A Framework for Comparing Models of Computation,” *IEEE Tran. on Computer-Aided Design of Integr. Circuits and Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[9] L. P. Carloni, “The role of back-pressure in implementing latency-insensitive design,” in *Second Intl. Workshop on Formal Methods for Globally Async. Locally Sync. Arch. (FMGALS)*, ser. Electronic Notes on Theoretical Computer Science, vol. 146, no. 2, Jul. 2006, pp. 61–80.

[10] R. Collins and L. P. Carloni, “Topology-based optimization of maximal sustainable throughput in a latency-insensitive system,” Dept. of Computer Science, Columbia University, Tech. Rep. CUCS-008-07, 2007.

[11] A. Agiwal and M. Singh, “An architecture and a wrapper synthesis approach for multi-clock latency-insensitive systems,” in *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, 2005, pp. 1006–1013.

[12] M. R. Casu and L. Macchiarulo, “A new approach to latency insensitive design,” in *Proc. of the Design Automation Conf. (DAC)*, 2004, pp. 576–581.

[13] T. Chelcea and S. M. Nowick, “Robust interfaces for mixed-timing systems,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 8, pp. 857–873, 2004.

[14] R. Lu and C.-K. Koh, “Performance analysis of latency-insensitive systems,” *IEEE Tran. on Computer-Aided Design of Integr. Circuits and Syst.*, vol. 25, no. 3, pp. 469–483, Mar. 2006.

[15] M. Singh and M. Theobald, “Generalized latency-insensitive systems for single-clock and multi-clock architectures,” in *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2004, pp. 21 008–21 013.

[16] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla, “Validating families of latency insensitive protocols,” *IEEE Tran. on Computers*, vol. 55, no. 11, pp. 1391–1401, 2006.

[17] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “NuSMV: a new Symbolic Model Verifier,” in *Proc. of the Intl. Conf. on Computer-Aided Verification (CAV)*, July 1999, pp. 495–499.

[18] L. P. Carloni and A. L. Sangiovanni-Vincentelli, “Performance analysis and optimization of latency insensitive systems,” in *Proc. of the Design Automation Conf. (DAC)*, Jun. 2000, pp. 361–367.

[19] C. V. Ramamoorthy and G. S. Ho, “Performance evaluation of asynchronous concurrent systems using Petri nets,” *IEEE Tran. on Software Engineering*, vol. 6, no. 5, pp. 440–449, Sep. 1980.

[20] R. M. Karp, “A characterization of the minimum cycle mean in a digraph,” *Discrete Mathematics*, vol. 23, pp. 309–311, 1978.

[21] A. Dasdan and R. Gupta, “Faster maximum and minimum mean cycle algorithms for system-performance analysis,” *IEEE Tran. on Computer-Aided Design of Integr. Circuits and Syst.*, vol. 17, pp. 889–899, Oct. 1998.

[22] *Design Compiler User Guide*, Synopsys, Inc.

[23] J. M. Rabaey, A. Chandrakasan, and B. Nikolić, *Digital integrated circuits: a design perspective*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2002.

[24] *DesignWare User Guide*, Synopsys, Inc.