What is the cost of Delay Insensitivity? *

Hiroshi Saito
Univ. of Aizu
Japan

Alex Kondratyev
Univ. of Aizu
Japan

Jordi Cortadella
Univ. Politècnica
Catalunya, Spain

Luciano Lavagno
Univ. of Udine
Italy

Alexander Yakovlev
Univ. of Newcastle
upon Tyne, UK

Abstract

Deep submicron technology calls for new design techniques, in which wire and gate delays are accounted to have equal or nearly equal effect on circuit behaviour. Asynchronous speed-independent (SI) circuits, whose behaviour is only robust to gate delay variations, may be too optimistic. On the other hand, building circuits totally delay-insensitive (DI), for both gates and wires, is impractical. The paper presents an approach for automated synthesis of *globally DI and locally SI* circuits. It is based on *order relaxation*, a simple graphical transformation of a circuit's behavioural specification, for which Signal Transition Graph, an interpreted Petri net, is used. The method is successfully tested on a set of benchmarks and a realistic design example. It proves effective showing average cost of DI interfacing at about 40% for area and 20% for speed.

1 Introduction

As the scale of integration increases, managing synchronization and control of computation and communication on deep sub-micron (DSM) integrated circuits using a global clock is becoming increasingly difficult. Asynchronous systems, free from the clock, offer a number of potential advantages, such as reduced risk of synchronization failures, low power consumption, improved noise and electromagnetic compatibility to name but a few.

Interpreted PNs (called Signal Transition Graphs (STGs) [2, 7]) are widely used in specifying an asynchronous system behavior in a formal timing diagram style. It is known that from an STG one can derive an implementation which has the speed-independent (SI) property, i.e., such that the behavior of the circuit is correct under any distribution of gate delays. The main drawback of SI circuits is in neglecting the influence of wire delays on circuit behavior. For the DSM technology, where wire and gate delays can become (over long wires) equally important, the implementation should be targeted at delay-insensitive (DI) circuits [19], which allow wire delays to be of arbitrary value. In fact, a reasonable strategy for future technologies would require one to partition the system into blocks of relatively small size, for which the designer can keep control on wire delays (SI blocks) [10, 18], with a DI interface between blocks [14].

Logic synthesis of hazard-free asynchronous control circuits from STG specifications has reached a good level of maturity and automation (comparable in several respects to that of synchronous FSMs), as exemplified by the tool Petrify [3]. Asynchronous CAD is being used both for industrial and academic design experiments [12, 15]. It is therefore most natural to introduce DI interfacing into the existing STG-based synthesis framework, so far supporting the synthesis of both speed-independent circuits and circuits optimized using a variety of timing assumptions [4].

This approach clearly differs from early ideas about externally DI and internally timed Macro-modules [16, 11], as well as from more recent implementation strategies for quasi-DI and DI circuits [8, 1]. The former relied on specially designed and potentially slow meta-stability detection circuits. The latter were based primarily on syntax-direct translation techniques from process algebraic specifications, rather than on logic synthesis with inherent optimization under different cost functions. An alternative technique, that permits a certain level of delay-insensitivity for inter-block communication and relies on local timing conditions (Fundamental Mode operation), is based on a Burst Mode (BM) Finite State Machine specification [9]. The BM approach, however, is not very flexible from the point of view of the level of concurrency and distribution of control flow, as we will discuss in Section 5. STG-based synthesis, which supports a more powerful Input/Output operation mode, allows one to build circuits with a completely *distributed* environment, as opposed to the *centralized* environment assumed by the FM conditions.

In this paper we investigate the STG-based approach to the design of locally SI and globally DI asynchronous control circuits, by posing the problem at the behavioral (STG) level. We believe that our method would be particularly effective in the following two design flow scenarios, both resulting in fairly large STG specifications that would benefit from DI interfacing:

1. circuits specified using a high-level behavioral notation (such as CSP or high-level Petri net), subse-

*This work was partially supported by ESPRIT ACiD-WG (21949), CICYT TIC 98-0410 and TIC 98-0949, UK EPSRC GR/L24038 and GR/L93775, British Council (Acción Integrada MDR/1998/99/2463).

0-7803-5832-5/99/ $10.00 © 1999 IEEE

quently refined into a large binary encoded STG,

2. control circuits for regular control structures.

For both scenarios it is appropriate to partition the large specification at the STG level and synthesize its blocks with DI interfaces. In this decomposition, a natural question arises: what is the cost of DI interfacing?

In order to answer this question, we developed the theory of iterative transformations of SI specifications towards DI interfacing (Section 3). The suggested approach was checked experimentally using the known set of asynchronous benchmarks and synthesis tool Petrify [3]. In our experiments (Section 4) we first partition the given circuit into two parts at the STG level, and then consider each part separately with the DI interface in between. We compared the original circuit (entirely SI) against the new one (SI circuit with DI interface). The results of this comparison show that the cost of DI interfacing is on average about 36% for area and 20% for performance. These figures are quite encouraging because in the known methods of DI synthesis the area and performance costs are much higher [6]. Finally, in Section 5 we generalize the proposed approach to obtain a globally DI implementation of a totally different specification formalism (BM machines). We believe that the combination of the SI and DI implementation styles opens up new perspectives for efficient asynchronous design for DSM technologies.

This work focuses on the automatic introduction of DI interfaces in the control part of the design. There are several possible approaches to handling the data part as well.

1. The data-path can be designed using a DI-encoding (e.g., dual rail, Sperner codes etc. [20]).

2. If a more efficient bundled data approach is chosen for the data-path, like in Micropipelines [17], the ordering conditions between data and a corresponding request signal are simpler to satisfy than the ordering conditions between several control signals, possibly coming from different parts of the overall design (e.g. two request lines accompanying a data bus and an address bus in an inter-module interface as in our first example in Figure 1). In particular, routing can be constrained so as to keep the skew of a bundle of wires to be under a small upper bound.

Moreover, many designs involve large pieces of control-dominated logic without any data-path processing. Those include modulo-N counters, multi-way pulse generators and distributors, arbiters etc. Their cell-by-cell layout, with DI interfaces between cells, which can internally be designed as SI or even locally timed, would make them suitable as firm or hard macros in a DSM context.

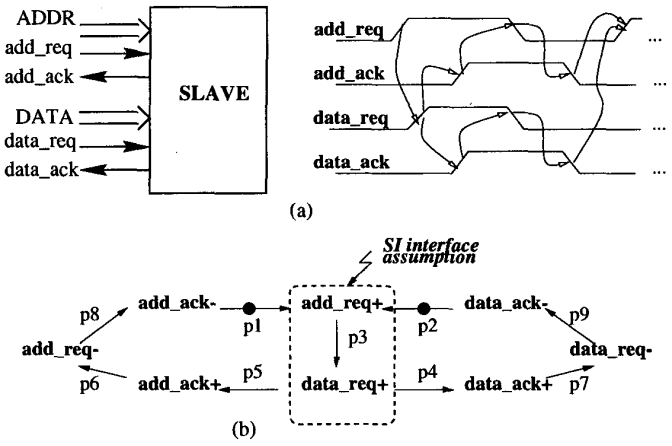


Figure 1: Simple asynchronous interface

2 Theoretical background

Figure 1.a shows a simple interface between two modules in an asynchronous system, a master (e.g., a processor) and a slave (e.g., memory). The interface involves two signal handshakes, one for controlling the transmission of an address (add_{req} and add_{ack}) and another for data ($data_{req}$ and $data_{ack}$). The timing diagram shown in Figure 1.a defines the synchronization protocol between the handshakes for the case of writing data into the slave. This protocol allows an additional skew compensation between address and data, making sure that the address is delivered to the slave strictly before data, thus permitting an additional delay in the corresponding address decode logic. This condition is captured by the arc directed from the rising edge of the add_{req} signal to that of $data_{req}$.

Figure 1.b shows the Petri Net (PN) corresponding to the timing diagram of the controller. All events in this PN are interpreted as signal transitions: rising transitions of signal a are labeled with "$a+$" and falling transitions with "$a-$". We also use the notation $a*$ if we are not specific about the sign of the transition. Petri Nets with such an interpretation are called *Signal Transition Graphs (or* STG*s)* [2]. STGs are typically represented in a "shorthand" form, where places with one input and one output arc are implicit.

An STG transition is *enabled* if all its input places contain a token. In the initial marking $\{p1, p2\}$ of the STG in Figure 1.c transition $add_{req}+$ is enabled. Every enabled transition can fire, removing one token from every input place of the transition and adding one token to every output place. After the firing of transition $add_{req}+$ the net moves to a new marking, $\{p3\}$, where $data_{req}+$ becomes enabled.

Transitions in STG could be involved in different order-

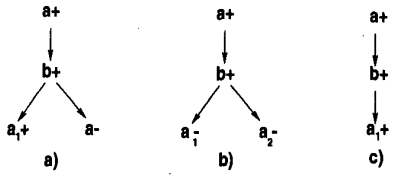


Figure 2: Consistency violations in STG

ing relations. Transitions $a*$ and $b*$ are in *direct conflict* if there exists a reachable marking in which both of them are enabled but firing of one of them disables the other. If $a*$ and $b*$ are enabled in some reachable marking but are not in direct conflict, they are *concurrent*. Conflict relations can be generalized by considering the transitive successors of directly conflicting transitions. Transitions which are not concurrent and are not in (transitive) conflict are *ordered*. An STG is *consistent* if in every transition sequence from the initial marking, rising and falling transitions alternate for each signal.

There are two sources of consistency violation in an STG:

1. *Auto-concurrency*, due to concurrency of transitions of the same signal (see Figure 2.a,b) and
2. *Switchover incorrectness*, due to ordered rising (falling) transitions which have no falling (rising) transition in between (see Figure 2.c).

The set of all signals in a STG is partitioned into a set of *inputs*, which come from the environment, and a set of *outputs* that must be implemented.

In addition to consistency, the persistency property is required for an STG to be implementable as a hazard-free asynchronous circuit. An event a^* is *persistent* in marking m if it is enabled in m and remains enabled in any other marking reachable from m by firing another event b^*. An STG is *output-persistent* if all output signal events are persistent in all reachable markings and input signals cannot be disabled by outputs. Output persistency therefore only allows input events to be in direct conflict (thus modeling non-deterministic choice in the environment).

The following important statement was proved in [2]: *an STG can be implemented by a speed-independent circuit if it is consistent and output-persistent.*

3 Delay-Insensitive Interfacing

Our approach has two distinctive features:

- It is focused not on *total* delay-insensitivity but on *delay-insensitive interfacing* only. The basic assumption is that within a module the designer or a physical design tool can keep wire delays under control and hence there is no point to ensure delay-insensitivity at the level of events internal to the module.
- Contrary to conventional approaches to DI synthesis, the tasks of designing a module and its environment are considered separately. This results in asymmetric DI interfacing requirements: only inputs are required to be *accepted* in a delay-insensitive fashion by the circuit, because delay-insensitivity with respect to outputs matters only when the implementation for the environment is synthesized.

The above conditions lead to a more relaxed axiomatic definition of delay-insensitive *interfacing* with respect to the classical definition of *delay insensitivity* given in [19]. A specification satisfies the *delay-insensitive interfacing requirement* if it meets the following conditions:

1. *No auto-concurrency.*
2. *Alternating inputs* (input events cannot be ordered with other input events).
3. *No cross-disabling* (inputs and outputs cannot disable each other).

Our design framework uses STGs as a model basis. The natural question is: what are the implications of the requirements of DI interfacing for the properties of the original STG?

Proposition 3.1 *A consistent and output persistent STG satisfies DI interfacing conditions if and only if no input transition directly precedes another input transition.*

The proof is trivial: non-auto-concurrency is a necessary condition of STG consistency, absence of cross-disabling is guaranteed by output persistency and alternation of inputs directly comes from the definition of DI interfacing.

Proposition 3.1 gives an idea about the places where DI interfacing might be violated in an STG: these are STG fragments in which input transitions are directly causally related. The addition of arbitrary delays to every input wire may unpredictably alter the order of originally ordered inputs to a module. This means that from the module point of view such inputs become concurrent. Hence the transformation of an STG for DI interfacing removes direct causal dependencies between inputs and makes them concurrent. This transformation can be performed by iterative application of a simple operation that is called *order relaxation* and is intuitively defined in Figure 3. Note that order relaxation makes previously ordered events a and b to occur concurrently "in a burst".

The following two properties of order relaxation help to clarify the transformation towards DI interfacing. Their proofs can be found in [13].

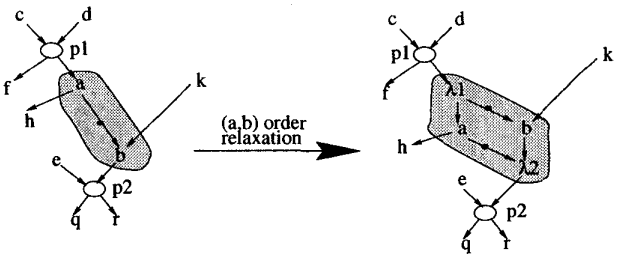


Figure 3: Order relaxation

Property 3.1 *Order relaxation between events a and b preserves pairwise ordering relations between all events except for a and b.*

Property 3.2 *Order relaxation between two events preserves output persistency in an STG.*

When in the original STG two inputs are directly causally related, then DI interfacing can be obtained only by an order relaxation between them. The latter, by Property 3.2, does not cause any new cross-disabling to occur. Unfortunately not all the requirements of DI interfacing are safely preserved during order relaxation. Indeed if events a and b correspond to transitions of the same signal their order relaxation immediately produces auto-concurrency. If non-auto-concurrency is preserved the above transformation is *strictly delay-insensitivity increasing* and by iterative application of it eventually (if non-auto-concurrency is preserved) all the requirements of DI interfacing are met in the modified specification.

The algorithm for STG transformation to ensure DI interfacing is presented in Figure 4. The result of the algorithm is either a new STG in which DI interfacing requirements are satisfied or a *failure* in case when input order relaxation leads to auto-concurrency. The latter implies that the original STG cannot be implemented with DI interfacing.[1]

Figure 5 illustrates the transformation to DI interfacing for the *chu133* benchmark example (DI violations are denoted by shading). DI interfacing is achieved by iterative application of order relaxation between input events.

4 Experimental results

Two types of experiments, corresponding to the design scenarios outlined in the introduction, have been performed to test the proposed method.

Case study: controller for analog-to-digital converter. In the first example, we consider the synthesis of a scalable control circuit, whose STG specification has a regular structure. It originates from a practical case study of

[1] Indeed, Property 3.1 implies that the order in which one chooses the pairwise order relaxation between inputs is irrelevant.

```
Input:  STG A = ⟨E, F, m0⟩ (E - events,
        F - precedence relations, m0 - initial marking)
Output: STG A_di = ⟨E, F', m0'⟩ with DI interfacing

  foreach input events a and b, a → b do
    /* order relaxation step */
      remove causal arc (a, b)
      /* ordering predecessors of a with b */
      foreach predecessor p of a do
        add arc p → b;
      /* ordering successors of b with a */
      foreach successor s of b do
        add arc a → s;
      /*modify m0 if necessary */
      foreach initially marked arc (p, a) do
        m0((p, b))+ = m0((p, a));
      foreach b → s with arc (b, s) initially marked do
        m0((a, s))+ = m0((b, s));
      if (a, b) is initially marked then
        foreach arc (p, b) do m0((p, b))+ = m0((a, b))
    if STG A becomes auto-concurrent then exit(failure);
  endfor
```

Figure 4: Algorithm for ensuring DI interfacing.

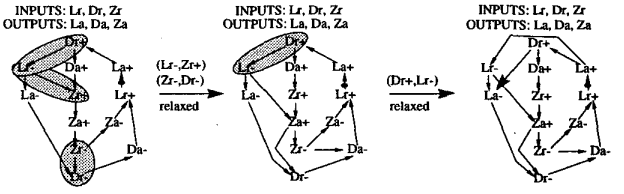


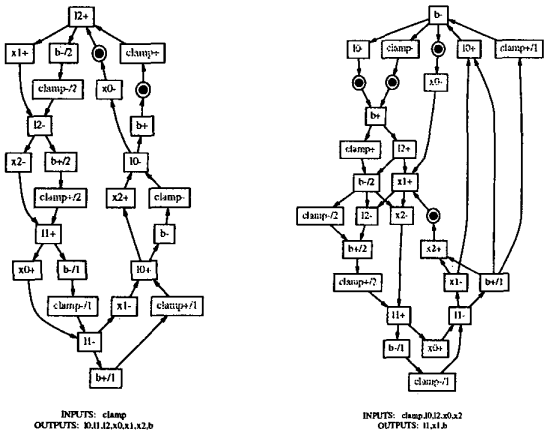
Figure 5: Example of order relaxation

an asynchronous SI controller for an analog-to-digital converter (ADC) [5].

This ADC implements a well-known successive approximation algorithm. According to this algorithm, a comparator is iteratively activated to compare the value of the given input voltage with the approximate voltage produced by a digital-to-analog converter (DAC), whose digital input comes from a register in which the n-bit value is refined bitwise, starting from the most significant bit. Each refining bit is produced by a one-bit buffer connected to the output of the comparator. The use of asynchronous logic allows this system to avoid synchronization errors due to meta-stability (which is known to be a problem in clocked converters due to the analog part of the circuit), and to smooth out the temporal effect of potential meta-stability resolution [5] over the whole conversion period.

The central part of the asynchronous ADC, which controls copying a bit value from the one-bit buffer to the n-bit register with a single bit shift, is an n-way scheduler; it is functionally similar to a classical pulse distributor. The scheduler's behaviour can be specified by an STG whose structure is regular. The specification of a scheduler with 3 cells is shown in Figure 6.a.

From the analysis of the causal relations between events one could see that the behavior of the i-th cell of the scheduler depends on the state of the (i-1)-th and (i+1)-th



(a) (b)

Figure 6: A specification of 3-cell scheduler (a) and the input order relaxation for the cell 1 (b)

cells, together with the signal *clamp* (produced by completion detection logic in a storage buffer; see Figure 7.a). Hence the speed-independent implementation of the scheduler might be obtained directly using the STG of Figure 6, which gives the following logic circuit:

$$l_i = clamp\,(\overline{x}_{i-1}\overline{x}_{i+1} + l_i) + l_i\overline{x}_{i-1};$$
$$x_i = \overline{x}_{i-1}x_i + l_i + l_{i+1};$$
$$b = \overline{l}_1\overline{l}_2 \ldots \overline{l}_n$$

The drawback of the SI implementation is that the designer is responsible for satisfying the SI assumptions about wiring delays between scheduler cells.

In case of conversion with a large data path (i.e., with many cells in the scheduler) or in order to increase the layout flexibility, it could be more convenient to partition the whole scheduler circuit into smaller parts. These could be placed in different positions on the chip (not necessarily adjacent) and thus require DI interfacing, while within each part the designer could still rely on the SI hypothesis, as shown in Figure 7.b.

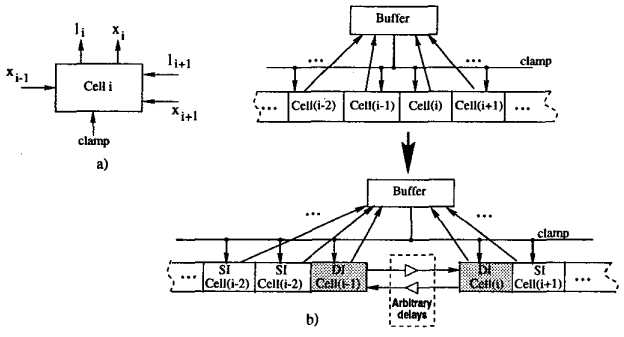


Figure 7: A scheduler circuit structure

In order to evaluate an upper bound for the cost of partitioning the scheduler we consider a partition into blocks with one cell each. Each cell communicates with its neighbors in a DI fashion, and therefore synthesis of such a scheduler reduces to the task of DI interfacing between cells. The result of order relaxation on the STG is shown in Figure 6, where for the i-th cell all the transitions of the inputs coming from the (i-1)-th and (i+1)-th cells are concurrent. The result is shown in Figure 6.b. From this STG the following logic equations can be derived:

$$l_i = clamp\,(\overline{x}_{i-1}\overline{x}_{i+1}x_i b + l_i) + l_i\overline{x}_{i-1};$$
$$x_i = \overline{x}_{i-1}(x_i + x_{i+1}l_{i+1}) + l_i;$$
$$b = \overline{l}_1\overline{l}_2 \ldots \overline{l}_n(b + \overline{clamp}) + b\,\overline{clamp}$$

A comparison between the SI and DI implementations shows that the latter is about 38% larger. We have also analyzed the performance of the SI and DI implementations, using *logic simulation*. We have synthesized both the scheduler circuit and its environment and simulated the resulting logic netlist. The degradation of performance due to the increased complexity is about 7%.

It is worth noting that these number are significantly lower than those usually reported when referring to synthesis results for DI implementations (see e.g [6] where the 3 times overhead was reported for a DI implementation of a stack against its SI countepart). The reason for that lies in our more flexible design strategy, that is *speed-independent circuits with DI interfacing instead of totally DI solutions.*

Delay-insensitive decomposition. Another group of experiments was targeted at DI decomposition of a relatively complex circuit into two simpler subcircuits with a DI interface. The experiment (illustrated in Figure 8) started from a well-known asynchronous benchmark set, in which also the environment was synthesized (thus yielding circuits without inputs). The set of signals of each benchmark was partitioned into two groups, thus yielding two separate modules as shown in Figure 8.b. Each module plays the role of the environment for its counterpart, and the interface between them is made delay-insensitive by applying order relaxation between events which are input for each module. Note that this process does not always converge to a correct implementation because of violations in non-auto-concurrency resulting from order relaxation (this means that decomposition for DI interfacing could be used as a guidance criterion for asynchronous system partitioning). For all cases where DI interfacing could be obtained for some wire partition, we compared the DI implementation (Figure 8.c) against the SI one (Figure 8.a) in terms of area and performance. The results are shown in Table 1. On average the area penalty is about 36% and the performance degradation is about 20%.

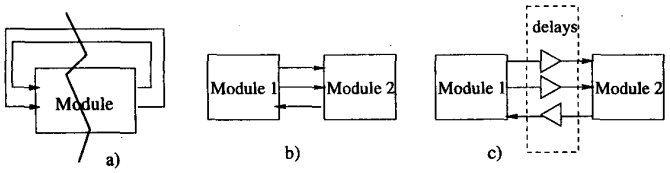


Figure 8: The experimental flow for DI decomposition

Circuit	Area			Performance		
	SI	DI	ratio	SI	DI	ratio
chu133 (1)	144	144	1.00	5026	5380	1.07
chu133 (2)	144	200	1.39	5026	6505	1.29
chu150	184	232	1.26	6268	7000	1.12
mmu (1)	360	552	1.53	5909	7246	1.23
mmu (2)	360	536	1.49	5909	7127	1.21
wrdatab	232	328	1.41	7325	9786	1.34
master-r (1)	376	584	1.55	5993	7613	1.27
master-r (2)	376	472	1.26	5993	7650	1.28
mr0 (1)	504	576	1.14	10094	10183	1.01
mr0 (2)	504	680	1.35	10094	10629	1.05
mr1 (1)	344	472	1.37	7799	9134	1.17
mr1 (2)	344	448	1.30	7799	9722	1.25
vbe10b	320	392	1.23	8053	8736	1.08
trimos	264	456	1.73	6764	7462	1.10
Total	4456	6072	1.36	98052	114173	1.16

Table 1: Area and performance penalty of DI interfacing

5 Other applications of DI interfacing

Up to now the DI interfacing approach has been discussed in the context of a system architecture consisting of *speed-independent* modules with *DI communication*. In that case the starting point for DI transformation is a speed-independent specification of the system, which is gradually refined to satisfy the conditions of DI interfacing.

However, the DI interfacing approach is certainly not restricted to that particular architecture. The main idea of the approach is that delays of system components (gates and wires) are roughly separated into two classes: controlled and uncontrolled. Components with controlled delays are restricted to be placed in close vicinity in the chip and are considered to be in the same logic module. Uncontrolled delays are due to communications between different logic modules [14]. Hence DI interfacing should work equally well when logic modules are implemented under more aggressive timing assumptions than speed-independence. It is the responsibility of the designer to ensure that each module functions correctly under these timing assumptions, while the overall correctness of the system is ensured by the DI interfacing between modules.

A possible extension of the suggested approach is illustrated below by implementing a system with DI interfacing starting from burst-mode (BM) behavioral specifications [9, 21].

Unlike the cases discussed in Section 4, it will result in order relaxation between *output* signals.

A burst-mode machine is an FSM-like specification in which each state transition is caused by a burst of concurrently switching inputs followed by a burst of concurrently switching output and state signals. Implementation of a BM specification relies on the so called *Fundamental Mode* hypothesis. This hypothesis states that the reaction of the environment is relatively slow, and a new input burst can only start when all the switching activity caused by the previous burst inside the circuit has stopped.

A burst-mode specification can be equivalently represented by an STG model. Figure 9(b) shows the BM specification of a FIFO for a SCSI controller [22], while Figure 9(c) shows its equivalent STG representation. Note that the fundamental mode assumption must be translated in the corresponding STG as causal arcs which synchronize output bursts (e.g., $aout-$ and $rout+$) with the next input bursts (e.g., $rin+$ and $ain+$)[2].

BM specification does not allow any direct ordering between inputs: either inputs occur in a burst (concurrently) or they are separated by transitions of output or/and state signals. This means that each *individual* BM machine naturally satisfies the conditions of DI interfacing (see Section 3). However ensuring DI interfacing between *a set of* communicating BM machines is more complicated than for SI modules, because the DI interfacing conditions (Section 3) take into account the behavior of input signals only.

Outputs can change in any order, and their proper reception must be ensured by the receiving modules. Therefore the notion of DI interfacing for a set of SI modules relies on "distributed responsibilities": each module can accept DI inputs, and all modules together cooperate in a globally DI fashion. This is reasonable because speed-independence makes only local timing assumptions (on gate fanout wires).

This approach will not work for the case of BM machines because of the non-locality of the fundamental mode assumption. Indeed for the FIFO in Figure 9(c) the fundamental mode assumption requires that the transitions $aout-$ and $rout+$ of *both* outputs precedes the transitions $rin+$ and $ain+$ of *both* inputs. However $ain+$ is produced by the (i+1)-th cell of the FIFO, *that receives only aout* as input, while $rin+$ is produced by the (i-1)-th cell, *that receives only rout as input.* Therefore the fundamental mode assumption is a timing requirement which cannot be ensured only by the local analysis of pairwise communications, but requires global timing analysis. Imposing timing assumptions on the speed of *independent* handshakes clearly contradicts the nature of DI communication. Hence for the case of outputs which communicate with different BM machines the fundamental mode assumption must be refined via relaxation of output synchronization. Note that, contrary to the input order relaxation in case of SI modules (which is defined purely by a syntactic transformation

[2]The dummy transition labeled with λ is equivalent to four arcs, between each output transition and each input transition.

of the STG), the refinement of the fundamental mode assumption requires additional semantic information about the structure of the distributed environment of the module (which signals communicate with which other modules).

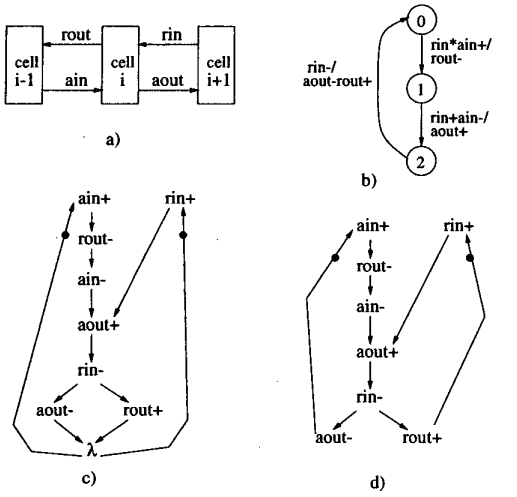


Figure 9: DI transformation of the BM FIFO specification

For the case of the FIFO in Figure 9(c), the refinement results in the relaxation of the synchronization for the *output* burst $aout-$ and $rout+$. Considering the natural separation of i-th FIFO cell environment into left and right handshakes $\langle rin, rout\rangle$ and $\langle ain, aout\rangle$, this results in the new STG shown in Figure 9(d).

Synthesizing the STG in Figure 9(d) requires one more state signal than the STG in Figure 9(c). The performance penalty was evaluated by checking the speed of a 3-cell FIFO buffer like the one shown in Figure 10.

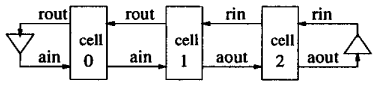


Figure 10: 3-cell FIFO buffer with closed environment

The resulting data is shown in Table 2 in columns BM and DI. Note that the only place where the fundamental mode assumption comes into play in the STG in Figure 9(c) is the output burst $\langle aout–rout+\rangle$. After its relaxation the STG in Figure 9(d) makes no implicit timing assumptions. Hence the FIFO buffer synthesized by this STG is in fact implemented as locally SI and globally DI.

We also exploited *locally* the same (usually reasonable) timing assumption that BM synthesis makes, namely that the delays of the circuit of a single FIFO cell are smaller than the delays of the handshakes between cells.

The results of this optimization are shown in Table 2 in column DIopt. Timing optimization improves the performance penalty to become only 23%.

Penalty	BM	DI	ratio	DIopt	ratio
Area (literals)	17	22	1.29	21	1.24
Performance	5438	9107	1.67	6707	1.23

Table 2: Area and performance penalties for DI interfacing of the BM FIFO

We also analyzed the cost of DI interfacing for other two parts of the SCSI controller (the Bus Interface Unit, BIU, and the Initiator Send, IS). The resulting area penalties (in terms of literals of the logic implementation) are presented in Table 3.

For these specifications the cost of transformation to DI interface is rather low, due to the fact that the fundamental mode is used only in a few cases in the SCSI controller, namely in 3 bursts out of 11 for IS and in 1 burst out of 9 for BIU.

Module	BM	DI	ratio
IS	54	59	1.09
BIU	41	42	1.03

Table 3: Area penalty for SCSI controller

6 Conclusions

Design styles which neglect wire delays seem to be overly optimistic even with the current technology, and will most likely become less and less applicable when moving to deep sub-micron implementations. The extreme case when wire delays are assumed to have arbitrary values leads to the well known delay-insensitive approach for circuit design. However delay-insensitive circuits are often unusable because of their excessive area and performance overheads. In this paper we suggested an approach which results in *partial* delay-insensitivity of an implementation. Under this approach a designer identifies a set of "dangerous" wires which should be implemented in a delay-insensitive fashion, while for the rest of a circuit other (more conventional) design styles might be applied. In particular, we used speed-independent implementation for the parts of a system in which wire delays could be controlled by the designer or a routing tool, and then applied the delay-insensitive hypothesis only to the wires running between such speed-independent "islands".

We have developed an automated method which transforms an originally speed-independent specification into a specification with DI interface. Contrary to the common belief about the high area and performance penalty of DI circuits, our experimental results show that the cost of DI interfacing is rather moderate: about 40% for area and 20% for speed. This is a direct consequence of a more flexible

strategy of partitioning a system into its speed-independent and delay-insensitive sub-domains.

Acknowledgments

We are grateful to Alexander Taubin from Theseus Logic Inc. for many useful discussions and critics.

References

[1] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[2] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[3] J. Cortadella, M. Kishinevsky, A.Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Lazy transition systems: application to timing optimization of asynchronous circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 324–331, November 1998.

[5] D. J. Kinniment, B. Gao, A. V. Yakovlev, and F. Xia. Toward asynchronous A-D conversion. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–215, 1998.

[6] M. Kishinevsky and Christian Nielsen. Naive design of an unbounded stack. Presentation at the Dagstuhl Seminar on Self-Timed Design, 1992.

[7] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[8] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[9] Steven M. Nowick and David L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 318–321. IEEE Computer Society Press, November 1991.

[10] R.H.J.M. Otten and R.K. Brayton. Planning for performance. In *Proceedings of the Design Automation Conference*, June 1998.

[11] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.

[12] S.-H.Chung and S.B. Furber. The design of the control circuits for an asynchronous instruction prefetch unit using STGs. In *Proc. Second Int. Workshop on Hardware Design and Petrin Nets (HWPN'99), Williamsburg, VA*, pages 131–148, June 1999.

[13] H. Saito, A.Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev. What is the cost of delay insensitivity? Technical Report 99-2-004, The University of Aizu, August 1999.

[14] C. L. Seitz. Chapter 7. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*. Addison Wesley, 1981.

[15] Ken Stevens, Shai Rotem, Steven M. Burns, Jordi Cortadella, Ran Ginosar, Michael Kishinevsky, and Marly Roncken. Cad directions for high performance asynchronous circuits. In *Proceedings of the Design Automation Conference*, pages 116–121, June 1999.

[16] Mishell J. Stucki, Severo M. Ornstein, and Wesley A. Clark. Logical design of macromodules. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 357–364, Atlantic City, NJ, 1967. Academic Press.

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

[18] D. Sylvester and K. Keutzer. Getting to the bottom of deep submicron. In *Proceedings of the International Conference on Computer-Aided Design*, November 1998.

[19] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[20] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[21] Kenneth Y. Yun and David L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, November 1992.

[22] Kenneth Y. Yun and David L. Dill. A high-performance asynchronous SCSI controller. In *Proc. International Conf. Computer Design (ICCD)*, pages 44–49. IEEE Computer Society Press, 1995.