# Handshake protocols for de-synchronization

I. Blunno\*, J. Cortadella†, A. Kondratyev‡, L. Lavagno\*‡, K. Lwin‡ and C. Sotiriou§

| \*Politecnico di Torino | †Univ. Politècnica de Catalunya | ‡Cadence Berkeley Labs | §ICS-FORTH |
| Torino, Italy | Barcelona, Spain | Berkeley, USA | Crete, Greece |

*Abstract—De-synchronization* appears as a new paradigm to automate the design of asynchronous circuits from synchronous netlists. This paper studies different protocols for de-synchronization and formally proves their correctness. A taxonomy of existing protocols for latch controllers is provided. In particular, four-phase handshake protocols devised for micro-pipelines are studied. A new controller with maximum concurrency for de-synchronization is also proposed. The applicability of de-synchronization on an implementation of the DLX microprocessor is also described and discussed.

## I. INTRODUCTION

The goal of this work is modest in the short term, but ambitious in the long term. For many years, our community of researchers has tried to persuade designers to use asynchrony in their circuits. Today we can say that this effort has had the impact of a few drops in the ocean.

We believe that there are two major reasons why asynchronous circuits have not been widely accepted:

- There are no good CAD tools that completely cover the design flow.
- Asynchrony involves changing most of the designers' mentality when devising the synchronization among different components in a system.

This work explains how asynchrony can be incorporated without changing the "synchronous mentality" and using conventional CAD tools. This is the short-term goal. After that, it will hopefully be easier to show that asynchronous circuits can perform better than the synchronous ones in different aspects, and thus pave the way for a truly asynchronous design flow. This the longer-term goal. We introduce *de-synchronization* as an intermediate step towards a wider acceptance of asynchronous circuits.

One could argue that the paradigm of *VLSI programming* [2], [16] provides a robust framework for synthesis. However, the specification model is based on the theory of communicating processes, which requires the aforementioned change of mentality, and the syntax-directed approach for synthesis concedes little support for logic-level optimizations.

The notion of *cycle* lives in the subconscious of most circuit designers. Finite state machines, pipelined microprocessors, multi-cycle arithmetic operations, etc, are typically studied with the underlying idea of cycle, which is inherently assumed to be defined by a clock. As an example, think about the traditional lecture on computer architecture explaining the DLX pipeline. One immediately imagines the students looking at the classical timing diagram showing the overlapped `IF-ID-EX-MEM-WB` stages, synchronized at the level of cycle. If you try to persuade the lecturer to explain the same ideas without the notion of cycle, you may find yourself involved in a tortuous crusade against a skeptical listener.

If we accept that cycles are nice for reasoning and designing, but we still want an underlying asynchronous behavior, we bring up the concept of *de-synchronization*. The essential idea is to start from a synchronous synthesized (or manually designed) circuit, and *replace directly the global clock network with a set of local handshaking circuits*. The circuit is then *implemented with standard tools, using a flow originally developed for synchronous circuits*. The only modification is the clock tree generation algorithm. With this approach we provide a design methodology that can be picked up almost instantaneously and without risk by an experienced team.

This work gets its inspiration from a number of contributions from past work, each providing a key element to a unique novel methodology. Many of the concepts that appear in this paper have been around for a long time: handshake protocols, asynchronous pipelines, local controllers, etc. The essential novelty of our contribution is that *it provides a fully automated synthesis flow, based on a sound theory that guarantees correctness, does not require any knowledge of asynchronous design by the designer, and does not change at all the structure of synchronous datapath and controller implementation, but only affects the synchronization network*. In particular, it starts from a standard synthesizable HDL specification or gate-level netlist, yet it provides several key advantages of asynchronicity, such as low EMI, global idling, and modularity.

We also argue that de-synchronization helps with determining the true speed of a circuit, by using standard functional testing equipment, thus providing means to partially cope with process variability. It handles variability *between dies*, while variability *within a die* must still be handled by using margins.

To show that the suggested methodology is sound, we provide formal proofs of correctness based on the theory of Petri nets. We study different handshake protocols for latch controllers and present a taxonomy determined by the degree of concurrency of each protocol. A controller that preserves the maximum concurrency for de-synchronization is also presented. Moreover, we validated our approach by comparing synchronous and de-synchronized designs of the DLX microprocessor [11]. Both designs were implemented using the same set of commercial EDA tools for synthesis, placement and routing. To the best of our knowledge, this is the first time an asynchronous design obtained through a conventional EDA flow does not show any penalty (area, power, performance) with respect to its synchronous counterpart.

## II. PREVIOUS WORK

Sutherland, in his Turing award lecture, proposed a scheme to generate local clocks for a synchronous latch-based datapath. His theory for asynchronous designs has been exploited successfully by both manual designs [9] and CAD tools [1]–[3]. That methodology is very efficient for dataflow type of applications but is less suitable to emulate the behavior of synchronous system by firing of local clocks in a sort of "asynchronous simultaneity".

In a different research area, Linder and Harden started from a synchronous synthesized circuit, and replaced each logic gate with a small sequential handshaking asynchronous circuit, where each signal was encoded together with synchronization information using

an LEDR delay-insensitive code [15]. That approach bears many similarities with ours, in particular because it generates an asynchronous circuit from a synchronous specification, but in our opinion it attempts to go too far because it transforms each combinational gate into a sequential block which must locally keep track of the odd/even phases. Thus it may have an excessive overhead, even when used for large-granularity gates such as in FPGAs. To alleviate this overhead, a coarse-grain approach was used in [19], but no direct apples-to-apples comparison with a synchronous design was presented there.

Similarly, Theseus Logic proposed a design flow [14] which uses traditional combinational logic synthesis to optimize the datapath, and uses direct translation and special registers to generate automatically a delay-insensitive circuit from a synchronous specification. That approach also has a high overhead, and requires designers to use a non-standard HDL specification style, different from the synchronous synthesizable subset.

Kessels et al. also suggested generating the local clocks of synchronous datapath blocks using handshake circuits [12], but used Tangram as a specification language. This has some advantages, in that synchronous block activation can be controlled at a fine granularity level as in clock gating, but does not use a standard synchronous RTL specification.

The generation of local clocks from handshaking circuitry while ensuring the global "synchronicity" was first suggested in [23]. That work however focused purely on implementation of control ignoring the datapath part of a system.

The closest approach to ours is a doubly-latched asynchronous pipeline suggested in [13]. That is the first work suggesting a conversion of synchronous circuits into asynchronous ones through replacement of flip-flops by master-slave latches with corresponding controllers for local clocking. Our paper extends the results from [13] by using more general synchronization schemes and provides a theoretical foundation for the de-synchronization approach, by proving a behavioral and temporal equivalence between a synchronous circuit and its de-synchronized counterpart.

We also extend with respect to our own previous work in [6], [7] because we use a much more concurrent synchronization mechanism (which we believe is *maximally concurrent* for this job), show how previously published handshake controllers can be derived from this maximally concurrent model by *concurrency reduction*, and finally prove its equivalence to the synchronous version.

## III. MARKED GRAPHS

*Marked Graphs* (MG) is the formalism used in this paper to model de-synchronization. They are a subclass of Petri nets [18] that can model decision-free concurrent systems.

*Definition 3.1 (Marked graph):* A *marked graph* is a triple $(\Sigma, \rightarrow, M_0)$, where $\Sigma$ is a set of events, $\rightarrow \subseteq (\Sigma \times \Sigma)$ is the set of arcs (precedence relation) between events and $M_0 : \rightarrow \longrightarrow \mathbb{N}$ is an initial marking that assigns a number of tokens to the arcs of the marked graph.

An event is *enabled* when all its direct predecessor arcs have a token. An enabled event can *occur* (fire), thus removing one token from each predecessor arc and adding one token to each successor arc. A sequence of events $\sigma$ is feasible if it can fire from $M_0$, denoted by $M_0 \xrightarrow{\sigma}$. A marking $M'$ is reachable from $M$ if there exist $\sigma$ such that $M \xrightarrow{\sigma} M'$. The set of reachable markings from $M_0$ is denoted by $[M_0\rangle$.

An example of marked graph is shown in Figure 3(b), where the events $A+$ and $A-$ represent the rising and falling transitions of signal $A$, respectively. In the initial marking (denoted by solid dots
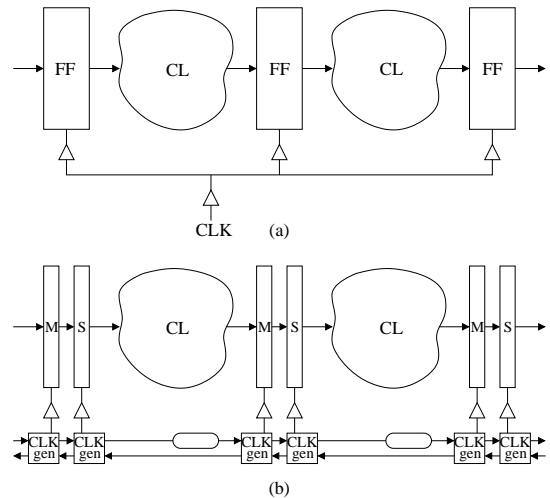


Fig. 1.   (a) Synchronous circuit, (b) de-synchronized circuit.

at arcs) two events are enabled: $B+$ and $D+$. The sequence of events $\langle D+\ D-\ C+\ B+\ B-\ A+\ C-\rangle$ is an example of a feasible sequence of the marked graph.

*Definition 3.2 (Liveness):* A marked graph is *live* if for any $M \in [M_0\rangle$ and for any event $e \in \Sigma$, there is a sequence fireable from $M$ that enables $e$.
Liveness ensures that any event can be fired infinitely often from any reachable marking.

*Definition 3.3 (Safeness):* A marked graph is *safe* if no reachable marking from $M_0$ can assign more than one token to any arc.

*Definition 3.4 (Event count in a sequence):* Given a firing sequence $\sigma$ and an event $e \in \Sigma$, $\overline{\sigma}(e)$ denotes the number of times that event $e$ fires in $\sigma$.

The following results were proven in [5] for *strongly connected* marked graphs.

*Theorem 3.1 (Liveness):* A marked graph is live *iff* $M_0$ assigns at least one token on each directed circuit.

*Theorem 3.2 (Invariance of tokens in circuits):* The token count in a directed circuit is invariant under any firing, i.e., $M(C) = M_0(C)$ for each directed circuit $C$ and for any $M$ in $[M_0\rangle$, where $M(C)$ denotes the total number of tokens on $C$.

*Theorem 3.3 (Safeness):* A marked graph is safe *iff* every arc belongs to a directed circuit $C$ with $M_0(C) = 1$.

In the rest of the paper, we will only deal with strongly connected marked graphs.

## IV. A ZERO-DELAY DE-SYNCHRONIZATION MODEL

The de-synchronization model presented in this section aims at the substitution of the global clock by a set of asynchronous controllers that guarantee an *equivalent* behavior. The model assumes that the circuit has combinational blocks (CL) and registers implemented with D flip-flops (FF), all of them working with the same clock edge (e.g. rising in Figure 1(a)).

### A. Steps in the de-synchronization method

The de-synchronization method proceeds in three steps:

1) *Conversion of the flip-flop-based synchronous circuit into a latch-based one (M and S latches in Figure 1(b)).*
   D-flip-flops are conceptually composed of master-slave latches. To perform de-synchronization, this internal structure is explicitly revealed (see Figure 1(b)) to:
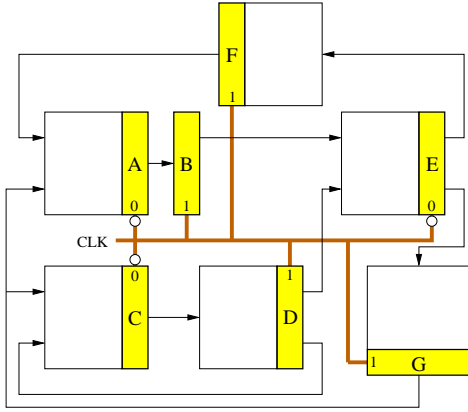
Fig. 2.   A synchronous circuit with a single global clock.

a) decouple local clocks for master and slave latches (in a D-flip-flop they are both derived from the same clock) and

b) optionally improve performance through retiming, i.e. by moving latches across combinational logic.

The conversion of a flip-flop-based circuit into a latch-based one is not specific to the de-synchronization framework only. It is known to give an improvement in performance for synchronous systems [4] and, for this reason, it has a value by itself.

2) *Generation of matched delays for the combinational logic* (denoted by rounded rectangles in Figure 1(b)).
   Each matched delay must be greater than or equal to the delay of the critical path of the corresponding combinational block. Each matched delay serves as a completion detector for the corresponding combinational block.

3) *Implementation of the local controllers*. This is the main topic of this section.

Figure 2 depicts a synchronous netlist after the conversion into latch-based design, possibly after applying retiming. The shadowed boxes represent latches, whereas the white boxes represent combinational logic. Latches must alternate their phases. Those with a label 0 (1) at the clock input represent the *even* (*odd*) latches. All latches are transparent when the control signal is high (CLK=0 for even and CLK=1 for odd). Data transfers must always occur from even (master) to odd (slave) latches and vice-versa. Usually, this latch-based scheme is implemented with two non-overlapping phases generated from the same clock.

Initially, only the latches corresponding to one of the phases store valid data. Without loss of generality, we will assume that these are the even latches. The odd latches store *bubbles*, in the argot of asynchronous circuits.

### B. The zero-delay model

This section presents a formal model for de-synchronization. The aim is to produce a set of distributed controllers that communicate locally with their neighbors and generate the control signals for the latches in such a way that the behavior of the system is preserved. For simplicity, we assume that all combinational blocks and latches have zero delay. Thus, the only important thing about the model is the sequence of events of the latch control signals. The impact of the data-path delays on the model will be discussed during the implementation of the model (Section VI).
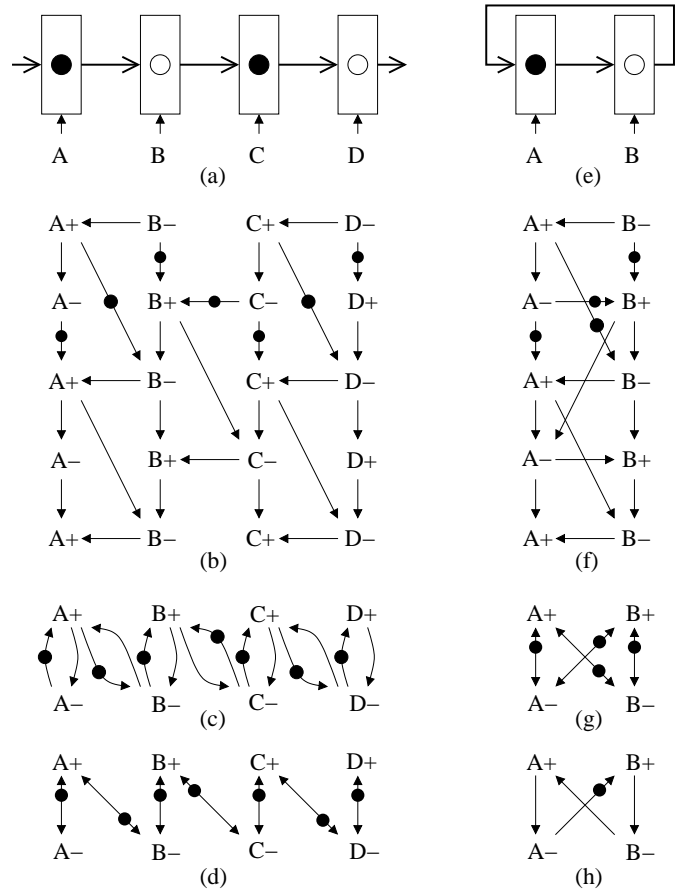


Fig. 3.   De-synchronization model for a linear pipeline and a ring.

For simplicity we will start by analyzing the behavior of a linear pipeline (see Figure 3(a)). The generalization for any arbitrary circuit will be discussed later. Black dots represent data tokens, whereas white dots represent bubbles. In the model, we assume that all latches become transparent when the control signal is high. The events $A+$ and $A-$ represent rising and falling transitions of the control signal $A$, respectively.

Figure 3(b) depicts a fragment of the unfolded marked graph representing the behavior of the latches. There are three types of arcs in this model (we only refer to those in the first stage of the pipeline):

- $A+ \rightarrow A- \rightarrow A+$, that simply denote that the rising and falling transitions of each signal must alternate.
- $B- \rightarrow A+$, that denotes the fact that for latch $A$ to read a new data token, $B$ must have completed the reading of the previous token coming from $A$. If this arc is not present, data overwriting can occur, or in other terms *hold constraints can be violated*.
- $A+ \rightarrow B-$, that denotes the fact that for latch $B$ to complete the reading of a data token coming from $A$ it must first wait for the data token to be stored in $A$. If this arc is not present, $B$ can "read a bubble" and a data token can be lost, or in other terms *setup constraints can be violated*.

The marking in Figure 3(b) represents a state in which all latch control signals are low and the events $B+$ and $D+$ are enabled, i.e. the latches $B$ and $D$ are ready to read the data tokens from $A$ and $C$, respectively.

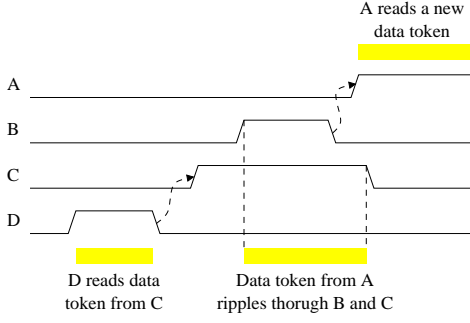Figure 3(c) shows the marked graph that derives from the unfolded

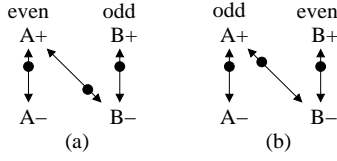Fig. 4. Timing diagram of the linear pipeline in Figure 3(a-d).



Fig. 5. Synchronization between latches: $A \rightarrow B$.

graph in Figure 3(b). A simplified notation is used in Figure 3(d) to represent the same graph, substituting each cycle $x \underset{\longleftarrow}{\overset{\bullet}{\longrightarrow}} y$ by a double arc $x \longleftrightarrow\!\!\!\bullet\, y$, where the token is located close to the enabled event in the cycle ($y$ in this example).

It is interesting to notice that the previous model is more aggressive than the classical one generating non-overlapping phases for latch-based designs. As an example, the following sequence can be fired in the model of Fig 3(a-d):

$$D+ \ D- \ C+ \ B+ \ B- \ A+ \ C- \ \cdots$$

After the events $\langle D+ \ D- \ C+ \ B+ \rangle$, a state in which $B = C = 1$ and $A = D = 0$ is reached, where the data token stored in $A$ is rippling through the latches $B$ and $C$. A timing diagram illustrating this sequence is shown in Figure 4.

But can this model be generalized beyond linear pipelines? Is it valid for any arbitrary netlist? Which properties does it have? We now show that this model can be extended to any arbitrary netlist, while preserving a property that makes the circuits observationally equivalent to their synchronous versions: *flow-equivalence* [10].

### C. General de-synchronization model

The general de-synchronization model is shown in Figure 5. For each communication between an even latch and an odd latch, the synchronization depicted in Figure 5(a) must be defined. If the communication is between odd and even, the one in Figure 5(b) must be defined. Note that the only difference is the initialization. The odd latches are always enabled in the initial state to read the data tokens from the even latches.

By abutting the previous synchronization models, it is possible to build the model for any arbitrary netlist, as shown in Figure 6. The marked graphs obtained by properly abutting the models in Figure 5 are called *circuit marked graphs* (CMG).

We will now show that a de-synchronized circuit mimics the behavior of its synchronous counterpart. For that, it must be proved that:

- a de-synchronized circuit never halts (*liveness*), and
- all computations performed by a de-synchronized circuit are the same as the ones performed by the synchronous counterpart (*flow-equivalence*).
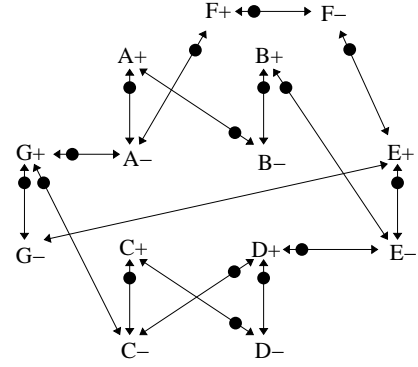


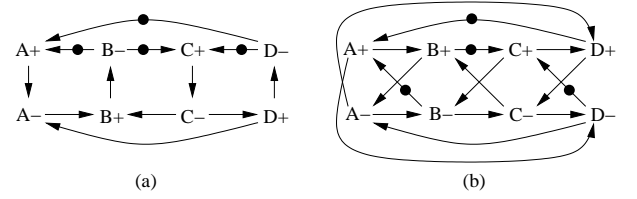Fig. 6. De-synchronization model for the circuit in Figure 2.



Fig. 7. Synchronization of a ring: (a) live model, (b) non-live model.

The remaining of this section is devoted to prove these two statements.

### D. Liveness

For the proof of liveness, the reader must bear in mind the meaning of the double arcs $x \longleftrightarrow\!\!\!\bullet\, y$, that represent $x \underset{\longleftarrow}{\overset{\bullet}{\longrightarrow}} y$.

*Theorem 4.1:* Any circuit marked graph is live.

*Proof:* By Theorem 3.1 it is enough to prove that there is no directed circuit in the CMG without any token. Rather than giving a formal proof, we merely give hints that can easily lead the reader to a complete proof. It is easy to see that there is no way to build an unmarked path longer than 3 arcs. As an example, let us try to find the longest unmarked path from $D+$ in the CMG of Figure 3(c). After building the path $D+ \rightarrow D- \rightarrow C+ \rightarrow C-$, it is not possible to extend it unless a marked arc is included, either $C- \longrightarrow\!\!\!\bullet\, C+$ or $C- \longrightarrow\!\!\!\bullet\, B+$. A case by case study leads to a complete proof. ∎

Liveness guarantees something crucial for the model: absence of deadlocks. This property does not hold automatically for every "reasonable" model. Figure 7 depicts two different de-synchronization models for a ring, that can be obtained by connecting the output of latch $D$ with the input of latch $A$ in Figure 3(a). Figure 7(a) depicts a non-overlapping model between adjacent latches, whereas Figure 7(b) uses a four-phase handshake with the sequence $A+ \ B+ \ A- \ B-$ for each pair of adjacent latches.

When building the protocol for a ring, the second model is not live due to the unmarked cycle:

$$A- \rightarrow B- \rightarrow C- \rightarrow D- \rightarrow A-.$$

One can easily understand that after firing events $A+$ and $C+$, the system enters a deadlock state. It is also easy to prove that this model is live for acyclic netlists.

The acid test of liveness for a handshake protocol consists of connecting two controllers back-to-back for a two-stage ring (see Figure 3(e)). Figure 3(f) depicts the unfolded behavior after including all causality constraints for the communication $A \rightarrow B$ and $B \rightarrow A$.

The folded behavior is shown in Figure 3(g), that can also be obtained by combining the synchronization models of Figure 5(a) and 5(b). Several arcs become redundant, thus deriving the simplified model shown in Figure 5(h).

Interestingly, the resulting protocol derived from the "aggressive" concurrent model is "naturally" transformed into one that is *non-overlapping, live and safe*. Note that a two-stage ring is typically derived from the implementation of a finite-state machine, in which the current state stored in a register is fed back to the same register after going through the combinational logic that calculates the next state. As an example, the handshake protocol between latches $C$ and $D$ in Figure 2 (see Figure 6 also) becomes non-overlapping.

### E. Flow-equivalence

In this section we will prove that a de-synchronized circuit mimics its synchronous counterpart. We will show that, for each latch, the value stored at the $i$-th pulse of the control signal is the same as the value stored at the $i$-th cycle of the synchronous circuit.

We first present some definitions that are relevant for synchronous circuits.

*Definition 4.1 (Synchronous behavior):* Given a block $A$ (combinational logic and latch), we call $F_A$ the logic function calculated by the combinational logic. We call $A_i$ the value stored in $A$'s latch after the $i$-th clock cycle. Let us call $E^1 \ldots E^p$ the (even) predecessor latches of an odd latch $O$, and $O^1 \ldots O^p$ the (odd) predecessor latches of an even latch $E$. Then,

- $O_i = F_O(E^1_{i-1}, \ldots, E^p_{i-1})$, and
- $E_i = F_E(O^1_i, \ldots, O^p_i)$

where all even blocks store a known initial value at cycle 0.

For the sake of simplicity here we model a *closed* circuit, i.e. one without primary inputs from the environment. The environment can be considered explicitly either by slightly changing the proofs, or by modeling it as a *non-deterministic function*. The latter mechanism also allows us to show how a de-synchronized circuit can be interfaced with a synchronous one (the environment), namely by driving its input handshake signals with the global clock and ignoring its output handshake signals. The latter must be shown to follow the correct protocol by means of appropriate timing assumptions.

The behavior of a synchronous circuit can be defined as the set of traces observable at the latches. If we call $E^1 \ldots E^n$ and $O^1 \ldots O^m$ the set of even and odd latches, respectively, the behavior of the circuit can be modeled by an infinite trace in which each element of the alphabet is an $(n + m)$-tuple of values:

| cycle | clk | trace | | | | | |
|-------|-----|-------|------|-----------|-------|------|-----------|
| initial | 0 | $E^1_0$ | $\ldots$ | $E^n_0$ | $O^1_0$ | $\ldots$ | $O^m_0$ |
| 1 | 1 | $E^1_0$ | $\ldots$ | $E^n_0$ | $O^1_1$ | $\ldots$ | $O^m_1$ |
| | 0 | $E^1_1$ | $\ldots$ | $E^n_1$ | $O^1_1$ | $\ldots$ | $O^m_1$ |
| 2 | 1 | $E^1_1$ | $\ldots$ | $E^n_1$ | $O^1_2$ | $\ldots$ | $O^m_2$ |
| $\vdots$ | $\vdots$ | | | $\vdots$ | | | |
| $i$ | 1 | $E^1_{i-1}$ | $\ldots$ | $E^n_{i-1}$ | $O^1_i$ | $\ldots$ | $O^m_i$ |
| | 0 | $E^1_i$ | $\ldots$ | $E^n_i$ | $O^1_i$ | $\ldots$ | $O^m_i$ |
| $i+1$ | 1 | $E^1_i$ | $\ldots$ | $E^n_i$ | $O^1_{i+1}$ | $\ldots$ | $O^m_{i+1}$ |

If we project the trace onto one of the latches, say $A$, we obtain a trace $A_0 A_1 \ldots A_i \ldots$, i.e. the sequence of values stored in latch $A$ at each cycle.

We now present a lemma that guarantees a good alternation of pulses between adjacent latches.

*Lemma 4.1 (Synchronic distance):* Let $(\Sigma, \rightarrow, M_0)$ be a CMG, $E$ and $O$ two adjacent blocks such that $E$ is even and $O$ is odd, and $\sigma$ a sequence fireable from $M_0$.
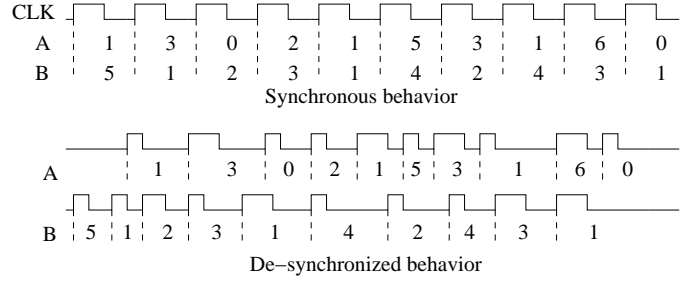


Fig. 8. Flow equivalence.

1) If $E$ transfers data to $O$ then

$$\overline{\sigma}(E+) \leq \overline{\sigma}(O-) \leq \overline{\sigma}(E+) + 1$$

2) If $O$ transfers data to $E$, then

$$\overline{\sigma}(E-) \leq \overline{\sigma}(O+) \leq \overline{\sigma}(E-) + 1$$

*Proof:* Both inequalities hold by the existence of the double arcs $E+ \leftrightarrow O-$ or $O+ \leftrightarrow E-$ that guarantee the alternation between both events. The initial marking is the one that makes the difference between the even-to-odd and odd-to-even connections. ∎

This lemma states that adjacent latches alternate their pulses correctly, which is crucial to preserve flow equivalence[1].

We now present the notion of *flow-equivalence* [10], which is related to that of *synchronous behavior* in [15], in terms of the projection of traces onto the latches of the circuit.

*Definition 4.2 (Flow equivalence):* Two circuits are flow-equivalent if

1) They have the same set of latches and
2) For each latch $A$, the projections of the traces onto $A$ are the same in both circuits.

Intuitively, two circuits are flow-equivalent if their behavior cannot be distinguished by observing the sequence of values stored at each latch. This observation is done individually for each latch and, thus, the relative order with which values are stored in different latches can change, as illustrated in Figure 8. The top diagram depicts the behavior of a synchronous system by showing the values stored in two latches, $A$ and $B$, at each clock cycle. The diagram at the bottom shows a possible de-synchronization. From the diagram one can deduce that latches $A$ and $B$ cannot be adjacent (see Lemma 4.1), since the synchronic distance of their pulses is sometimes greater than 1 (e.g. $B$ has received 5 pulses after having stored the values $\langle 5, 1, 2, 3, 1 \rangle$, while $A$ has only received two pulses storing $\langle 1, 3 \rangle$).

The following theorem is the main theoretical result of this paper.

*Theorem 4.2:* The de-synchronization model preserves flow-equivalence.

*Proof:* By induction on the length of the trace.

*Induction hypothesis:* For any latch $A$, flow-equivalence is preserved for the first $i - 1$ occurrences of $A-$ and until a marking is reached with the $i$-th occurrence of $A-$ enabled (see Figure 9). The marking of the arcs $E^k+ \rightarrow E^k- \rightarrow E^k+$ or $O^k+ \rightarrow O^k- \rightarrow O^k+$ is irrelevant for the hypothesis.

---

[1] A similar result was derived in [15] also based on Marked Graph Theory, using however a very different circuit structure and implementation philosophy.
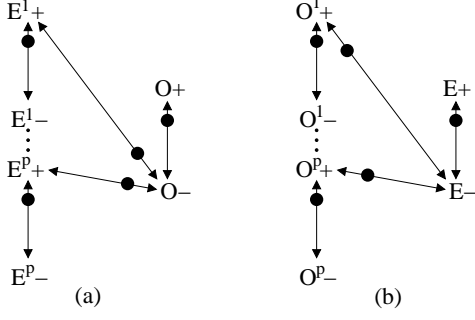
Fig. 9. Illustration of Theorem 4.2.

*Basis:* The induction hypothesis immediately holds for odd latches in the initial state (Figure 9(a)). For even latches (see Figure 9(b)), it holds after having fired $O^1+\ldots O^p+$ once from the initial state. This single firing preserves flow-equivalence since each latch $O^k$ receives the value

$$O_1^k = F_{O^k}(E_0^1, \ldots, E_0^m)$$

obtained from the initial value of $E^1, \ldots, E^m$, the (even) predecessor latches of $O^k$.

*Induction step (case O odd).* Since the $i$-th firing of $O-$ is enabled we know that each $E^k+$ transition has fired $i-1$ times (see Lemma 4.1) and, by the induction hypothesis, stores the value $E_{i-1}^k$. Therefore, the next firing of $O-$ will store the value

$$O_i = F_O(E_{i-1}^1, \ldots, E_{i-1}^p)$$

which preserves flow-equivalence. Moreover, the $i$-th firing of $E^k+$ will occur after $O$ has been closed, since the arc $O- \to E^k+$ forces that ordering. This guarantees that no data overwriting will occur on latch $O$.

*Induction step (case E even).* Since $E-$ has fired $i-1$ times, then $O^k+$ has fired $i$ times, according to Lemma 4.1. Since the $O^k$ latches are odd, they store the values $O_i^k$, by the induction hypothesis and the previous induction step for odd latches. The proof now is reduced to case of $O$ being even, in which:

$$O_i = F_O(E_i^1, \ldots, E_i^p)$$

This concludes the proof, since induction guarantees flow-equivalence for any latch $A$ and for any number firings of $A-$. ∎

## V. HANDSHAKE PROTOCOLS FOR DE-SYNCHRONIZATION

Section IV presented a model for de-synchronization that defines the causality relations among the latch control signals for a correct flow of data in the data-path. Now it is time to design the controllers that implement that behavior.

Several handshake protocols have been proposed in the literature for such purpose. The question is: are they suitable for a fully automatic de-synchronization approach? Is there any controller that manifests the concurrency of the de-synchronization model proposed in this paper?

We now review the classical four-phase micropipeline latch control circuits presented in [8]. For that, the specification of each controller (figures 5, 7 and 11 in [8]) has been projected onto the handshake
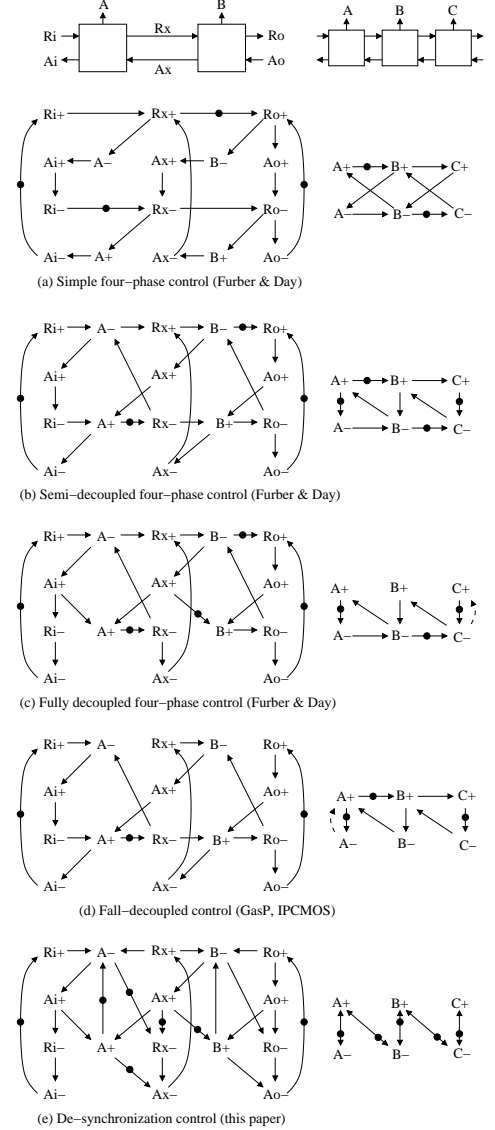


Fig. 10. Handshake protocols.

signals (*Ri, Ai, Ro, Ao*) and the latch control signal (*A*), thus abstracting away the behavior of the internal state signals[2]. The projection has been performed by preserving observational equivalence[3].

Figures 10(a-c) show the projections of the controllers from [8]. The leftmost part of the figure depicts the connection between an even and an odd controller generating the latch control signals $A$ and $B$ respectively. The rightmost part depicts only the projection on the latch control signals when three controllers are connected in a row.

The controllers from [8] show less concurrency than the de-synchronization model. For this reason, we also propose a new controller implementing the protocol with maximum concurrency proposed in this paper (Figure 10(e)). For completeness, a handshake decoupling the falling events of the control signals (*fall-decoupled*)

[2]In fact, $A$ is the signal preceding the buffer that feeds the latch control signal. The polarity of the signal has been changed to make the latch transparent when $A$ is high.

[3]For those users familiar with `petrify`, the projection can be obtained by hiding signals with the option `-hide`.
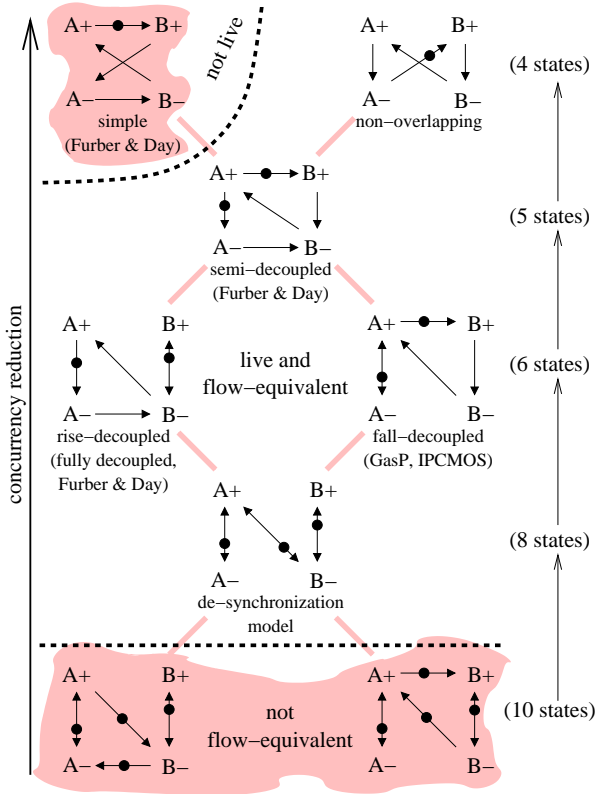
Fig. 11. Different degrees of concurrency in handshake protocols for de-synchronization.

is also described in Figure 10(d).

In all cases, it is crucial to properly define the initial state of each controller, which turns out to be different for the even and odd controllers. This is an important detail often missed in many papers describing asynchronous controllers.

The question now is: which ones of these controllers are suitable for de-synchronization? Instead of studying them one by one, we present a general study of four-phase protocols, illustrated in Figure 11. The figure describes a partial order defined by the degree of concurrency of different protocols. Each protocol has been annotated with the number of states of the corresponding state graph. The marked graphs in the figure do not contain redundant arcs.

An arc in the partial order indicates that one protocol can be obtained from the other by a local transformation (i.e. by moving the target of one of the arcs of the model). The arcs $A+ \leftrightarrow A-$ and $B+ \leftrightarrow B-$ cannot be removed for obvious reasons (they can only become redundant). For example, the semi-decoupled protocol (5 states) can be obtained from the rise-decoupled protocol (6 states) by changing the arc[4] $A+ \dashrightarrow B-$ to the arc $A+ \dashrightarrow B+$, thus reducing concurrency.

The model with 8 states, labeled as "de-synchronization model", corresponds to the most concurrent model presented earlier in this paper, for which liveness and flow-equivalence have been proved in Section IV. The other models are obtained by successive reductions or increases of concurrency.

The nomenclature *rise-* and *fall-decoupled* has been introduced to designate the protocols in which the rising or falling edges of the

[4]Note that this arc is not explicitly drawn in the picture because it is redundant.

pulses have been decoupled, respectively. The rise-decoupled protocol corresponds to the fully decoupled one proposed in [8].

We will now show that all models in Fig 11, except those that are shadowed, are suitable for de-synchronization. For that, we will analyze the properties of *liveness* and *flow-equivalence*.

### A. Liveness

We will focus on the liveness of the two sequential protocols: simple and non-overlapping. The proofs of liveness are similar to that of Theorem 4.1, and are left to the reader. Instead, we will illustrate intuitively the liveness proofs of these protocols by trying to pass the acid test mentioned above: building a two-stage ring (see Figure 3(e)).

It is clear that the non-overlapping protocol is live, given that this is the protocol to which the most concurrent model reduces when two controllers are connected back-to-back. However, the simple four-phase protocol does not pass this test:

$$A+ \to B+ \to A- \to B- \to \cdots$$

Any trace will eventually visit a state in which $A = B = 1$ which will produce data overwriting in the ring. To avoid data overwriting and deadlock, at least three latches in a ring are required, with only one data token circulating [17]. That would require substituting each flip-flop with three latches, which would in turn introduce a potentially substantial penalty in terms of both area and performance.

### B. Flow-equivalence

For the models with less than 8 states, obtained by concurrency reduction, the property of flow-equivalence holds automatically, since the traces produced by any of these models are always traces of the most concurrent model for de-synchronization, for which flow-equivalence has already been proved in Theorem 4.2.

On the other hand, the two models at the bottom, with 10 states each, have been obtained by increasing concurrency. The model on the left is obtained by changing the arc $B- \to A+$ to the arc $B- \to A-$. By observing Figure 3(b), this would correspond to shifting the arrow $A+ \leftarrow B-$ one step forward and converting it into $A- \leftarrow B-$. The reader can easily deduce that this transformation can produce data overwriting on latch $B$, since the value of $A$ can change without having stored the previous value in $B$. Therefore, this model does not preserve flow-equivalence.

The model on the right is obtained by changing the arc $A+ \to B-$ to the arc $A+ \to B+$. It can be easily shown that this model does not preserve flow-equivalence either.

This figure illustrates our belief (for which we do not have a formal proof) that the "de-synchronization model" is the maximally concurrent controller among those which preserve flow equivalence.

Both models at the bottom of Figure 11 are unsafe, since the arcs between events of the control signals for $A$ and $B$ can hold two tokens in some reachable markings.

The conclusion is that all handshake protocols in Figure 11, excepting the simple four-phase protocol presented in [8] and the two models at the bottom, are suitable for de-synchronization.

Only the specific implementation characteristics of each one (area, power, performance) determine the best choice.

### C. Hybrid de-synchronization approaches

An important aspect to notice is that de-synchronization can be performed by using different types of controllers, e.g. by choosing the most concurrent one for critical cycles within the latch-to-latch graph, and less concurrent ones for non-critical cycles. This would

reserve the most complex controllers only for those portions of the circuit where they are vital to improve the performance, and use cheaper ones where they do not affect the global performance.

*Property 5.1:* Any hybrid approach using any of the valid controllers shown in Figure 11 is valid for de-synchronization.

The proof of this property is simple and is briefly sketched now. Let us assume that we have a de-synchronized circuit with different types of controllers among all the latches. Liveness and flow-equivalence can be proved as follows:

- **Liveness.** Let us start from the sequential non-overlapping model for each controller. The obtained circuit is live, as proved in Theorem 4.1. By substituting each controller with the corresponding one in the hybrid approach, a new circuit with more concurrency is obtained. Therefore, the hybrid circuit is also live.
- **Flow-equivalence.** In this case we start from the most concurrent model for each controller. By substituting each controller with the corresponding one in the hybrid approach, a new circuit that is less concurrent is obtained. All the operations performed by this circuit are "flow-equivalent" to the ones of the most concurrent model.

Therefore, the flexibility of using any of the controllers for any latch in a de-synchronized circuit offers an avenue of possibilities to explore different trade-offs with respect to area, performance and power consumption.

### D. GasP, IPCMOS and MOUSETRAP

We briefly review some other existing protocols without analyzing the particular details of each implementation.

The behavior of the GasP [22] and IPCMOS protocols [20] corresponds to the fall-decoupled model. The arc $A+ \rightarrow B+$ is guaranteed by the logic of the controllers whereas the arc $B- \rightarrow A+$ is guaranteed by the timing assumptions used in the implementation (pulses are short). If the generated pulses at different stages have a similar width, then these protocols can be observably equivalent to the semi-decoupled or non-overlapping models.

MOUSETRAP [21] is also another protocol for asynchronous pipelines. It is extremely simple and efficient for acyclic pipelines, including fork and join structures. However, the causality relations of the abstract model are complex and cannot be represented by a marked graph. The model has causality arcs that go beyond neighboring stages. These extra arcs preclude the model to be used for cyclic structures. As an example, it is impossible to build a 2-stage MOUSETRAP ring that implements a live and flow-equivalent protocol. The states in which the two latch control signals are both low or high are deadlock states.

## VI. IMPLEMENTATION OF DE-SYNCHRONIZATION CONTROLLERS

The protocols described in Section V can be implemented in different ways using different design styles. In this section, the logic design of some controllers is presented.

### A. Semi-decoupled controller

We have chosen the semi-decoupled four-phase handshake protocol proposed by Furber and Day [8]. We present an implementation with static CMOS gates, while the original one was designed at transistor level. The reasons for the selection of this protocol with this particular design style are several:

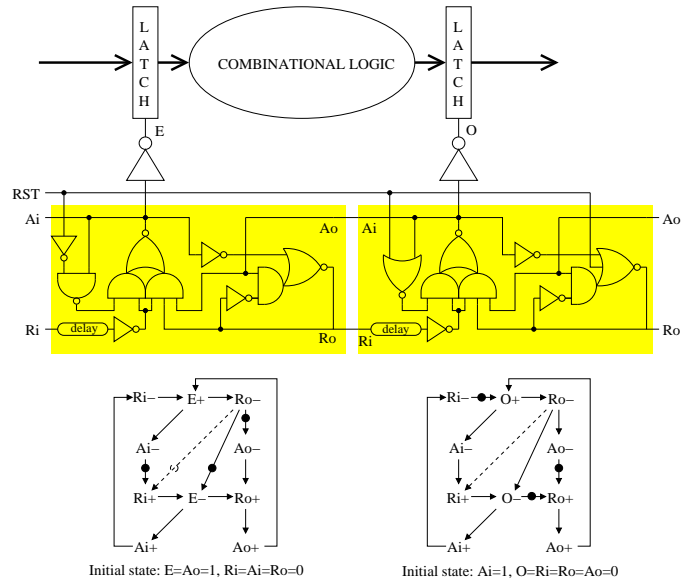- We pursue an approach suitable for semi-custom design using automatic physical layout tools.



Fig. 12. Implementation of semi-decoupled controllers for even ($E$) and odd ($O$) latches.

- The semi-decoupled protocol is a good trade-off between simplicity and performance.
- The pulse width of the latch control signals will be similar if all controllers are similar. Moreover, the depth of the data-path logic usually has a delay that can be overlapped with the controller's delay. Therefore, the arcs $A+ \rightarrow B+$ and $A- \rightarrow B-$ do not impose performance constraints in most cases.

In case of time-critical applications, other controllers can be used, including hybrid approaches combining protocols different from the ones shown in Figure 11.

Figure 12 depicts an implementation of a pair of controllers (even and odd) for a fragment of data-path. The figure also shows the marked graphs modeling the behavior of each controller. The only difference is the initial marking, that determines the reset logic (signal RST).

Resetting the controllers is crucial for a correct behavior. In this case, the even latches are transparent and the odd latches opaque in the initial state. With this strategy, only the odd latches must be reset in the data-path. The implementation also assumes a relative timing constraint (arc $Ro- \rightarrow Ri+$) that can be easily met with the actual design[5].

The controllers also include a *delay* that must match the delay of the combinational logic and the pulse width of the latch control signal.

Each latch control signal ($E$ and $O$) is produced by a buffer (tree) that drives all the latches. If all the buffer delays are similar, they can be neglected during timing analysis. Otherwise, they can be included in the matched delays, with a similar but slightly more complex analysis.

In particular, the delay of the sequence of events

$$E+ \rightarrow Ro/Ri- \rightarrow \boxed{\text{logic delay}} \rightarrow O+$$

is the one that must be matched with the delay of the combinational logic plus the delay of a latch. The event $Ro/Ri-$ corresponds to

[5]This assumption also allows us to simplify the implementation proposed in [8]: the equation for $A+$ becomes $R_{in}$ instead of $R_{in} \wedge \neg R_{out}$.
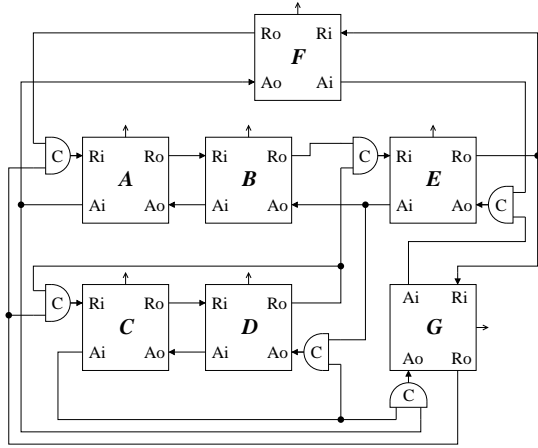
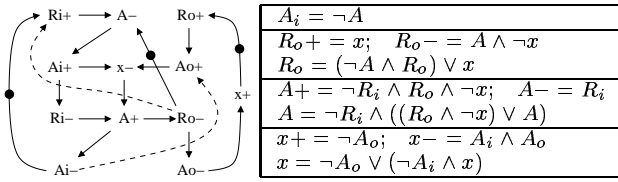Fig. 13. De-synchronized control for the netlist in Figure 2.



$$A_i = \neg A$$
$$R_o+ = x; \quad R_o- = A \wedge \neg x$$
$$R_o = (\neg A \wedge R_o) \vee x$$
$$A+ = \neg R_i \wedge R_o \wedge \neg x; \quad A- = R_i$$
$$A = \neg R_i \wedge ((R_o \wedge \neg x) \vee A)$$
$$x+ = \neg A_o; \quad x- = A_i \wedge A_o$$
$$x = \neg A_o \vee (\neg A_i \wedge x)$$

Fig. 14. Fall-decoupled controller.



$$A_i+ = \neg A \wedge x; \quad A_i- = \neg x$$
$$A_i = x \wedge (\neg A \vee A_i)$$
$$R_o = y$$
$$A+ = \neg y \wedge (A_i \vee \neg x); \quad A- = \neg A_i \wedge x$$
$$A = (A_i \vee \neg x) \wedge (A \vee \neg y)$$
$$x+ = \neg R_i; \quad x- = R_i \wedge R_o \wedge A \wedge y$$
$$x = R_i \wedge ((R_o \wedge A \wedge y) \vee x)$$
$$y+ = \neg A_o; \quad y- = A_o \wedge \neg A$$
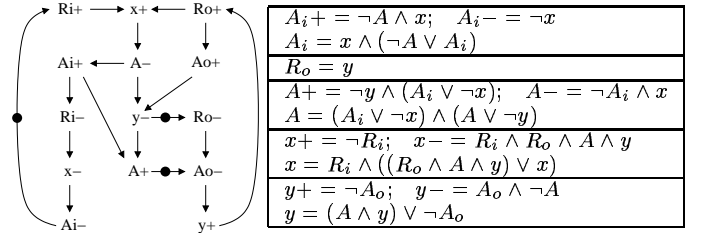$$y = (A \wedge y) \vee \neg A_o$$

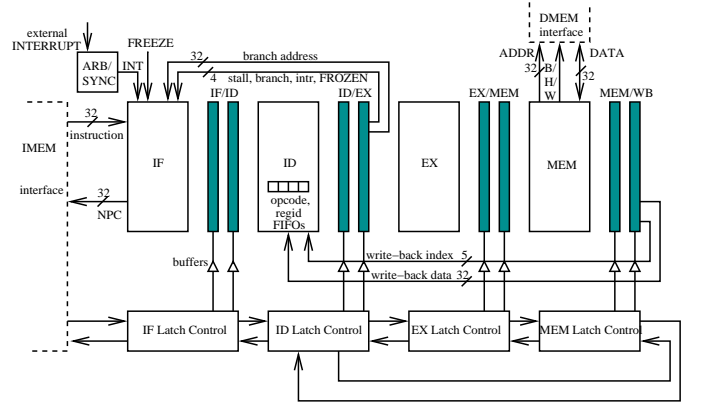Fig. 15. De-synchronization controller with maximum concurrency.



Fig. 16. De-synchronized DLX.

the falling transition of the signal $Ro/Ri$ between the $E$- and $O$-controllers. On the other hand, the delay of the sequence

$$O+ \rightarrow Ai/Ao- \rightarrow Ro/Ri+ \rightarrow \boxed{\text{pulse delay}} \rightarrow O-$$

is the one that must be matched with the minimum pulse width. It is interesting to note that both delays appear between transitions of the control signals of $Ri$ and $O$, and can be implemented with just one asymmetric delay.

The control can be generalized for multiple-input/multiple-output blocks. In that case the req/ack signals of the protocols must be implemented as a conjunction of those coming from the predecessor and successor controllers, by using C-elements. As an example, Figure 13 shows the de-synchronization control for the circuit depicted in Figure 2.

### B. Fall-decoupled and maximum concurrency controllers

For completeness, we present the design of the fall-decoupled controller and the controller with maximum concurrency, corresponding to the de-synchronization protocol presented in this paper.

Possible implementations are presented in Figures 14 and 15. The latch control signal is $A$ (transparent when $A = 1$), whereas $x$ and $y$ are internal signals required to properly encode the state space.

The Boolean equations that model the behavior are shown at the right of the marked graph. Two types of equations are provided: those for an implementation based on asymmetric C-elements (e.g. $A_i+$ and $A_i-$), and those for a complex-gate implementation (e.g. $A_i$). One can observe that the equations are simple and can be easily implemented with static CMOS gates. For brevity, no details are provided on where the matched delays must be inserted and on how to initialize the circuit for odd and even latches.

## VII. DE-SYNCHRONIZATION OF THE DLX MICROPROCESSOR

We present results on the application of de-synchronization to a DLX processor [11], using the semi-decoupled controllers depicted in

Fig. 12. The de-synchronized DLX consists of five architectural DLX pipeline stages, four of which actually correspond to circuit blocks (at the circuit level WB is merged with ID). Each block is controlled by its own latch controller. The arrows of the latch controllers correspond to their $P$ and $S$ signals, and illustrate the datapath dependencies. Stages ID, EX and MEM form a ring. ID is the heart of the processor containing the Register File and all hazard-detection logic and synchronizes stages IF and MEM. Thus, instructions leaving MEM (for WB) will synchronize with instructions coming from IF. Data hazard detection takes place by ID comparing the output register of instructions in other pipeline stages and their opcodes, and deciding on inserting the correct number of NOPs[6].

After the initial synthesis of each circuit block using latches (without retiming), the whole design is optimized incrementally to meet all timing requirements. Max-delay constraints between latches are used to ensure cycle time in the datapaths but the control blocks are untouched inside the synthesis tool. Then the gate-level netlist and matching timing constraints are placed and routed. We show the results of two flows, both of which use industrial-strength tools. The former uses the classical synthesis, placement and routing sequence. The latter adds one stage of placement-aware synthesis. The former is "safer", area-oriented and aimed equally at flip-flop-based and latch-based designs. The latter is more aggressive, timing-oriented and aimed at flip-flop-based designs. In all cases, post-route optimization is iterated until all timing violations are fixed. All layouts, except for de-synchronized flow 2, are flat.

Table I contrasts the characteristics of the synchronous flip-flop and latch-based designs, and of the de-synchronized latch-based design. The data are post-layout results based on gate-level simulations with back-annotation of extracted parasitics. Both clock period and

---

[6]This simple architecture does not include forwarding, but it makes manual insertion of the latch controllers easier. With the automation of the flow, we will be able to handle much more complex RTL and gate level designs.

|  |  | Sync. FF | Sync. Latch | De-Sync. Latch |
|---|---|---|---|---|
| | **Cycle Time** ($ns$) | 8.00 | 6.60 | 5.06 |
| Flow 1 | **Dyn. Pow.** ($mW$) | 44.75 | 53.88 | 48.94 |
| | **Area** ($mm^2$) | 2.66 | 2.22 | 2.43 |
| | **Cycle Time** ($ns$) | 3.60 | 4.10 | 3.86 |
| Flow 2 | **Dyn. Pow.** ($mW$) | 93.23 | 92.34 | 105.21 |
| | **Area** ($mm^2$) | 2.66 | 2.22 | 4.64 |

TABLE I
SYNCHRONOUS VS. DE-SYNCHRONIZED DLX.

waveform (in the synchronous cases) and controller matched delays (in the asynchronous case) were tuned in order to achieve the minimum cycle in each case. The reported cycle time of the de-synchronized design using flow 2 is actually the average between two alternating cycle times generated by the controllers, of 4.12 and 3.60 $ns$. If the circuit must be operated synchronously at its interfaces, then the worst-case value (4.12 $ns$) must be used. The area of this design is *much larger than the others because we had to implement it using hierarchical placement and routing*, in order to properly constrain the clock trees. This is just a temporary solution, and the results will certainly be improved when we will manage to implement it as a flat layout, as in all the other cases. It also explains the higher power consumption in this case.

One can see that all designs have approximately the same area, speed and power consumption. Differences between them can be attributed more to the different abilities of the two flows to optimize for different objectives (area vs. performance, latch vs. flip-flops), rather than to the synchronous or asynchronous implementation of each circuit. Other differences can be explained with the fact that the various implementation flows take slightly different constraints as input during the physical design phase (e.g., the de-synchronized circuit has smaller clock trees than the synchronous ones) and hence deliver slightly different results. No general conclusions can be drawn from a single example (even though we made our best effort to be fair and optimize every single implementation), other than the fact that de-synchronization is on a par with synchronous implementation in terms of area, performance and power.

## VIII. CONCLUSIONS

This paper presented a de-synchronization model that can be used to automatically substitute the clock network of a synchronous circuit by a set of asynchronous controllers.

To the best of our knowledge, this is the first successful attempt of delivering an automated design flow for asynchronous circuits that does not introduce significant penalties with respect to the corresponding synchronous designs. This opens wide opportunities of exploring the implementation space (both synchronous and asynchronous) within the very same set of industrial tools. This, we believe, is a valuable feature for a designer.

The suggested methodology can result in EMI improvements, shorter design cycles, and partitioning of the clock trees. Moreover, it provides the foundation for achieving power savings and other advantages of true asynchronous implementation through more fine-grained de-synchronization. We believe that our flow, while not providing all the advantages that asynchronous circuits promise, is a significant step towards spreading the use of asynchronous circuits among mainstream designers.

## REFERENCES

[1] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, Apr. 1997.

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

[3] I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92. IEEE Computer Society Press, Apr. 2000.

[4] D. Chinnery and K. Keutzer. Reducing the timing overhead. In *Closing the Gap between ASIC and Custom: Tools and Techniques for High-Performance ASIC design*, chapter 3. Kluwer Academic Publishers, 2002.

[5] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[6] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. From synchronous to asynchronous: an automatic approach. Submittted for publication.

[7] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. A concurrent model for de-synchronization. In *Proceedings of the International Workshop on Logic Synthesis*, pages 294–301, 2003.

[8] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[9] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, Oct. 1998.

[10] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, Apr. 2003.

[11] J. L. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[12] J. Kessels, A. Peeters, P. Wielage, and S.-J. Kim. Clock synchronization through handshake signalling. *Microprocessors and Microsystems*, 27(9):447–460, Oct. 2003.

[13] R. Kol and R. Ginosar. A doubly-latched asynchronous pipeline. In *Proc. International Conf. Computer Design (ICCD)*, pages 706–711, Oct. 1996.

[14] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, Apr. 2000.

[15] D. H. Linder and J. C. Harden. Phased logic: Supporting the synchronous design paradigm with delay-insensitive circuitry. *IEEE Transactions on Computers*, 45(9):1031–1044, Sept. 1996.

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

[17] D. E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.

[18] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, Apr. 1989.

[19] R. B. Reese and M. A. T. C. Traver. A coarse-grain phased logic CPU. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–13. IEEE Computer Society Press, May 2003.

[20] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at $3.3 - 4.5$GHz. In *International Solid State Circuits Conference*, pages 292–293, Feb. 2000.

[21] M. Singh and S. M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. International Conf. Computer Design (ICCD)*, pages 9–17, Nov. 2001.

[22] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, Mar. 2001.

[23] V. Varshavsky, V. Marakhovsky, and T.-A. Chu. Logical timing (global synchronization of asynchronous arrays). In *The First International Symposium on Parallel Algorithm/Architecture Synthesis*, pages 130–138, Aizu-Wakamatsu, Japan, Mar. 1995.