Abstract—This paper shows how asynchronous circuits can be derived from optimized synchronous circuits by replacing the clock distribution tree by a handshaking network. A concurrent model for de-synchronization is presented and behavioral properties are proved. A case study shows the applicability of the method and the potential benefits of de-synchronizing synchronous circuits.

I. INTRODUCTION

When the number of gates on a chip is quickly growing toward and beyond the one billion mark, keeping them all running at the beat of a single or a few rationally related clocks is becoming impossible. Nevertheless the synchronous methodology clearly dominates in today’s design community. The main problem that prevents a widespread adoption of asynchronous design technology is, in our opinion, the lack of a design methodology that can be picked up almost instantaneously and without risk by an experienced team, used to design synchronous logic with the standard RTL-to-GDSII flow based on synthesis, placement and routing.

This work gets its inspiration from a number of contributions from past work, each providing a key element to a unique novel methodology. The essential novelty of our contribution is that it does not require any knowledge of asynchronous design, 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 clock power, low EMI, automated global idling, and modularity. In the last section of this paper we also discuss how further design automation support can also provide some level of automated local idling and average case performance.

A. The Basic Idea

The essential idea of the de-synchronization approach is to start from a fully synchronous synthesized (or manually designed) circuit, and then replace directly the global clock network with a set of local handshaking circuits.

The key advantage of the de-synchronized circuit with respect to the corresponding synchronous circuit is that it no longer requires the clock tree. This tree must be designed with high-powered gates in order to ensure fast propagation with tightly controlled skews. Data from real designs shows that between 30 and 50% of the total dynamic power in a high-performance ASIC is dissipated in the clock network. Our method dramatically reduces it, since clocks are locally generated without dangerous skew effects. Moreover, EMI is also drastically reduced, since flip-flops or latches no longer switch simultaneously.

Margins due to transistor and interconnect variability can also be reduced, because matched delays can be laid out close to the logic they control. Thus the impact of temperature-related and manufacturing-related variations will be similar in both delay lines and combinational logic.

In this paper we also show that a partial elimination of the clock ensures synchronous interfacing with the rest of the world and improves yield, since it reduces the number of paths that are “almost critical” and can fail due to parametric variations.

B. Previous Work

If we look at the literature, we find several related ideas. Sutherland, in his Turing award lecture, proposed a scheme to generate local clocks for a synchronous latch-based datapath. His goal was to create a design theory for asynchronous designs, which has been exploited successfully by both manual designs [1] and CAD tools [2–4]. This 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 totally 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 [5]. This approach bears many similarities with ours, but in our opinion it attempts to go too far, and has an excessive overhead, even when used for large-granularity gates such as in FPGAs.

Similarly, Theseus Logic proposed a design flow [6] 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. This approach also has a high overhead.

The generation of local clocks from handshaking circuitry while ensuring the global “synchronicity” was first suggested in [7]. This work however focuses 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 [8]. This 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 [8] by using more general synchronization schemes and provides a theoretical foundation for de-synchronization approach by proving a behavioral and temporal equivalence between synchronous circuit and its de-synchronized counterpart. We also show experimentally that getting rid of the clock in synchronous design indeed materialises in significant power consumption savings.

In the rest of this paper we introduce the de-synchronization step as a purely logical transformation in Section III. Section IV-A discusses timing properties of de-synchronized designs and shows their timing compatibility to synchronous counterparts. A particular implementation of local controllers for de-synchronized designs is given in Section IV-C:implement. Finally Section VI presents an experimental study on de-synchronizing DES encryption core.

II. MARKED GRAPHS

Marked Graphs (MG) is the formalism used in this paper to model de-synchronization. They are a subclass of Petri nets [9] that can model decision-free concurrent systems.

Definition 2.1 (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 $M_0 : \rightarrow \rightarrow \rightarrow \mathbb{N}$ 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 $[M_0]$.

An example of marked graph is shown in Figure 3, where the events $A+$ and $A-$ represent the rising and falling transitions of signal $A$, respectively. Under the given initial marking (denoted by solid dots at arcs) two events are enabled $A+$ and $C+$. The sequence of events $A+, A-, C+, C-, B+,...$ is an example of a feasible sequence of the marked graph.

Definition 2.2 (Liveness): A marked graph is live if for any $M \in [M_0]$ 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 2.3 (Safeness): A marked graph is safe if no reachable marking from $M_0$ can assign more than one token to any arc.

Definition 2.4 (Event count in a sequence): Given a firing sequence $\sigma$ and an event $e \in \Sigma$, $\pi(e)$ denotes the number of times that event $e$ fires in $\sigma$.

The following results for marked graphs were proven in [10].

Theorem 2.1 (Liveness): A marked graph is live iff $M_0$ assigns at least one token on each directed circuit.

Theorem 2.2 (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 $[M_0]$, where $M(C)$ denotes the total number of tokens on $C$.

III. A ZERO-DELAY DE-SYNCHRONIZATION MODEL

The de-synchronization models presented in this section aim at the substitution of the global clock of a circuit by a set of asynchronous controllers that guarantee an equivalent behavior. The models assume 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)).

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 internally are often composed of master-slave latches. To perform de-synchronization this internal structure is revealed explicitly (see Figure 1(b)) to:

a) decouple local clocks for master and slave latches (in a D-flip-flop they are both derived from the same source $CLK$) and
b) 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 [11] for synchronous systems, and due to this it has a value by itself.

2) Generation of matched delays for 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. Matched delay serve as a completion detector for the corresponding combinational block.

3) Implementation of controllers for local clocks.

As illustrated in Figure 1(b), request signals from predecessor registers are delayed by at least the combinational
logic worst-case propagation time, in order to satisfy setup time constraints. The clock generation logic waits for the last of these requests to arrive, and for the last acknowledge, signaling that hold constraints are satisfied for all successor registers and that they are ready to proceed. Then a clock edge is generated, signaled to both predecessors and successors, and the handshake cycle is closed, waiting to produce a new edge.

Figure 2 depicts a synchronous netlist after conversion into latch-based design, possibly after applying the “safe” retiming mentioned above. The shadowed boxes represent latches, whereas the white boxes represent combinational logic. Latches must alternate their phases. Those latches with a label 0 (1) at the clock input represent the even (odd) latches, transparent when the clock is low (high). Data transfers must always occur from even (master) to odd (slave) latches and vice-versa.

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.

This section presents two models for de-synchronization. The first one is presented for its simplicity, and can be considered as a restricted case of the second. The formal proofs of correctness will be only presented for the second model.

For simplicity, we will also 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. A timed model, and its performance, will be presented in Section IV.

A. Non-overlapping de-synchronization model

In the synchronous methodology, latched designs are normally clocked by two-phase non-overlapping clocks. The first de-synchronization model is a direct implementation of this scheme.

A timing diagram and the corresponding marked graph for a simple pipeline is depicted in Fig. 3. The latches are transparent when the control signal is high. Initially, only half of the latches contain data (D). Data items flow in such a way that a latch never captures a new item before the successor latch has captured the previous one.

Since the pulses for adjacent latches are non-overlapping, data overwriting can never occur. However, contrary to common belief, the need for a non-overlapping scheme to avoid races could be relaxed, as discussed below.

B. Overlapping de-synchronization model

Figure 4 shows another model that allows clock pulses of adjacent latches to overlap. This model is based on the observation that a data item can ripple through more than one latch as long as the previous values stored in those rippling latches have already been captured by the successor latches. As an example, event B+ can fire as soon as data is available in A (arc A+ → B+) and the previous data in C (arc A− → B−). The formal model for this de-synchronization is depicted in the marked graph of Fig 4. The arc A+ → A+ is included to model the alternation of A+ and A− at one end of the pipeline. This arc is redundant for the other events.

It is easy to understand that the model of Fig. 3 can be obtained by reducing the concurrency of the model in Fig. 4. In particular, arcs A+ → B+, B+ → C+, ..., must be converted into arcs A− → B+, B− → C+, ... . This conversion unfortunately precludes the overlapping of pulses.

However, can we extend these models beyond linear pipelines? Are they valid for any arbitrary netlist? Which properties do these models have? We now show that these models can be extended to any arbitrary netlist, while preserving a property that makes the circuits observationally equivalent to
their synchronous versions: flow equivalence [12].

C. General de-synchronization model

We now generalize the overlapping model and prove its correctness. A similar analysis, not discussed in this paper, can be done for the non-overlapping model.

The de-synchronization model is shown in Fig. 5. It models the communication of data from block A to block B.

The procedure to build the de-synchronization marked graph model for a synchronous circuit is as follows:

- For each block A, define events A+ and A− and the arcs A+ → A− and A− → A+. Put a token on A− → A+.
- For each pair of blocks A and B, such that data are transferred from A to B, define the arcs A+ → B+ and B− → A+. If A is even, put a token on A+ → B+, otherwise put a token on B− → A+.

The previous procedure creates a marked graph of a certain class that we will call circuit marked graph (CMG). In this model, the arcs A− → A+ will be usually redundant.¹

Figure 6 depicts the de-synchronization marked graph for the circuit in Fig. 2. It is interesting to notice that, in those cases in which two latches create a cycle (e.g. C and D in Fig. 2), the model implicitly generates a non-overlapping protocol for the pulses, which is the only correct one in this case. For example, the pulses for C and D commit to the following sequence (see Fig. 6):

\[ D+ → D− → C+ → C− → D+ → \cdots \]

D. Properties of the de-synchronization model

We now discuss several properties of the model. The proofs of the theorems are presented in the appendix.

Theorem 3.1: Any circuit marked graph is safe.

Proof: See appendix.

¹We include these arcs for the sake of clarity of the model. Only in those cases in which a block has no incoming data, the arc will not be redundant.

Although this property is not essential to prove the main theorem of this section, it is interesting to notice that safeness intuitively guarantees that no data overwriting will occur.

Theorem 3.2: Any circuit marked graph \((G, M_0)\) is live.

Proof: See appendix.

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 Fig. 3. Figure 7(a) depicts the non-overlapping model presented in Section III-A, obtained by adding the arcs \(D− → A+ \) and \(D+ → A−\) to the marked graph of Fig. 3.

A different model is presented in Fig. 7(b). In this model, the events of two adjacent latches, say A and B, commit to the following handshaking protocol: \(A+ → B− → A− → B+ \cdots\). When building the protocol for a ring, the model is not live, due to the unmarked cycle:

\[ A− → B− → C− → D− → A− \]

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.

At this point, it is important to emphasize that the de-synchronization model presented in this paper has self-resetting pulses, i.e. the only causality arc that precedes an event \(X−\) is the corresponding \(X+\). This spontaneous “return-to-zero” guarantees liveness for any netlist, even cyclic.

Theorem 3.3 (Synchronic distance): Let \((\Sigma, r, M_0)\) be a CMG, A and B two blocks such that A transfers data to B, and \(\sigma\) a sequence firable from \(M_0\).

1) If A is even and B is odd, then

\[ \sigma(A+) ≤ \sigma(B+) ≤ \sigma(A+) + 1 \]

2) If A is odd and B is even, then

\[ \sigma(B+) ≤ \sigma(A+) ≤ \sigma(B+) + 1 \]

Proof: See appendix.

This theorem states that adjacent latches alternate their pulses correctly, which is crucial to preserve flow equivalence. We now present the main result of this paper.

E. Flow equivalence

The de-synchronization model previously presented enables data to flow across an asynchronous circuit. But is this data flow equivalent to the behavior of the synchronous circuit? To prove this equivalence, we first must define in what kind of
equivalence we are interested. For that, we first present some preliminary definitions for synchronous circuits.

**Definition 3.1 (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 \( P^1 \ldots P^n \) the predecessor blocks of \( A \).

- If \( A \) is odd, then \( A_i = F_A(P^1_{i-1}, \ldots , P^n_{i-1}) \)
- If \( A \) is even, then \( A_i = F_A(P^1_{i-1}, \ldots , P^n_{i}) \)

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

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:

<table>
<thead>
<tr>
<th>cycle</th>
<th>( \text{clk} )</th>
<th>trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>0</td>
<td>( E^1_0 \ldots E^n_0 \ldots O^1_0 \ldots O^m_0 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( E^1_1 \ldots E^n_1 \ldots O^1_1 \ldots O^m_1 )</td>
</tr>
<tr>
<td>0</td>
<td>( E^1_0 \ldots E^n_1 \ldots O^1_0 \ldots O^m_1 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>( E^1_2 \ldots E^n_1 \ldots O^1_2 \ldots O^m_2 )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( i )</td>
<td>1</td>
<td>( E^1_{i-1} \ldots E^n_{i-1} \ldots O^1_i \ldots O^m_i )</td>
</tr>
<tr>
<td>0</td>
<td>( E^1_0 \ldots E^n_i \ldots O^1_0 \ldots O^m_i )</td>
<td></td>
</tr>
<tr>
<td>( i+1 )</td>
<td>1</td>
<td>( E^1_{i+1} \ldots E^n_i \ldots O^1_{i+1} \ldots O^m_{i+1} )</td>
</tr>
</tbody>
</table>

In fact, it is enough to observe the behavior in one of the clock phases (e.g. \( \text{clk} = 0 \)) to completely define the behavior of the circuit.

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 the notion of flow-equivalence [12], which is related to that of synchronous behavior in [5], in terms of the projection of traces onto the latches of the circuit.

**Definition 3.2 (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 \( A \) is 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 at which values are stored in different latches can change.

**Theorem 3.4:** The de-synchronization model preserves flow-equivalence.

**Proof:** See appendix.

**Theorem 3.4** is the main theoretical result of this paper. Figure 8 illustrates the notion of flow equivalence. 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 Theorem 3.3), since the synchronic distance of their pulses is sometimes greater than

\[ T_T \geq T_C + T_L - \min(T_A, 0) \quad (1) \]

Constraint (1) indicates that the cycle time of a local clock (measured as a delay \( T_T \) between two rising edges of \( L_A \)), must be greater than the delay of the computational block \( (T_C) \) plus the latch controller delay \( (T_L) \) plus the penalty for opening the latch late, when \( T_A \) is negative. The control overhead in this scheme is reduced to a single delay \( T_L \) because control handshake overlaps with the computation cycle due to the early rising of the local clock. The constraint assumes that the depth of combinational logic is sufficiently
A B C

counterpart without visible changes. This makes the suggested design methodology modular and compositional.

A synchronous flip-flop-based circuit, the cycle time $T_S$ is bounded by [11]:

$$T_S \geq T_C + T_{setup} + T_{skew} + T_{CQ}$$

where $T_C$, $T_{setup}$, $T_{skew}$ and $T_{CQ}$ are maximum combinational logic, setup, skew and clock-to-output times respectively.

Comparing inequalities (1) and (2) and bearing in mind that due to retiming the maximal computation time in a de-synchronized circuit can only be reduced, one can conclude that under reasonable timing assumptions the cycle time of de-synchronized circuit $T_F$ should be smaller than the cycle time $T_S$ of the corresponding synchronous design.

There is a small caveat in the above statement. The notion of a cycle time is well defined only for a circuit with a periodic clock. In a de-synchronized system the separation time between adjacent rising edges of the same local clock might change during functioning (see Figure 8 e.g.). Therefore when talking about de-synchronized and synchronous systems one has to relate the perfect periodic behavior of one of them to a non-periodic one of another, which seems to be problematic.

The following properties provide a basis for relating these two systems in a sound way. Informally they show that latches that belong to critical computational paths of a de-synchronized system have well-defined constant cycle time while the rest of the latches operates in plesiochronous mode [13], in which their local clocks have transitions nominally at the same rate, with bounded time offsets.

Property 4.1: If in a de-synchronized circuit the computation delay $T_C$ is the same for every combinational block, then the separation time between adjacent rising edges of every local clock is also the same and equals $T_F$.

The proof is trivial because a perfectly balanced de-synchronized system behaves like a synchronous one with all local clocks paced at the same rate.

The observation that the rising transition of the local clock of any odd latch $i$-th happens at time $(i-1) \cdot T_F$ immediately follows from Property 4.1. A similar relationship can be defined for the clocks of even latches by adding a constant phase shift $T_{ph}$ to time stamps $(i-1) \cdot T_F$. Without losing generality of a timing analysis, one can limit the consideration to one type of latches only (odd e.g.).

Property 4.2: In any de-synchronized circuit the $i$-th rising transition of a local clock of odd latch cannot appear later than $(i-1) \cdot T_F$.

Proof: See appendix.

Let us call a latch critical if the delay of a combinational block connected to its output is equal to the maximal computational delay $T_C$. From Properties 4.1 and 4.2 follows that the separation time between any rising edges of clocks for critical latches is constant and equal to $T_F$. The synchronic distance between adjacent latches does not exceed 1 (Theorem 3.3). Therefore after at most one cycle latches adjacent to a critical latch must adapt their cycle time to $T_F$ (after one cycle they are paced by a critical latch). Pushing these arguments further implies that in a connected de-synchronized system system any latch sooner of later settles to the cycle time $T_F$. This shows that the behavior of a de-synchronized circuit has a well-defined periodicity, similar to that of a synchronous one, paced by a common clock.
Embedding of a de-synchronized circuit with clock cycle $T_T$ into a synchronous environment with a clock cycle $T_S$: $T_S \geq T_T$ results in the latches at the asynchronous/synchronous boundary becoming critical, since they are paced by external clock $T_S$. This makes de-synchronized and synchronous systems compatible in terms of timing, because their external timed behavior is the same.

V. AN IMPLEMENTATION OF THE MODEL

Figure 10 depicts an STG describing a possible implementation of the latch controller. For simplicity, a block with two predecessors ($P1$ and $P2$) and two successors ($S1$ and $S2$) has been considered. The output signal is $L$. The events $P+$ and $L+$ are separated by a delay greater than the delay of the combinational block. On the other hand, events $P-$ and $L-$ determine the pulse width. Therefore, $P$ and $L$ are separated by an asymmetric delay (faster falling than rising).

A possible implementation of the behavior is depicted in Fig. 11.

The dashed arcs represent timing assumptions. The following assumptions are required:

- The delay of event $x_i-$ is shorter than the delay of $P_i-$. This is guaranteed by having a pulse width for $P_i$ longer than the delay of the gate implementing $x_i$.
- A similar assumption is required for $y_i-$ with regard to the pulse width of $S_i$.
- The delay of $y_i+$ is shorter than the delay of $S_i+$. This is a realistic assumption, since $y_i$ is implemented by one small gate, whereas the delay from $L+$ to $S_i+$ is determined by the delay of the combinational block (equivalent to the delay from $P_i+$ to $L+$ in the successor block).
- $P_i-$ occurs before $L-$. This is realistic if we assume that all controllers in the circuit are designed to generate similar pulse widths.
- $L-$ must occur before $S_i-$. Again, this is realistic if all pulse widths are similar. In particular, the pulse for $L$ starts before the pulse for $S_i$, since there is a causality relation $L+ \rightarrow S_i+$.

Finally, the controller can guarantee a correct synchronization only if inequality (1) holds. The sequence of events that determines the corresponding delays is the following:

$$P_i+ \rightarrow x_i- \rightarrow P+ \rightarrow \text{delay} \uparrow \rightarrow L+ \rightarrow \text{delay} \downarrow \rightarrow L-$$

VI. DE-SYNCHRONIZATION CASE STUDY

We present results on the application of de-synchronization on a pipelined DES encryption core. We demonstrate that despite the fact this design contains thin logic between registers, our approach still manages to hide control overhead and achieves comparable performance at lower power.

A high-throughput DES core is essentially a 16-stage pipeline, where each stage implements a single iteration of the DES algorithm. The algorithm operates on a 64-bit data stream and 64-bit keys and consists of permutations, shifts and a limited amount of logic. Thus, the depth of each of these stages is small.

We first implemented a synchronous, edge-triggered flip-flop design realising the 16-stage DES design in the 0.18µm VST-UMC standard-cell technology library. We have actually compared our synchronous implementation with available synchronous Open DES cores (from www.opencores.org) and verified that it has indeed similar performance. We then employed the method of de-synchronization in order to derive a de-synchronized dual-latch design.

As this was a standard-cell library implementation we had to derive standard-cell realisations of the circuits presented in Section V. First, we used Petrify to obtain a standard-cell circuit for our latch controller designs in a given library. After that we described the circuits in standard-cell Verilog and from this step onwards we were able to follow a standard EDA tool flow, i.e. used Synopsys Design Compiler...
for Static Timing Analysis (STA) and (limited amount of) circuit optimization and NCVerilog for circuit simulation. We generated asymmetric delay elements as Verilog standard-cells automatically using a simple Perl script which chained a user-specified number of gates (in the form shown in Figure 12) and then performed STA on that delay element in Synopsys to verify its timing and fine-tune it.

Table I contrasts the characteristics of the two designs. This data are post-synthesis, pre-layout results based on gate-level simulations.

The cycle time is the time it takes to perform a single iteration of the DES algorithm. A total of sixteen iterations is required to produce the 64-bit result, i.e. the latency value shown in the table. The power consumption of the DES designs was measured by performing switching activity annotation of the circuit during simulation. The area figures are standard-cell totals. The area of the synchronous version does not include the area required by its clock tree, however the area of the asynchronous includes all the necessary buffers for register latching.

As can be seen by these figures, the de-synchronized design, despite an area increase of approximately 22%, presents only a very slight difference in cycle time and a power improvement of slightly over 12%.

We were able to exploit both the fact that latches present a smaller propagation delay and also that latches present a smaller delay from inputs to outputs, than from their enable signal to outputs. Thus, in order to hide control overhead we firstly removed the overhead of internal controller delay from the matched delay elements and also reduced the delay of these elements so as the enable pulse of the latches to arrive before their last inputs, thus achieving a faster latch response.

Table II shows the area breakdown of the de-synchronized DES in terms of asynchronous control, delay elements, registers and combinational logic. These numbers demonstrate that logic is thin as stated earlier. In fact most of the area overhead comes from using two latches instead of a single flip-flop. The register area of the asynchronous design is approximately 218560.

These results demonstrate the potential of this approach even on a simple pipelined design with thin logic. We believe that further experiments will demonstrate that the de-synchronization of more complex datapaths will be able to exhibit more significant power and potentially better performance than that of synchronous counterparts.

VII. 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. This results in power and EMI improvements, shortens the design cycle, and can allow one to measure more easily the performance of each manufactured circuit.

We believe that these techniques, while not providing all the advantages that asynchronous circuits promise, are a significant step toward spreading the use of asynchronous circuits among mainstream designers.

REFERENCES

Proof of Theorem 3.1

According to Theorem 2.3, it is enough to prove that every arc belongs to a directed circuit \( C \) with \( M_0(C) = 1 \). In any CMG we have four types of arcs, denoted by \( e_1 - e_4 \) in Fig. 5. Arcs \( e_1 \) and \( e_2 \) are a circuit with exactly one token on \( e_1 \). Assume that \( A \) and \( B \) are the source and target blocks for arcs \( e_3 \) and \( e_4 \) (see Fig. 5). Both arcs belong to a directed circuit with one token (\( A+ \rightarrow B+ \rightarrow B- \rightarrow A+ \)) in which either \( e_3 \) or \( e_4 \) will be marked, depending on whether \( A \) is even or odd, respectively. Therefore all arcs of a CMG belong to a directed circuit with exactly one token.

Proof of Theorem 3.2

By Theorem 2.1 it is enough to prove that there is no directed circuit in the CMG without any token. For that, we will try to build an unmarked directed circuit and we will show that it is not possible. Since all arcs of type \( e_1 \) are marked in \( M_0 \), we can ignore them. If we try to build a circuit that starts with an arc of type \( e_2 \), we will find the following possibilities, all of them ending up by crossing a marked arc \((E \ and \ O \ represent \ signals \ of \ an \ even \ and \ odd \ block, \ respectively, \ and \ the \ \bullet \ denotes \ a \ marked \ arc).$

(a) \( E+ \xrightarrow{e_2} E- \xrightarrow{e_2} O+ \)
(b) \( O+ \xrightarrow{e_2} O- \xrightarrow{e_3} E+ \xrightarrow{e_4} O+ \)
(c) \( O+ \xrightarrow{e_3} O- \xrightarrow{e_4} E+ \xrightarrow{e_3} E- \xrightarrow{e_4} O+ \)

If we try to build a circuit that starts with an arc \( O- \xrightarrow{e_4} E+ \), we will end up by exploring the cases (b) and (c). Finally, if we try to build a circuit that starts with an arc of type \( e_3 \), we have the following cases:

(d) \( O+ \xrightarrow{e_4} E+ \xrightarrow{e_3} O+ \)
(e) \( O+ \xrightarrow{e_3} E+ \xrightarrow{e_4} E- \rightarrow \xrightarrow{e_3} O+ \)

thus always crossing an arc with a token. Hence, we conclude there is no directed circuit without any token.

Proof of Theorem 3.3

In case \( A \) is even and \( B \) is odd, the circuit \( A+ \xrightarrow{\bullet} B+ \rightarrow B- \rightarrow A+ \) guarantees that these three events alternate in this order and, therefore, the inequality holds.

In case \( A \) is odd and \( B \) is even, the alternation is guaranteed by the same circuit, but initially marked on \( B- \xrightarrow{\bullet} A+ \).

Proof of Theorem 3.4

Let us call \( P_1^1 \ldots P_n^1 \) the predecessor latches of \( A \). The proof will be done 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 Fig. 13(a)). The marking of the arcs \( P^k+ \rightarrow P^k- \rightarrow \)

\( P^k+ \) is irrelevant for the hypothesis.

\textbf{Basis:} The induction hypothesis immediately holds for odd latches in the initial state. For even latches (see Fig. 13(b)), it holds after having fired \( P_1^1 + \ldots P_n^1 \) once from the initial state. This single firing preserves flow-equivalence since each latch \( P^k \) receives the value

\[ P_1^k = F_{P^k}(Q_0^1, \ldots, Q_m^m) \]

obtained from the initial value of \( Q^1, \ldots, Q^m \) (the predecessor latches of \( P^k \)).

\textbf{Induction step (case \( A \) odd).} Since the \( i \)-th firing of \( A+ \) is enabled we know that each \( P^k+ \) transition has fired \( i-1 \) times (see Theorem 3.3) and, by the induction hypothesis, stores the value \( P_{i-1}^k \). Therefore, the next firing of \( A+ \) will store the value

\[ A_i = F_A(P_{i-1}^1, \ldots, P_i^n) \]

which preserves flow-equivalence. Moreover, the \( i \)-th firing of \( P^k+ \) will occur after \( A \) has been closed, since the arc \( A- \rightarrow P^k+ \) forces that ordering. This guarantees that no data overwriting will occur on latch \( A \). Finally, the controller will move towards the marking with tokens in \( P^k+ \rightarrow A+ \) without opening the latch, thus reaching the same conditions of the induction hypothesis, but now for cycle \( i \).

\textbf{Induction step (case \( A \) even).} Since \( A+ \) has fired \( i-1 \) times, then \( P^k+ \) has fired \( i \) times, according to Theorem 3.3. Since the \( P^k \) latches are odd, they store the values \( P_{i}^k \), by the induction hypothesis and the previous induction step for odd latches. The proof now is reduced to case of \( A \) being even, in which:

\[ A_i = F_A(P_{i}^1, \ldots, P_i^n) \]

This concludes the proof, since induction guarantees flow-equivalence for any latch \( A \) and for any number firings of \( A+ \).

Proof of Property 4.2

Estimation of the firing time of \( i \)-th instance \( A_i \) of event \( A \) in a marked graph \( G \) is reduced to the following procedure [14]:

1) Annotate each edge of a graph with the corresponding delay.
2) Construct an unfolding of a graph.
3) Find the longest path from a set of events enabled initially (fireable at time \( t = 0 \)) to \( A_i \).

From Property 4.1 follows that for a well-balanced desynchronized circuit the length of the longest path to the \( i \)-th rising event at any odd latch is \( (i - 1) \cdot T_R \). For an arbitrary circuit the weight of edges in \( G \) could only be reduced from their worst case values. This immediately implies that none of the odd latches could have \( i \)-th rising transition happening later than \((i - 1) \cdot T_R \).