Sleep Convention Logic Isochronic Fork: an Analysis

Ricardo A. Guazzelli
Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – RS – Brazil
ricardo.guazzelli@acad.pucrs.br

Walter Lau Neto
Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – RS – Brazil
walter.lau@acad.pucrs.br

Matheus T. Moreira
Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – RS – Brazil
matheus.moreira@acad.pucrs.br

Ney L. V. Calazans
Pontifícia Universidade Católica do Rio Grande do Sul – Porto Alegre – RS – Brazil
ney.calazans@pucrs.br

ABSTRACT

Asynchronous quasi-delay-insensitive (QDI) circuits are a promising solution for coping with aggressive process variations faced by modern technologies, as they can gracefully accommodate gate and wire delay variations. Furthermore, due to their inherent robustness, such circuits are also promising for deep voltage scaling applications, where delays are orders of magnitude larger. However, QDI design has an Achilles heel, which is its associated area and power overhead penalties. These can hamper the adoption of this kind of design in current and future technologies. A recently proposed asynchronous circuit design template, the Sleep Convention Logic (SCL), does reduce these overheads significantly. SCL is an enhancement of the Null Convention Logic, a well-known asynchronous circuit QDI design template. This paper analyzes the architecture of circuits based on SCL, identifies and models associated timing constraints that were not described before. The paper also shows experimentally that respecting such constraints is fundamental to guarantee correct operation of these circuits, especially under low voltage supplies.

CCS CONCEPTS

• **Hardware → Asynchronous circuits**;

KEYWORDS

asynchronous circuits, asynchronous design, quasi-delay-insensitive, QDI, NCL, SCL

ACM Reference format:
Ricardo A. Guazzelli, Walter Lau Neto, Matheus T. Moreira, and Ney L. V. Calazans. 2017. Sleep Convention Logic Isochronic Fork: an Analysis. In *Proceedings of SBCCI '17, Fortaleza, Ceará, Brazil, August 28-September 01, 2017,* 7 pages.
https://doi.org/10.1145/3109984.3110022

SBCCI '17, August 28-September 01, 2017, Fortaleza, Ceará, Brazil

ACM ISBN 978-1-4503-5106-5/17/08…$15.00
https://doi.org/10.1145/3109984.3110022

1 INTRODUCTION

Asynchronous circuit design is becoming an increasingly important topic for the VLSI research community. These circuits can tolerate process, voltage and temperature variations more easily than their synchronous counterparts [1, 10]. In fact, by avoiding the use of a global clock signal, it is possible to reduce the number of timing assumptions required to fulfill at design time, or even eliminate some of these assumptions altogether. This leads to more efficiency in coping with the growing amount of timing uncertainties arising in modern technologies. These circuits rely on the use of local handshake protocols for control and sequencing of events [1]. Therefore, asynchronous logic is only active *when* and *where* required. In other words, parts of the circuit can be quiescent while data flows only through the path that is required to be active, potentially providing power savings and making it easier to cope with contemporary power efficiency requirements.

Differently from synchronous circuits, asynchronous circuits are implemented using one of many available distinct design templates. However, Martin and Nyström [10] cite that practical circuits most often employ 1-of-n, 4-phase, quasi-delay-insensitive (QDI) templates. This is because these allow easier design and timing closure and are more robust. Moreover, due to its robustness to PVT variations, QDI design has been introduced as an alternative to cope with the side effects of aggressive techniques such as voltage scaling and subthreshold operation, deemed for extremely low power applications. Because QDI design often requires special cells, distinct from those available in commercial standard cell libraries, and because such templates are not naturally supported by conventional EDA tools, QDI design is far from reaching wide adoption today.

Among the several QDI templates proposed to date, a recent one, called Sleep Convention Logic (SCL), constitutes an interesting alternative to reduce QDI circuits area consumption and increase performance [11]. The SCL template inherits several aspects from its basis template, the Null Convention Logic (NCL) and combines these with a fine-grained sleep logic architecture, which allows logic simplification that leads to significant area gains. QDI circuits are known to imply significant area overhead, around 2 to 4 times the area of a equivalent synchronous circuit. SCL can reduce this overhead and others associated to it, such as leakage power. The proponents of SCL [11] point that its early completion scheme helps in achieving higher throughput.

Despite its advantages, the early completion scheme adds timing constraints that are usually absent in more traditional QDI templates, but this has been so far overlooked. This occurs due to the fact that using early completion schemes imply the acknowledge signal can fire before or at the same time as data is stored. Thus, if the acknowledge signal is propagated and processed faster than the concurrent data propagation through a storage element, data can be corrupted. The SCL template relies on the result of a race between the acknowledgement signaling and data storage. Proponents of SCL do pinpoint the existence of this constraint, but do not deeply explore it in the available literature. This work analyzes the SCL timing constraint and estimates how it impacts the SCL architecture. The analysis quantifies the criticality of this timing constraint during circuit operation, and pinpoints the gates and wires that compose the critical paths of the mentioned race. Experiments with three commercial technologies (CMOS 180nm, CMOS 65nm and FDSOI 28nm) under two widely distinct supply voltage regimes (nominal and subthreshold) demonstrate the importance of considering the identified timing constraint during the design of SCL circuits.

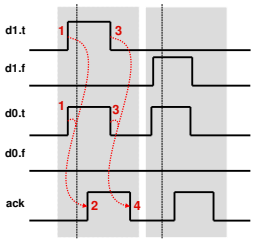
2 QUASI-DELAY-INSENSITIVE DESIGN

Among the different asynchronous design templates available in the literature, bundled-data (BD) and quasi-delay-insensitive (QDI) are the main template families. An advantage of BD design templates is that these can benefit, to some extent, of the use of conventional design tools due to its similarity to synchronous design. The drawback of BD design is that such templates still require extra care with the definition and verification of timing constraints existing among data and control signals. An alternative to avoid these BD design issues is to employ delay-insensitive (DI) channels [15], which is the main strategy defining QDI design templates, associated with appropriate communication protocols. In fact, QDI design is reported by Martin and NystrÃűm as the most practical template, due to its relaxed timing constraints [10]. It requires the choice of handshaking protocol and of a DI code to represent data.

One of the most used DI codes is called dual-rail [10]. Channels based on this code represent each data bit with two wires, which of course brings the need of extra hardware, but relaxes timing matching. A typical dual-rail channel works as follows: use two wires *d.t* and *d.f* to represent each data bit and employ a return-to-zero (RTZ), 4-phase, handshake protocol [13]. Figure 1(a) shows the encoding convention for this channel. To represent a '1' logic level, it is necessary to set *d.t* high and *d.f* low. The representation of a '0' logic level assumes *d.t* is set low and *d.f* high. Both signals set to 1 is an invalid state that must never occur. Figure 1(b) displays an example of operation for a 2-bit (four data wires) channel operation. Note that there is no control signal indicating data validity (such as a request signal), given that each bit can be 0, 1 or be absent (denoted by the occurrence of the spacer encoding). Data is valid when, for every data bit, *d.t* and *d.f* have distinct logic values. The communication protocol requires that between each pair of valid data a spacer must occur. In this RTZ example, all wires return to zero after sending data.

Signals		Value
d.t	d.f	RTZ
0	0	SPACER
0	1	0
1	0	1
1	1	INVALID

(a)



(b)

Figure 1: The encoding basis for QDI circuits, the dual-rail code and a selected protocol: (a) Conventions for encoding a 1-bit dual-rail channel using the RTZ protocol; (b) An example of waveform depicting the transmission of codeword "11" followed by codeword "10", using a 2-bit, dual-rail channel based on the 4-phase, RTZ protocol.

Figure 1(b) shows the waveform that corresponds to the activity of sending two 2-bit data through an RTZ 4-phase handshake dual-rail channel. Initially, all data wires contain 0s, indicating a *spacer*, also called *absence of data* or *empty state* or *null*. The receiver announces it is available to get/process new data, by setting its ack control signal to 0. Next, the value "11" is sent (*d0.t* and *d1.t* are set to 1 and *d0.f* and *d1.f* are set to 0). Note that the receiver computes when the new input data is valid, and announces it to the sender by raising ack to 1. After data reception is acknowledged by the receiver, the sender substitutes data bits by spacers. Then, the receiver acknowledges the reception of the spacer and that its has processed the data sent, by setting ack back to 0. The next data transmission repeats the process for the data value "01". The denomination 4-phase protocol should now be clear, since 4 steps are included in each data communication: (1) data sending; (2) data reception recognition; (3) spacer sending; (4) spacer reception recognition.

Several other codes and protocols exist [1]. This code and behavior, combined with specific circuit structures defines a specific QDI design template.

2.1 The QDI Limitation

In a QDI circuit, gates and wires can have arbitrary delays. However, differently from strictly DI circuits [9], there is a set of designated wire forks that must respect an *isochronic timing constraint*. Such *isochronic forks* imply the requirement that the delay to different ends of these must be the same [9]. Sparsø & Furber [13] explain the isochronic fork as follows. Consider Figure 2. It shows a circuit with three logic blocks (*B0*, *B1* and *B2*) that are interconnected by three wire segments, each with a specific delay (*d0*, *d1* and *d2*). The structure shows there is a fork *F* through which any value produced by the output of the logic block *B0* passes before reaching the respective inputs of blocks *B1* and *B2*. *F* begins after the wire delay *d0* and has two ends, each one with a wire delay: *d1* and *d2*. Following the definition proposed by Martin [9], if wire delays *d2* and *d3* are identical ($d1 = d2$), the circuit in Figure 2 respects the isochronic fork constraint and is thus called an *isochronic fork*. Despite its elegance, this definition has been later refined to ease the practical implementation and analysis of QDI circuits.

In 1995, Manohar and Martin proposed an enhanced definition of isochronic fork and of the *isochronicity assumption* [8]. Consider the same structure presented in Figure 2. Consider also, without loss of generality, that any change in F must be acknowledged (cause a change) at either output *o1* or output *o2* but not necessarily on both. According to Manohar & Martin [8], if fork F is isochronic, it means that observing an acknowledgement in either output is sufficient to assume that both $B1$ and $B2$ have processed the change in F.

For example, when a transition on the input of $B1$ (after delay $d1$) has been acknowledged by a transition on $o1$, then a transition on the input of $B2$ (after delay $d2$) has also completed, even though $o2$ may not have acknowledged it. This is called the *isochronicity assumption*. More specifically, consider that a rising transition occurs in F ($F\uparrow$). This transition will cause $f1\uparrow$ and $f2\uparrow$, respectively after delays $d1$ and $d2$. Next, assume that only $B1$ generates a transition in its output ($o1\uparrow$), while output $o2$ keeps the same logic level. In this case, it is not possible to observe at $o2$ whether the transition $f2\uparrow$ was processed by $B2$. However, if the fork is isochronic, after seeing an effect in $o1$ due to $F\uparrow$, it is safe to assume that $B2$ already processed transition $F\uparrow$ or that it will process this as expected by the design.

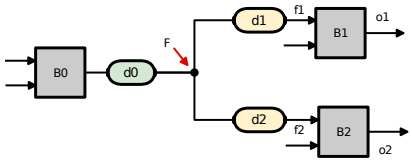


Figure 2: Defining isochronic forks with a lumped delay wire model [13].

In parallel with Manohar & Martin, van Berkel et al. also proposed the definition of *extended isochronic fork* [14]. According to these authors, the fork in Figure 2 is an extended isochronic fork if the delay difference between $F \rightarrow o1$ and $F \rightarrow o2$ is smaller than the delays of the gates driven by the output nodes $o1$ and $o2$. That means that all output nodes must be stable when the following gates are triggered. An important aspect is that this definition does not consider the wire delays of the fork only, but also the gate delays. This is different from the original definition of the isochronic fork, where gate delays were not considered.

To conclude this discussion, it is relevant to ask what consequences could result from violating an isochronic fork assumption. Returning to Figure 2, assume that F is used to trigger the generation of related control and data signals $o1$ and $o2$, respectively in blocks $B1$ and $B2$, which work based on the fact that F is isochronic. If F is not isochronic we could e.g. generate the control signal $o1$ before data signal $o2$ is stable. This could clearly lead to data corruption in downstream hardware using $o1$ and $o2$. Section 4 shows the SCL design template produces isochronic fork constraints that need to be carefully designed.

3 SLEEP CONVENTION LOGIC

SCL is a QDI asynchronous logic design template [11] inspired on NCL [2]. Its structure has two main characteristics: (i) use of NCL gates with early completion [12] and fine-grained multi-threshold CMOS (MTCMOS) power-gating [16]. In fact, SCL was initially called Multi-Threshold Null Convention Logic (MTNCL) [16]. Compared to NCL, SCL brings gate-level and architectural that may result in area and performance advantages [11]. As an example, Figure 3 shows a 3-stage SCL pipeline. Each stage contains a combinational logic block F, a register R, a completion detector CD and a settable C-element[1] C.[2]

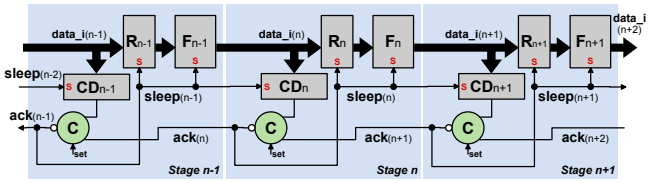


Figure 3: Structure of a 3-stage SCL pipeline.

The SCL template integrates the acknowledgement signal *ack* with the sleep mechanism. Basically, focus attention on Stage n. When this pipeline stage acknowledges the previous Stage $n-1$, it also resets the sleep signal, which takes Stage n out of sleep mode. To better understand the SCL pipeline operation, consider the structure in Figure 3 in a sleep state: all $data\_i(n) = null$ (contain a spacer) and all $ack(n) = sleep(n) = 1$, i.e. all pipeline stages are in sleep mode. Initially, when a first data ($data\_i(n-1)$ and $sleep(n-2) = 0$) arrives, $sleep(n-1)$ is reset, awaking CD_n to check the validity of $data\_i(n)$. When $data\_i(n)$ presents a valid data token, CD_n asserts its output, which forces the output of the C-element of Stage n to 0 ($ack(n) = 0$). Remember that $ack(n+1) = 1$. At this moment, two concurrent events take place, as the output of the C-element of Stage n forks to two paths: (a) the acknowledge signal $ack(n)$ is lowered, signaling that stage $n-1$ confirms the data validity and (b) the stage n sleep signal is disabled ($sleep(n) = 0$), awaking R_n, F_n and CD_{n+1}. Enabled, register R_n and the combinational block F_n can now propagate their inputs, generating a valid data token in $data\_i(n)$, which will be detected by CD_{n+1}. Next, stage $n+1$ can execute the same process as described previously for Stage n.

The combinational logic blocks F_i consist basically of SCL gates, which couple a threshold function with positive integer weights assigned to inputs and power-gating logic. Compared to NCL gates, SCL gates present a notable area reduction, due to the fact that the latter only use two logic blocks from the original definition of NCL gates: the HOLD0 and SET [2]. The remaining logic blocks (HOLD1 and RESET) are replaced by the sleep logic, which is responsible for generating logic 0 at the output. This optimization removes the need for using feedback logic present in most NCL gates, to provide the NCL hysteresis mechanism. An SCL gate is named according to its threshold value, equivalent to what is used in NCL gates. Following the terminology presented in [11], an SCL gate is denoted by $THmnW w_1, ..., w_n$, where n is the number of gate inputs, m is the gate threshold, and $w_1, w_2, ..., w_n$ are weights of

[1] This is a C-element with inverted output. When set=0, both inputs at 0 force output to 1. Conversely, both inputs at 1 force the output to 0. Otherwise, the previous output is kept unchanged. Of course, set=1 alone forces 1 to the output.

[2] In [11], authors refer to this C-element as "resettable", which is inaccurate. The SCL template requests that all C-element outputs are 1 to maintain all pipeline stages in sleep mode ($sleep = 1$).

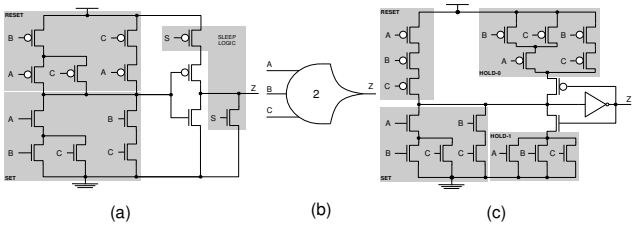


Figure 4: An example of SCL and NCL gate comparison using the TH23 gate: (a) transistor-level implementation of the TH23 SCL gate; (b) the TH23 gate symbol and (c) transistor-level implementation of the TH23 NCL gate. Note that both SCL and NCL implementation use the same symbol, except the fact that the SCL version has a additional input: the sleep signal, not shown in the symbol.

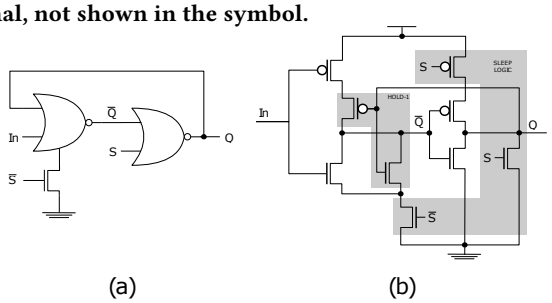


Figure 5: A single-bit SCL register: (a) gate-level implementation; (b) transistor-level implementation. Note that the NMOS gating uses the negated value of S ($\overline{S}$), which implies that an inverter must be added to the structure.

inputs if the weights are > 1. If the weight = 1, usually its value is omitted from the classification. To illustrate this, Figure 4 shows the transistor-level implementation of a TH23 SCL and a TH23 NCL gates, each with threshold=2, 3 inputs, and all input weights=1 (accordingly, weight information is omitted).

A completion detection block CD is responsible for detecting valid data and spacers in its input $data\_i$. Its implementation is similar to completion detectors from NCL, except for the fact that SCL completion detectors employ SCL gates, which allow incorporating the sleep logic, further reducing static power consumption.

Figure 5 presents the structure of a single-bit SCL register. The SCL register employs a two-level logic with sleep transistors and an unconventional latch. Figure 5(a) shows the gate-level representation of the register. Basically, the SCL register uses two NOR gates, where the first NOR gate has an NMOS gating and the second NOR gate partially implements the sleep logic. Note that the first NOR receives the output Q as input, implementing the latch feedback signal. In addition, Figure 5(b) brings the transistor-level representation of the register, highlighting the sleep logic and the feedback structure (HOLD-1).

4 THE SCL ISOCHRONIC FORK

Except for the fact that early completion improves the architecture throughput, its implementation brings with it timing constraints. The idea to acknowledge the previous pipeline stage before storing

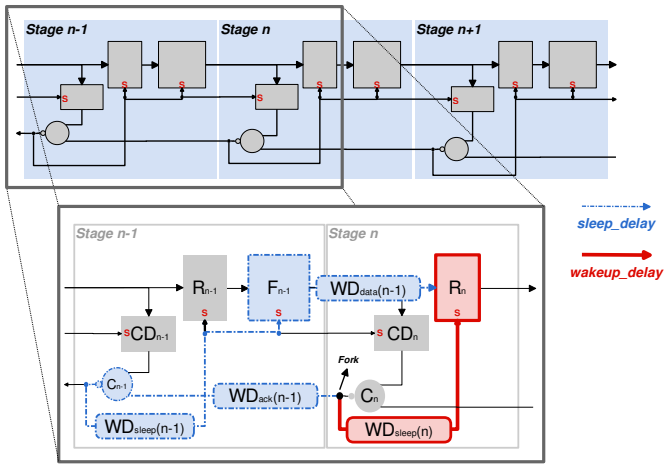


Figure 6: Delay analysis of the SCL isochronic fork between two pipeline stages.

data in a current pipeline stage may compromise the circuit QDI functionality. This can happen if the previous pipeline stage resets (i.e. produces NULL at its output) before the register of the current pipeline stage stores valid data. This problem can be visualized in Figure 6, which focuses on the two first stages of the SCL pipeline presented in Figure 3. Suppose that Stage n receives valid data at its input and CD_n detects it. Consequently, the inverted C-element of Stage n lowers its output. At this point, two concurrent events take place, as the output of this gate forks to two paths: (1) The Stage n sleep signal is disabled, awaking pipeline Stage n to store and compute its input data, output of Stage $n-1$; (2) The acknowledge signal is sent to the previous pipeline stage (Stage $n-1$), forcing it to sleep and reset all its logic blocks to null. If Stage n awakes faster than the reset process of Stage $n-1$ completes, register R_n should sample and store the valid data correctly, and the pipeline will operate normally. However, if Stage $n-1$ is able to sleep and reset all its logic before the awaking of Stage n, register R_n may not be able to correctly store the new data. This possible functionality failure implies that the architecture must respect an isochronic fork constraint that it is not entirely covered by previously published works. From this analysis, it is possible to observe that there is a timing assumption between the awakening of registers and the sleeplessness of logic blocks. In fact, recalling Section 2, this assumption relies on the isochronicity of the fork in the output of the inverted C-element. To quantify this timing assumption, Figure 6 shows the two concurrent paths between Stage $n-1$ and Stage n: a continuous thick (red) path and a dashed (blue) path, both comprised by a sequence of wires and logic components. The continuous thick path represents the delay path to *wake up* Stage n and incorporates three delay elements: a wire delay $WD_{sleep}(n)$, the rise propagation delay of register $R_n \uparrow$ and the register hold time R_{n_hold}. Hence, it is possible to indicate that the wake up delay of Stage n amounts to:

$$wakeup\_delay(n) = WD_{sleep}(n) + R_n \uparrow + R_{n\_hold} \quad (1)$$

The dashed path represents the delay path to *put to sleep* Stage $n-1$ and incorporates five delay elements: (i) the wire delay $WD_{ack}(n-1)$; (ii) the propagation delay of the inverted C-element of Stage

$n-1$; (iii) the wire delay $WD_{sleep}(n-1)$; (iv) the fall delay of combinational logic block $F_{n-1}\downarrow$; and (v) the wire delay $WD_{datapath}(n)$. Note that in this case the propagation delay of R_n need not be considered. As both register and combinational logic are reset during the sleep process, the combinational logic will reset its output independently of the delay of the register. In that way, the sleep delay of a Stage n can be equated to:

$$sleep\_delay(n) = WD_{ack}(n) + C_n + WD_{sleep}(n) + F_n\downarrow + WD_{datapath}(n) \quad (2)$$

Consequently, the timing constraint between two contiguous pipeline stages is given by the following inequation:

$$wakeup\_delay(n) \leq sleep\_delay(n-1) \quad (3)$$

This Inequation states that the *wake up delay* of pipeline Stage n must be less than or equal to the *sleep delay* of pipeline Stage $n-1$. In words, Equations 1 and 2 combine to determine which delay element of a delay path is to be manipulated to satisfy the timing constraint in Inequation 3.

In case the circuit does not respect the timing constraint, it is necessary to manipulate the delay elements of the fork, increasing (or decreasing) delay values. This can be done by establishing timing constraints to an EDA tool to add (optimize) delay elements of the fork. However, it is important to understand the trade-offs that could come to play as delay is added (or reduced) in the fork paths. Hence, each delay element must be analyzed individually to point the negative and positive aspects of increasing (decreasing) its values. We analyze two alternatives to satisfy the timing constraint proposed by Inequation 3: (i) increasing delay values of the $sleep\_delay(n-1)$; and (ii) decreasing delay values of $wakeup\_delay(n)$.

Focusing on $sleep\_delay(n-1)$, consider increasing the wire delay $WD_{ack}(n-1)$, which connects the C-elements. This can be a valid option at the cost of a higher *ack* propagation delay. This same trade-off occurs whether the delay propagation of the Stage $n-1$ C-element is increased. Another option is to increase the sleep delay of the combinational block F_{n-1} to reset its output. F_{n-1} is out of the *ack* path and will not affect the *ack* propagation delay. However, this implies that the sleep logic in the SCL gates should be resized and, consequently, will affect the datapath delay. Note that the same effect happens whether $WD_{datapath}$ is increased. As a last resort, the $WD_{sleep}(n-1)$ could be increased and the *ack* propagation delay would not be affected, neither the datapath's delay. Unfortunately, $WD_{sleep}(n)$ composes the $(wakeup\_delay(n-1))$ and its increase may also affect the isochronic forks of previous stages. For instance, if $WD_{sleep}(n-1)$ is increased to satisfy $wakeup\_delay(n) \leq sleep\_delay(n-1)$, it also increases $wakeup\_delay(n-1)$. This increase may affect the previous isocrohonic fork constraint $wakeup\_delay(n-1) \leq sleep\_delay(n-2)$, requiring more delay manipulations in previous isochronic forks. This may cause a ripple performance degradation effect on previous stages. All these points suggest that manipulating $sleep\_delay(n-1)$ brings performance trade-offs, specially when interdependent delay elements between fork paths are manipulated.

The second alternative is to decrease the delay value of the amount $wakeup\_delay(n)$. In this case, there are only two options: decrease $WD_{sleep}(n)$ or/and R_n. As before, manipulating the amount $WD_{sleep}(n)$ may be a complex option, due to the interdependence between interacting fork paths. Decreasing $WD_{sleep}(n)$ will not only decrease $wakeup\_delay(n)$ but also $sleep\_delay(n)$, wherein the latter must be a higher value than $wakeup\_delay(n+1)$. This manipulation forces the next pipeline stage timing constraint to have lower delay values and may not be achievable, if the next fork paths cannot be optimized. Register R_n could be optimized whether multiple drive strengths are available to decrease the register delay.

4.1 An Analysis of Process Variation Effects

Another possible set of problems while operating with the SCL isochronic fork are the issues caused by current variations, which imply that only guaranteeing that

$$wakeup\_delay(n) \approx\leq sleep\_delay(n-1) \quad (4)$$

may not be enough. Delays inside the circuit can change arbitrarily due to current variations, and the isochronic fork may not always respect the timing constraint defined by Equation 3, if delay variations are too significant. This determines the need to consider margins on delay paths to guarantee circuit functionality, even with a certain amount of delay variations on its logic and wires. In order to support current variations, Equation 5 modifies Inequation 3, adding a margin Var_{margin} to the $wakeup\_delay(n)$ branch.

$$wakeup\_delay(n) + Var_{margin} \leq sleep\_delay(n-1) \quad (5)$$

In super-threshold operation, the current variation of transistors can be translated to a Gaussian distribution $N(\mu, \sigma)$, where μ represents the mean value of the distribution and σ the standard deviation. Often, the confidence interval is defined by a $n \times \sigma$ distance from μ, which can define Var_{margin} in Equation 5. According to [3], using $\mu \pm 3\sigma$ as endpoints defines an internal confidence that covers all samples with a 99.7% probability. For subthreshold operation, however, the relation of endpoints with the standard deviation is not straightforward. As subthreshold currents display an exponential dependency on the threshold voltage, the subthreshold current variation translates to a log-normal distribution [7]. Thus, it would be inaccurate the usage of a confidence interval such as $\mu \pm 3\sigma$ to determine the endpoints. Fortunately, it is possible to overcome this issue by transforming a log-normal distribution to a Gaussian distribution, allowing correlation between $n \times \sigma$ with a log-normal confidence interval. Considering the approach demonstrated in [3] and a confidence interval of 3σ, Var_{margin} can be defined as:

$$\begin{aligned} Var_{margin} &= \mu \pm 3\sigma, \quad \text{for nominal operation} \\ Var_{margin} &= 10^{\mu \pm 3\sigma}, \quad \text{for subthreshold operation} \end{aligned} \quad (6)$$

4.2 The Influence of Wire Delays

It has been shown that wire delays cannot be disregarded as transistor sizes shrink [5]. There are two kinds of wires: local ones, which dominate implementations, and global ones. Local wires in recent technologies can either track the gate delay or be larger than gate delays [5]. These wires are short and often used to route, for instance, internal signals of standard cells. Global wires, in turn, are a bigger problem. These wires are used to route among logic blocks, and usually require optimizations, such as the addition of intermediate repeaters [5].

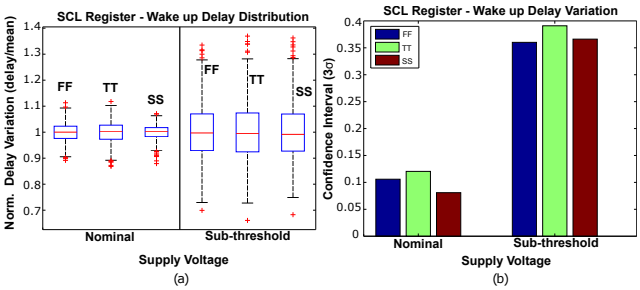


Figure 7: An analysis of the delay variation behavior for an SCL register cell: (a) the propagation delay variation of the SCL register; All distributions are normalized to the mean, three process corners are considered (FF, TT, SS) together with two supply voltages (nominal and subthreshold); (b) the interval 3σ distance for the distributions in (a).

5 EXPERIMENTS

The experiments described here examine the SCL isochronic fork timing constraint fulfillment in practice. Experiments are based on the formal definition expressed by Inequation 3, derived in Section 4. The objective of the experiments is to highlight the feasibility and overheads associated with implementing SCL circuits while respecting its timing constrains, subject to different conditions of voltage supply, circuit structures and technology nodes. Moreover, the experiments employ Monte Carlo simulations to consider within-die variations (process corner mismatches).

In the analysis, only regular two-stage SCL pipelines are considered. These are sequentially connected as highlighted in Figure 6. As the fork comprises several delay components, additional assumptions were set to simplify the analysis. First, all existing wire delays in the fork are assumed equal ($WD_{sleep}(n) = WD_{ack}(n-1) = WD_{sleep}(n-1) = WD_{datapath}(n-1)$). Second, all delays in the fork are normalized to the delay of the SCL sleep logic, which is always implemented as a NOR gate in the output of all SCL logic (see Figures 5 and 4). The normalization allows to relate all fork delays, including gates and wires. As this analysis considers different technology nodes, $F_{n-1}\downarrow$, $R_n\uparrow$ and C-elements are implemented in three different technology nodes: 180nm Bulk CMOS, 65nm Bulk CMOS and 28nm FDSOI-CMOS. Nominal and subthreshold operation (30% of the nominal supply voltage) are examined. Besides, delay variations due to process corner mismatch are investigated. Considering nominal and subthreshold operation allows determining the delay relations among logic gates (R_{norm} and the C-elements) and the delay variation Var_{margin} in both supply regimes.

Figure 7(a) illustrates the propagation delay variations for an SCL register cell implemented in 65nm bulk CMOS technology. Results were obtained after 1000 Monte Carlo simulations using three different process corners (SS, TT and FF) and two supply voltages (1V and 0.3V). As expected, the SCL register displays larger delay variations in the subthreshold regime when compared to variations under nominal operation. Corners within a same supply voltage had small normalized differences among themselves. This latter observation can be better observed in Figure 7(b), which shows the confidence distance 3σ of the distributions in Figure 7(a). While the three process corners under nominal operation show a maximum distance of around 10% from the mean value, in subthreshold this maximum distance is around 35 to 40%. This higher delay variation implies that if an SCL circuit is targeted to subthreshold operation, higher values of Var_{margin} must be considered.

Having established the delay variation parameters of each target technology and the relation among all logic delays, Figures 8(a) and (b) indicate when a Delay Element (DE) is required in nominal and subthreshold regimes, respectively, depending of the wire delay contribution versus the logic delay contribution in the isochronic fork timing constraint. Note that both axis are normalized to the SCL sleep logic delay. The "DE required" region covers positive values of the delay element, while "No DE required" region covers negative values. If the DE value is negative, the fork itself already guarantees its isochronicity. Otherwise, a DE must be inserted according to the normalized DE value. For 180nm, the fork has a significant margin to respect its isochronic fork timing constraint in nominal operation. If wires reach 40% or more of the delay of the SCL sleep logic, no DE is required. Now, an insignificant wire delay contribution ($\leq$ 40%) in the fork implies the need to add a DE. Fortunately, the overhead seems low as the normalized DE value only reaches until 1. In other words, adding one or a couple of buffers in the fork could fullfil the isochronic fork timing constraint. Regarding 65nm and 28nm technology nodes in nominal operation, the required margin increases slightly. In these cases, no DE is required only if wires have a delay that is the same as the SCL sleep logic or if this delay is larger than that. This type of scenario is feasible, since the delay of wires has not decreased as much as the delay of gates along the evolution of technology nodes. In fact, Huang & Ercegovac [6] point out that wire delays can exceed the gate delays in several critical paths and that these can also increase depending the circuit size. However, if the fork comprises only small wires, a DE is required and its implementation may require several buffers.

As supply voltages reach the subthreshold region (see Figure 8 (b)) significantly higher margins are required to guarantee the fork isochronicity. Consider again the 180nm technology node. The isochronic fork timing constraint requires the insertion of a DE if wire delays are 3× or less the SCL sleep logic delay. For 65nm and 28nm, wire delays must be, respectively around 4× and 5× larger than the logic delay or more, to avoid the need for DEs. This pinpoints that the SCL template will face difficulties in respecting the isochronic fork timing constraint without insertion of extra DEs. In addition, the implementation of a DE in subthreshold operation can require a large number of buffers, increasing area and power overheads.

6 DISCUSSIONS AND CONCLUSIONS

The analysis presented in this work shows that delay variations and wire delay contributions play a significant role in isochronic fork timing constraints fulfillment for the SCL QDI asynchronous design template. This was showed for SCL, but the work can easily be generalized to be applicable to other QDI design templates. Operating under both nominal and subthreshold regimes, the paper showed a large wire delay at specific points in an SCL circuit can guarantee the respect of the timing constraint, even in worst-case delay variations scenarios.

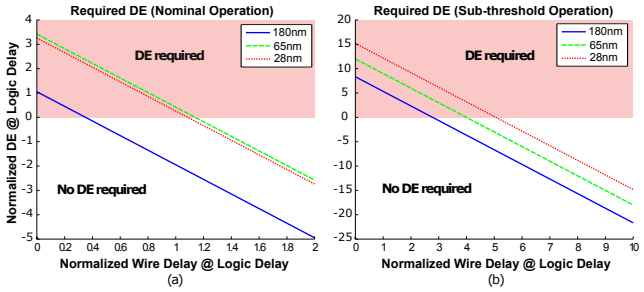


Figure 8: An analysis of the requirement for delay elements (DEs) considering three technology nodes: 180nm bulk CMOS, 65nm bulk CMOS and 28nm FDSOI CMOS. Axis X represents the delay value of all wires in the fork ($WD_{sleep}, WD_{ack}, WD_{sleep}, WD_{datapath}$) and axis Y indicates the required number of DEs: (a) nominal operation data and (b) subthreshold operation data. Note that both axis are normalized to the SCL sleep logic delay.

Obviously, wire delay contributions are directly tied to how far from each other logic stages are. More specifically, consider the fork branches from Figure 6. It is possible to claim that internal wires in pipeline stages such as WD_{sleep} most often connect to nearby logic and, thus contribute with relatively small delays. However, when wires establish connection between pipeline stages, wire delays depend on how far the stages are from each other. In Figure 6, the wires that connect pipeline stages are related to WD_{data} and WD_{ack}, where WD_{data} typically stands for a datapath that may contain several bits (related to a chosen data width) and WD_{ack} indicates a single wire acknowledgement control signal.

For example, if stage $n-1$ and n implement the logic in the communication interface of neighbor routers of a Network-on-Chip (NoC), WD_{data} and WD_{ack} are probably long wires and correspond to larger delay contributions. On the other hand, if stage $n-1$ and n implement physically neighbor logic stages, such as in a pair of contiguous positions of a FIFO, WD_{data} and WD_{ack} will probably correspond to short wires. As Section 5 indicates, a larger wire delay contribution to the fork branches helps an SCL circuit to respect its timing constraint, despite the fact that it also brings higher propagation delay during the acknowledgement process as a side effect. This implies that communication between distant stages are more likely to respect the timing constraint without the need of additional DEs, whereas communication between neighbor stages can lead to requirements of DE(s) insertion. If the latter scenario is true, the required DE(s) affects the circuit according to the adopted supply voltage. For nominal operation, the required delay element is usually small and can impact only lightly the circuit characteristics. However, the addition of DEs under subthreshold regimes is expected to cause a more important impact on circuit features.

Considering the work conducted here, it is possible to devise two mains aspects for ongoing and future work. First, the experiments in Section 6 present a straightforward approach to model the wire delays and relate them with logic gates. Despite its simplicity, this approach does not provide a clear notion of how wire delays change and how these same delays behave under supply voltage scaling situations. This has motivated the analysis of the distribution of wires in case-study circuits and model their respective delays, according to variables such as capacitance, wire length, fan-in and supply voltage. These collected data have as target to provide a statistical view of wires in a given circuit and enable more sophisticated analysis on complex constraints arising during asynchronous circuit design. The second aspect is the proposition of improvements in the SCL architecture to avoid or reduce the timing constraints identified in this work. The early completion detection scheme used in the current SCL specification can be easily replaced by a traditional completion detection scheme, drastically reducing the magnitude of the timing constraints associated to SCL, while keeping the SCL qualities compared e.g. with NCL. In addition of that, the authors have already evaluated the performance issues caused by the SCL completion detection scheme in [4]. This work reinforces the need for improvement on the original SCL template.

REFERENCES

[1] P. A. Beerel, R. O. Ozdag, and M. Ferretti. 2010. *A Designer's Guide to Asynchronous VLSI.* Cambridge University Press.

[2] K.M. Fant and S.A. Brandt. 1996. NULL Convention LogicTM: a complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application Specific Systems, Architectures and Processors (ASAP).* 261–273. https://doi.org/10.1109/ASAP.1996.542821

[3] Tobias Gemmeke, Maryam Ashouei, and Tobias G. Noll. 2013. *Noise Margin Based Library Optimization Considering Variability in Sub-threshold.* Springer Berlin Heidelberg. 72–82 pages. https://doi.org/10.1007/978-3-642-36157-9_8

[4] Ricardo A. Guazzelli, Walter L. Neto, Matheus T. Moreira, and Ney L. V. Calazans. 2017. A Comparison of Asynchronous QDI Templates Using Static Logic. In *8th IEEE Latin American Symposium on Circuits & Systems (LASCAS).* 261–264.

[5] Ron Ho, Kenneth W Mai, and Mark A Horowitz. 2001. The future of wires. *Proc. IEEE* 89, 4 (2001), 490–504.

[6] Zhijun Huang and M. D. Ercegovac. 2000. Effect of Wire Delay on the Design of Prefix Adders in Deep-Submicron Technology. In *34th Asilomar Conference on Signals, Systems and Computers (ACSSC)*, Vol. 2. 1713–1717.

[7] Bo Liu and M. Ashouei. 2012. Standard cell sizing for subthreshold operation. In *49th ACM/IEEE Design Automation Conference (DAC).* 962–967.

[8] Rajit Manohar and Alain J. Martin. 1996. Quasi-delay-insensitive circuits are Turing-complete. In *2nd IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC).* Available as Caltech Technical Report CS-TR-95-11, Nov 1995, at http://vlsi.cornell.edu/ rajit/ps/qdi.pdf.

[9] Alain J. Martin. 1990. The Limitations to Delay-insensitivity in Asynchronous Circuits. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI (AUSCRYPT '90).* MIT Press, Cambridge, MA, USA, 263–278. http://dl.acm.org/citation.cfm?id=101415.101434

[10] A. J. Martin and M. Nyström. 2006. Asynchronous Techniques for System-on-Chip Design. *Proc. IEEE* 94, 6 (June 2006), 1089–1120.

[11] F. A. Parsan, S. C. Smith, and W. K. Al-Assadi. 2016. Design for Testability of Sleep Convention Logic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 2 (2016), 743–753.

[12] S.C. Smith. 2002. Speedup of self-timed digital systems using Early Completion. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* 98–104. https://doi.org/10.1109/ISVLSI.2002.1016884

[13] J. Sparsø and S. Furber. 2001. *Principles of Asynchronous Circuit Design âĂŞ A Systems Perspective.* Springer.

[14] K. van Berkel, F. Huberts, and A. Peeters. 1995. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Second Working Conference on Asynchronous Design Methodologies (ASYNC).* 99–106. https://doi.org/10.1109/WCADM.1995.514647

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

[16] Liang Zhou, S.C. Smith, and Jia Di. 2010. Bit-Wise MTNCL: An ultra-low power bit-wise pipelined asynchronous circuit design methodology. In *53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS).* 217–220.