# Asynchronous Circuit Principles and a Survey of Associated Design Tools

Ney L. V. Calazans*, Marcos L. L. Sartori†
*Independent Researcher, Brazil
†Pontifícia Universidade Católica do Rio Grande do Sul - Porto Alegre, Brazil
nlvcalazans@gmail.com, marcos.sartori@acad.pucrs.br

***Abstract*—Planning and implementing a semiconductor integrated circuit is a highly complex process. Although physical limits seem to be approaching, it currently follows a growing evolutionary path. As deep submicron technologies evolve towards perhaps even sub-nano geometries, the design process complicates accordingly. Once subtle in higher geometry nodes, some effects become relevant or even dominant. Examples are effects that tamper the reliability of wires, such as crosstalk, or the adequate behaviour of gates, such as the increasing sensitivity to single event effects. Design techniques must thus also evolve, to provide a wide range of tools to handle new effects during the integrated circuit design and test processes. This tutorial covers one set of design techniques that is often overlooked, but which can reveal themselves instrumental in coping with the mentioned technology evolution, the use of clockless or asynchronous circuits. The tutorial is divided into three parts: first it introduces a metamodel for the digital circuit design process, enabling to reason about distinct design styles; second, it covers the main principles of asynchronous circuit design, differentiating it from mainstream circuit design techniques such as conventional synchronous design; the third and last part presents a set of tools and systems that can be employed to effectively design asynchronous circuits, with emphasis on material that can be used to produce manufacturable circuits and systems, often associated to commercial integrated circuit synthesis, implementation and test tools and frameworks.**

## I. Introduction

The evolution of the technology for semiconductor manufacturing nodes in the last half century is astonishing. If in the beginning of years 70´s of the previous century the state of the art was around a $10\mu m$ feature size, the world now sees designs being sent for fabrication in $3nm$ (and soon $2nm$) technology nodes. Also, recent researches indicate it is feasible to consider the availability of $1nm$ nodes for industrial use in a near future [1].

In tandem with this evolution, problems to employ such new technologies defying both design companies and designers. Designing latest node chips is very hard and very costly, with integrated circuit (IC) mask sets reaching a cost of dozens of US$ millions and designs ending up by costing several hundreds of US$ millions. Also, to deal with newer node design complexity, designers must rely much more intensively on electronic design automation (EDA), but these are quite behind in addressing all features and added complexities of new nodes. Consequently, state of the art chip design and manufacturing is an asset limited in reach to a handful of design houses and just to a few manufacturers, even considering the whole planetary scope.

In advanced technology nodes there are several relevant circuit parameters which are hard to predict and or control, including but not restricted to: manufacturing faults, design variability, yield rate for good dies. Also, as nodes evolve, models to compute the longevity of circuits must be constantly enhanced. Nonetheless, the range of commercial technologies available for fabrication widens, since older technology nodes continue to occupy certain market niches, because their use continues to be economically advantageous. For example, commercial ICs using a $180nm$ or even a $250nm$ technology are still designed and commercialised. Note that this corresponds to feature sizes two orders of magnitude or more larger than what is available in state-of-the-art nodes. Many other technology choices are there in between.

Another design dimension to consider is that newer features intervene as the technology node shrinks. Thus, decisions to use features unavailable in older technologies add up to the design process. For instance, transitioning from $45nm$ or $32nm$ to smaller feature sizes enables a change in the way transistors are to be designed, forcing the abandonment of pure CMOS bulk technologies in favour of the more advanced technologies. Examples of the latter are FinFET and FDSOI. As another example of design choice unavailable before, there is the possibility to employ multiple transistor types in a same or in alternate designs for a given technology node. This starts to be available for technologies around the $130nm$ feature size node. In this and subsequent nodes there are choices of for example transistors with distinct threshold voltages, that enable to better control device characteristics such as speed and/or current leakage.

Within the context of the discussed technological scenarios, adopting the synchronous digital circuit design paradigm is one of the reasons behind the rapid development of the VLSI industry, due to the enhanced productivity it enables for IC designers. This paradigm reduces design complexity, through the use of a global control signal called *clock* dictating all event sequencing in the circuit. A synchronous designer can ignore wire and gate delays during several of the design phases, as long as the logic path between each pair of storage elements always takes less time than the clock period. Unfortunately, distributing a clock signal across a complex IC is challenging today, due to the exponential growth of integration capabilities, among other factors. Albeit there are different techniques and EDA support to automatically generate the clock distribution, the required circuitry may take something from 30% to 50% of the total power in synchronous circuits [2]. This is further complicated by the inevitable delay uncertainties caused by data dependencies and process, voltage and temperature (PVT) variations. To cope with these, synchronous designs rely on the addition of delay margins to the clock signal, which translates to performance losses, and can require tuning the operating voltage, further adding power overheads [3], [4]. Asynchronous design, on the other hand, does not rely on global timing assumptions and treat time as a continuous

Digital Object Identifier 10.29292/jics.v17i3.677

variable such that synchronisation and sequencing of events take place locally, between communicating entities [5].

A set of relevant issues for available technology nodes include energy efficiency and robustness. Energy efficiency is easy to define and measure. Approaching design robustness requires more insight. Investigating design robustness in detail is outside the scope of this tutorial, but Calazans et al. have recently described ways on how the use of asynchronous design can be beneficial to improve robustness in [6]. Interested readers can refer to this publication for more on robust design using asynchronous circuits.

This tutorial covers two aspects of asynchronous circuits. The first is to describe the main principles underlying the design of asynchronous circuits, showing that the umbrella name *asynchronous circuits* encompasses a large set of completely distinct and/or complementary digital design techniques, having in common a single assumption of not using one or only small set of synchronising signals usually denominated *clocks*. The second aspect is a limited vision about usable methods and tools to deal with the design of asynchronous circuits. The focus here is on covering only a mainstream set of tools and systems available (often as open source) to design such circuits. Most of these intrinsically rely on commercial design automation frameworks such as Cadence and/or Synopsys and tweek these to make the frameworks deal with asynchronous circuits idiosyncrasies while automatically generating circuits.

The rest of this tutorial comprises four sections. Section II provides a metamodel called digital circuit design template (DCDT), useful for analysing digital circuit design process models. This enables confronting, on common grounds, the panoply of current and future digital design techniques asynchronous or not. Section III next introduces a main set of principles and concepts behind asynchronous circuit design styles, sometimes using the DCDT metamodel as a reasoning to explain similarities and divergences of distinct design processes. Section IV discusses a selected set of relevant state-of-the-art design tools for asynchronous design. Finally, the paper ends with Section V, where a few conclusions on the subject, a process helped by some references to recent examples. In these, asynchronous design achieves better results than synchronous counterparts, producing effective digital circuits in selected, relevant application niches.

## II. A Metamodel for the Digital Design Process

Digital design currently enables composing billions of logic gates into working circuits with a large set of functionalities to process information in fast and accurate ways. To handle the huge amount of gates and wires that finally implement a powerful circuit requires effective disciplines to manage the design process complexity. These disciplines involve devising efficient methods for combining very simple devices (e.g. logic gates) into basic modules, methods for abstracting modules, methods for interfacing modules and methods to reuse components with a given complexity, to cite only a few of the techniques involved. To organise the way a digital circuit is planned, it is useful to introduce the concept of a digital circuit design template. The concept is an evolution of the *Asynchronous Circuit Template* definition, first proposed by Moreira in [7]. It enables to reason about how a digital circuit can be systematically implemented, organising the design process with a set of encompassing design abstractions that can be naturally mapped to any set of specific design techniques used for digital circuit design. The concept definition below is illustrated by the diagram of Figure 1.

**Definition 1** (Digital Circuit Design Template (DCDT))**.** *A* **digital circuit design template** *(DCDT) is a metamodel composed by two entities: a* **design style** *and a* **channel***. The design style comprises two sub-entities, a* **set of cells** *and an* **architecture***. The channel, in turn, is also a composition of two sub-entities, a* **communication link** *and a* **protocol***. Cells in the set of cells are the basic blocks available to build circuits (e.g. the logic gates of a standard cell library), while the architecture is a set of rules for combining cells into valid circuit configurations. A communication link may be as simple as a wire connecting a pair of gates, or it can be a much more elaborate interconnection structure. A protocol establishes how information must flow in communication links and must be defined in accordance to the set of cells and the template architecture rules.*

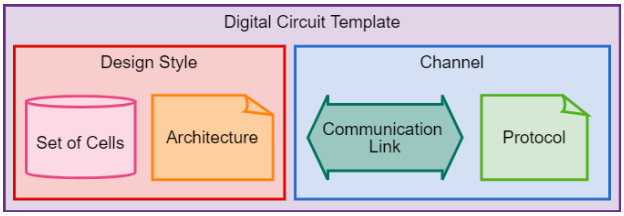


Fig. 1. A diagram illustrating the digital circuit design template concept.

To enable readers to appreciate the usefulness of the DCDT metamodel, it will be fitted to the very well known synchronous register-transfer level (RTL) digital design template. Later, this baseline modelling of a familiar template is compared to asynchronous design templates, which enables to better embrace the similarities and differences among synchronous and asynchronous digital design processes.

The basic way to model a known template using the DCDT metamodel consists simply in mapping the template characteristics to the the metamodel four entities, which for the synchronous RTL template gives:

1) The *set of cells* for synchronous RTL design can be mapped to the set of logic gates finally used to design a circuit. Even if the circuit behaviour and structure is often captured using hardware description languages (HDL) such as Verilog or VHDL, there is always an underlying library which, to support synchronous RTL design, is most often composed by combinational gates and sequential elementary components such as flip-flops. Of course, other design implementation options such as Field-Programmable Gate Arrays (FPGAs) may add to these a set of technology-dependent cells such as LUTs, carry-chains, etc.
2) The *architecture* for synchronous RTL design implies three set of rules: (i) rules to interconnect gates, forming functional combinational logic (CL) modules, able to transform data; (ii) rules to interconnect functional modules inputs and outputs to registers (or primary inputs/outputs); (iii) rules to connect registers to functional modules (or primary inputs/outputs).
3) The *communication link* for synchronous RTL relies on the assumption that: (i) wires encode information, being somehow ordered to represent digital numbers; (ii) besides, a special wire (or more generally a clock value distribution structure usually called *clock tree*) controls the flow of data everywhere in the circuit.

4) Finally, the *protocol* for synchronous RTL design is the widespread synchronous protocol, stating that when the clock ticks (i.e. transitions in one of two possible directions, from 0 to 1 or from 1 to 0, respectively defining either clock rising edge or falling edge sensitive synchronous templates), every register in the circuit potentially gets new data (exceptions can of course apply).

The above modelling exercise shows the DCDT metamodel modularity enables describing multiple templates, either some very close to the popular synchronous RTL template or others very different from it. For example, if instead of acting on only one clock edge (as most synchronous designs do) a circuit is defined to act on both clock edges, a new template arises (say synchronous, double edge sensitive), that requires at least a different set of cells and certainly a distinct set of architectural rules. Other templates from the literature can also be seen as possible to model with DCDT, e.g. synchronous two-phase design, latch-based (instead of flip-flop-based) design, or clock-skew tolerant design [8]. Exercising such metamodel mappings is left as exercise to readers.

It is important to assess how to deal with the DCDT metamodel for describing asynchronous design templates. In fact, the variety of such templates is quite large, and there is none among them that matches the popularity of the synchronous RTL template. The question arises as to what can/must change in DCDT entities when these are used to model asynchronous design templates. The answer goes from *almost no change* to *basically everything*, depending on the template. The closer to synchronous an asynchronous design template is, the easier it is for synchronous designers to understand it, and the easier it is to use synchronous electronic design automation (EDA) tools in the process of designing such circuits, both of which are clear advantages. The other side of the coin reveals that the farther an asynchronous design template is from synchronous design, the better is the potential to achieve: (i) power efficiency; (ii) robustness to variations, to single event effects (SEEs) and to technology migration; (iii) graceful circuit ageing and reduced electromagnetic interference and resistance to side channel attacks (SCA).

To conclude this Section it is worth to exemplify how each DCDT entities/sub-entities vary for asynchronous design templates. Summarizing:

1) The *set of cells* for asynchronous design templates is often distinct from simple Boolean gates and simple flip-flops, frequently employing additional or simply different gates, such as C-elements and/or NCL gates [9], multi-rail pseudo-dynamic gates [10], etc.
2) The *architecture* for asynchronous design templates is often very different, since a new set of rules apply to generate data transformations and to synchronise operations. Fundamentally, the device interconnection rules are substituted to enable implement local handshake operations (see the channel sub-entities description below).
3) The *communication link* for asynchronous design templates is usually defined based on two possible information encoding schemes, one identical to the one used in synchronous data representation and another based on encoding information with some form of delay-insensitivity property, more expensive but more robust than the former scheme.
4) The *protocol* for asynchronous design templates is again widely different from those in synchronous protocols, given the absence of global or semi-global control signals and depending of the encoding scheme choice.

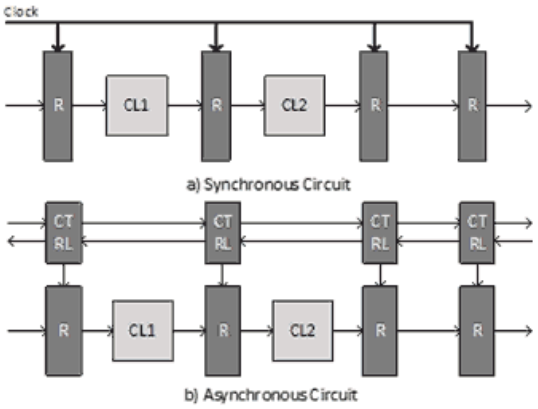


Fig. 2. Simplified linear pipeline circuit structure using (a) synchronous and (b) asynchronous designs. Blocks $CL_i$ represent combinational logic, $R$ represent registers, and $CTRL$ indicates control logic. Adapted from [11].

## III. Asynchronous Circuits Design Principles

Most synchronous circuits rely on the assumption that the value on the inputs of all its registers will only be sampled at the rising (or/and falling) edge of the clock signal. Refer to Figure 2(a) to notice that in a classic linear pipeline this enables to define timing constraints for the maximum delay in combinational logic paths, which must be typically smaller than the clock period. Using synchronous design techniques allows ignoring gate and wire delays, as long as clock timing constraints are respected. In other words, combinational logic is allowed to switch as it computes data during, say, the interval between two consecutive rising clock edges, but the logic outputs must be stable and correct at each such edge. Having this simple model for circuit design is possible only because the clock is a global and periodic signal, *i.e.* its edges only occur at specific and known points in time, and occur simultaneously (this is an assumption) at every point where required. Hence, in synchronous circuits, events will only take place at specific moments; time can thus be treated as a discrete variable.

A look at the alternative, Figure 2(b) shows that in asynchronous circuits there is no such thing as a single clock to simultaneously signal data validity on the inputs of all registers. Here, events can happen at any moment, and time must, quite often, be regarded as a continuous variable. Asynchronous designers rely on local handshake protocols for communication and synchronisation, and on different design templates to build circuits, each with its own specific assumptions about gate and wire delays [5].

Asynchronous design templates can be broadly classified in two main families: bundled-data (BD) [12] and self-timed [13]. Using the DCDT metamodel, the main distinction leading to this classification relies in the DCDT Channel entity, where communication links and protocols differ widely across BD and self-timed templates. Refer to Figure 3 to note that the design of a BD circuit is similar to a synchronous one; the difference is that BD relies on carefully matching the delay of data path combinational logic blocks and controlling registers to the delays in the control block that generates a local clock, rather than employ a single, global clock signal.

Communication and synchronisation in BD circuits are accomplished through some handshake protocol, the more common choices being 4-phase, return to zero (RTZ) protocols [5]. Again using the DCDT concepts, the channel protocol characteristics provide a way to classify asynchronous design templates. Data representation in BD circuits follows the same

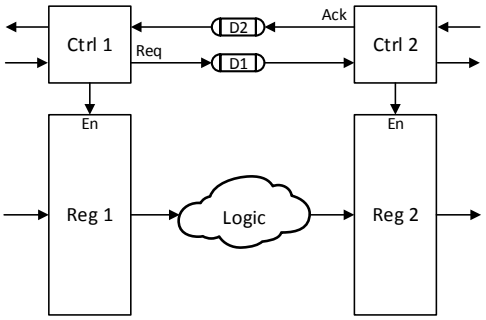


Fig. 3. Example of a typical BD asynchronous pipeline fragment, with delay elements explicitly represented as $D1$ and $D2$ on request ($Req$) and acknowledge ($Ack$) paths on the control part of the circuit; blocks $Ctrli$ represent the local stage controllers. Blocks $Regi$ are data path registers and the $Logic$ cloud represents combinational data processing between pipeline temporal barriers (the registers).

Boolean encoding used in synchronous circuits[1]. This means that according to DCDT BD and synchronous design share a same channel communication link type. Also, unlike what happens in synchronous circuits, controllers are local and usually comprise just a few logic gates. An illustrative extreme example is the very efficient MOUSETRAP pipeline stage controller, which includes only an $XNOR$ logic gate and a 1-bit latch [14]. This simplicity helps, since controllers are replicated at each and every stage of a circuit.

A major hurdle in BD circuit design is how to guarantee that the control and data paths are always precisely delay-matched, since the data and control flows typically run parallel to each other. This is the reason why in Figure 3 delay elements (DEs) are explicitly shown in the request and acknowledge paths. Much research exists to further the design of DEs to achieve working BD circuits. As an illustration of such research efforts, Heck [15] developed a PhD Thesis where the focus was obtaining a single programmable delay element to support the design of asynchronous BD circuits resilient to timing errors. This was in fact the culmination of a joint research between a research group at the University of Southern California in USA and the authors' research group, which had previously generated research results on several aspects of DE design for BD circuits [16]–[19]. Specifically addressed research in these publications are analysis and optimisation of programmable DEs, performance analyses on how fine-grained and coarse-grained delay adjustments work in practice, and to control the effect of voltage variations over the delay-matching characteristic of DEs. Relating to DCDT, DEs are thus part of the design style set of cells to use and affect the architectural rules of asynchronous BD templates.

The required delay-matching design effort is one of the main issues to design robust circuits using the BD family of templates. BD circuit implementations can be as small as an equivalent synchronous implementation, or even smaller, as described for example by Teifel in [20]. However, BD techniques share some of the disadvantages that plague synchronous design techniques, including a potential reduction in circuit robustness to variations, mostly due to the decoupling of control and data parts of the circuit.

### A. Asynchronous Self-Timed Design

A fundamental difference between BD and self-timed design is that the latter relies on data encoding schemes that allow data to carry their own validity information, which enable receivers to compute the presence or absence of data at inputs/outputs, and renders possible the local exchange of information in a mostly delay insensitive way, matching control and data information processing more easily. Because of this characteristic, self-timed circuits can adapt more gracefully to wire and gate delay variations, and are thus one of the best choices to obtain robust circuits. On the negative side self-timed circuits further robustness often at the expense of larger area and/or power overheads. Associating the above discussion to DCDT, it shows that the choice of the channel type to use (the communication link and protocol choices) has deep influence on the design trade-offs of distinct asynchronous and synchronous templates.

Self-timed designs rely on the use of *delay-insensitive* (DI) codes [21]. DI codes use only part of the Boolean encoding spectrum possible over $n$ bits. An $n$-bit code length potentially allows representing $2^n$ distinct *codewords*. A DI code pledges the use of only a subset of these codewords, to obtain the delay insensitivity property for the code. Verhoeff [21] explores the basic details of the theory behind DI codes. These include some codewords to represent valid data, and at least one special (invalid) codeword to represent the absence of data. This codeword is usually called a *spacer*. Since valid codewords and spacer codeword(s) do not comprise all $2^n$ different codewords possible with $n$ bits, it is clear that some codewords are wasted as invalid and the matter of *code efficiency* arises. This is treated in the work of Verhoeff [21], which defines the *rate* $R$ of a code. Given a code with M valid codewords and a length of $n$ bits its rate is $R = (log_2 M)/n$. Of course, $0 \leq R \leq 1$ always holds, and Verhoeff proves that Sperner codes, those where every codeword has as structure $(n \,\mathbf{div}\, 2)$-out-of-n, are DI codes, and such codes provide the highest possible encoding efficiency. For example, if the code length is $n = 20$ bits, all codewords with 10 bits at 1 and 10 bits at 0 are valid Sperner codewords, and there are a total of $184,756$ distinct codewords in this code. Even though this is much less than the $1,048,576$ codewords of a non-DI, 20-bit ordinary Boolean code, this Sperner code is much more efficient than the commonly used dual-rail DI code with length 20 bits that contains only $1,024$ valid codewords. As it can be verified, although as $n \rightarrow \infty$ the rate of Sperner codes tends to 1, practical $n$-bit Sperner codes (and all DI codes) have $R << 1$, while a non-DI code such as the $n$-bit Boolean code has $R = 1$ for any $n$. Under the point of view of DCDT, codes are clearly characteristics defined in the communication link and protocol entities.

To understand how DI codes achieve delay independence, Figure 4 shows the basic communication protocol of a channel as a state transition diagram for transmitting data on a 1-bit DI channel. Clearly, three codewords are necessary and the minimum code length to represent 0, 1 and the spacer is 2. Assume transmission always starts with a spacer (S). A transition from the spacer codeword to 1 (or to 0) characterises the transmission of a valid 1 (resp. 0) and a transition from 1 (resp. 0) to S characterises the removal of data. In other words, DI communication protocols assume there is a spacer between any pair of consecutive data values.

This in fact depicts just a specific family of communication protocols, often associated to DI codes, that can be called *return to spacer* (RTS) protocols. Since the spacer is frequently a code with all bits in 0, a more commonly used term is *return to zero* (RTZ) protocols, although other spacer codewords are sometimes used.

[1]This is not the case for self-timed circuits, as Section III-A details.

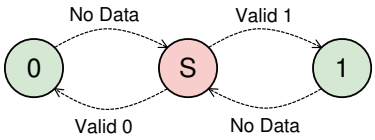


Fig. 4. The basic protocol for transmitting binary data in a DI channel, using a state transition diagram. Here, S stands for the spacer codeword, a bit separation symbol.

Referring back to Figure 2(b) and its relation to the protocol that Figure 4 depicts, asynchronous circuit pipelines can be implemented using one of two approaches: (i) half-buffer, where data and spacers alternate occupying successive pipeline stages; (ii) full-buffer, where all stages can contain data at every moment. Although at first counter-intuitive, since half-buffer schemes seem wasteful, these are more frequently used, because they are simpler to built and in general execute faster than full-buffer schemes. Also it is easier to achieve robust circuits using half-buffer schemes.

In circuit design, often used examples of DI codes are the $k$-of-$n$ codes, where $n$ is the number of wires used to represent data (or its absence) and $k$ is the number of wires that must be at a given logic value for the codeword to represent valid data (usually using 1 value for these wires and 0 values for the others). Albeit different codes are available in the contemporary literature (see e.g. [21]), according to Martin and Nyström [13] the most practical class of DI codes is the 1-of-$n$ (or one-hot), and more specifically the 1-of-2 code. The latter is the basis to form codes to represent any $n$-bit information using two wires to denote each of the $n$ bits, producing the so-called *dual-rail* code. Furthermore, Martin and Nyström argue that DI codes can be coupled to either 2-phase or 4-phase handshake protocols, but 2-phase protocols often lead to more complex circuits. Thus, 4-phase is frequently chosen by self-timed circuit designers. In fact, the majority of self-timed designs available in the state-of-the-art, from networks-on-chip [22], [23], to general purpose processors [24], and network switches [25], primarily rely on 4-phase protocols and dual-rail or 1-of-4-based codes[2]. The 1-of-4 code is equivalent to two 1-of-2 codes considered together, but these codes are different. In fact, switching a 1-of-4 codeword (say 0100, corresponding to decimal 2) to a 0000 spacer implies switching just 1 bit, while the same value encoded in dual-rail, 1001 (equivalent to 10 in binary or decimal 2), requires two bits to switch to reach the same spacer. Thus 1-of-4 codes present roughly a 50% switching power advantage over dual-rail encoding.

Figure 5(a) depicts a 4-phase dual-rail DI channel *D*, where a single bit datum is represented using two wires, *D.0* and *D.1* that together carry the datum value, and one signal *ack* to control data flow [3]. A spacer is encoded here as a codeword with all wires at 0. Valid data are encoded using exactly one wire at 1, *D.1*=1 for a logic 1 and *D.0*=1 for a logic 0. In this case, both wires at 1 is a codeword that does not correspond to any valid datum and is not used. Figure 5(b) shows an example of data transmission using this convention to demonstrate the control flow allowed by the *ack* wire combined to codewords represented in wires *D.1* and *D.0*. In this example, a sender provides dual-rail data in *D.1* and *D.0* to a receiver

[2]Note that 1-of-2 (resp. dual-rail) and 1-of-4 codes have the same rate, $R = 1/2 = 0.5$, while a 1-of-8 code has a rate of just $R = 3/8 = 0.375$ and is accordingly never used.

[3]Some works prefer the use of *true* and *false* suffixes instead of 0 and 1 to distinguish between the two wires of a 1-of-2 or dual-rail code, which would lead e.g. to the terminology $D_f$ and $D_t$ in place of *D.0* and *D.1*, respectively.

| Wire | Spacer | Value 0 | Value 1 |
|---|---|---|---|
| D.0 | 0 | 1 | 0 |
| D.1 | 0 | 0 | 1 |

(a)


(b)

Fig. 5. The RTZ, dual-rail channel operation: (a) data encoding; (b) example of data transmission waveform.

| Wire | Spacer | Value 0 | Value 1 |
|---|---|---|---|
| D.0 | 1 | 0 | 1 |
| D.1 | 1 | 1 | 0 |

(a)


(b)

Fig. 6. The RTO, dual-rail channel operation: (a) data encoding; (b) example of data transmission waveform.

that acknowledges received data through *ack*. Communication starts with a spacer, all signals at 0. Note that the *ack* wire also starts at 0, signaling the receiving side is ready to receive new data. Next, the sender puts a valid 0 bit in the channel, by raising the logic value of *D.0*, which is acknowledged by the receiver raising the *ack* wire. After the sender receives *ack*, it produces a spacer to end communication, bringing all data signals in the channel back to 0. The receiver then lowers its *ack* signal, after which another communication can take place. Due to its nature, which requires all signals to go to 0 before each new data transmission starts, this justifies the return-to-zero (RTZ) denomination for this protocol.

Another protocol for dual-rail self-timed designs is the return-to-one (RTO) protocol [26]. RTO is similar to RTZ, but its data values are inverted compared to the latter. As Figure 6(a) shows, a spacer here is the codeword with all wires at 1 and valid data is represented by one wire at 0, *D.1*=0 for a logic 1 and *D.0*=0 for a logic 0. Figure 6(b) depicts an example RTO data transmission, which starts with all wires at 1 in the data channel. As soon as the sender puts valid data in the channel, the receiver may acknowledge it by lowering *ack*. Next, all data wires must return to 1 to denote a spacer, ending transmission. When the spacer is detected by the receiver, it raises the *ack* signal and new data can follow. The idea behind the RTO protocol is simple but powerful and allows a better design space exploration for QDI circuits, enabling optimisations in power [27] and robustness [28]. Furthermore, as demonstrated in [29], RTZ and RTO can be mixed in a same dual-rail design and the conversion of values between them requires only an inverter per wire. According to Martin and Nyström [13], such conversion is DI and does not compromise the robust functionality of a self-timed circuit. This tutorial refers to signals operating under the RTZ (RTO) protocol as RTZ (RTO) signals.

A seasoned reader might have come across the term *quasi-delay-insensitive* (QDI) being used extensively, where here the term self-timed circuits is used. It is useful to draw attention to the distinction between these terms. QDI is in fact a large and important subset of self-timed circuits; but QDI circuits are self-timed circuits with few timing assumptions. In fact, there is even a class of self-timed circuits with no timing assumption called *delay insensitive* (DI) circuits; they are the most robust of all ways to design circuits, where any delay of any wire or any gate (or other logic components) is irrelevant to define the overall circuit functionality, meaning the functionality of the design is fully insensitive to delays in wires or gates. Unfortunately, this *ideal* class of designs was proven to be

limited, too limited to be of practical use [30]. QDI offers a compromise that can produce a set of design techniques that are expressive enough to be used in the construction of any digital circuit and be *mostly* or quasi-delay-insensitive. This compromise consists in constraining some selected wire forks in a design to be *isochronic*. Assume a wire fork has a source terminal and two sink terminals. Saying this fork is isochronic basically means the propagation time from the source terminal to both sink terminals can only differ by a negligible amount. The isochronic fork assumptions defines the QDI paradigm within the broader self-timed design style discussed here. An early theoretical result showed QDI design is Turing-complete, unlike the DI design paradigm [31], which opens the door to use QDI as a useful class of circuit design techniques.

Of course, the simply stated isochronic fork constraint can be hard to ensure, specially in large circuits. Furthermore, the strictness of limiting the timing assumptions to wire delays often comes with an area and performance impact. Also, in some cases the robustness requirements are not so strict. Others self-timed design styles [32], [33] exist with more relaxed limitations on timing assumptions to offer better trade offs among robustness, performance, area and power.

## IV. Asynchronous Design Tools and Systems

This Section explores some state-of-the-art asynchronous design tools. They can be classified as either event- or channel-driven tools. Event-driven design tools often operate at the individual signal transition level. They are suited to design small scale components, such as asynchronous state machines or controllers with limited amount of logic gates, usually containing in the order of dozens or hundreds of gates, like those previously discussed for BD design templates. Conversely, a channel based design tool aims at design larger scale, complex circuits, counting thousands or more gates. These two approaches are complementary; circuits implemented with an event-driven design tool can form components and controllers integrated by a channel-driven design tool to produce larger circuits.

### A. Event-Driven Design Tools

A traditional event-driven approach is to model the circuit as Petri nets [34]. Figure 7 depicts an example of a Petri net; these are directional bipartite graphs where nodes are of two types: places and transitions. In a Petri net places hold tokens and transitions move tokens between places. When a transition *fires*, it consumes a token from each of its predecessor places and creates a new token in every one of its successor places. A transition can only fire if there is at least one token at each of its input places. Of course, the firing rules guarantee that the number of tokens in the net varies along the net operation. Petri nets are excellent models to capture both concurrency and causality.

A signal transition graph (STG) is a Petri Net where transitions capture signal transitions in a circuit, e.g. a rising or falling signal transition. STGs are used to specify the behaviour of asynchronous circuits and their environment; this specification can be used to both produce a circuit and also to verify if an existing circuit correctly implements it. Petrify [36] is a tool capable of synthesising a circuit to the netlist level from an STG specification. However, Petrify does not provide a friendly environment for designing, analysing and simulating STG specifications.

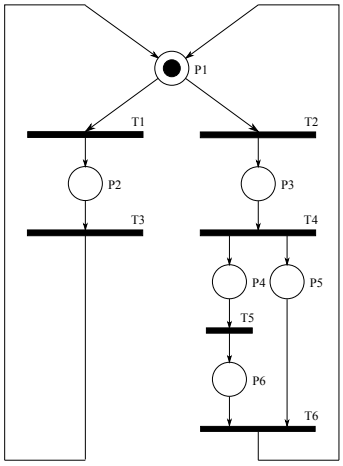


Fig. 7. Simple reversible, lively, deadlock-free Petri net with 6 places, 6 transitions and an initial marking [35].

Workcraft [37]–[39] builds on top of Petrify to offer a more polished framework to design and verify asynchronous circuits from event-driven specifications. It enable designers to specify, test, formally verify and synthesise circuits from their specifications. Workcraft modelling capabilities is not limited to STG, it also allows designers to model circuits using burst mode finite state machines [40].

However, the application scope of Workcraft as a event-driven design tool is limited to controllers and other small circuits. For instance, it requires the designer to manually insert the reset signal and loop breakers in their circuit specification after synthesis; this and the computation-intensive nature of event-driven synthesis highlights the need for the higher abstraction level of channel-driven design tools to curb design complexity in devising bigger and more complex circuits.

### B. Channel-Driven Design Tools

Channel-driven design tools can be further classified based on the design template of the circuits they produce; they can be either BD or self-timed. BD often requires less area and switching activity, but rely on establishing strict timing assumptions on the timing paths to reach computation completion, and large delay margins to cope with delay variations. Self-timed circuits in turn employ DI codes and completion detection circuitry to explicitly define computation completion; however, the additional area overhead and the increased switching activity of DI codes can be prohibitive for many applications. Nonetheless, self-timed circuits are more resilient to delay variations, thus enabling the use of aggressive voltage scaling.

A possible approach for bundled-data design is desynchronisation [41], i.e. synthesising a conventional RTL-described synchronous circuit and later replacing the clock tree with asynchronous controllers that generate a local clock to register groups. These controllers use a *request* signal that is delay-matched to the data propagation paths, marking data availability to the asynchronous controller responsible for generating clock pulses to registers or latches. Figure 8 shows the start and end of applying desynchronisation during design.

A tool following a similar approach using latch-based design is Blade [42]. The generic template structure for a Blade pipeline stage is illustrated in Figure 9. Blade is a template for circuits resilient to timing errors. Note in the upper part

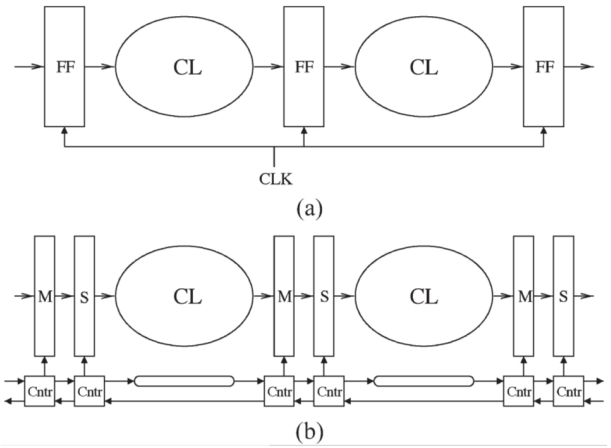


Fig. 8. The desynchronization asynchronous design template [41]: (a) after a first synthesis, pipeline stages are clocked circuits that use flip-flop based registers; (b) desynchronization changes registers by master-slave latches commanded by asynchronous controllers, which communicate through local handshakes.

of the Figure the forward and backward stages handshake signals, and the local asynchronous controller. Note also the error detecting latch (EDL), used to store stage data obtained after processing using conventional combinational logic.

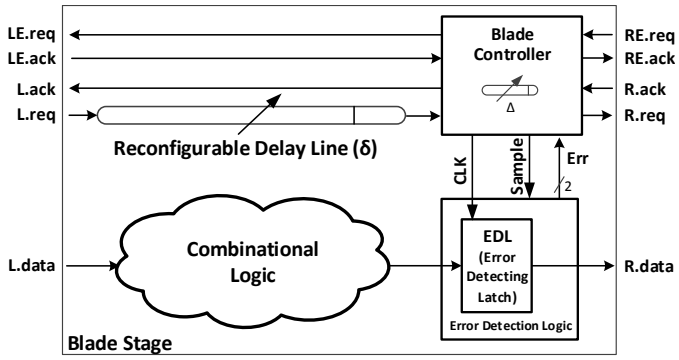


Fig. 9. The Blade stage circuit template [42].

One of the main features of Blade is the error detection logic, containing the EDL and additional circuits that enable on-the-fly data errors detection and correction. The proposed EDL structure is depicted in Figure 10. Note that the error signal (Err1/Err0) is a dual-rail signal, generated by a special component called Q-flop, which contains a metastability filter. When metastability does occur, the Q-flop keeps its output in the no-data state (Err0=Err1=0) until it resolves. When no metastability occurs or when it resolves, Err0 is always distinct from Err1, indicating either a timing error (Err0=0, Err1=1) or no timing error (Err0=1, Err1=0).

In Blade, after circuit synthesis using conventional tools (e.g. Cadence Genus), timing paths are analysed and asynchronous controllers and matching delay lines are inserted. Blade employs controllers capable of error recovery on critical timing paths. This requires the use of complex, reconfigurable delay lines to achieve a better average performance ($\delta$ and $\Delta$ in Figure 9). When timing violations occur on critical paths, the error recovering controller is able to intervene and correct operation. This enables Blade to reduce the timing margins and significantly increase circuit resilience to delay variations.

However, the degree of resilience provided by Blade is debatable when compared to self-timed circuits. As previously discussed, self-timed circuits are naturally more resilient to delay variations. A noteworthy self-timed template is *Null Convention Logic* (NCL) [43]. It relies on hysteretic threshold gates to implement quasi-delay-insensitive (QDI) logic blocks.

Uncle [44] is another synthesis system proposed to implement asynchronous NCL circuits, available as open-source. It uses specially constructed Verilog RTL templates as input description format. Uncle relies on standard EDA tools to synthesise RTL code to a netlist of *virtual logic gates*. A custom tool within Uncle performs dual-rail expansion[4]; it replaces virtual logic gates with their NCL equivalent, implements the completion detection logic and performs some NCL-specific logic optimisations. However, such logic optimisations are limited when compared to the optimisations performed by timing driven technology mapping tools in commercial EDA tools. Furthermore, Uncle does not allow setting a performance target to trade e.g. performance power and area goals for the circuit.

The Spatially Distributed Dual-Spacer Null Convention Logic (SDDS-NCL) asynchronous design template [45], [46] is an evolution of NCL that enables the use of industry-standard EDA tools to synthesise self-timed NCL-based logic from *Boolean virtual functions*. Conventional EDA tools were shown to require both positive and negative unate gates to successfully perform unconstrained technology mapping for asynchronous circuits synthesis [46]. SDDS-NCL uses a combination of conventional NCL threshold gates (which are all positive unate) and the dual, negative unate, NCLP threshold gates to accommodate signal inversions from negative unate gates. Signal inversions swap protocols between RTZ and RTO in successive circuit logic stages; NCL gates operate using RTZ and the active-low NCLP operates correctly on RTO. For every NCL gate that correctly works with the RTZ protocol, there is functionally equivalent[5] NCLP gate that works with the RTO protocol. As long as the virtual function being realised is unate, a simple graph colouring algorithm can correctly mark regions where the protocol is RTZ and RTO to swap between functionally equivalent NCL and NCLP gates and to guarantee correct circuit operation.

Pulsar [47], [48] is an open-source [49] design framework for synthesising constrained self-timed circuit from

[4]Simply stated, *dual-rail expansion* is a process that transforms a circuit where each wire carries a single bit into a new circuit where each bit is represented by two wires using a dual-rail code.

[5]Functionally equivalent gates are gates that present the same virtual function

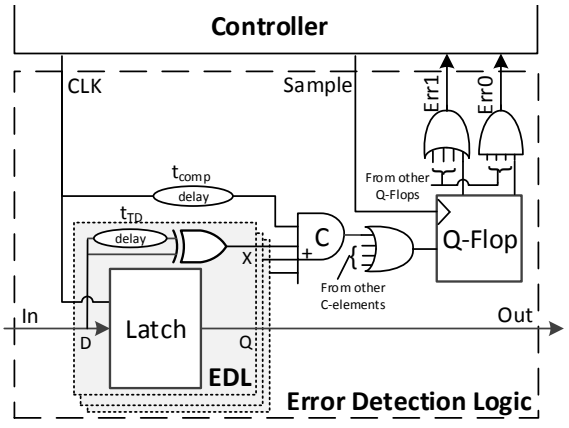


Fig. 10. Blade error detection logic, including a block level diagram of the EDL [42].

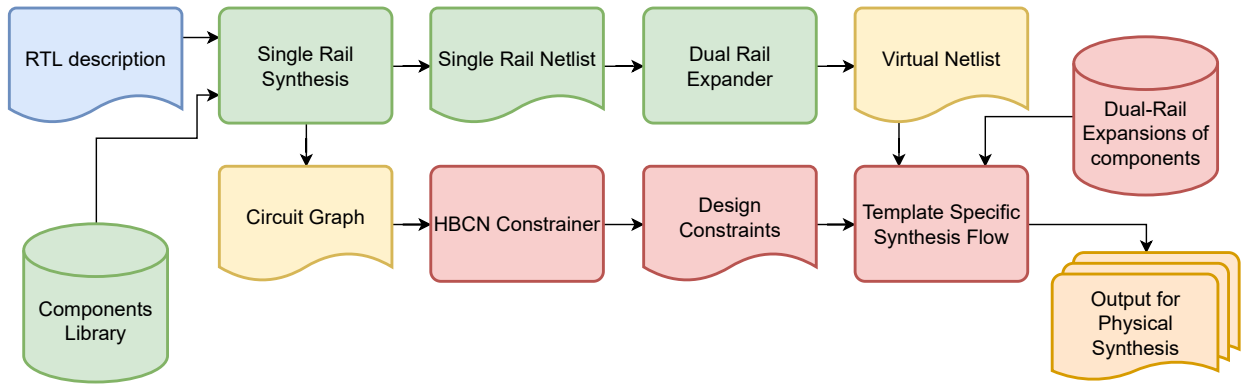


Fig. 11. The Pulsar flow. In blue is the RTL-Like user input in (System)Verilog or VHDL; yellow items are either third party (commercial) tools or conventional output of such tools; green items comprise the front-end synthesis flow, while red items are components of the back-end synthesis flow.

an RTL-like input description. It synthesises and optimises QDI circuits under cycle time constraints, and allows trading off performance and power targets. Pulsar can synthesise SDDS-NCL circuits, but it is not limited to this template. The timing constraining in Pulsar is achieved using a combination of modelling techniques: (i) the pseudo-synchronous Weak-Conditioned Half-Buffer (WCHB) temporal barrier model [47], [50]; and (ii) the Half-Buffer Channel Network (HBCN) timing model [47]. Technique (i) allows using standard static timing analysis (STA) tools to compute the propagation paths in asynchronous pipelines, and additionally to accurately simulate synthesised circuits; Technique (ii) enables the performance analysis of complex non-linear QDI circuits. The Pulsar synthesis flow is depicted in Figure 11. It comprises a back-end and a front-end synthesis flows. The front-end is responsible for the template-independent design capture and generation of the intermediate virtual netlist. Pulsar design capture methodology shares similarities with the Uncle synthesis tool [51], as it uses an especially crafted RTL descriptions and traditional EDA tools to synthesise a single-rail netlist. The front-end synthesises the RTL-like description using Cadence Genus to produce a single-rail netlist of *components*. These components are defined in the *components library*, presented as Liberty files to the front-end synthesis.

Contrasting to Uncle, which relies on its own tool to perform dual-rail expansion, Pulsar uses Genus to perform the dual-rail expansion in the back-end synthesis. It models channels as SystemVerilog interfaces [52]. A simple tool replaces every wire in the single-rail netlist with a channel to create the *virtual netlist*. This virtual netlist is the input to the template-dependent back-end synthesis flow. Each component on the virtual netlist has an equivalent SystemVerilog module defining its template-dependent dual-rail expansion. Channels interconnect these modules implementing the dual-rail expansion of components. The channel abstraction is also used to construct the acknowledgement network during synthesis by cleverly employing wired AND (`wand`) and wired OR (`wor`) net types.

The Pulsar flow also constructs the HBCN and automatically creates cycle time constraints for synthesis. Concurrent to the creation of the virtual netlist, cycle time constraints are computed. The scripts used for the single-rail synthesis produce a *structural graph* describing the pipeline topology. This circuit graph is used to model the HBCN of the expanded circuit and to compute the path constrains.

## V. Conclusions

The panorama this work provides highlights the field of digital circuit design is open to accept novel techniques to deal with the increasing complexity brought about by new technologies.

Asynchronous design is dominated by few engineers today, but finds use in multiple niche applications where it constitutes an invaluable resource. A few example fields where asynchronous techniques already find ample use are:

1) Interface design - Guaranteeing the achievement of correct timing in the communication between two distinctly designed complex circuit modules can become a nightmare. Asynchronous circuit design is an good way to streamline module communication and adjust data rate exchanges, independently of the often found modules clock frequency mismatches. As an example, see Sokolov et al. that propose a library of specialised analogue-to-asynchronous components to interface analogue and asynchronous modules in [53]. In another work, an industry giant, Intel, recently suggested four-pin input output (FPIO), an asynchronous delay-insensitive protocol for bit-serial multi-chip management protocol to substitute with advantages the well-known SPI synchronous protocol [54].
2) Intrachip communication - The increasing number of modules inside a chip requires changing traditional bus-based communication by networks on chip (NoCs). Asynchronous NoCs are a way to cope with the control of long wires employed for long range reliable communication inside chips. This is especially true in advanced technology nodes, such as Thonnart et al. describe in [55].
3) Secure data communication - Security is an increasing concern in data exchange. The use of synchronous design is known to provide mostly leaky ways to communicate data. Today, cryptography is mandatory in the exchange of information, in practically every system. The leakage of information in synchronous circuits can be used by system attackers to obtain knowledge about the system, including access to cryptographic keys with already observed catastrophic consequences. Side channels attacks (SCAs) can find cryptographic keys by exploring electrical or electromagnetic emissions, which in the case of clocked systems is mandatorily periodic, facilitating analysis significantly. Regarding use cases for cryptographic asynchronous circuits, see e.g. the work of Ho et al. in [56] that propose an asynchronous implementation of a NoC router with characteristics of resistance to

differential power analysis (DPA) attacks. In another effort, related to the efficiency of cryptography, Li et al. propose a 3.6 Gbps throughput implementation of a SHA-256 cryptographic module, in [57]. The module relies on the design of a very efficient asynchronous FIFO.

The final goal of this tutorial consists in encouraging readers to explore the often daunting amount of new information regarding the design of asynchronous circuits. Accordingly, Authors provide several pointers to good references where to begin learning about how designing digital circuits using asynchronous techniques can become feasible. The DCDT metamodel is a simple and formal tool to undertake systematic explorations and to understand asynchronous templates and their use in circuit design.

## ACKNOWLEDGEMENTS

This research was partially funded by CAPES and CNPq (grant no. 312917/2018-0), Brazilian research funding organisations.

## REFERENCES

[1] P.-C. Shen, C. Su, Y. Lin, A.-S. Chou, C.-C. Cheng, J.-H. Park, M.-H. Chiu, A.-Y. Lu, H.-L. Tang, M. M. Tavakoli, G. Pitner, X. Ji, Z. Cai, N. Mao, J. Wang, V. Tung, J. Li, J. Bokor, A. Zettl, C.-I. Wu, T. Palacios, L.-J. Li, and J. Kong, "Ultralow contact resistance between semimetal and monolayer semiconductors," *Nature*, vol. 593, no. 7858, pp. 211–217, 13 May 2021.

[2] J. Lu, W.-K. Chow, and C.-W. Sham, "Fast Power- and Slew-Aware Gated Clock Tree Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 11, pp. 2094–2103, Nov 2012.

[3] S. Kim, I. Kwon, D. Fick, M. Kim, Y.-P. Chen, and D. Sylvester, "Razor-lite: A side-channel error-detection register for timing-margin recovery in 45nm SOI CMOS," in *IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2013, pp. 264–265.

[4] K. Bowman, J. Tschanz, N. S. Kim, J. Lee, C. Wilkerson, S. Lu, T. Karnik, and V. De, "Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 49–63, Jan 2009.

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

[6] N. L. V. Calazans, T. A. Rodolfo, and M. L. L. Sartori, "Robust and Energy-Efficient Hardware:The Case for Asynchronous Design," *Journal of Integrated Circuits and Systems*, vol. 16, no. 2, pp. 1–11, 2021.

[7] M. T. Moreira, "Asynchronous Circuits: Innovations in Components, Cell Libraries and Design Templates," Ph.D. dissertation, Faculty of Computer Science, PUCRS, 2016.

[8] D. Harris, *Skew-tolerant Circuit Design*. Morgan Kaufmann, 2001.

[9] K. M. Fant, *Logically Determined Design*. Hoboken, NJ, USA: Wiley-Interscience, 2005.

[10] P. Beerel, G. Dimou, and A. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design and Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[11] J. Sparsø, *Introduction to Asynchronous Circuit Design*. Independently published, 2020. [Online]. Available: https://orbit.dtu.dk/en/publications/introduction-to-asynchronous-circuit-design.

[12] I. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[13] A. Martin and M. Nyström, "Asynchronous Techniques for System-on-Chip Design," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, 2006.

[14] M. Singh and S. M. Nowick, "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, Jun. 2007.

[15] G. Heck, "The Impact of Voltage Scaling over Delay Elements with Focus on Post-Silicon Tests," Ph.D. dissertation, PPGCC - FACIN - Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Mar. 2018.

[16] A. Singhvi, M. T. Moreira, R. Tadros, N. L. V. Calazans, and P. A. Beerel, "A Fine-Grained, Uniform, Energy-Efficient Delay Element for 2-Phase Bundled-Data Circuits," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 2, pp. 1–23, Jan. 2017.

[17] R. Tadros, W. Hua, M. Gibiluka, M. T. Moreira, N. L. V. Calazans, and P. A. Beerel, "Analysis and Design of Delay Lines for Dynamic Voltage Scaling Applications," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2016, pp. 11–18.

[18] A. Singhvi, M. T. Moreira, R. Tadros, N. L. V. Calazans, and P. A. Beerel, "A Fine-Grained, Uniform, Energy-Efficient Delay Element for FD-SOI Technologies," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2015, pp. 27–32.

[19] G. Heck, L. Heck, A. Singhvi, M. T. Moreira, P. Beerel, and N. L. V. Calazans, "Analysis and Optimization of Programmable Delay Elements for 2-Phase Bundled-Data Circuits," in *International Conference on VLSI Design (VLSID)*, Jan. 2015, pp. 321–326.

[20] J. Teifel, "Asynchronous Cryptographic Hardware Design," in *Annual IEEE International Carnahan Conference on Security Technology (ICCST)*, Oct. 2006, pp. 221–227.

[21] T. Verhoeff, "Delay-insensitive Codes - An Overview," *Distributed Computing*, vol. 3, no. 1, pp. 1–8, 1988.

[22] E. Beigné, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An Asynchronous NOC Architecture Providing Low Latency Service and its Multi-Level Design Framework," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2005, pp. 54–63.

[23] J. Pontes, M. Moreira, F. Moraes, and N. Calazans, "Hermes-AA: A 65nm asynchronous NoC router with adaptive routing," in *IEEE International System on Chip Conference (SoCC)*, Sep. 2010, pp. 493–498.

[24] A. Martin, M. Nyström, and C. Wong, "Three Generations of Asynchronous Microprocessors," *IEEE Design and Test of Computers*, vol. 20, no. 6, pp. 9–17, 2003.

[25] M. Davies, A. Lines, J. Dama, A. Gravel, R. Southworth, G. Dimou, and P. Beerel, "A 72-Port 10G Ethernet Switch/Router Using Quasi-Delay-Insensitive Asynchronous Design," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2014, pp. 103–104.

[26] M. Moreira, R. Guazzelli, and N. Calazans, "Return-to-One Protocol for Reducing Static Power in QDI Circuits Employing m-of-n Codes," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2012.

[27] M. T. Moreira, R. A. Guazzelli, and N. L. V. Calazans, "Return-to-One DIMS Logic on 4-phase m-of-n Asynchronous Circuits," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2012, pp. 669–672.

[28] M. Moreira, R. Guazzelli, G. Heck, and N. Calazans, "Hardening QDI Circuits Against Transient Faults Using Delay-insensitive Maxterm Synthesis," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2014, pp. 3–8.

[29] M. Moreira, J. Pontes, and N. Calazans, "Tradeoffs between RTO and RTZ in WCHB QDI Asynchronous Design," in *International Symposium on Quality Electronic Design (ISQED)*, March 2014, pp. 692–699.

[30] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *6th MIT Conference on Advanced Research in VLSI (AUSCRYPT)*, 1990, pp. 263–278.

[31] R. Manohar and A. J. Martin, "Quasi-Delay-Insensitive Circuits Are Turing-Complete," California Institute of Technology - CalTech, USA, Tech. Rep., 1995, rVM33.

[32] C. Brej, "Early Output Logic and Anti-tokens," Ph.D. dissertation, The University of Manchester (United Kingdom), 2005.

[33] C. Brej and D. Edwards, "Forward and Backward Guarding in Early Output Logic," in *IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2009, pp. 226–229.

[34] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[35] C. Schweiger, 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:Petrinetz.svg

[36] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on information and Systems*, vol. 80, no. 3, pp. 315–325, 1997.

[37] I. Poliakov, D. Sokolov, and A. Mokhov, "Workcraft: a static data flow structure editing, visualisation and analysis tool," in *International Conference on Application and Theory of Petri Nets*. Springer, 2007, pp. 505–514.

[38] I. Poliakov, V. Khomenko, and A. Yakovlev, "Workcraft–a framework for interpreted graph models," in *International Conference on Applications and Theory of Petri Nets*. Springer, 2009, pp. 333–342.

[39] D. Sokolov, V. Khomenko, and A. Mokhov, "Workcraft: Ten years later," *This asynchronous world. Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pp. 269–293, 2016.

[40] A. Chan, D. Sokolov, V. Khomenko, D. Lloyd, and A. Yakovlev, "Burst automaton: Framework for speed-independent synthesis using burst-mode specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022.

[41] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.

[42] D. Hand, M. T. Moreira, H. H., D. Chen, F. Butzke, M. Gibiluka, M. Breuer, N. L. V. Calazans, and P. A. Beerel, "Blade - A Timing Violation Resilient Asynchronous Template," in *IEEE International*

*Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2015, pp. 21–28.

[43] K. M. Fant and S. A. Brandt, "NULL Convention Logic$^{TM}$: A complete and consistent logic for asynchronous digital circuit synthesis," in *International Conference on Application Specific Systems, Architectures and Processors (ASAP)*, Aug. 1996, pp. 261–273.

[44] R. Reese, S. Smith, and M. Thornton, "Uncle - An RTL Approach to Asynchronous Design," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2012, pp. 65–72.

[45] M. T. Moreira, G. Trojan, F. G. Moraes, and N. L. V. Calazans, "Spatially Distributed Dual-Spacer Null Convention Logic Design," *Journal of Low Power Electronics*, vol. 10, no. 3, pp. 313–320, 2014.

[46] M. T. Moreira, P. A. Beerel, M. L. L. Sartori, and N. L. V. Calazans, "NCL Synthesis With Conventional EDA Tools: Technology Mapping and Optimization," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 6, pp. 1981–1993, 2018.

[47] M. L. L. Sartori, R. N. Wuerdig, M. T. Moreira, and N. L. V. Calazans, "Pulsar: Constraining QDI Circuits Cycle Time Using Traditional EDA Tools," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2019, pp. 114–123.

[48] M. L. L. Sartori, M. T. Moreira, and N. L. V. Calazans, "A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2020, pp. 114–123.

[49] ——, "Pulsar - A Flow to Support the Design of QDI Asynchronous Circuits," Jun. 2020. [Online]. Available: https://github.com/marlls1989/pulsar

[50] Y. Thonnart, E. Beigné, and P. Vivet, "A pseudo-synchronous Implementation Flow for WCHB QDI Asynchronous Circuits," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2012, pp. 73–80.

[51] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle - An RTL Approach to Asynchronous Design," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2012, pp. 65–72.

[52] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*, 2nd ed. Springer Science & Business Media, 2006.

[53] D. Sokolov, V. Khomenko, A. Mokhov, V. Dubikhin, D. Lloyd, and A. Yakovlev, "Automating the Design of Asynchronous Logic Control for AMS Electronics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 5, pp. 952–965, 2020.

[54] A. Lines, "Asynchronous Serial Infrastructure Using FPIO," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2021, pp. 62–63.

[55] Y. Thonnart, P. Vivet, S. Agarwal, and R. Chauhan, "Latency Improvement of an Industrial SoC System Interconnect using an Asynchronous NoC Backbone," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, May 2019, pp. 46–47.

[56] W.-G. Ho, N. K. Z. Lwin, N. A. Kyaw, J.-S. Ng, J. Chen, K.-S. Chong, B.-H. Gwee, and J. S. Chang, "Asynchronous Serial Infrastructure Using FPIO," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, Oct. 2020, pp. 1–5.

[57] J. Li, Z. He, and Y. Qin, "Design of Asynchronous High Throughput SHA-256 Hardware Accelerator in 40nm CMOS," in *IEEE 13th International Conference on ASIC (ASICON)*, Oct. 2019, pp. 1–4.