Components to Support Choice in Self-Timed Asynchronous Design Flows

Marcos Luiggi Lemos Sartori*, Willian Analdo Nunes*, Ney Laert Vilar Calazans†
*GAPH Research Group – PPGCC – School of Technology – PUCRS – Porto Alegre – RS – Brazil
†GME – PGMICRO – Federal University of Rio Grande do Sul – UFRGS – Porto Alegre – RS – Brazil
{marcos.sartori,willian.nunes}@edu.pucrs.br, ney.calazans@inf.ufrgs.br

***Abstract*—The design of digital circuits on recent technologies brings several challenges, among which robustness to variations stands out. Variation sources are multiple, and the evolution of integrated circuit fabrication techniques increases the number and relevance of such sources, and the complexity of ensuring circuit robustness against them. Some design paradigms naturally counter variations of one or more types. Asynchronous self-timed design is one such paradigm that can provide robustness to process, voltage, temperature, ageing and IR drop variations, to cite some of the main types. This paper proposes an enhancement to the Pulsar environment, a recently proposed open source automated flow for the design of self-timed clockless circuits. The six components proposed here enable describing choices and decisions on the flow of data tokens inside asynchronous circuits. Design capture in Pulsar can then employ these. To implement the abstract (synthesis-enabled) components, the paper also brings the proposal of the handshaking mutex, a versatile complex gate that eases the design of `probe` and `arbiter`, the two most complex among the new components. Results demonstrate the new version of Pulsar is more powerful than the previous, baseline, version, enabling the design capture and the automated synthesis steps of more complex asynchronous self-timed circuits. They also indicate the handshaking mutex operates correctly, and with a good level of attested fairness.**

***Index Terms*—Digital circuits, Asynchronous design, Dual-rail circuits, Electronic design automation, Automated synthesis.**

I. Introduction and Related Work

The continuous evolution of integrated circuit (IC) technologies into nanometer scales brings uncertainties with it. These are related to several aspects of the IC manufacturing process and of the environment where circuits operate. Uncertainties indeed affect the circuit design process. Increasing manufacturing process variations have to be more carefully considered, together with environmental conditions such as voltage supply and temperature variations [1], maybe even added with accounting for ionising radiation incidence during circuit operation [2]. Besides, design concerns such as IR drop [3][1] and long wire delays become more relevant. IC lifetime can also be negatively affected by technology evolution, bringing the need to consider ageing-aware design techniques [1].

[1]IR drop refers to a supply voltage drop that appears at the resistive component of any impedance. The acronym IR comes from the traditional Ohm´s Law $V = IR$ equation. IR drop is the electrical potential difference (i.e. a voltage value) between the two ends of a conducting phase during a current flow. This voltage drop across any resistance is the product of current (I) passing through it and the resistance value (R). In VLSI design it is known, for example, that the supply voltage drops from outer regions to inner regions of an IC die due to the IR drop effect. It is not uncommon that such drops reach 10% or more of the original voltage supply.

Synchronous design techniques prevail in most of digital circuit domains, but they are increasingly stressed by the uncertainties in the process of creating circuits robust to the large amount of possible variations. As a consequence, sophisticated techniques arise to keep the synchronous paradigm usable. For example, Jain et al. [4] propose highly sophisticated design techniques to implement a clock network reconfiguration scheme. This enables operation of synchronous circuits from nominal voltage down to deep sub-threshold voltages. If alternative design techniques can guarantee automatic adaptability of the circuit to the level of voltage scaling, the overhead caused by such techniques can be avoided. As an example, Sartori et al. in [5] describe an experiment towards such an alternative, where an asynchronous class of design techniques called self-timed design demonstrates to be resilient to voltage scaling, allowing correct operation of circuits over a wide range of supply voltages, from nominal to sub-threshold.

A. Asynchronous and Self-Timed Circuits

A *synchronous* circuit relies on a *global clock* signal to provide a single discrete common time reference. The clock is typically a wave with a period greater than the worst combinational logic delay in any path in the circuit between two consecutive temporal barriers, usually clocked registers. All circuit registers simultaneously capture data at every clock transition. This guarantees designers can ignore wire and gate delays during several of the design phases, and circuit timing computations are expedite.

Asynchronous circuits have no global clock, and rely solely on local handshakes for every data exchange within the circuit. The synchronous global clock distribution network no longer needs to exist. If local handshakes are systematically organised, only local timing constraints need to be reinforced and variations effects have a more restricted scope where they actuate within the circuit. This property of asynchronous circuits brings the potential advantage over synchronous design for dealing with variations and ageing [6].

Asynchronous design methods can be split into two large classes, depending on how local handshakes are organised into a design style: (i) *bundled-data* (BD) design assumes the existence of local controllers, one for each logic stage, and which interact through traditional asynchronous request/acknowledge signals. These signals are delay-matched to the data processing part of the hardware, usually using delay elements interposed among control signals. BD design is thus very similar to synchronous design; they are easy to generate and have area cost in the same order as equivalent synchronous circuits; (ii) *self-timed* (ST) design employs data representation schemes based

Digital Object Identifier 10.29292/jics.v18i1.671

on delay-insensitive (DI) codes, which enable to represent data presence unequivocally without using separate control signals for attesting validity. Thus, instead of using asynchronous controllers, data itself carries control information and control signals can be produced directly from the structure of the data, if needed, such as in the case of backward acknowledge signals.

Asynchronous design is naturally more robust than synchronous design to delay variations and can better handle process, voltage and temperature (PVT) variations, due to the use of local handshakes, which reduces the uncertainty of controlling the timing of long wires distributed along the whole circuit. ST design is potentially more robust in the same aspects than BD design, because DI representations integrate data and control information in a single entity.

Unfortunately, designing ST circuits is often a laborious manual work, requiring detailed knowledge on convoluted specific design techniques. Also, ST designs are frequently handcrafted cell by cell, impairing adoption in larger scale. Synchronous design has coped with technology scaling to nanometer tens or even units. Accordingly, the last decades saw little interest from traditional electronic design automation (EDA) vendors and IC manufacturers in supporting ST design. Thus, asynchronous design automation is still crawling when compared to what synchronous designers have available.

B. A Few Related Works

Despite not being in widespread use, ST design occupies niches in areas like security [7] and high speed circuits [8]. Often, given a promising application, a new ST design style is devised and a specific set of tools is built to support it [9]. Examples of tools and flows proposed recently are Balsa [10], Teak [11], Uncle [12] and Proteus [13].

The literature reveals that Proteus, Uncle and Pulsar [14] are works that got the closest to leveraging traditional EDA, design capture models and methods for use in ST design. Uncle provides a way to use traditional EDA for design capture and limited logical optimisation, relying on custom software for technology mapping and specialised optimisations, e.g. relaxation, retiming, cell merging and net buffering. However, Uncle cannot take full advantage of seasoned synthesis and logic optimisation algorithms, mostly because the cells it instantiates are not modelled according to the specifications expected by traditional tools. Proteus counts with a sophisticated frontend flow, where asynchronous channels are modelled using SystemVerilog and design capture relies on a communicating sequential processes (CSP) model. Compared to Uncle, Proteus targets an even more specific set of cells. These are implemented as dynamic domino logic gates, limiting its use for broader ranges of ST design templates. Pulsar is explored next, in Sections I-C and II.

C. Contributions of this Work

Most ST circuit design methods do not rely on traditional EDA tools for synthesis and optimisation, usually requiring specific languages and models for design capture. Pulsar [14] departs from this paradigm. It supports, for example, an asynchronous ST template called Spatially-Distributed Dual Spacer Null Convention Logic (SDDS-NCL)[2]. SDDS-NCL enables the use of commercial EDA tools from e.g. Cadence or Synopsys as basic circuit synthesis and optimisation frameworks. Pulsar can use standard synthesis, optimisation and static timing analysis (STA) tools to determine the asynchronous cycle time of the circuits it generates[3]. The current version of Pulsar supports design capture at a level similar to the register-transfer level (RTL) in the SystemVerilog hardware description language (HDL). Nonetheless, its current version can only synthesise *static circuits*, those where data always follow pre-determined paths. Behaviours where data has to be dynamically steered or selected cannot be captured. The main contribution of this article is to propose an enhancement to Pulsar, by adding six new abstract components and the support in Pulsar to realise standard cell designs with them. These components allow Pulsar to deal with generalised hardware organisations, containing choice and decision constructs expressed in RTL-like HDL descriptions. The six components, detailed in Section III, are the decision-making components `probe` and `arbiter`, and the token-steering components `discard`, `hold`, `condhi` and `condlo`.

The rest of this work comprises four Sections. Section II covers a few concepts on asynchronous circuits and gives an overview of the Pulsar flow. Next, Section III describes the reasoning for the new components and details their implementations. Follows Section IV, which summarises how to use the components, gives examples of their use in circuits and shows some demonstrative simulation results. The paper ends with Section V, which draws some conclusive statements and cites ongoing work.

II. Asynchronous Basics and the Pulsar Flow

Resuming the discussion of asynchronous circuits started in the Section I-A, to ensure correct operation, asynchronous circuit blocks communicate with each other using local *handshake channels* [20]. This approach eliminates the need for distributing a global clock, and the complexity implied in this requirement of synchronous design. It also produces circuits that operate based on the average delay of combinational circuit blocks, not on their worst-case combinational path.

A. Handshake Protocols and Data Encoding

Handshake protocols comprise two distinct steps: (i) data *request*, when an entity announces (or requests) data availability; and (ii) data *acknowledgement*, when another entity acknowledges (or grants) data, enabling subsequent communication. Implementing these steps is protocol-dependent, and such protocols can be categorised in two main classes: (i) 2-phase (2ph) and (ii) 4-phase (4ph). A 2ph protocol implements the handshake steps with a single transition in each control signal, allowing transmission of new data immediately after acknowledgement. A 4ph protocol in turn requires that request and acknowledgement signals return to a neutral state prior to the transmission of new data.

The use of dedicated request/acknowledge signals separated from data lines characterises the BD design style introduced in Section I-A. BD design allows simpler, close to synchronous, data path implementations, at the expense of more

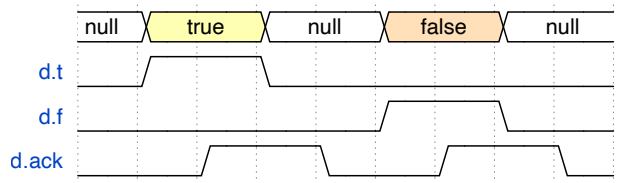
[2]Abundant details about SDDS-NCL are available e.g. in references [15]–[17].

[3]Note that Pulsar is open-source code [18], and can use the open-source, asynchronous standard cell library ASCEnD-FreePDK45 [19] and others, non-open-source ASCEnD libraries [16].

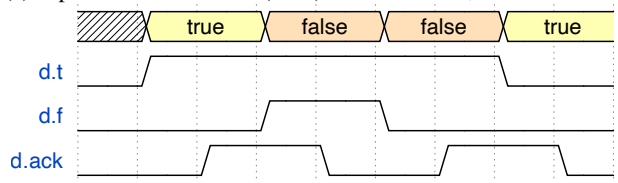
complex timing assumptions. Since combinational logic data transformations must be transparent to the local handshake protocol [20], requests must arrive at the consumer only after all computations on data channels are concluded and results are complete and ready at the consumer inputs, otherwise the latter can capture incorrect or inconsistent data. This poses a design challenge, and control signals often require delay lines to match their propagation delay to that of the data path.

As an alternative to BD design, the request information can be embedded within the data itself, by using DI codes. The most employed and the simplest DI codes use two wires to represent each bit, being these accordingly designated by the term *dual-rail* (DR) codes. DR codes and other DI codes are naturally the base for ST design styles, also introduced in Section I-A [21]. Several DI codes other than DR ones do exist, such as 1-of-4, generic m-of-n codes, and others [22], but DR codes are by far the most commonly used. This paper restricts attention to circuits using only DR codes to represent data. ST circuits require less restrictive timing assumptions than BD or synchronous circuits. This makes them less sensitive to PVT variations and ageing. ST circuits rely on local *completion detection* circuits to recognise data availability.

Figure 1 depicts two examples of ST handshake *push protocols* to transmit a single bit, several other protocols exist [23]. *Push* protocols are those where requests follow the same sense as data, by opposition to *pull* protocols, where requests follow the opposite sense of data. In ST pull protocols the single control wire is accordingly renamed *req*. Both examples in Figure 1 employ an ST DI code. It should be clear from the figure how to interpret the *true* (‘1’) and *false* (‘0’) bit representations. The *return to zero* (RTZ) protocol from Figure 1(a) classifies as 4ph, level-sensitive. It requires transmission of a spacer[4] (or null value) between every consecutive data transmission. The *no return* (NR) protocol depicted in Figure 1(b) classifies as 2ph, transition-signalling. It can transmit twice the amount of data as RTZ with the same switching activity. However, handling NR or other 2ph protocols usually requires more complex circuitry than RTZ or other 4ph protocols.



(a) 4-phase return-to-zero (RTZ) ST handshake, level-encoded.



(b) 2-phase no-return (NR) ST handshake, transition-encoded.

Fig. 1: Two ST handshake communication protocols. The drawings depicting values (true, false, null) in the top of waveforms do not physically exist, they are an interpretation of the information in the real signals *d.t* and *d.f*. Suffixes $.f$ and $.t$ identify the parts of the d DR data, false and true rails. In (a), $d.f$ =‘0’ and $d.t$ =‘0’ encode the spacer or null, which denotes absence of data in d. Accordingly, $d.f$ =‘1’ and $d.t$ =‘0’ stand for the bit ‘0’ data value and $d.f$ =‘0’ and $d.t$ =‘1’ stand for the bit ‘1’ data. In (b), transitions on $d.f =$ represent a ‘0’ value and transitions on $d.t =$ represent a ‘1’ value.

B. Channels, Token Flow, Forks and Special Gates

The data propagation behaviour in asynchronous circuits is often captured by *token flow* diagrams [20]. In such diagrams, tokens carry and abstract data. Tokens and spacers propagate between communicating entities such as latches[5] through handshake channels. Tokens and spacers wavefronts are only allowed to propagate over bubbles[6], otherwise information is potentially lost, leading to circuit failure. Models and internal forms of tools that manipulate asynchronous circuits employ the so-called *handshake components*, an example of which appear in Figure 2 [20]. The example is a *fork* component in BD and ST versions. In synchronous designs, a fork can be simply a wire that connects a place to two other places. In asynchronous hardware it consists in one wire (or multiple wires in ST circuits) going to two or more places forward and a C-element[7] merging the two or more *ack* signals backward. From this simple example it is easy to infer the multiplicity of implementations for this and other components, by varying the chosen handshake protocol and data encoding combination. The interested reader can refer to Chapters 3, 5 and 9 of the open access Sparsø’s book [20] or to the tutorial in [21] for a more thorough approach to asynchronous concepts.

Channels can be merged or split. They are merged by combinational logic blocks; these combine tokens or spacers at their inputs. However, a token cannot combine with a spacer, and a spacer cannot combine with a token. Thus, a token (or a spacer) can only propagate through a combinational logic block when all inputs of the logic block hold tokens (resp. spacers). Channels split when they feed multiple latches; tokens (or spacers) are duplicated on channel splits.

To operate correctly, ST circuits must often respect the so-called *indication principle*. This principle implies that the state of the output of a circuit must indicate the state of its inputs. This means that a circuit can only produce a token at its output once all of its inputs are complete. However, sometimes it is useful to take action on the absence of tokens in a channel, e.g. to select a different source or producing a default value. To overcome this limitation, it is possible to use *uncoupled*

[4]Binary DI codes do not use all bit configurations available, and this is behind their delay insensitivity characteristic [24]. A *spacer* is in fact associated to one of the unused or invalid bit configurations (those not used to represent data) and is meant to explicitly indicate the absence of data in a data channel. In the example here, the single bit DI code states that 01 means a ‘0’ bit value, 10 means a ‘1’ bit value, 00 is the spacer and 11 has no meaning and should never occur as a stable value.

[5]The use of the *latch* term in asynchronous circuits and their token flow model representation is associated to the general notion of a temporal barrier, and it does not necessarily imply a specific physical implementation.

[6]A *bubble* corresponds to either a token or a spacer which has already been forwarded to a next stage in the circuit, not being needed in the present stage anymore.

[7]Consider the simplest, 2-input, 1-output, symmetric C-element. This component has ‘1’ in its output if the two inputs are ‘1’, has ‘0’ in its output if the two inputs are ‘0’, and keeps the previous output value otherwise. Note this clearly implies a sequential behaviour.

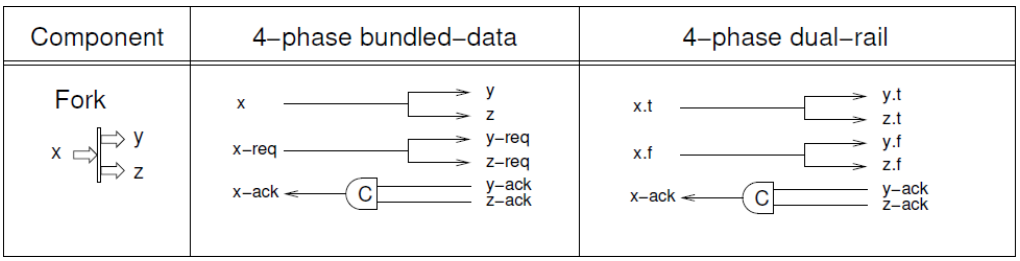


Fig. 2: Asynchronous fork component in BD and in ST dual-rail form for 4ph protocols, as Sparsø describes in [20].

channels. These are channels that are not required to be complete to ensure proper circuit functionality. If a circuit can peek the presence of a token in an uncoupled channel it is able to decide a course of action. Regarding the indication principle, asynchronous function blocks can fall in one of two classes [20]: (i) *strongly indicating* blocks wait for all of its inputs to become valid before it starts to compute and produce valid outputs, and it waits for all of its inputs to become spacers before it starts to produce spacer outputs; *weakly indicating* blocks start to compute and produce valid outputs as soon as possible, when some input signal(s) has(have) become valid, and it starts to produce null outputs as soon as possible, after some input signal(s) has(have) become spacer(s). The use of strongly or weakly indicating logic imposes different design compromises, choosing between them is a relevant issue.

The design of ST circuits often relies on the availability of a set of specific logic gates, distinct from ordinary gates like ANDs, ORs and inverters. ST design benefits from gates with hysteretic functions such as C-elements. Many of these gates are constructed in CMOS as transistor network with feedback(s). Although such gates can be composed with ordinary gates, this is sub-optimal in terms of area and performance. More importantly, feedback lines are usually part of *isochronic forks*[8], and are thus better left inside a cell. It is better to avoid their generation between design cells by automated routing tools, which are often unaware of the isochronic fork constraint. The reason why they are unaware is because such tools are constructed with only synchronous designs in mind. This work employs the ASCEnD-FreePDK45 library of asynchronous standard cells [19], which targets the FreePDK $45\,\mathrm{nm}$ predictive bulk CMOS technology.

C. Rudiments on the Pulsar Flow

Sartori et al. proposed Pulsar [14], [25], a synthesis flow pledging:

- The use of the SDDS-NCL asynchronous ST template [17], [26][9];
- An enhancement of the original pseudo-synchronous modelling strategy proposed by Thonnart et al. [27] to enable timing analysis of asynchronous ST circuits with conventional static timing (STA) tools;
- The half-buffer channel network (HBCN) timing constraint representation model;
- A linear programming technique to constrain ST circuits' cycle time.

Later, Pulsar was extended to incorporate a front-end with pre-synthesis and dual-rail expansion steps, as described by Sartori et al. in [28].

The SDDS-NCL template [15] [17] allows the synthesis and optimisation of ST combinational logic using standard EDA tools. SDDS-NCL achieves this by interleaving positive (NCL) and negative (NCLP) unate gates in the logic, as available e.g. in the ASCEnD-FreePDK45 library of asynchronous standard cells [19] and similar libraries. The cell interleaving is what enables the use of standard EDA tools, e.g. Cadence Genus to perform technology mapping and optimisation of DR circuits.

Pulsar uses conventional EDA tools to perform timing-driven synthesis of the virtual netlist to a pseudo-synchronous SDDS-NCL circuit. However, the relation between the propagation time of paths and performance is dependent on the circuit structure. To precisely compute such relation and design circuits taking advantage of computations, Sartori et al. [14] proposed the half-buffer channel network (HBCN) timing model. HBCN is a structure comprising a timed marked graph that models the propagation of tokens and spacers in the circuit; this enables setting a *cycle-time constraint*[10] as performance target during synthesis.

The Pulsar execution process starts with a user-generated RTL-like SystemVerilog input containing a clock signal. The RTL-like semantics deviates from standard RTL semantics with respect to the clock behaviour treatment. On standard RTL, variable assignments in clocked blocks capture data at each clock tick, regardless of their validity. On asynchronous ST circuits, the presence of valid data is self-evident. Thus, variable assignments inside clocked blocks in RTL-like code demand that data is only captured when available, which removes the burden of flow control from the designer and enables fine-grain pipelining during synthesis. A Pulsar RTL-like description uses a clock signal (`clk`) to guide the synthesis process only; it is used in the pre-synthesis step to

[8] *Isochronic forks* are selected wire forks within a design. Assume a wire has three or more ends, one being the fork input and the other being fork outputs. The delay from the input end to the output ends of the fork are all identical, or differ by a "negligible" amount if the fork is isochronic. When using DR logic with feedback, and given the absence of clocks for controlling the information flow, isochronic forks are a critical assumption and their definition, computation and control commands the functionality and performance of any asynchronous design.

[9] Remembering, SDDS-NCL is an acronym for Spatially-Distributed Dual Spacer Null Convention Logic. Details about this template are available in references [15]–[17].

[10] *cycle time* is the time a single-stage asynchronous circuit takes to produce an output value when presented with an input. It is analogous to the clock period from synchronous circuits.

select handshaking components and later to guide the *pseudo-synchronous* synthesis [27]. However, this clock signal is not synthesised in the final implementation, nor it produces a clock distribution tree, as expected for an asynchronous circuit. Listing 1 presents an RTL-like description of a 4-bit, 4-stage asynchronous pipeline accumulator, to demonstrate some of the differences between standard RTL and RTL-like descriptions. From Line 7 to 11, there is a sequence of assignments on a clocked block with no reset. The construction enables retiming the adder in Line 9 into a pipeline to increase performance.

Listing 1: SystemVerilog RTL-like input description for generating a 4-stage pipeline accumulator with Pulsar.

```
1  module acc #(WIDTH=4)
2     (input  logic             clk, reset,
3      input  logic [WIDTH-1:0] in,
4      output logic [WIDTH-1:0] out);
5     logic [WIDTH-1:0]         in_reg, sum, acc;
6
7     always @(posedge clk) begin
8        in_reg <= in;
9        sum <= acc + in_reg;
10       out <= sum;
11    end
12
13    always @(posedge clk or negedge reset)
14      if (!reset) acc <= '0;
15      else acc <= sum;
16 endmodule
```

On the synchronous RTL interpretation of this description, the values assigned to these variables are unknown at initialisation. Since this is a loop arrangement and registers capture data on each clock tick, these unknown values are captured and propagated; the result is the faulty behaviour depicted in Figure 3. However, when considering the Pulsar RTL-like interpretation of this description, these variables are initialised to spacers and the registers only capture data when valid. The correct behaviour resulting from the synthesis of this circuit by Pulsar is depicted in Figure 4. Notice how data progresses in waves between registers only after it becomes available.

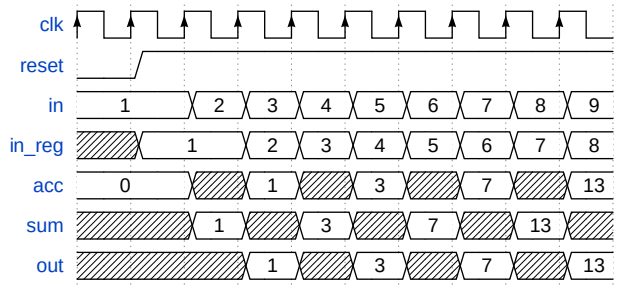


Fig. 3: Faulty behaviour caused by an ordinary RTL synthesis of the circuit in Listing 1. Note that reset is activated (='0') for a single clock cycle, and that it is not affecting the first (`always`) block. Greyed out areas here are unknown values.

III. A New Set of Components for Pulsar

The version of Pulsar previously described in [18] allows the design capture of linear and non-linear pipelines, but all circuits must be *static*, in the sense that the path data tokens follow is always the same. This paper proposes an enhancement to Pulsar to allow the dynamic choice of circuit paths tokens take during computations. The enhancement takes the form of a set of six new components which can be explicitly instantiated in RTL-like design descriptions. *Choice* is the ability to dynamically change the flow of data or control tokens in an asynchronous pipeline. To enable choice it is necessary to provide the capacity to steer tokens and make decisions. The proposition here is to provide two components, `probe` and `arbiter`, for decision making, and four components, `hold`, `discard`, `condhi` and `condlo`, for token steering.

Decision making components do share several similarities, but have significantly distinct behaviours. This prompts to a process of factoring out the similarities of the `probe` and the `arbiter` into a basic sub-component to support the construction of both. The sub-component is the *handshaking mutex*, detailed in Section III-A before exploring the `probe` and `arbiter` component architectures themselves.

A. The Decision Making Components

Decision components produce tokens with different values, depending on the presence of tokens on their inputs. `Probe` is the simplest decision making component; it detects the presence of a token in a channel, being useful when a circuit is expected to perform a default action whilst no new data is provided. When a token is present at its input channel, a `probe` consumes the token and produces a true-valued token at its output. Conversely, when no token is present at the input channel, `probe` produces a false-valued token at its output channel. An `arbiter`, in its turn, only produces a token if at least one of its two input channels has a token; this is useful when a circuit must wait for data coming from multiple sources, e.g. in a bus arbiter circuit. If a token is present on input channel *a* of an `arbiter`, it produces a true-valued token; whereas if a token is present on input channel *b*, the `arbiter` produces a false-valued token. The token arriving first is consumed by the `arbiter`. If two tokens arrive simultaneously, or close enough such that it is indistinguishable which arrived first, the `arbiter` selects one of them at random. The implementation of both `probe` and `arbiter` components is logic template-dependent. These components display a behaviour sensitive to race conditions, usually solved by mutual exclusion components, also called mutexes.

The SDDS-NCL version of the two decision components uses a complex gate, the *handshaking mutex* (HM) sub-component detailed by the CMOS transistor network in Figure 5. The HM design proposition is one of the main original contributions of this work. It combines the functionality of a traditional mutex and resetable C-elements used in asynchronous ST circuits temporal barriers. At reset, outputs (QA and QB) are both set low, placing the HM in a known state. Each output (QA and QB) of the HM is controlled by a corresponding asymmetric input pair (A+/A- and B+/B-), and by a common input (NACK). QA only rises when A+ and NACK are high, and QB is low. Similarly QB only rises when B+ and NACK are high, and QA is low. However, if both QA and QB rise simultaneously, only one of the outputs will rise after some (unbounded) arbitration time; this deliberation is ideally random. Furthermore, the condition for QA (QB) to fall is that both NACK and A- (B-) are low, regardless of the status of the other inputs and outputs.

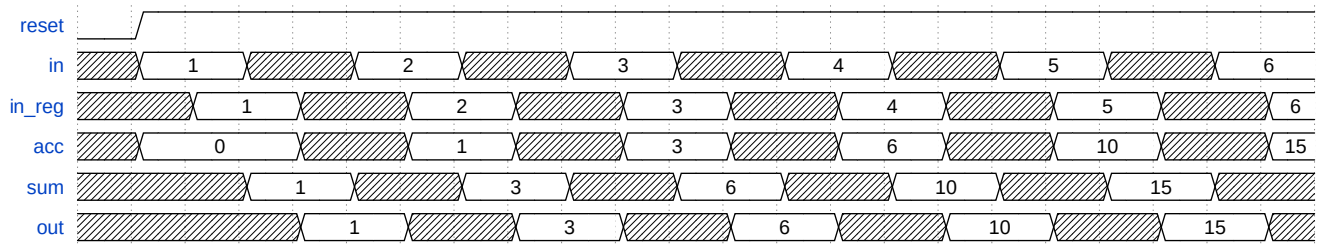


Fig. 4: Waveform depicting the (correct) asynchronous behaviour of the RTL-like description in Listing 1 after synthesis with Pulsar. Greyed-out areas are spacers.

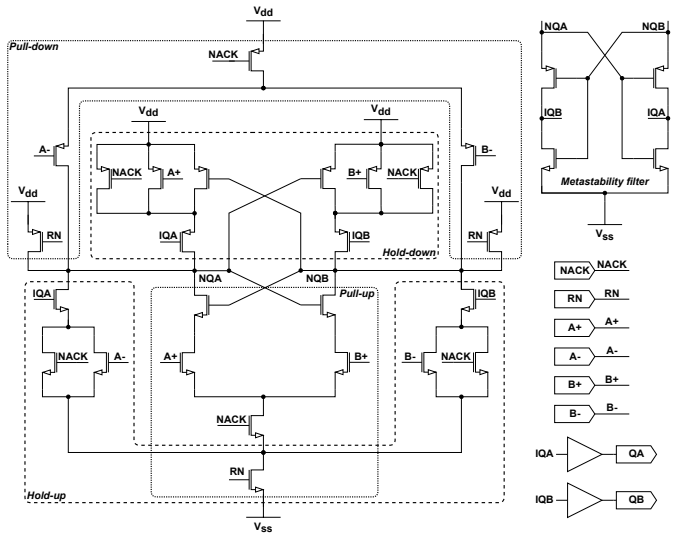
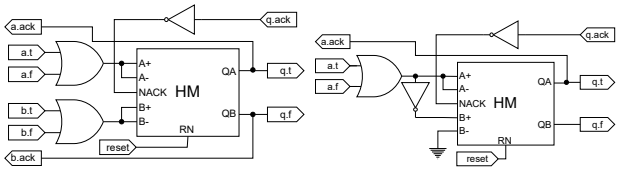


Fig. 5: The handshaking mutex (HM) - a resetable state-holding mutual exclusion gate. It comprises two cross-coupled resetable asymmetric C-elements, a metastability filter and two output buffers. Output QA (QB) rises when inputs NACK and A+ (B+) are high; conversely, QA (QB) falls when both NACK and A- (B-) are low. Just one output can be high at anytime, but both can be simultaneously low. If NACK rises when both outputs are low, and inputs A+ and B+ are both high, the internal nodes NQA and NQB can enter in a metastable state. The metastability filter ensures outputs QA and QB are both low until the metastability resolves.



(a) Arbiter (b) Probe

Fig. 6: SDDS-NCL template decision making components implementation, Asymmetric inputs on the HM allow it to implement both decision components. The NACK signal indicates whether the consumer connected to the output waits for a token or for a spacer. OR-gates act as CDs to recognise the presence of tokens in the dual-rail input channel. HM guarantees that only a single rail of the output raises on the evaluation step of the arbitration cycle. It is worth noting that `probe` relies on HM to guarantee its correct behaviour in the rare occasion where both NACK and the output of the input channel CD change simultaneously.

Figure 6 depicts the complete implementation of the `probe` and `arbiter` decision components; they employ OR-gates as completion detectors (CDs), and an HM as arbitration and latching logic. Both `arbiter` and `probe` operate in arbitration cycles comprising an evaluation step and a reset step. The evaluation step occurs when a consumer element connected to the output channel signals its availability to receive a token. At this point, the decision component can generate a true or a false token to indicate the state of its input channels. The HM raises one of its outputs based on the state of the CD connected at its input. When NACK falls, this indicates that the consumer has absorbed the token and is ready to accept a spacer, which puts the decision component in the reset step. The precise behaviour of the evaluation and reset steps are different in `arbiter` and `probe` components.

The `arbiter` has two (dual-rail) input channels, each with its own CD. Their outputs feed each a pair of asymmetric inputs on the HM. This arrangement guarantees that the `arbiter` is strongly indicating; tokens and spacers produced at the output always match tokens and spacers consumed at the selected input. At the evaluation phase, the `arbiter` only outputs a token when a token is present in at least one of its inputs. During the reset phase, the `arbiter` waits for a spacer in the selected input, after which it outputs a spacer. Notice that this behaviour affects only the selected input being acknowledged. A token (if any) at the other input must wait until its input is selected; this ensures a consistent behaviour, as no token can be lost, nor can the same token compete for arbitration twice.

In contrast to `arbiters`, `probes` have a single (dual-rail) input channel and their behaviour on the computation and reset step changes on the presence of tokens and spacers in its input channel. The CD output indicates the presence/absence of a token; it connects to the A+/A- input pair of the HM. The inversion of the CD output indicates the presence of a spacer; the negated CD output is connected only to the B+ input of the HM, and the B- input is tied to ground. When a token is detected at the input during the computation step, `probe` behaves similarly to an `arbiter`; the token is acknowledged and the output only returns to a spacer after the input returns to a spacer. When a spacer is detected at the input during the computation step, `probe` completes the arbitration cycle, regardless to changes in the input; it produces a false-value token at the output, and resets back to a spacer as soon as NACK falls, indicating the consumer stage absorbed the token. This allows a token to arrive at any time during the arbitration cycle, but its presence is only recognised when the `probe` begins its computation step. If a token arrives too close to the beginning of a computation, the `probe` may not detect the token at that arbitration cycle; but it is guaranteed to produce

a valid response and to recognise the token at its input.

B. The Token Steering Components

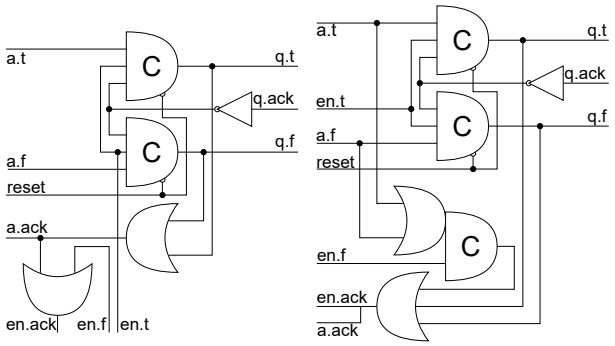
Both `discard` and `hold` act like temporal barriers that selectively perform handshakes on their data channels; this behaviour is governed by a control channel. When they receive true-valued tokens on their control channels, both components act as a conventional temporal barrier, latching and propagating data. Conversely, when the control channel receives a false-valued token, the token received in the data channel is inhibited from propagating. However, the inhibition behaviour of these control flow components is not the same. A `hold` component inhibits token propagation by withholding handshake with the input data channel; this effectively blocks the token at the input data channel. The `discard` component, on the other hand, inhibits token propagation by acting as a token sink; it performs handshake with its input data channel but do not propagate the token to its output channel. It is worth mentioning that the `discard` component only performs handshake on its control channel synchronised with its data channel, while the `hold` component can perform handshake with the control channel only. Likewise, the `condhi` and `condlo` components are conditional token sources. They consume a token from their control channel; if a false valued token is received, the token production at the output channel is inhibited. When a true-valued token is received on the control channel, a `condhi` component produces a true-valued token and a `condlo` component produces a false-valued token.

The SDDS-NCL implementations of `hold` and `discard` are depicted in Figure 7; they are similar to half-buffer components[11], but employ 3-input resetable C-elements to gate the propagation of tokens. They only propagate data arriving from the input channel if the true-rail of the control channel (*en*) is activated. The difference between `hold` and `discard` lays on the generation of the *ack* signal for the input and control channels. On both components, the control channel is always acknowledged, regardless of which rail of the control channel is activated. However, `hold` only acknowledges the data channel (*a*) when the token propagates; whereas `discard` acknowledges the data channel when either the token propagates or when the false-rail of the control channel is activated.

The SDDS-NCL implementations of `condhi` and `condlo` appear in Figure 8. Each component comprises a C-element, an OR-gate, and a constant assignment to nil. The C-element is connected to the true-rail of the control channel and the negation of acknowledgement from the output channel. It generates the true or false output rail regarding the respective component, whilst the other output rail is tied to logic nil. An OR-gate generates the acknowledgement signal for the control channel; it is connected to the output of the aforementioned C-element and to the false rail of the control channel.

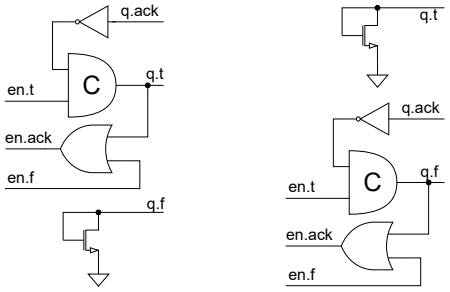
Token steering enables dynamic operation of asynchronous pipelines, in the sense that the path followed by tokens can vary. This is possible through either *fan-out* or *fan-in* steering. Fan-out steering is used to select the destination of a token,

[11] A half-buffer component is the simplest temporal barrier implemented in Pulsar; on the SDDS-NCL asynchronous template it is implemented as a pair of 2-input resetable C-elements. On initialisation, the half-buffer component is started with a spacer token.



(a) `Hold`. (b) `Discard`.

Fig. 7: Token steering components - Part 1.



(a) `Condhi`. (b) `Condlo`.

Fig. 8: Token steering components - Part 2.

i.e. demultiplexing channels. Fan-in steering is used to select the source of tokens for a channel, i.e. multiplexing channels. Using the token steering components for fan-in steering is accomplished by merging the output of multiple token steering components into a single channel and enabling only one of them. In Pulsar RTL-like input descriptions, channel merging is accomplished by placing multiple drivers in a wire; this is similar to using tristate buffers on synchronous circuits. During dual-rail expansion this merging is implemented by ORing the data rails of the merged channels.

IV. Component Utilisation and Validation

To demonstrate the use of components described in Section III in Pulsar RTL-like descriptions, this Section discusses two example circuits. The first is a loadable up-counter, which employs two token steering components and one `probe`. The second circuit is a 2-to-1 data-channel multiplexer employing two `hold` and one `arbiter` components.

The two example circuits explored here are simplified versions of circuits employed in large, complex designs. The loadable up-counter is a 16-bit simplification of a 32-bit program counter (PC) used in an asynchronous RISC-V processor implementation [29]; whilst the 2-to-1 data-channel multiplexer is part of a router used in an asynchronous ring topology network-on-chip (NoC).

A. The Loadable Up-counter

Listing 2 contains the Pulsar RTL-like description of the 16-bit loadable up-counter.

A processor PC is usually incremented, but control flow instructions such as jumps and branches may load new values

Listing 2: Loadable 16-bit up-counter Pulsar RTL-like design.

```
1  module counter #(WIDTH=16)
2     (input  wire                  clk, reset,
3      input  logic [WIDTH-1:0] load,
4      output logic [WIDTH-1:0] out);
5     logic [WIDTH-1:0]          acc, sum, load_reg;
6     wire  [WIDTH-1:0]          new_val;
7     wire                       selector;
8
9     always @(posedge clk or negedge reset)
10      if (!reset) acc <= '0;
11      else acc <= new_val;
12
13    probe s (.a(|load), .q(selector), .*);
14
15    for (genvar i = 0; i < WIDTH ; i++) begin
16       hold load_h (.a(load_reg[i]), .en(selector),
             .q(new_val[i]), .*);
17       discard sum_d (.a(sum[i]), .en(!selector),
             .q(new_val[i]), .*);
18    end
19
20    always @(posedge clk) begin
21       load_reg <= load;
22       sum <= acc + 1;
23       out <= acc;
24    end
25 endmodule
```

in it at any moment. This circuit uses a `probe` component to detect when a value is present at its load input channel. If no load value is present, the circuit proceeds with incrementing the current counter value; otherwise it discards the counter value and loads the received token. The token steering components, instantiated between lines 15-18, are in a fan-in steering arrangement; they multiplex the *new_val* channel from either *load_reg* or *sum*.

Figure 9 depicts the token flow diagram for this circuit. It is possible to observe that the `discard` component takes part in the accumulator loop. Before the arrival of a token at the load channel, `probe` produces false-valued tokens interposed with spacers. False-valued tokens prevent the `hold` component from executing a handshake on its input and output data channels. If a token arrives at the load channel after the `probe` has issued a false-valued token, it is held there until the `probe` can acknowledge its presence.

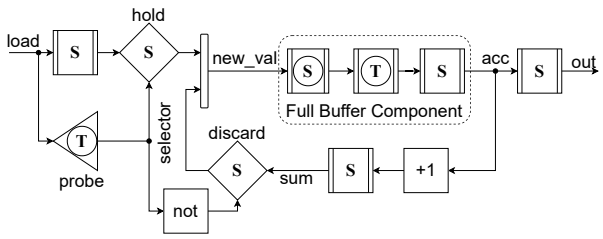


Fig. 9: Simplified token flow diagram for the loadable 16-bit up-counter. Note that except for the *+1*, *not* and the merger that produces the *new_val* channel, all elements in this diagram are handshake entities.

Analogue-mixed signal (AMS) simulation was employed to validate the circuit. The counter is synthesised using Pulsar to the worst corner of the ASCEnD-FreePDK45 library (125 °C, 0.95 V and slow transistors); the resulting netlist is placed and routed using Cadence Innovus. Mentor Calibre is used to extract the analogue netlist from the resulting GDS. A digital testbench in SystemVerilog simulating an ideal environment performs handshake and collects output values. The digital testbench also feeds random values in the 0-30 interval into the load input channel at random moments. All events in the testbench are logged to a file with the simulation time where they occurred. The AMS simulation ran for $1\mu s$. The collected output and load values w.r.t. simulation time appear in Figure 10.

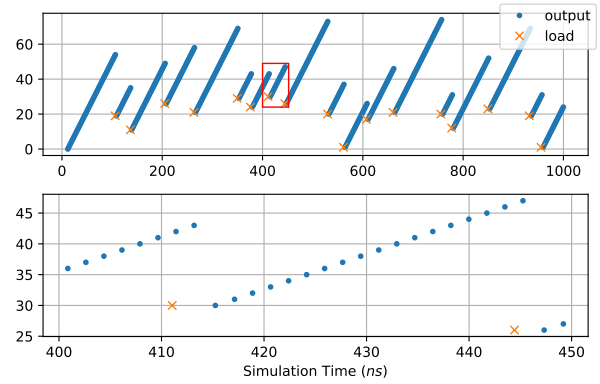


Fig. 10: AMS loadable counter simulation results. The highlighted red rectangle in the upper plot is detailed by the bottom plot; it shows two load procedures and their subsequent increment cycles.

The top plot depicts the entire simulation time; the bottom plot zooms in the area marked by the red rectangle on the top one. Albeit asynchronous, the circuit yields a new output value in a fairly consistent rate. On the zoomed portion, there are two load procedures: on the first, the circuit yields two output values before accepting the load value; on the second, the circuit yields a single output value before loading the new value. This derives from the circuit asynchronous nature, its structure and the random moments the load input is fed; although delayed, the load procedure is guaranteed to occur.

B. The Channel Multiplexer

Listing 3 depicts the RTL-like description for the 16-bit channel multiplexer. This multiplexer has two input channels (a, b) and one output (o) channel. Tokens can arrive at either of the inputs anytime; the `arbiter` detects the presence of the token and yields a decision in favour of the associated `hold` component. If a token arrives at one input whilst another token is propagating through the other input, the corresponding `hold` component blocks the late token until the handshake at the `arbiter` deliberates in favour if its propagation. In the situation where tokens arrive at the two input channels at sufficiently near times, the `arbiter` must deliberate between either of the input channels; ideally this is a random selection.

Lines 12-26 implement the input and output buffers; Lines 28-33 instantiate the `arbiter` component; Lines 35-48 instantiate the `hold` components for both channels. A simplified token flow diagram of this circuit is depicted in Figure 11. Notice that on the RTL-like description inputs a and b of the `arbiter` are fed by the least significant bit of the input channels. The `arbiter` is able to detect the presence of a token in a single-bit channel; sampling the least significant bit assumes that all bits are inserted in tandem. The token flow diagram simplifies this by representing the multiple parallel channels required to carry multiple bits as a single

Listing 3: Pulsar RTL-like 16-bit channel multiplexer.

```
module merge
#(WIDTH=16, INPUT_DEPTH=2, OUTPUT_DEPTH=2)
( input logic clk, reset,
  input logic [WIDTH-1:0] a, b,
  output logic [WIDTH-1:0] o );

  logic [WIDTH-1:0] a_pipe [INPUT_DEPTH];
  logic [WIDTH-1:0] b_pipe [INPUT_DEPTH];
  logic [WIDTH-1:0] o_pipe [OUTPUT_DEPTH];
  logic decision;

  always_ff @(posedge clk) begin
    a_pipe[0] <= a;
    b_pipe[0] <= b;
  end

  for (genvar i = 1; i < INPUT_DEPTH ; i++)
    always_ff @(posedge clk) begin
      a_pipe[i] <= a_pipe[i-1];
      b_pipe[i] <= b_pipe[i-1];
    end

  for (genvar i = 1; i < OUTPUT_DEPTH ; i++)
    always_ff @(posedge clk) begin
      o_pipe[i] <= o_pipe[i-1];
    end

  arbiter arb(
    .clk(clk), .reset(reset),
    .a(a_pipe[INPUT_DEPTH-2][0]),
    .b(b_pipe[INPUT_DEPTH-2][0]),
    .q(decision)
  );

  for (genvar i = 0; i < WIDTH; i++) begin
    hold select_a (
      .clk(clk), .reset(reset),
      .a(a_pipe[INPUT_DEPTH-1][i]),
      .en(decision),
      .q(o_pipe[0][i])
    );
    hold select_b (
      .clk(clk), .reset(reset),
      .a(b_pipe[INPUT_DEPTH-1][i]),
      .en(~decision),
      .q(o_pipe[0][i])
    );
  end

  assign o = o_pipe[OUTPUT_DEPTH-1];
endmodule
```

channel. Also, notice that the `arbiter` samples the input channel pipeline buffer one stage before the last; this is made to delay-match the arrival of tokens at the `hold` component and increase performance.

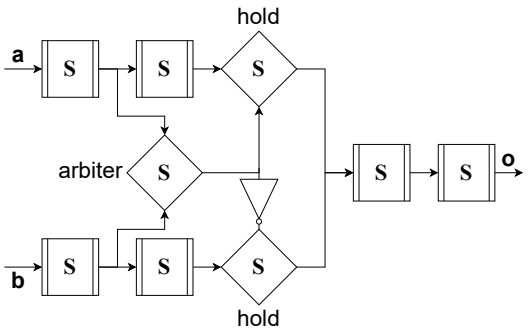


Fig. 11: Simple token flow diagram for the channel multiplexer.

The synthesis and simulation flow for the channel multiplexer is the same as for the counter: the circuit is first synthesised using Pulsar to a netlist of ASCEnD-FreePDK45 gates; this implementation is then placed and routed using Innovus; Calibre parasitic-extraction (PEX) is used to obtain an analogue Spice netlist from the resulting GDS file. XCelium simulates the resulting Spice netlist in conjunction with a digital testbench for the duration of 1 µs. The digital testbench performs handshake with the circuit inputs and outputs. It inserts positive numbers on input *a* and negative numbers on input *b*; the insertion of tokens and spacers on each input is delayed by a random time value between 1-2 ns after acknowledgement, and the output channel is acknowledged with nil delay. This arrangement simulates two computationally limited producers competing for a shared fast consumer.

Figure 12 depicts the cumulative distribution of arrival times at the input and output channels between 0 and 100 ns. In this plot, the count value for each channel is incremented by one every time a token is sent or received. Consequently, the throughput is captured by the average slope of the curve. Here, it is possible to observe that at any time the token count at the output is less than or equal to the sum of the token counts at the inputs; this implies that the throughput at the output is the sum of the throughput of each input.

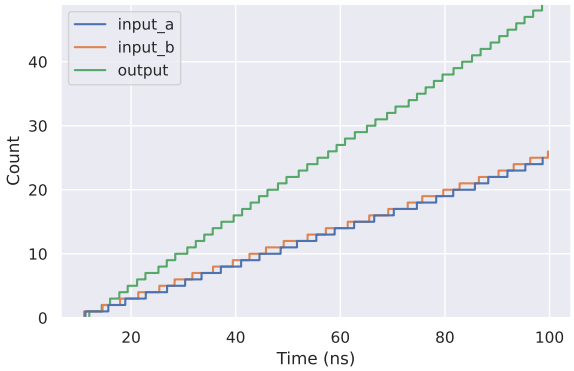


Fig. 12: Empirical cumulative distribution function of token arrival times at both inputs and at the output of the channel multiplexer between 0 and 100 ns. The average slope of the curves captures the throughput at input/output channels.

Gauging the multiplexer fairness is achievable by observing the data latency for traversing the channel multiplexer from either input. The pipeline depth, the competition for the `arbiter`, and the producer behaviour affect latency. Given enough samples and equally behaving producers, a fair `arbiter` ideally shows identical latency distributions. Figure 13 shows the latency distributions for the multiplexer from either input in the simulation interval from 0 to 1 µs.

It is possible to observe that albeit fairly similar, latency distributions from inputs *a* and *b* are slightly skewed towards the right-hand side of the histogram. This behaviour can be attributed to the deterministic nature of analogue transient Spice simulation. When two tokens arrive at similar times the HM sub-component can enter a metastable state. Without parasitic leakage, this metastability is numerically stable in Spice simulation, it does not resolve randomly to either side, as it would on real silicon subject to quantum effects. However, since the analogue Spice netlist employed here results from layout extraction, it comprises parasitic circuits that induce this metastable state to numerically solve more often to a same outcome. Such parasitic circuit behaviour is sensitive

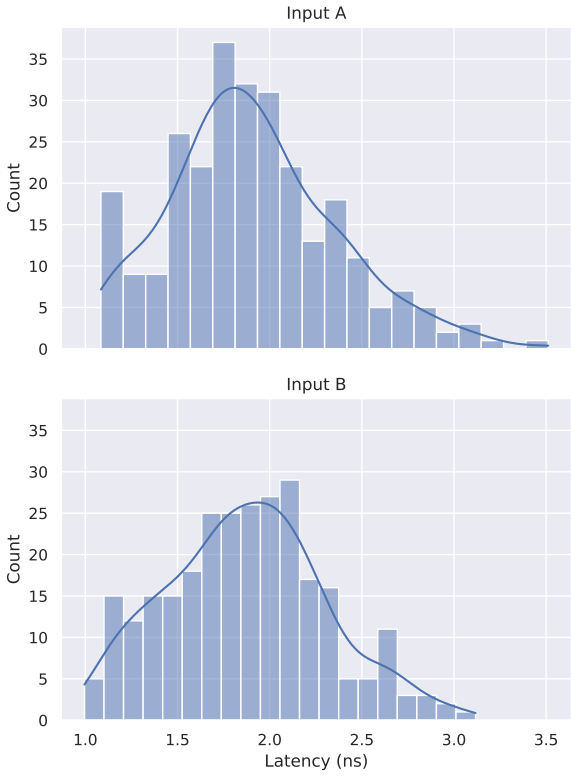


Fig. 13: Channel multiplexer latency distributions for each of the `arbiter` inputs.

to layout place and route choices. Consequently, different placements for the same circuit yield different unbalanced outcomes. Regardless the slight bias in simulation, the multiplexer never starves any input channel and guarantees data delivery integrity.

V. Conclusions and Ongoing Work

This Section highlights a few points overlooked or only lightly touched upon along the main body of this article. Also, it presents links to a few related ongoing works.

First, the paper relies on the assumption that asynchronous design, and particularly ST design is a design style more robust to variations (PVT, ageing etc.). The assumption is not demonstrated, it is associated intuitively to two structural characteristics of ST design, using local handshakes and employing DI codes. The former of these avoids most long wires and the associated timing problems they bring. The latter reduces the crossed timing dependence between data and control lines, and the number of implied relative timing constraints to solve. Demonstrations, even if partial, that the advantage takes place in practical circuits designed with Pulsar is available in previously published work, including [5], [6], [30], [31].

Second, the six proposed components are necessary and sufficient to design complex behaviours with Pulsar using RTL-like constructions. The components are abstract constructions, pre-processed by Pulsar before the synthesis with commercial tools takes place. Several processing and modelling steps not covered here allow reaching an operational IC module layout from the RTL-like input. For example a simple component, such as `condhi` (refer to Figure 8) is transformed into a set of four standard cells, an inverter, a 2-input C-element, a 2-input OR gate and a *tielo*. Library cells for complex behaviours are nonetheless advantageous. Figure 14 illustrates this for the HM cell, a specially designed layout.

Third, Pulsar processing is dependent on the employed ST template, on the underlying technology node and on the chosen standard cell library. This work illustrated concepts with the SDDS-NCL template, the predictive FreePDK45 nm node and the ASCEnD-FreePDK45 library. Pulsar is however independent of all three choices and currently supports multiple templates, technology nodes and libraries.

Fourth, the new components enable the use of Pulsar to design more complex behaviours than its previous version did. For example, asynchronous processors require the new components in several parts of an instruction set architecture (ISA) processor design. The new Pulsar enabled e.g. implementing a complete asynchronous RISC-V RV32I core [29]. The associated RTL-like code uses the components extensively. Also, an implementation of a ring-topology network-on-chip (NoC) router is under way. Arbitration is a fundamental action performed inside NoC routers. Asynchronous implementations of routers bring advantages to communication networks, providing a seamless way to adapt multiple circuits using distinct clocks.

Finally, the end of Section IV-B above discussed that the electrical simulation of metastable circuit behaviour is not reliable, which is a fact well-known to back-end circuit designers and researchers [32]. Limitations in the electrical simulation methodology hinder the verification of the HM gate that is expected to act correctly under conditions of metastability. Ongoing work to prototype HM and other cells from some ASCEnD library in silicon should be able to confirm precise, correct and fair operation of the components.

Acknowledgements

This research was partially funded by CAPES and CNPq under grant no. 311587/2022-4, Brazilian research funding organisations.

References

[1] M. A. Scarpato, “Digital Circuit Performance Estimation under PVT and Aging Effects,” Ph.D. dissertation, Université Grenoble Alpes, Dec. 2017.

[2] F. A. Kuentzer and M. Krstic, “Soft Error Detection and Correction Architecture for Asynchronous Bundled Data Designs,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4883–4894, Dec. 2020.

[3] A. Karlsson, O. Andersson, J. Sparsø, and J. N. Rodrigues, “IR-Drop Reduction in Sub-VT Circuits by De-synchronization,” in *IEEE Subthreshold Microelectronics Conference (SubVT)*, Oct. 2012, pp. 1–3.

[4] S. Jain, L. Lin, and M. Alioto, *Adaptive Digital Circuits for Power-Performance Range beyond Wide Voltage Scaling*. Springer, 2020, ch. Reconfigurable Clock Networks, Automated Design Flows, Run-Time Optimization, and Case Study, pp. 115–144.

[5] M. L. L. Sartori, R. N. Wuerdig, M. T. Moreira, S. Bampi, and N. L. V. Calazans, “Leveraging QDI Robustness to Simplify the Design of IoT Circuits,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[6] K.-L. Chang, J. S. Chang, B.-H. Gwee, and K.-S. Chong, “Synchronous-Logic and Asynchronous-Logic 8051 Microcontroller Cores for Realizing the Internet of Things: A Comparative Study on Dynamic Voltage Scaling and Variation Effects,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 1, pp. 23–34, Mar. 2013.

[7] M. Renaudin and A. Fonkoua, “Tiempo Asynchronous Circuits System Verilog Modeling Language,” in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2012, pp. 105–112.

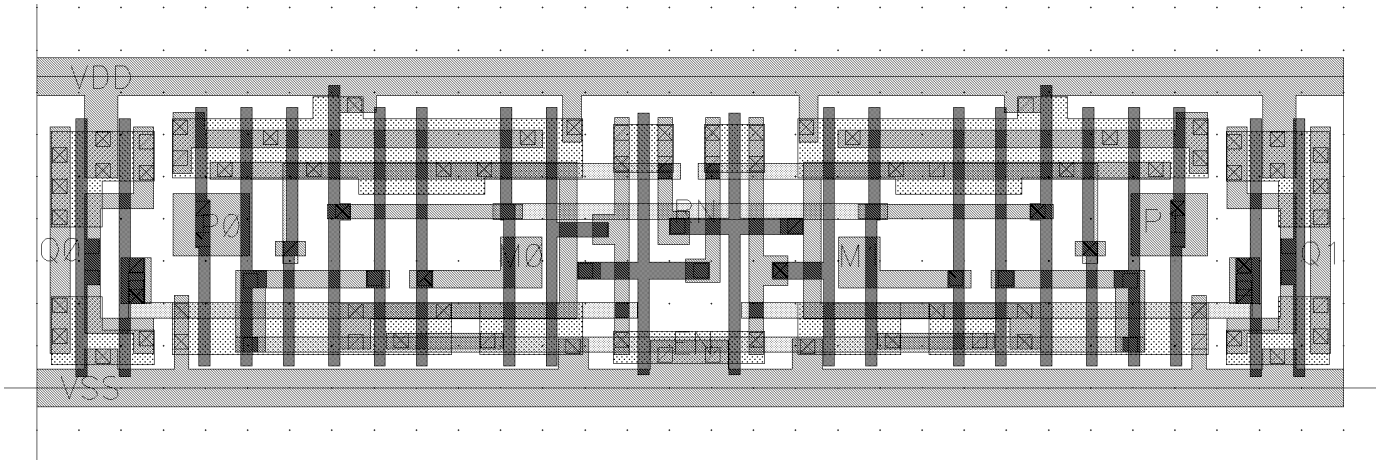


Fig. 14: This is an HM standard cell layout. It uses the predictive bulk CMOS technology FreePDK $45\,\mathrm{nm}$. The cell is part of the ASCEnD-FreePDK45 standard cell library. Comparing it to the schematic of Figure 5, it is noticeable that pin labels are not the same in general. This arises due to the naming conventions required by the layout tool. The only pin named identically in both representations is the active-low reset (RN). The other pin correspondences are (schematic to layout): NACK→EN, M0→A-, P0→A+, M1→B-, P1→B+, Q0→QA and Q1→QB. The attentive reader can also note that some transistors are duplicated w.r.t. to the schematic; the large cell size requires a few transistors doubling to reduce the employed metal layers.

[8] J. Tse and A. Lines, "NanoMesh: An Asynchronous Kilo-Core System-on-Chip," in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2013, pp. 40–49.

[9] 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.

[10] D. Edwards and A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[11] A. Bardsley, L. Tarazona, and D. Edwards, "Teak: A Token-Flow Implementation for the Balsa Language," in *2009 Ninth International Conference on Application of Concurrency to System Design*, Jul. 2009, pp. 23–31.

[12] 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.

[13] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC Flow for GHz Asynchronous Designs," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 36–50, 2011.

[14] 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.

[15] 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.

[16] M. T. Moreira, "Asynchronous Circuits: Innovations in Components, Cell Libraries and Design Templates," Ph.D. dissertation, Pontifícia Universidade Católica do Rio Grande do Sul, FACIN-PPGCC, 2016.

[17] 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.

[18] M. L. L. Sartori and N. L. V. Calazans, "Pulsar - A Flow to Support the Design of QDI Asynchronous Circuits," Jun. 2020. [Online]. Available: https://github.com/marlls1989/pulsar

[19] M. L. L. Sartori, M. T. Moreira, and N. L. V. Calazans, "ASCEnD-FreePDK45 - A Free Standard Cell Library for SDDS-NCL Circuits," Jun. 2020. [Online]. Available: https://github.com/marlls1989/ascend-freepdk45

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

[21] N. L. V. Calazans and M. L. L. Sartori, "Asynchronous Circuit Principles and a Survey of Associated Design Tools," *Journal of Integrated Circuits and Systems*, vol. 17, no. 3, pp. 1–10, Dec. 2022. [Online]. Available: https://jics.org.br/ojs/index.php/JICS/article/view/677/421

[22] M. Y. Agyekum and S. M. Nowick, "Error-Correcting Unordered Codes and Hardware Support for Robust Asynchronous Global Communication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 75–88, 2012.

[23] P. Beerel, R. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010. [Online]. Available: https://books.google.com.br/books?id=7bw6-yob7mAC

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

[25] M. L. L. Sartori, "PULSAR: Towards a Synthesis flow for QDI Circuits," Master's thesis, PPGCC - School of Technology - PUCRS, Aug. 2019, 195p.

[26] M. T. Moreira, A. Neutzling, M. Martins, A. Reis, R. Ribas, and N. L. V. Calazans, "Semi-custom NCL Design with Commercial EDA Frameworks: Is it possible?" in *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2014, pp. 53–60.

[27] 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.

[28] 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.

[29] W. A. Nunes, M. L. L. Sartori, and N. L. V. Calazans, "Pulsar ARV - A QDI Asynchronous RISC-V Implementation," Jul. 2022, Private Gitlab Repository. [Online]. Available: https://lesvos.pucrs.br/williannunes/PARV

[30] T. A. Rodolfo, M. L. L. Sartori, M. T. Moreira, and N. L. V. Calazans, "Quasi Delay Insensitive FIFOs: Design Choices Exploration and Comparison," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2021, pp. 1–5.

[31] R. N. Wuerdig, M. L. L. Sartori, B. A. Abreu, S. Bampi, and N. L. V. Calazans, "Mitigating Asynchronous QDI Drawbacks on MAC Operators with Approximate Multipliers," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2022, pp. 1–5.

[32] S. Yang and M. R. Greenstreet, "Simulating Improbable Events," in *Design Automation Conference (DAC)*, Jun. 2007, pp. 154–157.