A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow

Marcos L. L. Sartori, Matheus T. Moreira, Ney L. V. Calazans
PUCRS - School of Technology - Ipiranga Av., 6681 - Porto Alegre - Brazil, 90619-900
{marcos.sartori,matheus.moreira}@acad.pucrs.br, ney.calazans@pucrs.br

Abstract— **Asynchronous quasi-delay-insensitive circuits are known for their robustness against variations, but their widespread use has been prey to the absence of adequate design methods and lack of design and verification tools. The recently proposed Pulsar flow enables the design and optimisation of quasi-delay-insensitive circuits using conventional EDA tools, enhanced by adequate libraries, methods and models. Pulsar enables designers to naturally trade performance for power or area, whenever there is slack in timing budgets. However, Pulsar lacked an automated dual-rail expansion method to support its operation, requiring that designers manually develop a timing model as input to the computation of asynchronous cycle time constraints. This paper proposes and describes the features of a frontend for Pulsar. Pulsar-F, the new flow version can be used as a push-button design tool for asynchronous QDI circuits. Pulsar-F adds the following features to Pulsar: (i) an RTL-based design capture method; (ii) a heuristic, timing-driven single-rail pre-synthesis process using commercial EDA tools; (iii) a dual-rail expansion technique with fine-grain acknowledgement network generation; (iv) a tool that automates the computation of the Hal-Buffer Channel Network (HBCN) graph-based timing model for pre-synthesised circuits and derives a set of timing constraints for it. Experiments show that Pulsar-F improves Pulsar to further aid asynchronous designers to trade off power, area and performance.**

I. Introduction and Related Work

The evolution of integrated circuit technologies brings uncertainties concerning the results of the fabrication process, as well as the susceptibility of chips to supply voltage effects such as IR drop and to environmental effects such as temperature variations. These effects can be particularly relevant during the design and deployment of systems for eagerly expected newer technologies such as the Internet of Things (IoT). Dealing with process, supply voltage and temperature (PVT) variations, harsh environmental conditions such as radiation or strict limits on power and heat dissipation requires circuits that are robust in several aspects. In this scenario, quasi-delay-insensitive (QDI) circuits can provide an elegant solution.

QDI circuits are naturally robust against delay variations and can handle PVT effects arguably better than synchronous design or even better than other asynchronous circuit templates, such as bundled-data. Unfortunately, designing QDI circuits is often a laborious manual work that requires detailed knowledge on convoluted asynchronous techniques. Also, QDI designs are frequently handcrafted cell by cell, impairing its adoption in a larger scale. Synchronous design has coped with technology scaling to the tens of manometers and the last decades saw little interest from traditional electronic design automation (EDA) vendors in supporting QDI circuits. As a consequence, asynchronous design automation is still in its infancy, especially when compared to what is available to synchronous designers.

Despite not being in widespread use, QDI design found niches like security [1] and high speed circuits [2]. Often, given a promising application, a new QDI design (style) is devised and a specific set of tools built to support it [3]. Research groups proposed tools and flows like Balsa [4], Teak [5], Uncle [6] and Proteus [7]. What most of these QDI design tools share was the fact that they could not rely on traditional EDA for synthesis and optimisation, usually requiring specific languages and models for design capture.

An analysis of the state of the art reveals that Proteus, Uncle and Pulsar [8] are works that got the closest to leveraging traditional EDA, design capture models and methods for QDI 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 of traditional tools. Proteus counts with a sophisticated frontend flow, where asynchronous channels are modelled using SystemVerilog and design capture is done through a communicating sequential processes (CSP) model. It targets however, an even more specific set of cells that are implemented as dynamic domino logic gates, limiting its use to a larger set of QDI templates.

Pulsar, on the other hand, has its roots in an asynchronous template called SDDS-NCL, which was proposed to support the use of traditional commercial EDA frameworks, being mostly compatible with traditional commercial tools. Furthermore, Pulsar allows using standard static analysis (STA) tools for optimising the asynchronous cycle time of its target circuits. Unfortunately, in its first version Pulsar required a manually designed Verilog input file describing the behaviour of the circuit, logic gate by logic gate. This paper describes the latest innovations added to this flow: a frontend environment that enables an RTL description to be fully mapped to the SDDS-NCL QDI template, generating a Verilog compatible with its synthesis flow and an HBCN model for cycle time constraining. The new flow is accordingly called Pulsar-F, where F stands for full, in the sense that Pulsar has become a complete flow for QDI design.

The rest of this document is organised as follows. Section II gives an overview of how to use the Pulsar-F flow. Next, Section III explores some basic concepts that enable capturing QDI circuit structures from conventional RTL descriptions.

2643-1483/20/$31.00 ©2020 IEEE
DOI 10.1109/ASYNC49171.2020.00009

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.

Section IV explores the need for specific libraries to support the Pulsar-F flow and their structure. Section V approaches the construction of timing models and their processing, which produces a set of constraints to drive the Pulsar flow application. Experiments and their results are the subject of Section VI, while Section VII addresses conclusions and ongoing work.

II. Pulsar-F Synthesis Flow Overview

Pulsar-F is the main contribution of this paper. It complements the Pulsar flow described in [8], constituting a push-button tool for synthesising asynchronous QDI circuits from an RTL description. The resulting circuit follows the SDDS-NCL template. Pulsar applies a pseudo-synchronous design approach and capitalises in the existence of a previously computed HBCN timing model, leveraging the use of commercial EDA tools to perform design and optimisation of asynchronous QDI circuits. Figure 1 depicts the main features of the Pulsar-F flow, discussed in the remaining of this Section. To enable future enhancements, Pulsar-F comprises two parts, a design template-independent one and a template-dependent one. Currently, only a single asynchronous logic template is supported (SDDS-NCL), but the scripts include provisions to adopt other templates. Reference [8] details the physical synthesis support by Pulsar. This is accordingly ignored herein.

As in conventional approaches, a design starts with the user generating an RTL Verilog or VHDL synchronous description, which can be validated by simulation and/or semi-formal or formal verification. Once the user obtains a functionally valid RTL description, he or she runs `syn_rtl` a bash shell script that calls the Cadence Genus tool to execute the Tcl script `syn_rtl.tcl`[1]. The latter contains conventional synthesis commands, interspersed with the firing of specific actions to: (1) read in a special component library to make the single-rail output amenable for the ensuing dual-rail expansion; (2) generate a circuit graph of the synthesised circuit (aided by one specific Tcl script, `analysis.tcl`), which will later be used to construct the HBCN circuit timing model. The last action in `syn_rtl.tcl` is calling a Haskell program (`expander.hs`) to perform the dual-rail expansion of the single-rail netlist into a virtual gate netlist [8], one of the entries to the Pulsar flow. The input netlist is then processed and transformed into a Virtual Netlist with virtual gates and pseudo-flops [8], completing the Pulsar-F template-independent part.

After the end of the Pulsar-F template-dependent part follows the execution of the bash shell script `syn_sdds`. This script calls the Cadence Genus synthesis tool to execute the Tcl script `syn_sdds.tcl`. The latter comprises all commands of the Pulsar flow execution, but this is now preceded by calling the Haskell program (`constrainer.hs`), to perform two tasks: (1) generate a marked graph from the structural circuit graph produced in the Pulsar-F template-independent part; (2) Employ the marked graph for computing the timing constraints to apply during execution of the Pulsar flow.

The single-rail RTL netlist uses the Components Library from Figure 1. This is a Liberty format library containing a simple set of combinational and sequential components, used by the synthesis tool to produce the netlist.

[1]Frameworks like Synopsys can also be used, by editing a few Tcl scripts.

Each component has an associated SystemVerilog module, which defines its dual-rail expansion. The dual-rail expansion takes advantage of the SystemVerilog language *interface* feature to represent dual-rail, four-phase, RTZ channels. These channels interconnect modules that implement the dual-rail expansion of wires. SystemVerilog interfaces are also used for constructing channel acknowledgement networks. The Dual-Rail Expander replaces every wire in the single rail netlist with a channel to create the *Virtual Netlist*.

III. Dual-rail Channels

Handshake channels allow the communication of tokens between entities in an asynchronous circuit. Each channel is an interconnection between components. They are analogous to wires in a synchronous circuit. However, in contrast to a wire, a channel provides synchronisation between elements. The Pulsar-F dual-rail expansion exploits this analogy by replacing all wires from the single-rail netlist with channels. However, a channel requires multiple wires and some logic. To this end, this work takes advantage of SystemVerilog *interfaces*. An interface is a SystemVerilog construct that allows abstracting interconnections on a system-level description. They are instantiated and bound to module instances in place of wires. However, each interface bundles multiple wires that are part of the same interconnect, e.g. data, address and control lines of a processor bus. Besides wires, interfaces include *modports* that allow defining how modules relate to the interface, e.g. as a master or as a slave. Modports define the direction that each wire takes when the interface is bound to a module. SystemVerilog interface and modport constructs are supported by current commercial synthesis tools. Since an interface binds to module instances like a wire, it is straightforward to replace a wire with an interface instance. It only requires modifying the Verilog file of the single-rail netlist, replacing every wire declaration with an instance of the desired interface.

To exploit the concept, a `drwire` interface (see Listing 1), is defined. This interface bundles three wires that implement a dual-rail four-phase RTZ channel: wires `t` and `f` encode the forward-propagating token and the `ack` wire carries the back-propagating acknowledgement. This interface also features two modports. These are used by SystemVerilog modules to implement the dual-rail expansion of components: (i) modport `in` is a replacement for input ports, binding to the channel as a consumer; (ii) modport `out` is a replacement for output ports, binding to the channel as a producer.

Listing 1: The `drwire` interface.

```
interface drwire();
  wire t, f;
  wand ack;

  modport in (input t, input f, output ack);
  modport out (input ack, output t, output f);
endinterface // drwire
```

Since channels are replacements for wires, they must support multiple consumers. At any given time instant the producer may send tokens, starting a handshake on the

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.

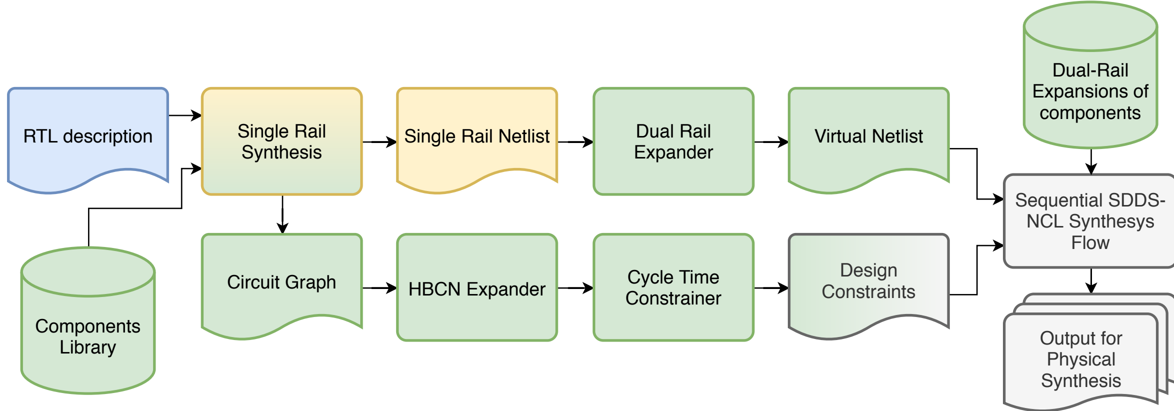


Fig. 1: The Pulsar-F synthesis flow. Blue items are user input; yellow items are either third party (commercial) tools or conventional output by these tools; green items are this work contributions; grey items are covered in [8].

channel. The token reception must be acknowledged by all consumers listening to the channel before the producer can send more tokens. This means that the producer must receive the acknowledgement only after all consumers have acknowledged reception. The situation where a channel feeds two or more consumers characterises a fork. This is implemented by merging the concurrent acknowledgements coming from different consumers with n-input C-elements, i.e. the channel acknowledgement only raises (lowers) when every consumer has raised (lowered) its acknowledgement. A multiple n-input C-element ($n > 2$) is commonly implemented using a tree of simple C-elements. This tree of C-elements is also known as the *acknowledgement network*, because it propagates the acknowledgement from consumers to producers in the channel.

To implement forks transparently in channels, this work uses the Verilog wired-and type (*wand*) to create the acknowledgement network. A wire declared as *wand* is a wire that and-reduces multiple assignments. Synthesis tools implement wand wires as n-input AND gates, where every assignment creates a new input. Reading from a wand wire yields the output of the associated n-input AND gate. During synthesis, this n-input AND gate corresponds to an and-reduction virtual function (v-function)[2]. V-functions are used to select NCL or NCLP gates during synthesis, according to the required protocol. By design, all channels in the Virtual Netlist implement the RTZ protocol. Therefore, the v-functions associated with the channel are interpreted as v-functions of NCL gates. C-elements are valid NCL gates and they can be selected by their v-function during synthesis. The v-function of an NCL gate is its activation function. The activation function of an n-input C-element is the and-reduction of its inputs, i.e. its output changes to one when all inputs are one. Therefore, the and-reduction v-function is realised by an n-input C-element on RTZ. This allows creating the acknowledgement network for the channel, by setting the `ack` wire type to wand in the `drwire` interface definition. The use of `drwire` allows the construction of fine-grain channel networks, where each channel represents a single dual-rail "bit". Combinational components bind to channels passively, manipulate tokens and propagate acknowledgements between inbound and outbound channels. Conversely, sequential components bind to channels as active producers and consumers. They consume tokens from their inbound channels and commence handshaking on their outbound channels. The construction of this fine-grain channel network allows handshakes to be performed only where data dependency exists. Parallel independent channels in a bus can handshake concurrently. This allows e.g. that individual bits in an adder complete handshaking as soon as their computation is ready, regardless of other bits computations.

[2]A virtual function is the ON-set or activation function of an NCL gate or the OFF-set or activation function of an NCLP gate, as detailed in [8].

IV. The Library of Components

A single-rail netlist instantiates sequential and combinational components from the Components Library, which have no associated physical layout. Instead, each component is expanded by a SystemVerilog module that binds to channels implemented by the `drwire` interface.

Combinational components bind passively to their input and output channels, meaning that they do not complete handshakes, and act as passive consumers and producers. Combinational components combine tokens from their input channels on the output channel. They propagate the acknowledgement between their output and input channels. Conversely, sequential components are active handshaking elements that control the propagation of tokens in the pipeline. They complete handshakes between input and output channels.

The mentioned SystemVerilog modules are instantiated in the Virtual Netlist. The contents of these modules are used during the synthesis process to implement a circuit following the SDDS-NCL template. Combinational components are expanded using virtual functions and sequential components are expanded to pseudo-synchronous WCHB registers. The next Sections covers each component type expansion process.

A. Combinational Components

A combinational component binds to `drwire` channels. Since the `drwire` interface implements dual-rail RTZ encoded channels, a component expansion needs to implement adequate DI logic to manipulate dual-rail RTZ codewords. To this end, each component implements the Delay Insensitive

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.

Minterm Synthesis (DIMS) [9], [10] expansion of its equivalent gate as a SystemVerilog module. The module expresses the activation function of each rail as a sum-of-products with all valid minterm combinations.

In DIMS the activation function is realised by directly mapping conjunctions (AND) to C-Elements and disjunctions (OR) to OR gates. Similarly, when employing the RTZ protocol an NCLnOFn gate (a C-Element) is represented by the conjunction v-function and an NCL1OFn (an OR gate) is represented by the disjunction v-function. This implies that the DIMS activation function is equivalent to a v-function on the RTZ protocol, which enables the use of the SDDS-NCL flow to realise the combinational component dual-rail expansions. However, the traditional DIMS mapping is only one of the possible realisations for this v-function. The SDDS-NCL flow enables commercial EDA tools to optimise and map v-functions resulting from the combination of elements to gates of an existing library.

To map v-functions during synthesis, each of these expressions are assigned to an output rail in the expansion module. This module, an example of which appears in Listing 2, is implemented in SystemVerilog. Ports of the module are channels created with `drwire` interfaces. Input ports use the `drwire.in` modport and outputs use the `drwire.out` modport. The output channel acknowledgement directly connects to the input channels acknowledgement. The module is instantiated by the Virtual Netlist during the SDDS-NCL flow execution.

Listing 2: The `nand2` expanded module.

```
module nand2
  (drwire.in a,
   drwire.in b,
   drwire.out y);
   assign y.t = a.f & b.f |
                a.f & b.t |
                a.t & b.f;
   assign y.f = a.t & b.t;

   assign a.ack = y.ack;
   assign b.ack = y.ack;
endmodule // nand2
```

To construct the Virtual Netlist, it is first necessary to synthesise a single-rail netlist. This instantiates standard cells by name. The Virtual Netlist is a simple textual transformation of the single-rail netlist that preserves the name of the instantiated cells. Therefore, for the Virtual Netlist to instantiate components, a cell with the same name must be selected during the single-rail synthesis. For this reason, components are modelled as cells in the Components Library. Table I presents the list of available combinational components in this library. `nand2` and `nor2` are the most fundamental components. Paired to `inv` these can generate any combinational logic. `xor2` is provided to allow efficient implementations of arithmetic logic and `buff` is provided to allow forking a channel.

The synthesis tool is driven by timing, it attempts to synthesise the circuit with the smallest area that meets the timing constraint. This work takes advantage of this fact to guide the single-rail synthesis to produce a virtual netlist that is simpler to synthesise. For that, virtual delay and area values are assigned to combinational elements of the Components Library according to the complexity of their v-function expansions. This heuristics attempts to produce a dual-rail netlist with simpler v-functions for each rail.

Table I also presents the virtual delay of combinational components in the library. The virtual delay corresponding to the true rail v-function is expressed in the rise transition. Conversely, the virtual delay corresponding to the false rail is expressed in the fall transition. Timing estimations are based on the minimal number of 2-input gates required to activate the virtual function in a disjunctive normal form, for each OR gate a 5 ps delay is attributed, and for each AND gate a 10 ps delay is attributed. For instance, consider the `nand2` gate. Activating its true rail v-function requires activating one AND gate and two OR gates, totalling 20 ps. In contrast, the `inv` component, when using dual rail code, is basically two crossed wires, it does not introduce any additional logic to channels, therefore it has a 0 ps delay.

B. Sequential Components

Another class of components used during the dual-rail expansion are sequential components. These implement registers in asynchronous pipelines. They are modelled as flip-flops on the single-rail synthesis, allowing standard synthesis tools to instantiate them from canonical RTL constructions. The sequential components expand to dual-rail pseudo-synchronous WCHB registers, providing endpoints to be used during the SDDS-NCL flow. Sequential elements bind to `drwire` channels actively, performing handshake on their input and output channels. There are three sequential components in the Library: (i) the `dff` half-buffer dual-rail RTZ register; (ii) the `dffr` resetable full-buffer component; and (iii) the `dffs` settable full-buffer component. These are detailed next.

Figure 2a depicts the implementation of a half-buffer dual-rail RTZ register. The half-buffer register is the simplest sequential component and is the base for constructing pipelines. It connects to an input channel d and to an output channel q. The half-buffer consists of: two resetable C-Elements, here named using the NCL-style threshold gate denomination RNCL2OF2; one OR gate (NCL1OF2); and an inverter. When the output channel acknowledges the reception of a null (valid) token, the resetable C-Elements can latch an incoming valid (null) token. This register operates on RTZ channels, valid token codewords are one-hot and null token codewords are presented by all bits in 0. Therefore, an OR gate acknowledges when either a valid or null token has been latched in the input channel. A null token is acknowledged by lowering the ack wire of the input channel and a valid token is acknowledged by rising the same signal. The inverter on the output ack wire enables the alternation between reset and evaluation phases.

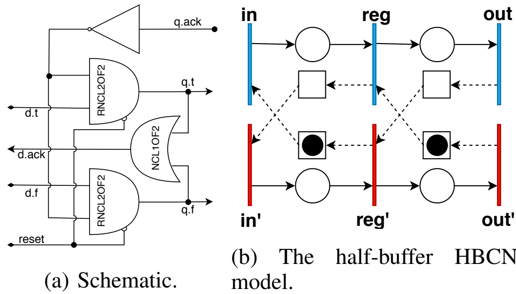
During initialisation it is important to place the circuit in a known state. This is due the fact that all components in the circuit consist of gates with hysteresis, which start at unknown initial states. A QDI circuit with an unknown state may operate improperly, as it may start at an invalid state. Both RTZ and RTO protocols require that combinational components outputs are null prior to entering the evaluation phase. Thus, it is important to initialise all combinational components in the circuit by propagating the null codeword.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Combinational gates in the Components Library.

Component	Rail	Virtual Function	Transition	Virtual Delay
`nand2`	True	$y.t = (a.f \wedge b.f) \vee (a.f \wedge b.t) \vee (a.t \wedge b.f)$	Rise	20 ps
	False	$y.f = a.t \wedge b.t$	Fall	10 ps
`nor2`	True	$y.t = a.f \wedge b.f$	Rise	10 ps
	False	$y.f = (a.t \wedge b.t) \vee (a.t \wedge b.f) \vee (a.f \wedge b.t)$	Fall	20 ps
`xor2`	True	$y.t = (a.t \wedge b.f) \vee (a.f \wedge b.t)$	Rise	15 ps
	False	$y.f = (a.t \wedge b.t) \vee (a.f \wedge b.f)$	Fall	15 ps
`inv` or `buff`	True	$y.t = a.f$ or $a.t$	Rise	0 ps
	False	$y.f = a.t$ or $a.f$	Fall	0 ps

For the reason stated above, half-buffer registers employ resetable C-Elements that initialise their output channels to null codewords. Null codewords propagate through combinational components in cascade, placing the forward propagation logic in a well-known state. Similarly, the backward propagation logic on the input channel must also be initialised. When the register initiates the output rails low, the OR gate sets the acknowledgement signal of the input channel low. This signals that the channel is ready to receive new data, an information which cascades in the backward propagation of the inbound channel, initialising it. This is evident when the HBCN model depicted in Figure 2b is analysed. Here, the inbound channel is represented by four places preceding the register and the outbound by four places succeeding the register. The initial marking represents the initial state of the channel, it marks that both channels are ready to accept new data tokens.

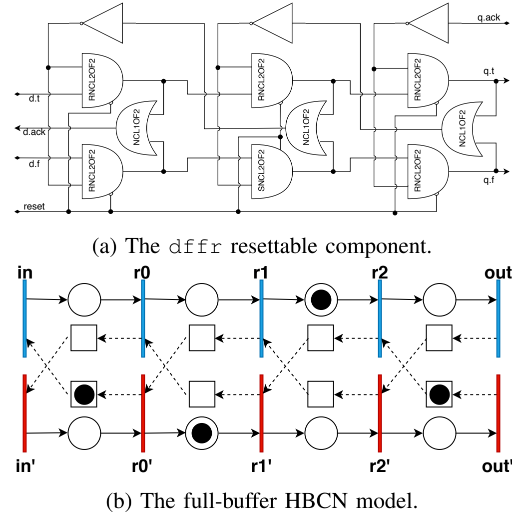


(a) Schematic. (b) The half-buffer HBCN model.

Fig. 2: The `dff` half-buffer dual-rail register.

Sometimes it is necessary to initialise a circuit with data. For example, a counter must initialise to a known data value. This implies initialising some channels with valid tokens. This can be achieved using the full-buffer components depicted in Figure 3. The resetable full-buffer component, depicted in Figure 3a, places a valid false data token in the circuit. Similarly, the settable full-buffer component places a valid true data token in the circuit. A full-buffer component can simultaneously hold a data and a null token. This separates its interfacing channels by a full handshake cycle.

The full-buffer component comprises three half-buffer registers in sequence. These are required to place a data token in the pipeline while correctly initialising the inbound and outbound channels. Propagating data tokens in a circuit at an unknown state would yield invalid results. Therefore, the first and the last are regular half-buffer registers that reset to null. These two registers are responsible for initialising the inbound and outbound channels. Due to the provided isolation, the middle register can safely reset to a data token without compromising circuit initialisation. This is implemented by instantiating a settable C-Element (SNCL2OF2) for either the true or the false rail, depending on the required behaviour.



(a) The `dffr` resettable component.

(b) The full-buffer HBCN model.

Fig. 3: A full-buffer component.

The behaviour of full-buffer components is further evidenced by an analysis of its HBCN model, depicted in Figure 3b. Here, the inbound and outbound channels are initialised to a state where both are ready to accept new tokens, similar to the initial state of the half-buffer register. However, two channels internal to the component are initialised respectively to a data and a null token. These internal channels contain no logic, thus they do not need to be initiated by a null token.

The settable and resettable C-elements employed in full-buffer components are pseudo-flop instances. A pseudo-flop allows breaking the cycles of WCHB pipelines and using STA to analyse the forward and backward propagation paths. This is important during the sequential synthesis part of the SDDS-NCL flow. However, a commercial EDA tool does not safely infer these gates from implicit register construction. Therefore, pseudo-flops are instantiated in the SystemVerilog

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.

module implementing the sequential component expansion. The implementation of sequential component expansions is in fact a technology-dependent step.

Just as in the case of combinational components, sequential ones are also instantiated by name during the single-rail synthesis. They are modelled as D-type flip-flops in the component library Liberty file. The settable and resettable full-buffer components are modelled as flops with preset and reset, respectively. The half-buffer register is modelled as a D-type flip-flop with neither reset nor preset control signals. This approach contrasts with Uncle [6], where half-buffer registers are modelled as latches. Note that according to the authors experience, standard EDA tools do not support retiming latch circuits. Therefore, modelling sequential components as flops additionally enables performing retiming during single-rail synthesis. This balances the amount of components employed in each pipeline stage and opens new opportunities for optimisation in early synthesis steps.

V. HBCN Building and Cycle Time Constraining

Pulsar-F computes the pseudo-synchronous Design Constraints used during the Sequential SDDS-NCL Synthesis Flow (see Figure 1). The flow computes the HBCN model that serves to automatically produce constraints it requires.

The HBCN model is calculated from a structural graph extracted from the single-rail netlist. This is a directed graph that describes the single-rail netlist timing paths. In it each vertex represents sequential components or ports, and each edge represents a channel connecting two registers. Since combinational components are transparent to the handshaking process, they are abstracted. Each vertex has a name, identifying the component it represents. There are three types of vertices: (i) `Port` identifies input and output ports; (ii) `NullReg` signals half-buffer components; and (iii) `DataReg` identifies full-buffer components. The structural graph adjacency list is exported to a file during the single-rail synthesis. Each file line represents a vertex, containing its type, name and a list of successor vertices names.

Since the expansion process is well-defined, it is possible to use the structural graph to construct the HBCN. To this end, it suffices to traverse the structural graph, building the HBCN based on the expansion of components represented by vertices. For each `Port` vertex, a single transition pair is created, which models the (ideal) environment[3]. Each `NullReg` vertex represents a half-buffer register, which is modelled by a transition pair. Each `DataReg` vertex represents a full-buffer component. These comprise three registers in sequence, modelled by a sequence of three transition pairs connected by two channels, one channel initialising with a null token, the others initialising with a data token. For each edge in the structural graph, a channel is created from the last transition pair corresponding to the source vertex to the first transition pair corresponding to the target vertex.

After constructing HBCN, it is possible to apply the linear programming (LP) technique presented in [8] to compute the pseudo-clock and timing exceptions. The pseudo-clock is used to constrain the delays of combinational components in the circuit. However, the full-buffer component contains two internal channels comprising no logic. These internal channels can be individually constrained to a minimal delay value. This minimal delay is parameterisable and depends on the target technology, but it must be enough to cover the delay of a C-element and a NOR gate. The constraining of these paths to a minimal delay affects the computation of the pseudo-clock. If these take part in a critical cycle, they can allow the pseudo-clock constraint to assume a more relaxed value.

The process of computing the timing constraints is handled by the *HBCNConstrainer* program. This program computes HBCN from the structural graph. It uses the HBCN model to define the system of arrival equations constraining the cycle time to a specified target cycle time constraint. The program invokes the GLPK LP solver (https://www.gnu.org/software/glpk/glpk.html) to solve the system of arrival equations. From the solution provided by GLPK, it produces a Synopsys design constraints (SDC) file. This SDC file contains the pseudo-clock constraint used during the Virtual Netlist synthesis. HBCNConstrainer can optionally generate timing exceptions for paths with free slack, allowing more relaxed timing constraints on non-critical paths.

VI. Experimental Results

The frontend for the Pulsar-F flow was initially validated by synthesising and simulating a set of multiply and accumulate (MAC) units under a range of timing constraints. MACs were chosen as example circuits due to their logic complexity and non-linear pipeline structure. Non-linear pipelines are good case studies for evaluating the HBCN timing constraining capabilities, because they present non-trivial maximum cycle times. The accumulator of a MAC comprises a circular buffer. This buffer can be implemented with different numbers of pipeline stages. Four different MAC architectures were thus designed comprising 3 to 6 circular buffer stages. These MACs present the same external interface and behaviour, they multiply two 16-bit numbers from the input and add the 32-bit result in an accumulation loop. The new accumulator value is presented to the output after every computation cycle. This allows using the same testbench for simulating each MAC architecture.

The MACs were synthesised using the Pulsar-F flow from synchronous RTL descriptions. Single-rail synthesis was conducted by the Cadence Genus 18.1 tool with retiming enabled. The optimisation effort was set to extreme and the circuit was synthesised with a nought clock period constraint. Genus was also configured to optimise the total negative slack. These settings were used to minimise the logical depth of each pipeline stage. The single-rail synthesis of the RTL descriptions resulted in virtual netlists with the characteristics summarised in Table II. Here, the worst virtual delays indicate the complexity of the v-function composition for the longest path in the circuit.

The final SDDS-NCL netlists were implemented with NCL and NCLP gates from the ASCEnD-ST65 [11] library for the STMicroelectronics 65nm technology node. This library is

[3]An *ideal* environment provides a input token immediately upon request by any circuit input, and immediately consumes every token produced at any circuit output.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Characteristics of the virtual netlists for each MAC.

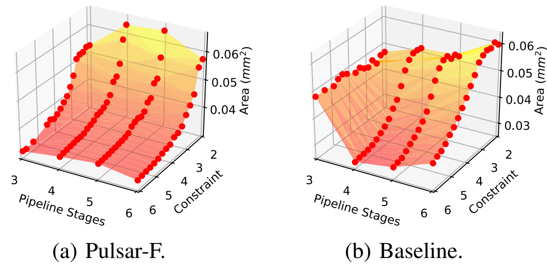
Circuit	Worst vDelay	Component Count		
		Comb	Seq	Total
3-stage MAC	335 ps	2094	206	2300
4-stage MAC	205 ps	2088	267	2355
5-stage MAC	150 ps	2121	344	2465
6-stage MAC	130 ps	2086	442	2508

characterised at three PVT corners: (i) the *worst corner* with slow transistors operating at 0.9 V and 125 °C; (ii) the *nominal corner* with typical transistors operating at 1.0 V and 25 °C; (iii) the *best corner* with fast transistors operating at 1.1 V and −40 °C. Dual-rail synthesis was performed using the worst corner. This synthesis employs a library that models settable and resettable C-Elements as pseudo-flops. The pseudo-flop model introduces a small error on the delay model. A clock uncertainty of 5 ps was used to compensate for this error. Each circuit was synthesised under a range of target cycle time constraints, from 2 ns to 6 ns in steps of 250 ps. The minimal delay was set to 200 ps during cycle time constraint computation. Synthesis was performed using Genus with the effort set to high. Physical-aware optimisation was performed with the effort set to extreme. After running the Fix X-netlist algorithm (the algorithm that corrects errors possibly introduced in the SDDS-NCL QDI netlist during synchronous logic synthesis [8].), physical-aware optimisation was performed on each set of gates iteratively, until the timing was met or a maximum number of 10 iterations was reached.

For comparison purposes, the same set of circuits were synthesised using the methodology employed in [8]. Here, the single-rail synthesis and dual-rail expansion were performed using Uncle [6] with retiming optimisation enabled. Here, the circuit HBCN models were manually computed and their pseudo-clock constraints calculated. The gates from the Uncle netlists were replaced by their equivalent v-functions and pseudo-flop gates. Uncle netlists were synthesised using the same process as the Virtual Netlist generated by Pulsar-F. This yields a set of SDDS-NCL netlists that serve as a *baseline* for comparison. The two netlist sets were produced from the same RTL description and processed by the same backend flow, the only difference between the two being the frontend.

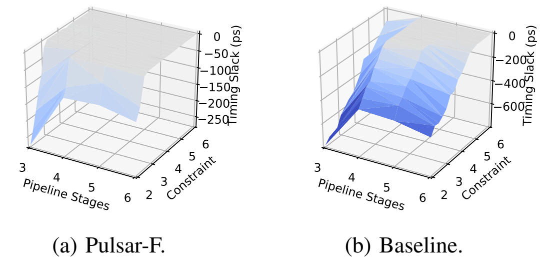
The SDDS-NCL netlists of both sets of experiments were delay-annotated using the three corners from the sign-off library. This library models the settable and resettable C-elements as pseudo-latches. Thus, annotated delays are not affected by the error introduced in the pseudo-flop model. Each delay-annotated netlist was simulated with a testbench emulating an ideal environment, isolating external influences.

From this it is possible to extract data and draw a comparison. For instance, the gate area utilisation depicted in Figure 4 shows that Pulsar-F is able to better take advantage of the timing budget to optimise area. This is especially evidenced in the 3-stage MAC, where the Baseline implementation presents mostly constant area values across the various constraints, while Pulsar-F is able to trade area usage against performance. For the 4-stage MAC, the Baseline shows better area results on



(a) Pulsar-F. (b) Baseline.

Fig. 4: Gate Area comparison.



(a) Pulsar-F. (b) Baseline.

Fig. 5: Worst timing slack as reported by Genus.

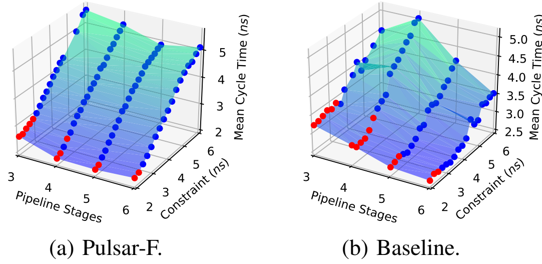
tighter clock constraints, but this is misleading, as the synthesis presented timing violations whereas Pulsar-F could met the timing constraint with some additional area cost (see Figure 5).

The better use of the timing budget by the new frontend is also corroborated by the timing slacks depicted in Figure 5. Here, Baseline rapidly consumes the timing budget provided by the pseudo-clock, achieving a negative timing slack rather quickly. For the same cycle time constraint, the Pulsar-F flow was able to benefit better from the timing budget, achieving a negative slack much later. This behaviour might be due to the virtual delays enabling better logical optimisation and re-timing on the pre-synthesis. It also might be due to the comprehensive design constraints derived from the automatic computation of the circuit HBCN by the Pulsar-F flow.

However, the negative slack in Baseline implementation does not necessarily reflects in cycle time violations. The mean cycle time extracted from the delay-annotated simulation, depicted in Figure 6, shows that both circuits present cycle time violations at similar constraints for each MAC version. This further corroborates the advantages of computing comprehensive cycle time exceptions. The baseline flow relies only on the pseudo-clock for cycle time constraining, which leads to situations where empty buffer pipeline stages receive the same timing budget as pipeline stages that perform computations. The automatic computation of HBCN emits timing exceptions for these special stages, thus allowing a more relaxed pseudo-clock constraint for the same cycle time constraint.

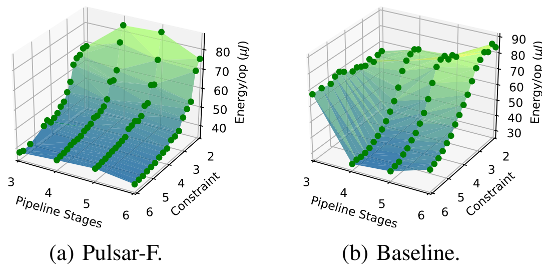
Over-constraining a circuit impacts its energy consumption. Power estimations capitalized on the Synopsys PrimePower tool with the sign-off library and switching activity extracted from delay annotation at the best corner. These results, normalised as energy per operation, appear in Figure 7, showing that Pulsar-F can achieve an overall higher power efficiency compared to the Baseline. This is a step towards the application of QDI in low-power, high-performance applications.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.



(a) Pulsar-F. (b) Baseline.

Fig. 6: Mean cycle time from worst-case delay-annotated simulation. Dots mark datapoints, red dots indicate a cycle time violation. Missing datapoints presented malfunction.



(a) Pulsar-F. (b) Baseline.

Fig. 7: Energy consumption per operation, estimated by PrimePower with activity annotation from the best corner delay-annotated simulation.

VII. Conclusions and Ongoing Work

The experiments show that Pulsar-F improves Pulsar to further aid designers to trade power, area and performance goals. This is enabled by the extensive use of mature EDA tools that take advantage of the available timing budget. The tools that automate the computation of the HBCN model provide additional timing budgets for EDA tools to explore. However, experiments have also shown that using the GLPK solver to compute the constraints is non-scaleable. Preliminary experiments point that adopting more efficient tools like Gurobi as the LP solver can improve computation time. Nonetheless, changing solver alone does not address the current intrinsic complexity of the problem formulation. An ongoing work to address the issue comprises pre-processing the structural graph to collapse parallel registers, reducing the problem complexity and improve scaleability.

Currently, the Pulsar synthesis flow only targets the SDDS-NCL template, but the synthesis flow frontend is designed to be extendable. Ongoing work aims to develop backend flows to support different QDI templates, e.g. SDDS-Velo [12], LCL [13] and others. Another limitation of Pulsar is its lack of formal verification, relying mostly in post-synthesis simulation for validation. A designer has to validate the synthesis result for possible orphan-paths hazards. An automated verification tool that aids the designer to identify possible hazards is highly desirable and is a planned future work.

At the moment, Pulsar supports only deterministic pipelines, i.e. pipelines with no choice. Ongoing work attempts to solve this issue by modelling a limited form of choice using RTL-constructions and additional components. However this approach has presented mixed results so far, with the synthesis tool occasionally selecting these components in undesirable ways. An alternative approach would be to manually instantiate these choice components. However, this could obscure the original RTL description. Also, writing valid RTL targeting asynchronous circuits is not trivial. Not all valid RTL translates to functional QDI circuits, since the clock assumption creates design patterns common to RTL designers that may not map adequately to asynchronous structures. Moreover, RTL synthesisable to a functional QDI using Pulsar-F may in some cases not be suitable to traditional HDL simulation. Proteus [7] presents an interesting solution, by introducing a description language suited for communicating systems and a compiler that produces valid RTL. Search for alternative approaches to provide Pulsar-F with a suitable high level entry format is ongoing.

Acknowledgements

This research was partially funded by the CNPq under grants no. 200147/2014-5 and no. 312917/2018-0 (Brazil).

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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*, July 2009, pp. 23–31.

[6] 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.

[7] 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.

[8] M. L. L. Sartori, R. 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.

[9] N. P. Singh, "A Design Methodology for Self-Timed Systems," Master's thesis, Massacussets Institute of Technology, Laboratory for Computer Science (EEMCS), Cambridge, MA, Feb. 1981.

[10] J. Sparsø, J. Staunstrup, and M. Dantzer-Sørensen, "Design of delay insensitive circuits using multi-ring structures," in *European Design Automation Conference (EURO-DAC)*, 1992, pp. 15–20.

[11] M. T. Moreira, B. S. Oliveira, J. J. H. Pontes, and N. L. V. Calazans, "A 65nm Standard Cell Set and Flow Dedicated to Automated Asynchronous Circuits Design," in *IEEE International System on Chip Conference (SoCC)*, 2011, pp. 99–104.

[12] R. A. Guazzelli, "QDI Asynchronous Design and Voltage Scaling," Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul, FACIN-PPGCC, 2017.

[13] L. D. Tran, T. C. Pham, O. Kavehei, and G. I. Matthews, "Asynchronous 2-Phase Level-Encoded Convention Logic (LCL)," in *2019 International Symposium on Electrical and Electronics Engineering (ISEE)*, Oct 2019, pp. 1–6.

Authorized licensed use limited to: Pontificia Universidade Catolica do Rio Grande do Sul (PUC/RS). Downloaded on November 30,2021 at 11:08:59 UTC from IEEE Xplore. Restrictions apply.