# A Bundled-Data Asynchronous Circuit Synthesis Flow Using a Commercial EDA Framework

Matheus Gibiluka, Matheus Trevisan Moreira, Ney Laert Vilar Calazans
GAPH – Faculty of Computer Science – PUCRS – Porto Alegre – RS – Brazil
{matheus.gibiluka, matheus.moreira}@acad.pucrs.br, ney.calazans@pucrs.br

***Abstract*—Contemporary silicon technology enables integrating billions of transistors and allows the creation of complex systems-on-chip. At the same time, strict power dissipation budgets and growing interest in high performance battery-powered devices drive the need for energy-efficient high performance circuits. Bundled-data asynchronous circuits are good candidates for high performance low power systems, as they operate with average-case delays and present reduced switching activity when compared to other asynchronous templates. The correct operation of bundled-data circuits relies on constraints that describe the timing relationships between data and control signals. However, commercial EDA frameworks do not offer an encompassing support to ensure the closure of such constraints, making implementation challenging. This paper proposes a synthesis flow to enable the description and enforcement of relative timing constraints at both logic and physical synthesis levels, using the Synopsys framework and a set of in-house scripts. Two case studies illustrate the flow: a pipelined multiplier and a network on chip input buffer FIFO, the latter comprising a non-linear pipeline and complex control circuits. Both case studies target the STMicroelectronics 28nm FDSOI technology, and validation occurs with post-layout simulations. Overall, the flow provides an automatic approach to meet relative timing constraints in a template-agnostic manner for bundled-data circuits design.**

## I. Introduction

While the increasing level of integration enabled by contemporary VLSI technology allows the creation of complex multiprocessor systems-on-chip (MPSoCs), designers face strict power dissipation budgets. Additionally, the growing demand for high-performance battery-powered devices increases the requirements for power-efficient circuit implementations. Traditionally, circuits are implemented in a fully synchronous manner, that is, all registers are controlled by a single clock signal that must be properly routed throughout the chip. In modern technology nodes, however, it has become very difficult to design a correct and efficient chip-wide clock distribution tree. In fact, the circuitry required to distribute the clock signal in a high-speed processor accounts, in average, for 45% of total power [1]. Asynchronous circuits are a potential solution for coping with the issues raised by a global control signal, as they do not require a clock, employing local handshaking to perform communication instead.

Different templates are available to design asynchronous circuits, each defined by choices over two essential parameters: (i) a data encoding scheme; and (ii) a handshake protocol. For (i), templates rely on either single-rail encoding, which requires explicit control signals to ascertain data validity, or on multi-rail delay-insensitive (DI) codes, where validity is encoded within data signals [2]. The former encoding requires the use of delay elements (DEs), also called delay lines, to match the timing of data paths and their associated control signals. The latter, on the other hand, allow relaxed timing assumptions, but generally imply added costs in area and power. This difference ends up splitting asynchronous design templates in two major families: bundled-data (BD), and quasi-delay-insensitive (QDI) [3], [4]. Parameter (ii) choices define asynchronous templates relying on either a 2-phase or a 4-phase handshake protocol [4]. Generally speaking, QDI provides more relaxed timing assumptions, but its cost in terms of area and power can be prohibitively high for many applications. BD promises to reduce power and increase performance at area/power costs similar to those of synchronous circuits [5], [6]. BD templates are thus increasingly popular [6]–[8], and different schemes exist for their implementation, e.g. desynchronization [9], Mousetrap [10] and Blade [8].

Unfortunately, BD design is challenging and has scarce support from commercial EDA frameworks. That is due to the fact that these frameworks primarily target synchronous design, with fundamentally different timing assumptions, and rely on the discrete notion of time provided by a global clock signal [4]. Asynchronous design uses local handshake operations, where specific control signals define the validity of each data item. There are three basic challenges for automating the synthesis of BD circuits:

- **Challenge 1**: Design DEs to add to control signals, without compromising performance figures;
- **Challenge 2**: Identify all relevant timing constraints among data and control signals;
- **Challenge 3**: Ensure the meeting of all constraints during circuit synthesis.

The automatic insertion of buffers in synthesis tools can overcome **Challenge 1** and different works have exploited it in the past [6], [9], [11], [12]. Also, techniques similar to those explored for clock distribution in synchronous designs, like the works presented in [13], [14], are useful to cope with **Challenge 1**. **Challenges 2** and **3** are more complex and the degree of automation to cope with these in conventional EDA frameworks is quite limited. This paper addresses **Challenge 3**, by proposing ACDC, a synthesis flow that enables a commercial EDA framework to fulfill a set of relative timing constraints (RTCs) [15] during both logic and physical synthesis. An illustration of the method takes place through two case studies: a pipelined multiplier with linear structure, and a network on chip input buffer (a circular FIFO with non-linear pipeline structure and complex control constraints). Both designs were successfully synthesized in ST-Microelectronics 28nm FDSOI technology.

The remaining of this paper comprises four sections. Section II discusses related work. Section III presents ACDC, explores how to define and ensure timing constraints using the Synopsys EDA framework, and discusses the developed in-house scripts. Section IV presents the two case studies and discusses the obtained results. Finally, Section V presents conclusions and directions for future work.

## II. Related Work

Recent years saw the proposition of methods for synthesizing asynchronous circuits using commercial EDA tools. However, most of these target a specific design or design

978-1-4673-8035-5/15 $31.00 © 2015 IEEE
DOI 10.1109/DSD.2015.104

style. This section reviews previous work on synthesis of BD circuits. Iizuka et al. [12] propose a toolset for synthesizing BD circuits with a resource-sharing model and a Q-module based control. The flow starts with a synchronous circuit, controlled by a Finite State Machine (FSM), and a set of latency constraints. After synchronous synthesis, a non-automated desynchronization step generates a Q-module asynchronous netlist. The designer needs to specify all the paths that must have the timing verified, and the cells that can be used as delay elements, along with their respective delays. Static Timing Analysis (STA) extracts timing information, and DEs are inserted directly on the netlist. After physical synthesis, the flow performs a final timing verification and, if it finds violations, DEs are adjusted using an Engineering Change Order (ECO) flow. Unfortunately, this design flow supports only one design style for BD design (based on Q-modules), and there is no automation for delay matching, because the synthesis tool is not aware of RTCs.

Using a regular Place & Route (P&R) flow, Sotiriou [16] proposes to implement an one-hot-encoded asynchronous FSM. The flow takes a gate-level description of the circuit as input and uses a logic synthesis tool to map it to the target technology. A General Constraints Format (GCF) describes timing constraints, but the format limits constraints applicability only to external top-level pins of the design. Therefore, the method creates block boundaries on each handshake interface, and each block undergoes P&R separately. Dummy pins can be added to help the constraining process. This design flow focuses only on control circuits, and constraint fulfillment is limited to top-level constraints. Thus, even simple designs must be separately synthesized and assembled in a final circuit.

Cortadella et al. propose the desynchronization flow [9] that obtains a BD asynchronous circuit from a synthesizable synchronous HDL specification. STA is used to determine the DE's delay, which is embedded in the controllers. Pessimistic logic-path delays are considered when creating DEs during the P&R flow. After post layout timing verification, DEs adjustment takes place, removing cells from the delay chain to match the post physical synthesis logic-path delay. Unfortunately, the work is limited to desynchronize an otherwise synchronous design, not giving support for setting and fulfilling RTCs. As the DEs are part of the controllers, they can only be matched to the logic path relative to that controller.

Ghiribaldi et al. [6], describe the design and synthesis of an asynchronous BD network on chip router, starting from a low-level asynchronous RTL description. Circuit optimization and delay matching take place during logic synthesis. If after physical synthesis constraints are not met, DE values are updated and P&R is re-executed. Although it mentions constraints and their fulfillment, this work provides little information on how to execute the synthesis process or how to model constraints.

This article proposes a method for modeling and fulfilling RTCs in BD designs using a commercial EDA framework, without any other assumption regarding design style or template employed. Creation of DEs takes place automatically, with no need for manual selection or cell characterization. It supports two synthesis modes: (i) top-down; and (ii) bottom-up. In (i), all the environment and RTCs are defined with respect to the top-level design, which requires less synthesis iterations, as the next Section explains. However, this mode can lead to non-optimal designs when RTC inter-block dependencies exist. In these cases, mode (ii) suits better, as it synthesizes each block of the design separately and has its

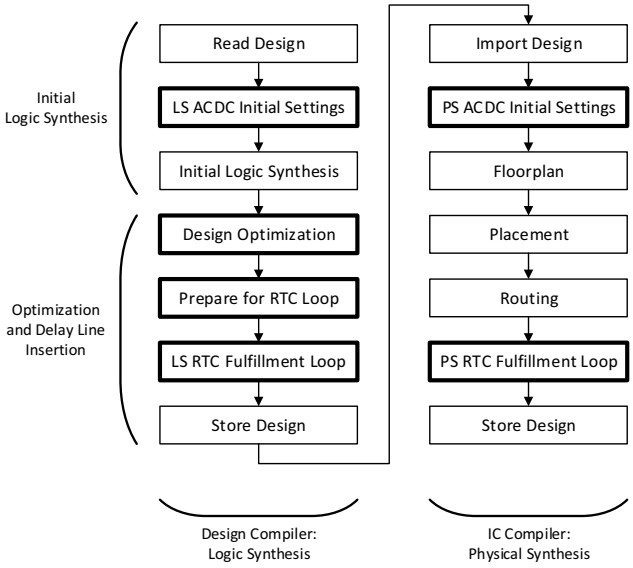


Fig. 1. The ACDC synthesis flow, partitioned in Logic and Physical synthesis. Boxes with thicker borders show ACDC steps, while thinner-border boxes designate regular synthesis steps, performed with commercial EDA tools.

RTCs independently fulfilled. In this way, the latter mode enables a hierarchical and modular approach that eases the process of resolving interlocked RTCs in the hierarchy.

## III. ACDC

The correct operation of self-timed circuits such as BD relies on strict timing relationships between control and data signals. These relationships are of key importance when designing asynchronous BD systems, and can be modeled as RTCs [15]. Commercial EDA tools, however, do not offer support to such types of timing relationships. The ACDC synthesis flow enables the fulfillment of such constraints during circuit synthesis using the Synopsys EDA framework. The flow is based on a set of custom tools that enable the use of STA to translate RTCs into minimum delay constraints (*min_delay*), which are supported by Synopsys Design Compiler and IC Compiler tools.

As Figure 1 shows, ACDC is similar to a regular VLSI synthesis flow, with intermediate steps for optimizations and DE insertion. Thicker border boxes designate steps exclusive to ACDC. The flow comprises three main tasks: Initial Logic Synthesis, Optimization and DE Insertion, and Physical Synthesis. Initially, the circuit description is read and elaborated (Read Design), and logic synthesis settings are defined (LS ACDC Initial Settings). The initial configuration allows the designer to set which libraries and cells the flow is allowed to use during synthesis. Next, the design is mapped to cells from a technology library (Initial Synthesis) and resynthesis is performed to optimize the design based on the constraints set by the designer (Design Optimization). Settings related to DEs insertion (Prepare for RTC Loop) take place before the RTCs are loaded and fulfilled (LS RTC Fulfillment Loop). Finally, the design is stored at the end of the logic synthesis flow (Store Design) These steps are performed within the Synopsys Design Compiler (DC) tool. After logic synthesis, the design is loaded into IC Compiler (Import Design) and the physical synthesis library settings are carried out (PS ACDC Initial Settings). Next, traditional physical synthesis steps take place

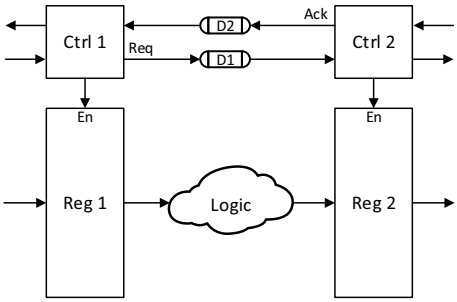


Fig. 2. BD asynchronous pipeline fragment, with DEs on Req and Ack paths.

(Floorplan, Placement, and Routing), followed by an ACDC specific step to verify the RTCs and adjust DEs, if needed (PS RTC Fulfillment Loop). Finally, the design is stored (Store Design). The next Sections detail each step of the flow.

In general, synthesis tools enforce maximum delay constraints, defined by the target operating frequency, and perform setup and hold verification on storage elements. Maximum delay constraints are usually applied on the logic path between registers to ensure the circuit can operate under a given clock period. In addition, some tools also support minimum delay constraints. Commercial tools support both types of constraints, but only allow assigning constant values to these. RTCs, on the other hand, relate delays across two paths, which are not constant values and depend on gate mapping, fan-in and fan-out, physical placement and routing, among other factors. Since RTCs are not required for synchronous circuits and due to their higher degree of complexity, commercial synthesis tools do not offer support to such constraints.

### A. Relative timing constraints (RTCs)

Bundled-data circuits rely on carefully tuned DEs to ensure that handshake events take place only when data is valid – that is, the request signal must arrive only after the data signal is stable. RTCs define signal arrival order, and can be used to fulfill such requirements. In the context of BD circuits, these constraints relate data path and control path: the minimum control path delay must be greater than the maximum data path delay; if it is not, a DE must be inserted in the control path to satisfy that requirement. In addition to DEs on the request path, it may be necessary to add DEs on acknowledge paths as well, to guarantee hold constraints of registers are met.

Figure 2 shows a fragment of a linear BD pipeline, where data coming from *Reg1* goes through some combinational logic before arriving at *Reg2*. Request (*Req*) and acknowledgment (*Ack*) paths contain each a DE (*D1* and *D2*). In this example, the request signal must only arrive at *Ctrl2* after data at the input of *Reg2* is valid and stable. Equation 1 models this relationship: the minimum delay of *D1* must be equal to or greater than the maximum delay of the data signal propagating from *Reg1* to *Reg2* (logic path) added to the setup time of *Reg2*. Likewise, the minimum delay of *D2* must be defined to respect the hold constraint of *Reg2*, as defined by Equation 2. Accordingly, this delay must be greater than or equal to the minimum delay of the data signal propagating from *Reg1* to *Reg2* subtracted from the sum of the hold constraint of *Reg2* and the maximum delay from the *Ctrl2* to *Reg2*. For simplicity sake, this model accounts for wire delays of control paths as part of the DEs' delay.x

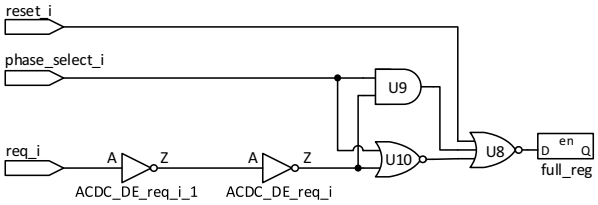


Fig. 3. Example of a bundled-data control circuit where there is a shared path between the data signal (*phase_select_i*) and the control signal (*req_i*). *ACDC_DE_req_i_1* and *ACDC_DE_req_i* show, respectively, the start and end points of the DE.

$$min\_delay(D1) \geq max\_delay(Reg1, Reg2) + setup(Reg2) \quad (1)$$

$$min\_delay(D2) \geq max\_delay(Ctrl2.req, Reg2.en) + hold(Reg2) - min\_delay(Reg1, Reg2) \quad (2)$$

The complexity RTCs present to circuit design greatly depends on the circuit topology. Linear pipelines such as the one in Figure 2 have a very regular structure and no overlap between control and data signals. This makes their constraints easy to extract. In fact, when assuming a pipeline designed with a template such as Mousetrap [10], there are two constraints present on each pipeline stage: one forward (going from stage *i* to stage *i+1*), similar to the one characterized by Equation 1, and one backward (from stage *i* to stage *i-1*), described by Equation 2. On non-linear pipelines (e.g. pipelines with forks and joins), also supported by Mousetrap, a more careful analysis and planning is necessary. For example, Figure 3 shows a fragment of the control logic for the bundled-data circular FIFO proposed in [6]. In this circuit, the *phase_select_i* signal must propagate to the input pin of latch *full_reg* before *req_i* does – that is, following the RTC naming, *phase_select_i* behaves as the data signal, and *req_i* as the control signal. Careful positioning of the DE is mandatory in this circuit, as incorrect placement can prevent correct operation. If the DE is inserted after either one of the logic gates shown in the Figure 3 (U8, U9, or U10), increasing the delay of the control path line would also increase the delay of the data path, creating contention between data and control paths, rendering the constraint impossible to be fulfilled. Therefore, *req_i* needs to be delayed before arriving to the combinational logic that precedes the latch. To guarantee correct placement, a pair of inverters, named *ACDC_DE_req_i_1* and *ACDC_DE_req_i*, are inserted in the *req_i* path. These inverters act, respectively, as the start and end points of the DE – i.e. additional delay elements required by this constraint are only inserted between these two cells.

The RTC model employed by ACDC does not make any assumption regarding circuit implementation style, allowing the creation of timing relationships between any two signal paths on the design. Each RTC comprises one *base* and one *enforced* path, and it is fulfilled if the minimum delay of the enforced path is equal to or greater than the maximum delay of the base path, as Equation 3 defines. Regarding Figure 2, the base path is the logic path between the registers, and the enforced path is that of the *Req* signal. Each signal path is the path going from a start point to an end point, where the start and end points are cell pins or circuit ports. To increase the flexibility when modeling RTCs, enforced and base paths can be defined either by a single path or by a set of paths – which are referred as *base set* and *enforced set*.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<design name="input_interface">

    <constraint type="relative" name="wr_ctrl:: req_i vs. phase_select ">
        <description>data_o vs. wr/full_o to fifo/req_rd for all registers </description>

        <base>
            <path>
                <startpoint>phase_select_i</startpoint>
                <endpoint>full_reg/D*</endpoint>
            </path>
        </base>

        <enforced>
            <set action="sum">
                <path>
                    <startpoint>req_i</startpoint>
                    <endpoint>ACDC_DE_req_i_1/A</endpoint>
                </path>
                <path delayTarget="true">
                    <startpoint>ACDC_DE_req_i_1/A</startpoint>
                    <endpoint>ACDC_DE_req_i/Z</endpoint>
                </path>
                <path>
                    <startpoint>ACDC_DE_req_i/Z</startpoint>
                    <endpoint>full_reg/D*</endpoint>
                </path>
            </set>
        </enforced>

    </constraint>

</design>
```

Fig. 4. XML file format accepted by ACDC. Example describes constraints of the circuit presented in Figure 3.

$$min\_delay(enforced) \geq max\_delay(base) \qquad (3)$$

ACDC expects descriptions of RTCs in the XML format illustrated in Figure 4, which describes the constraints of the example circuit of Figure 3. Each constraint has a unique name, a base group, an enforced group, and an optional description string. A single path can describe each group, as illustrated by the base group in Figure 4. Alternatively it is possible to define a set with several paths and sets nested, as the enforced group in Figure 4 depicts. Each set has an associated *action* property, which defines how to compute the delay of each set. Action *max* makes the set delay equal to the largest path delay contained in the set, enabling the grouping of several parallel paths in one constraint. Action *sum* computes the set delay by summing all path delays contained in the set, allowing the easy description of a group of paths in series (see an example in Figure 4). The use of nested sets and actions allows constraint grouping and the description of complex signal relationships, reducing the number of RTCs needed to describe the circuit.

In addition, to increase description flexibility, it is possible to use wildcards, such as asterisk, when defining paths. Wildcards are directly interpreted by DC and IC Compiler, and follow the behaviour defined for Synopsys tools. To avoid issues with incorrect placement of DEs, the DE insertion point needs to be explicitly set to one of the paths in the enforced set. This path is identified by the *delayTarget* attribute. As STA tools cannot compute timing across DEs, this construct needs to be used to compute the delay in paths with DEs. This happens because the synthesis tool can only apply the timing constraints needed to create the DE at the start and end point of a timing path. Therefore, the timing path at the beginning and end of the DEs is broken during synthesis.

### B. Initial Logic Synthesis

The initial synthesis maps the design to a standard-cell library and prepares it for ACDC. The flow assumes the input design to be the hardware description of a bundled-data circuit, which could be a circuit originally handcrafted, or the output of a flow like [9]. First, DC reads, elaborates, and applies the initial constraints to the design. At this level, the constraints set false paths and define the output loads and input driving cells of the circuit, without setting timing-related constraints. In asynchronous systems, not only the logic function, but also the structure of the circuit defines the system behavior [6]. To guarantee that DC does not make structural changes during synthesis, logic optimizations are disabled with the command *set_structure*. Depending on the used cell libraries, it may be necessary to limit which cells can are to be employed for synthesis – for instance, core libraries are interesting for implementing logic and control circuits, whereas clock libraries, due to presence of delay cells and balanced buffers and inverters, are good candidates for DEs. Such settings can also be applied during this step. After these initial settings, the design can be synthesized and mapped to the target cell library. To ensure that the circuit's hierarchical structure is kept all synthesis are performed with the *no_autoungroup* option enabled.

### C. Optimization and DE Insertion

After the initial mapping, obtained at the end of the Initial Logic Synthesis step, ACDC executes a series of tasks to optimize the circuit and fulfill RTCs. A *timing loop* is a combinational circuit with an unregistered feedback signal – e.g. a Mousetrap [10] handshake control creates a timing loop where the done signal is fed back as the latch enable signal, after passing through an XNOR gate. Since such loops prevent DC from performing STA, DC normally inserts extra buffers in the feedback path and disables the timing analysis through them, breaking the feedback path. Such buffers are called *loop breakers*. The insertion of loop breakers in a path with an RTC can prevent the STA tool to compute the path delay, which may render the constraint impossible to be fulfilled. Accordingly, in ACDC timing loops need to be disabled to avoid the insertion of loop breakers. This can be achieved by disabling the register's timing arc that causes the loop – in the case of a Mousetrap controller, the loop must be broken at the latch, by disabling the arc from the enable pin to the output.

The next step is the optimization of the circuit to fulfill performance constraints. This is done by setting maximum delay constraints on the data paths of the circuit and performing an incremental synthesis. Contrary to synchronous systems, each path can present different max delay constraints, allowing different levels of optimizations throughout the circuit. This increases the freedom a designer has to meet constraints, as important modules can be made fast, at the expense of added area, while non-critical modules can remain slower. After meeting performance constraints, the preparation of the design for DE insertion takes place. As previously mentioned, ACDC requires explicit delimiters to constrain where to insert DEs. Therefore, as in the example of Figure 3, a pair of inverters may need to be inserted in the circuit to create such delimiters. The instance names of the delimiters must match the start and end points of the delay target path specified in the XML file that describes the RTC. Delimiter insertion takes place after the max delay optimization, and can be easily automated for designs with a regular structure. To avoid changes in circuit structure after optimization, the flag *size_only* is set for the whole design. This flag allows DC to change the driving strength of gates, but prevents logic optimizations, i.e. that parts of the circuit be replaced by logically equivalent parts. Before RTCs are loaded, the command *set_prefer -min* allows to set the preferred cells to use as DEs. Additionally, using command *set_cost_priority* allows increasing the priority of minimum delay constraints, used by ACDC to create DEs.

Next, DC loads the XML file that describes the RTCs. The ACDC environment uses a set of in-house scripts to parse and validate the constraints file, and to check for non-existing paths and malformed constraints. If an issue is detected, ACDC dis-

```
# AC_set_constraints Function (Generated Automatically)

proc AC_set_constraints {} {
    # Get Delays
    set AC_path0 [custom_get_delay req_i ACDC_DE_req_i_1/A]
    set AC_path0_min [custom_get_delay req_i ACDC_DE_req_i_1/A min]
    set AC_path1 [custom_get_delay phase_select_i full_reg/D*]
    set AC_path1_min [custom_get_delay phase_select_i full_reg/D* min]
    set AC_path2 [custom_get_delay ACDC_DE_req_i_1/A ACDC_DE_req_i/Z]
    set AC_path2_min [custom_get_delay ACDC_DE_req_i_1/A ACDC_DE_req_i/Z min]
    set AC_path3 [custom_get_delay ACDC_DE_req_i/Z full_reg/D*]
    set AC_path3_min [custom_get_delay ACDC_DE_req_i/Z full_reg/D* min]

    # Set Constraints
    echo "\n*********************************************************** "
    echo " Setting min_delay Constraints: "

    ###### Set Constraints
    # Constraint 'wr_ctrl:: req_i vs. phase_select ' :
    set AC_aux_base [expr $AC_path1 ]
    set AC_aux_enforced [expr [expr $AC_path0_min + $AC_path2_min + $AC_path3_min ] ]
    set AC_aux_delta [expr $AC_aux_base - $AC_aux_enforced + $AC_path2_min ]
    set AC_cnst0 $AC_aux_delta
    echo "\tConstraint 'wr_ctrl:: req_i vs. phase_select ' set. "

    set AC_aux $AC_cnst0
    custom_set_min_delay ACDC_DE_req_i_1/A ACDC_DE_req_i/Z $AC_aux
}
```

Fig. 5. Fragment of the TCL file generated by ACDC to set the constraints defined in the XML file illustrated by Figure 4.

ables the associated constraint and reports this to the user. The integration between DC and the ACDC environment occurs seamlessly through TCL functions that behave similarly to the native functions of DC, but internally access ACDC resources in a user-transparent way. The ACDC function to load the XML constraints file calls an in-house tool, which processes the data and generates a new TCL file with the result. The RTC model is constructed inside this tool, which is described in Python and supports sophisticated data structures, allowing levels of abstraction higher than TCL, which facilitates the manipulation of complex constraints.

The output of the flow is a set of TCL functions to compute, report, and set the minimum delay constraints based on the RTCs defined in the XML file. An example of one of the resulting functions appears in Figure 5, where the minimum DE delay for the circuit in Figure 3 is computed and a minimum delay constraint is set accordingly. Initially, the minimum and maximum delays of the paths related to the constraint are extracted using STA. A modified *get_delay* function that is part of ACDC (*custom_get_delay*) is used to allow the delays to be computed from intermediate pins in the path, and not just from timing start-points to timing end-points, as in the original function. Next, the maximum delay of the base path and the minimum delay of the enforced path are calculated, as specified in the XML file. Finally, the minimum DE delay is computed as defined in Equation 4 and applied as a minimum delay constraint on the DE.

$$delay\_line = max(base) - min(enforced) + min(delay\_line) \quad (4)$$

DE creation is an iterative process that computes and applies the minimum delay constraints on the DE paths, resynthesizes the circuit and checks if constraints were met. Except in situations where the path is both a control and a data path for different constraints, one iteration is usually enough to fulfill the RTCs. Once the design is ready, the flow exports the design netlist and the new Synopsys Design Constraints (SDC) file (containing the min, DEs, and max, datapath optimization, constraints). The flow then continues with the IC Compiler tool to implement the physical layout of the circuit.

### D. Physical Synthesis

The physical synthesis phase of ACDC is very similar to a regular flow. After the design and SDC constraints are loaded, ACDC-related settings are performed. These settings

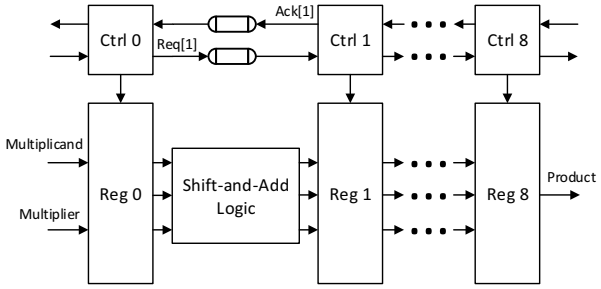


Fig. 6. The 8-bit shift-and-add multiplier implemented in a linear pipeline.

are analogous to the ones on the logic synthesis, such as setting preferred cells to be used as DEs and increasing the priority of minimum delay constraints. Next, a regular floorplan, P&R flow is executed.

Even though RTCs were fulfilled during logic synthesis, after P&R some constraints may require DE adjustments, as DC does not provide a placement-aware delay estimation. Therefore, after the initial physical flow, RTCs must be loaded again through the ACDC environment, and checked for violations. If violations occur, an iterative process to fulfill the constraints must be invoked to fix the DEs. In IC Compiler, the process consists of setting the new constraints using ACDC-generated functions, performing incremental P&R, and fixing possible DRC violations. Since new cells will be added, it is important to set a smaller area utilization ratio when creating the initial floorplan, to accommodate extra DE cells.

## IV. Case Studies

### A. Pipelined Multiplier (Linear Pipeline)

An 8-bit, pipelined, shift-and-add, natural numbers multiplier was used to validate ACDC. The circuit implements a linear pipeline, as Figure 6 illustrates, where each stage performs an addition and a shift operation. The design employs a 2-phase bundled-data template implemented with Mousetrap [10] pipelines. The multiplier takes as input 8-bit multiplicand and multiplier, which are processed through 8 pipeline stages to generate the 16-bit product. This circuit comprises two types of RTCs: request and acknowledge. The request RTC is similar to that presented in Section III (Equation 1), and ensures that a request signal can only arrive at the next pipeline control after the data input on the next register is stable. The acknowledge RTC guarantees that an Ack signal can only be received by the previous pipeline control block after the current pipeline stage has successfully stored the data – e.g. *Ack[1]* only arrives at *Ctrl0* after the input data of *Reg1* is successfully stored.

On the circuit optimization step of the synthesis, a maximum delay constraint of 200ps was set on the logic path of each stage (the base path of the request constraint). Additionally, a 100ps maximum delay constraint was applied to the base path of the acknowledge constraint, which comprises the control logic from the request input, at the pipeline controller, to the register's enable pin. To help reducing the DE slack, a maximum delay constraint 20% above the DE's target delay was added. This helps the tool bounding which cells will be selected to create each DE, reducing performance penalties due to oversized DEs. Table I shows a summary of timing reports for some of the RTCs enforced on the multiplier. Each constraint can be characterized by three values: enforced-path delay, base-path delay, and slack. Slack is the difference between enforced- and base-path delays. A negative slack indicates a constraint violation: the base-path delay is larger

TABLE I. ABSTRACT OF TIMING REPORTS FOR RTCS AT THE END OF SOME SYNTHESIS STEPS OF THE MULTIPLIER CIRCUIT.

| Constraint | | Pre Logic (ns) | Post Logic (ns) | Pre Physical (ns) | Post Physical (ns) |
|---|---|---|---|---|---|
| Req[1] | Enforced | 0.05294 | 0.08689 | 0.08613 | 0.11236 |
| | Base | 0.08678 | 0.08678 | 0.09718 | 0.09147 |
| | Slack | −0.03384 | 0.00011 | −0.01105 | 0.02089 |
| Req[3] | Enforced | 0.05294 | 0.22376 | 0.33025 | 0.21191 |
| | Base | 0.20810 | 0.20810 | 0.21136 | 0.20504 |
| | Slack | −0.15516 | 0.01566 | 0.11889 | 0.00687 |
| Ack[3] | Enforced | 0.08676 | 0.16083 | 0.10193 | 0.14493 |
| | Base | 0.09945 | 0.09944 | 0.08465 | 0.09306 |
| | Slack | −0.01269 | 0.06139 | 0.01728 | 0.05187 |
| Req[9] | Enforced | 0.03104 | 0.03104 | 0.13165 | 0.10296 |
| | Base | 0.0 | 0.0 | 0.00057 | 0.00058 |
| | Slack | 0.03104 | 0.03104 | 0.13108 | 0.10238 |

than the enforced-path delay. The delay figures presented on the table were extracted via STA at the following moments of the synthesis flow:

*Pre Logic* Before the *RTC Fulfillment Loop* step of logic synthesis (refer to Figure 1). In this moment, the circuit is already optimized to meet the performance requirements – i.e. maximum delay constraints were successfully applied to the circuit's logic path.

*Post Logic* At the end of the logic synthesis, when all RTCs have been fulfilled.

*Pre Physical* After the initial P&R, where delay figures are more reliable as STA takes into account the interconnection delay.

*Post Physical* At the end of ACDC, after all RTCs have been successfully fulfilled at the physical synthesis level.

Delay variation between *Post Logic* and *Pre Physical* are due to interconnection and cell placement. Delay figures may increase or decrease as the synthesis tool further optimizes the design trying to meet all constraints – this may change delay margins for already fulfilled constraints. In some cases, as illustrated by constraint *Req[1]*, the delay margin left by the logic synthesis is not enough to fulfill all RTCs, requiring a new *RTC Fulfillment Loop* during physical synthesis. Some constraints, however, are fulfilled in the initial synthesis and do not require further synthesis iterations, as exemplified by *Req[9]* in the last line of Table I. The RTC *Req[9]* relates the product output of the circuit with its associated request signal. During logic synthesis, since no interconnect delay is being considered, the base path for this RTC is zero. The enforced path is composed by the gate delays of buffers added to create the DE boundary (the *Prepare for RTC Loop* step of ACDC). However, when the first steps of the physical synthesis execute, the wire delay referring to the request signal is computed, as column *Pre Physical* shows. The final delay margins (slack values in column *Post Physical*) are in the range of almost 7 to around 100 picoseconds. The 20% maximum delay margin mentioned earlier can help controlling these slacks, but the final result depends on the synthesis tools ability to fulfill all maximum delay constraints set.

For comparison purposes, a synchronous multiplier was synthesized, evaluating its performance metrics. The synchronous implementation uses the same VHDL description as the asynchronous one, except for the control and latches, which were replaced by a clock signal and flip-flops in the former. Synthesis targets the minimum clock period that present zero slack and employs clock gating. Thus, the design had relaxed timing constraints that enabled a good area, power and performance compromise, and allowed a fair comparison. To analyze the circuits performance, the method utilizes exported post-layout netlists. Also, the method employs annotated nets and gate delays, as well as timing simulations of the designs for a burst operation, where each multiplier computes 65,025 successive multiplications. Simulation allowed computing the switching activity of the circuits, exporting this to a VCD file, which is the source to conduct our power analysis on the Synopsys Framework. The environment allows extracting area, performance and power results. A summary of the obtained results appears in Table II.

TABLE II. PERFORMANCE METRICS OF THE ASYNCHRONOUS MULTIPLIER, COMPARED TO ITS SYNCHRONOUS COUNTERPART.

| Metric | Asynchronous | Synchronous |
|---|---|---|
| Forward Latency | 2.6ns | 4.5ns |
| Cycle Time | 579ps | 500ps |
| Burst Time | 37,947.365ns | 32,772.0ns |
| Burst Power | 2.0417mW | 2.5216mW |
| Burst Power-Delay Product | 77.477nJ | 82.637nJ |
| Area | 900.538$\mu m^2$ | 971.856$\mu m^2$ |

Forward latency represents the time to compute a product when the pipeline is empty – that is, without taking into account contentions that could be created by the handshake protocol when the pipeline is full. The asynchronous implementation allows data to flow to the next pipeline stage as soon as the computation on the current stage is ready, reducing the initial latency. The synchronous implementation, on the other hand, is bound by the critical path delay and can only move data to the next stage at the end of each clock cycle. The cycle time of the asynchronous implementation is the average delay between successive request events issued on the output of the circuit, assuming a consumer that is always accepting data and never stalls the pipeline. In the synchronous implementation, the cycle time is the clock period.

The reduced cycle time of the synchronous circuit presents and advantage regarding the time to complete multiplications, as the forward latency advantage of the asynchronous implementation is rapidly amortized by the large number of operations. However, when analyzing power, the asynchronous implementation performs better due to the local handshaking that does not require the fixed switching of a global clock signal. To more fairly analyze these parameters, the overall energy-efficiency of each implementation was computed as the power-delay product. This measure correlates to energy efficiency as it accounts for power and time to compute the product jointly. The asynchronous implementation achieved a smaller power-delay product, suggesting a more energy-efficient circuit that, however, presents a smaller throughput.

### B. Circular FIFO Buffer (Non-linear Pipeline)

The transition signaling circular FIFO proposed in [6] for a network-on-chip input buffer allows evaluating ACDC as applied to a hierarchical design with complex constraints. Differently from the multiplier circuit, this design has constraints inside control blocks, in addition to data-path related constraints. The former are more complex to resolve. Figure 7 shows the block diagram for a 2-place FIFO, composed of four main blocks: read (*rd_ctrl*) and write (*wr_ctrl*) controllers, address pointers (*read_counter and write_counter*), and registers. The synthesized version is an 8-place FIFO with 16-bit words. Read and write addresses use one-hot encoding with pointers generated by ring counters and level-sensitive latches store data.

In this FIFO, each buffer position has dedicated read and write control circuits. These connect to each other with

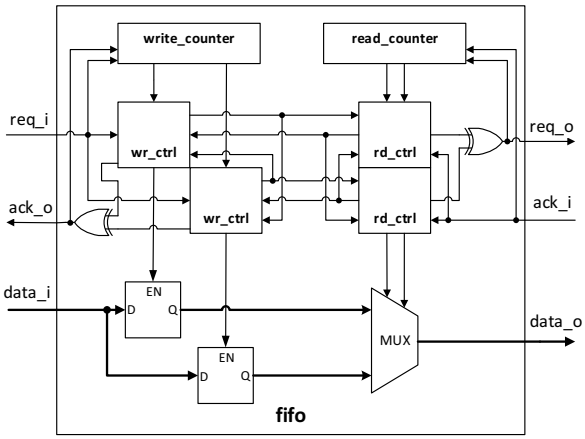


Fig. 7. Architecture of the Circular FIFO circuit, adapted from [6]. The reset signal is omitted.

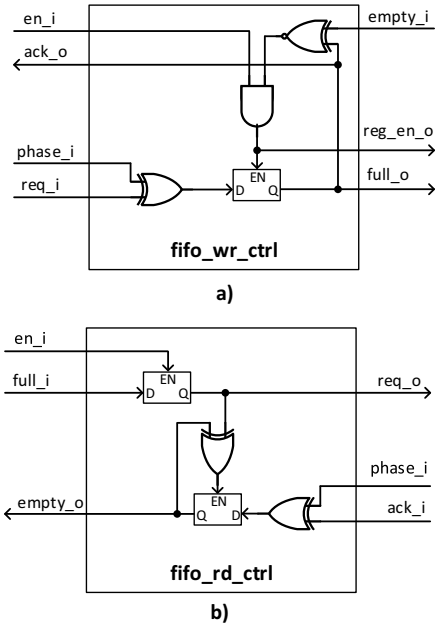


Fig. 8. Circular FIFO control circuits: a) write control circuit and b) read control circuit. Adapted from [6]. The reset signal is omitted.

transition signalling that indicates when each register is empty or full. Figure 8 details the control circuits. When the buffer position is empty and selected to be written, the associated latch becomes transparent (the write controller asserts its signal *reg_en_o*). Once new data arrives (*req_i* transition), the latch becomes opaque to store the data, and the signal *full_o* transitions to indicate to the read controller the register is full. The transition indicating the buffer position is full propagates through the *req_o* signal once the address pointer selects the read controller. At the same time, the MUX selects the appropriate register and propagates its data to the *data_o* output. A transition on *ack_i* indicates that data has been read. This transition propagates to the write controller via a transition of the *empty_o* signal, to indicate the buffer position is ready to accept new data. Since all control is based on transition signaling, phase matching between internal and top-level control signals is required. XOR gates are good candidates for this, as each input transition on the gate results in an output transition. More details about the transition signaled FIFO design are available in [6].

TABLE III. ABSTRACT OF TIMING REPORTS FOR RTCS AT THE END OF SOME SYNTHESIS STEPS OF THE CIRCULAR FIFO CIRCUIT.

| Constraint | | Pre Logic (ns) | Post Logic (ns) | Pre Physical (ns) | Post Physical (ns) |
|---|---|---|---|---|---|
| wr_ctrl phase | Enforced | 0.02692 | 0.03811 | 0.06143 | 0.04815 |
| | Base | 0.03183 | 0.03186 | 0.03511 | 0.03361 |
| | Slack | −0.00491 | 0.00625 | 0.02632 | 0.01454 |
| rd_ctrl phase | Enforced | 0.02684 | 0.03811 | 0.05419 | 0.04656 |
| | Base | 0.03183 | 0.03186 | 0.03348 | 0.03388 |
| | Slack | −0.00498 | 0.00625 | 0.02071 | 0.01268 |
| req_i data_i | Enforced | 0.08699 | 0.09859 | 0.14657 | 0.13109 |
| | Base | 0.0 | 0.0 | 0.00251 | 0.00397 |
| | Slack | 0.08699 | 0.09859 | 0.14405 | 0.12713 |
| req_o data_o | Enforced | 0.10011 | 0.12723 | 0.17532 | 0.22059 |
| | Base | 0.10939 | 0.11064 | 0.13369 | 0.13467 |
| | Slack | −0.00927 | 0.01659 | 0.04163 | 0.08591 |
| full reg_en | Enforced | 0.04840 | 0.09087 | 0.09144 | 0.08557 |
| | Base | 0.06795 | 0.06794 | 0.07176 | 0.07408 |
| | Slack | −0.01955 | 0.02292 | 0.01968 | 0.01150 |

Table III illustrates the fulfillment of selected RTCs for the circular FIFO circuit at the same synthesis points used in the multiplier circuit. All constraints in this Table refer to buffer position 0. Besides RTCs related to data propagation from input to registers (*req_i data_i*), and from registers to output (*req_o data_o*), this circuit has constraints inside and between control blocks. In fact, some top level constraints take into account the delay on the controller's enforced path to compute their base and enforced paths, creating overlapping constraints across the circuit hierarchy. Read and write controller blocks have one RTC each: the *phase_i* signal must arrive at the latch to which it is connected before the *req/ack_i* signal does – this is similar to the example depicted in Figure 3, which is a fragment of the write controller (the difference lies in the fact that the *reset_i* signal is omitted in Figure 8 and in the way the synthesis implements the XOR gate). These constraints correspond to *wr_ctrl phase* and *rd_ctrl phase* in the Table. The constraint *full reg_en* guarantees that data is properly stored at the latch (latch enable signal propagation) before the *full* signal propagates to the read controller, which indicates that data is available. This is an example of a constraint that crosses hierarchical boundaries, as it starts on the write controller, takes into account delay propagation on the top level, and finishes on the read controller. Other constraints exist and were enforced during synthesis, but were omitted from this discussion for the sake of clarity and simplicity.

A hierarchical bottom-up flow was used to synthesize the circular FIFO. Initially, controllers were synthesized and their constraints were fulfilled. Next, the post-synthesis controller netlists were instantiated and the top-level buffer was synthesized. This technique optimizes the use of DE in situations of constraint overlap, as previously mentioned, allowing top level constraints to take into account realistic delays of the inner modules. Maximum delay constraints of 100ps were added to the input and output data paths in the optimization step of the flow. The delays presented in Table III illustrate the same synthesis behaviour depicted in the synthesis of the multiplier, where delays are matched during logic synthesis and further refinements are made in the physical synthesis to take into account placement and interconnect delays. Changes from *Pre Physical* to *Post Physical* columns are due to further optimizations made by the synthesis tool to fulfill constraint violations that are not shown in the Table. Post synthesis results appear in Table IV. Power measurements were made with a full FIFO operating at maximum throughput.

### C. Discussion and Guidelines

DEs are critical in BD systems, as there is a large number of them throughout the circuit and they directly impact the

performance of the system. With the evolution of silicon technology and the trend for low power design, it becomes very difficult to find buffers and inverters with balanced rise and fall time on core libraries, as they are typically power hungry. Also, unbalanced DEs may result in performance degradation, specially for 2-phase BD circuits, as all constraint calculations are performed considering the minimum delay of the cells. Cells from clock tree libraries, on the other hand, are balanced to reduce clock skew, and can be leveraged when creating DEs. Additionally, clock libraries contain delay cells that usually present a smaller area footprint than a series of buffers adding to the same delay, and such cells can be used to create more efficient circuits. In the experiments performed here, the tool was set up to prefer the use of cells from the clock library for minimum delay constraints (i.e. DEs). Therefore, we advise employing these cells for BD designs.

A strategy to ease the description and fulfillment of RTCs is to keep the design hierarchical. Asynchronous circuits can be generalized as modules that exchange information through handshake. By keeping clear boundaries between components, as in the multiplier example, the minimum and maximum timing constraints affect only that block, easing the fulfilment of constraints. The same approach can be applied to ease manipulation of constraints for more complex control circuits, such as the FIFO control logic. These can be split in smaller circuits with simpler constraints, synthesized and then assembled together hierarchically. In hierarchical designs, however, higher level RTCs may be constrained by enforced paths of lower level constraints. This constraint overlap may cause redundant DEs to be added at the top level of the design due to constraints not yet fulfilled in the inner modules. In such cases, a bottom-up synthesis flow becomes an interesting approach. In fact, this approach was taken for the circular FIFO synthesis, where controllers were synthesized individually, before the top-level circuit synthesis. By guaranteeing the RTC fulfillment at lower level design blocks, constraints that depend on paths on those blocks may be already fulfilled, avoiding the insertion of extra delays in the circuit.

The synthesis results for both circuits showed the ability of ACDC to model and fulfill RTCs using the Synopsys Framework, considering different degrees of constraint complexity. The flow could be extended to support other EDA frameworks as long as they support minimum delay constraints. Note that if the design requires special cells, such as C-elements or MUTEXes, ACDC assumes that a library of such components [17] is available. Currently, the designer has to provide the XML file that describes the RTCs, which means that a deep understanding of the circuit behaviour is required. This process can be easily automated for linear pipelines, such as the multiplier. However, for complex circuits such as the circular FIFO, extraction of RTCs is not a trivial task. Nonetheless ACDC stands off by allowing the automatic fulfilment of these constraints.

## V. Conclusions

This paper proposed ACDC, a synthesis flow to enable the description and fulfillment of RTCs in BD circuits using a commercial EDA framework. The flow provides a generic way to specify RTCs, increasing the degree of automation when designing BD asynchronous circuits without the need for *ad hoc* solutions. Two circuits of different complexity were successfully synthesized using ACDC and targeting a 28nm technology node, at both logic and physical levels. The multiplier circuit represents the relevant class of linear pipeline circuits, and the buffer design illustrates how the flow is able to fulfill constraints in both data and control paths of the circuit. The proposed flow stands off by allowing the automatic fulfillment of RTCs in a template-agnostic manner, increasing the degree of automation for bundled-data circuit designers. As future work, enabling automated RTC extraction will further improve the capabilities of this synthesis flow and ease the process of designing BD circuits. Also, an analysis of how the circuit behaves under variability will help determine the level of delay margins required on DEs.

TABLE IV. Performance metrics for the transition signaling, 16-bit word, circular FIFO with depth 8.

| Metric | Asynchronous Circular FIFO |
|---|---|
| Forward Latency | 399ps |
| Average Cycle Time | 634ps |
| Power @ Max. Rate | 0.4577mW |
| Area | 728.362$\mu m^2$ |

## References


[1] M. Amde, T. Felicijan, A. Efthymiou, D. Edwards, and L. Lavagno, "Asynchronous On-chip Networks," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 273–283, Mar 2005.

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

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

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

[5] K. Stevens, D. Gebhardt, J. You, Y. Xu, V. Vij, S. Das, and K. Desai, "The Future of Formal Methods and GALS Design," *Electronic Notes in Theoretical Computer Science*, vol. 245, no. 0, pp. 115–134, 2009.

[6] A. Ghiribaldi, D. Bertozzi, and S. Nowick, "A transition-signaling bundled data NoC switch architecture for cost-effective GALS multicore systems," in *DATE*, March 2013, pp. 332–337.

[7] M. Amde, I. Blunno, and C. Sotiriou, "Automating the Design of an Asynchronous DLX Microprocessor," in *DATE*, June 2003, pp. 502–507.

[8] D. Hand, M. Moreira, H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, B. M., N. Calazans, and P. Beerel, "Blade - A Timing Violation Resilient Asynchronous Template," in *ASYNC*, May 2015.

[9] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Transactions on Computer-Aided Design*, vol. 25, no. 10, pp. 1904–1921, Oct 2006.

[10] M. Singh and S. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on VLSI Systems*, vol. 15, no. 6, pp. 684–698, June 2007.

[11] A. Saifhashemi, D. Hand, P. Beerel, W. Koven, and W. Hong, "Performance and Area Optimization of a Bundled-Data Intel Processor through Resynthesis," in *ASYNC*, May 2014, pp. 110–111.

[12] M. Iizuka, N. Hamada, H. Saito, R. Yamaguchi, and M. Yoshinaga, "A Tool Set for the Design of Asynchronous Circuits with Bundled-data Implementation," in *ICCD*, Oct 2011, pp. 78–83.

[13] J. Liu, S. Nowick, and S. Mingoo, "Soft MOUSETRAP: A Bundled-Data Asynchronous Pipeline Scheme Tolerant to Random Variations at Ultra-Low Supply Voltages," in *ASYNC*, May 2013, pp. 1–7.

[14] S. Gupta and S. Sapatnekar, "Variation-Aware Variable Latency Design," *IEEE Transactions on VLSI Systems*, vol. 22, no. 5, pp. 1106–1117, May 2014.

[15] K. Stevens, R. Ginosar, and S. Rotem, "Relative Timing," in *ASYNC*, 1999, pp. 208–218.

[16] C. Sotiriou, "Implementing Asynchronous Circuits using a Conventional EDA Tool-flow," in *DAC*, 2002, pp. 415–418.

[17] M. Trevisan Moreira, M. Arendt, A. Ziesemer, R. Reis, and N. Vilar Calazans, "Automated synthesis of cell libraries for asynchronous circuits," in *Integrated Circuits and Systems Design (SBCCI), 2014 27th Symposium on*, Sept 2014, pp. 1–7.