Validating an Automated Asynchronous Synthesis Environment with a Challenging Design: RISC-V

Willian Analdo Nunes*, Marcos Luiggi Lemos Sartori*, Matheus Trevisan Moreira*, Fernando Gehm Moraes*, Ney Laert Vilar Calazans†
*School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
†PGMicro, Federal University of Rio Grande do Sul – UFRGS – Porto Alegre, Brazil
{willian.nunes,marcos.sartori}@edu.pucrs.br, matheus.trev@gmail.com, fernando.moraes@pucrs.br, nlvcalazans@inf.ufrgs.br

Abstract—**The coupling of synchronous digital design techniques with electronic design automation tools and support systems ensured several decades of steadfast evolution of electronics based on semiconductor devices, where humankind now depends on products derived from this evolution. As technology nodes advance towards semiconductor physical limits, the obtainment of better circuits faces challenges such as increased variability. Addressing this and other challenges requires new and/or alternative devices and/or tools, and/or design techniques. Asynchronous quasi-delay insensitive design potentially ensures automatic or streamlined adaptability to several variations plaguing digital design. The Pulsar environment proposes using commercial design automation tools and specific software to enable the high-level design of complex circuits and their automated semi-custom implementation. Several small to medium circuits such as multiply and accumulate pipelined operators, first-in-first-out modules and others helped develop and validate Pulsar's basic functionality. The main contribution of this paper is to demonstrate the feasibility of using an evolved version of Pulsar to design and automatically implement more complex asynchronous quasi-delay insensitive circuits, such as processors, starting from RTL-like descriptions. Results include 174 post-synthesis/post-layout RISC-V processor simulations, originating from 29 synthesis processes, for cycle-times varying from 10ns to 3ns, in 0.25ns steps.**

Index Terms—**Asynchronous design, Electronic design automation, RISC-V, Digital circuits, Automated synthesis.**

I. Introduction and Related Work

Nanometer scale design of integrated circuit (IC) technologies bring uncertainties with them, related to several aspects. Manufacturing process variations need to be considered more closely, together with environmental conditions such as voltage supply, temperature variations [1], and even ionising radiation incidence. Besides, design concerns such as IR drop and long wire delays become more relevant. IC lifetime can also be negatively affected by new technologies, requiring to consider ageing-aware design techniques.

Synchronous techniques prevail in most digital circuit domains, but they are increasingly stressed by the mentioned uncertainties, which lead to proposing sophisticated techniques to keep the synchronous paradigm usable [2]. If alternative techniques can guarantee e.g. automatic adaptability of the circuit to voltage scaling, the overhead caused by the previous techniques can be avoided. For example, reference [3] describes an experiment towards such an alternative, where asynchronous design (quasi-delay insensitive or QDI) resilient to voltage scaling allows correct operation of circuits over a range of supplies, from sub-threshold to nominal.

This work was financed in part by CAPES (Finance Code 001), CNPq (grants 309605/2020-2, 317087/2021-5, 311587/2022-4 and 407477/2022-5), and FAPERGS (grants 21/2551-0002047-4 and 22/2551-0000570-5).

Although academic and industrial communities have been prodigal with proposals of the RISC-V family of instruction set architectures (ISAs) in the last years, few of these are asynchronous implementations. Three such proposals in the literature are revised herein. Li et al. [4] describe a bundled-data (BD) asynchronous implementation of a RISC-V RV32I architecture. They use the well-known *clique* BD template [5] and devise a method for defining timing constraints with multiple virtual clocks to enable applying commercial tools in the design and optimisation phase. Authors report a 3× improvement in dynamic power over the equivalent synchronous version implementation. But results are limited to FPGA implementation and no evaluation of adaptability to variations is discussed. Babu et al. [6] propose the design of a 32-bit asynchronous RISC-V using the well-known BD micropipeline template [7], but the design is incompletely described, addresses FPGA implementation only and has just been simulated at the RTL level. Kim [8] proposes the implementation of an asynchronous QDI RV32IM architecture using both academic and commercial, dedicated tools. The design uses the Null Convention Logic (NCL) design template [9]. Area comparison of small synchronous implementations reveals the QDI implementation has an area overhead of 2.5× larger when implemented using the commercial NCL tool and around 7× larger when implemented with the academic tool.

QDI design is in general more robust than synchronous design against delay variations and can handle PVT effects better. This is also true among asynchronous circuit templates, where QDI in general surpasses bundled-data (BD) design. The downside is that the design of QDI circuits usually requires laborious manual work and knowledge of convoluted asynchronous design techniques. Asynchronous design automation is evolving but has a long way before it can compete with current synchronous design techniques.

Pulsar was initially associated with an asynchronous template called Spatially-Distributed, Dual-Spacer NCL (SDDS-NCL), proposed to support the use of available commercial EDA frameworks e.g., Cadence or Synopsys. It employs standard synthesis, optimisation, and static timing analysis (STA) tools for determining the asynchronous cycle-time of the circuits it generates. Pulsar code is open-source, available through Github [10], and capitalises on the ASCEnD-

FreePDK45 [11] open-source asynchronous std cell library.

The main *contribution* of this paper is a demonstration of the feasibility of an evolved version of Pulsar to design and automatically implement complex asynchronous QDI circuits starting from RTL-like descriptions. A complete, asynchronous QDI, operational organisation for the RISC-V RV32I instruction set architecture (ISA) is a side contribution.

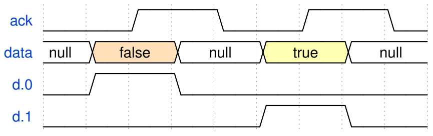
The article is divided into four main sections. Section II introduces asynchronous design and the Pulsar flow. Section III delves into the implementation of the asynchronous processor based on the RISC-V organization, highlighting enhancements made to the Pulsar flow. Section IV presents the processor design validation process, discussing synthesis parameters, logical and physical synthesis results, as well as simulation outcomes using benchmarks. Section V concludes this paper and suggests future directions.

II. Asynchronous Basics and Pulsar Design Flow

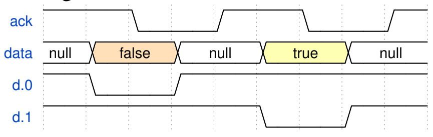
Synchronous circuits rely on a global clock signal providing a discrete common time reference. Typically, the clock is a wave with a period larger than the worst combinational logic delay in any path in the circuit between two consecutive temporal barriers, e.g. registers. All synchronous circuit registers simultaneously capture data (subject to clock skew and jitter), as determined by clock transitions at the temporal barriers. This must guarantee that the computation will have finished when registers capture data. Asynchronous circuits have no single common timing reference, communication takes place through local *handshakes* [12], which avoids the global clock requirement. This also produces circuits that operate based on the average delays of combinational blocks, not on the worst-case circuit path, as is the case in synchronous circuits.

Handshake protocols operate in two steps: (i) data *request* and (ii) data *acknowledgement*. Using dedicated control signals separate from data lines characterises *Bundled Data* (BD) asynchronous design styles. BD allows building simpler data paths that are similar to the ones of synchronous circuits. Requests must arrive at the consumer only after computation on data is concluded and results are ready at the inputs, otherwise a consumer may get incorrect data. The challenge is that request lines may need to be delayed to guarantee correct operation. Accordingly, BD design often rely on the use of specifically designed *delay lines*, components required to adapt the circuit control flow to the data flow in the circuit. The design of optimised delay lines demand careful stage-wise analysis and may require significant timing margins to account for circuit sensitivity to variations.

To avoid or reduce sensitivity to variations and the added delay line design complexity, control information can be embedded within the data flow, using special data encoding schemes, known collectively as delay insensitive (DI) codes. Binary DI codes use only part of the Boolean encoding spectrum possible over n bits. It is well known that an n-bit binary code allows representing 2^n distinct *codewords*. A DI code pledges the use of only a subset of these codewords to achieve the delay insensitivity property. Verhoeff [13] explores the basic details of the theory behind DI codes. Practical n-bit DI codes consist of a set of codewords to stand for data, leaving the remaining n-bit combinations as invalid.



(a) 4-phase return-to-zero (RTZ) DI handshake, active-high *ack*.



(b) 4-phase return-to-one (RTO) DI handshake, active-low *ack*.

Fig. 1: Two examples of asynchronous handshake protocols using a single bit dual-rail code: (a) RTZ and (b) RTO.

Figure 1 depicts the operation of two asynchronous handshake protocols using DI codes; others exist. The examples illustrate the simplest possible handshake structure and behaviour. Communication is unidirectional (with a sender and a receiver), the data channel transports a single bit at a time using a dual-rail code (two wires, *d.0* and *d.1*, represent one bit), and just a single control wire is necessary (the *ack* signal). The examples assume the sender produces data and the receiver produces the *ack* signal. Also, between any two consecutive data transmissions there is a null phase, where data lines return to a *rest* situation (referred to as *null* or *spacer*), using one of the DI invalid bit configurations (*d.0=0* and *d.1=0* for RTZ and *d.0=1* and *d.1=1* for RTO). Note the data encoding is different for each protocol, and the *ack* polarity is reversed in RTO with regard to RTZ. Note also that the *data* signal does not exist, it is included in the Figure just to clarify the ongoing communication phase.

Circuits that embed control information within data and use asynchronous handshake communication protocols as those depicted above follow either Delay Insensitive (DI), Quasi-Delay Insensitive (QDI) or Self-Timed (ST) asynchronous design *templates*. The DI template is ideal for maximum robustness, but it was demonstrated to be of little use [12]. QDI arises from a least compromise between robustness and design feasibility. QDI is a class of templates that perform computation on DI-encoded information. QDI designs require far less restrictive timing assumptions than BD designs. This makes QDI circuits less sensitive to process, voltage and temperature (PVT) variations, and to ageing. There are several proposed QDI templates [12]. QDI circuit implementations are often criticised when compared to synchronous circuits, due to their large overheads in area or/and speed or/and power. ST templates can improve such circuit metrics, while compromising part of the robustness achievable with DI and QDI templates. DI, QDI and ST techniques define circuit classes with the same respective names. The properties of such circuits allow stating that every DI circuit is a QDI circuit and every QDI circuit is an ST circuit; using set notation: $DI \subset QDI \subset ST$.

A popular and well-established template to produce asynchronous QDI circuits using the RTZ communication protocol is the Null Convention Logic (NCL) [9]. NCL relies on a semi-custom design technique that requires using a specific family of logic gates with hysteretic behaviour. However, logic design with hysteretic gates is not directly supported by commercial tools from e.g. Cadence or Synopsys. A typical hysteretic threshold NCL gate is described by a 3-valued function that outputs "0" (the NULL value) when all its inputs are "0", outputs "1" when a weighted sum of the "1"s at its inputs reaches the gate threshold and holds the previous output value otherwise (i.e. a hold or "H" value). Accordingly, NCL synthesis tools and cell libraries are specifically designed, as is the case of the open source Uncle environment [14]. Unfortunately, Uncle and other efforts employ a limited set of optimisations and thus do not compete favourably with powerful commercial tools in terms of resulting circuits and in the ability to allow design exploration.

A. *The Pulsar Design Flow*

Moreira et al. [15] proposed SDDS-NCL, a QDI template that allows circumventing the ignorance of commercial tools about hysteretic behaviours. Extending the NCL gate set with support to both RTZ and RTO protocols enabled this, which doubles the gate library size, but allows using unate function synthesis and optimisation from commercial tools seamlessly to design QDI and ST circuits.

The Pulsar design flow [16] proposes automating the synthesis of ST circuits from some asynchronous design template. Pulsar comprises a set of scripts and adjacent tools to leverage conventional EDA tools to synthesise asynchronous circuits; an open source version is available at [10]. Similar to Uncle, Pulsar also performs synthesis in two steps: (i) the front end produces an intermediary single rail netlist from an RTL-like description; (ii) the back end performs dual-rail expansion and technology mapping. Unlike Uncle, Pulsar employs commercial EDA tools extensively at both synthesis steps.

Pulsar's front end uses Genus to synthesise the RTL-Like description to an intermediary single-rail netlist comprising components and wires. Components are virtual gates with well-defined behaviour and interface; they are the combinational and sequential ST logic gates available to build the circuit.[1] These components are presented to Genus in a *Liberty* file with dummy delays. However, they are not implemented in the front end but in the back end as SystemVerilog modules. The front end uses a custom tool to process the single-rail netlist and to produce a virtual netlist. The tool also replaces all wires on the netlist with instances of a *SystemVerilog interface*[2] that models handshake channels between components; the implementation of this interface is also delegated to the back end. The resulting virtual netlist comprises component modules connected by handshake channel interfaces.

The implementation of the handshake channel SystemVerilog interface is protocol dependent; it comprises the data and control wires to operate a channel. This interface contains two *modports*[3] to connect consumer and producer modules. The SystemVerilog modules that implement components contain the necessary logic to interact with handshaking channels and implement its behaviour. The construction of ST circuits requires that the protocol, sequential and combinational components work in tandem. There are multiple ways to implement any such parts, and a working combination of them is called an *asynchronous design template* or *template* for short. The Pulsar back end supports synthesising ST circuits using different templates by changing the components and dual-rail channel interface implementations. The back end uses Genus to synthesise the virtual netlist together with the components and handshake channel implementations to produce the final ST circuit netlist. The employed gates in this netlist and the exact synthesis process depend on both the template and the target technology choices. The template choice impacts most circuit characteristics, such as area, power and robustness.

This work restricts attention to circuits produced from the SDDS-NCL design template. This was the first template supported by Pulsar, which contains mostly stable code; it is thus an adequate choice to achieve the main goal of this work, validating the viability of using the Pulsar flow for automatic synthesis of complex asynchronous circuits. Section V explores the current perspective on using alternative ST templates to improve circuits metrics with Pulsar.

III. The ARV Asynchronous RV32I RISC-V

This Section describes the proposed asynchronous RISC-V processor organisation, ARV. ARV originated from a concurrent communication processes description [18]. It exploits the inherent structure of asynchronous circuits to enable instruction level parallelism. Figure 2 depicts the ARV organisation. It comprises two main inter-wound loops, the control and the datapath loops. A third loop, the register lock loop, resolves data hazards within the pipeline.

The *control loop* is responsible for the program execution flow, beginning at the *Fetch Unit*, which contains the *Program Counter* (PC) and ending at the *Retire Unit*; the control loop is closed by the *Jump Address* channel between these units. Instruction fetch addresses are generated at the pipeline first stage, where they receive a *stream tag* to identify the instruction execution flow. When a jump occurs, a new program counter value is received through the Jump Address channel and the stream tag is updated; instructions holding an outdated stream tag can then be discarded.

The *datapath loop* is responsible for performing computation and handling data hazards. The datapath loop begins at the *Hazard Detection and Operand Fetch Unit* and terminates at the *Register Bank*; the loop is closed by the Register Bank. After instructions are fetched and decoded, they enter the Hazard Detection and Operand Fetch Unit to retrieve operands data from the Register Bank, when these are required. The *Register Locking Queue* keeps track of pending writes to the Register Bank to detect possible data hazards. A hazard avoidance mechanism is part of the register lock loop. If the current instruction depends on data of a pending write,

[1]Basic components are described in [16]. Reference [17] extends the list.

[2]An interface is a SystemVerilog construct that enables bundling together wires that are logically related, e. g. the wires of a system bus in a SoC.

[3]In SystemVerilog, a *modport* defines how modules interact with interfaces, e.g. defines signal directions for modules with different bus roles.

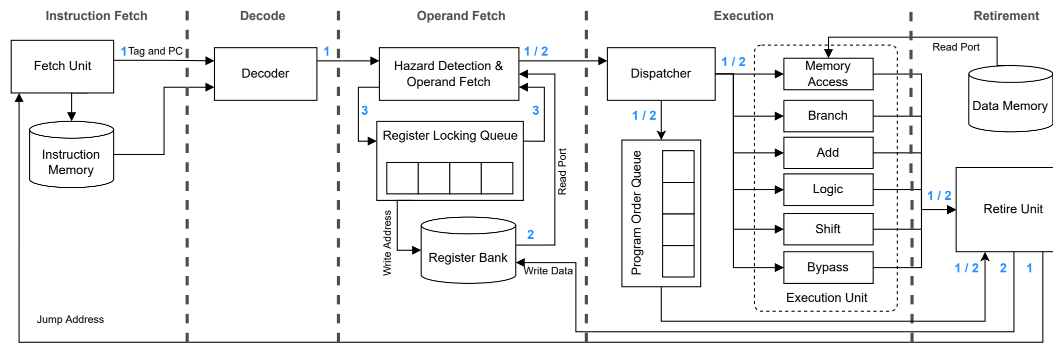


Fig. 2: The ARV processor organisation. There are five stages, identified and separated by vertical dashed lines. Numerals in edges show elements that take part in three processing loops: (1) control loop; (2) datapath loop; and (3) register lock loop.

the former is stalled at the Hazard Detection and Operand Fetch Unit and bubbles are issued until the required data is ready. Once all operands are available, the instruction is dispatched to the suitable *Execution Unit*. Here, a limited form of parallelism can occur, as instructions with different speeds may execute concomitantly. To recover the execution order, a program ordering queue parallel to the *Execution Unit* is used to preserve the program order, thus allowing the *Retire Unit* to retire them in order.

A. The Asynchronous RISC-V Organisation Implementation

The original, high-level, open-source functional model of the processor was written in the Go language (https://go.dev/) [18]. Next, the RTL-like SystemVerilog language subset was defined by the Authors. It relies on a set of components with known *dual-rail* expansions. The ARV implementation results from a translation of the Go description to an equivalent Pulsar RTL-like SystemVerilog description; It corresponds to around 1,400 lines of code distributed along 15 source files.

The development of ARV required a concomitant effort to enhance the Pulsar flow, which now comprises a new set of abstract components to provide choice, components which are described in [17]. Choice is the ability of dynamically change the flow of data in pipelines. This evolution of Pulsar was fed back by the ARV design process, and produced a new version of the flow, adapted to deal with more complex asynchronous data and control flows. A total of six new components are available: two components for decision making (`arbiter` and `probe`) and four components for token steering (`hold`, `discard`, `condhi`, `condlo`); see details of these in [17]. An `arbiter` randomly selects one of two tokens available on its input channels. A `probe` detects the presence of tokens in a channel. `Discard` and `hold` selectively control the propagation of data in the pipeline. `Condhi` and `condlo` are conditional token sources.

The ARV design employs three of the new components extensively: `probe`, `hold` and `discard`. These are used e.g. at the: (i) PC - a `probe` detects the presence of a new Jump Address, `discard` and `hold` enable loading a new value to the PC; (ii) Hazard Detection and Operand Fetch unit - `discard` and `hold` allow stalling an instruction whilst inserting bubbles; (iii) Dispatcher - `discard` allows choosing one of the modules in the Execution Unit to receive an instruction; (iv) Retire Unit - `hold` helps reordering instructions arriving from individual modules in the Execution Unit; and (v) Register Bank - `discard` and `hold` help in selectively write only in the correct register.

To illustrate some details of the ARV design process, and exemplify the use of some abstract components in its design, Listing 1 shows the Pulsar RTL-like description of the 32-bit PC module, part of the *Fetch Unit*. Figure 3 depicts the corresponding token flow diagram for this module.

Listing 1: The Pulsar RTL-like ARV PC description.

```
module PC #(WIDTH=32)
    (input  wire                 clk, reset,
     input  logic [WIDTH-1:0] load,
     output logic [WIDTH-1:0] out);
    logic [WIDTH-1:0]          acc, sum, load_reg;
    wire  [WIDTH-1:0]          new_val;
    wire                       selector;

    always @(posedge clk or negedge reset)
      if (!reset) acc <= '0;
      else acc <= new_val;

    probe s (.a(|load), .q(selector), .*);

    for (genvar i = 0; i < WIDTH ; i++) begin
       hold load_h (.a(load_reg[i]), .en(selector),
            .q(new_val[i]), .*);
       discard sum_d (.a(sum[i]), .en(!selector),
            .q(new_val[i]), .*);
    end

    always @(posedge clk) begin
       load_reg <= load;
       sum <= acc + 1;
       out <= acc;
    end
endmodule
```

The RTL-like code is quite similar to a synchronous circuit description, it even counts with signals named `clk` and `reset`. The `reset` acts as expected, initialising the module registers, but the `clk` signal treatment is distinct. A Pulsar RTL-like description does use a clock signal (`clk`) to guide the synthesis process; it is employed in the pre-synthesis step, to select handshaking components and later to guide the *pseudo-synchronous* synthesis process [19]. However, this clock signal is not synthesised in the final circuit, nor it produces a clock distribution tree, as expected for an asynchronous circuit.

While executing instructions, the PC is usually incremented, but control flow instructions such as jumps and branches may load new values in it at any moment. The absence of a clock adds uncertainties to when the PC is updated, stressing the importance of choice components. The circuit uses a `probe` component to detect when a value is present at its load input channel. If no load value is present, the circuit proceeds with incrementing the current counter value; otherwise it discards the counter value and loads the received token. The token steering components, instantiated between lines 15-18, are in a fan-in steering arrangement; they multiplex the *new_val* channel from either *load_reg* or *sum*.

In Figure 3, it is possible to observe that the `discard` component takes part in the accumulator loop. Before the arrival of a token at the load channel, `probe` produces false-valued tokens interposed with spacers[4]. False-valued tokens prevent the `hold` component from executing a handshake on its input and output data channels. If a token arrives at the load channel after the `probe` has issued a false-valued token, it is held there until the `probe` can acknowledge its presence.

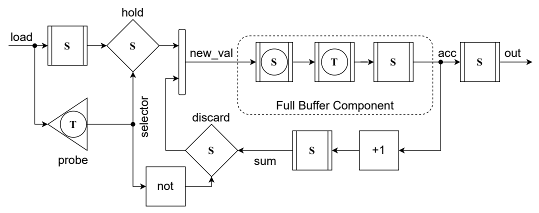


Fig. 3: Simplified token flow diagram for the ARV PC. Most depicted elements are handshake entities.

IV. Design Validation and Experiments

Delay-annotated simulations were performed to validate the ARV design implementation. The complete SystemVerilog input description of the organisation mentioned in Section III was synthesised using the Pulsar flow targeting the ASCEnD-FreePDK45 standard cell library. Synthesis of the RTL-like description addressed cycle-time constraints from 10ns to 3ns in 0.25ns steps, resulting in 29 distinct netlists. The synthesis process employed the *typical* PVT library corner, with typical transistors under 1.10 V supply and 25 °C. Post-synthesis gate delay annotations were extracted using Cadence Genus STA engine. Delays were extracted for three PVT corners: (i) the *worst* corner, with slow transistors under 0.95 V and 125 °C; (ii) the cited *nominal* corner; and (iii) the *best* corner, with fast transistors under 1.25 V and 0 °C.

The testbench simulates an ideal 0-delay environment; this enables measuring the maximum circuit performance independently. It provides a simulated memory model with pre-loaded programs and an IO peripheral to enable the software to write to the screen after terminating the simulation. It also verifies the test programs execute correctly and extracts some metrics.

[4]The synthesis produces dual-rail data signals operating under RTZ/RTO protocols. E.g. 32-bit signals `load` and `out` will each comprise 64 wires.

Program execution time in processors is directly impacted by the time required to execute individual instructions. A proxy to this measure is the time between instruction fetches, or inter-fetch time[5], the period of time elapsed between the start of two consecutive instruction fetches. Figure 4 resumes the simulation results. The Cadence XCelium tool simulated netlists from each synthesis in two forms: (i) post-mapping annotated delays (*_nwd* suffix in Figure 4); (ii) post-layout gate and wire annotated delays (*_wd* suffix in Figure 4). The proposed organisation was validated using the RISC-V Foundation `rv32ui` test virtual machine (TVM) for unit tests [20], a set of programs to test an RV32I implementation.

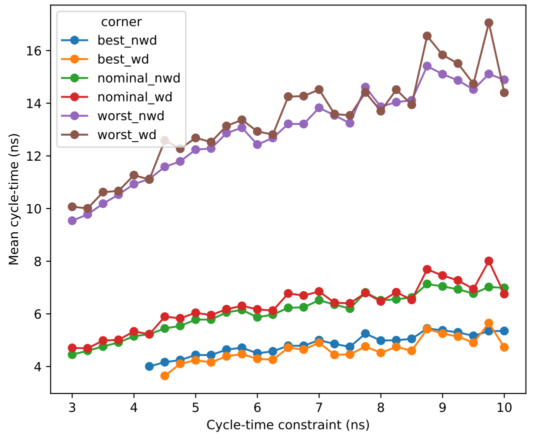


Fig. 4: Inter-fetch mean cycle-time for all corners and all cycle-time constraints for simulations of the `rv32ui` TVM.

It is worth noting that simulations showed correct behaviour across all tested scenarios except for the 5 tightest best corner cases. This problem derives from Genus performing logic duplication optimisations to reduce the gate fan-out and meet the very restrictive timing constraints, which produces *gate orphans* [9] in the circuit. An ST circuit with gate orphans is no longer QDI as the isochronic fork assumption may no longer be sufficient to guarantee its correct operation. Additional timing constraints are required to guarantee the correct operation at all corners. This issue is further evidenced when observing Figure 5. The affected cycle time constraints display a substantial gate area growth compatible with logical duplication optimisations.

Figure 6 depicts the distribution of time between instructions observed in simulations at the best corner for every cycle-time constraint. Ideally, the inter-fetch time is smaller or equal to the cycle-time constraint. The Figure shows a right-skewed distribution, indicating that 50% of the inter-fetch delays fall between the minimal and mean values. This suggests that the majority of instructions execute efficiently and quickly. On the other hand, values between the mean and the maximum values correspond to data hazards, which introduce bubbles and pipeline stalls, resulting in increased inter-fetch delays.

[5]The inter-fetch time is only a proxy to the actual instruction execution time as it does not account for cancellation in branches not taken.

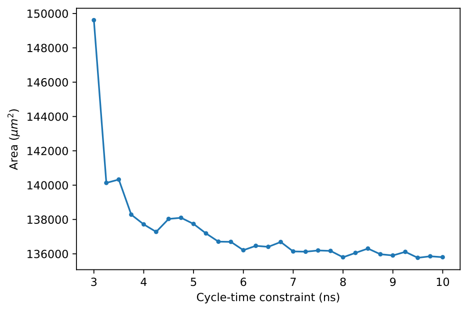


Fig. 5: Gate area results for all cycle-time constraints.

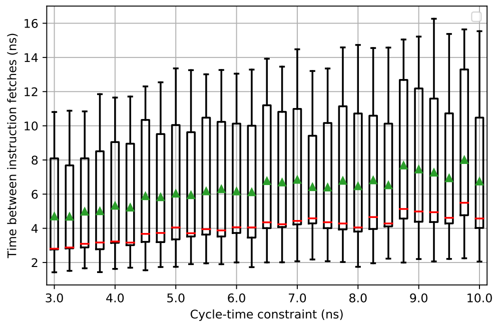


Fig. 6: Boxplot with the ARV inter-fetch time distribution, for the corner `nominal_wd`, given a range of constraints.

V. Conclusions, Ongoing and Future Work

This paper brings a promising approach to address digital design issues that rise as technology nodes advance toward semiconductor physical limits. By leveraging the benefits of asynchronous QDI design and the enhanced version of the Pulsar flow, complex circuits can be designed and implemented in an automated, semi-custom manner. The validation of the approach using a challenging RISC-V processor design highlights the Pulsar potential to address real-world applications. The RTL-like proposed model, enhanced with specific abstract components, demonstrated to be a sound and practical front-end for efficient asynchronous design capture.

The few timing problems mentioned in Section IV arose for the first time while using Pulsar to synthesise a large design such as the ARV processor description. Ongoing work includes a formalisation of the found timing issues (dealing with Genus-generated gate orphans) and proposing a tool to constrain circuit design further using the commercial synthesis tool Genus. Authors believe that one interesting approach available in the literature, *relative timing constraints* [21], might help Pulsar achieve fully robust timing closure.

Processors are a challenging target for asynchronous design, which served the purpose of this work to stress the Pulsar flow. Besides, QDI templates, while enabling the implementation of robust circuits, are known to impose significant area and power overheads when compared to their synchronous counterparts. Therefore, some directions are under investigation, including:

1) Investigate the trade-offs between a traditional synchronous RV32I processor and the processor resulting from this work, evaluating power, performance and area metrics under different operating conditions, including voltage scaling applications.
2) Propose more efficient ST templates - preliminary work on novel templates showed gains in the order of $3.5\times$ in area and performance compared to SDDS-NCL, with significant gains in power, too.
3) Address circuit classes where asynchronous often display advantages - this includes communication circuits such as network-on-chip routers and secure systems such as crypto cores, among others.

References

[1] M. A. Scarpato, "Digital Circuit Performance Estimation under PVT and Aging Effects," Ph.D. dissertation, UGA-Grenoble, 2017.

[2] S. Jain, L. Lin, and M. Alioto, *Adaptive Digital Circuits for Power-Performance Range beyond Wide Voltage Scaling*. Springer, 2020, pp. 115–144.

[3] M. L. L. Sartori, R. N. Wuerdig, M. T. Moreira, S. Bampi, and N. L. V. Calazans, "Leveraging QDI Robustness to Simplify the Design of IoT Circuits," in *ISCAS*, 2020, pp. 1–5.

[4] Z. Li, Y. Huang, L. Tian, R. Zhu, S. Xiao, and Z. Yu, "A Low-Power Asynchronous RISC-V Processor with Propagated Timing Constraints Method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 68, no. 9, pp. 3153–3157, 2021.

[5] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, "Click Elements: An Implementation Style for Data-Driven Compilation," in *ASYNC*, 2010, pp. 3–14.

[6] G. R. B. andM. Bhanu Prakash, M. Kumari, and C. V. D. A. K. andG.Sai, "Design of a 32 bit Asynchronous RISC-V Processor using Verilog," *Journal of Emerging Technologies and Innovative Research (JETIR)*, vol. 7, no. 3, pp. 1717–1722, 2020.

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

[8] M. M. Kim, "Null Convention Logic Circuits for Asynchronous Computer Architecture," Ph.D. dissertation, RMIT University - Melbourne, 2019.

[9] K. M. Fant, *Logically Determined Design: Clockless System Design with NULL Convention Logic*. Wiley, 2005. [Online]. Available: https://books.google.com.br/books?id=igVTAAAAMAAJ

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

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

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

[13] T. Verhoeff, "Delay-Insensitive Codes - An Overview," *Distributed Computing*, vol. 3, no. 1, pp. 1–8, 1988. [Online]. Available: https://link.springer.com/article/10.1007/BF01788562

[14] R. B. Reese, S. C. Smith, and M. A. Thornton, "Uncle - An RTL Approach to Asynchronous Design," in *ASYNC*, 2012, pp. 65–72.

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

[16] M. L. L. Sartori, M. T. Moreira, and N. L. V. Calazans, "A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow," in *ASYNC*, 2020, pp. 114–123.

[17] M. L. L. Sartori, W. A. Nunes, and N. L. V. Calazans, "Components to Support Choice in Self-Timed Asynchronous Design Flows," *Journal of Integrated Circuits and Systems*, vol. 18, no. 1, pp. 1–11, 2023.

[18] M. L. L. Sartori and N. L. V. Calazans, "Go Functional Model for a RISC-V Asynchronous Organisation - ARV," in *ICECS*, 2017, pp. 381–348.

[19] Y. Thonnart, E. Beigné, and P. Vivet, "A pseudo-synchronous Implementation Flow for WCHB QDI Asynchronous Circuits," in *ASYNC*, 2012, pp. 73–80.

[20] R.-V. Foundation, "Unit Tests for RISC-V Processors," 2015. [Online]. Available: https://github.com/riscv/riscv-tests

[21] H. Kim, P. A. Beerel, and K. Stevens, "Relative timing based verification of timed circuits and systems," in *ASYNC*, 2002, pp. 115–124.