PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

MARCOS LUIGGI LEMOS SARTORI

CURBING THE DESIGN COMPLEXITY OF ASYNCHRONOUS CIRCUITS

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

CURBING THE DESIGN COMPLEXITY OF ASYNCHRONOUS CIRCUITS

MARCOS LUIGGI LEMOS SARTORI

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Fernando Gehm Moraes

Porto Alegre
2023

Ficha Catalográfica

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

MARCOS LUIGGI LEMOS SARTORI

CURBING THE DESIGN COMPLEXITY OF ASYNCHRONOUS CIRCUITS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on 5th September, 2023.

COMMITTEE MEMBERS:

Milos Krstic (Universität Potsdam/IHP)

Renato Peres Ribas (PGMICRO/UFRGS)

Prof. César Augusto Missio Marcon (PPGCC/PUCRS)

Prof. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

To all humankind,

“The acquisition of wealth is no longer the driving force of our lives. We work to better ourselves and the rest of humanity.”
(Jean-Luc Picard)

AGRADECIMENTOS

REDUZINDO A COMPLEXIDADE DE PROJETO DE CIRCUITOS ASSINCRONOS

RESUMO

O projeto de circuitos auto-temporizados (ST) oferece robustez às variações de atraso, mas enfrenta desafios que exigem soluções inovadoras. Esta tese representa um aprimoramento abrangente do Pulsar, uma contribuição anterior voltada para o projeto de circuitos assíncronos com ferramentas EDA comerciais, e introduz dois modelos específicos como parte dessa extensão.

WInDS, um aprimoramento de indicação fraca para PS-SDDS-NCL, e ALHO, empregando portas não-histeréticas, são propostos como modelos dentro da estrutura do Pulsar. Esses modelos abordam desafios na implementação de circuitos ST, refletindo a motivação da tese para superar obstáculos no projeto de circuitos assíncronos, ao mesmo tempo em que ataca sobrecargas de área, potência e desempenho.

Estendendo ainda mais o Pulsar, a tese detalha aprimoramentos para suportar escolhas, oferecendo maior flexibilidade de design, e demonstra o uso do Pulsar para projetar um processador RISC-V funcional com os modelos introduzidos.

Contribuições teóricas adicionais incluem o desenvolvimento de um sistema de classificação para circuitos assíncronos, proposto como uma necessidade de diferenciar circuitos ST Quasi-Delay-Insensitive (QDI) e mais relaxados em termos de suposições de temporização. A tese também formaliza os requisitos para construir circuitos ST funcionais, fornecendo princípios fundamentais para design e implementação eficazes.

Coletivamente, essas contribuições fornecem uma abordagem estruturada para o projeto de circuitos assíncronos, com foco no design não-histerético do ALHO e na extensão do Pulsar. O trabalho estabelece uma base para a exploração, pesquisa e desenvolvimento contínuos no campo dos circuitos assíncronos, aproveitando as capacidades do Pulsar e oferecendo novos insights teóricos.

Palavras-Chave: Circuitos Assincronos, ST, QDI, EDA.

CURBING THE DESIGN COMPLEXITY OF ASYNCHRONOUS CIRCUITS

ABSTRACT

The design of Self-Timed (ST) circuits offers robustness to delay variations but faces challenges that require innovative solutions. This thesis represents a comprehensive enhancement to Pulsar, a previous contribution focused on designing asynchronous circuits with commercial EDA tools, and introduces two specific templates as part of this extension.

WInDS, a weakly-indicating enhancement to PS-SDDS-NCL, and ALHO, employing non-hysteretic gates, are proposed as templates within the Pulsar framework. These templates address challenges in ST circuit implementation, reflecting the thesis's motivation to overcome hurdles in designing asynchronous circuits whilst attacking area, power and performance overheads.

Further extending Pulsar, the thesis details enhancements to support choice, offering greater design flexibility, and demonstrates the use of Pulsar to design a functional RISC-V processor with the introduced templates.

Additional theoretical contributions include the development of a classification system for asynchronous circuits, proposed as a need to differentiate Quasi-Delay-Insensitive (QDI) and more relaxed ST circuits in terms of timing assumptions. The thesis also formalises the requirements to build functional ST circuits, providing foundational principles for effective design and implementation.

Collectively, these contributions provide a structured approach to asynchronous circuit design, with a focus on the non-hysteretic design of ALHO and the extension of Pulsar. The work lays a foundation for continued exploration, research, and development in the field of asynchronous circuits, building on the capabilities of Pulsar and offering new theoretical insights.

Keywords: Asynchronous Circuits, ST, QDI, EDA.

LIST OF ACRONYMS

ALHO – Asynchronous Limited Hysteresis Organisation

ASCEND – Asynchronous Standard Cells Enabling n-Designs

BD – Bundled-data

DI – Delay Insensitive

FBCN – Full-buffer Channel Network

HBCN – Half-buffer Channel Network

NCL – Null Convention Logic

QDI – Quasi Delay Insensitive

SDDS – Spatially Distributed Dual Spacer

ST – Self Timed

STA – Static Timing Analysis

WCHB – Weak Conditioning Half Buffer

WINDS – Weakly-indicating Dual Spacer

CONTENTS

1. INTRODUCTION AND MOTIVATION

Advances in semiconductor fabrication technologies allow higher integration, energy savings and better performance but impose design challenges. Some of the challenges faced in newer technologies are: (i) higher sensitivity to process variations; (ii) higher static power; and (iii) longer wire delays. Process variations result from imperfections in the fabrication process and manifest themselves as changes in the circuit's electrical properties. Static power is the power dissipated by the circuit when idle that is when no switching activity is taking place. Higher static power can be attributed to higher leakage current due, e.g. to thinner oxide layers between the transistor channel and the transistor gate. It is roughly proportional to the number of transistors in the die, each contributing to the overall leakage power. Larger wire delays are attributable to smaller wire cross sections and relatively longer wire lengths, which cause reduced wire current capacity, higher parasitic capacitance and (relatively) increased coupling effects with neighbour wires. Wire delays in recent technologies make it unfeasible to route global signals in large circuits without using buffers to "repeat" the signal during its propagation. This imposes challenges on synchronous circuits, as they employ a global "clock" signal to provide a discrete-time reference for synchronisation.

The clock signal distribution must present a controlled, ideally null *skew*, i.e. it must present sufficiently close (or at least predictable) delays from the clock source to all registers controlled by that clock signal. Meeting this clock distribution criterion requires a clock distribution network composed by buffers and other signal distribution components. This clock distribution network can have a high cost in area and power. The power consumption associated with the clock distribution can comprise a significant percentage of the overall power dissipated especially in complex circuits. It is not unusual for the clock distribution circuit to take 40% or more of the total integrated circuit (IC) power consumption [DMM04]. Also, as ideal clock distribution networks with no skew are challenging to achieve or even impossible to obtain, it becomes necessary to compensate for the skew. Due to timing uncertainties arising from process variation affecting the clock distribution delay predictability, it becomes important to introduce margins in the clock period. This, of course, impacts the overall circuit performance.

Modern designs mitigate these issues by dividing the circuit into clock domains and using synchronisers to transfer signals between clock domain boundaries; this approach only helps solve the problem locally. However, logic spread across a large area (e.g. interconnects) still suffers from clock distribution problems. Furthermore, using multiple clock domains can result in significant synchronisation overheads, as different clock domains have their specific clock distribution network, possibly operating at different phases. Traversing clock domains can become even more challenging when dealing with modern designs em-

ploying multiple operation frequencies and voltages. A possible solution is the overall elimination of global or semi-global clock signals. Digital circuits without any global or semi-global clock signals are known as *asynchronous circuits*.

1.1 Asynchronous Circuits as an Alternative

Synchronous designs assume that the value on the inputs of all its registers will only be sampled at the rising (and/or falling) edge of the clock signal. This assumption enables designers to define timing constraints for the maximum delay in logic paths, which must always be lower than the clock period. This allows ignoring gate and wire delays, as long as timing constraints are respected. In other words, combinational logic is allowed to switch as it computes data during, say, the interval between two consecutive rising clock edges. Still, it must be stable and correct before the clock edge. Having such a simple model for circuit design is possible only because the clock is a periodic signal, *i.e.* its edges only occur at specific and known points in time. Hence, in synchronous circuits, events will only occur at specific moments; time can thus be treated as a discrete variable.

However, in asynchronous circuits, there is no such thing as a single clock to signal data validity on the inputs of all registers. In these, events can happen at any moment, and time must be regarded as a continuous variable. Thus, asynchronous designers rely on local handshake protocols for communication and synchronisation and on different design templates to build circuits, each with its own specific assumptions about gate and wire delays [BOF10]. These templates can be classified into two main families: bundled-data (BD) [Sut89] and Self-Timed (ST) [Mar90]. The design of a BD circuit is similar to a synchronous one; the difference is that BD relies on carefully designed delay elements for matching the timing of logic paths and controlling registers rather than having a clock signal. Communication and synchronisation are accomplished through handshake protocols. ST, on the other hand, uses special data encoding schemes and protocols that allow data to carry their own validity information. This enables data receivers to compute the presence or absence of complete data at its inputs/outputs and renders possible the local exchange of information. Because of this characteristic, ST circuits can adapt more gracefully to wire and gate delay variations. This is especially true for a subset of ST circuits, named *quasi delay insensitive* (QDI), whose structure enables their correct operation with fewer timing assumptions, thus less sensibility to delay variability.

Common sources of delay variations are PVT variations and ageing, all faced in recent sub-micron technologies. For example, intra-die process variations cause the same gate in different parts of the circuit to present different switching delays, which can possibly lead to faulty circuit behaviour and/or lower production yield. This problem affects interconnect circuits more acutely, as they span over a large silicon area. ST circuits are thus excel-

lent candidates to tolerate the conditions imposed by intrinsic intra-die process variations. Circuits that use ST interconnects between synchronous components can take advantage of this aspect of ST circuits. Furthermore, the mixed-use of asynchronous circuits as interconnect for synchronous components solves some of the known synchronisation problems occurring in the frontier between clock domains.

1.2 Challenges to ST Adoption

The design of ST circuits often relies on specialised infrastructures, which can frequently hinder the adoption of ST circuit design. This infrastructure often includes (i) specific gate libraries, containing, e.g. C-Elements, NCL [FB96] gates or PCHB logic cells; (ii) specific synthesis tools; and (iii) specific design capture languages. ST circuits normally require gates with hysteretic behaviour to facilitate or enable handshake synchronisation; a hysteretic gate holds the output stable until specific criteria are met. These special gates are not usually available in conventional cell libraries designed for synchronous semi-custom ASIC flows, nor are they readily usable by commercial EDA tools. Specialised tools like Uncle [RST12] and Balsa [EB02] can be used to produce asynchronous circuits with these gates. However, these specialised tools do not integrate well with semi-custom ASIC flows; they also lack the power and flexibility provided by advanced commercial EDA tools.

Beyond the hysteretic gates' availability and compatibility with EDA tools, these specialised gates can be a hurdle themselves. They require additional logic to implement hysteresis; this makes them bulkier compared to non-hysteretic *conventional gates*. Their widespread use to implement random logic in templates like NCL or DIMS [Sin81] produces circuits with potentially huge area overheads; of course, this overhead potentially also impacts static power. Hysteretic gates also tend to be slower to switch as they often present a greater logic depth and higher capacitance. These drawbacks are aggravated on QDI circuits which, to reduce timing assumptions to only some wire forks, are *strong indicating*, meaning that every transition in every intermediary signal must be acknowledged by at least another transition. Strong indicating circuits often require additional circuitry compared to their weak indicating counterparts.

Another challenge in designing asynchronous circuits is satisfying non-functional requirements such as performance, power and area targets. These are often conflicting among themselves, and balancing them requires employing constraints to guide the synthesis effort. The circuit's throughput is often the primary performance metric; it dictates how fast the circuit can accept new input and/or produce results. On asynchronous circuits, the throughput is bounded by the circuit's maximum cycle time. However, on complex concurrent asynchronous systems, cycle time is not trivial to capture. Synchronous circuits typically rely on register transfer level (RTL) models, where the maximum throughput is limited by a clock

period. This not only makes design capture simpler but also eases the task of optimising a netlist, as every timing path has the same fixed maximum delay constraint, the clock period. Synchronous RTL models drove decades of development on commercial EDA tools, which provide solid means for designers to explore power, performance and area optimisation in modern technologies. These means are nonetheless very specific, and efforts to abandon the synchronous paradigm in exchange for more powerful design techniques can easily make commercial tools not applicable. Accordingly, the support for asynchronous design lags; as technologies get less predictable and wire-dominated, there is a particular need for new solutions that allow asynchronous circuit optimisation after technology mapping and during physical design.

The lack of mature tools also affects the timing closure of ST circuit designs. Albeit resilient to delay variation, ST circuits must observe some timing assumptions to operate correctly. This is true even for QDI circuits, which must observe the isochronic fork assumption [Mar90], i.e. the delay difference between end-points of a wire fork must be negligible compared to gate delays; this requirement can become too restrictive in modern technologies, dominated by wire delay. The isochronic fork requirement is eased by the concept of *orphan paths* [Fan05]; these are paths that are not acknowledged by an output transition under certain conditions. If an orphan path is slower than the fastest propagation of a spacer, malfunction may occur. Both orphans paths and the isochronic fork assumption are ways of addressing the same issue, which is required to achieve timing closure in ST designs. Due to the lack of proper tooling, the design timing closure process is often manually crafted by specialised designers. Also, ST circuits are often implemented in an overly conservative fashion, to isolate such timing issues. The availability of more sophisticated timing checks and design closure automated techniques can enable a less restricted synthesis process for ST templates, thus allowing further optimisation, which is the main motivation for this Thesis.

1.3 Thesis Contributions

Some of the perceived hurdles to ST adoption have already been addressed in previous works. For instance, SDDS-NCL [MBSC18] proposes a method to allow conventional EDA tools to correctly employ hysteretic gates in the synthesis of ST circuits. Later, in the context of his MSc dissertation [Sar19], the Author has extended SDDS-NCL and proposed Pulsar, an automated synthesis flow capable of synthesising ST circuits using Cadence Genus. Pulsar employs the Half-buffer Channel Network (HBCN) timing model, an original contribution by the Author, and the pseudo-synchronous WCHB model [TBV12] to leverage static timing analysis (STA) and timing-driven synthesis capabilities of commercial EDA to design ST circuits with bounded cycle time, guaranteeing a performance floor. Pulsar also offers a design capture method based on register transfer level (RTL) descriptions,

a paradigm familiar to most synchronous designers, here called RTL-like. Ultimately, at the end of the Author's MSc, Pulsar enabled the synthesis of performance-bound ST circuits from RTL-like descriptions.

This Thesis builds on and furthers Pulsar to explore and advance the capabilities of commercial EDA tools in the synthesis of ST circuits. The technique designed to enable the use of STA provides greater control over the timing of ST circuits. This ultimately allowed the exploration of more relaxed templates to overcome some of the issues arising from the widespread use of hysteretic gates and strong indication. This process leads to the proposal of two new templates:

- Weakly-Indicating Dual Spacer (WInDS)
- Asynchronous Limited Hysteresis Organisation (ALHO)

WInDS is an enhancement from SDDS-NCL, which replaces strongly-indicating combinational logic by a weakly-indicating version. ALHO is a more radical departure from SDDS-NCL, in the sense that it eliminates the use of hysteretic gates in its implementation of combination logic. These templates do require other timing assumptions besides the aforementioned isochronic fork assumption.

To correctly describe the new templates, this Thesis proposes a classification system for asynchronous circuits that distinguish templates based on their timing assumption and structure. This system proposes adopting the term ST as circuits that use DI encoding and completion detection to work correctly and QDI as a strict subset of ST circuits whose timing assumption is limited to wire forks. Within this context, this Thesis also proposes a formal framework to describe and analyse logic gates and netlists comprising them. This formal framework helps define the requirements for building functional ST circuits. When analysing previous works through this new lens, the Author has found issues in the synthesis technique for SDDS-NCL employed in Pulsar that result in circuits that violate the QDI criteria but still meet those for ST.

Furthermore, this Thesis also improved Pulsar to enable the construction of complex circuits. One of these improvements is the ability to implement circuits with choice. Originally Pulsar was limited to synthesise deterministic pipelines, i.e. circuits where all data must always pass through the same pipeline stages, always in the same order. "Choice" in this context is the ability to dynamically change data progression in a pipeline. This enables the efficient construction of dynamic pipelines, i.e. pipelines where data can be redirected through different paths on demand. The implementation of choice in Pulsar enabled using it to synthesise a RISC-V processor core. This RISC-V implementation served as a challenging benchmark to evaluate the templates proposed in the Thesis.

To summarise, the contributions of this Thesis are:

1. The proposal of two new templates asynchronous templates:

   (a) WInDS
   (b) ALHO

2. The introduction of a formal framework to analyse logic gates and netlists comprising gates.
3. A classification system for asynchronous circuits.
4. The formalisation of the requirements to build functional ST circuits.
5. A critique of the SDDS-NCL template.
6. Enhancements on Pulsar to support choice.
7. The use of Pulsar to design a functional RISC-V processor in the three templates.

Contribution 1 is considered the main contribution of this Thesis.

1.4 Structure of the Thesis Text

This Thesis' division into Chapters attempts to group related topics; the original contributions are not necessarily presented in the order they were introduced in this Chapter. The Thesis does not include a dedicated chapter about the related state of the art. Instead, comparison to relevant related works appear within the context of the discussed topics.

Chapter 2 introduces some formal definitions required to understand the Thesis; Contribution 2 is covered in this Chapter. Chapter 3 elaborates on the design of asynchronous circuits; this Chapter covers Contributions 3 to 5. Chapter 4 covers the HBCN timing model and some of its potential applications. Chapter 5 is the central Chapter of the Thesis; it provides a comprehensive portrayal of the current state of Pulsar and approaches the new templates. Contributions 1 and 6 are covered in this Chapter. Finally, Chapter 6 presents a set of experiments and evaluates the proposed templates; this final Chapter covers Contribution 7.

2. DEFINITIONS

This Chapter provides formal definitions that are used in this work. Section 2.1 formally defines Boolean expressions and functions, providing a formal framework to define gates and their properties in Section 2.2. In the end, Section 2.3 provides a formal definition of graphs and Petri nets employed in the dynamic analysis of asynchronous circuits.

2.1 Boolean Expression and Functions

This section provides a precise definition of Boolean expressions and some related applicable definitions; these are thoroughly used in this thesis. The definitions here aim at being comprehensive but not exhaustive. A Boolean expression draws variable names from an arbitrary set V.

Definition 1 (Boolean Expression)**.** *E_V is the set of* **Boolean expressions** *over the set of variables V, defined recursively:*

$$E_V = \{e + f : e \in E_V, f \in F_V\} \cup F_V \tag{2.1}$$

$$F_V = \{f \cdot t : f \in F_V, t \in T_V\} \cup T_V \tag{2.2}$$

$$T_V = \{!t : t \in T_V\} \cup \{(e) : e \in E_V\} \cup \{0, 1\} \cup V \tag{2.3}$$

Where $\{\cdot, +, (,), !, 0, 1\}$ is the set of reserved symbols, F_V is the constructor set for factors, T_V is the constructor set for terms, and V is an arbitrary variable set.

The use of multiple sets of constructor forces operator precedence and associativity. It is helpful to interpret each constructor set separately:

- 2.1 defines the left-associative OR (+) operator constructor; in its absence, it accepts any of the constructors defined in F_V.
- 2.2 defines the left-associative AND ($\cdot$) operator constructor; in its absence, it accepts any of the constructors defined in T_V.
- 2.3 defines the highest precedence operator and leaf constructors; it defines the constant true (1), false (0), and variable leaf constructors, the negation (!) and the parenthesis operator constructors. The parenthesis subverts the operator precedence.

An example of a Boolean expression over the variable set $V = \{A, B, C, D\}$ is $A + B \cdot (C + !D)$.

From the definition of a Boolean expression, it is helpful defining a function to count references to a variable in an expression.

Definition 2 (Reference Count Function)**.** *The function* $v \Subset e : V \times E_V \mapsto \mathbb{N}$ **counts references** *of* v *in* e*. It is recursively defined as:*

$$v \Subset (e) = v \Subset e \tag{2.4}$$

$$v \Subset !e = v \Subset e \tag{2.5}$$

$$v \Subset e + e' = v \Subset e + v \Subset e' \tag{2.6}$$

$$v \Subset e \cdot e' = v \Subset e + v \Subset e' \tag{2.7}$$

$$v \Subset 0 = 0 \tag{2.8}$$

$$v \Subset 1 = 0 \tag{2.9}$$

$$v \Subset v' = \begin{cases} 1, & \textbf{if } v' \in V, v = v' \\ 0, & \textbf{otherwise} \end{cases} \tag{2.10}$$

As an example, consider the expression $x = A \cdot B + !A \cdot B$, it references the variable A two times and B one time; therefore $A \Subset x = 2$ and $B \Subset x = 1$

Definition 1 describes the structure of Boolean expressions, which alone are not sufficient. Their semantics arise when the Boolean expressions are *evaluated* with respect to the variable values.

Definition 3 (State Function)**.** *The* **state function** *s for Boolean variables V is defined as:*

$$s : V \mapsto \{0, 1\} \tag{2.11}$$

Definition 4 (Evaluation Function)**.** *The* **evaluation function** $eval_s(e) : s \times E_V \mapsto \{0, 1\}$ *exists if and only if:*

$$\forall v \in V : v \Subset e \geq 1 \implies s(v) \in \{0, 1\} \tag{2.12}$$

and it is defined recursively as:

$$eval_s(v \in V) = s(v) \tag{2.13}$$

$$eval_s((e)) = eval_s(e) \tag{2.14}$$

$$eval_s(1) = 1 \tag{2.15}$$

$$eval_s(0) = 0 \tag{2.16}$$

$$eval_s(!e) = \begin{cases} 1, \textbf{ if } eval_s(e) = 0 \\ 0, \textbf{ if } eval_s(e) = 1 \end{cases} \tag{2.17}$$

$$eval_s(a \cdot b) = \begin{cases} 1, \textbf{ if } eval_s(a) = eval_s(b) = 1 \\ 0, \textbf{ otherwise} \end{cases} \tag{2.18}$$

$$eval_s(a + b) = \begin{cases} 1, \textbf{ if } eval_s(a) = 1 \textbf{ or } eval_s(b) = 1 \\ 0, \textbf{ otherwise} \end{cases} \tag{2.19}$$

The Boolean expression provides a formal method to express Boolean functions with named variables. This is useful when working with switching networks. Nonetheless, every Boolean expression is equivalent to a Boolean function.

Definition 5 (Boolean Function)**.** *An n-input* **Boolean function** $f(x)$ *is a function from a combination of binary values to a binary value:*

$$f : \{0,1\}^n \rightarrow \{0,1\} \tag{2.20}$$

Definition 6 (Boolean Expression and Function Equivalency)**.** *A Boolean expression* $e \in E_V$ *is equivalent to a Boolean function* $f(x_1 \ldots x_n)$*, if:*

$$\forall x_i, \exists s(v_i) = x_i, eval_s(e) = f(x_1 \ldots x_n) \tag{2.21}$$

This fact is exploited in the next section to formally define some properties for both Boolean expressions and functions.

2.1.1 Properties of Boolean expressions and functions

This section defines a comprehensible, but not exhaustive, set of properties of Boolean expressions and functions. These properties are used extensively in this thesis.

First, it is helpful to define *distributive normal form* (DNF) expressions as a more restrictive set of Boolean expressions. They comprise Boolean expressions without parenthesis or constants where only variables can be negated.

Definition 7 (Disjunctive Normal Form)**.** *D_V is the set of* **disjunctive normal form (DNF)** *expressions on variables V, defined recursively:*

$$D_V = \{d + c : d \in D_V, c \in C_V\} \cup C_V \tag{2.22}$$

$$C_V = \{c \cdot n : c \in C_V, n \in N_V\} \cup N_V \tag{2.23}$$

$$N_V = \{!v : v \in V\} \cup V \tag{2.24}$$

Where C_V is the set of clauses and N_V is the set of strict terms without constants or parenthesis. A Boolean expression $e \in E_V$ is said to be **in DNF** *if and only if $e \in D_V$; e is said to be a* **clause** *if $e \in C_V$.*

As an example, the Boolean expression $A + B \cdot (C + !D)$ is not in DNF, but the equivalent expression $A + B \cdot C + B \cdot !D$ is in DNF.

An expression in DNF can be decomposed into a set of clauses using the clause decomposition operator.

Definition 8 (Clause Decomposition)**.** *Let $e \in D_V$ be a Boolean expression in DNF, for which the decomposition operator $\eth(e)$ is recursively defined:*

$$\eth(c) = \{c\}, \text{ if } c \in C_V \tag{2.25}$$

$$\eth(cs + c) = \{c\} \cup \eth(cs) \tag{2.26}$$

As an example of decomposition, regard $\eth(A + B \cdot C + B \cdot !D) = \{A, B \cdot C, B \cdot !D\}$.

An important concept when constructing glitch-less asynchronous circuits is unateness, formally defined as:

Definition 9 (Boolean Function Unateness)**.** *A Boolean function $f(x_1, x_2, ..., x_n)$ is said to be* **positive unate** *in x_i ($1 \leq i \leq n$), if:*

$$\forall x_j, i \neq j, f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) \geq f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n) \tag{2.27}$$

Similarly, f is called **negative unate** *in x_i if:*

$$\forall x_j, i \neq j, f(x_1, \dots, x_{i-1}, 0, \dots, x_n) \geq f(x_1, \dots, x_{i-1}, 1, x_{i+1}, ..., x_n) \tag{2.28}$$

If a function is neither positive unate nor negative unate in x_i, it is said to be **binate** *in x_i. Moreover, if a function is positive or negative unate in all its variables, it is called a* **unate function**. *An unate function that is positive (negative) unate in all its variables is called a* **positive (negative) unate function**.

Unateness is similarly defined for Boolean expressions:

Definition 10 (Boolean Expression Unateness)**.** *A Boolean expression $e \in E_V$ is* **positive unate** *in $x \in V$, if:*

$$\forall(s, s') : s(x) = 1, s'(x) = 0 \implies eval_s(e) \geq eval_{s'}(e) \tag{2.29}$$

Similarly, e is **negative unate** *in x, if:*

$$\forall(s, s') : s(x) = 0, s'(x) = 1 \implies eval_s(e) \geq eval_{s'}(e) \tag{2.30}$$

If an expression is neither positive unate nor negative unate in x, it is said to be **binate** *in x. Furthermore, if an expression is positive or negative unate in all its variables, it is called an* **unate expression***. An unate expression that is positive (negative) unate in all its variables is called a* **positive (negative) unate expression***. If an expression is binate in all its variables, it is called a* **binate expression***.*

An example of a positive unate expression is $A \cdot B$, of a negative unate expression is $!A+!B$, and of a binate expression is $A \cdot !B+!A \cdot B$ (aka, A xor B).

Also, it is good to formalise the concept of complements for both Boolean functions and expressions.

Definition 11 (Boolean Function Complement)**.** *Let f and g be Boolean functions. f* **complements** *g if:*

$$\forall x : f(x) \neq g(x) \tag{2.31}$$

Definition 12 (Boolean Expression Complement)**.** *Let e and e' be Boolean expressions. e* **complements** *e' if:*

$$\forall s : eval_s(e) \neq eval_s(e') \tag{2.32}$$

Finally, relevant to the definition of logical gates in this thesis is the definitions of orthogonality:

Definition 13 (Orthogonal Boolean Function)**.** *Two Boolean functions f and g are* **orthogonal** *if:*

$$\forall x : f(x) = g(x) \implies f(x) = g(x) \neq 1 \tag{2.33}$$

By extension, a set of Boolean functions F is orthogonal if:

$$\forall f, g \in F : f \neq g \implies f \text{ and } g \text{ are orthogonal} \tag{2.34}$$

The two orthogonal Boolean functions can never be true for the same input. Orthogonality is similarly defined for Boolean expressions:

Definition 14 (Orthogonal Boolean Expression)**.** *Two Boolean expressions e and e′ are* **orthogonal** *if:*

$$\forall s : eval_s(e) = eval_s(e') \implies eval_s(e) = eval_s(e') \neq 1 \tag{2.35}$$

By extension, a set of Boolean expressions $Œ_V$ is orthogonal if:

$$\forall e, e' \in Œ_V : e \neq e' \implies e \text{ and } e' \text{ are orthogonal} \tag{2.36}$$

For the same state, the two orthogonal expressions should never evaluate both to true, e.g. $A \cdot B$ and $!A \cdot !B$ are orthogonal, but $A \cdot B$ and $A + B$ are not.

2.2 Gates, Netlists and Production Rules

This Section provides the formal definitions of gates and netlists used in this thesis. They are loosely based on the analytical framework introduced by Martin [Mar90] and extended by Keller et al. [KKM09]. However, they define a framework capable of analysing the correct operation of asynchronous circuits at a transistor level, whilst this thesis is concerned only with the correct operation of asynchronous circuits at a discreet logical gate level. These gates are assumed to be adequately verified and sound, with any internal fork not affecting the overall correctness of the circuit. This layered “divide-and-conquer” approach enables scaling the analysis to more extensive circuits.

Firstly, the basic definition for constructing the analytical framework in this section is the production rule:

Definition 15 (Production Rule)**.** *A* **production rule** *(PR) is a triple:*

$$(g, v, d) : D_V \times V \times \{\uparrow, \downarrow\} \tag{2.37}$$

Where $g \in D_V$ is a guard expression in DNF, $v \in V$ is the target variable, and d is the direction of change. A production rule is typically denoted $g \rightarrowtail vd$.

A PR describes a sufficient condition to pull a circuit node up or down. Each variable $v \in V$ represents a node in the circuit. A state function captures the state of the circuit at any given moment. When the guard expression g evaluates to 1, the variable v is set to be pulled up or down according to the direction d, possibly changing a future state function. The basic intuition for the PR $g \rightarrowtail v\uparrow$ is that the condition g is sufficient to enable the pull-up

network driving the circuit node v. It is possible to describe a netlist comprising wires and logic gates using PRs.

Definition 16 (Netlist)**.** *A* **netlist** *is a quadruple*

$$\mathcal{N} = \langle Vp, Vg, Vw, \mathcal{P} \rangle \tag{2.38}$$

Where $Vp \subset V$ is the set of port variables, $Vg \subset V$ is the set of gate output variables, $Vw \subset V$ is the set of wire termination variables, and $\mathcal{P}$ is a finite set of production rules.

Definition 17 (Wire)**.** *A* **wire** *is a PR pair:*

$$\langle x \rightarrowtail y \uparrow,\ !x \rightarrowtail y \downarrow \rangle \subset \mathcal{P} \tag{2.39}$$

Which satisfies the predicates:

$$x \in Vg \cup Vp \tag{2.40}$$
$$y \in Vw \cup Vp \tag{2.41}$$

Where x is the wire origin variable, and y is the wire termination variable.

Definition 18 (Logic Gate)**.** *A* **logic gate** *is a PR pair:*

$$\langle g_{\uparrow} \rightarrowtail y \uparrow,\ g_{\downarrow} \rightarrowtail y \downarrow \rangle \subset \mathcal{P} \tag{2.42}$$

Which satisfies the predicates:

$$g_{\uparrow} \text{ and } g_{\downarrow} \text{ are orthogonal} \tag{2.43}$$
$$g_{\uparrow}, g_{\downarrow} \in E_{Vw} \tag{2.44}$$
$$y \in Vg \tag{2.45}$$

Where y is the gate output, $g_{\uparrow}$ is the gate activation expression, and $g_{\downarrow}$ is the gate deactivation expression. Collectively, the activation and deactivation expressions are called the guard expressions.

The intuition for Definition 18 is that one PR models the pull-up network, and the other models the pull-down network of a logic gate. Similarly, Definition 17 models wires as “non-inverted” buffers. Also, notice that the logic gate definition limits the guard expressions to be orthogonal. Since it is assumed that logic gates are sound by design, short-circuiting conditions are assumed not to occur, or at least not to affect the output value. The orthogonality of the guard expressions guarantees that short-circuiting never occurs, thus simplifying the model.

The definitions presented in this Section are structured to disallow gate-to-gate connections. Predicate 2.41 dictates that wires can only drive wire termination or port variables. Similarly, Predicate 2.45 dictates that a gate can only drive gate output variables. Predicates 2.40 and 2.44 complement the aforementioned predicates; the former dictates that wires can only be driven by gates' outputs or ports, and the latter dictates that gates' inputs can only be connected to wires. However, limiting gate-to-gate connections is not sufficient to avoid implicit forking. for that, any wire can only be referenced by a single gate. This is important later when using this construction to analyse timing issues. Furthermore, no circuit node should be driven by more than one single gate or wire. Here, additional predicates are defined to enforce the netlist correctness.

Definition 19 (Netlist Corectness)**.** *A netlist* $\mathcal{N} = \langle Vg, Vw, \mathcal{P} \rangle$ *is* **correct***, if:*

$$\forall v \in Vg \; \exists! \; \{g \rightarrowtail yd, g' \rightarrowtail y'd'\} \subset \mathcal{P} : y = y' = v, d \neq d' \tag{2.46}$$

$$\forall v \in Vw \; \exists! \; \{g \rightarrowtail yd, g' \rightarrowtail y'd'\} \subset \mathcal{P} : y = y' = v, d \neq d' \tag{2.47}$$

$$\forall v \in Vw \; \exists! \; \{g \rightarrowtail y\uparrow, g' \rightarrowtail y\downarrow\} \subset \mathcal{P} : (v \Subset g) + (v \Subset g') \geq 1 \tag{2.48}$$

Predicate 2.46 and 2.47 state that each wire termination and gate output variables should be driven by a single PR pair with opposing directions, a gate or a wire respectively. Whereas, Predicate 2.48 states that every wire termination variable should only be referenced by the guards in single production rule pair, i.e. referenced by a single gate. Notice that these restrictions also limit the netlist to be fully connected.

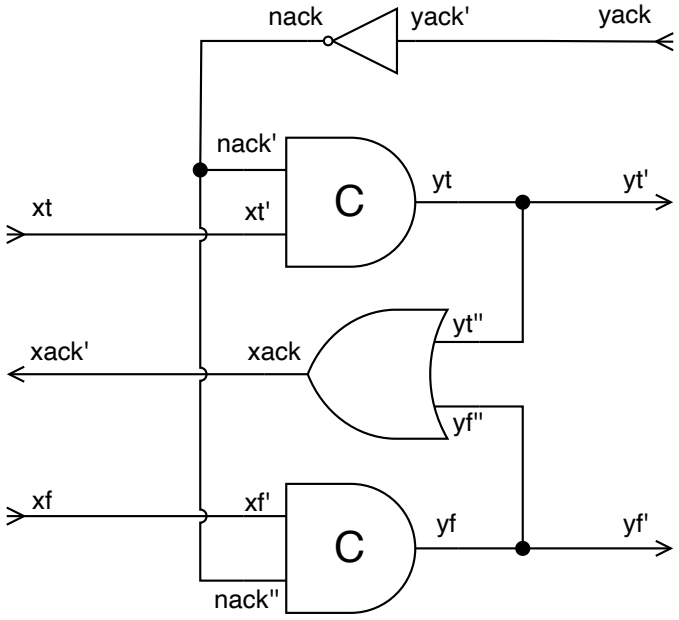


Figure 2.1: WCHB register with explicit forks. The variable names are marked on the corresponding circuit nodes.

As an example, consider the WCHB register depicted in Figure 2.1. This is a netlist with explicit wire forks. Every gate output and every wire termination is marked with a vari-

able name. The forks are explicit, as each wire termination has its own variable name. The variable sets for this netlist are:

$$Vp = \{xt, xack', xf, yack, yt', yf'\} \tag{2.49}$$
$$Vg = \{nack, yt, xack, yf\} \tag{2.50}$$
$$Vw = \{nack', xt', xf', nack'', yack', yt', yt'', yf'', yf'\} \tag{2.51}$$

Its wires are defined by PR pairs:

$$\langle xt \rightarrowtail xt' \uparrow,\ !xt \rightarrowtail xt' \downarrow\rangle \tag{2.52}$$
$$\langle xf \rightarrowtail xf' \uparrow,\ !xf \rightarrowtail xf' \downarrow\rangle \tag{2.53}$$
$$\langle xack \rightarrowtail xack' \uparrow,\ !xack \rightarrowtail xack' \downarrow\rangle \tag{2.54}$$
$$\langle yt \rightarrowtail yt' \uparrow,\ !yt \rightarrowtail yt' \downarrow\rangle \tag{2.55}$$
$$\langle yt \rightarrowtail yt'' \uparrow,\ !yt \rightarrowtail yt'' \downarrow\rangle \tag{2.56}$$
$$\langle yf \rightarrowtail yf' \uparrow,\ !yf \rightarrowtail yf' \downarrow\rangle \tag{2.57}$$
$$\langle yf \rightarrowtail yf'' \uparrow,\ !yf \rightarrowtail yf'' \downarrow\rangle \tag{2.58}$$
$$\langle yack \rightarrowtail yack' \uparrow,\ !yack \rightarrowtail yack' \downarrow\rangle \tag{2.59}$$
$$\langle nack \rightarrowtail nack' \uparrow,\ !nack \rightarrowtail nack' \downarrow\rangle \tag{2.60}$$
$$\langle nack \rightarrowtail nack'' \uparrow,\ !nack \rightarrowtail nack'' \downarrow\rangle \tag{2.61}$$

And its gates are defined by the PR pairs:

$$\langle yt'' + yf'' \rightarrowtail xack \uparrow,\ !yt''\cdot!yf'' \rightarrowtail xack \downarrow\rangle \tag{2.62}$$
$$\langle !yack \rightarrowtail nack \uparrow,\ yack \rightarrowtail nack \downarrow\rangle \tag{2.63}$$
$$\langle nack' \cdot xt' \rightarrowtail yt \uparrow,\ !nack'\cdot!xt' \rightarrowtail yt \downarrow\rangle \tag{2.64}$$
$$\langle nack'' \cdot xf' \rightarrowtail yf \uparrow,\ !nack''\cdot!xf' \rightarrowtail yf \downarrow\rangle \tag{2.65}$$

2.2.1 Logic Gates Properties

To further analyse a logic gate, consider the OR-gate driving *xack* defined by PR Pair 2.62. When $s(yt'') = s(yf'') = 0$, the activation expression for *xack* evaluates to 1, setting to pull *xack* down; but for every other input state, the deactivation expression for *xack* evaluates to 1, setting to pull *xack* up. This occurs because the guard expressions for the OR-gate complement each other.

It is possible to have logic gates that, for some states, neither guard expression evaluates to 1. These are called *hysteretic gates*; they are logic gates whose guard expressions don't complement each other.

Definition 20 (Gate Hysteresis)**.** *A gate* $\langle g_\uparrow \rightarrowtail y\uparrow,\ g_\downarrow \rightarrowtail y\downarrow\rangle$ *is said to be* **hysteretic** *if*

$$\exists s : eval_s(g_\uparrow) = eval_s(g_\downarrow) = 0 \tag{2.66}$$

Otherwise, it is said to be **conventional***.*

For simplicity, if no change is set to occur on the output variable of a gate, it holds its value indefinitely. This behaviour, in fact, is often implemented using an internal feedback mechanism in the logic gate. The full behaviour of any gate can be expressed by a single characteristic function that references a delayed output value:

Definition 21. *Let* $\langle g_\uparrow \rightarrowtail q\uparrow,\ g_\downarrow \rightarrowtail q\downarrow\rangle$ *be a gate with output q,* q_i *is its current value and* q_{i-1} *is its previous value. The gate* **characteristic function** *is defined by the expression:*

$$q_i = g_\uparrow + q_{i-1}\cdot !(g_\downarrow) \tag{2.67}$$

As an example, consider the C-element defined by PR Pair 2.64. When $s(nack') = s(xt') = 1$, the activation expressions for *yt* evaluates to 1, setting to pull *yt* up; similarly, when $s(nack') = s(xt') = 0$, the deactivation expression for *yt* evaluates to 1, setting to pull *yt* down. However, when $s(nack') \neq s(xt')$, neither guard expression evaluates to 1, then no change on *yt* is set to occur. Compare that to the OR-gate example where the two guard expressions complement each other; when one of the guards evaluates to 0, the other evaluates to 1. The OR-gate is an example of a *conventional gate*, whereas the C-Element is an example of a *hysteretic gate*

Logic gates can be further classified by their unateness, i.e. how the direction of change in an input variable affects the direction of change in the output variable.

Definition 22 (Gate Unateness)**.** *A logic gate* $\langle g_\uparrow \rightarrowtail y\uparrow,\ g_\downarrow \rightarrowtail y\downarrow\rangle$ *is said to be* **positive unate** *if:*

$$g_\uparrow \text{ is positive unate and } g_\downarrow \text{ is negative unate} \tag{2.68}$$

Similarly, a logic gate is said to be **negative unate** *if:*

$$g_\uparrow \text{ is negative unate and } g_\downarrow \text{ is positive unate} \tag{2.69}$$

If a logic gate is neither positive nor negative unate, it is said to be **non-unate***.*

The OR-gate and C-Element are examples of positive unate gates, and as an example of a negative unate gate, consider the inverter defined by PR Pair 2.63. Compare the guard expressions in the inverter with the other gates. Notice that when $s(yack) = 0$, the activation expression for *nack* evaluates to 1, setting to pull *nack* up; conversely, when $s(yack) = 1$, the deactivation expression for *nack* evaluates to 1, setting to pull *nack* down.

This behaviour, where the output is set to rise when the inputs are at a logic low, is the opposite of what happens with the OR-gate or the C-element; this characterises the inverter as a negative unate gate and the C-element as a positive unate gate.

2.3 Graphs and Petri Nets

The definitions related to timing modelling used in this work rely on or derive from the fundamental concept of *graphs*, or more specifically on *directed graphs*. Accordingly, a precise definition of this concept is provided here, based on classical definitions such as the one provided by Cormen et al. [CLRS09].

Definition 23 (Directed Graph)**.** *A* **directed graph (or digraph)** *G is a pair $G = (V, E)$, where V is a finite set and E is a binary relation on V. The set V is called the* **vertex set** *of G, and its elements are called* **vertices** *(singular:* **vertex***). The set E is called the* **edge set** *of G, and its elements are called* **edges***.*

Given a vertex $v \in V$ of a graph $G = (V, E)$, the subset of V with the form $\{w | w \in V \wedge (w, v) \in E\}$ is called the **preset** *of vertex v. Accordingly, given a vertex $v \in V$ of a graph $G = (V, E)$, the subset of V with the form $\{w | w \in V \wedge (v, w) \in E\}$ is called the* **postset** *of vertex v.*

Unless otherwise noted, in this work all references to graphs refer to directed graphs and the word *directed* is omitted. Note that the previous definition includes describing the predecessor and successor vertex sets in graphs, a concept very important for more elaborate structures used herein. Graphs are generic structures that can be specialised to address more specific modelling needs. One such specialisation relevant here is that of *bipartite graphs*.

Definition 24 (Bipartite Graph)**.** *A* **bipartite graph** *is a directed graph $G = (V, E)$ where the set V is in the union of two sets, $V = W \cup X$ and where E is formed by edges having exactly one element from W and one element from X, i.e. $E \subseteq \{(a, b) | ((a \in W) \wedge (b \in X)) \vee ((a \in X) \wedge (b \in W))\}$.*

2.3.1 Petri Nets

Often, the modelling of asynchronous circuits relies on Petri nets, whose static structure can be captured by graphs. The next definitions formalise the general concept of Petri nets and particularise it to more specific forms useful in asynchronous circuit modelling.

Note that a Petri net has a static structure, an initial marking and a marking evolution behaviour, the two later ones encompassing the dynamics of the net. The definition covers all parts of the concept and is based on [Mur89].

Definition 25 (Petri Net (PN))**.** *A* **Petri net** *is a* 5*-tuple* $PN = (P, T, F, W, M_0)$ *where:* $P = \{p_1, p_2, ..., p_m\}$ *is a finite set of* **places***,* $T = \{t_1, t_2, ..., t_n\}$ *is a finite set of* **transitions***,* $F \subseteq (P \times T) \cup (T \times P)$ *is a set of* **arcs***, collectively called the* **Petri net flow relation***,* $W : F \longrightarrow \mathbb{N}^*$ *is the* **weight** *function, and* $M_0 : P \to \mathbb{N}$ *is the* **initial marking***, with* $P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$*. The* **Petri net structure** *is the 4-tuple* $N = (P, T, F, W)$ *with no consideration of marking. A Petri net with a given initial marking can be alternatively denoted by* (N, M_0)*.*

The **behaviour** *of a Petri net relies on a set of rules that dictate how a* **marking** *or* **state** *evolves into another state, according to the following set of* **firing rules***:*

1. *A transition* t *is said to be enabled if each input place* p *of* t *is marked with at least* $w(p, t)$ *tokens, where* $w(p, t)$ *is the weight of the arc from* p *to* t*;*
2. *An enabled transition may or may not fire, depending on whether or not the event actually takes place;*
3. *A firing of an enabled transition* t *removes* $w(p, t)$ *tokens from each input place* p *of* t*, and adds* $w(t, p)$ *tokens to each output place* p *of* t*, where* $w(t, p)$ *is the weight of the arc from* t *to* p*.*

A transition without any input place is called a **source transition***, and one without any output place is called a* **sink transition***. Note that a source transition is unconditionally enabled and that the firing of a sink transition consumes tokens but does not produce any.*

A pair of a place p *and a transition* t *is called a* **self-loop** *if* p *is both an input and output place of* t*. A Petri net is said to be* **pure** *if it has no self-loops. A Petri net is said to be* **ordinary** *if all of its arc weights are* 1*.*

It should be clear from the PN definition and from Definition 24 that the structure N of a PN can be represented by a bipartite graph where the vertex set V of the graph is the union of the set of places and of the set of transitions of the Petri net, i.e. $V = P \cup T$. Because of this, it is common and practical to informally state that PNs *are* bipartite graphs, ignoring the underlying marking and behaviour concepts. A big advantage of treating a PN as a graph is inheriting to PNs all graph concepts, e.g. vertex degrees, vertices (places or transitions) presets and postsets, etc. Where precision is not compromised, this document adopts this little abuse.

2.3.2 Petri Net Properties

An extensive set of behavioural properties derives from the definition of a PN; Techniques to analyse PN instances for such properties abound in the literature. This Section explores PN properties specifically relevant to this work. The interested reader can refer to [Mur89] or to PN books such as [Rei13] for a more complete discussion of PN properties.

According to Murata [Mur89], there are two types of PN properties: those that depend on the initial marking M_0, called **behavioural properties**, and those independent of M_0, called **structural properties**. This works addresses only some of the behavioural properties.

The first important property is **reachability**. This is a fundamental property to study the dynamic properties of any system described by PNs. Reachability relies on the PN´s *firing rules* and on the initial marking M_0 of a PN. A marking M_n is said to be reachable from marking M_0 if there exists a sequence of firings that transforms M_0 into M_n. A firing or occurrence sequence is denoted by $\sigma = M_0 t_1 M_1 t_1 M_2 ... t_n M_n$, or simply $\sigma = t_1 t_1 t_2 ... t_n$. In this case, M_n is reachable from M_0 by σ and we write $M_0[\sigma > M_n$. The set of all possible markings reachable from M_0 in a PN (N, M_0) is denoted by $R(N, M_0)$, or simply $R(M_0)$. The set of all possible firing sequences from M_0 in a PN (N, M_0) is denoted by $L(N, M_0)$, or simply $L(M_0)$. The **reachability problem** for PNs is the problem of finding, for a given marking M_n, if $M_n \in R(M_0)$ in a PN (N, M_0). Sometimes it is interesting to define the **submarking reachability problem**, where instead of a PN marking M_n attentions is restricted to M'_n, a marking is limited to just some subset of places of P.

A second property worth defining here is **boundedness**, related to the maximum amount of tokens a place of some PN holds. A PN (N, M_0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e. $M(p) \leq k$ for every place p and every marking $M \in R(Mo)$. A PN (N, M_0) is said to be **safe** if it is 1-bounded. In (asynchronous) hardware design, places of a PN are often used to represent buffers and registers for storing intermediate data. By verifying that the net is bounded or safe, it is guaranteed that there will be no overflows in buffers or registers, no matter what firing sequence is taken.

The last property needed to define in this work is **liveness**, associated with concepts in system design like the absence of deadlocks. A PN (N, M_0) is said to be **live** (or equivalently M_0 is said to be a live marking for N) if no matter what marking has been reached from M_0, it is possible to ultimately fire any transition of the PN by progressing through some firing sequence. This means that a live PN guarantees deadlock-free operation, no matter what firing sequence is chosen.

2.3.3 Marked Graphs

Marked graphs constitute a limited class of PNs that allows modelling concurrency but not choice (to avoid, e.g. non-determinism). This enables capturing the behaviour of handshaking circuits. A timed marked graph can capture not only the inter-dependency and concurrency in asynchronous circuits but also the timing of certain events occurring within the circuit.

As a type of Petri net, marked graphs are bipartite graphs. Informally, a marked graph is a Petri net where each place has exactly one transition in its preset and exactly one transition in its postset. Also, a marked graph is guaranteed to be a safe Petri net, meaning that places can hold at most one token at any moment in time. These characteristics enable deriving a simple definition for a marked graph.

Definition 26 (Marked Graph)**.** *A marked graph is a 3-tuple* $MG = (T, P, M_0)$*, where* T *is the set of transitions,* $P \subset \{(u, v) : u, v \in T\}$ *is the set of edges connecting transitions, and* $M_0 \subset P$ *is the subset of edges initially marked (the initial marking). The marking* $M_i \subseteq P$ *corresponds to the subset of places holding tokens at some given instant* $i \in \mathbb{N}$*. Of course,* $i = 0$ *corresponds to the initial state of MG, where the initial marking* M_0 *in in place.* M_i *represents the state of MG at instant* i*.*

Compared to a regular PN, MGs suppress the representation of places but marks still occupy their position between transitions. In a marked graph, token movements (i.e. state changes) obey a deterministic causality relation formally defined in this work called *token flow*.

Definition 27 (Token Flow)**.** *Let* M_i *be the marking of a marked graph MG at instant* $i \in \mathbb{N}$*. Let* $\bullet t = \{(u, v) \in P | v = t\}$ *be the preset of transition* t *and let* $t\bullet = \{(u, v) \in P | u = t\}$ *be the postset of transition* t*.*

Then, it is true that $\exists t \in T : \bullet t \subseteq M_i \implies \exists n \in \mathbb{N}_* : t\bullet \subseteq M_{i+n} \wedge \bullet t \nsubseteq M_{i+n}$*. This means that if all elements in the preset of a transition (*$\bullet t$*) are marked, there is a moment in the future (*$i + n$*) where all elements in the postset of that transition (*$t\bullet$*) will be marked and its preset* $\bullet t$ *will be unmarked.*

Transitions control the flow of tokens in a network through the firing process. As defined for any PN (MGs obviously included), when a transition fires, it removes tokens from all places in its preset and deposits tokens to all places in its postset. Said otherwise, when a transition fires, it simultaneously marks its postset and unmarks its preset. An MG transition can only fire after all edges in its preset are marked. Further limitations can be imposed as to when a transition is allowed to fire (including time counting and other conditions).

A valuable extension of the MG concept is adding labels to account for time in either places and/or transitions, giving rise to the *timed marked graphs* or TMGs. The former is relevant to this work and is accordingly precisely defined.

Definition 28 (Place-Delayed Marked Graph)**.** *Given a marked graph MG, it is possible to define a* **place-delayed marked graph** *as a 3-tuple PDMG* = (T, P, M_0)*, where* T *and* M_0 *are defined as in the corresponding MG and* $P \subset \{(u, v, d) | u, v \in T, d \in \mathbb{R}+\}$*. The edges, as in an MG, connect transition u to transition v, with a label d, representing the delay assigned to each edge.*

A token flowing into a PDMG edge (u, v, d) experiences a delay d before enabling the firing of a transition. That is, once receiving a token, an edge (u, v, d) must remain marked at least for the duration d before the token is removed. As in any PN, a transition can only fire in a PDMG after all its predecessor edges are holding tokens. When a transition fires it removes the tokens from its predecessor edges and deposits tokens in all of its succeeding edges.

3. ASYNCHRONOUS CIRCUIT DESIGN

3.1 Principles of Asynchronous Design

Most synchronous circuits rely on the assumption that the value on the inputs of all its registers will only be sampled at the rising (or/and falling) edge of the clock signal. Refer to Figure 3.1(a) to notice that in a classic linear pipeline, this enables to definition of timing constraints for the maximum delay in combinational logic paths, which must be typically smaller than the clock period. Using synchronous design techniques allows ignoring gate and wire delays, as long as clock timing constraints are respected. In other words, combinational logic is allowed to switch as it computes data during, say, the interval between two consecutive rising clock edges, but the logic outputs must be stable and correct at each such edge. Having this simple model for circuit design is possible only because the clock is a global and periodic signal, *i.e.* its edges only occur at specific and known points in time and occur simultaneously (this is an assumption) at every point where required. Hence, in synchronous circuits, events will only take place at specific moments; time can thus be treated as a discrete variable.

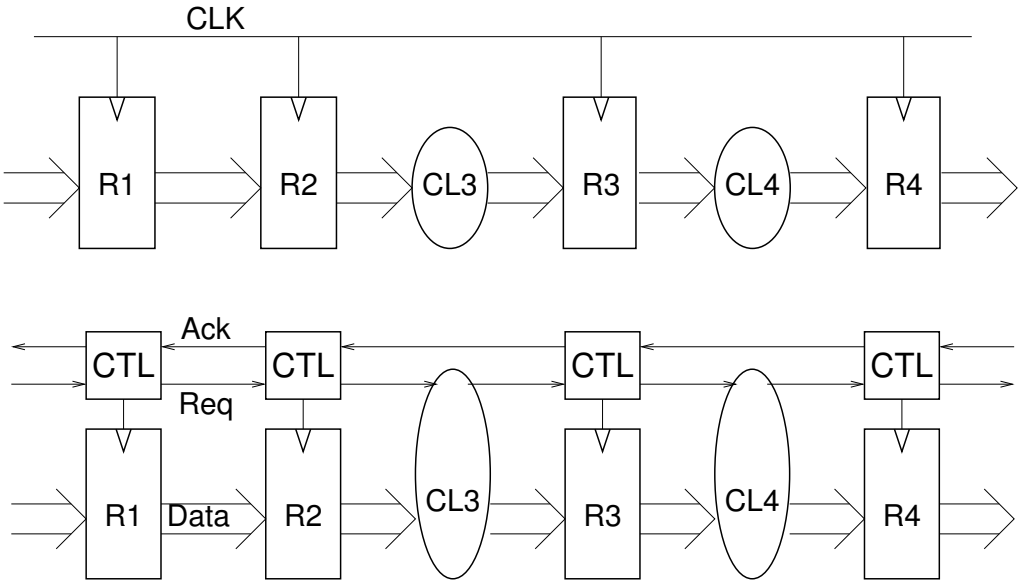


Figure 3.1: Simplified linear pipeline circuit structure using (a) synchronous and (b) asynchronous designs. Blocks CL_i represent combinational logic, *R* represent registers, and *CTRL* indicates control logic. Adapted from [Spa20].

A look at the alternative, Figure 3.1(b) shows that in asynchronous circuits there is no such thing as a single clock to simultaneously signal data validity on the inputs of all registers. Here, events can happen at any moment, and time must, quite often, be regarded as a continuous variable. Asynchronous designers rely on local handshake protocols for

communication and synchronisation, and on different design templates to build circuits, each with its own specific assumptions about gate and wire delays [BOF10].

Asynchronous design templates can be broadly classified in two main families: bundled-data (BD) [Sut89] and self-timed [Mar90]. Using the DCDT metamodel, the main distinction leading to this classification relies in the DCDT Channel entity, where communication links and protocols differ widely across BD and self-timed templates. Refer to Figure 3.2 to note that the design of a BD circuit is similar to a synchronous one; the difference is that BD relies on carefully matching the delay of data path combinational logic blocks and controlling registers to the delays in the control block that generates a local clock, rather than employ a single, global clock signal.

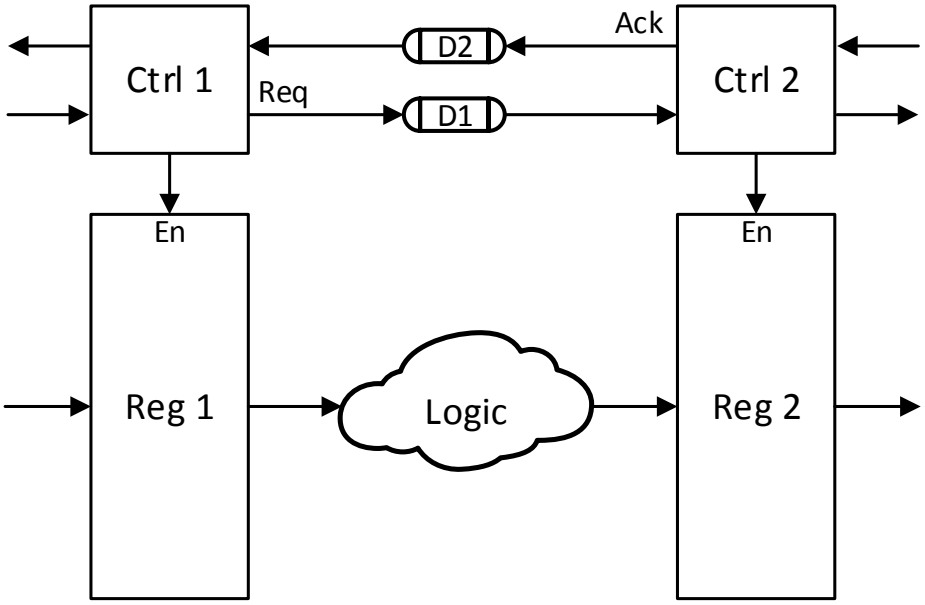


Figure 3.2: Example of a typical BD asynchronous pipeline fragment, with delay elements explicitly represented as *D*1 and *D*2 on request (*Req*) and acknowledge (*Ack*) paths on the control part of the circuit; blocks *Ctrli* represent the local stage controllers. Blocks *Regi* are data path registers and the *Logic* cloud represents combinational data processing between pipeline temporal barriers (the registers).

Communication and synchronisation in BD circuits are accomplished through some handshake protocol, the more common choices being 4-phase, return to zero (RTZ) protocols [BOF10]. Again using the DCDT concepts, the channel protocol characteristics provide a way to classify asynchronous design templates. Data representation in BD circuits follows the same Boolean encoding used in synchronous circuits[1]. This means that according to DCDT BD and synchronous design share a same channel communication link type. Also, unlike what happens in synchronous circuits, controllers are local and usually comprise just a few logic gates. An illustrative extreme example is the very efficient MOUSETRAP pipeline stage controller, which includes only an *XNOR* logic gate and a 1-bit latch [SN07]. This simplicity helps, since controllers are replicated at each and every stage of a circuit.

[1]This is not the case for self-timed circuits, as Section 3.2 details.

A major hurdle in BD circuit design is how to guarantee that the control and data paths are always precisely delay-matched, since the data and control flows typically run parallel to each other. This is the reason why in Figure 3.2 delay elements (DEs) are explicitly shown in the request and acknowledge paths. Much research exists to further the design of DEs to achieve working BD circuits. As an illustration of such research efforts, Heck [Hec18] developed a PhD Thesis where the focus was obtaining a single programmable delay element to support the design of asynchronous BD circuits resilient to timing errors. This was, in fact, the culmination of joint research between a research group at the University of Southern California in the USA and the authors' research group, which had previously generated research results on several aspects of DE design for BD circuits [SMT+17, THG+16, SMT+15, HHS+15]. Specifically addressed research in these publications are analysis and optimisation of programmable DEs, performance analyses on how fine-grained and coarse-grained delay adjustments work in practice, and control of the effect of voltage variations over the delay-matching characteristic of DEs. Relating to DCDT, DEs are thus part of the design style set of cells to use and affect the architectural rules of asynchronous BD templates.

The required delay-matching design effort is one of the main issues in designing robust circuits using the BD family of templates. BD circuit implementations can be as small as an equivalent synchronous implementation, or even smaller, as described for example by Teifel in [Tei06]. However, BD techniques share some of the disadvantages that plague synchronous design techniques, including a potential reduction in circuit robustness to variations, mostly due to the decoupling of control and data parts of the circuit.

3.2 Asynchronous Self-Timed Design

A fundamental difference between BD and ST design is that the latter relies on data encoding schemes that allow data to carry their own validity information, which enables receivers to compute the presence or absence of data at inputs/outputs, and renders possible the local exchange of information in a mostly delay insensitive way, more easily matching control and data information processing. Due to this characteristic, ST circuits can adapt more gracefully to wire and gate delay variations, and are thus a good choice to achieve circuit robustness. On the negative side, ST circuits further robustness often at the expense of larger area and/or power.

ST designs rely on the use of *delay-insensitive* (DI) codes to represent data [Ver88]. Binary DI codes use only part of the Boolean encoding spectrum possible over n bits. An n-bit binary code potentially allows representing 2^n distinct *codewords*. A DI code pledges the use of only a subset of these to achieve the delay insensitivity property. Verhoeff [Ver88] explores the basic details of the theory behind DI codes. Practical n-bit binary DI codes

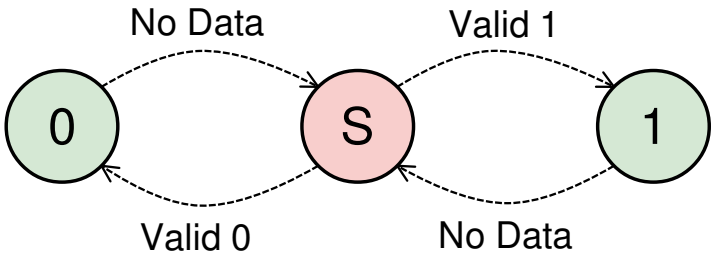


Figure 3.3: A state transition diagram with the basic protocol for transmitting data in a DI channel. S stands for the spacer.

consist of a set of codewords to represent data, and a distinct *n*-bit value to represent the absence of data. The latter is called a *spacer* or *null*. Since valid codewords and the spacer never include all 2^n different *n*-bit combinations, some of these are invalid and *code efficiency* is an issue. Verhoeff [Ver88] treats this defining the *rate R* of a code. Given a code with M valid codewords and a length of *n* bits, its rate is $R = (log_2 M)/n$. Of course, $0 \leq R \leq 1$ always holds. Verhoeff proves that *n*-bit Sperner codes, where every codeword has a structure (n **div** 2)-out-of-n, are DI codes, and that such codes provide the highest possible efficiency. For example, if code length is $n = 20$ bits, all codewords with 10 bits at 1 and 10 bits at 0 are valid Sperner codewords, and there are a total of $184,756$ distinct codewords in this code. Even if this is much less than the $1,048,576$ 20-bit combinations of the non-DI, 20-bit Boolean code, this Sperner code is much more efficient than the often-used 20-bit dual-rail code, which has only $1,024$ valid codewords. As $n \to \infty$ the rate of Sperner codes tends to 1, but practical *n*-bit Sperner codes, and thus any DI code, have $R << 1$, while a non-DI code such as the *n*-bit Boolean code has $R = 1$ for any *n*.

To understand how DI codes achieve delay insensitivity, Figure 3.3 shows the basic communication protocol of a channel as a state transition diagram for transmitting data on a 1-bit DI channel. Clearly, three values are necessary and the minimum code length to represent 0, 1 and the spacer is 2. Assume transmission always starts in the spacer (S) state. A transition from the spacer to 1 (or to 0) characterises the transmission of a valid 1 (resp. 0) and a transition from 1 (resp. 0) to S characterises data removal. DI communication protocols thus assume there is a spacer between any pair of consecutive data values. This in fact depicts just a specific family of communication protocols, often associated with DI codes, that can be called *return to spacer* (RTS) protocols. Since the spacer is frequently a value with all bits in 0, a more commonly used term is *return to zero* (RTZ) protocols, although other spacer values are sometimes used.

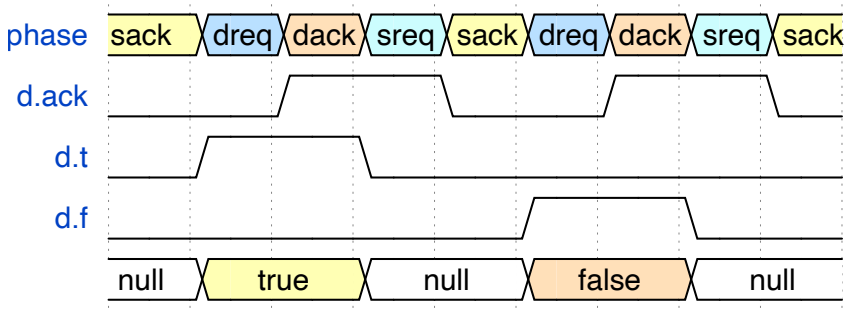
Asynchronous circuit pipelines can be implemented using one of two approaches: (i) half-buffer, where data and spacers alternate occupying successive pipeline stages; (ii) full-buffer, where all stages can contain valid data at every moment. Although at first counter-intuitive, as half-buffer schemes seem wasteful, these are more frequently used. They are simpler to build and in general, execute faster than full-buffer schemes. Also, the former ease robust circuit design.

In circuit design, often used examples of DI codes are the *k*-of-*n* codes, where *n* is the number of wires used to represent data (or its absence) and *k* is the number of wires that must be at a given logic value in any codeword (usually using the 1 bit value for these wires and the 0 bit value for the others). Albeit different codes are available in the contemporary literature (see e.g. [Ver88]), according to Martin and Nyström [MN06] the most practical class of DI codes is the 1-of-*n* (or one-hot), and more specifically the 1-of-2 code. The latter is the basis to form codes to represent any *n*-bit information using two wires to denote each of the *n* bits, producing the family of the so-called *dual-rail* codes. Furthermore, Martin and Nyström argue that DI codes can be coupled to either 2-phase or 4-phase handshake protocols, but 2-phase protocols often lead to more complex circuits. Thus, 4-phase is frequently chosen by self-timed circuit designers. In fact, the majority of self-timed designs available in the state-of-the-art, from networks-on-chip [BCV+05, PMMC10], to general purpose processors [MNW03], and network switches [DLD+14], primarily rely on 4-phase protocols and dual-rail or 1-of-4-based codes[2]. The 1-of-4 code is equivalent to two 1-of-2 codes considered together, but these codes are different. In fact, switching a 1-of-4 codeword (say 0100, corresponding to decimal 2) to a 0000 spacer implies switching just 1 bit, while the same value encoded in dual-rail, 1001 (equivalent to 10 in binary or decimal 2), requires two bits to switch to reach the same spacer. Thus 1-of-4 codes present roughly a 50% switching power advantage over dual-rail encoding.

Figure 3.4(a) shows the operation of a 4-phase, dual-rail, RTZ DI channel *d*, carrying a single bit datum in two wires, *d.t* and *d.f*, and one signal *d.ack* to control data flow. Note that valid data are encoded using exactly one wire at 1, *d.t*=1 for a logic 1 and *d.f*=1 for a logic 0. The value with both wires at 1 does not correspond to any valid datum and is never used. Figure 3.4(b) shows a waveform for transmitting two data values. Note the control flow allowed by the *d.ack* wire combined with the data in wires *d.t* and *d.f*.

	d.t	d.f
null	0	0
false	0	1
true	1	0

(a) Data encoding.



(b) A waveform for transmitting two values.

Figure 3.4: A DI, dual-rail, RTZ channel, and its operation.

Communication starts with a spacer, all wires at 0. Note that the *ack* wire also starts at 0, which signals the receiving side is ready to receive. Next, the sender puts a valid 0 in the channel, raising the logic value of *d.f*, which is acknowledged by the receiver raising

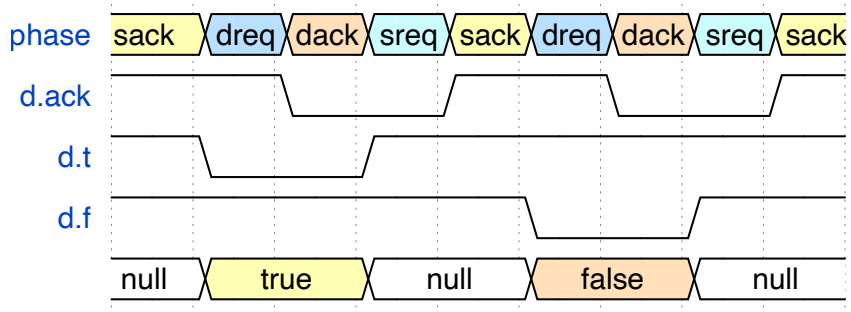
[2]Note that 1-of-2 (resp. dual-rail) and 1-of-4 codes have the same rate, $R = 1/2 = 0.5$, while a 1-of-8 code has a rate of just $R = 3/8 = 0.375$ and is accordingly never used.

the *ack* wire. After the sender receives *ack*, it produces a spacer-to-end communication, bringing all data signals in the channel back to 0. The receiver then lowers its *ack* signal, after which another communication can take place. Since all signals need to go to 0 before a data transmission starts, the RTZ denomination applies.

Another protocol for dual-rail ST design is the return-to-one (RTO) protocol, described in [MGC12a]. RTO is similar to RTZ, but its data values are inverted compared to the latter, and the spacer representation is also distinct. As Figure 3.5(a) shows, a spacer is a value with all wires at 1 and valid data is represented by a single wire at 0, *d.t*=0 for a logic 1 and *d.f*=0 for a logic 0. Figure 3.5(b) depicts an example RTO data transmission, which starts with all wires at 1 in the data channel.

	d.t	d.f
null	1	1
false	1	0
true	0	1

(a) Data encoding.



(b) A waveform for transmitting two values.

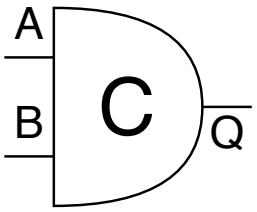
Figure 3.5: A DI, dual-rail, RTO channel, and its operation.

As soon as the sender puts valid data in the channel, the receiver may acknowledge it by lowering *ack*. Next, all data wires must return to 1 to denote a spacer, ending transmission. When the spacer is detected by the receiver, it raises the *ack* signal and operation resumes. The idea behind the RTO protocol allows a better design space exploration for asynchronous circuits, enabling optimisations in power [MGC12b] and robustness [MPC14]. Furthermore, as demonstrated in [MTMC14], RTZ and RTO can be mixed in a same dual-rail design and the conversion of values between them requires only an inverter per wire. According to Martin and Nyström [MN06], such conversion is DI and does not compromise the robust functionality of a self-timed circuit. This thesis refers to signals operating under the RTZ (RTO) protocol as RTZ (RTO) signals.

The correct operation of ST circuits depends on adherence to the *indication principle*, which dictates that the output of a circuit must acknowledge the state of its inputs, meaning a circuit generates output only when all input data are valid. Indication requires that circuits present some form of *hysteresis*. Hysteresis is the property of a gate to hold its output stable until a transition condition is met.[3] The C-element, depicted in Figure 3.6, is a fundamental hysteretic gate used in constructing many asynchronous circuits. It gener-

[3]The term hysteresis is analogous to its use in Schmitt-trigger circuits, but instead of expressing the transition condition as a voltage level at an input, it expresses transitions as specific combinations of logical values at the gate inputs.

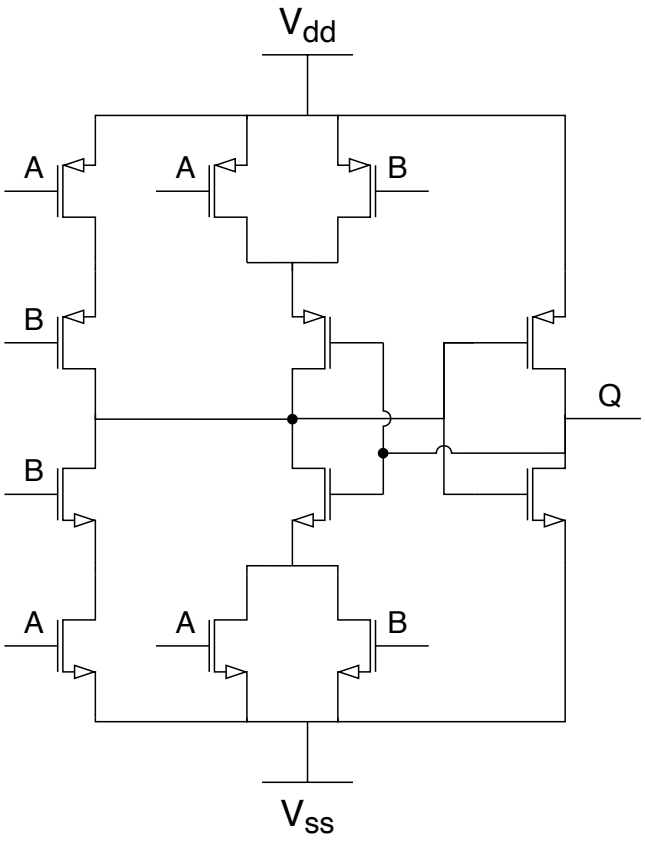
ates an output only when all inputs are identical, which ensures it individually respects the indication principle.



(a) Logic symbol.

A	B	Q_i
0	0	0
0	1	Q_{i-1}
1	0	Q_{i-1}
1	1	1

(b) Truth table.



(c) A CMOS static implementation.

Figure 3.6: A 2-input C-element.

3.3 Classification of Asynchronous Circuits

A seasoned reader might have encountered the term *quasi-delay-insensitive* (QDI) being used extensively, where this thesis opts for ST instead. It is helpful to draw attention to the distinction between the two terms and the overall classification of asynchronous circuits.

Sparsø [Spa20] classifies asynchronous circuits into three classes: *self-timed* (ST), *speed-independent* (SI) and *quasi-delay-insensitive* (QDI). He distinguishes them based on the timing assumptions that circuits of each class require to operate correctly. SI circuits allow gates to have arbitrary delays but require ideal wires with zero delays; this is an unrealistic assumption. An SI circuit can still operate correctly if all wire forks are isochronic, i.e. if the wire delay is equal on all forks for all wires. If all forks in a wire are isochronic, the wire delay can be bundled into the gate delay, and the SI requirement is met. QDI circuits relax the SI isochronic fork assumption by requiring that only specific, sensitive wire forks are required to be isochronic. However, Sparsø does not specify which forks must be isochronic,

or how to determine them. Martin [Mar90] provides a formal criterion to determine which forks must be isochronic in QDI circuits. Later, Keller et al. [KKM09] refine this criterion to be less strict. He also defines an *ideal* class of circuits that he calls *delay-insensitive* (DI) circuits, where no timing assumption is required. At last, Sparsø uses ST as a catch-all category for all circuits not fitting the previous classes; this includes bundled-data circuits.

This thesis' author finds Sparsø's classification to be of limited usefulness. First, in modern technologies where wire delays are not negligible, SI circuits are unfeasible. It also provides little distinction from the QDI class, making it hard to justify its existence. Also, the ST class is too broad to be useful; it provides little information about the nature of the circuit besides not having a clock and not being QDI. This thesis proposes a refinement of Sparsø's classifications. Figure 3.7 depicts the proposed classes and their relations. This classification system covers synchronous and asynchronous designs or combinations of both.

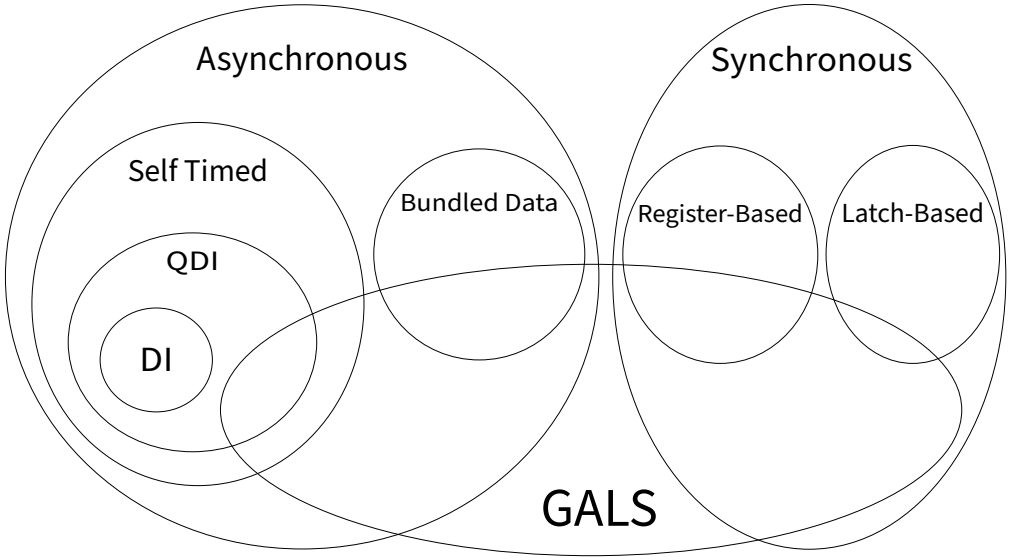


Figure 3.7: Classification of circuits.

In this thesis, ST circuits are all those that rely on the indication principle to ensure proper operation. That is, the computation completion is self-evident from the state of the wires. ST circuits often employ some form of DI coding and completion detection circuitry, but not always, especially on controllers. QDI circuits are a subset of ST circuits that only need to observe the isochronic fork assumption, as defined by Martin [Mar90]. The proposed classification excludes SI circuits but includes DI circuits as a subset of QDI circuits with no timing assumptions. Also, BD circuits are classified as a separate class from ST circuits, as they do not rely on the indication principle to detect computation completion. BD circuits rely on a tuned delay line matching the worst-case delay of the circuit to ensure correct operation. As such, the state of the wires carrying the computation result is insufficient to determine its completion.

It is noteworthy that parts of the same circuit can fit different classes. For instance, a GALS circuit can encompass register-based synchronous designs and bundled-data asynchronous circuits. Also, the bundled data circuit can employ QDI or even DI controllers. In fact, the asynchronous controller employed in Micropipelines [Sut89] is an example of a DI circuit.

Of course, the simply stated isochronic fork constraint can be hard to ensure, especially in large circuits. Furthermore, the strictness of limiting the timing assumptions to wire delays often comes with an area and performance impact. Also, in some cases, the robustness requirements are not so strict. Other self-timed design styles [BG03, Bre05, BE09] exist with more relaxed limitations on timing assumptions to offer better trade-offs among robustness, performance, area and power.

3.4 Self-Timed Sequential Circuits

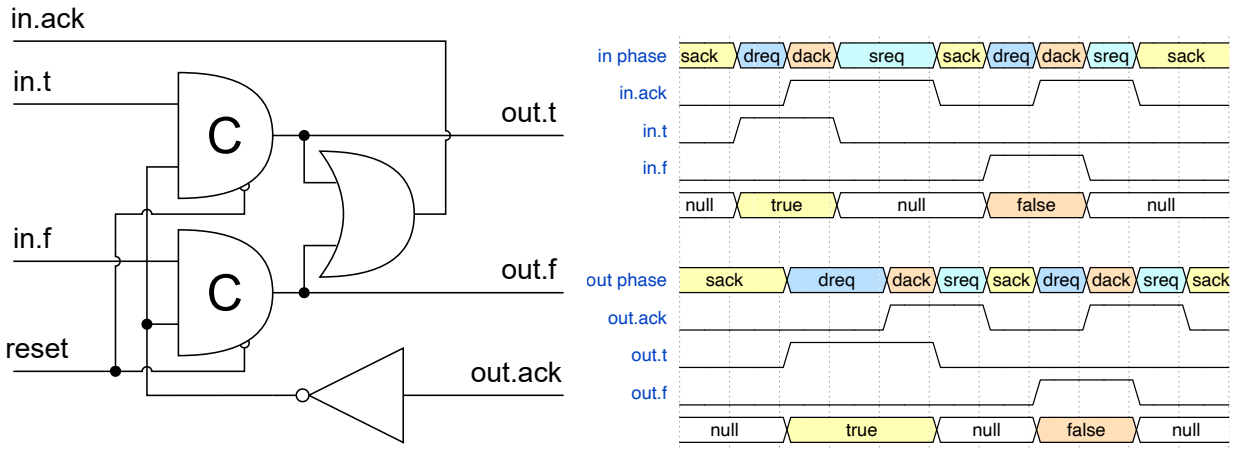
Registers are storage elements that enable the sequencing of operations in digital circuits. They are paramount to performing complex computations by holding circuit states, providing ways to increase performance, allowing to break computations into small chunks and have each chunks processed sequentially. Of course, this enables to use of pipelines and other, more advanced, forms of parallelism.

In synchronous circuits, a clock guides the progression of computation, dictating when registers capture new data. As Section 3.2 discusses, computation progress in asynchronous circuits depends on the handshake protocol, where registers act as active handshake coordinators. They consume input data and act as producers to their output. This requires registers to handle the handshake process simultaneously in two channels.

ST registers employed in circuits using 4-phase protocols transition between data and spacer. These registers can be either full- or half-buffers. A full-buffer register is capable of completing a full handshake cycle on both sides independently. Whereas a half-buffer register is capable of completing half a handshake cycle on both sides at each computation step. A half-buffer register must only accept a transition from the left-hand side after the consumer on the right-hand side has acknowledged the previous transition. It must uphold the indication principle; thus it can only transition its right-hand side after a transition occurs on its left-hand side. Also, the register can only acknowledge the transition on the left-hand side after capturing its state.

Figure 3.8 depicts a Weak Condition Half Buffer (WCHB), one possible RTZ half-buffer register implementation. In this Figure, the left-hand side is labelled `in`, and the right-hand side is labelled `out`. At the core of the WCHB lays two C-elements (see Section 3.2). The C-element captures the transition arriving from the left-hand side and propagates it to the right-hand side. The reset signal sets the initial state of the register to all outputs low. At

this stage, it is ready to capture data from the left-hand environment. The capturing process is controlled by the right-hand side acknowledgement signal (`out.ack`). When the right-hand side acknowledges the data transmission by raising its `ack` signal, it enables the C-element to capture the propagation of the spacer on the left-hand side. Similarly, when the right-hand side lowers its `ack` signal to acknowledge the reception of a spacer, it enables the propagation of new data. After a transition propagates, the OR gate generates the proper acknowledgement signal, signalling that it is safe for the producer on the left-hand side to transition. This OR gate is called a *completion detector* (CD), and its implementation is dependent on the DI coding and protocol adopted; on a dual-rail RTZ it is implemented using a 2-input OR gate.



(a) Schematic. (b) Waveform.

Figure 3.8: WCHB, a dual-rail RTZ half-buffer register.

Also, notice in Figure 3.8(b) how the delayed generation of the right-hand acknowledgement affects transition propagation in the register. After the rising transition of `in.t` propagates to `out.t`, the right-hand environment is slow to acknowledge the data reception. This delays the propagation and acknowledgement of the spacer in the left-hand channel. After data acknowledgement is received, the spacer propagates, and the next handshake cycle occurs as expected. The behaviour depicted here shows how the handshake protocol and ST registers can cope with delay variability in computations. Also, notice how a delay in the `dreq` phase on the right-hand channel results in a delay in the `sreq` phase in the left-hand channel, which are half a handshake cycle apart, highlighting the half-buffer nature of this register implementation. Full-buffer registers that allow one complete cycle separation between the two environments can be constructed from a sequence of two half-buffer registers. An example of a template employing full-buffer registers is the pre-charge full-buffer (PCFB) [OB02].

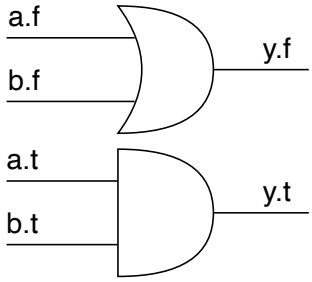
3.5 Self-Timed Combinational Logic

The ST circuits reliance on DI codes and handshake protocols requires special logic blocks to perform computation on DI-coded data. As Sparsø [Spa20] explains, logic blocks placed between handshake entities[4] must be transparent to the handshake process. Unlike a register, they cannot act as an active participant coordinating handshakes. Furthermore, ST circuits rely on the indication principle to guarantee computation completeness. The level of adherence to this principle varies within the ST circuit class: logic blocks for QDI circuits are required to respect the indication principle for every individual intermediary dual-rail encoded signal. To respect the indication principle, a combinational logic block should only produce a codeword at its output when all of its inputs are codewords. Also, when employing a 4-phase protocol, the logic block must only produce a spacer when all its inputs are spacers. Also, it is implied in the previous statements that to adhere to the indication principle, a logic block must preserve its output stable until all of its inputs are presented with valid data or spacers. These requirements mean that to uphold the indication principle, a logic block must present *unateness* and *hysteresis*. Both described in Section 2.1.1.

To understand these requirements, consider a naive approach to design a 2-input dual-rail RTZ AND gate. First, when considering the encoding of the true and false values, a designer might be tempted only to consider the combination of codewords and overlook all other bit combinations as “don’t care”. When applying logic minimisation, one might arrive thus at the circuit depicted in Figure 3.9. This circuit produces correct results for all dual-rail codeword combinations but does not respect the indication principle. When any of its inputs are presented with a false-value codeword, the output immediately produces a false-value codeword at the output. Moreover, during the reset phase, after producing a true-valued codeword, the output will switch to null when any of the two inputs offer a null value. This behaviour at both the evaluation and reset phases violates the indication principle, as the output does not reflect the state of the inputs. Consequently, it also violates the handshake protocol transparency, as the output of the logic block can advance to the next phase before all of its inputs do. A dual-rail gate that violates the indication principle is called *weakly indicating*.

Solving this issue requires gates respecting the properties of unateness and hysteresis. A gate with these properties is the C-element introduced in Section 3.2. Using C-elements, it is possible to construct arbitrarily complex self-timed logic gates respecting the indication principle. *Delay Insensitive Minterm Synthesis* (DIMS) [Sin81] is an approach to designing QDI logic blocks from C-elements and OR gates. Figure 3.10 depicts a functionally correct dual-rail RTZ 2-input DIMS AND gate. Each distinct input rail combination is

[4] Handshake entities are entities capable of coordinating handshakes, e.g. ST registers or the external environment that interact with the circuit.



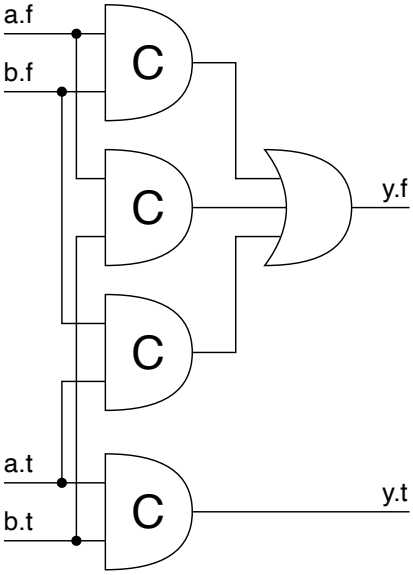
(a) Schematic.

	N	T	F
N	N	N	F
T	N	T	F
F	F	F	F

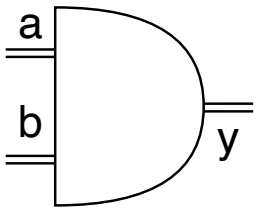
(b) Truth table. N is null, T is true, and F is false.

Figure 3.9: Naive weakly indicating dual-rail RTZ 2-input AND.

merged by a C-element[5]. C-elements outputs are combined using OR gates, where needed, to produce the output rails. Assuming the isochronic fork assumption is respected, the resulting dual-rail logic gate respects the indication principle. During the evaluation phase, the gate can only produce a valid output when all of its inputs are valid. It also respects the indication principle during the reset phase, as it will only produce a null when both inputs are null. A dual-rail logic gate that respects the indication principle is called *strongly-indicating*.



(a) Schematic.



(b) Symbol

	N	T	F
N	N	-	-
T	-	T	F
F	-	F	F

(c) Truth table. N is null, T is true, F is false, and - means “hold output value”.

Figure 3.10: Dual-rail, 2-input DIMS AND gate.

DIMS can be used to design libraries of strong-indicating self-timed logic gates. Such dual-rail gates can be combined to compose arbitrarily complex logic blocks. The resulting logic block respects the indication principle and is thus also called strong-indicating.

[5]Codeword combinations in dual-rail are just T-T, T-F, F-T and F-F.

3.6 NCL: A More Efficient Hysteretic Gate

DIMS gates clearly incur high area overheads. An n-input dual-rail gate presents 2^n codeword combinations, each requiring a 2-input C-Elements. Each C-Element comprises at least 12 transistors on a fully static CMOS implementation. Also, increasing the number of codeword combinations increases the number of sensitive isochronic forks. In light of these limitations, Fant et al. propose *Null Convention Logic* (NCL) [FB96, Fan05], a design technique that reduces the area required to implement self-timed logic gates. It employs hysteretic gates, called NCL gates, to synthesise logic respecting the indication principle.

NCL gates are often called *threshold gates*, but this is imprecise because they do not precisely implement threshold logic functions (TLFs) as defined, e.g. in [Hur69]. Instead, these gates implement modifications of such TLFs coupled to specific mechanisms to ensure the completeness of input criterion [FB96]. The modified TLF is defined here; for inputs not covered in the *TLF*, the hysteretic gate holds its previous output.

Definition 29 (Threshold Logic Function)**.** *An n-variable* **threshold logic function (TLF)** τ *is an n-variable unate partial function defined by a* **activation threshold value** $T_\uparrow \in N^*$*, a* **deactivation threshold value** $T_d ownarrow$*, and* **weights** $w_i \in N^*$ *assigned to each of its variables* x_i *such that:*

$$\tau = \begin{cases} 1, & \sum_{i=0}^{n-1} w_i x_i \geq T_\uparrow \\ 0, & \sum_{i=0}^{n-1} w_i x_i \leq T_\downarrow \end{cases} \tag{3.1}$$

Hurst [Hur69] defines threshold gates with the threshold and weights being real numbers. This work restricts attention to non-zero, natural thresholds and weights. TLFs can be either negative or positive unate in a given input – but not binate. However, to ensure the completeness of input criterion, NCL gates must be positive unate in all their inputs, as they target RTZ templates, where data validity is given by wires at 1.

NCL gates are threshold gates with a constant deactivation threshold of 0; henceforth, "threshold" is used as synonymous with activation threshold when discussing NCL gates. Figure 3.11 depicts some possible NCL gates. Notices that gates with threshold 1 are equivalent to OR gates, and gates with a threshold equal to the number of inputs are equivalent to C-Elements.

Although this thesis presents the concept of TLFs, it prefers the production rules (PR) presented in Section 2.2 to describe threshold gates. PRs are more expressive and versatile, allowing to capture the behaviour of a greater gamut of hysteretic gates. This thesis also employs the PR representation due to its similarity to boolean functions employed by traditional EDA synthesis. As an example of a threshold gate, consider the 2-input C-element

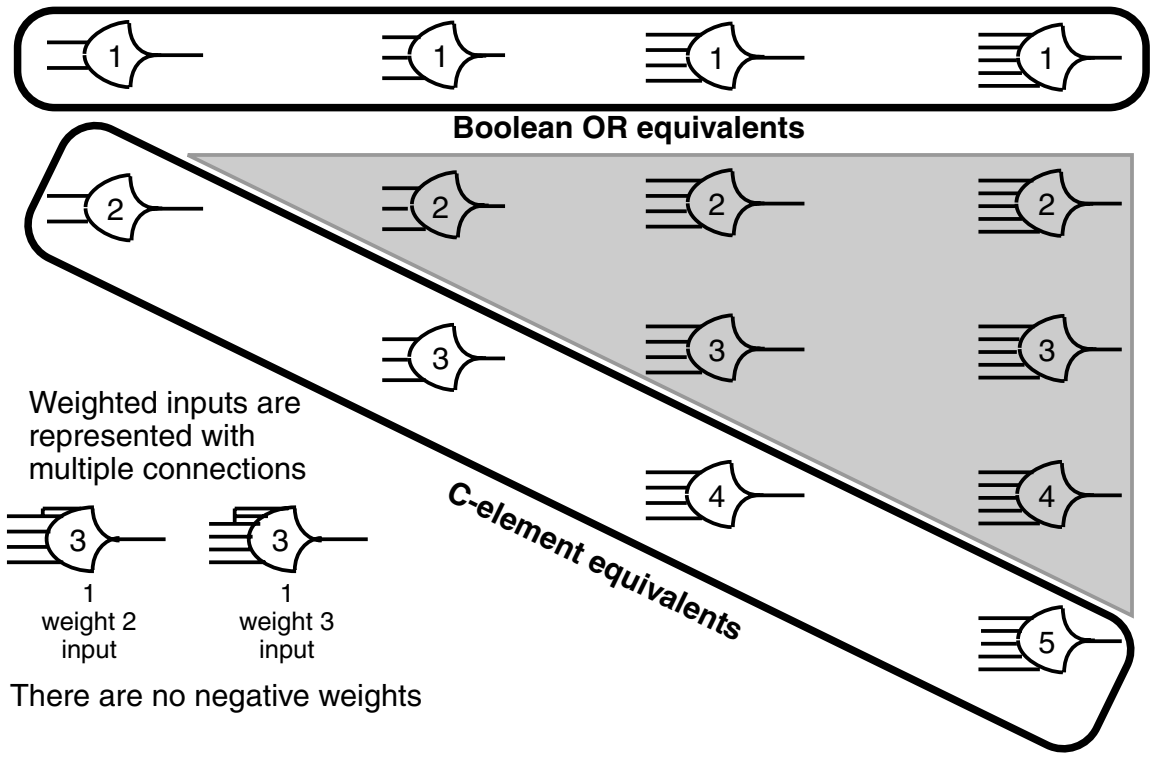


Figure 3.11: NCL threshold gates. Adapted from [Fan05].

depicted in Figure 3.6. As depicted in Figure 3.11, the C-Element is also an NCL gate. Its activation threshold is 2, and its deactivation threshold is 0; the corresponding PR pair is:

$$\langle A \cdot B \rightarrowtail Q\uparrow,\ !A \cdot !B \rightarrowtail Q \rangle \tag{3.2}$$

Its characteristic function is $Q_i = A \cdot B + Q_{i-1} \cdot (A + B)$ A careful reader can check the characteristic function against the truth table in Figure 3.6(b).

Production rules are suitable to define not only the behaviour of hysteretic gates but also the behaviour of rails in dual-rail logic gates and combinational blocks. From this, it is possible to select hysteretic gates that implement that rail function. To exemplify this process, take the strongly indicating dual-rail RTZ AND gate depicted in Figure 3.10. The production rules for the activation of these rails are:

$$a_t \cdot b_t \rightarrowtail y_t\uparrow \tag{3.3}$$

$$a_f \cdot b_f + a_t \cdot b_f + a_f \cdot b_t \rightarrowtail y_f\uparrow \tag{3.4}$$

Notice that for simplicity, the forks in these PRs are implicit. The corresponding production rules for the rails deactivation are, by protocol, equal to input rails low:

$$!a_t \cdot !b_t \rightarrowtail y_t\downarrow \tag{3.5}$$

$$!a_f \cdot !b_f \cdot !a_t \cdot !b_t \rightarrowtail y_f\downarrow \tag{3.6}$$

Mapping these assertion functions to threshold gates yields the NCL implementation depicted in Figure 3.12. The PR pair driving the rail y_t is the PR pair for the C-element; whereas the PR pair for y_f corresponds to the threshold-3 4-input NCL gate with two weight-2 inputs depicted in Figure 3.13. Using this gate instead of 3 C-Elements and OR-gate from the DIMS expansion reduced the transistor count required to implement this AND gate from 56 to 36.

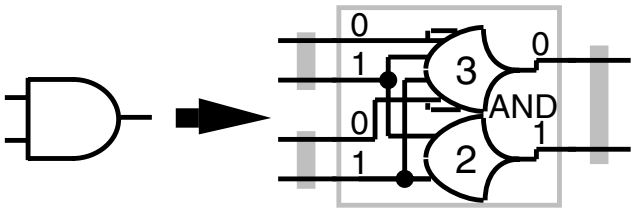
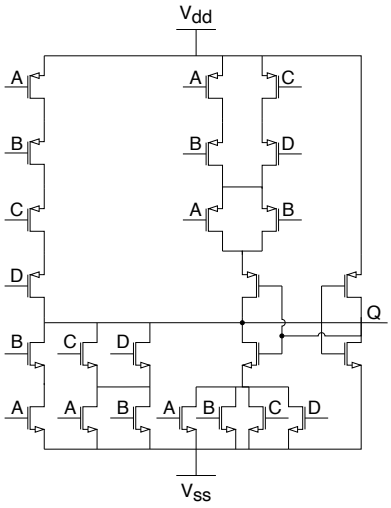


Figure 3.12: NCL AND2 dual-rail RTZ gate. The grey bands indicate where the dual-rail pair is bound to respect the indication principle. Adapted from [Fan05].



(a) Static CMOS implementation comprising 24 transistors.

A	B	C	D	Q_i
0	0	0	0	0
0	0	x	1	Q_{i-1}
0	0	1	x	Q_{i-1}
0	1	0	0	Q_{i-1}
x	1	x	1	1
x	1	1	x	1
1	0	0	0	Q_{i-1}
1	x	x	1	1
1	x	1	x	1
1	1	x	x	1

(b) Truth table depicting the hold states. Here x means “don’t care”.

Figure 3.13: Threshold-3 4-input NCL gate with two weight-2 inputs employed in the AND2 gate presented in Figure 3.12.

The automated synthesis of NCL requires specialised tools to perform technology mapping and optimisation. Uncle [RST12] is one such tool; it employs a two-step synthesis. First, a commercial EDA tool synthesises an RTL description to a single rail netlist. Then a custom tool translates it to a dual-rail netlist using predefined dual-rail gate implementations and generates the backward propagation network. This custom tool also optimises the dual-rail netlist using specialised NCL optimisation techniques.

3.7 SDDS-NCL: Spatially Distributed Dual Spacer NCL

Since NCL gates' deactivation threshold is always 0, different NCL gates are distinguishable solely from their activation. One might consider using conventional EDA tools to automate the mapping of NCL gates from their activation expressions only. However, conventional EDA tools are not capable of correctly synthesising circuits comprising NCL gates. Such tools are unaware of the full hysteretic behaviour of these gates, and they also require negative unate gates to implement combinational logic. These restrictions pose challenges to the adoption of conventional EDA tools for the synthesis of ST circuits using NCL gates. If standard EDA tools were capable of synthesising combinational logic from positive unate gates alone, using the RTZ protocol, NCL gates could be selected solely from their activation expressions; the deactivation behaviour could be safely ignored as it only affects the reset phase of the circuit in the RTZ protocol. However, when inversions occur, the protocol changes from RTZ to RTO, and the computation occurs in the gate's deactivation. An EDA tool oblivious to the hysteretic behaviour of NCL gates assumes all gates are conventional; they assume that the deactivation expression of a gate is always equal to the complement of its activation expression described in the liberty file. When accounting for the inversion requirement and the obliviousness to the complete behaviour of hysteretic gates, the resulting circuit is most likely implemented incorrectly.

Spatially Distributed Dual Spacer Null Convention Logic (SDDS-NCL) [MNM+14, MTMC14, MBSC18] is a technique that eliminates these restrictions and enables the use of conventional EDA tools to synthesise and optimise self-timed circuits. SDDS-NCL employs two types of hysteretic gates (NCL and NCLP)[6] and two distinct protocols (RTZ and RTO) to accommodate inversions. NCLP gates [MGC12a, MOPC13] are hysteretic gates where the activation threshold equals the sum of all its input weights (i.e. all of its inputs at 1), and the computation is expressed in its deactivation threshold. For comparison, the computation in NCL gates is expressed in its activation threshold, and their deactivation threshold is always 0 (i.e. all of its inputs at 0). NCL gates' deactivation makes them suitable to operate during the reset phase of the RTZ protocol, whereas NCLP gates' activation threshold makes them suitable to operate on the reset of the RTO protocol.

A virtual function characterises the behaviour of NCL and NCLP gates' computation. This virtual function is used by the standard EDA tool to select the gate during synthesis. For positive unate NCL gates, the virtual function equals its activation expression; and for positive unate NCLP gates, it equals the complement of its deactivation expression. For negative unate NCL and NCLP gates, this relationship is inverted; the virtual function for the negative unate NCL gate is expressed in the negation of its deactivation function, and the

[6] In [MBSC18], the authors also mention INCL and INCLP gates as proper classes. However, these are equivalent to an NCL and NCLP gate followed by an inverter. For simplicity, this work considers the former part of classes NCL and NCLP, respectively.

virtual function of a negative unate NCLP gate is expressed in its activation function. Every NCL gate has an NCLP counterpart with the same virtual function, and vice-versa. As an example, consider the NCL gate depicted in Figure 3.13, its PR pair is:

$$\langle A \cdot B + A \cdot C + A \cdot D + B \cdot C + B \cdot D \rightarrowtail Q\uparrow,\ !A{\cdot}!B{\cdot}!C{\cdot}!D \rightarrowtail Q\downarrow \rangle \tag{3.7}$$

Its NCLP counterpart's PR pair is:

$$\langle A \cdot B \cdot C \cdot D \rightarrowtail Q\uparrow, !A{\cdot}!B{+}!A{\cdot}!C{\cdot}!D{+}!B{\cdot}!C{\cdot}!D \rightarrowtail Q\downarrow \rangle \tag{3.8}$$

Their virtual function is:

$$Q = A \cdot B + A \cdot C + A \cdot D + B \cdot C + B \cdot D \tag{3.9}$$

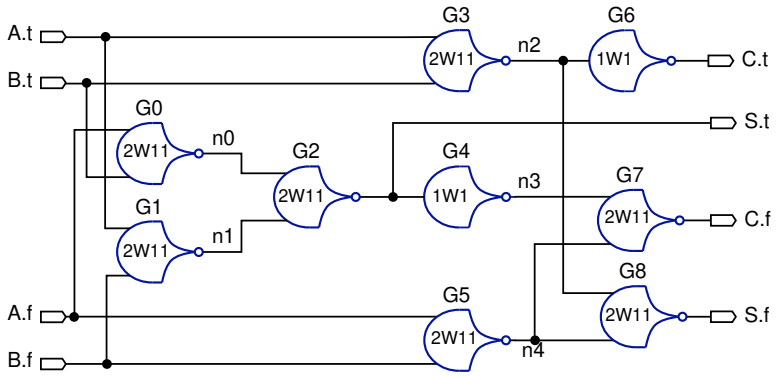
It is an exercise left to the reader to verify that the virtual function matches the inverse of the deactivation expression of the NCLP gate.

The synthesis occurs in two steps; first, the EDA tool selects NCL gates based on their virtual functions. The EDA tool, unaware of the hysteretic behaviour or the protocol, produces a potentially incorrect circuit. A graph colouring algorithm transverses the circuit breath-first to identify protocol changes due to inversions. This algorithm then swaps between NCL and NCLP gates of the same virtual function where applicable. The intermediary synthesis result and the finally corrected circuit are depicted in Figure 3.14.

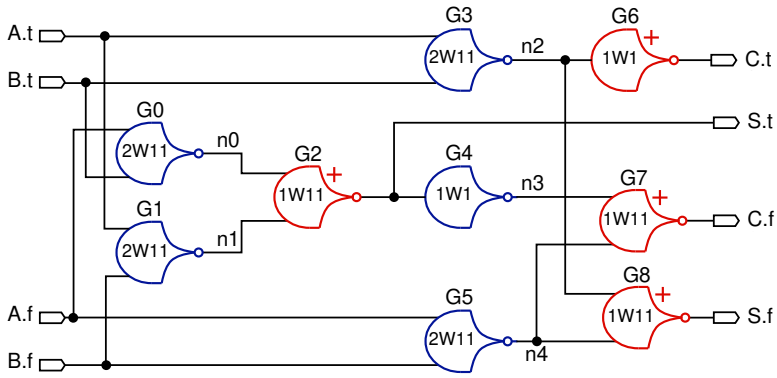
SDDS-NCL has been demonstrated in multiple experiments to be reliable in the construction of asynchronous circuits:

- In [MBSC18], it presents advantages in power and performance for combinational circuits compared with results from Uncle [RST12].
- In [SWMC19], with the introduction of Pulsar, SDDS-NCL continues to show advantages over Uncle for sequential circuits.
- In [SWM+20, WSA+22], it shows to be resilient to extreme voltage scaling.

These peer-reviewed results show that SDDS-NCL presents a good level of robustness.



(a) Incorrectly mapped NCL netlist produced by EDA tool.



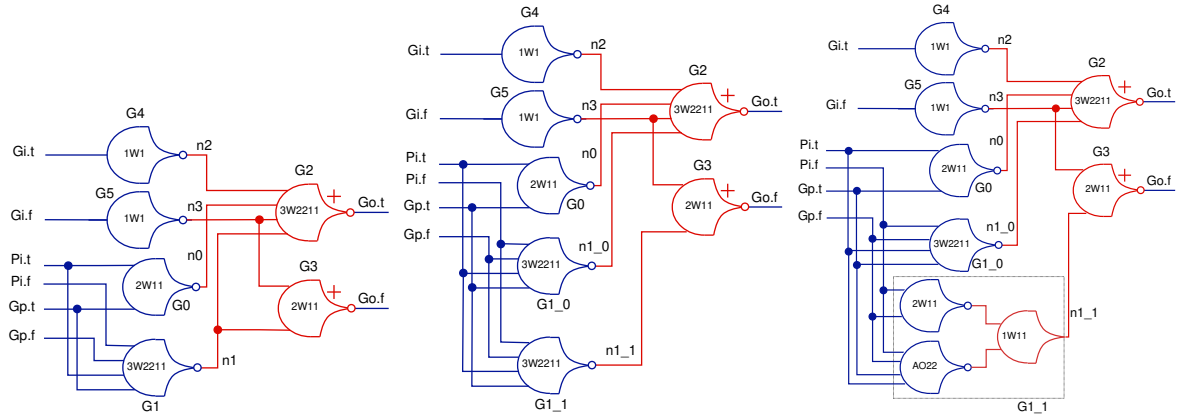
(b) Correctly mapped SDDS-NCL netlist after correction step.

Figure 3.14: Example of SDDS-NCL synthesis. Blue gates are NCL, and red gates are NCLP. Adapted from [MBSC18].

3.8 A critique of SDDS-NCL

SDDS-NCL is a template capable of generating QDI circuits; neither the interleaving of NCL and NCLP gates nor the inclusion of negative unate gates affects the timing assumptions set by pure NCL circuits. Additionally, it has been demonstrated that one of its main properties is to enable the use of traditional synchronous automatic synthesis tools to design asynchronous digital circuits. However, in practice, using such methods for asynchronous circuit synthesis is not without drawbacks. The synthesis method described initially in [Mor16, MBSC18] and later extended for sequential circuits in [SWMC19, Sar19] does not guarantee that the resulting circuit meets the QDI criteria. The use of conventional EDA tools for technology mapping and optimisation can result in a circuit with gate orphans, thus requiring timing assumptions other than just the isochronic fork assumption. This occurs because the EDA tool is unaware of the logic requirements, and it is free to perform as many optimisations as it sees fit as long as the logic cones implement the same Boolean function. The resulting circuit, although not necessarily meeting the criteria to be considered QDI, is still considered an ST circuit.

To illustrate the just discussed issue, consider the three Kogge Stone generator blocks depicted in Figure 3.15. They are all valid SDDS-NCL circuits, and from the perspective of the logical cones terminating in `Go.0` and `Go.1`, they all implement the same virtual functions. However, in (a), the only potential orphans are the wire forks in `n1`, `n3`, `Pi.t` and `Gp.t`; whilst (b) duplicates gate `G1` into `G1_0` and `G1_1`, transforming the potential wire orphan at the fork of `n1` into a gate orphan. In (c), this situation is further worsened by the EDA tool decomposing gate `G1_1` into three logic gates implementing the same virtual function.



(a) QDI implementation, adapted from [Mor16]. (b) ST implementation with orphan gate. (c) ST implementation with orphan decomposed gate.

Figure 3.15: Three possible SDDS-NCL Kogge Stone generator blocks.

To guarantee the correct operation of the circuit with gate orphans, the timing assumption must cover not only the wire forks as demanded by the isochronic fork assumption but also the orphaned gates. Since only one of the outputs (`Go.t` or `Go.f`) is activated at any time, during the reset phase of the handshake protocol, only `n1_1` or `n1_0` will have its transition to spacer acknowledged by the deactivation of either output. The other wire will be assumed to have transitioned to spacer before the propagation of the next valid data; this assumption dictates that the difference in delay between the paths passing through `G1_1` and through `G1_0` must be smaller than the time required for new data to arrive at either gate. This timing assumption can be generalised to any circuit comprising gate orphans; the difference in propagation time between the slowest and the fastest propagation paths leading to a dual-rail output must be smaller than the fastest delay in the acknowledgement network matching the propagation paths.

When reanalysing the circuits used to generate experimental results in previously published articles featuring SDDS-NCL and Pulsar, it was found that the occurrence of gate orphans in SDDS-NCL is not uncommon. The presence of gate orphans, combined with the lack of a mechanism to enforce the observance of the proposed timing assumption, explains the faults leading to missing data points in, e.g. [SMC20a, NSMC23]. Yet, the timing assumption is still met in most cases and, as experimental results have shown, still grants a good level of robustness against delay variations.

4. TIMING AND PERFORMANCE

Asynchronous circuits' timing and throughput analysis are often not as straightforward as they are on synchronous circuits. The latter rely on the clock as an external stable timing reference to control the propagation of data; their throughput is an integer multiple of the clock period, which is limited by the slowest propagation time. In synchronous circuits, static timing analysis (STA) is used to find the slowest propagation time and guide timing-driven synthesis and optimisations.

Conversely, asynchronous circuits rely on sequences of handshakes between concurrent entities to control data propagation; these handshakes occur in cyclic propagation paths, called *cycles*. The throughput of a handshake-oriented asynchronous circuit depends on the propagation time of its slowest cycle. However, the slowest cycle is not necessarily evident due to complex interactions and can even span multiple handshakes. Furthermore, the absence of a clock signal and the cyclic nature of the propagation paths in asynchronous circuits hinders the use of STA tools to perform timing analysis.

This Chapter focuses on the performance analysis of handshaking circuits, especially ST circuits. Section 4.1 presents a technique that breaks cycles of asynchronous circuits into multiple linear *propagation paths* and allows the use of conventional STA tools to analyse the delays of these propagation paths. Section 4.2 formally defines the *Half-buffer channel network*, a model to capture the behaviour of 4-phase handshake circuits.

4.1 Pseudo Synchronous WCHB

Conventional EDA tools rely on the clock signal to perform static timing analysis (STA) on a circuit. In synchronous circuits, the STA computes propagation delays between registers, inputs and outputs in acyclic paths. The clock signal provides a stable and predictable transition to begin timing analysis from. The propagation delay computation begins with a clock event triggering registers and primary inputs to switch and terminates at a register's input or a primary output. In the absence of a clock signal, such reference is lost. Furthermore, the WCHB pipeline presents combinational loops as depicted in Figure 4.1. These loops force standard EDA tools to insert loop breakers at unpredictable locations in the circuit. Thonnart et al. propose the Pseudo Synchronous WCHB [TBV12] technique to tackle these issues. The author of this thesis extended Pseudo-Synchronous WCHB in [SWMC19] to enhance the support of delay-annotated simulation. This extension introduces a pseudo-clock signal used during synthesis to provide a stable timing reference for STA, but which is never realised in the final circuit. It also provides predictable loop breakers to act as start and end points in the timing analysis.

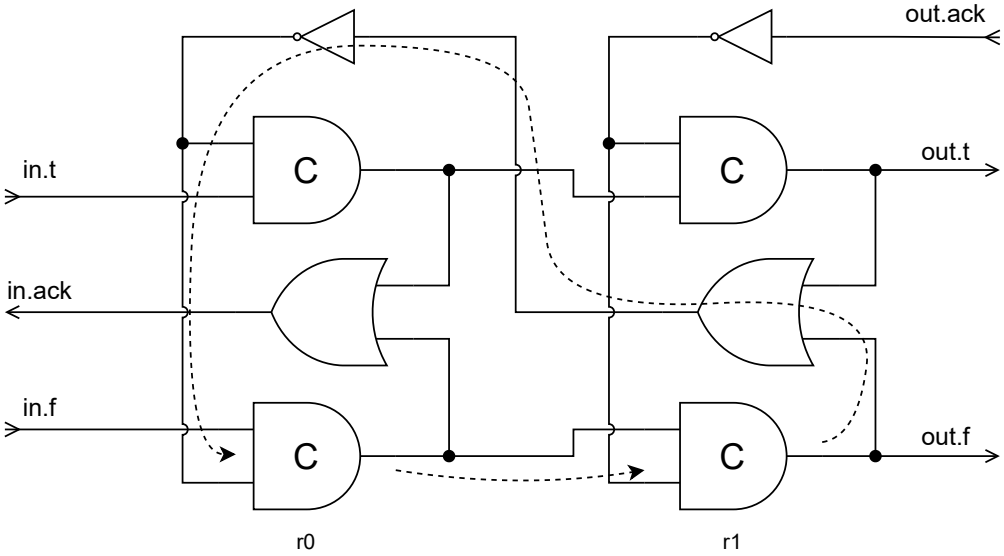
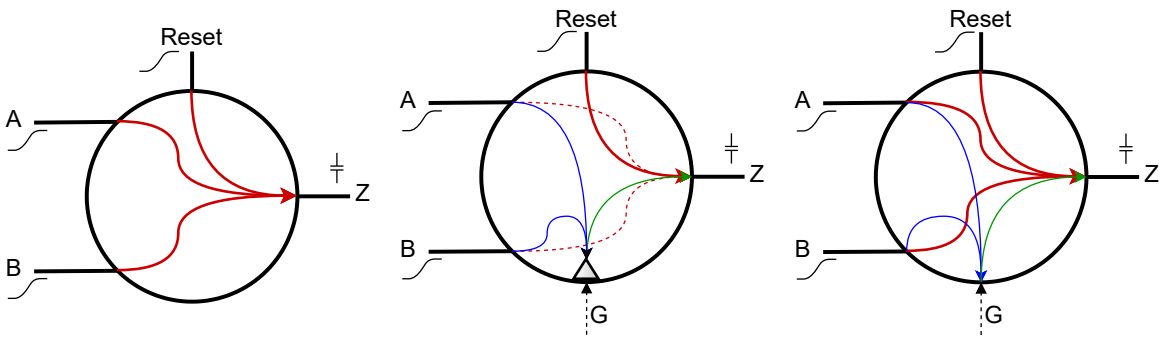


Figure 4.1: WCHB pipeline with dashed arrows highlighting the segments of one of the combinational loops as broken by the Pseudo-Synchronous WCHB technique. The reset signal is omitted for simplicity.

The C-element employed in the WCHB pipeline is naturally analogue to edge-triggered flip-flops employed in conventional asynchronous circuits as state-holding components. Pseudo-synchronous WCHB models this C-element as clocked sequential gates. Figure 4.2 depicts the original arcs and the two clocked sequential models proposed in [SWMC19]. The original C-element contains 3 propagation arcs to the output: two from its conventional inputs and one from the reset pin. Both the pseudo-flop and -latch models introduce a pseudo-clock pin G that is not physically present in the gate layout. The inclusion of the pseudo-clock pin G differs from [TBV12], where they use the reset pin as the pseudo-clock. The adoption of this additional pin enables preserving the characterised arcs related to the reset pin in both models.



(a) Original. (b) Pseudo-Flop. (c) Pseudo-Latch.

Figure 4.2: Arcs of the three C-element characterisation models. Red lines are the original propagation arcs, blue lines are the setup constraint arcs, and green lines are the propagation arcs from the pseudo-clock G. Adapted from [SWMC19].

The pseudo-flop model breaks the propagation arcs between the C-element's inputs and output into two arcs related to the pseudo-clock G: a propagation arc and a setup constraint arc. The new propagation arc from G to the output is solely dependent on the output capacitance, and the setup arc is solely dependent on the input slew. Experimental results show an error of less than 1% between the sum of the pseudo setup and propagation arcs and the initially characterised arcs. This indicates independence between the influence of input slew and output capacitance in the propagation delay, as expected from a circuit with 2 logic levels like a C-Element. Furthermore, this decoupling also means that the input slew hardly affects the output slew on C-Elements, weakening the justification for an STA tool dedicated to asynchronous circuits presented in [HLPM20].

The pseudo-flop model is used during synthesis, optimisation and sign-off to enable the STA tool to the linear propagation path delays. However, it removes the original propagation arc between the input and the output. This removal hinders the use of the pseudo-flop for delay-annotation. The pseudo-latch model presents the same pins and introduces the same propagation and setup arcs related to G as the pseudo-flop model, thus allowing it to replace the pseudo-flop model seamlessly by just replacing the liberty file. The pseudo-latch model takes advantage of the transparency mode present in latches to preserve the originally characterised arcs. When a transparent latch propagates changes from the inputs to the output, modelled as propagation arcs. These propagation arcs are annotated in the SDF file used in delay annotated simulations; these properties make the pseudo-latch model suitable for delay annotation and post-synthesis simulations.

It is worth noting that although the technique is presented here only for resettable C-Elements, it can be applied to any gate, hysteretic or not.

4.2 The Half-Buffer Channel Network

As discussed before, an asynchronous circuit's performance depends on its worst cycle time. However, this is not evident from the individual propagation path delays due to complex interactions between cycles. Finding the cycle times in a circuit efficiently requires models that capture its dynamic behaviour without having to resort to simulation.

Beerel's *Full-Buffer Channel Network* (FBCN) [BOF10], depicted in Figure 4.3, is one such model. It is a Petri net that captures the propagation and acknowledgement of data in a handshake circuit. It models each handshake channel with two places, one corresponding to the data propagation path and the other to the acknowledgement propagation path.

As the name states, the FBCN models full-buffer handshaking circuits, i.e. circuits that allow two data tokens to coexist in successive pipeline states; this matches the behaviour of 2-phase handshake circuits. However, since it models full-buffer circuits, the

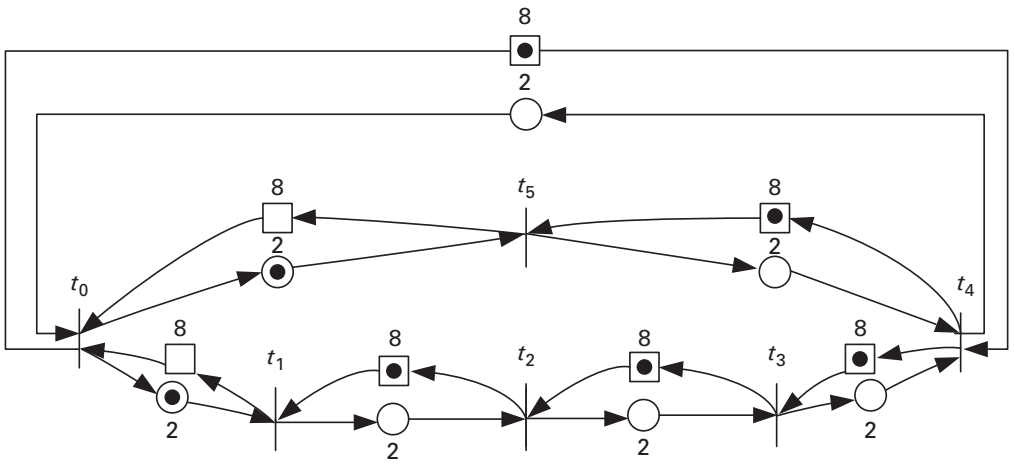


Figure 4.3: Example of Full-Buffer Channel Network (FBCN), adapted from [BOF10]. The square places model the acknowledgement propagation paths, the round places represent the data propagation paths, and the transitions models handshaking entities such as registers. The numbers associated with the places are the path delays, and the black dots in places are the initial markings.

FBCN fails to capture the behaviour of 4-phase handshaking circuits. A 4-phase handshake protocol interleaves the propagation of data with the propagation of spacers. This means that handshaking circuits implementing a 4-phase protocol are half-buffers, i.e. circuits that require data tokens to be interleaved with spacers in successive pipeline states.

To correctly model handshaking circuits implementing a 4-phase protocol, the author of this thesis proposes the *Half-Buffer Channel Network* (HBCN) model. It employs four places per handshake channel, each corresponding to a phase of a 4-phase handshake protocol. The formal definition of the HBCN presented here is adapted from the author's MSc thesis original definition [Sar19].

Definition 30 (Half-Buffer Channel Network)**.** *A Half-Buffer Channel Network (HBCN) is a 4-tuple* $N = \langle T, \Gamma, C, i_0 \rangle$*, where* T *is a set of transitions,* $\Gamma \subseteq T \times T$ *is a set of transition pairs,* C *is a set of channels defined as* $C \subseteq \{\langle u, v \rangle : u, v \in \Gamma, u \neq v\}$*, and* $i_0 : C \mapsto \{ack_{null}, req_{data}, ack_{data}, req_{null}\}$ *is a function defining each channel's initial state. In any valid HBCN the following predicates always hold:*

$$\forall t, t' : \langle t, t' \rangle \in \Gamma \implies \langle t', t \rangle \notin \Gamma \tag{4.1}$$

$$\forall x \neq t' : \langle t, t' \rangle \in \Gamma \implies \langle t, x \rangle \notin \Gamma \tag{4.2}$$

$$\forall x \neq t : \langle t, t' \rangle \in \Gamma \implies \langle x, t' \rangle \notin \Gamma \tag{4.3}$$

It is useful to interpret this definition, explaining its predicates. Predicate 4.1 states that any two transitions t *and* t' *are in at most one transition pair in* Γ*, while the Predicates 4.2 and 4.3 state that every transition* t *is paired to a single other transition* t'*.*

Every HBCN $N = \langle T, \Gamma, C, i_0 \rangle$ *has a* equivalent marked graph *defined as the 3-tuple* $G = \langle T, P, M_0 \rangle$ *where* T *is a set of transitions of the HBCN,* $P \subseteq T \times T$ *is a set of places, and* $M_0 \subset P$ *is a set of initially marked places. For simplicity, the equivalent marked graph of an HBCN is referred to as HBCN. The relationship between N and G is expressed by six predicates:*

$$\langle\langle a, a' \rangle, \langle b, b' \rangle\rangle \in C \implies \{\langle a, b \rangle, \langle b, a' \rangle, \langle a', b' \rangle, \langle b', a \rangle\} \subseteq P \tag{4.4}$$

$$\langle\langle a, a' \rangle, \langle b, b' \rangle\rangle \in C \implies |\{\langle a, b \rangle, \langle b, a' \rangle, \langle a', b' \rangle, \langle b', a \rangle\} \cap M_0| = 1 \tag{4.5}$$

$$i_0(\langle\langle a, a' \rangle, \langle b, b' \rangle\rangle) = ack_{null} \implies \langle b', a \rangle \in M_0 \tag{4.6}$$

$$i_0(\langle\langle a, a' \rangle, \langle b, b' \rangle\rangle) = req_{data} \implies \langle a, b \rangle \in M_0 \tag{4.7}$$

$$i_0(\langle\langle a, a' \rangle, \langle b, b' \rangle\rangle) = ack_{data} \implies \langle b, a' \rangle \in M_0 \tag{4.8}$$

$$i_0(\langle\langle a, a' \rangle, \langle b, b' \rangle\rangle) = req_{null} \implies \langle a', b' \rangle \in M_0 \tag{4.9}$$

Predicate 4.4 states that each channel is expanded to four places, two forward propagation places (for data and spacer) and two cross-connected backward propagation places; Predicate 4.5 states that for each channel, one and only one place is initially marked; whilst Predicates 4.6 to 4.9 define which places are initially marked according to the corresponding channel respective initial state.

Note that $\{ack_{null}, req_{data}, ack_{data}, req_{null}\}$ is the set of states any channel is at any moment. In this sense, a channel comprises four places, and precisely one carries a token, making its current state explicit. This is obviously related to the phases of a 4-phase communication protocol.

Definition 30 enables deriving some useful functions:

$$m_0(u, v) = \begin{cases} 1, & \text{if } \langle u, v \rangle \in M_0 \\ 0, & \text{if } \langle u, v \rangle \notin M_0 \end{cases} \tag{4.10}$$

$$pair(t) = \begin{cases} y', & \text{if } \langle t, y' \rangle \in \Gamma \\ y, & \text{if } \langle y, t \rangle \in \Gamma \end{cases} \tag{4.11}$$

The function $m0$ returns 1 if the place (u, v) contains a token at the initial time; otherwise, it returns 0. Function *pair* returns the transition paired to the argument. This function takes advantage of Predicates 4.1 to 4.3 from Definition 30, which guarantees there is just a single transition pair satisfying the guard condition (t in the function statement).

Figure 4.4 illustrates the above concepts, showing the HBCN for a simple 3-stage half buffer pipeline. Each register or handshake port is modelled by a transition-pair[1] ($\gamma \in \Gamma$) capturing the propagation of tokens and spacers in the circuit, e.g. the pair $(r0, r0')$. Hand-

[1] The elements of *Gamma* are the stacked blue-red transition pairs.

shake channels are each modelled by four places[2], each representing a handshake phase, e.g. places $\{(r0, r1), (r1, r0'), (r0', r1'), (r1', r0)\}$ model the channel between the transition pairs $(r0, r0')$ and $(r1, r1')$. The initial state of the HBCN defines its initial marking set, where places $\{(r0', in), (r1', r0), (out', r1)\}$ hold the initial tokens, indicating that the corresponding channels are in the ack_{null} state. It is worth noting that the first transition t in a transition pair (t, t') represents the propagation of valid data tokens in the circuit, whilst the second transition t' represents the propagation of a spacer.

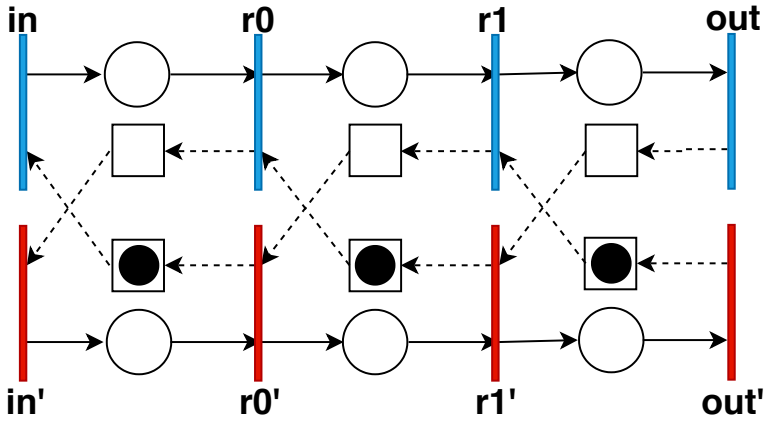


Figure 4.4: Example HBCN, extracted from [SWMC19].

4.3 Timing Analysis

The use of HBCN as a timing model arises when assigning timing information to its equivalent marked graph. It is helpful to define timed marked graphs precisely.

Definition 31 (Timed Marked Graph)**.** *A* timed marked graph*, defined as a quintuple* $G_t = \langle T, P, M_0, d, a \rangle$*, is a marked graph* $G = \langle T, P, M_0 \rangle$*, where* $d : P \mapsto \{x : x \in \mathbb{R}, x > 0\}$ *is a function defining the delay of each place, and* $a : P \mapsto \{x : x \in \mathbb{R}, x \geq 0\}$ *is a function defining the* arrival time *of each transition. In a timed marked graph, the following predicate always holds.*

$$\forall \langle u, v \rangle \in P : \langle u, v \rangle \notin M_0 \implies a(v) \geq a(u) + d(\langle u, v \rangle) \tag{4.12}$$

The predicate in Definition 31 states that for every place not initially marked, the transition succeeding the place must fire no earlier than the moment when the transition preceding it fires plus the place delay. The arrival time is the first time a transition fires with regard to a start time reference (noted as time 0). Notice that the firing times of a transition are cyclical; it is possible for a place to fire multiple times, including before the start time

[2]In practice, delays are associated to these places, but Definition 30 does not include delays. Definition 31 complements HBCNs to include delay modeling.

0. The initial marking indicates the state of the marked graph at the start time reference 0. Also, notice that it is possible for a transition to fire at time 0 if all its preceding places are initially marked.

The timed marked graph of an HBCN provides a framework to enable the timing analysis of circuits employing the pseudo-synchronous model. The place delays capture the worst delays of propagation paths; these are annotated with the delays extracted from STA. Assuming that the circuit employs the RTZ protocol, for every transition pair $(t, t') \in \Gamma$ corresponding to a register, the transition corresponding to data propagation (t) is related to a rise transition in the output of one of the pseudo-synchronous C-elements in the register. Similarly, the transition corresponding to the spacer propagation (t') is related to a fall transition in the output of one of the C-elements. For each channel $(\langle a, a' \rangle, \langle b, b' \rangle) \in C$, between the handshaking register or port a and b, the place delays are adjusted from STA accordingly:

- Delay $d(\langle a, b \rangle)$ is the worst propagation time passing through a rise on any output of a and a rise on any input of b.
- Delay $d(\langle b, a' \rangle)$ is the worst propagation time passing through a rise on any output of b and a fall on any input of a.
- Delay $d(\langle a', b' \rangle)$ is the worst propagation time passing through a fall on any output of a and a fall on any input of b.
- Delay $d(\langle b', a \rangle)$ is the worst propagation time passing through a fall on any output of b and a rise on any input of a.

When using the pseudo-synchronous technique, the start point of any propagation path passing through registers is the pseudo-clock. This means that when annotating the HBCN with delays, the capacitance-dependent propagation delay of the start point register is included in the path propagation delay, as is the slew-dependent setup constraint of the endpoint register.

The delay annotation on a timed marked graph is not sufficient to fully describe the timing characteristics of the circuit. For any given set of place delays, there are multiple possible timed marked graphs, each with its own set of arrival times. For the purpose of computing the *worst cycle time*, the relevant timed marked graph is the one where the transitions fire as early as possible, limited only by the place delays. Maggot's algorithm [Mag84] can compute this exact timed marked graph and consequently the worst cycle time. It employs linear programming to find a minimal cycle time value θ that satisfies the constraints:

$$\forall(\langle u, v \rangle \in P) : a(v) \geq a(u) + d(\langle u, v \rangle) - \theta \cdot m_0((u, v)) \tag{4.13}$$

The rationale of this algorithm is similar to predicates in Definition 31, the arrival time for the transition succeeding a place $a(v)$ must be greater than the arrival time for the transition

preceding it $a(u0)$ plus the place delay $d(\langle u, v\rangle)$; unless the place is initially marked, then it is assumed that the transition preceding the place has fired one cycle time θ before. This exploits the fact that the firing interval of transitions is cyclical, and the arrival time only captures the first firing after the arbitrary initial time 0, but nothing hinders the transition to have fired before the initial time 0.

The worst cycle time limits the performance of the entire circuit. However, it is not the only significant performance metric when performing timing analysis. During operation, the slowest cycle dominates faster cycles, making tokens wait longer in a place than their place delays require; this slowest cycle is called the *most critical cycle*. The time a token waste in a place is called the *slack time*, i.e. the time a place holds a token after its delay has elapsed waiting for the transition to fire. Computing the slack time enables finding the most critical cycle limiting the performance of a circuit. The function $s : P \mapsto \{x \in \mathbb{R} : x \geq 0\}$ defines the slack time of places on the timed marked graph:

$$s(\langle u, v\rangle) = a(v) - d(\langle u, v\rangle) - a(u) \tag{4.14}$$

A critical cycle is a path with the least slack travelled by a token. For each initial marking, a shortest path algorithm like a Dijkstra is able to compute the path with the least slack time between the transition succeeding the marked place and the transition proceeding it. Using this method, the *most critical cycle* in a circuit is the critical cycle with the least slack, and the *least critical cycle* is the critical cycle with the most slack. This information can produce timing reports that aid a designer in focusing their architectural optimisation efforts.

4.4 Cycle Time Constraining

Using the timed marked graph, it is also possible to compute the maximum delay propagation paths can have to respect a worst cycle time constraint. To find a timed marked graph that adequately constrains a circuit's cycle time, Sartori et al. [SWMC19] proposed using a variation of Maggot's algorithm [Mag84] to discover a (single) pseudo-clock delay. In [SMC20a], the authors extended this technique to relax paths that present slack time. The proposed techniques employ linear programming to find a maximum delay λ that satisfies the cycle time θ subject to:

$$\forall(p \in P) : d(p) \geq \lambda \tag{4.15}$$

$$\forall((u, v) \in P), d((u, v)) = d((pair(u), pair(v))) \tag{4.16}$$

$$\forall((u, v) \in P) : a(v) = a(u) + d((u, v)) - \theta.m_0((u, v)) \tag{4.17}$$

Constraint 4.15 states that every place in the timed marked graph is at least as slow as the maximum delay λ. Constraint 4.16 ensures that places capturing the same propagation path are constrained to the same delay. Constraint 4.17 states that for every place, the arrival time of the transition succeeding the place is equal to the arrival time of the preceding transition plus the place delay, except when the place is initially marked. In this last case, it assumes that the preceding transition has fired one cycle time of θ before. The technique proposed in [SWMC19] uses the maximum delay λ as a pseudo-clock during synthesis to constrain all propagation paths. The intuition here is that if every propagation path in the critical cycle[3] is at most as slow as λ, the maximum cycle time is bounded to θ. But according to Constraint 4.15, some delays may be greater than the pseudo-clock period λ if they present free-slack [BOF10]. In these cases, a timing exception is applied to these paths using techniques such as the `set_max_delay` SDC [BC09] command. Also, albeit not covered here for simplicity, [SMC20a] presents a limited number of cases where place delays can be safely set to a value lower than the pseudo-clock period for optimisation.

4.4.1 Proportional Cycle Time Constraints

The approach used in Pulsar [SWMC19,SMC20a] assigns the same constraint λ to every propagation path regardless of their complexity. Nonetheless, individual propagation paths broken by the pseudo-synchronous model naturally produce paths with distinct depths and complexities, e.g. the forward and backward propagation paths of a WCHB pipeline stage with different functional units may present varying logic depths. Albeit this approach constrains the worst cycle, it does so at a non-optimal cost, as it potentially over-constrains some paths whilst leaving others slacked.

A better approach would be to consider the complexity of each propagation path in the circuit during constraining. This paper introduces a new HBCN-based technique that can constrain each path proportionally to its logical depth. Constraining the propagation paths requires a fair estimation of their cost. For this reason, this work introduces an HBCN extension that can model the cost of propagation paths, as captured in the following definition.

Definition 32 (Weighted HBCN)**.** *A weighted HBCN, is completely defined by the 6-tuple given by $N_w = \langle T, \Gamma, C, i_0, w_f, w_b \rangle$, is an HBCN $N = \langle T, \Gamma, C, i_0 \rangle$, where $w_f : C \mapsto \mathbb{R}$. It is a function defining each channel forward propagation cost, and $w_b : C \mapsto \mathbb{R}$ is a function defining each channel's backward propagation cost.*

Every weighted HBCN $N_w = (T, \Gamma, C, i_0, w_f, w_b)$ has a characteristic weighted marked graph *defined by the 4-tuple $G_w = (T, P, M_0, w)$ that is a characteristic graph $G = \langle T, P, M_0 \rangle$, where $w : P \mapsto \mathbb{R}$ is a function defining the cost of places. The relationship between N_w and*

[3] A *critical cycle* is the cycle with the largest delay.

G_w implies all predicates defined for the relation between N and G, and also the following four additional predicates.

$$(\langle a,b\rangle,\langle c,d\rangle) \in C \implies w(\langle a,c\rangle) = w_f(\langle a,b\rangle,\langle c,d\rangle) \tag{4.18}$$

$$(\langle a,b\rangle,\langle c,d\rangle) \in C \implies w(\langle b,d\rangle) = w_f(\langle a,b\rangle,\langle c,d\rangle) \tag{4.19}$$

$$(\langle a,b\rangle,\langle c,d\rangle) \in C \implies w(\langle c,b\rangle) = w_b(\langle a,b\rangle,\langle c,d\rangle) \tag{4.20}$$

$$(\langle a,b\rangle,\langle c,d\rangle) \in C \implies w(\langle d,a\rangle) = w_b(\langle a,b\rangle,\langle c,d\rangle) \tag{4.21}$$

Interpreting the Definition, Predicates 4.18 and 4.19 state that the cost of a place taking part in a forward propagation path is the same as the forward propagation cost of its constituent channel. Predicates 4.20 and 4.21 state that the cost of a place taking part in a backward propagation path is the same as the backward propagation cost of its constituent channel.

The virtual delays introduced in [SMC20a] can be used as costs in the weighted HBCN. The cost of the forward propagation path can then be extracted from Pulsar's pre-synthesis virtual delay. For the backward propagation cost, the virtual delays can be estimated from the fanout on the forward path, i.e. the number of channels driven by the register or port. The backward propagation path virtual delay estimation assumes that 2-input C-elements implement the completion detection tree. It is also assumed that an OR gate performs the completion detection, and the register is implemented using a C-Element. The maximum depth of a binary tree with n leaves is $\lceil \log_2 n \rceil$. Also, according to the virtual delay definition in [SMC20a]: a C-element has 10 ps of virtual-delay, and an OR gate has 5 ps of virtual delay. Therefore, the virtual delay of any backward propagation path arriving at a register or port can be defined by $15+10\lceil \log_2 n \rceil$, where n is the number of channels starting at the register or port.

To derive the timed marked graph constraining the weighted HBCN, the author proposed a linear programming (LP) formulation derived from Magott's Algorithm [Mag84]. Here, a *balloon factor* τ correlates the place delays and their costs. An LP solver is used to find the maximum τ that satisfies the chosen cycle time θ, subject to the constraints:

$$\forall \langle u,v\rangle \in P : d((u,v)) = d(\langle pair(u), pair(v)\rangle) \tag{4.22}$$

$$\forall \langle u,v\rangle \in P : a(v) = a(u) + d(\langle u,v\rangle) - \theta.m_0(u,v) \tag{4.23}$$

$$\forall p \in P : d(p) \geq \tau.w(p) \tag{4.24}$$

Constraint 4.22 ensures that places capturing the same propagation path are constrained to the same delay. Constraint 4.23 states that for all places, the arrival time of the transition succeeding the place is equal to the arrival time of the of preceding transition plus the place delay, except when the place is initially marked. In this last case, it is assumed that the preceding place has fired one cycle time θ before. Constraint 4.24 sets the minimal place

delay proportional to its weight. The intuition behind this formulation is that the sum of the place delays taking part in a cycle is equal to θ. The balloon τ inflates the place delays proportionally to their costs. The place delay grows to accommodate the ballooned place cost $\tau.w(p)$.

The solution of this formulation is used to create timing constraints during the synthesis. To constrain the circuit to the cycle time θ, the paths captured by place delays are constrained using the `set_maximum_delay` SDC command.

5. PULSAR AND ITS TEMPLATES

To automate the synthesis of ST circuits, the Author has proposed Pulsar [SWMC19, Sar19, SMC20a]. Figure 5.1 gives an overview of the Pulsar system. It comprises a set of scripts and related tools to leverage the use of conventional EDA to synthesise asynchronous circuits; an open public version of Pulsar is available at [SMC20c]. Like the Uncle [RST12] tool, Pulsar performs two synthesis 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 throughout the entire synthesis process. This comprehensive use of commercial EDA tools allows Pulsar to employ timing-driven synthesis to trade off performance, power and area goals. However, since the performance of handshaking circuits is not straightforward, Pulsar proposes the use of the half-buffer channel network (HBCN) timing model (Refer to Definition 30 in Chapter 4) to derive timing constraints for individual propagation paths, based on a chosen a cycle time constraint.

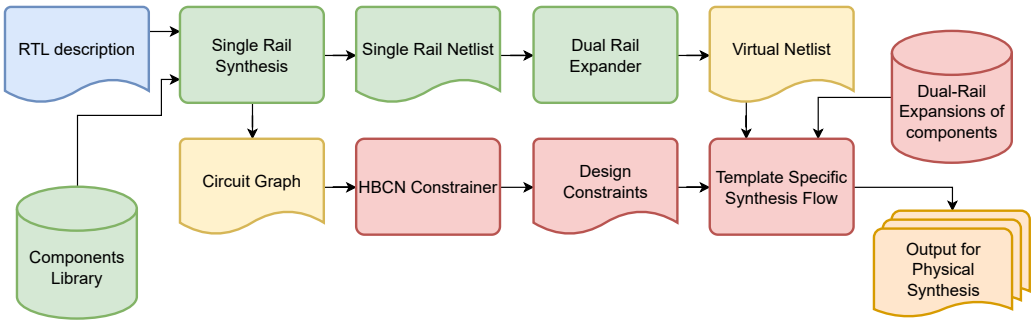


Figure 5.1: Overview of Pulsar: green boxes are part of the front end, red boxes are part of the back end, light-yellow boxes are the main intermediate files, blue is the user input and orange is the output.

Pulsar's front end uses Genus to synthesise the RTL-Like description to an intermediate 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. 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 this single-rail netlist to produce a *virtual netlist*. This tool replaces all wires on the netlist with instances of a *SystemVerilog interface*[1] that models handshake channels between components; the implementation of this interface is also delegated to the back end. The resulting virtual netlist comprises components connected by handshake channels. The front end also produces a

[1] 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.

high-level graph describing the handshaking relations between primary inputs, outputs and sequential components; a custom tool processes this graph to generate the circuit's HBCN.

The implementation of the handshake channel SystemVerilog interface is dependent on the employed handshake communication protocol; it comprises the data and control wires required to perform channel handshakes. This SystemVerilog interface contains two *modports*[2] to connect consumer and producer component modules. The SystemVerilog modules implementing components contain the necessary logic to interact with the handshaking channels and to implement its behaviour. The construction of ST circuits requires that the protocol, sequential and combinational components work in tandem. There are multiple ways of implementing any of these parts, and the working combination of them is called an *asynchronous design template* or template for short. Pulsar's back end supports synthesising ST circuits using different templates by changing the components and dual-rail channel interface implementations. A custom tool derives propagation time constraints from the HBCN to constrain the circuit cycle time. Pulsar uses Genus to perform timing-constrained synthesis of the virtual netlist together with the components and handshake channel implementations to produce the final ST circuit netlist. The gates employed in this final 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 Chapter first introduces Pulsar's RTL-like design capture process in Section 5.1. Next, Section 5.2 discusses how Pulsar produces virtual netlists, as well as the proposal of a set of components that enable the seamless construction of ST circuits. Finally, the last three sections of the Chapter explore the ST templates currently available to use with Pulsar. These are presented in chronological order of their proposition.

5.1 Design Capture

Pulsar's RTL-like descriptions assume the use of a positive-edge clock and an active-low asynchronous global reset signal. As a convention imposed by Pulsar, the positive-edge clock is always named `clk` and the active-low global reset is always named `reset`. The clock of the RTL-like description is not synthesised in the final circuit; it is merely used to guide the EDA tool synthesis process. This RTL-like description assumes the designer will employ constructions usually seen in flop-based RTL designs e.g. D-type, edge-triggered flip-flops. However, the descriptions have different behaviour in the final asynchronous circuit. A D-type Flip-Flop in a synchronous design latches data on clock edge events. Here, since the clock is non-existent, the description implies an instantiation of components that

[2] In SystemVerilog, a *modport* defines how modules interact with an interface, e.g. it defines signal directions for modules with different roles in a bus.

perform handshakes with neighbouring components. This distinction means that the behaviour of the synthesised circuit does not match the behaviour of a corresponding synchronous circuit synthesised from the same description, nor does it matches the behaviour of some conventional RTL simulation. There are three types of D-type flip-flops that are translated to components from a Pulsar RTL-like description: (i) flip-flops without reset translate to half-buffer components; (ii) flip-flops with asynchronous preset or reset translate to full-buffer components (form more on half-buffers and full-buffers, refer to Section 3.2); and (iii) special control-flow components which are explicitly instantiated in the source code[3]. The actual implementation of these components is template specific.

A circuit implementing a 4-stage 32-bit accumulator is used here to demonstrate the code-interpretative distinctions. This circuit's intended behaviour is to sum the values inserted in its input `in`, producing results at the output `out` after each insertion. The Pulsar RTL-like input description of the accumulator is depicted in Listing 5.1. Lines 13-15 infer full-buffer components; these are responsible for placing initial tokens in the circuit accumulator. Lines 7-11 infer half-buffer components: Line 8 infers the input buffer; Line 9 infers a buffered adder; and Line 10 infers the output buffer. This description is subject to retiming; although the adder here is placed before a half-buffer, it is spread around the buffer during synthesis.

Listing 5.1: RTL-like, Pulsar Verilog input description of the 4-stage accumulator.

```
1  module acc #(WIDTH=32)
2     (input  logic             clk, reset,
3      input  logic [WIDTH-1:0] in,
4      output logic [WIDTH-1:0] out);
5     logic [WIDTH-1:0]         in_reg, sum, acc;
6
7     always @(posedge clk) begin
8        in_reg <= in;
9        sum <= acc + in_reg;
10       out <= sum;
11    end
12
13    always @(posedge clk or negedge reset)
14      if (!reset) acc <= '0;
15      else acc <= sum;
16 endmodule
```

This RTL-like description corresponds to distinct behaviours on synchronous and asynchronous implementations. The registers on a synchronous circuit capture data on every relevant clock edge, despite of its presence or correctness. Meanwhile, registers in asynchronous circuits perform a handshake with their neighbouring stages to coordinate data capture; therefore, it only captures data when these are ready. The asynchronous behaviour is depicted in Figure 5.2. Here, it is possible to verify that values produced at

[3]The control-flow components are characterised as D-type flip-flops, but marked as "don't-use" in the library. Therefore, the EDA tool sees them as timing end- and start-points, but it is not able to instantiate such components to implement inferred flip-flops.

`out` are the sum of the previous values inserted at `in`. It is also possible to observe the propagation of data in the circuit. Spacers separate data values; they propagate in a wave-like way.

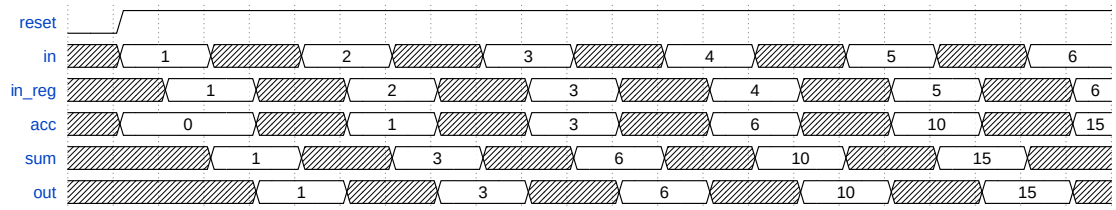


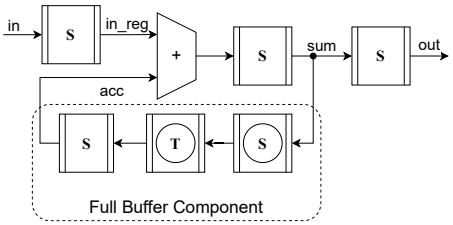
Figure 5.2: Waveform depicting the asynchronous behaviour of the RTL-like description in Listing 5.1. Greyed-out areas represent spacers.

The data propagation behaviour is better captured by token flow diagrams [Spa20]. In such diagrams, tokens carry abstract data. Tokens and spacers propagate from latch to latch through channels. A token (or spacer) is a bubble if all latches succeeding it contain an identical token (or spacer); otherwise, it is part of the wavefront. Tokens and spacers wavefronts are only allowed to propagate over bubbles; otherwise, information is potentially lost, leading to circuit failure.

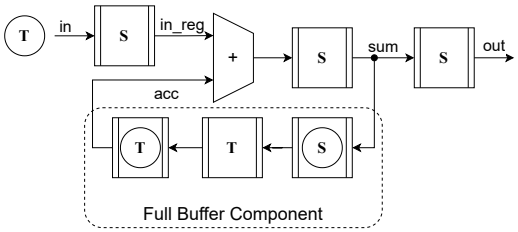
Channels can be merged or split. Combinational logic blocks merge channels; these combine tokens or spacers at their inputs. However, a token cannot combine with a spacer, nor a spacer with a token. Thus, a token (or spacer) can only propagate through a combinational logic block when all inputs of the logic block hold tokens (resp. spacers). Channels split when they feed into multiple latches; tokens (or spacers) are duplicated on channel splits.

The token flow diagram for the 4-stage accumulator is depicted in Figure 5.3, employing the notation available, e.g. in [Spa20]. Here it is possible to observe how the inferred registers on the RTL-like input description are expanded. The resetable D-type flops expand to a full-buffer component; a sequence of three latches implements this. The middle latch in a full-buffer component is initialised with a token carrying the reset value. The non-resetable D-type flip-flops expand to half-buffer components, each being implemented by a single latch. Unless stated otherwise, every latch defaults to initialising with a spacer. Spacers are responsible for resetting combinational blocks to a ready state accepting new computations. It is paramount to initialise latches neighbouring logic blocks with spacers during the initial reset phase; this adequately prepares combinational logic blocks for computation. This is also the reason a data token is placed between two spacers during the full-buffer component initialisation.

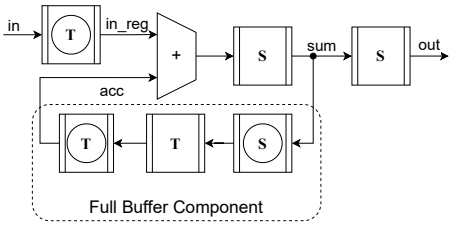
The synchronous behaviour of this description is depicted in Figure 5.4. Here, two possible faulty behaviours are depicted: (a) when the reset duration is shorter than the time it takes to propagate data across all non-resettable flops; (b) when the reset duration is long enough to propagate the initial computational value through all non-resettable flops. At reset time, `acc` is set to 1; on Figure 5.4(a), `sum` is left at an unknown state, whereas on



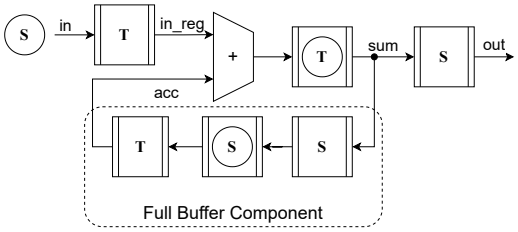
(a) At reset time: a token T containing the initial value is placed at the middle latch of the full-buffer component; the remaining latches are initialised with spacers S.



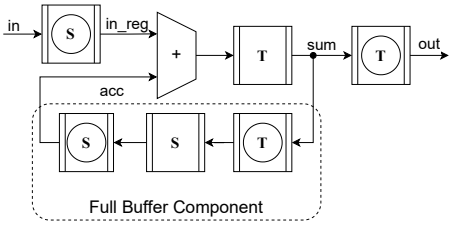
(b) After the reset is released: the token in the full buffer component propagates; a token arrives on the external input channel `in`.



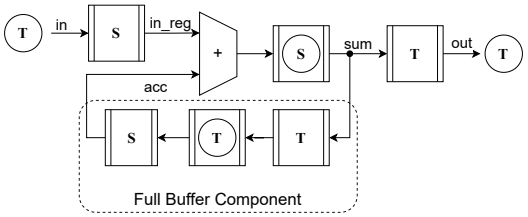
(c) The token placed at the input propagates to the input register, but the token at the full buffer stays in place because it cannot propagate to the next latch until all latches preceding the adder hold tokens.



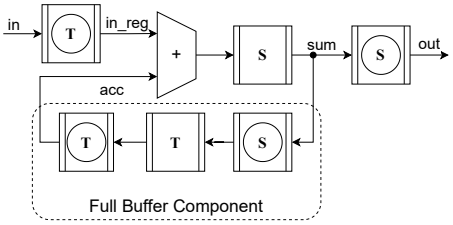
(d) After a token becomes available in `in_reg`, the adder merges it with the token at `acc` and they propagate to the next latch. The environment produces a new spacer, and the spacer inside the full buffer propagates.



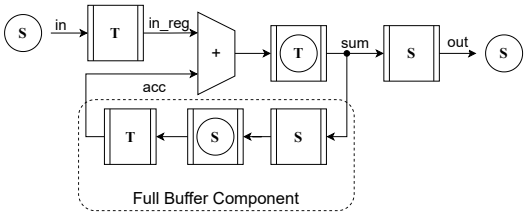
(e) The token at `sum` splits in two and is propagated to the output buffer and back to the first stage of the full buffer, whilst the spacer at the input propagates to the input buffer, and the spacer in the full buffer propagates to its last latch.



(f) The token containing the first result is consumed by the environment whilst it independently inserts a new token. Spacers are merged by the adder and propagate to the next latch; this prepares the adder for the next computation. Also, the token containing the last result propagates through the full buffer.



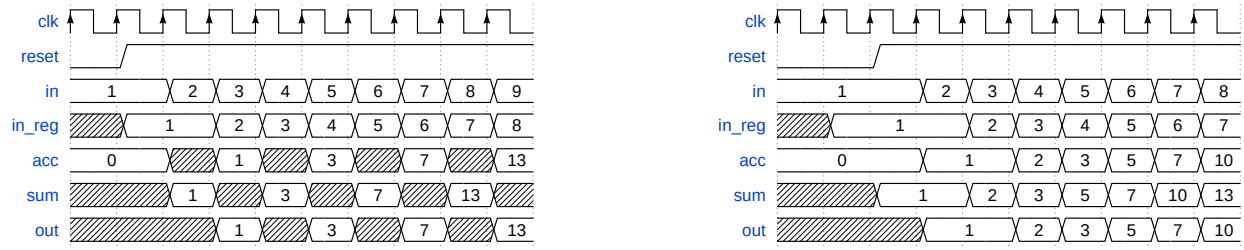
(g) The spacer splits and propagates to the next latches. The input buffer consumes the token from the environment.



(h) Tokens are merged by the adder and propagate to the next latch whilst the environment produces a new spacer on the input and consumes the spacer at the output. This process continues indefinitely from step (e) onward.

Figure 5.3: Simplified token-flow diagram for the 4-stage 32-bit accumulator. Parallel channels and latches are collapsed. *T* and *S* are respectively tokens and spacers; encircled values are wavefronts and plain values are bubbles. A token can only propagate over a spacer bubble and a spacer can only propagate over a token bubble.

Figure 5.4(b), the reset time is long enough to propagate the result of the first addition to `sum`. Both yield incorrect values; the former accumulates 1 with only the even values of the input sequence; the latter alternates between accumulating the even and the odd values. This faulty behaviour is due to the presence of a register in a closed loop; on synchronous circuits, this causes a 1-cycle delay.



(a) Faulty behaviour when reset is enabled (='0') for a single clock cycle.

(b) Faulty behaviour when reset is enabled (='0') for multiple clock cycles.

Figure 5.4: Waveform showing the synchronous behaviour of the RTL description in Listing 5.1. Here greyed-out areas are unknown values.

Besides the differences depicted in the previous example, another source of incompatibilities with standard RTL is how choice operations are captured in Pulsar's RTL-like descriptions. To demonstrate how such components are used in RTL-like descriptions, it is useful to take a look at the design of a 32-bit loadable up-counter. This circuit explicitly instantiates 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 counter value, otherwise it discards the current counter value and loads the one received. The token-flow diagram for this circuit is depicted in Figure 5.5.

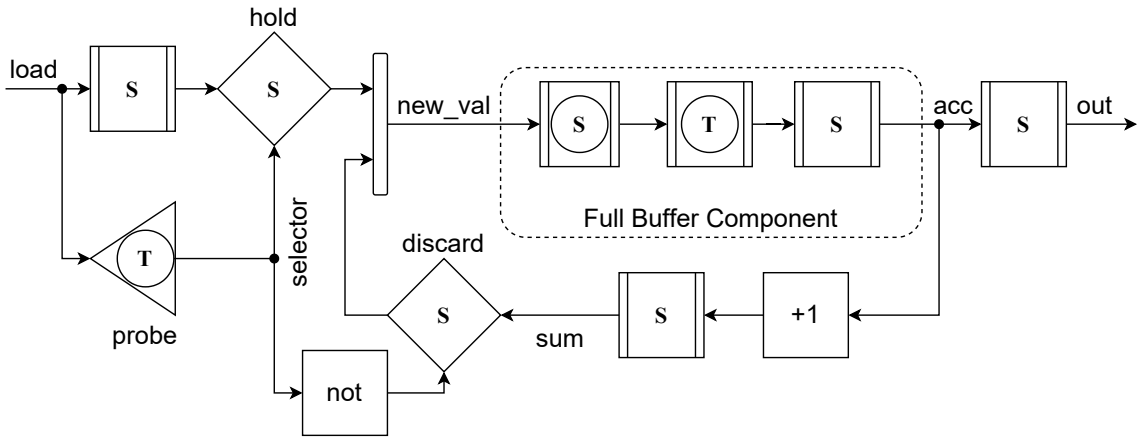


Figure 5.5: Simplified token-flow diagram for the loadable 32-bit up-counter.

Here, it is possible to observe that the discard component takes part in the accumulator loop. While a token does not arrive at the load channel, the probe produces false valued tokens interposed with spacers. These false value tokens inhibit the hold component from handshaking on its input and output data channels. If a token arrives at the load chan-

nel after the probe has issued a false valued token, it is held until the probe can acknowledge its presence.

Listing 5.2 presents the design of the counter. The token steering components instantiated in Lines 15-18 are in a fan-in steering arrangement; they multiplex the `new_val` channel from either `load_reg` or `sum`.

Listing 5.2: RTL-like description of a loadable 32-bit up-counter.

```
1  module counter #(WIDTH=32)
2     (input  wire                clk, reset,
3      input  logic [WIDTH-1:0] load,
4      output logic [WIDTH-1:0] out);
5     logic [WIDTH-1:0]          acc, sum, load_reg;
6     wire  [WIDTH-1:0]          new_val;
7     wire                       selector;
8
9     always @(posedge clk or negedge reset)
10      if (!reset) acc <= '0;
11      else acc <= new_val;
12
13    probe s (.a(|load), .q(selector), .*);
14
15    for (genvar i = 0; i < WIDTH ; i++) begin
16       hold load_h (.a(load_reg[i]), .en(selector), .q(new_val[i]), .*);
17       discard sum_d (.a(sum[i]), .en(!selector), .q(new_val[i]), .*);
18    end
19
20    always @(posedge clk) begin
21       load_reg <= load;
22       sum <= acc + 1;
23       out <= acc;
24    end
25
26 endmodule
```

The next Section details the mentioned and other special components used by Pulsar synthesis process.

5.2 Virtual Netlist and Components

Synthesising ST circuits from RTL-like descriptions using Pulsar involves two synthesis steps. The first is performed on the front end to generate an intermediate template-independent SystemVerilog representation called a *virtual netlist*. This virtual netlist is later processed using a template-dependent back end to synthesise a functional ST dual-rail netlist on a target technology node. This process of transforming the virtual netlist into an ST implementation employing dual-rail encoding is called dual-rail expansion. In Pulsar, it exploits features of the SystemVerilog language, such as interfaces, to model channels and produce the template-specific circuitry that implements and manipulates them.

The first step in the construction of a virtual netlist is the synthesis of a single-rail netlist. This netlist is constructed using components presented as gates to Genus in a *liberty* file. Each component presents a standardised interface and behaviour but no physical layout implementation. Instead, their implementation is template-dependent and left to the back end. A custom program bundled within Pulsar, named *DRExpander*, transforms the single-rail netlist into a SystemVerilog virtual netlist. In this transformation, the component names are preserved, but all wires are replaced with instances of the `drwire` interface. Each input and output of the single-rail netlist is replaced by three wires, two data-rail and one acknowledgement wire. The environment is expected to interact with the end circuit using the RTZ dual-rail protocol. Since the protocol implemented by `drwire` is template-specific, two modules, `drinput` and `droutput`, are placed between the internal `drwire` and the virtual-netlist ports. Just like components and the `drwire` interface, the `drinput` and `droutput` modules' implementation is template specific. The standardised interface allows replacing circuits implemented from the same virtual netlist to different templates implementations without changing the environment. It also allows reusing the same test environments to compare implementations of circuits using different templates derived from the same virtual netlist.

Table 5.1 depicts the components available on Pulsar's front end to construct the virtual netlist. There are two types of components, *combinational* and *sequential*; the latter comprise components capable of initiation handshake, and the former make the class of components transparent to the handshake process. Most components are inferred from RTL constructions, but those employed to support choice must be explicitly instantiated. Sections 5.2.3 and 5.2.4 adapted from [SNC23] cover these special components used for decision-making and token steering in detail.

Table 5.1: Pulsar components.

<table>
<tr><th>Type</th><th>Instantiation</th><th>Name</th><th>Presented as</th><th>Inputs</th><th>Output</th></tr>
<tr><td rowspan="5">Combinational</td><td rowspan="8">Inferred</td><td>inv</td><td>Inverter gate</td><td>a</td><td rowspan="5">y</td></tr>
<tr><td>nand</td><td>2-input NAND gate</td><td rowspan="4">a, b</td></tr>
<tr><td>nor</td><td>2-input NOR gate</td></tr>
<tr><td>xor</td><td>2-input XOR gate</td></tr>
<tr><td>mux</td><td>2-input multiplexer</td></tr>
<tr><td rowspan="6">Sequential</td><td>dff</td><td>Flip-flop</td><td>d, clk</td><td rowspan="6">q</td></tr>
<tr><td>dffr</td><td>Resettable flip-flop</td><td>d, clk, reset</td></tr>
<tr><td>dffs</td><td>Settable flip-flop</td><td>d, clk, set</td></tr>
<tr><td rowspan="4">Explicit</td><td>discard</td><td rowspan="4">Blackbox</td><td rowspan="2">d, sel, clk, reset</td></tr>
<tr><td>hold</td></tr>
<tr><td>arbiter</td><td>a, b, clk, reset</td></tr>
<tr><td>probe</td><td>a, clk, reset</td></tr>
</table>

5.2.1 A Case Study

To illustrate the generation of a virtual netlist, this section explores the synthesis of a 2-bit 2-stage adder, an example simple enough for the reader to follow through all the steps. The starting point for synthesis is the SystemVerilog RTL-like description depicted in Listing 5.3. The description must contain a clock port named `clk`. This clock is used to infer rise-sensitive flops that translate to sequential components. An active-low reset port named `reset` is used to infer full-buffer components.

Listing 5.3: Input RTL example.

```
1  module adder(a, b, out, clk);
2     input wire [1:0] a, b;
3     output reg [1:0] out;
4     input wire       clk;
5
6     always @(posedge clk)
7       out <= a + b;
8  endmodule // adder
```

The RTL-like code is synthesised using Genus with a nought period clock constraint and retiming. The output of the single-rail synthesis is the netlist depicted in Listing 5.4. It instantiates the components by name. Note the number of dff instances; there should be enough to have only a two-bit register. However, Genus retiming engine considered advantageous to break one of the flops in two to balance logic between the pipeline stages (see the code lines starting with **dff retime_**).

Listing 5.4: Single-rail netlist example, reordered for clarity.

```
1  module adder(a, b, out, clk);
2    input [1:0] a, b;
3    input clk;
4    output [1:0] out;
5    wire [1:0] a, b;
6    wire clk;
7    wire [1:0] out;
8    wire n_0, n_1, n_2, n_3, n_4, n_5;
9
10   xor2 g176__4296(.a (a[0]), .b (b[0]), .y (n_3));
11   nand2 g178__1474(.a (a[0]), .b (b[0]), .y (n_1));
12   dff out_reg_0_(.ck (clk), .d (n_3), .q (out[0]));
13   dff retime_s1_2_reg(.ck (clk), .d (n_1), .q (n_4));
14   inv g179(.a (a[1]), .y (n_0));
15   xor2 g177__3772(.a (n_0), .b (b[1]), .y (n_2));
16   dff retime_s1_1_reg(.ck (clk), .d (n_2), .q (n_5));
17   xor2 g172__8780(.a (n_4), .b (n_5), .y (out[1]));
18 endmodule
```

From the single-rail netlist, the *DRExpander* program constructs a virtual netlist. DRExpander parses the Verilog netlist and replaces every wire in the netlist by instances of the `drwire` interface. Buses of wires, such as `a`, `b`, and `out` are broken in multiple `drwire` instances.

DRExpander is also responsible for recreating the module ports. Each port of the single-rail netlist is expanded to three ports suffixed by `_t`, `_f` and `_ack`, which are respectively the true, false and acknowledgement wires of the channel. The DRExpander instantiate `drinput` and `droutputs` SystemVerilog modules to connect these ports to the internal `drwire` instances. This provides a consistent interface for the asynchronous module to connect with the external world.

If the single-rail netlist does not include a `reset` port, the DRExpander creates one and connects it to all sequential components in the virtual netlist. This is important to reset half-buffer registers. Remember that half-buffer registers are modelled as D-flip-fops with no reset. On the single-rail netlist, the `dff` instances do not contain a reset pin. However, their expanded module may require a reset to function correctly.

Finally, the virtual netlist contains a clock port `clk` that connects to all its sequential components. This pseudo-clock signal is left available to guide the synthesis in the back end flow. This signal, however, is not necessarily implemented in the final circuit; it is only present to guide EDA tools to perform the synthesis correctly.

Listing 5.5: Virtual netlist example, edited for clarity.

```
1  module adder
2     (a_t, a_f, a_ack,
3      b_t, b_f, b_ack,
4      out_t, out_f, out_ack,
5      clk, reset);
6     input [1:0] a_t, a_f;
7     output [1:0] a_ack;
8     input [1:0]  b_t, b_f;
9     output [1:0] b_ack;
10    output [1:0] out_t, out_f;
11    input [1:0]  out_ack;
12    input  clk, reset;
13
14    // Instances of drwire for internal channels
15    drwire n_0 ();
16    drwire n_1 ();
17    drwire n_2 ();
18    drwire n_3 ();
19    drwire n_4 ();
20    drwire n_5 ();
21    drwire a_0 ();
22    drwire a_1 ();
```

```
23     drwire b_0 ();
24     drwire b_1 ();
25     drwire out_0 ();
26     drwire out_1 ();
27
28     // Connections between external ports and drwire channels
29     drinput ia_0 (.t(a_t[0]), .f(a_f[0]), .ack(a_ack[0]), .drw(a_0));
30     drinput ia_1 (.t(a_t[1]), .f(a_f[1]), .ack(a_ack[1]), .drw(a_1));
31     drinput ib_0 (.t(b_t[0]), .f(b_f[0]), .ack(b_ack[0]), .drw(b_0));
32     drinput ib_1 (.t(b_t[1]), .f(b_f[1]), .ack(b_ack[1]), .drw(b_1));
33     droutput iout_0 (.t(out_t[0]), .f(out_f[0]), .ack(out_ack[0]), .drw(out_0));
34     droutput iout_1 (.t(out_t[1]), .f(out_f[1]), .ack(out_ack[1]), .drw(out_1));
35
36     // Component Instances
37     xor2 g176__4296 (.a(a_0), .b(b_0), .y(n_3));
38     nand2 g178__1474 (.a(a_0), .b(b_0), .y(n_1));
39     dff out_reg_0_ (.reset(reset), .ck(clk), .d(n_3), .q(out_0));
40     dff retime_s1_2_reg (.reset(reset), .ck(clk), .d(n_1), .q(n_4));
41     inv g179 (.a(a_1), .y(n_0));
42     xor2 g177__3772 (.a(n_0), .b(b_1), .y(n_2));
43     dff retime_s1_1_reg (.reset(reset), .ck(clk), .d(n_2), .q(n_5));
44     xor2 g172__8780 (.a(n_4), .b(n_5), .y(out_1));
45 endmodule // adder
```

5.2.2 Sequential Components

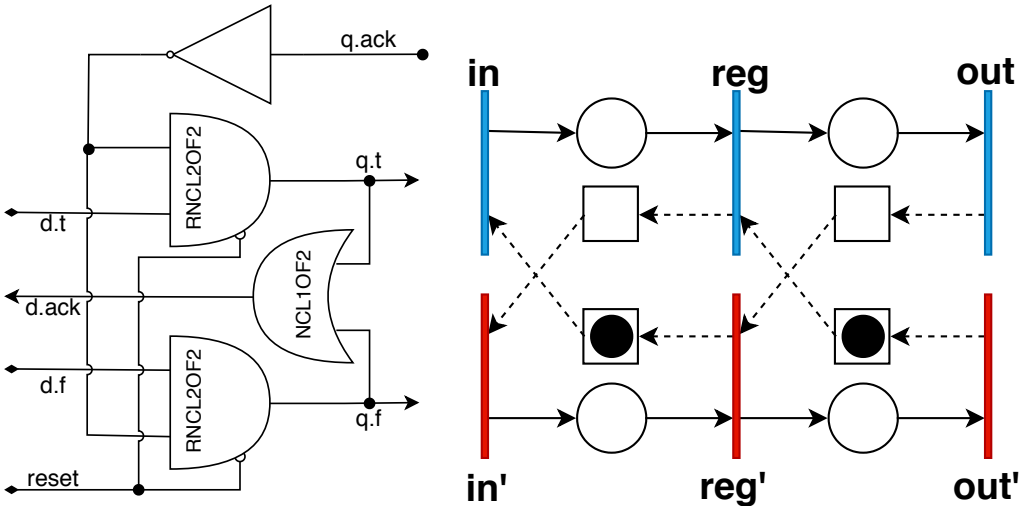
The sequential components implement registers in asynchronous pipelines implemented using Pulsar. They are modelled as flip-flops on the single-rail synthesis, allowing standard synthesis tools to instantiate them from canonical RTL constructions. The sequential components expand to template-dependent implementations in the back end synthesis. Sequential elements bind to `drwire` channels actively, performing handshakes 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` resettable full-buffer component; and (iii) the `dffs` settable full-buffer component. These are detailed next.

As an example, Figure 5.6(a) depicts an 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 resettable 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 resettable C-Elements can latch an incoming valid (null) token. This register operates on RTZ channels, valid token

codewords are one-hot[4] 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 raising the same signal. The inverter on the output ack wire enables the alternation between the reset and evaluation phases.

During initialisation, it is important to place the circuit in a known state. This is due to the fact that all components in the circuit consist of gates with hysteresis, which start at unknown initial states. An ST circuit with an unknown state may operate improperly, as it may start with an invalid encoding. 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 a null codeword everywhere.

For the above reason, half-buffer registers employ resettable C-Elements that initialise their output channels to null codewords. Null codewords propagate through combinational components in a 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, a piece of information which cascades in the backward propagation of the inbound channel, initialising it. This is evident when the HBCN model depicted in Figure 5.6(b) 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 signals that both channels are ready to accept new data tokens.



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

Figure 5.6: The `dff` half-buffer dual-rail register.

[4]Notice that a channel here captures a single bit, thus one-hot refers to either the true or false rails being enabled.

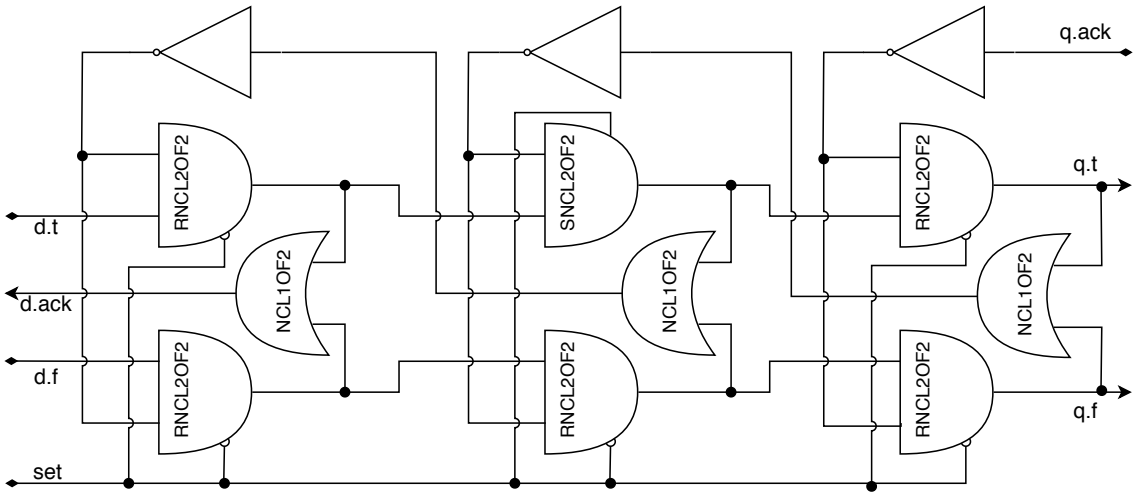
Sometimes it is necessary to initialise a circuit with data. For example, a counter must initialise to a known data value. This implies initialising some channels with valid tokens. This can be achieved using the full-buffer components depicted in Figure 5.7. The resettable full-buffer component, depicted in Figure 5.7(b), 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 token 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 start value.

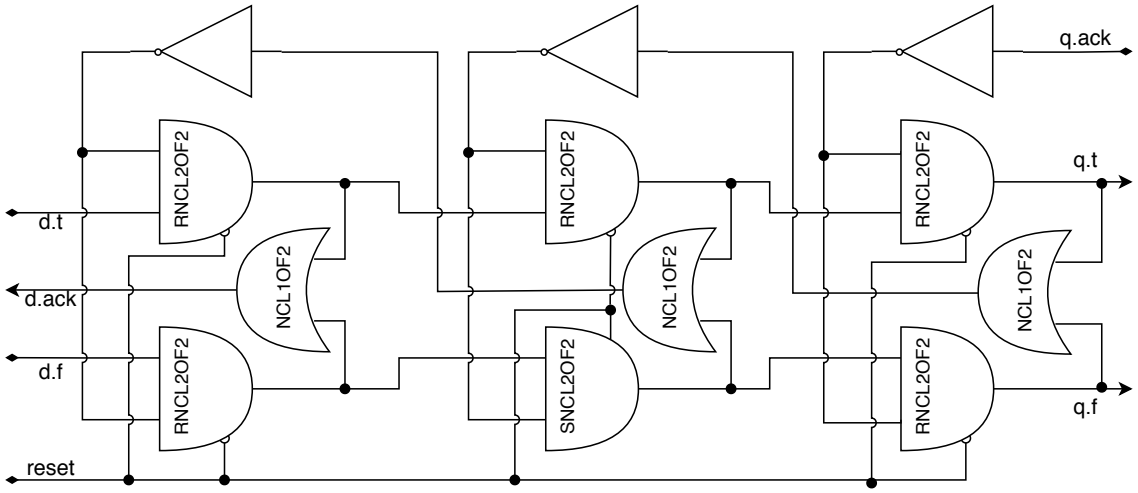
The behaviour of full-buffer components is further evidenced by an analysis of its HBCN model, depicted in Figure 5.7(c). Here, the inbound and outbound channels are initialised to a state where both are ready to accept new tokens, similar to the initial state of the half-buffer register. However, two channels internal to the component are initialised respectively to a data and a null token. These internal channels contain no logic; thus, they need not be initialised with a null token.

The settable and resettable C-elements employed in full-buffer components are pseudo-flop instances. As discussed in Section 4.1, a pseudo-flop breaks the cycles of WCHB pipelines to enable 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 module implementing the sequential component expansion. The implementation of sequential component expansions is in fact a technology-dependent step.

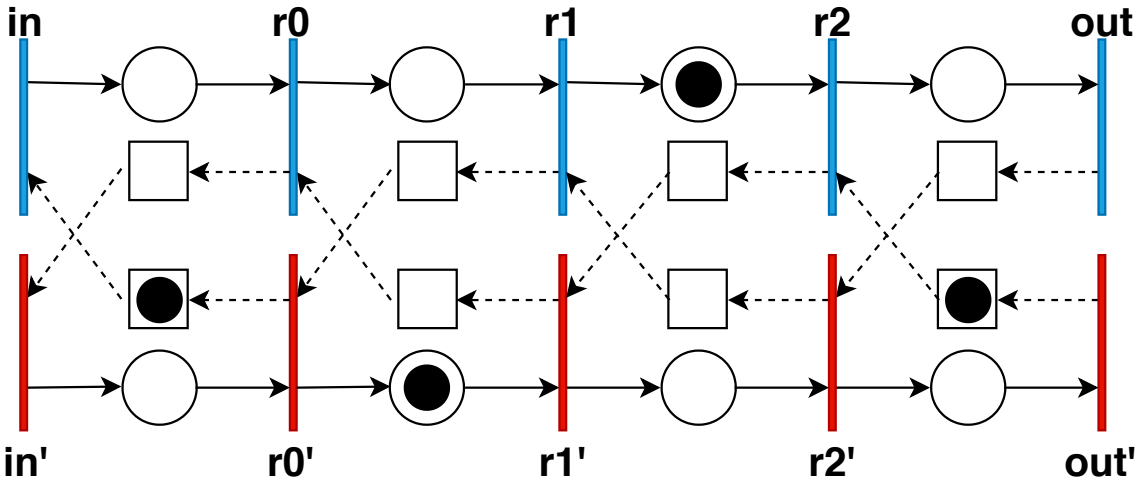
Just as in the case of combinational components, sequential ones are also instantiated by name during the single-rail synthesis. They are modelled as D-type flip-flops in the component library Liberty file. The settable and resettable full-buffer components are modelled as flops with preset and reset, 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 [RST12], where half-buffer registers are modelled as latches. According to the author's experience, Genus does not support retiming latch circuits. Therefore, modelling sequential components as flops enables performing retiming during single-rail synthesis. This balances the number of components employed in each pipeline stage and opens new opportunities for optimisation in early synthesis steps.



(a) The `dffs` settable component.



(b) The `dffr` resettable component.



(c) The full-buffer HBCN model.

Figure 5.7: The full-buffer components.

5.2.3 The Decision-Making Components

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

The dual-rail RTZ version of the two decision components uses a complex gate, the *handshaking mutex* (HM) sub-component detailed by the CMOS transistor network in Figure 5.8. The HM design proposition is one of the main original contributions of this work. It combines the functionality of a traditional mutex and resettable C-elements used in asynchronous ST circuits' temporal barriers. At reset, outputs (QA and QB) are both set low, placing the HM in a known state. Each output (QA and QB) of the HM is controlled by a corresponding asymmetric input pair (A+/A- and B+/B-), and by a common input (NACK). QA only rises when A+ and NACK are high, and QB is low. Similarly QB only rises when B+ and NACK are high, and QA is low. However, if both QA and QB rise simultaneously, only one of the outputs will rise after some (unbounded) arbitration time; this deliberation is ideally random. Furthermore, the condition for QA (QB) to fall is that both NACK and A- (B-) are low, regardless of the status of the other inputs and outputs.

Figure 5.9 depicts the complete implementation of the `probe` and `arbiter` decision components; they employ OR-gates as completion detectors (CDs), and an HM as arbitration and latching logic. Both `arbiter` and `probe` operate in arbitration cycles comprising an evaluation step and a reset step. The evaluation step occurs when a consumer element connected to the output channel signals its availability to receive a token. At this point, the decision component can generate a true or a false token to indicate the state of its input channels. The HM raises one of its outputs based on the state of the CD connected at its input. When NACK falls, this indicates that the consumer has absorbed the token and is

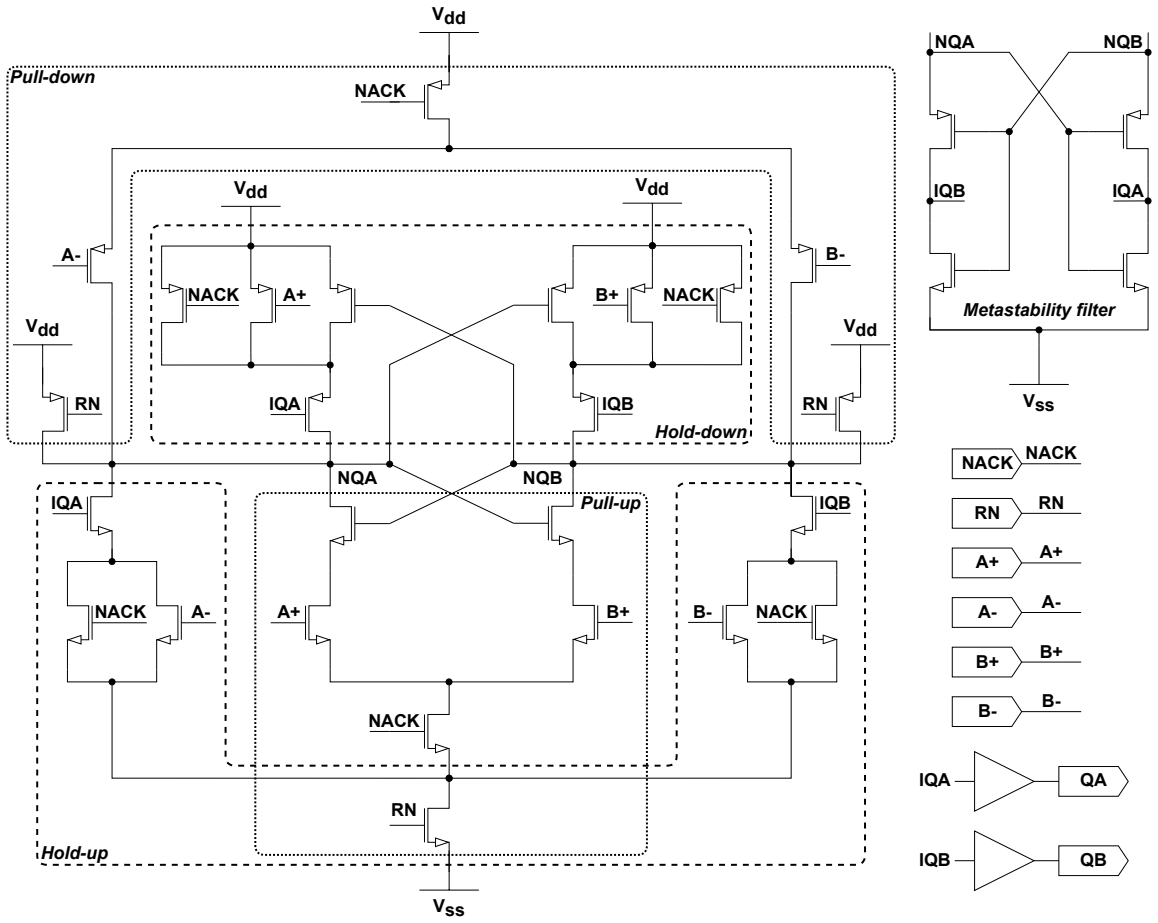
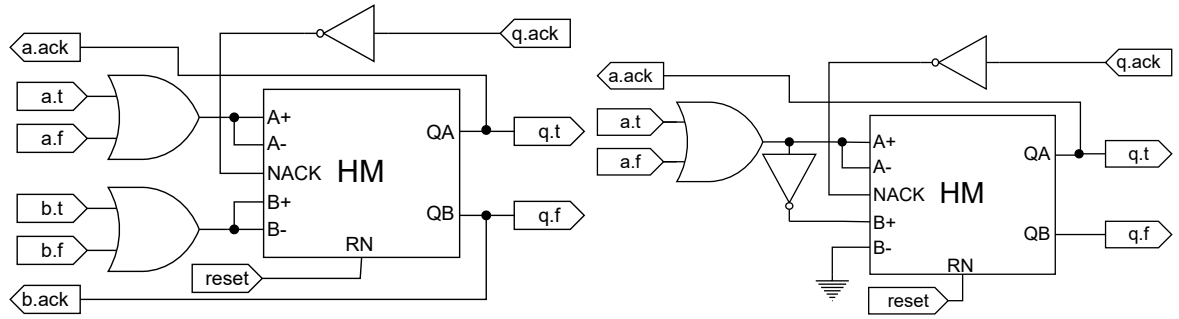


Figure 5.8: The handshaking mutex (HM) - a resettable state-holding mutual exclusion gate. It comprises two cross-coupled resetable asymmetric C-elements, a metastability filter and two output buffers. Output QA (QB) rises when inputs NACK and A+ (B+) are high; conversely, QA (QB) falls when both NACK and A- (B-) are low. Just one output can be high at any time, but both can be simultaneously low. If NACK rises when both outputs are low and inputs A+ and B+ are both high, the internal nodes NQA and NQB can enter a metastable state. The metastability filter ensures outputs QA and QB are low until the metastability resolves.

ready to accept a spacer, which puts the decision component in the reset step. The precise behaviour of the evaluation and reset steps are different in `arbiter` and `probe` components.

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

In contrast to the `arbiter`, a `probe` has a single (dual-rail) input channel and its behaviour on the computation and reset step changes on the presence of tokens and spacers



(a) Arbiter (b) Probe

Figure 5.9: The SDDS-NCL template decision-making components implementation. Asymmetric inputs on the HM allow implementing both decision components. The NACK signal indicates whether the consumer connected to the output waits for a token or for a spacer. OR-gates act as CDs to recognise the presence of tokens in the dual-rail input channel. HM guarantees that only a single rail of the output raises on the evaluation step of the arbitration cycle. It is worth noting that `probe` relies on HM to guarantee its correct behaviour in the rare occasion where both NACK and the output of the input channel CD change simultaneously.

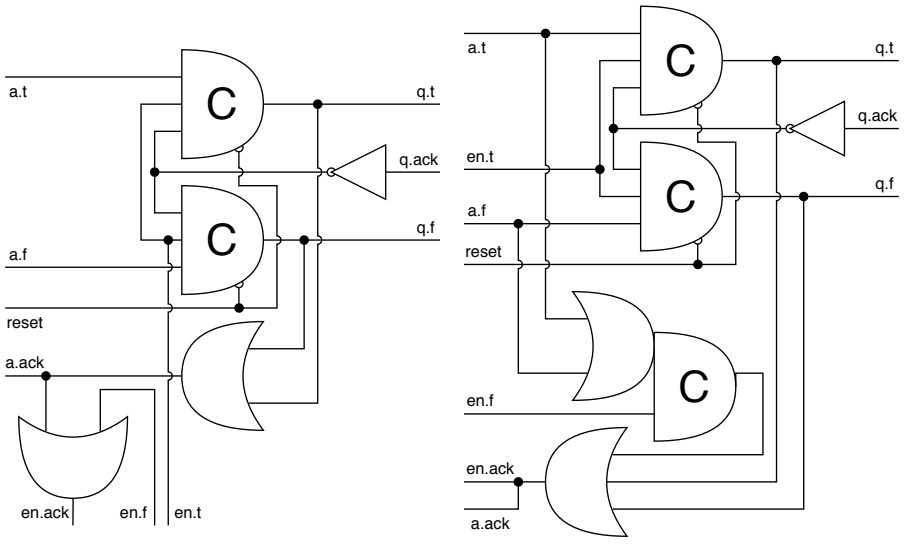
in its input channel. The CD output indicates the presence/absence of a token; it connects to the A+/A- input pair of the HM. The inversion of the CD output indicates the presence of a spacer; the negated CD output is connected only to the B+ input of the HM, and the B- input is tied to earth. When a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

5.2.4 The Token Steering Components

Both `discard` and `hold` act like temporal barriers that selectively perform handshakes on their data channels; a control channel governs this behaviour. When they receive true-valued tokens on their control channels, both components act as a conventional temporal barrier, latching and propagating data. Conversely, when the control channel receives a false-valued token, the token received in the data channel is inhibited from propagating. However, the inhibition behaviour of these control flow components is not the same. A `hold` component inhibits token propagation by withholding handshake with the input data channel; this effectively blocks the token at the input data channel. The `discard` component, on

the other hand, inhibits token propagation by acting as a token sink; it performs a handshake with its input data channel but does not propagate the token to its output channel. It is worth mentioning that the `discard` component only performs a handshake on its control channel synchronised with its data channel. In contrast, the `hold` component can perform a handshake on the control channel alone. Likewise, the `condhi` and `condlo` components are conditional token sources. They consume a token from their control channel; if a false valued token is received, the token production at the output channel is inhibited. When a true-valued token is received on the control channel, a `condhi` component produces a true-valued token and a `condlo` component produces a false-valued token.

The dual-rail RTZ implementations of `hold` and `discard` are depicted in Figure 5.10; they are similar to half-buffer components[5], but employ 3-input resettable C-elements to gate the propagation of tokens. They only propagate data arriving from the input channel if the true-rail of the control channel (*en*) is activated. The difference between `hold` and `discard` lies in the generation of the *ack* signal for the input and control channels. On both components, the control channel is always acknowledged, regardless of which rail of the control channel is activated. However, `hold` only acknowledges the data channel (*a*) when the token propagates; whereas `discard` acknowledges the data channel when either the token propagates or when the false-rail of the control channel is activated.



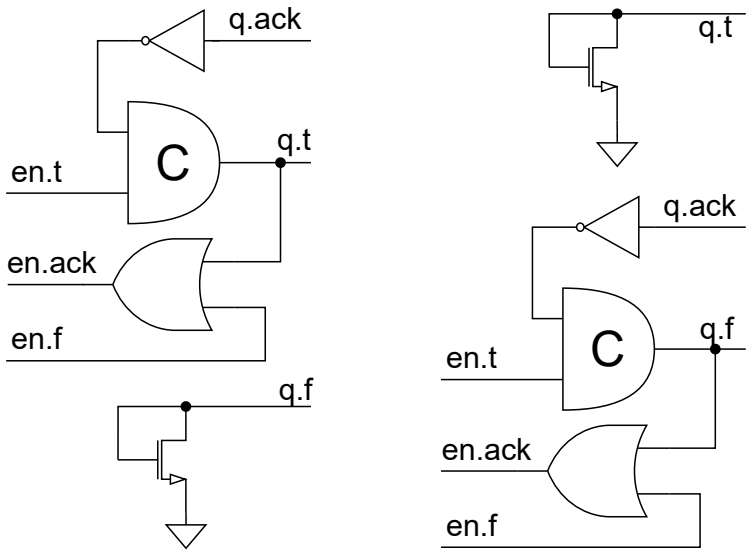
(a) `Hold`. (b) `Discard`.

Figure 5.10: Token steering components - Part 1

The dual-rail RTZ implementations of `condhi` and `condlo` appear in Figure 5.11. Each component comprises a C-element, an OR-gate, and a constant assignment to nil. The C-element is connected to the true-rail of the control channel and the negation of acknowledgement from the output channel. It generates the true or false output rail regarding

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

the respective component, whilst the other output rail is tied to logic nil. An OR-gate generates the acknowledgement signal for the control channel; it is connected to the output of the aforementioned C-element and to the false rail of the control channel.



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

Figure 5.11: Token steering components - Part 2.

Token steering enables the dynamic operation of asynchronous pipelines in the sense that the path followed by tokens can vary. This is possible through either *fan-out* or *fan-in* steering. Fan-out steering is used to select the destination of a token, i.e. demultiplexing channels. Fan-in steering is used to select the source of tokens for a channel, i.e. multiplexing channels. Using the token steering components for fan-in steering is accomplished by merging the output of multiple token steering components into a single channel and enabling only one of them. In Pulsar RTL-like input descriptions, channel merging is accomplished by placing multiple drivers in a wire; this is similar to using tristate buffers on synchronous circuits. During dual-rail expansion, this merging is implemented by ORing the data rails of the merged channels.

5.3 PS-SDDS-NCL - The Pseudo-synchronous SDDS-NCL Template

The Pseudo-synchronous SDDS-NCL template is the first template developed for Pulsar. It combines SDDS-NCL for combinational logic with the Pseudo-Synchronous WCHB for sequential logic. To synthesise Pseudo Synchronous WCHB circuits, pulsar uses the following handshake channel and components configuration: the handshake channel interface comprises two data rails and the acknowledgement signal; the combinational components are implemented using strong indicating virtual functions; and the sequential components manually instantiate pseudo-synchronous WCHB pseudo-flops from the target technology

library. At the virtual netlist level, every channel and component uses the RTZ protocol. This virtual netlist is synthesised using Genus into a flattened netlist comprising NCL and NCLP gates. During this synthesis, negative unate gates are inserted, creating regions using the RTO protocol. The pseudo-flops are instantiated manually and preserved with “size-only” during synthesis. They, along with primary inputs and outputs, are guaranteed to be at the RTZ domain, thus serving as start and endpoints for the graph colouring netlist fixing algorithm. The registers, primary inputs and outputs also are the start and endpoints for the timing constraints derived from the HBCN.

Figure 5.12 depicts the template structure on a simplified view of a pipeline stage. It shows only two handshake channels (out and in) with a combinational logic block placed in between. Each handshake channel has two opposing propagation paths, the forward- and the backwards-propagation paths. The two data rails (*t* and *f*) is the forward-propagation path, and the *ack* signal is the backwards-propagation path. The combinational logic block comprises both the strong indicating SDDS-NCL logic in the forward-propagation path and the C-Element completion tree in the backwards-propagation path. It is worth noting that this logic block, in fact, has multiple input and potentially multiple output channels; however, for simplicity, only a pair of input and output channels are shown.

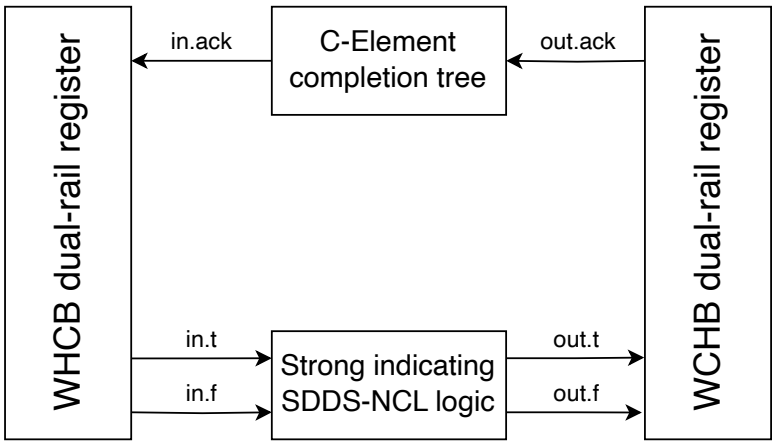


Figure 5.12: Pseudo-Sync SDDS-NCL template structure of a pipeline stage. This Figure depicts a simplified view with a single channel between two dual-rail registers, when in fact, a real pipeline stage comprises multiple channels and multiple registers at both sides.

The combinational logic block consumes data from multiple input channels to produce results in one or more output channels. This creates a dependency relationship between the inputs and output of a logical block; an output channel can only produce data once all inputs are valid, and an input channel can only be acknowledged once all output channels are acknowledged. Some coordination is required to guarantee a consensus on the acknowledgement signal to each input channel. The C-element provides the desired behaviour; however, due to physical limitations, it is impractical to build C-elements with more than 4 inputs on a CMOS process. However, C-elements placed in a tree structure are equivalent to a C-element with an arbitrary number of inputs. A C-element tree is placed in

the back-propagation path for each input channel to collect the acknowledgements from all output channels dependent on it. This C-element tree is called a *completion tree* because it collects the acknowledgements from the completion detectors in the registers.

The completion trees are constructed as a byproduct of using a SystemVerilog interface to model the handshake channels. Listing 5.6 depicts the implementation for the handshake channel interface and a combinational component. Notice how in line 3 the acknowledgement signal is declared using Verilog's *wand* wire type. When a channel has multiple consuming components, each consumer propagates their acknowledgements back by assigning them to the acknowledgement wire (lines 14-15). A wire of type wand performs an AND-reduction of multiple assignments; since channels are described using the RTZ protocol, the AND virtual function maps to C-elements. Similarly, when a channel has multiple components acting as producers, each assigns some value to the data rails (lines 12-13). The data rails' declaration in line 2 uses the *wor* wire type to OR-reduce these multiple assignments to a channel. Notice that multiple producers are not allowed to use a channel simultaneously. However, choice [SNC22] requires channels with multiple producers and the choice components provide the multiplexing logic to avoid the simultaneous use of a channel.

```
1  interface drwire();
2     wor t, f;
3     wand ack;
4
5     modport in  (input t, f, output ack);
6     modport out (input ack, output t, f);
7  endinterface // drwire
8
9  module nand2
10   (drwire.in a, drwire.in b, drwire.out y);
11
12    assign y.t = a.f & b.t | a.f & b.f | a.t & b.f;
13    assign y.f = a.t & b.t;
14    assign a.ack = y.ack;
15    assign b.ack = y.ack;
16 endmodule // nand2
```

Listing 5.6: SystemVerilog implementation of the handshake channel and NAND2 combinational component for the Pseudo-Synchronous SDDS-NCL template.

The construction of combinational logic from virtual functions using SDDS-NCL leverages advanced optimisation techniques employed by commercial EDA tools. Among these optimisations, logic sharing aids in reducing the gate count. For highly regular structures such as the completion trees, it enables multiple channels to share common subtrees.

5.4 WInDS - The Weakly Indicating Dual Spacer Logic Template

Strong indication logic requires that every input channel is complete prior to producing an output. Although robust, this can incur some performance and area penalties. Consider the two implementations of the two-input dual-rail AND gate depicted in Figures 3.9 and 3.10. The former is a weakly indicating version of the AND gate; it outputs false as soon as any of its inputs are false. The latter is the strongly indicating version; it waits for all inputs to be complete before producing a result, even if it has enough information to produce a correct result. A combinational component implementing a strong indicating dual-rail gate requires extra logic to validate its input completion, thus incurring some area overhead. This overhead compounds when constructing logic blocks comprising strong indicating components.

Weakly indicating dual spacer (WInDS) logic is the second template developed for Pulsar; it is first introduced here in this thesis. The synthesis procedure for WInDS is identical to the one used for the Pseudo-synchronous SDDS-NCL. The only differences are in the implementation of the handshake channel and components. WInDS aims at reducing the compound overhead of associating multiple strong indicating combinational components in sequence. A simplified view of a WInDS pipeline stage is depicted in Figure 5.13. WInDS is similar to a pseudo-synchronous SDDS-NCL pipeline stage: they both employ a C-element completion tree on their back-propagation paths, and they both employ pseudo-synchronous dual-rail registers. However, WInDS employs a weakly indicating variant of SDDS-NCL logic in the forward completion path. This simple change, of course, requires modifications in the template to guarantee its proper operation. Among these changes is the inclusion of a ready signal and a second completion detection tree.

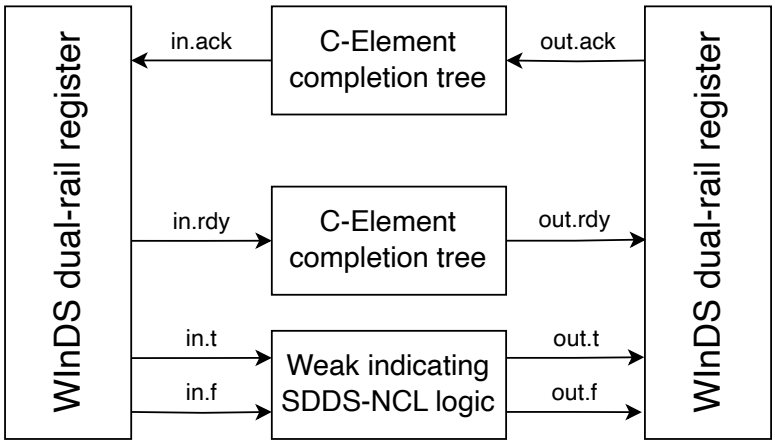


Figure 5.13: Simplified cut of a WInDS pipeline stage structure. The WInDS dual-rail register is a modified pseudo-synchronous WCHB dual-rail register.

Weakly indicating components do not observe the indication principle as they can produce results prior to having all of their inputs complete. Using them to construct com-

binational blocks can result in logic blocks that can produce data without having all inputs complete. When a logic block produces some data at its output, the sequential component can acknowledge this data. The C-element completion tree in the back-propagation path thus collects this acknowledgement and delivers it to the input channels. This can cause the input to be acknowledged before it even introduces data, thus violating the handshake protocol. Moreover, the weakly indicating logic block can also produce a spacer before all of its inputs have returned to spacers, causing similar protocol violations at the inputs. The solution for this problem is the adoption of forward completion detectors to indicate the completion of the logic block's inputs as proposed by Brej [Bre05] in his thesis. Adding a forward completion detector implies changing the RTZ handshake protocol to include a ready signal; this new protocol, promptly called WInDS-modified RTZ protocol, is depicted in Figure 5.14. A C-Element completion tree is introduced in the forward-propagation path to collect the ready signals from the input channels and propagates them to the output channels. The propagated ready signal, in conjunction with the DR code, indicates the codeword validity; in this manner, the indication principle is upheld at a block level.

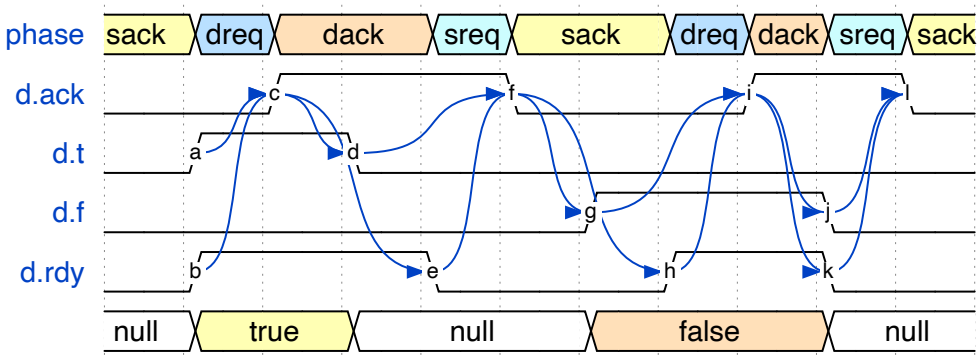


Figure 5.14: The WInDS-modified RTZ protocol, depicting the inclusion of a ready signal (rdy). The arrows indicate the causal relations between signals involved in the protocol. The example depicts the early generation of a spacer (d) and the early generation of a false token (g). In both cases, the received only acknowledges the spacer (f) and the token (i) after the ready signal agrees with the data lines (e and h).

Listing 5.7 depicts the `drwire` interface implementing the WInDS handshake channel. It is similar to the `drwire` implementation for the PS-SDDS-NCL template; its only difference is the inclusion of the ready signal in line 2. It is declared using the *wor* wire type, just as the data rails, to allow multiple producers on a channel. The combinational components are responsible for collecting the ready signals from their inputs and propagating them to their outputs. Since WInDS employs the same synthesis technique as SDDS-NCL, the ack signal is declared using the *wand* wire type to construct the C-element propagation tree.

The WInDS-modified RTZ protocol can easily be converted to and from the conventional RTZ protocol employed by PS-SDDS-NCL. Converting to conventional RTZ requires only C-elements, each placed in a data rail to merge with the ready signal. Whilst converting from conventional RTZ requires only an OR-gate acting as a completion detector to generate

```
1  interface drwire();
2     wor t, f, rdy;
3     wand ack;
4
5     modport in  (input t, f, rdy, output ack);
6     modport out (input ack, output t, f, rdy);
7  endinterface // drwire
```

Listing 5.7: SystemVerilog implementation of the handshake channel on WInDS.

the ready signal. This ease of conversion enables using sequential and choice components designed for the conventional dual-rail RTZ protocol. This protocol conversion is necessary to interface the environment when producing WInDS circuits with Pulsar. Listing 5.8 presents the WInDS implementation of the `drinput` and `droutput` modules instantiated on the virtual netlist to interface the environment. The `drinput` module converts from conventional RTZ to WInDS-modified RTZ, the only logic involved in this process is the instantiation of a completion detector in Line 9, the rest of the lines are just connecting the wires coming from the environment to the internal wires in the handshake channel interface. The `droutput` module converts from the WInDS-modified RTZ to conventional RTZ; since WInDS employ the same technology mapping as SDDS-NCL, the C-elements merging the ready signal with the data rails can be inferred from the AND virtual functions in lines 19 and 20.

```
1  module drinput
2    (input logic  t,
3     input logic  f,
4     output logic ack,
5     drwire.out   drw);
6
7     assign drw.t = t;
8     assign drw.f = f;
9     assign drw.req = t | f;
10    assign ack = drw.ack;
11 endmodule // drinput
12
13 module droutput
14   (output logic t,
15    output logic f,
16    input logic  ack,
17    drwire.in    drw);
18
19    assign t = drw.t & drw.req;
20    assign f = drw.f & drw.req;
21    assign drw.ack = ack;
22 endmodule // droutput
```

Listing 5.8: SystemVerilog implementation of the input and output modules on WInDS.

Although it is possible to convert protocols and adopt the dual-rail RTZ register from PS-SDDS-NCL, it is preferable to employ dual-rail registers designed for the WInDS-modified RTZ protocol. Figure 5.15 illustrates the register designed for WInDS; it is almost

identical to the WCHB register, except for including a third input pin to its C-elements. The ready signal from the left-hand channel is connected to this pin; it guarantees that the register does not latch any prematurely generated data or spacer. The completion detector (the OR-gate) is placed after the C-element; therefore, it only acknowledges the left-hand side after the data is safely latched. The same completion detector also generates the ready signal for the right-hand side. Different from early output logic [BG03, BE09, Bre05], the WInDS register only allows data and spacer propagation to the next pipeline stage after all inputs are complete. This simplifies the timing analysis, as the timing assumptions are constrained to within the pipeline stage.

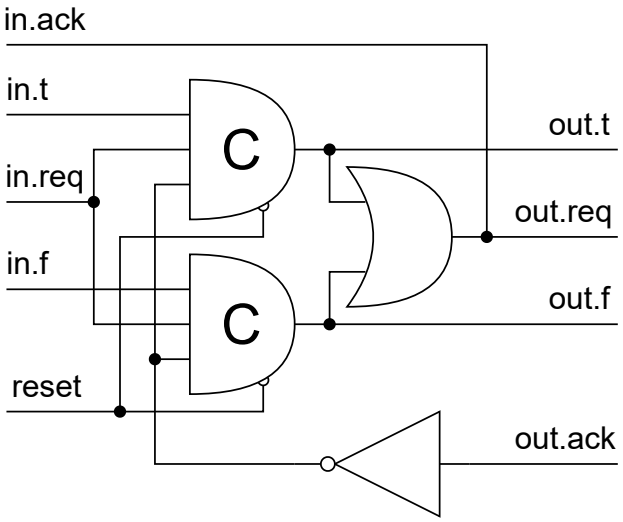


Figure 5.15: WInDS dual-rail register. The C-elements used here are modelled as pseudo-flops during synthesis; however, the pseudo-clock pin G is omitted here since it is not present in the circuit layout.

The forward completion tree depth grows logarithmically with the number of inputs entering the logic cone of a given logic block output. Depending on the circuit, this approach can have a lower overhead than a strongly-indicating logic block, which must acknowledge every single intermediary signal. Also, multiple outputs of a logic block can share parts of their completion trees, reducing its overhead. Of course, the addition of forward completion detectors leads to gate orphans. However, their presence in SDDS-NCL circuits indicates they are not as troublesome as first hypothesised.

Figure 5.15 depicts the changes to the sequential components, which were already discussed. Listing 5.9 details the changes to the implementations of the `nand2` combinational component. The forward-propagation completion tree is constructed using the C-element RTZ virtual function (AND) in an assignment (line 6) inside the combinational component description. The data rails are assigned to the weakly indicating virtual function (lines 4-5) corresponding to the component function. The back-propagation completion tree is constructed by assigning the ack signal from the output channel y to the ack signals declared as *wand* in channels a and b.

```
1 module nand2
2   (drwire.in a, drwire.in b, drwire.out y);
3
4     assign y.t = a.f | b.f;
5     assign y.f = a.t & b.t;
6     assign y.rdy = a.rdy & b.rdy;
7     assign a.ack = y.ack;
8     assign b.ack = y.ack;
9 endmodule // nand2
```

Listing 5.9: SystemVerilog implementation of the NAND2 combinational component on WInDS.

The use of weakly indicating functions in WInDS grants the EDA tool more room for optimisation of combinational logic. This comes at the cost of additional control circuitry in the form of an additional completion tree and bigger dual-rail registers. This additional area employed in the forward-completion completion tree can be offset by some logic sharing with the back-propagation completion tree of the previous pipeline stage. Also, depending on the logical depth of the pipeline stage, this completion tree can be smaller than the compound overhead of associating multiple levels of strong indicating components.

5.5 ALHO - The Asynchronous Limited Hysteresis Organisation Template

WInDS employs hysteretic gates to implement weak indication logic; this is, in fact, wasteful, as the role of hysteresis in ST circuits is to guarantee the indication principle. Hysteretic gates are bigger, consume significantly more power and are slower than non-hysteretic gates of the same virtual function. Consider the 2 input C-element in Figure 3.6; it comprises 12 transistors. On the RTZ protocol, its virtual function[6] is the 2-input AND, and it is the 2-input OR for the RTO protocol. Both 2-inputs AND and OR gates can be implemented with as little as 6 transistors on a static CMOS configuration, half the number of transistors in a C-element. This difference is even more staggering when considering the difference between a C-element with inverted output (12 transistors) and a NAND or NOR gate (both at 4 transistors). Given the EDA tools' preference for negative unate gates, this represents a one-third best-case area reduction.

Asynchronous Limited Hysteresis Organisation (ALHO)[7] is the third template proposed for Pulsar that aims to reduce area and increase performance. Like WInDS, it relies on the use of forward completion trees to indicate the completion of inputs in a logical block. However, it uses conventional logic gates for implementing both logic and completion trees. Figure 5.16 presents a simplified view of the template structure. The completion trees

[6] Remember that the virtual function captures the behaviour of the gate during the data propagation phase of the handshake cycle.

[7] Fun fact: the name ALHO is a wordplay. It means garlic in Portuguese.

for both the readiness and acknowledgement propagation are now split in two. Instead of using a C-Element tree, ALHO adopts an AND-reduction and an OR-reduction tree. The AND-reduction produces a high logical level only when all of its inputs are high; similarly, the OR-reduction produces a low logical level when all its inputs are low. These properties make the AND-reduction suitable to propagate the presence of data in the completion detectors and the OR-reduction suitable to propagate the presence of spacers.

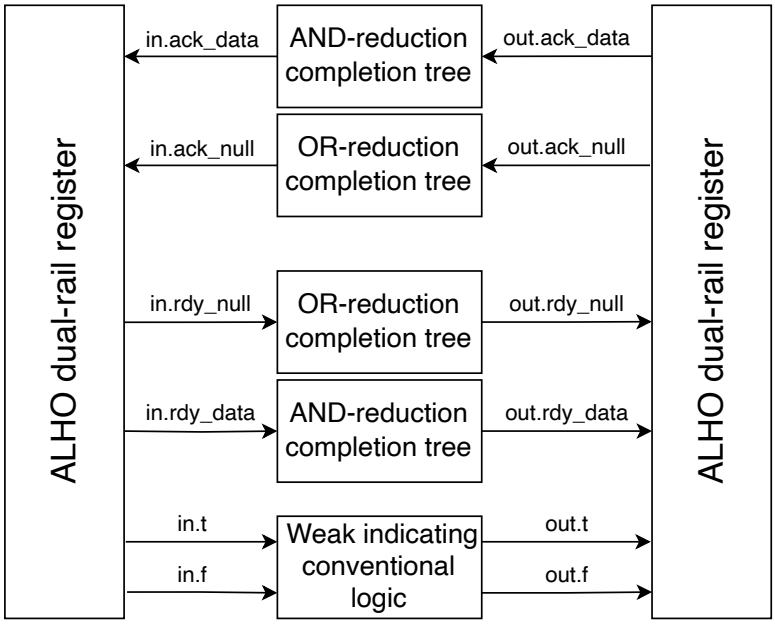


Figure 5.16: ALHO simplified pipeline stage structure.

This split in the completion trees demands changes to the handshake protocol; the aptly named ALHO-modified RTZ handshake protocol, depicted in Figure 5.17, splits the ready and acknowledge signals from the WInDS-modified RTZ protocol in two each. The data acknowledgement (`ack_data`) and data ready (`rdy_data`) signals are active high. The spacer acknowledgement (`ack_null`) and spacer ready (`rdy_null`) signals are active low. The protocol relies solely on the rise of `ack_data` and `rdy_data` to acknowledge and signalise data readiness, ignoring these signals' fall. Also, it relies solely on the fall of `ack_null` and `rdy_null` to acknowledge and signalise spacer readiness, ignoring their rise. This behaviour is what enables employing AND and OR reduction trees to propagate acknowledgement and readiness. It is worth noting that the data rails may change ahead of or after the ready signals; a weak indicating logic block can produce its result before all inputs are complete. But it may also produce a result in the data rails long after the ready signals indicate the completion of the input. This is not an issue since the register will only latch the data or the spacer once the relevant ready signals and the data rails agree.

Listing 5.10 depicts the ALHO implementation of the `drwire` handshake channel interface. Like the previous templates, the back-propagation completion trees are synthesised from the expansion of the handshake channel interface; the AND-reduction tree is

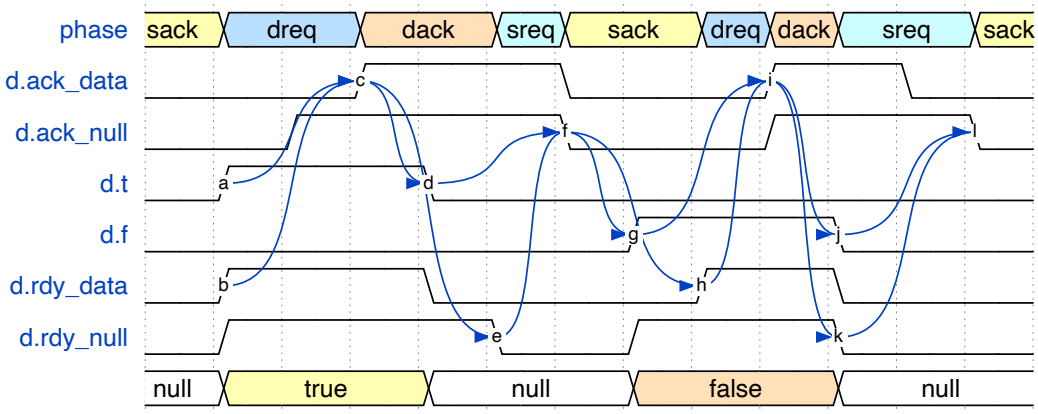


Figure 5.17: The ALHO-modified RTZ protocol, depicting the splitting of the ready (rdy) and acknowledge (ack) signals in their data (_data) and null variants (_null). The arrows indicate the causal relations between signals involved in the protocol. The example depicts the early generation of a spacer (d) and a false token (g). In both cases, the received only acknowledges the spacer (f) and the token (i) after the relevant ready signals agree with the data lines (e and h).

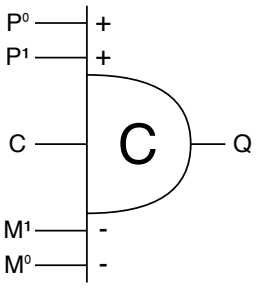
constructed using the *wand* wire type in line 4, and the OR-reduction tree is constructed using the *wor* wire type in line 5.

```
1  interface drwire();
2     wor t, f;
3     wor rdy_data, rdy_null;
4     wand ack_data;
5     wor  ack_null;
6
7     modport in  (input t, f, rdy_data, rdy_null,
8                  output ack_data, ack_null);
9     modport out (input ack_data, ack_null,
10                 output t, f, rdy_data, rdy_null);
11 endinterface // drwire
```

Listing 5.10: SystemVerilog implementation of the `drwire` handshake channel interface on ALHO.

However, hysteresis is still required to guarantee the proper functionality of the ST circuits. ALHO employs an asymmetric C-element to collect the completion information from the completion trees in both forward- and back-propagation paths. This specific C-element is depicted in Figure 5.18. It presents two positive and two negative asymmetric inputs (P^0, P^1 and M^0, M^1 respectively) and one symmetric common input (C). The asymmetric inputs are only sensitive to a single logical level: the positive inputs are sensitive to logical high, and the negative inputs are sensitive to logical low. The C-element output (Q) goes high when both positive inputs and the common input is high, regardless of the state of the negative inputs, and it goes low when both negative inputs and the common input are low, independently from the state of the positive inputs. This arrangement enables collecting 5 signals (the

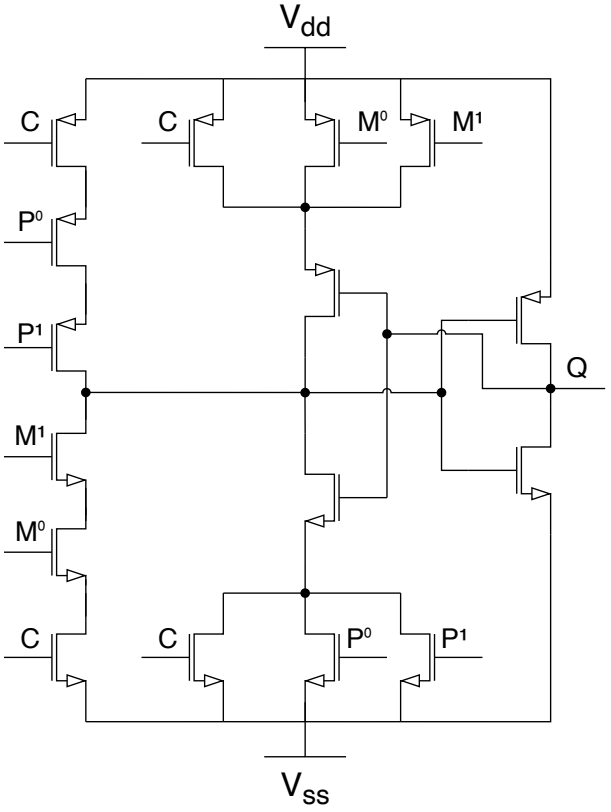
output of the combinational logic, the two back-propagating and the two forward-propagating completion trees) in a gate with the same number of transistors as a 3-input C-element.



(a) Symbol

P^0	P^1	C	M^0	M^1	Q_i
1	1	1	-	-	1
0	-	1	-	-	Q_{i-1}
-	0	1	-	-	Q_{i-1}
-	-	0	1	-	Q_{i-1}
-	-	0	-	1	Q_{i-1}
-	-	0	0	0	0

(b) Truth Table



(c) Schematic

Figure 5.18: C-element with two positive and two negative asymmetric inputs and one symmetric input.

ALHO employs the asymmetric C-Element from Figure 5.18 in its register implementation. Figure 5.19 presents its schematic. The C-elements are presented to the EDA tools as a pseudo-flop model; the pseudo-clock G is not depicted in the diagram as it is not realised in the final circuit. The data rails of the left-hand side (`lh.t` and `lh.f`) are connected to the common input of either C-elements. The data ready and spacer ready signals from the left-hand environment (`lh.rdy_data` and `lh.rdy_null`) are respectively connected to a positive and a negative asymmetric input in both C-elements. This arrangement conditions the latching and propagation of the data rails to the completion of the logic block inputs. Similarly to the conventional WCHB, the acknowledgement signals from the right-hand environment are inverted before entering the C-element. The inverted data and spacer acknowledgement signals (`rh.ack_data` and `rh.ack_null`) are respectively connected to the remaining negative and positive asymmetric inputs. This configuration guarantees the correct implementation of the handshake protocol; the progression of data and spacers is conditioned to the progression on the next pipeline stage. Like on the WInDS register, the OR-gate placed after the C-elements generates the acknowledgement and readiness signals to the left- and right-hand environments. At this point, the distinction between the spacer and data readi-

ness (and acknowledgement) is non-existent; it only becomes relevant after the AND- and OR-reduction trees.

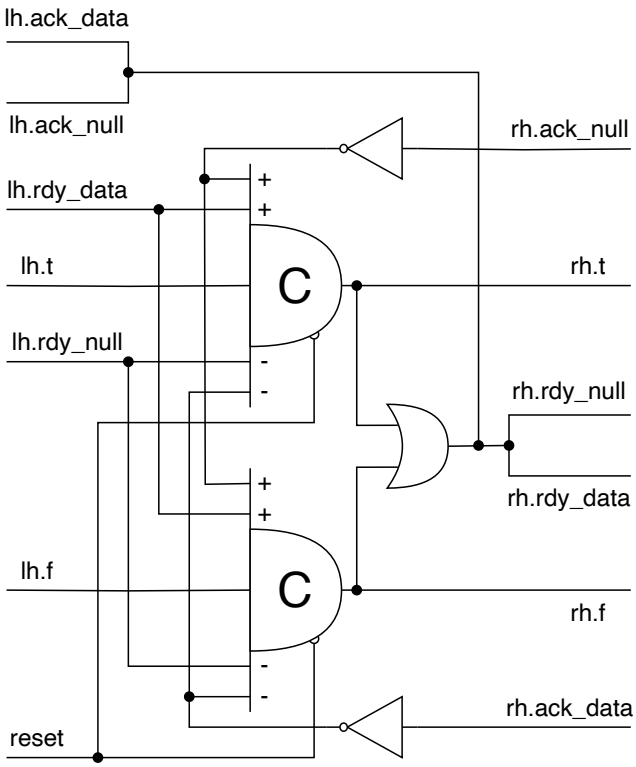
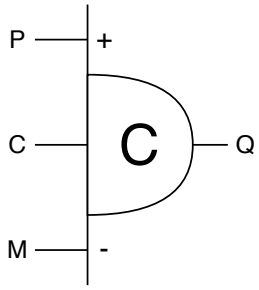


Figure 5.19: ALHO register. `lh` is the left-hand channel and `rh` is the right-hand channel. The pseudo-clock pin is omitted for simplicity, as it is only present to aid synthesis but is not realised in the final circuit.

Since circuits synthesised with Pulsar present to its environment an interface implementing the RTZ protocol, ALHO circuits must be able to convert between standard RTZ and the ALHO-modified RTZ protocol. Converting to conventional RTZ involves merging the ready signals with the data rails to produce spacers and tokens only when the ready signals and the data rails agree. This can be achieved using the asymmetric C-element depicted in Figure 5.20. Its behaviour is similar to the asymmetric C-element previously; however, it only has a single positive and negative input.

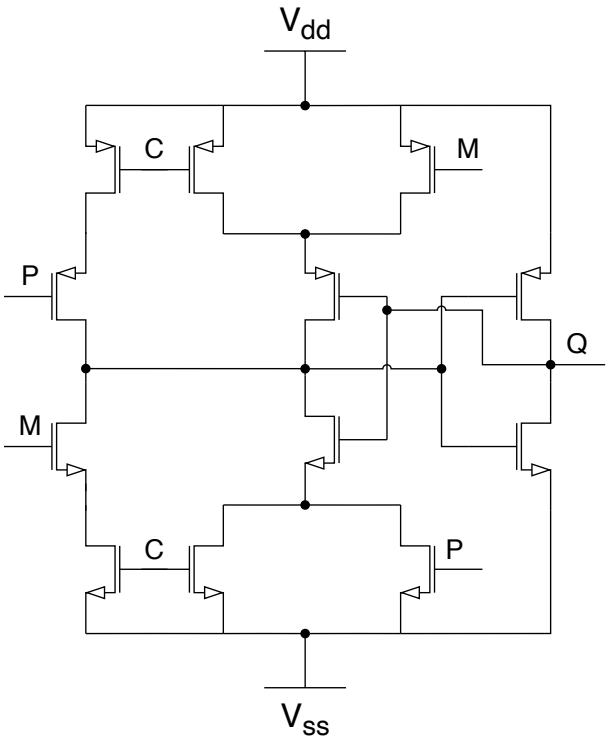
Listing 5.11 presents the `drinput` and `droutput` modules implementation for ALHO. At the `droutput`, lines 7 and 8 place an asymmetric C-element for each rail; the `rdy_data` signal is connected to its positive asymmetric input, the `rdy_null` to its negative asymmetric input, and the data rail to the common input. Since ALHO employ conventional gates in its synthesis process, hysteretic gates cannot be inferred from virtual functions and must be manually instantiated. Still on `droutput`, the acknowledgement signal coming from the environment can be simply connected to both acknowledgement signals in lines 10 and 11. The `drinput` module converts from conventional RTZ to the ALHO-modified RTZ protocol. This process involves inferring an OR-gate to act as a completion detector to generate the two request signals in lines 21 to 23 and instantiating a 2-input C-element to merge the two



(a) Symbol

P	C	M	Q_i
1	1	-	1
0	1	-	Q_{i-1}
-	0	1	Q_{i-1}
-	0	0	0

(b) Truth Table



(c) Schematic

Figure 5.20: C-element with one positive and one negative asymmetric input and one symmetric input.

ack signals into one in line 25. Notice that the two data rails from the environment are simply bypassed to the `drwire` interface internal rails in lines 27 and 28.

```
1  module droutput
2    (output logic t,
3     output logic f,
4     input  logic ack,
5     drwire.in    drw);
6
7     ACELEM11X4 t_i (.A(drw.t), .P(drw.req_data), .M(drw.req_null), .Q(t));
8     ACELEM11X4 f_i (.A(drw.f), .P(drw.req_data), .M(drw.req_null), .Q(f));
9
10    assign drw.ack_data = ack;
11    assign drw.ack_null = ack;
12 endmodule // droutput
13
14 module drinput
15   (input logic  t,
16    input logic  f,
17    output logic ack,
18    drwire.out   drw);
19    logic req;
20
21    assign req = f | t;
```

```
22      assign drw.req_data = req;
23      assign drw.req_null = req;
24
25      CELEM2X4 ack_i (.A(drw.ack_data), .B(drw.ack_null), .Q(ack));
26
27      assign drw.t = t;
28      assign drw.f = f;
29  endmodule // drinput
```

Listing 5.11: SystemVerilog implementation of the input and output modules on ALHO.

However, the widespread use of asymmetric C-element in ALHO assumes that the data-ready signal will be de-asserted before the propagation of new data; likewise, it is assumed that the null-ready signal will be asserted before the arrival of a new spacer. This is a somewhat safe assumption given that the OR-reduction tree starts propagating a rise in cascade when any of its inputs rise, and the AND-reduction tree behaves similarly when propagating a fall on any of its inputs. To avoid this assumption, alternatively, a 5-input C-element (or a multiple C-element equivalent) could be employed in registers instead of the asymmetric C-element; likewise, a 3-input C-element could replace the asymmetric C-elements at outputs. These changes would acknowledge the rise and fall of all signals involved at the cost of area and power.

The synthesis procedure for ALHO is rather straightforward. Since it employs conventional gates, Genus is capable of synthesising the correct final netlist directly, thus eliminating the netlist correction step required for circuits employing hysteretic gates. Listing 5.12 depicts the implementation of the `nand2` component. Similar to WInDS, the forward-propagation completion trees are constructed within the combinational components: line 6 constructs the forward-propagation AND-reduction tree, and line 7 the OR-reduction tree. The combinational component's logical functionality is implemented using weakly indicating virtual functions (lines 4 and 5). The assignments that construct the back-propagation trees are depicted from line 8 to 11.

```
1  module nand2
2    (drwire.in a, drwire.in b, drwire.out y);
3
4      assign y.t = a.f | b.f;
5      assign y.f = a.t & b.t;
6      assign y.rdy_data = a.rdy_data & b.rdy_data;
7      assign y.rdy_null = a.rdy_null | b.rdy_null;
8      assign a.ack_data = y.ack_data;
9      assign a.ack_null = y.ack_null;
10     assign b.ack_data = y.ack_data;
11     assign b.ack_null = y.ack_null;
12 endmodule // nand2
```

Listing 5.12: SystemVerilog implementation of the NAND2 combinational component on ALHO.

6. EXPERIMENTAL RESULTS AND CONCLUSION

This Chapter evaluates and compares the three templates using a set of experiments. The experiments are split into two subsets; first, the templates are subject to a set of low-complexity designs crafted to stress specific aspects of the templates, and later the three templates are employed to implement an asynchronous RISC-V processor [NSMC23]. The last Section of this Chapter is reserved for the author's final remarks and ideas for future work.

6.1 Experimental Setup

The synthesis process for all experiments in this Chapter is the same. The Pulsar front end uses Cadence Genus to produce virtual netlists from the RTL-like description for each circuit. In the front end, each circuit is re-timed to reduce the logic depth in the forward propagation paths. Similarly to the front end, the Pulsar back end also relies on Cadence Genus for logical synthesis. The resulting virtual netlists are synthesised to each template using the suitable Pulsar back end under a range of cycle time constraints. At the end of the logical synthesis, the back end produces reports that are used to extract some of the results depicted in this Chapter; the pulsar back end also extracts delay-annotated netlists for simulation at this step. After the logical synthesis step, all circuits are placed and routed using Cadence Innovus, where delay-annotated netlists are extracted again for simulation.

The delay-annotated netlists extracted at the logical synthesis step comprise only gate delays, the wires at this step are considered ideal, and the delay resulting from their estimated capacitance is lumped into the gates' output delay. This implies that all wire forks are guaranteed to be isochronic in the netlist delayed-annotated at logical synthesis. Conversely, the delay-annotated netlists extracted after place-and-routing adopt a more realistic wire model. In this wire model, the delays resulting from resistive and capacitive effects affect each wire branch differently, thus possibly violating the isochronic fork assumption on sensitive forks.

All syntheses are performed using the worst corner of ASCEND-FreePDK45, a standard cell library developed by the author of this thesis in the course of his PhD elaboration. A version of this library is publicly available in his GitHub [SMC20b] and will be updated by the time of this thesis' publication. The corners characterised in this library are:

- *best*: fast PMOS and NMOS transistors at 1.25 V and 0 °C.
- *nominal*: typical PMOS and NMOS transistors at 1.1 V and 25 °C.
- *worst*: slow PMOS and NMOS transistors at 0.95 V and 25 °C.

The resulting delay-annotated netlists, including and excluding wire delays, are simulated using Cadence XCelium at all library corners. Since Pulsar produces netlists with the same interface for all templates, all implementations of a particular design share the same SystemVerilog testbench. This guarantees that the stimulus applied to all implementations of each circuit is equal across templates, constraints and corners. Each testbench checks the correction operations of the design under test and collects the cycle time as a performance metric at its inputs and outputs. The testbenches simulate a null-delay ideal environment; as such, only the design under test affects the cycle time measures. Power figures were extracted using Cadence Voltus for each post-layout simulation using the characterised liberty file for the respective corners. The power estimations were elaborated using the switching activity annotation collected during the first 2 µs of each simulation.

6.1.1 Low complexity Benchmarks

The low-complexity designs employed as benchmarks are:

1. *ripple_adder*: A 32-bit purely combinational ripple-carry adder.
2. *piped_adder*: A 32-bit 4-stage pipelined adder comprising 8-bit ripple-carry adders.
3. Four 8-by-8 to 24bit multiply-accumulate circuits:
   (a) *mac3*: With a 3-stage accumulation loop.
   (b) *mac4*: With a 4-stage accumulation loop.
   (c) *mac5*: With a 5-stage accumulation loop.
   (d) *mac6*: With a 6-stage accumulation loop.
4. Four 32-by-32 to 64bit booth multiplier circuits:
   (a) *mult3*: With a 3-stage accumulation loop.
   (b) *mult4*: With a 4-stage accumulation loop.
   (c) *mult5*: With a 5-stage accumulation loop.
   (d) *mult6*: With a 6-stage accumulation loop.

Circuit 1 explores the corner case of a pure combination circuit; it is also a circuit that potentially benefits from weakly-indication to produce output before carry propagation. Circuit 2 also potentially benefits from weakly indication; however, the inclusion of pipelining aims at stressing how the templates and pulsar deal with sequential logic compared to an equivalent pure combinational approach. Circuits 3a to 3d are non-linear deterministic pipelines; as such, they have been used previously to stress the cycle-time constraining capabilities

of Pulsar in [SWMC19] and [SMC20a]. Circuits 4a to 4d are non-linear non-deterministic pipelines comprising a state-machine control logic; they stress not only the cycle-time constraining capabilities but also how the circuit copes with token steering.

The RTL-like source code for these designs is included in Apendix B. Figure 6.1 depicts the component count extracted from the front-end synthesis reports detailing the proportion of combinational and sequential components employed in each test circuit. These proportions will be important when evaluating the extracted figures and comparing the templates.

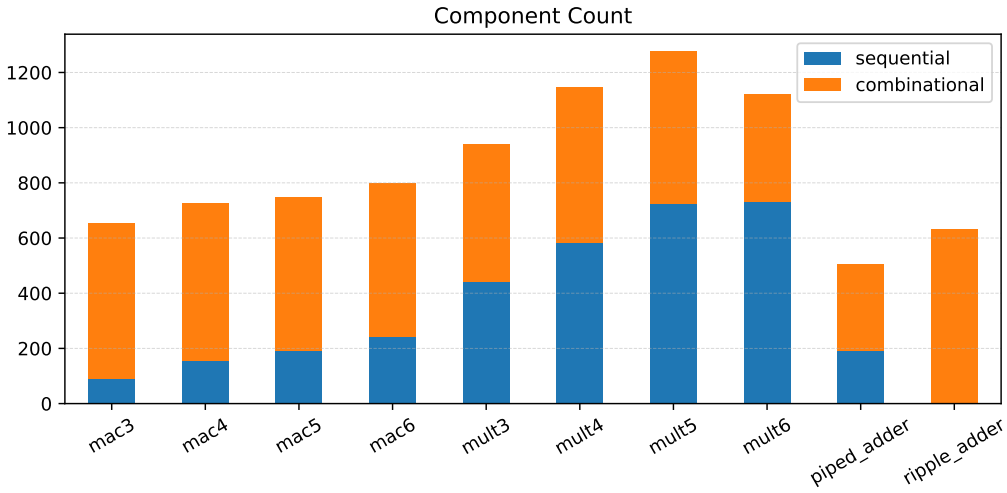


Figure 6.1: Component count of each circuit.

The designs were synthesised to each template under a range of cycle-time constraints from 2 ns to 10 ns in steps of 250 ps, limited to a minimum delay of 500 ps per propagation path. This minimum delay limitation guarantees a minimal timing budget for each propagation path, especially in tighter cycle-time constraints. However, it restricts the lower bound of the cycle-time constraint range in circuits comprising long critical cycles. The back-end synthesis resulted in 924 circuit implementations combining all design variations, templates and constraints. All these implementations were subsequently placed and routed using Innovus. The netlists resulting from the back-end synthesis and their placed-and-routed layouts were each delay-annotated and simulated to all three corners of the ASCEnD FreePDK45 Library. This resulted in 6 simulation corners for each implementation, three annotated from the layout and three from the logical synthesis.

6.1.2 High Complexity Benchmark: Asynchronous RISC-V Processor (ARV)

ARV, an asynchronous RISC-V processor, was chosen to evaluate the templates' performance on a high-complexity real-world design. ARV originated from a concurrent communication processes description [Sar17], and later Nunes et al. [NSMC23] translated

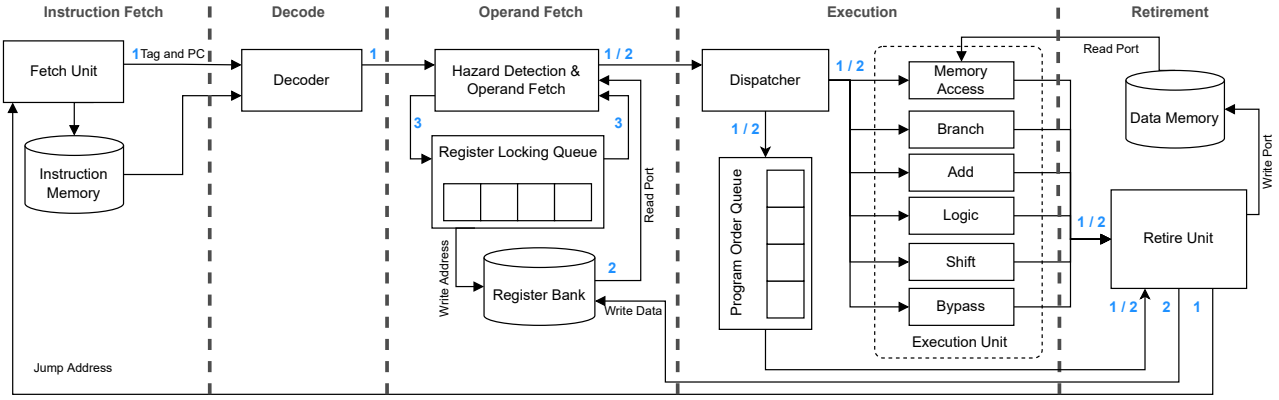


Figure 6.2: The ARV processor organisation. There are five stages, identified and separated by vertical dashed lines. Numerals in edges show elements that take part in three processing loops: (1) control loop; (2) datapath loop; and (3) register lock loop.

it to Pulsar's RTL-like SystemVerilog. ARV exploits the inherent structure of asynchronous circuits to enable instruction-level parallelism. Figure 6.2 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's 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, 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.

The decision-making and token steering components introduced in Section 5.2 are heavily employed in the construction of ARV. In fact, these components were developed concomitantly with ARV to enable the use of Pulsar in its synthesis.

- The fetch unit employs the `probe` component to detect the arrival of a jump address and update the program counter.
- The register locking loop uses the `hold`, `discard` and `condhi` components to hold instructions and issue bubbles.
- The register bank relies on the `discard` component to efficiently multiplex the register read and write operations.
- The dispatcher uses the `discard` component to steer instructions into the proper execution unit.
- The retire unit uses the `hold` component to recover instructions in the program order from their respective execution units.

The only choice component not employed in the construction of ARV is the `arbiter`. This widespread use of choice makes ARV the ideal candidate to stress the implementation of these components in all three templates.

The front-end synthesis resulted in a virtual netlist with the component count profile depicted in Figure 6.3. The resulting virtual netlist was synthesised to all three templates under a range of cycle time constraints from 4 ns to 10 ns in steps of 250 ps with a minimum propagation delay of 400 ps.

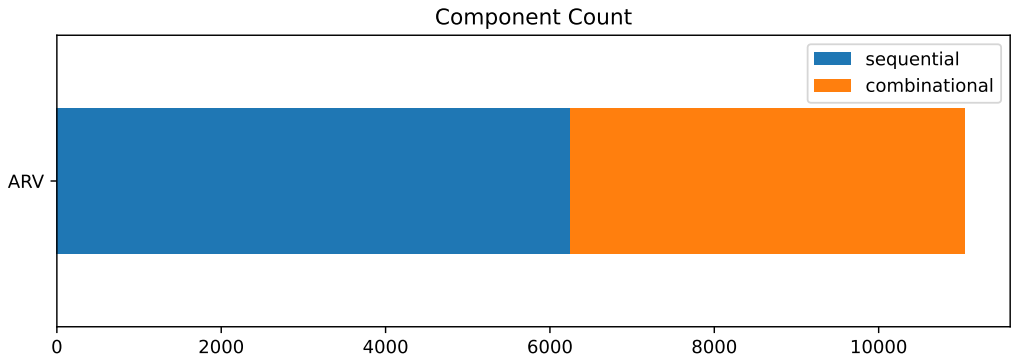


Figure 6.3: ARV component count

The back-end logical synthesis process resulted in 75 implementations of ARV, comprising 25 cycle-time constraints for each of the 3 templates, from which 74 resulted in placed-and-routed layouts. Unfortunately, a bug in Innovus triggered a memory explosion deadlock when placing and routing the 10 ns ALHO implementation. Due to time limitations in finishing the experiments for this thesis submission, the bug was not resolved, but it will be for the final revision of this volume.

The 75 resulting synthesised netlists and 74 placed-and-routed layouts were delay annotated and simulated using the three corners of the ASCEnD FreePDK45 Library. Resulting in six simulation corners for each implementation (sauf the missing layout), three annotated from logical synthesis and three from layout. Two testbenches were employed in these

simulations: the first ran RISC-V Fundation's official RISC-V compliance test suite [RIS15], henceforth wrongly named BerkeleySuite in plotted results; the latter ran one iteration of EMBC's Coremark [EMB02]. The cycle time figures extracted from ARV are measured as the time between two instruction fetches from memory. Due to ARV's internal organisation, internal execution factors like bubble insertion for data-dependency hazard correction, branching, and variable execution time of instructions impact the collected cycle time figure.

6.2 Comparisons of Experimental Results

The testbenches employed in the simulations checked the circuits' behaviour correctness. They tested if the circuit completed their computation and if they were producing the expected result. From all simulations in all corners, unfortunately, some presented faults. Figure 6.4 depicts the simulation success rate for each design in each corner. Theoretically, it was expected that ALHO would present more faults as it relies on more timing assumptions, followed by WInDS and PS-SDDS-NCL, with the last expected to have fewer faults. However, the experiments show a different picture; ALHO fared pretty well at tolerating the limited delay variability presented at the simulated corners. It presented the highest success rate on ARV and presented fewer faults than WInDS on the low-complexity benchmarks. Moreover, WInDS fared rather badly, presenting most faults in both high- and low-complexity benchmarks. PS-SDDS-NCL fared great on the low-complexity benchmarks, presenting no faults, but struggled with the ARV benchmark.

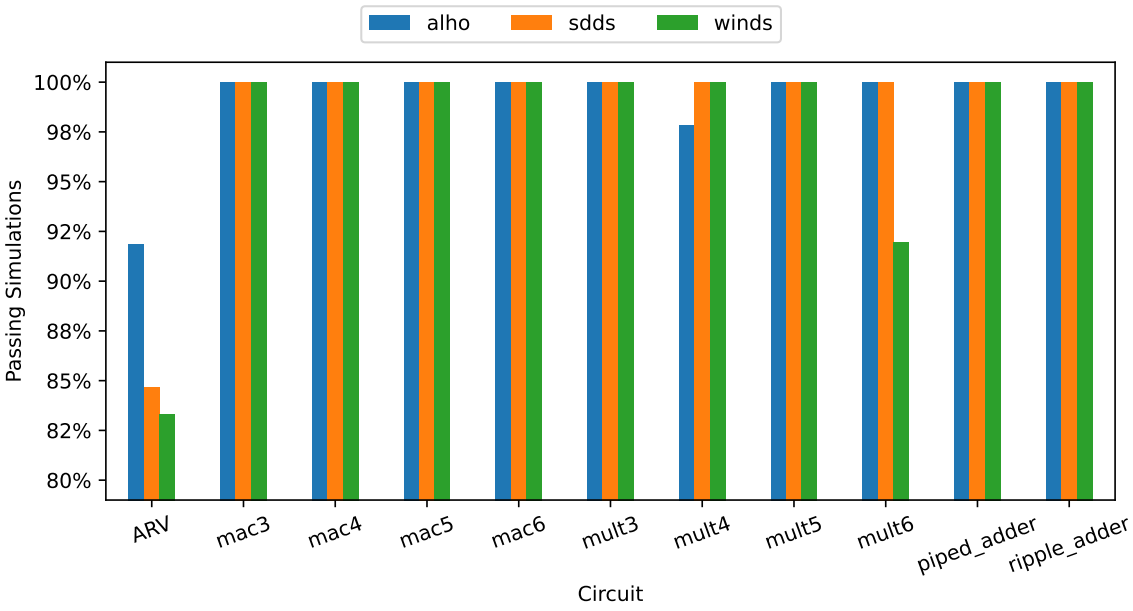


Figure 6.4: Percentage of circuits passing tests at all corners for each template.

A detailed view of the faults observed in the multiplier circuits is depicted in Figure 6.5. Whilst the circuits implemented with WInDS failed by producing wrong results, the

faulty ALHO circuits only became deadlocked. Producing a wrong output is worse than just stopping to compute results altogether. A deadlocked circuit is non-functional and can be easily detected and discarded, whilst a circuit producing wrong results, especially if it only does so under specific conditions, requires more elaborated testing to detect the fault. Also, observe that most faults occur in the least tightly-constrained implementations. On more relaxed cycle-time constraints, the path constraints allow the creation of very slow paths concurrent with fast paths; this increases the delay difference in the propagation paths of combinational logic, increasing the chance to violate the assumption proposed in Section 3.8. Furthermore, notice that in some cases, wire delays fix the observed faults; this is true for all observed faults on the mult4 ALHO circuits. This indicates that these faults observed in mult4 do not arise from violating the isochronic fork assumption, as such violations are only observable when modelling wire delays. The correction of delays in these cases is attributed to the wire delays, potentially reducing the delay difference between the fastest and slowest paths. Conversely, for the mult6 WInDS circuits, wire delays have introduced faults in most constraints and eliminated them in just one. Introducing faults with the introduction of wire delays is the expected behaviour, as wire forks can become another source of orphans in the circuit.

Figure 6.6 details the faults observed when simulating ARV. Albeit very similar, the fault profile on ARV depends on the program being executed. When running Coremark, the circuits implemented with ALHO only presented faults on the post-layout worst corner. However, when running the BerkeleySuite, the circuit synthesised to a constraint of 4.5 ns becomes deadlocked on the best post-layout corner. This is mostly due to the instruction ordering and timing exercising an unresolved orphan in one program but not in the other. The widespread use of choice components potentially exacerbates this effect. Also, notice that the trend of presenting faults at the more relaxed cycle time constraints is maintained, except that on ARV the circuit presents more faults at the worst post-layout corner on all templates. Furthermore, the only template observed to produce circuits misexecuting programs in these experiments was PS-SDDS-NCL.

It is noticeable that the only circuits presenting faults in Figure 6.4 are the ARV and two of the booth multipliers, all circuits employing choice components. This indicates that these components might have introduced unpredicted timing problems or made the circuits more sensitive to sources of delay variations. Further research on this phenomenon ought to be the subject of future work.

The rationale behind the construction of ALHO and WInDS was trading robustness for performance, power and area (PPA). An analysis of the observed faults in simulation show that their robustness varies, with PS-SDDS-NCL showing the best robustness on the low-complexity benchmarks but with ALHO faring better on the high-complexity ARV design. It is expected that due to the widespread use of conventional gates, ALHO will fare better in all PPA metrics compared to the two other templates.

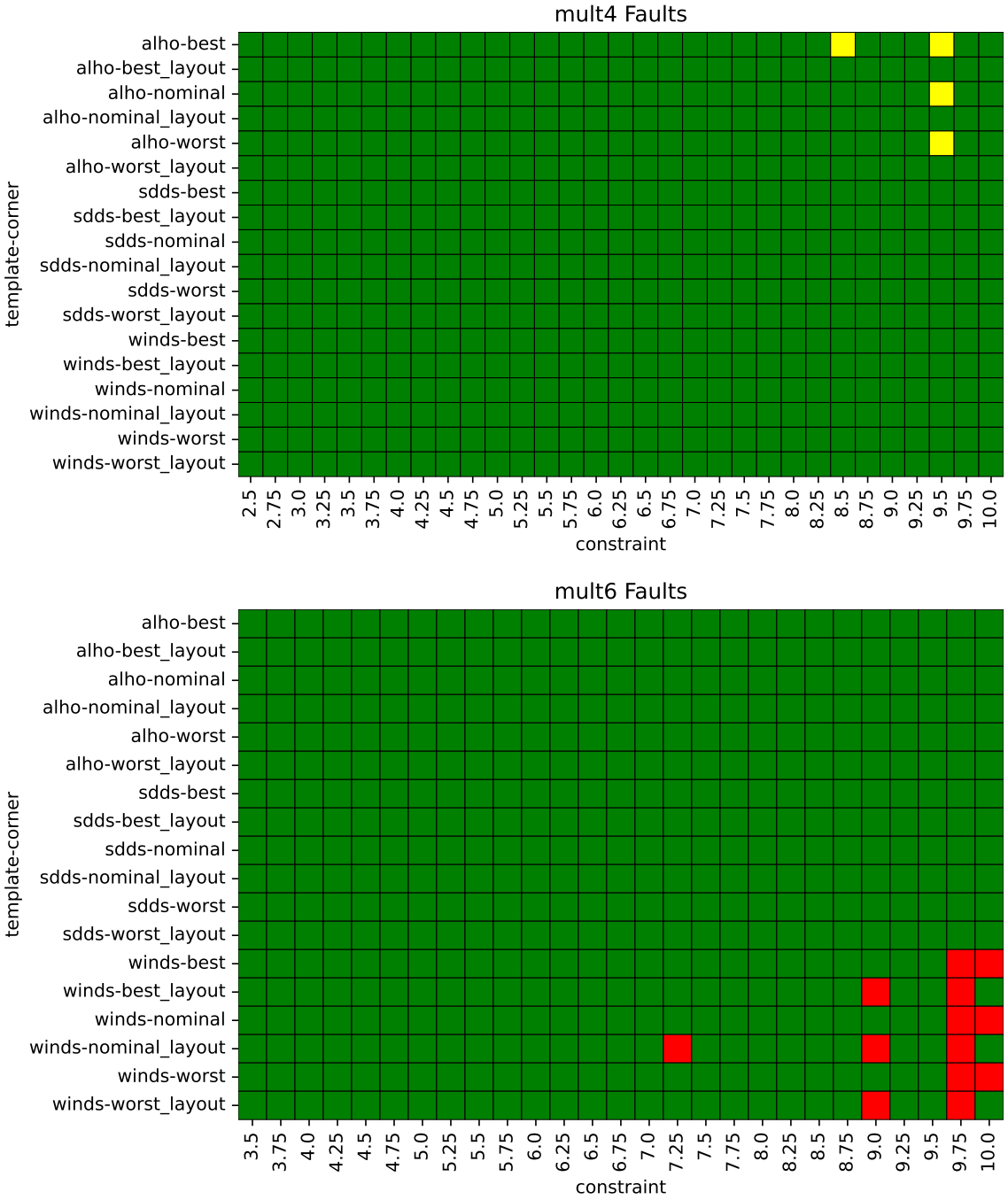


Figure 6.5: Detailing per template and cycle-time constraint of the multiplier faults observed in Figure 6.4. Green means correctly computing all 10000 multiplications, yellow means the circuit became deadlocked at some point, and red means the circuit produced a wrong output.

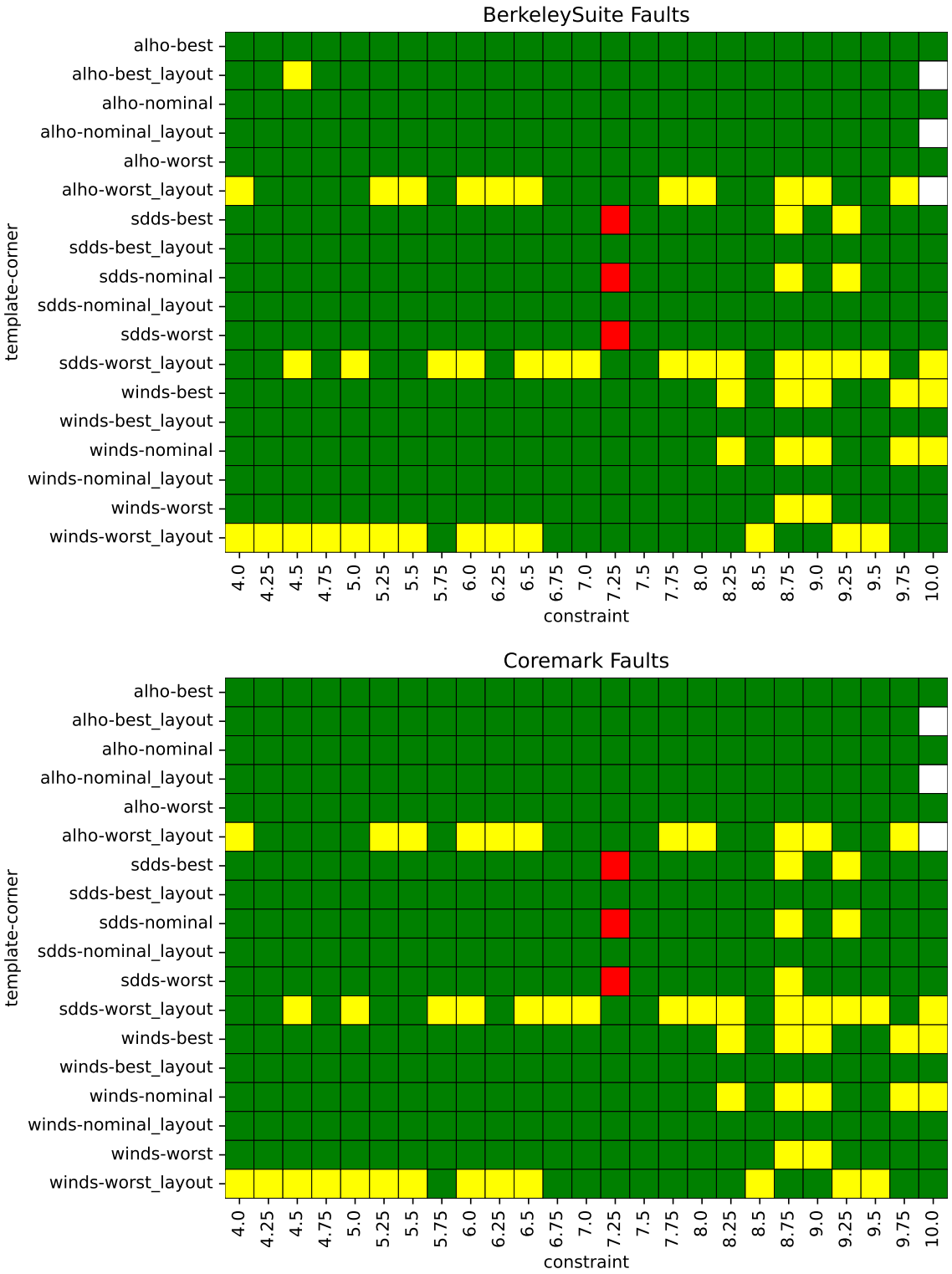


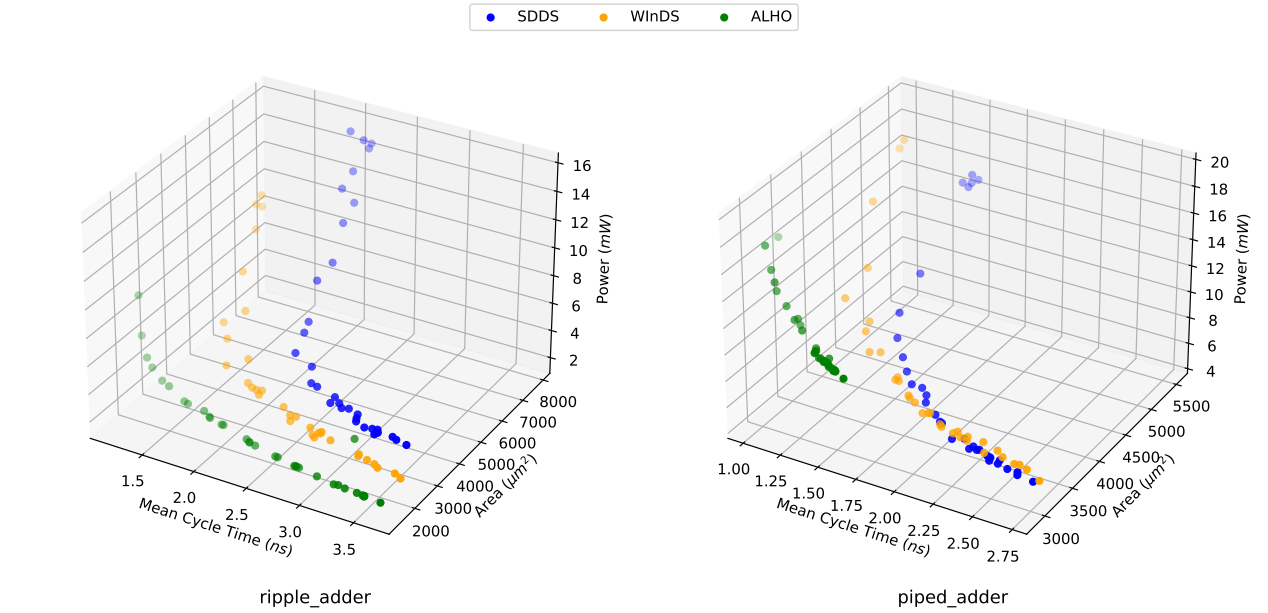
Figure 6.6: Detailing per template and cycle-time constraint of the ARV faults observed in Figure 6.4. Green that the program completed its execution correctly, yellow means the circuit became deadlocked at some point, red means the program misexecuted, and white means it lacks that simulation.

Figure 6.7 depicts the PPA figures of the two adders. It plots the mean cycle time against the gate area and the power figure for each circuit implementation. The mean cycle time is collected from the nominal corner post-layout simulation, the gate area is extracted from synthesis reports, and the power figures are computed using Voltus with activity annotation collected from the aforementioned simulation. These results show that the expectations for the templates' PPA are clearly met when analysing the pure-combinational ripple_adder design. However, when analysing the figures for the piped_adder, it is possible to notice that although ALHO's performance and area figures are far superior, the three templates present similar power figures in the range of cycle times they overlap. Still, ALHO can reach low cycle times that the two other templates cannot.

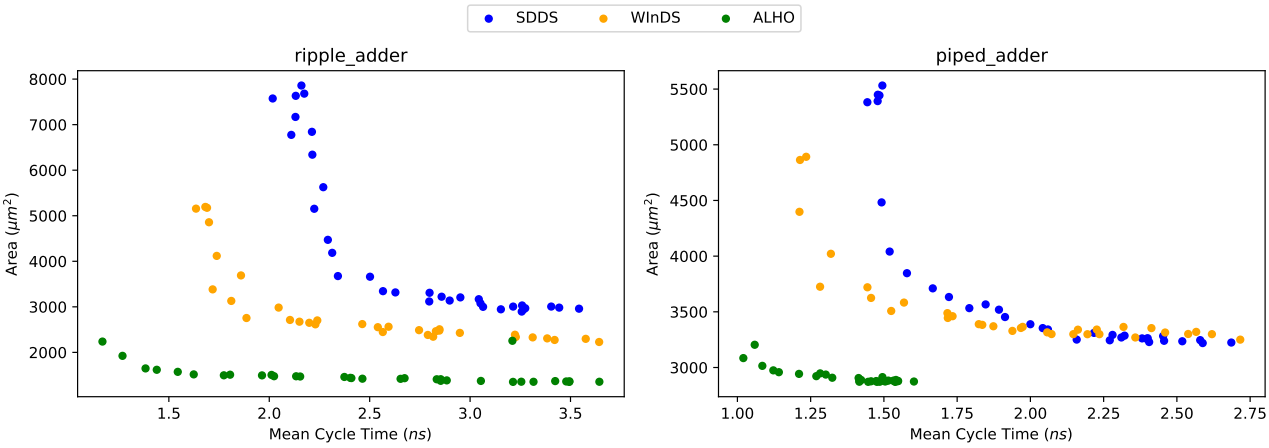
The effort made by the EDA tool for each template can also be observed when analysing how the cycle-time constraint affects the adders' PPA in Figure 6.8.[1] Remember that all circuits were synthesised to the worst corner; therefore, the cycle time constraint attempts to restrict the cycle time of the worst corner. It is possible to observe that the area and power figures explode on circuits where the worst corner simulation shows violations of the cycle time constraint. This behaviour shows the EDA tool tries to apply many optimisation techniques to increase performance and meet the constraint; however, although the increase in area and power, the tool ultimately fails, as shown by the consecutive marginal gains in performance not being enough to meet the cycle time constraint. As expected, the circuits designed with PS-SDDS-NCL start violating their cycle time constraint at a higher constraint value. WInDS has shown a bit more flexibility in this regard, with its area not increasing as much as PS-SDDS-NCL's, as it is able to meet slightly tighter constraints. ALHO excelled in meeting the cycle time constraint, only violating it for constraints under 3 ns on the ripple adder and only violating the 2 ns constraint on the piped adder. However, ALHO presents higher power figures overall on the piper adder, whilst its mean cycle time remains comparatively low regardless of the constraint. This higher power is attributed to the higher switching activity caused by the overall faster circuit.

Figures 6.9 and 6.10 present the PPA results for multiply-accumulate (MAC) designs. The 4-stage MAC is the fastest design for ALHO, presenting a good balance between sequential and combinational components. Whereas PS-SDDS-NCL and WInDS benefit from the extra pipeline stages on the 5- and 6-stage designs. The 3-stage design is the most dominated by combination components of all MAC. In the 3-stage MAC, ALHO achieves a lower cycle time at a fraction of the power and area of the fastest 3-stage WInDS or PS-SDDS-NCL implementation. Also, the 3-stage ALHO implementations presented slightly lower power figures on all cycle times achieved by the three templates, indicating a gain in power efficiency. As the ratio of sequential to combinational components increase, ALHO's advantage in power efficiency diminishes. It is possible to observe in the 6-stage MAC de-

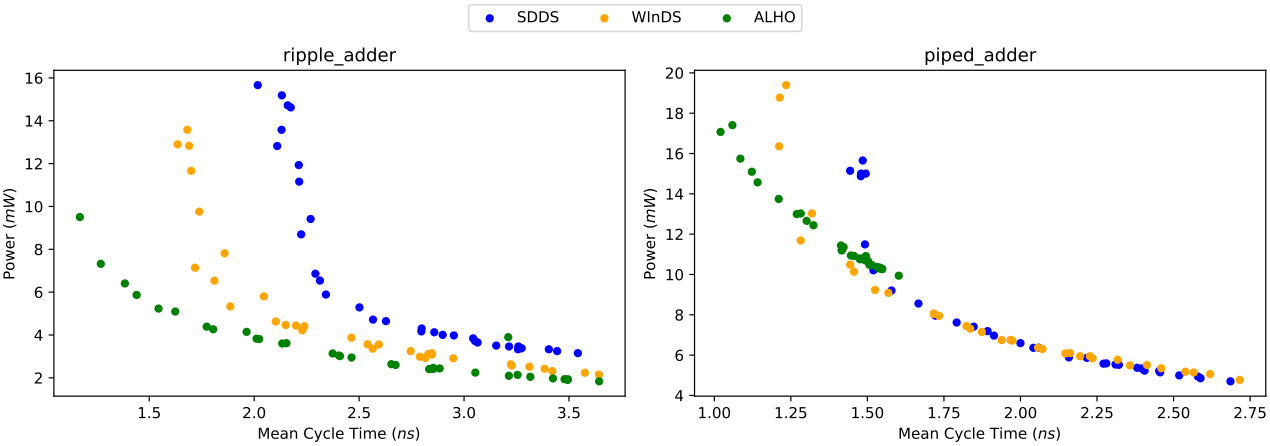
[1] Notice that although the ripple adder is a pure combination circuit, the C-Elements manually instantiate on the ALHO implementation to convert from and to the RTL-protocol account here as sequential gates, whereas the equivalent C-elements on WiNDS are inferred, thus account as combinational gates.



(a) Combined power, area and cycle time figures.

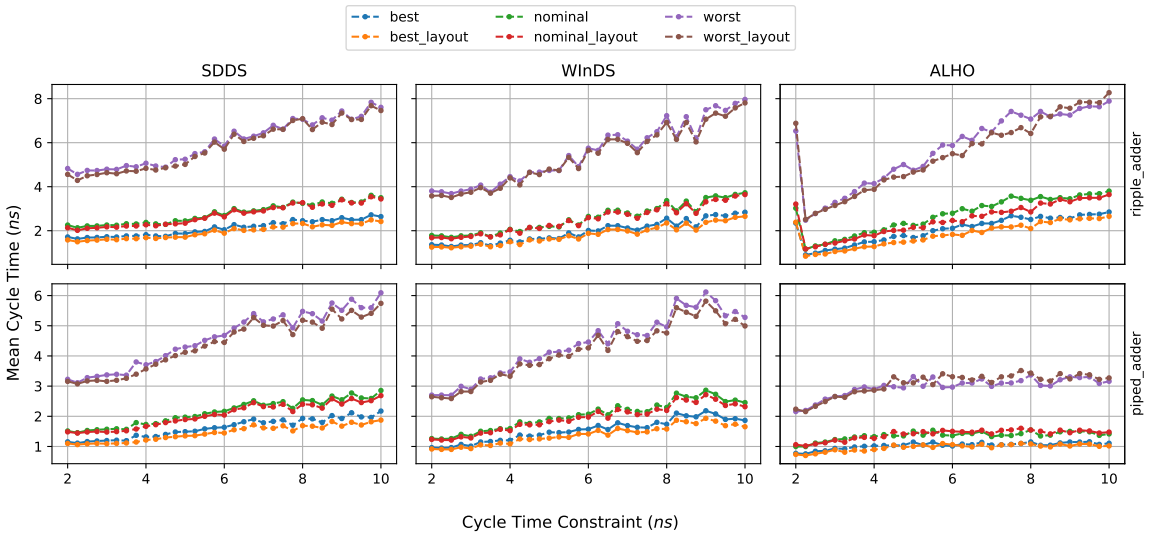


(b) Detailed figures of area vs mean cycle time.

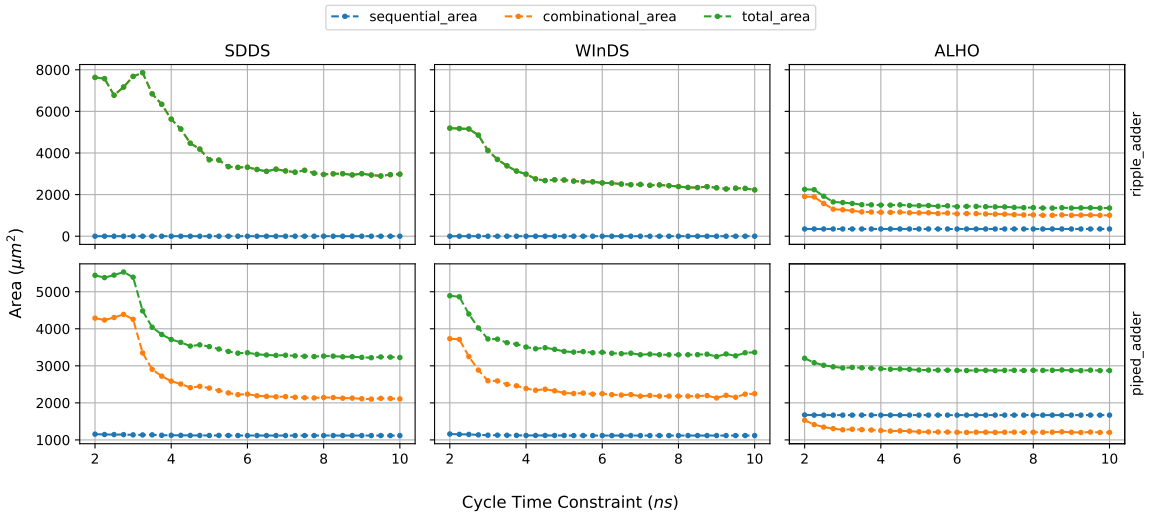


(c) Detailed figures of power vs mean cycle time.

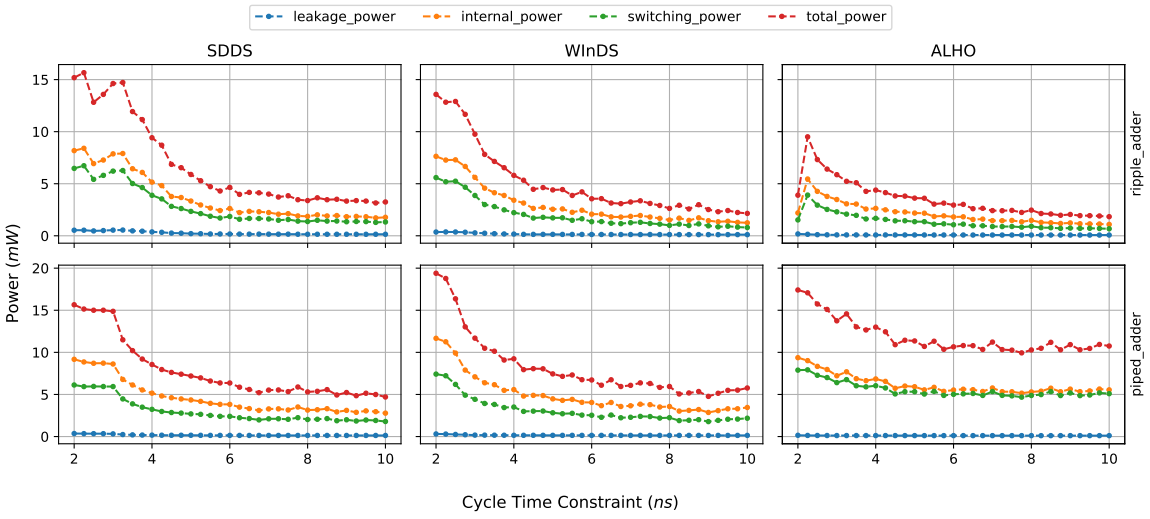
Figure 6.7: PPA results for the adder designs. Power and cycle-time results extracted from nominal corner post-layout simulations.



(a) Observed mean cycle time.



(b) Gate area results.



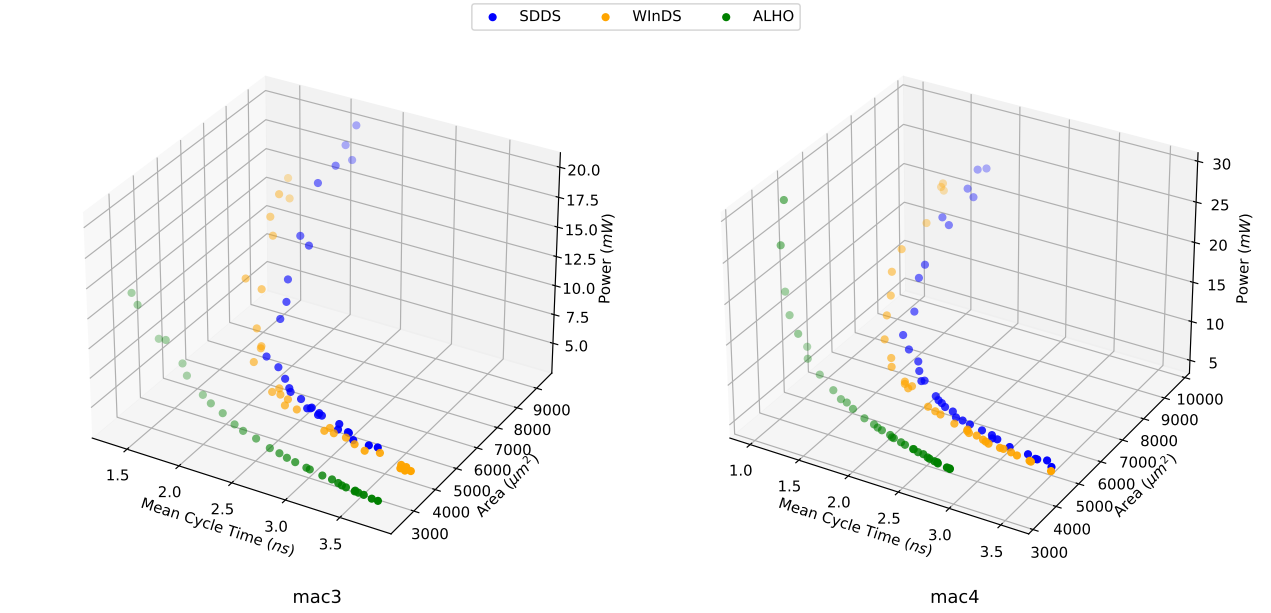
(c) Power extracted from the nominal corner.

Figure 6.8: Effect of the cycle-time constraint on the PPA of the adder designs.

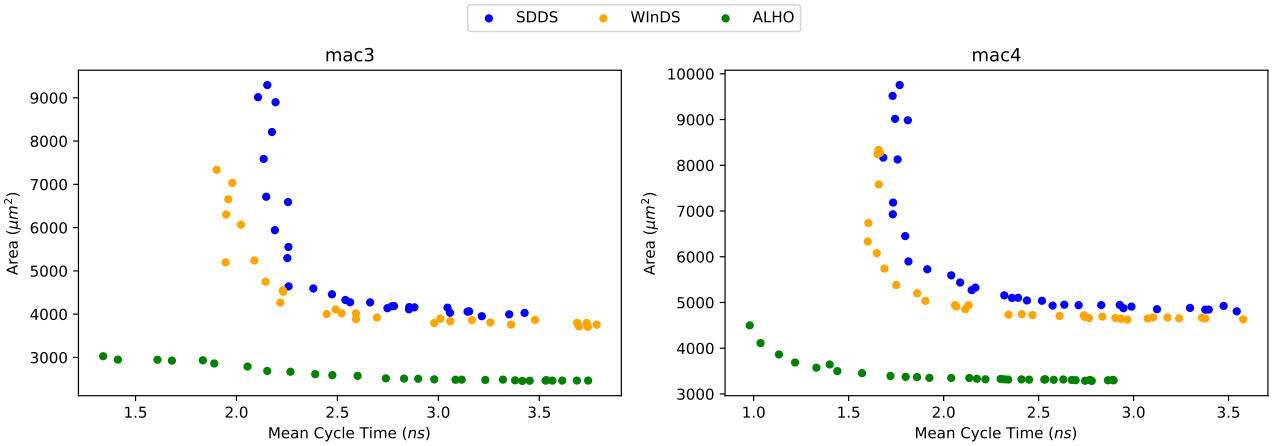
sign that, although achieving a faster cycle time with less peak power, the power figures of the ALHO implementations are slightly higher for the same cycle times observed in the other templates. Also, similar to the ripple adder, the ALHO slowest 6-stage MAC is significantly faster than the slowest implementations on the other templates for the same constraint range. This indicates that the ALHO implementation has a lot of free slack, and with a deeper pipeline, the power employed on sequential logic and the completion trees dominates. Since ALHO dual-rail registers are more complex and the completion trees present double the number of gates compared to WInDS, albeit simpler, the energy spent switching the additional gates on sequential-dominated circuits overcomes the gains of using convention gates.

This effect occurs even more strongly on the booth multipliers. Figure 6.11 presents the PPA figures for the 5- and 6-stage booth multipliers. This circuit takes 32 internal cycles to complete computation, and it employs token steering components to selectively handshake with the environment. Therefore, the cycle time measured in simulation is the time between two computations and not the time of the worst cycle internally. Sequential components dominate these two designs; in this scenario, the power efficiency of ALHO is the worst of the three templates. Surprisingly, against all expectations, in these circuits, PS-SDDS-NCL presents the best power efficiency. Also, WInDS presents the worst area efficiency overall, and surprisingly, PS-SDDS-NCL presents the best area efficiency on its slowest implementations of the 6-stage design. However, ALHO still presents the best performance, presenting the fastest cycle time in both designs.

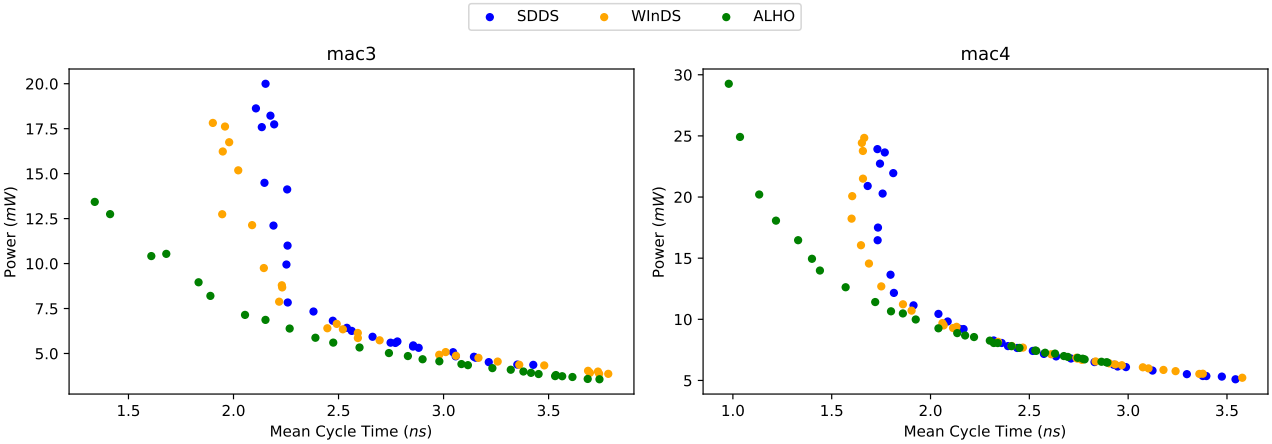
Lastly, to conclude this Section, Figure 6.12 depicts ARV's PPA results. On this complex circuit, ALHO produced circuits with the smallest area and highest performance but with the worst power efficiency. Similar to what was previously observed in most sequential-dominated circuits. Also, WInDS fared slightly worst than PS-SDDS-NCL in terms of both area and power on ARV.



(a) Combined power, area and cycle time figures.

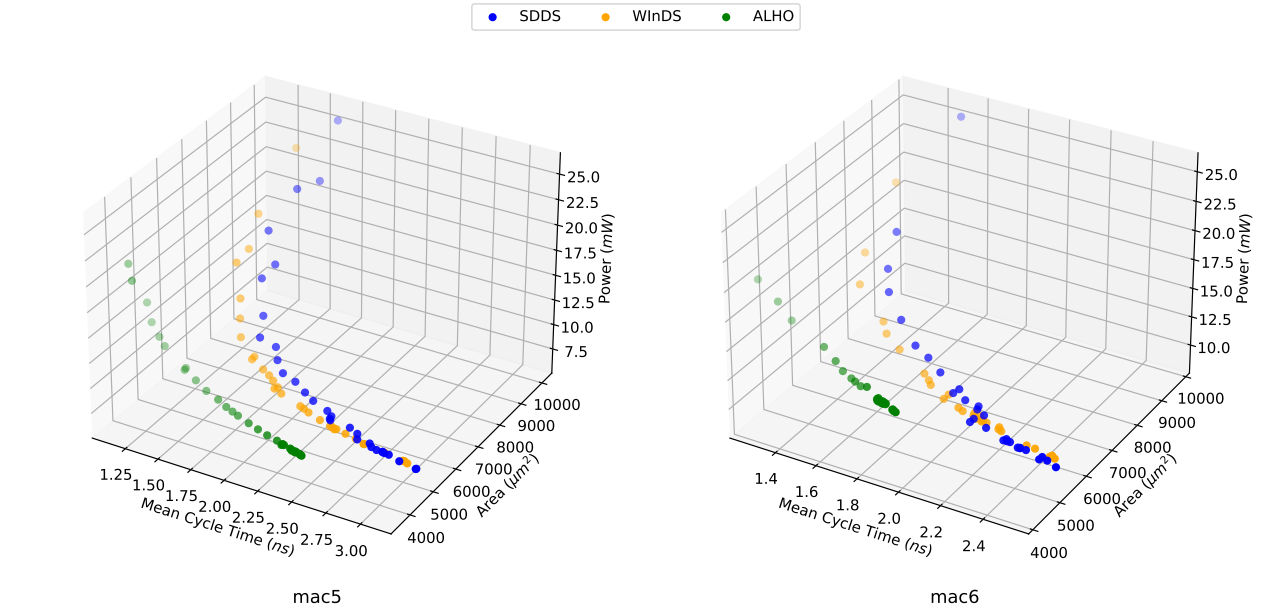


(b) Detailed figures of area vs mean cycle time.

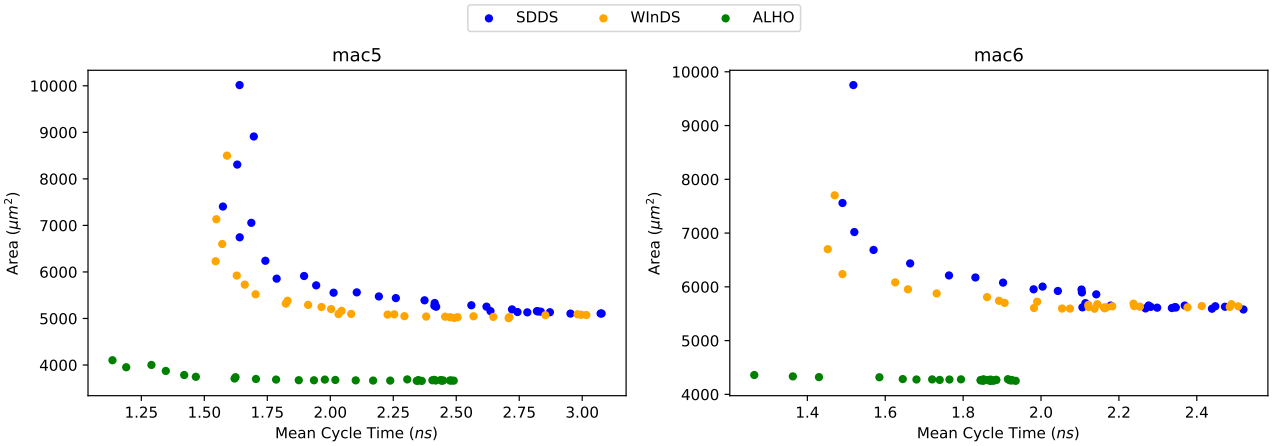


(c) Detailed figures of power vs mean cycle time.

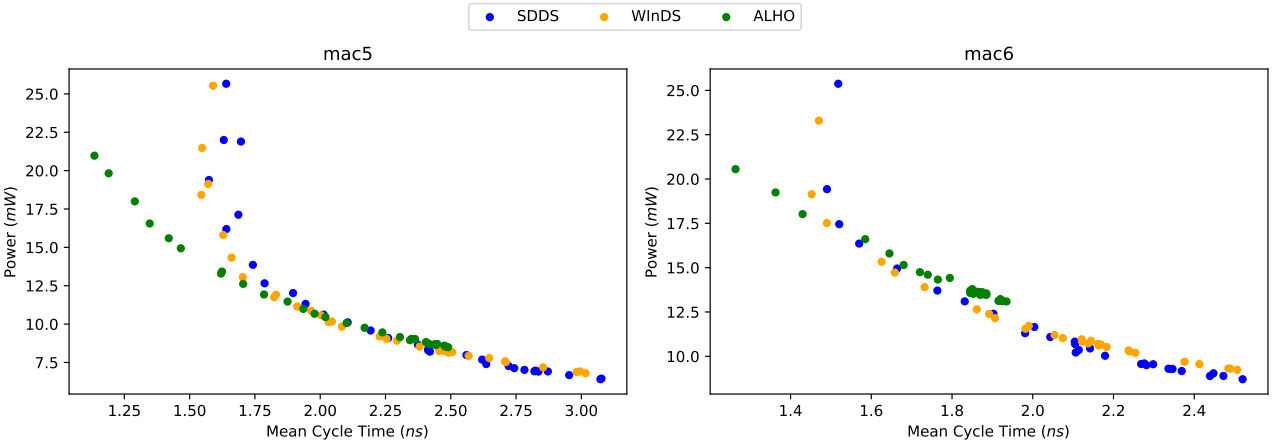
Figure 6.9: Power, performance and area results for the 3- and 4-stage multiply-accumulate designs. Power and cycle-time results extracted from nominal corner post-layout simulations.



(a) Combined power, area and cycle time figures.

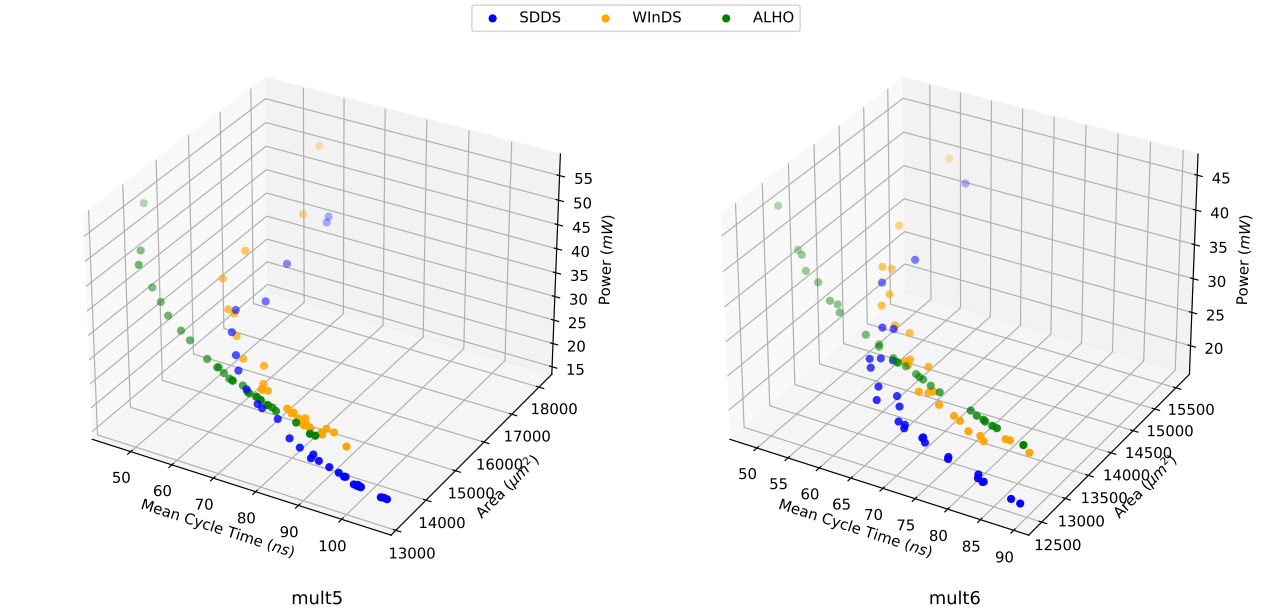


(b) Detailed figures of area vs mean cycle time.

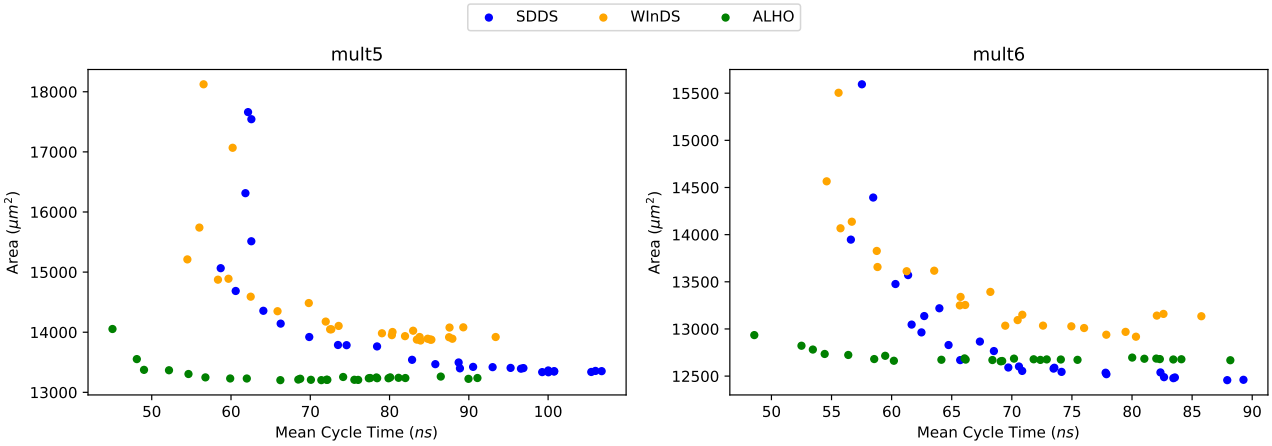


(c) Detailed figures of power vs mean cycle time.

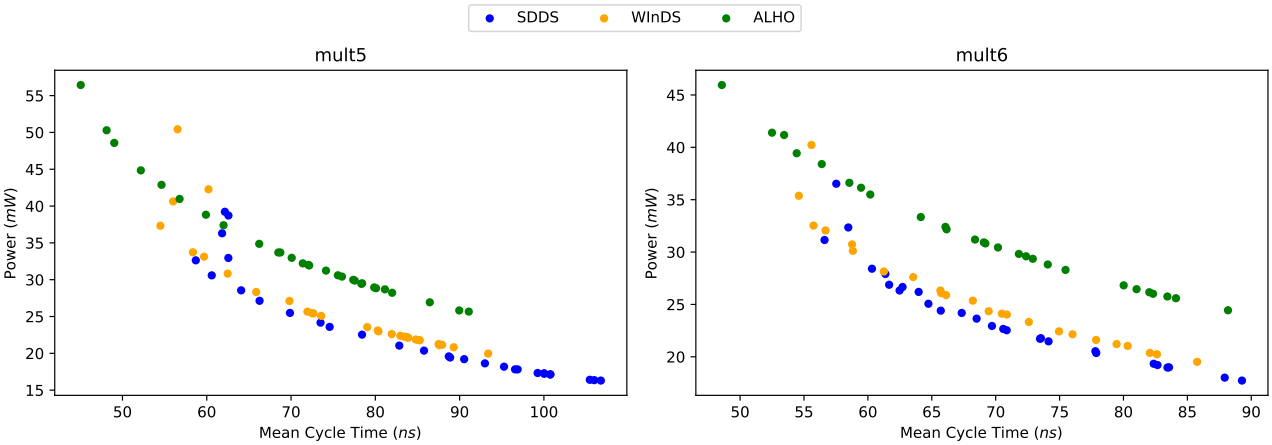
Figure 6.10: Power, performance and area results for the 5- and 6-stage multiply-accumulate designs. Power and cycle-time results extracted from nominal corner post-layout simulations.



(a) Combined power, area and cycle time figures.

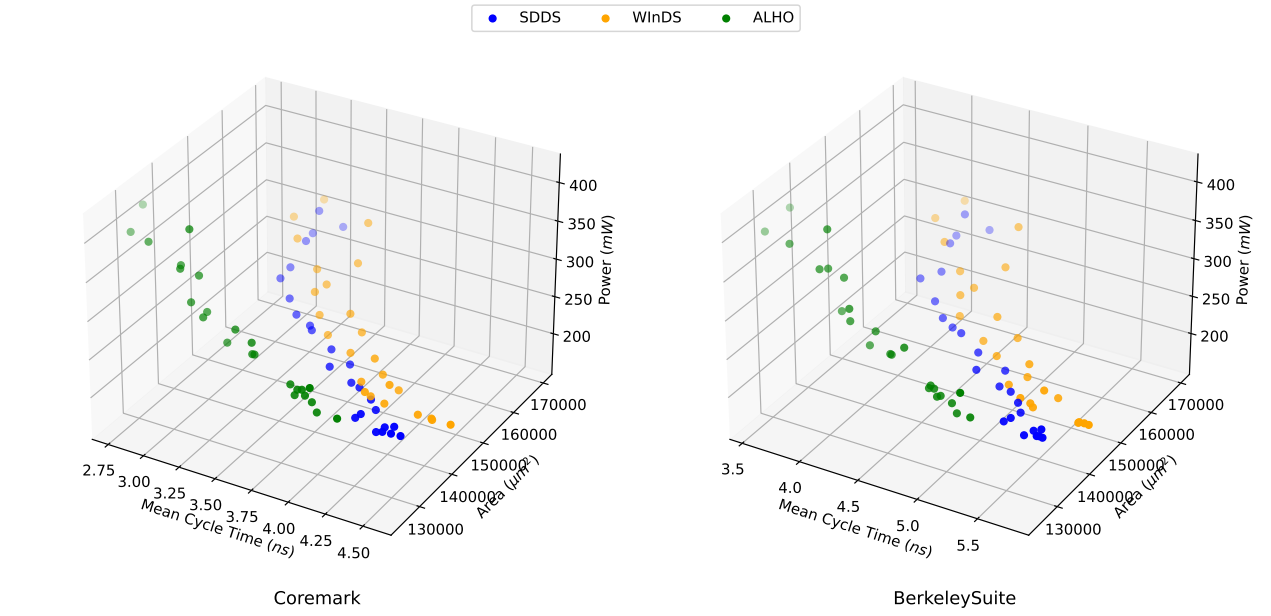


(b) Detailed figures of area vs mean cycle time.

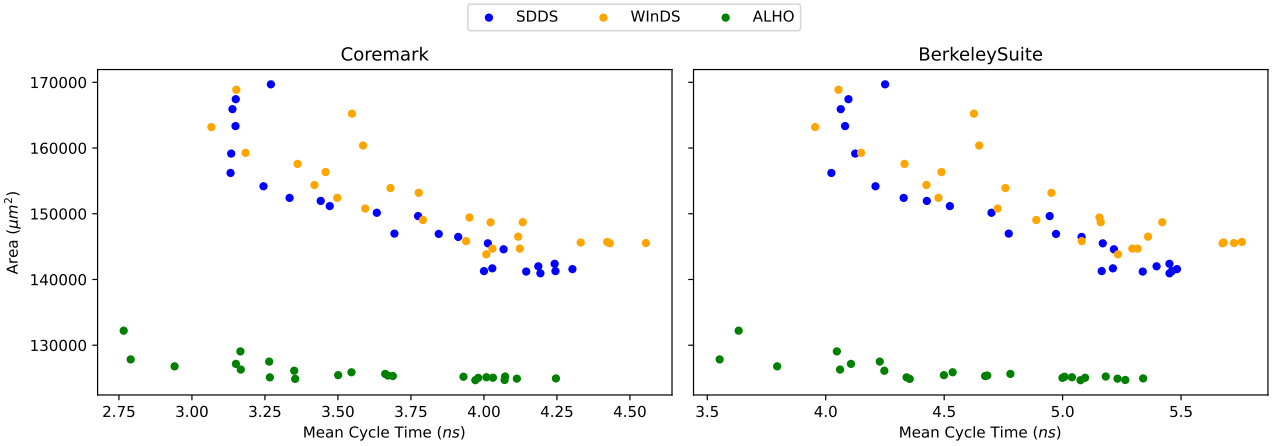


(c) Detailed figures of power vs mean cycle time.

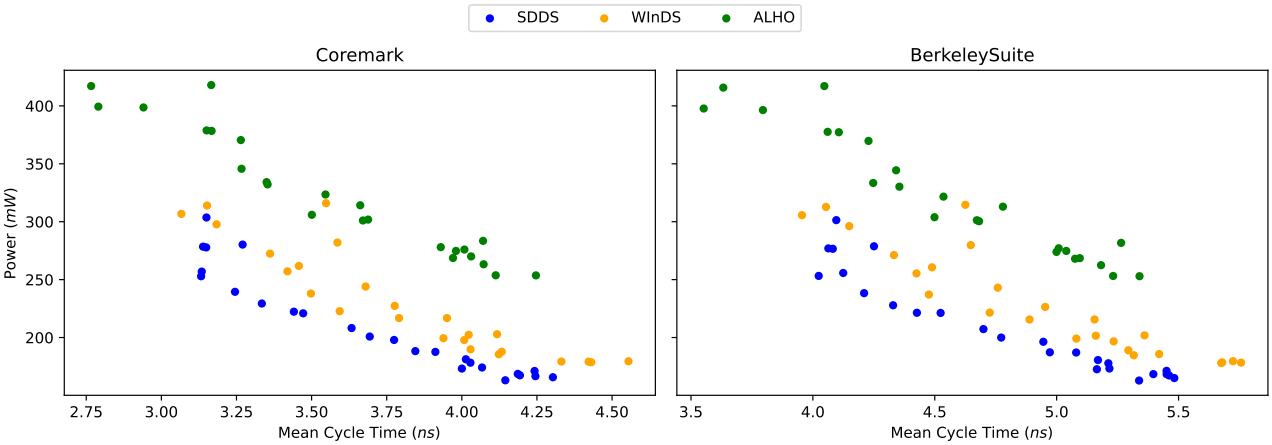
Figure 6.11: Power, performance and area results for the 5- and 6-stage booth multiplier designs. Power and cycle-time results extracted from nominal corner post-layout simulations.



(a) Combined power, area and cycle time figures.



(b) Detailed figures of area vs mean cycle time.



(c) Detailed figures of power vs mean cycle time.

Figure 6.12: Power, performance and area results for ARV, stratified by test program. Power and cycle-time results extracted from nominal corner post-layout simulations.

6.3 Final Remarks and Future Work

The results show some of the advantages of employing conventional gates in the automated synthesis ST circuits with the proposed ALHO template. ALHO consistently shows the potential of producing faster ST circuits compared to PS-SDDS-NCL and WInDS. It also presented a tendency to produce area-efficient circuits, sauf exceptions. Also, ALHO advantages may extend to power efficiency if the circuit is predominantly combinational. Its ability to deal well with combinational logic is no surprise, given that the initial intention behind the proposal of using weakly-indicating logic blocks of conventional gates was to cheapen combination logic. However, originally inconspicuous to the author, using an AND and an OR reduction tree instead of a single C-element tree seems to increase the switching activity on heavily pipelined circuits, consequently reducing its power efficiency.

WInDS, the other proposed template, has shown limited potential in its current form. It presents little to no clear advantage over its predecessor, PS-SDDS-NCL. WInDS acted more as a “stepping stone” in the author’s exploration. However, it might be key to reducing the power consumption of ALHO. A merge of the two templates, employing the hysteretic completion trees from WInDS with the non-hysteretic combinational logic blocks from ALHO, might aid in reducing the switching activity. This, of course, would come at some area and perhaps performance expense. Also, in the current state of Pulsar and with the use of virtual functions to instantiate hysteretic gates, it would be somehow tricky to limit the use of hysteretic gates to only the completion trees. It is left as future work exploring the viability and benefits of this merged template.

A significant issue hinted at but not solved in this thesis is how to guarantee the correct operation of ST circuits in the presence of orphans. Section 3.8 hypothesise a constraint sufficient to guarantee the correct operation of ST circuits comprising handshake channels in the presence of any kind of orphans. However, despite some shreds of evidence of its veracity, this hypothesis remains to be tested. Since the proposed constraint involves clearly defined linear propagation paths, it might be possible to use the pseudo-synchronous model and STA tools to delay annotate an HBCN model, possibly in the signoff phase. From the delay-annotated HBCN, it could be possible to derive `min_delay` constraints to be applied to the design, guaranteeing the correct operation of the circuits, at least on the characterised corners. It is also left as future work proofing the hypothesised timing assumption and developing the tools to correctly constrain it.

Even after accounting for the proper treatment of orphans, the robustness of the proposed templates is still not validated. This would require extensive analogue simulation with random local and global variability. This is tricky since the timing sensitivities tend to show on more complex circuits, not small-scale, low-complexity designs friendly to analogue simulation. Some weak evidence in the form of having validated PS-SDDS-NCL circuits built

with Pulsar to work with both random variability and voltage scaling exists. This is especially interesting because the method employed in synthesising the evaluated PS-SDDS-NCL circuits is susceptible to producing gate orphans. Evaluating the robustness of the PS-SDDS-NCL circuit with orphans gives a glimpse of what to expect from other circuits with orphans. This is not definitive, as the length and number of these orphans are also expected to impact the circuit's robustness. Evaluating the robustness of the new proposed templates is also the subject of possible future work.

REFERENCES

[BC09] Bhasker, J.; Chadha, R. “Static Timing Analysis for Nanometer Designs: A Practical Approach”. New York, NY: Springer, 2009, chap. Appendix A - SDC, pp. 447–466, available at https://link.springer.com/content/pdf/bbm:978-0-387-93820-2/1?pdf=chapter toc.

[BCV+05] Beigné, E.; Clermidy, F.; Vivet, P.; A., C.; Renaudin, M. “An Asynchronous NOC Architecture Providing Low Latency Service and its Multi-level Design Framework”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2005, pp. 54–65.

[BE09] Brej, C.; Edwards, D. “Forward and Backward Guarding in Early Output Logic”. In: IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2009, pp. 226–229.

[BG03] Brej, C.; Garside, J. “Early output logic using anti-tokens”. In: International Workshop on Logic Synthesis, 2003, pp. 302–309.

[BOF10] Beerel, P. A.; Ozdag, R. O.; Ferretti, M. “A Designer´s Guide to Asynchronous VLSI”. Cambridge University Press, 2010, 353p.

[Bre05] Brej, C. “Early Output Logic and Anti-tokens”, Ph.D. Thesis, The University of Manchester (United Kingdom), 2005, 137p.

[CLRS09] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. “Introduction to Algorithms”. The MIT Press, 2009, 3rd ed., 1292p.

[DLD+14] Davies, M.; Lines, A.; Dama, J.; Gravel, A.; Southworth, R.; Dimou, G.; Beerel, P. “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.

[DMM04] Donno, M.; Macii, E.; Mazzoni, L. “Power-aware clock tree planning”. In: International Symposium on Physical Design (ISPD), 2004, pp. 138–147.

[EB02] Edwards, D.; Bardsley, A. “Balsa: An Asynchronous Hardware Synthesis Language”, *The Computer Journal*, vol. 45–1, 2002, pp. 12–18.

[EMB02] EMBC. “Coremark”. Source: https://www.eembc.org/coremark/, 2023-08-10.

[Fan05] Fant, K. M. “Logically Determined Design: Clockless System Design with NULL Convention Logic”. Wiley, 2005, 292p.

[FB96] Fant, K.; Brandt, S. “NULL Convention Logic^{TM}: a complete and consistent logic for asynchronous digital circuit synthesis”. In: International Conference on Application Specific Systems, Architectures and Processors (ASAP), 1996, pp. 261–273.

[Hec18] Heck, G. “The Impact of Voltage Scaling over Delay Elements with Focus on Post-Silicon Tests”, Ph.D. Thesis, PPGCC - FACIN - Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), 2018, 149p.

[HHS+15] Heck, G.; Heck, L.; Singhvi, A.; Moreira, M. T.; Beerel, P.; Calazans, N. L. V. “Analysis and Optimization of Programmable Delay Elements for 2-Phase Bundled-Data Circuits”. In: International Conference on VLSI Design (VLSID), 2015, pp. 321–326.

[HLPM20] Hua, W.; Lu, Y.-S.; Pingali, K.; Manohar, R. “Cyclone: A static timing and power engine for asynchronous circuits”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2020, pp. 11–19.

[Hur69] Hurst, S. “An Introduction to Threshold Logic: A survey of present theory and practice”, *The Radio and Electronic Engineer*, vol. 37–6, 1969, pp. 339–351.

[KKM09] Keller, S.; Katelman, M.; Martin, A. J. “A necessary and sufficient timing assumption for speed-independent circuits”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2009, pp. 65–76.

[Mag84] Magott, J. “Performance Evaluation of Concurrent Systems using Petri Nets”, *Information Processing Letters*, vol. 18–1, 1984, pp. 7–13.

[Mar90] Martin, A. J. “The limitations to delay-insensitivity in asynchronous circuits”. In: MIT Conference on Advanced Research in VLSI (AUSCRYPT), 1990, pp. 263–278.

[MBSC18] Moreira, M. T.; Beerel, P. A.; Sartori, M. L. L.; Calazans, N. L. V. “NCL Synthesis With Conventional EDA Tools: Technology Mapping and Optimization”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65–6, 2018, pp. 1981–1993.

[MGC12a] Moreira, M. T.; Guazzelli, R. A.; Calazans, N. L. V. “Return-to-One DIMS Logic on 4-phase m-of-n Asynchronous Circuits”. In: IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2012, pp. 669–672.

[MGC12b] Moreira, M. T.; Guazzelli, R. A.; Calazans, N. L. V. “Return-to-One Protocol for Reducing Static Power in QDI Circuits Employing m-of-n Codes”. In: Symposium on Integrated Circuits and Systems Design (SBCCI), 2012, pp. 1–6.

[MN06] Martin, A.; Nyström, M. “Asynchronous Techniques for System-on-Chip Design”, *Proceedings of the IEEE*, vol. 94–6, 2006, pp. 1089–1120.

[MNM+14] Moreira, M. T.; Neutzling, A.; Martins, M.; Reis, A. I.; Ribas, R. P.; Calazans, N. L. V. “Semi-custom NCL Design with Commercial EDA Frameworks: Is it possible?” In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2014, pp. 53–60.

[MNW03] Martin, A. J.; Nyström, M.; Wong, C. G. “Three Generations of Asynchronous Microprocessors”, *IEEE Design and Test of Computers*, vol. 20–6, 2003, pp. 9–17.

[MOPC13] Moreira, M. T.; Oliveira, C. H. M.; Porto, R. C.; Calazans, N. L. V. “NCL+: Return-to-one Null Convention Logic”. In: IEEE International Midwest Symposium on Circuii rod Systems (MWSCAS), 2013, pp. 836–839.

[Mor16] Moreira, M. T. “Asynchronous Circuits: Innovations in Components, Cell Libraries and Design Templates”, Ph.D. Thesis, PPGCC - FACIN - Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), 2016, 276p.

[MPC14] Moreira, M. T.; Pontes, J. H.; Calazans, N. L. V. “Tradeoffs between RTO and RTZ in WCHB QDI Asynchronous Design”. In: International Symposium on Quality Electronic Design (ISQED), 2014, pp. 692–699.

[MTMC14] Moreira, M. T.; Trojan, G.; Moraes, F. G.; Calazans, N. L. V. “Spatially Distributed Dual-Spacer Null Convention Logic Design”, *Journal of Low Power Electronics*, vol. 10–3, 2014, pp. 313–320.

[Mur89] Murata, T. “Petri nets: Properties, analysis and applications”, *Proceedings of the IEEE*, vol. 77–4, 1989, pp. 541–580.

[NSMC23] Nunes, W. A.; Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “Validating an Automated Asynchronous Synthesis Environment with a Challenging Design: RISC-V”. In: Symposium on Integrated Circuits and Systems Design (SBCCI), 2023, pp. Accepted for publication.

[OB02] Ozdag, R.; Beerel, P. “High-speed qdi asynchronous pipelines”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2002, pp. 13–22.

[PMMC10] Pontes, J. H.; Moreira, M. T.; Moraes, F. G.; Calazans, N. L. V. “Hermes-AA: A 65nm asynchronous NoC router with adaptive routing”. In: IEEE International System on Chip Conference (SoCC), 2010, pp. 493–498.

[Rei13] Reisig, W. “Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies”. Springer, 2013, 230p.

[RIS15] RISC-V Foundation. “Unit Tests for RISC-V Processors”. Source: https://github.com/riscv/riscv-tests, 2017-06-01.

[RST12] Reese, R. B.; Smith, S. C.; Thornton, M. A. “Uncle - An RTL Approach to Asynchronous Design”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2012, pp. 65–72.

[Sar17] Sartori, M. L. L. “ARV: Towards an Asynchronous Implementation of the RISC-V Architecture”. Source: http://www.inf.pucrs.br/~calazans/publications/2017_MarcosSartori_EoTW.pdf, jul 2017.

[Sar19] Sartori, M. L. L. “PULSAR: Towards a Synthesis Flow for QDI Circuits”, Master’s Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, PPGCC, Porto Alegre, Brazil, 2019, 195p.

[Sin81] Singh, N. P. “A Design Methodology for Self-Timed Systems”, Master’s Thesis, Massachussets Institute of Technology, Laboratory for Computer Science (EEMCS), Cambridge, MA, 1981, 98p.

[SMC20a] Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2020, pp. 114–123.

[SMC20b] Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “ASCEnD-FreePDK45 - A Free Standard Cell Library for SDDS-NCL Circuits”. Source: https://github.com/marlls1989/ascend-freepdk45, jun 2020.

[SMC20c] Sartori, M. L. L.; Moreira, M. T.; Calazans, N. L. V. “Pulsar - A Flow to Support the Design of QDI Asynchronous Circuits”. Source: https://github.com/marlls1989/pulsar, jun 2020.

[SMT+15] Singhvi, A.; Moreira, M. T.; Tadros, R.; Calazans, N. L. V.; Beerel, P. A. “A Fine-Grained, Uniform, Energy-Efficient Delay Element for FD-SOI Technologies”. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2015, pp. 27–32.

[SMT+17] Singhvi, A.; Moreira, M. T.; Tadros, R.; Calazans, N. L. V.; Beerel, P. A. “A Fine-Grained, Uniform, Energy-Efficient Delay Element for 2-Phase Bundled-Data Circuits”, *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13–2, jan 2017, pp. 1–23.

[SN07] Singh, M.; Nowick, S. M. “MOUSETRAP: High-speed transition-signaling asynchronous pipelines”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15–6, jun 2007, pp. 684–698.

[SNC22] Sartori, M. L. L.; Nunes, W. A.; Calazans, N. L. V. “Enhancing an asynchronous circuit design flow to support complex digital system design”. In: Symposium on Integrated Circuits and Systems Design (SBCCI), 2022, pp. 1–6.

[SNC23] Sartori, M. L. L.; Nunes, W. A.; Calazans, N. L. V. “Components to support choice in self-timed asynchronous design flows”, *Journal of Integrated Circuits and Systems*, vol. 18–1, 2023, pp. 1–12.

[Spa20] Sparsø, J. “Introduction to Asynchronous Circuit Design”. Independently published, 2020, 269p.

[Sut89] Sutherland, I. “Micropipelines”, *Communications of the ACM*, vol. 32–6, 1989, pp. 720–738.

[SWM+20] Sartori, M. L. L.; Wuerdig, R. N.; Moreira, M. T.; Bampi, S.; Calazans, N. L. V. “Leveraging QDI Robustness to Simplify the Design of IoT Circuits”. In: IEEE International Symposium on Circuits and Systems (ISCAS), 2020, pp. 1–4.

[SWMC19] Sartori, M. L. L.; Wuerdig, R. N.; Moreira, M. T.; Calazans, N. L. V. “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.

[TBV12] Thonnart, Y.; Beigné, E.; Vivet, P. “A Pseudo-Synchronous Implementation Flow for WCHB QDI Asynchronous Circuits”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2012, pp. 73–80.

[Tei06] Teifel, J. “Asynchronous Cryptographic Hardware Design”. In: Annual IEEE International Carnahan Conference on Security Technology (ICCST), 2006, pp. 221–227.

[THG+16] Tadros, R.; Hua, W.; Gibiluka, M.; Moreira, M. T.; Calazans, N. L. V.; Beerel, P. A. “Analysis and Design of Delay Lines for Dynamic Voltage Scaling Applications”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2016, pp. 11–18.

[Ver88] Verhoeff, T. “Delay-Insensitive Codes - An Overview”, *Distributed Computing*, vol. 3–1, mar 1988, pp. 1–8.

[WSA+22] Wuerdig, R. N.; Sartori, M. L. L.; Abreu, B. A.; Bampi, S.; Calazans, N. L. V. “Mitigating asynchronous qdi drawbacks on mac operators with approximate multipliers”. In: IEEE International Symposium on Circuits and Systems (ISCAS), 2022, pp. 1269–1273.

APPENDIX A – LIST OF THE AUTHOR PREVIOUS PUBLICATIONS RELATED TO THIS THESIS

1. NUNES, W. A.; SARTORI, M. L. L.; MOREIRA, M. T.; MORAES, F. G.; CALAZANS, N. L. V. “Validating an Automated Asynchronous Synthesis Environment with a Challenging Design: RISC-V”. In: Symposium on Integrated Circuits And Systems Design (SBCCI), Rio de Janeiro, August-September 2023. To be presented.
2. SARTORI, M. L. L.; NUNES, W. A.; CALAZANS, N. L. V. “Components to Support Choice in Dual-Rail Asynchronous Design Flows”. Journal of Integrated Circuits and Systems (JICS), 18(1), May 2023, 11p.
3. CALAZANS, N. L. V.; SARTORI, M. L. L. “Asynchronous Circuit Principles and a Survey of Associated Design Tools”. Journal of Integrated Circuits and Systems (JICS), 17(3), Dec 2022, 10p.
4. LODÉA, N.; NUNES, W. A.; SARTORI, M. L. L.; OST, L. C.; CALAZANS, N. L. V.; GARIBOTTI, R. F.; MARCON, C. A. M. “Early Soft Error Reliability Analysis on RISC-V”. IEEE Latin America Transactions, 20(9), September 2022, pp. 2139–2145.
5. NUNES, W. A.; SARTORI, M. L. L.; CALAZANS, N. L. V. “Enhancing an Asynchronous Circuit Design Flow to Support Complex Digital System Design”. In: Symposium on Integrated Circuits and Systems Design (SBCCI), Pelotas, August 2022, 6p.
6. WUERDIG, R. N; SARTORI, M. L. L.; ABREU, B.; BAMPI, S.; CALAZANS, N. L. V. “Mitigating Asynchronous QDI Drawbacks on MAC Operators with Approximate Multipliers”. In: IEEE International Symposium on Circuits and Systems (ISCAS), Austin, May 2022, pp 1269-1273.
7. CALAZANS, N. L. V.; RODOLFO, T. A.; SARTORI, M. L. L. “Robust and Energy-Efficient Hardware: The Case for Asynchronous Design”. Journal of Integrated Circuits and Systems (JICS), 16(2), September 2021. pp.1-11.
8. RODOLFO, T. A.; SARTORI, M. L. L.; MOREIRA, M. T.; CALAZANS, N. L. V. “Quasi Delay Insensitive FIFOs: Design Choices Exploration and Comparison”. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS), Daegu, May 2021, 5pp.
9. SARTORI, M. L. L.; MOREIRA, M. T.; CALAZANS, N. L. V. “A Frontend using Traditional EDA Tools for the Pulsar QDI Design Flow”. In: IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Snowbird, May 2020, pp. 3-10.
10. SARTORI, M. L. L.; WUERDIG, R. N.; MOREIRA, M. T.; BAMPI, S.; CALAZANS, N. L. V. “Leveraging QDI Robustness to Simplify the Design of IoT Circuits”. In: IEEE International Symposium on Circuits and Systems (ISCAS), Seville. October 2020, 5p.

11. SARTORI, M. L. L. “PULSAR: Towards a Synthesis flow for QDI Circuits”. MSc Dissertation, PPGCC - FACIN - PUCRS, Porto Alegre, Brazil. August 2019. 193p. (Presented and Approved. Research Advisor: Ney Laert Vilar Calazans, Co-Advisor: Matheus Trevisan Moreira).

12. SARTORI, M. L. L.; WUERDIG, R. N.; MOREIRA, M. T.; CALAZANS, N. L. V. “Pulsar: Constraining QDI Circuits Cycle Time Using Traditional EDA Tools”. In: IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Hirosaki, 2019, pp. 114-123.

13. WUERDIG, R. N.; SARTORI, M. L. L.; CALAZANS, N. L. V. “Asynchronous Quasi-Random Number Generator: Taking Advantage of PVT Variations”. In: IEEE Latin American Symposium on Circuits and Systems (LASCAS), Armenia, February 2019, pp. 137-140.

14. MOREIRA, M. T.; BEEREL, P. A.; SARTORI, M. L. L.; CALAZANS, N. L. V. “NCL Synthesis with Conventional EDA Tools: Technology Mapping and Optimization”. IEEE Transactions on Circuits and Systems I - Regular Papers, 65(6), June 2018, pp. 1981-1993.

15. SARTORI, M. L. L.; CALAZANS, N. L. V. “Go Functional Model for a RISC-V Asynchronous Organization - ARV”. In: IEEE International Conference on Electronics, Circuits and Systems (ICECS), Batumi, December 2017, pp. 381-384.

16. SARTORI, M. L. L. “ARV: Towards an Asynchronous Implementation of the RISC-V Architecture”. Computer Engineering Bachelor Thesis. Computer Engineering - PUCRS, July 2017. 57p. (Presented and approved. Advisor: Ney Laert Vilar Calazans).

APPENDIX B – BENCHMARKS' RTL-LIKE SOURCE CODE

Listing B.1: rtl/ripple_adder.sv

```
1  module ripple_adder #(WIDTH=32)
2    (input  logic clk, reset,
3     input  logic [WIDTH-1:0] a, b,
4     input  logic             c,
5     output logic [WIDTH:0]   out);
6   logic [WIDTH:0] carry;
7
8   assign carry[0] = c;
9
10  for (genvar i = 0; i < WIDTH ; i++) begin
11    assign out[i] = a[i] ^ b[i]  ^ carry[i];
12    assign carry[i+1] =  carry[i] & (a[i] | b[i]) | a[i] & b[i];
13  end
14
15  assign out[WIDTH] = carry[WIDTH];
16
17 endmodule
```

Listing B.2: rtl/piped_adder.sv

```
1  module ripple_adder #(WIDTH=32)
2    (input  logic clk, reset,
3     input  logic [WIDTH-1:0] a, b,
4     input  logic             c,
5     output logic [WIDTH:0]   out);
6   logic [WIDTH:0] carry;
7
8   assign carry[0] = c;
9
10  for (genvar i = 0; i < WIDTH ; i++) begin
11    assign out[i] = a[i] ^ b[i]  ^ carry[i];
12    assign carry[i+1] =  carry[i] & (a[i] | b[i]) | a[i] & b[i];
13  end
14
15  assign out[WIDTH] = carry[WIDTH];
16
17 endmodule
18
19 module piped_adder #(DEPTH=4, WIDTH=8)
20   (input logic clk, reset,
21    input logic  [WIDTH*DEPTH-1:0] a, b,
22    input logic                    c,
23    output logic [WIDTH*DEPTH:0]   out);
```

```
24
25    logic [WIDTH-1:0] a_pipe[DEPTH*(DEPTH+1)/2];
26    logic [WIDTH-1:0] b_pipe[DEPTH*(DEPTH+1)/2];
27    logic [WIDTH:0] out_pipe[DEPTH*(DEPTH+1)/2];
28    logic [DEPTH:0]   c_pipe;
29
30    assign c_pipe[0] = c;
31
32    for (genvar y = 0; y < DEPTH; y++) begin
33      assign a_pipe[y*(y+1)/2] = a[y*WIDTH +: WIDTH];
34      assign b_pipe[y*(y+1)/2] = b[y*WIDTH +: WIDTH];
35      assign out[y*WIDTH +: WIDTH] = out_pipe[(DEPTH-y)*(DEPTH-1-y)/2][WIDTH-1:0];
36      ripple_adder #(WIDTH) adder (.clk, .reset, .a(a_pipe[y*(y+1)/2+y]),
            .b(b_pipe[y*(y+1)/2+y]), .c(c_pipe[y]),
            .out(out_pipe[(DEPTH-y)*(DEPTH-1-y)/2+DEPTH-1-y]));
37      always_ff @(posedge clk)
38        c_pipe[y+1] <= out_pipe[(DEPTH-y)*(DEPTH-1-y)/2+DEPTH-1-y][WIDTH];
39
40      for (genvar x = 1; x<=y; x++)
41        always_ff @(posedge clk) begin
42          a_pipe[y*(y+1)/2+x] <= a_pipe[y*(y+1)/2+x-1];
43          b_pipe[y*(y+1)/2+x] <= b_pipe[y*(y+1)/2+x-1];
44        end
45
46      for (genvar x = DEPTH-1-y; x > 0; x--)
47        always_ff @(posedge clk)
48          out_pipe[(DEPTH-y)*(DEPTH-1-y)/2+x-1] <=
                out_pipe[(DEPTH-y)*(DEPTH-1-y)/2+x];
49    end
50
51    assign out[DEPTH*WIDTH] = out_pipe[0][WIDTH];
52
53  endmodule
```

Listing B.3: rtl/mac3.sv

```
1  module mac3 #(WIDTH=16)
2     (input logic [(WIDTH/2)-1:0] a,
3      input logic [(WIDTH/2)-1:0] b,
4      input logic                 clk, reset,
5      output logic [WIDTH+(WIDTH/2)-1:0]  out);
6     logic [(WIDTH/2)-1:0]       reg_a;
7     logic [(WIDTH/2)-1:0]       reg_b;
8     logic [WIDTH+(WIDTH/2)-1:0]          acc;
9     logic [WIDTH+(WIDTH/2)-1:0]          prod;
10    logic [WIDTH+(WIDTH/2)-1:0]          result;
11
12    assign result = prod + acc;
```

```
13
14      always @(posedge clk) begin
15          reg_a <= a;
16          reg_b <= b;
17          prod <= reg_a * reg_b;
18          out   <= result;
19      end
20
21      always @(posedge clk or negedge reset)
22          if (!reset)
23            acc <= '0;
24          else
25            acc <= result;
26
27  endmodule // mac3
```

Listing B.4: rtl/mac4.sv

```
1   module mac4 #(WIDTH=16)
2       (input logic [(WIDTH/2)-1:0] a,
3        input logic [(WIDTH/2)-1:0] b,
4        input logic                 clk, reset,
5        output logic [WIDTH+(WIDTH/2)-1:0]  out);
6       logic [(WIDTH/2)-1:0]       reg_a;
7       logic [(WIDTH/2)-1:0]       reg_b;
8       logic [WIDTH+(WIDTH/2)-1:0]          result;
9       logic [WIDTH+(WIDTH/2)-1:0]          prod;
10      logic [WIDTH+(WIDTH/2)-1:0]          acc;
11
12      always @(posedge clk) begin
13          reg_a <= a;
14          reg_b <= b;
15          prod <= reg_a * reg_b;
16          result <= prod + acc;
17          out <= result;
18      end
19
20      always @(posedge clk or negedge reset)
21          if (!reset)
22            acc <= '0;
23          else
24            acc <= result;
25
26  endmodule // mac4
```

Listing B.5: rtl/mac5.sv

```
1   module mac5 #(WIDTH=16)
2       (input logic [(WIDTH/2)-1:0] a,
```

```
3      input logic [(WIDTH/2)-1:0] b,
4      input logic                 clk, reset,
5      output logic [WIDTH+(WIDTH/2)-1:0]  out);
6     logic [(WIDTH/2)-1:0]        reg_a;
7     logic [(WIDTH/2)-1:0]        reg_b;
8     logic [WIDTH+(WIDTH/2)-1:0]          result[2];
9     logic [WIDTH+(WIDTH/2)-1:0]          acc;
10    logic [WIDTH+(WIDTH/2)-1:0]          prod;
11
12    always @(posedge clk) begin
13       reg_a <= a;
14       reg_b <= b;
15       prod <= reg_a * reg_b;
16       result[0] <= prod + acc;
17       result[1] <= result[0];
18       out <= result[1];
19    end
20
21    always @(posedge clk or negedge reset)
22       if (!reset)
23         acc <= '0;
24       else
25         acc <= result[1];
26
27 endmodule // mac5
```

Listing B.6: rtl/mac6.sv

```
1  module mac6 #(WIDTH=16)
2     (input logic [(WIDTH/2)-1:0] a,
3      input logic [(WIDTH/2)-1:0] b,
4      input logic                 clk, reset,
5      output logic [WIDTH+(WIDTH/2)-1:0]  out);
6     logic [(WIDTH/2)-1:0]        reg_a;
7     logic [(WIDTH/2)-1:0]        reg_b;
8     logic [WIDTH+(WIDTH/2)-1:0]          result[3];
9     logic [WIDTH+(WIDTH/2)-1:0]          acc;
10    logic [WIDTH+(WIDTH/2)-1:0]          prod;
11
12    always @(posedge clk) begin
13       reg_a <= a;
14       reg_b <= b;
15       prod <= reg_a * reg_b;
16       result[0] <= prod + acc;
17       result[1] <= result[0];
18       result[2] <= result[1];
19       out <= result[2];
20    end
```

```
21
22      always @(posedge clk or negedge reset)
23          if (!reset)
24            acc <= '0;
25          else
26            acc <= result[2];
27
28  endmodule // mac6
```

Listing B.7: rtl/mult3.sv

```
 1  module mult_control #(WIDTH)
 2      (input logic  clk, reset,
 3       output logic accept_output, accept_input
 4       );
 5      logic [$clog2(WIDTH-1)-1:0] counter, counter_next;
 6      typedef enum              bit[1:0] {LOAD, OPERATE, UNLOAD} state_t;
 7      state_t                   ps, ns;
 8
 9      // Control FSM
10      always_ff @(posedge clk or negedge reset)
11        if (!reset)
12          ps <= LOAD;
13        else
14          ps <= ns;
15
16      always_ff @(posedge clk)
17        case (ps)
18          LOAD    : ns <= OPERATE;
19          OPERATE :
20            if (!counter)
21              ns <= UNLOAD;
22            else
23              ns <= OPERATE;
24          UNLOAD : ns <= LOAD;
25          default : ns <= ps;
26        endcase // case (ps)
27
28      always_comb begin
29          accept_input <= ps == LOAD ? 1'b1 : 1'b0;
30          accept_output <= ps == UNLOAD ? 1'b1 : 1'b0;
31      end
32
33      // Counter
34      always_ff @(posedge clk or negedge reset)
35        if (!reset)
36          counter <= (WIDTH/2)-2;
37        else
```

```
38          counter <= counter_next;
39
40      always_ff @(posedge clk)
41        if (ps == OPERATE)
42          counter_next <= counter-1;
43        else
44          counter_next <= (WIDTH/2)-2;
45
46  endmodule // mult_control
47
48  module mult3 #(WIDTH=64)
49      (input logic clk, reset,
50       input logic [WIDTH/2-1:0] a, b,
51       output logic [WIDTH-1:0]  result);
52      logic [WIDTH/2-1:0]       a_r, b_r, a_next;
53      logic [WIDTH/2-1:0]       upper, a_comp;
54      logic [WIDTH:0]           acc, acc_r;
55      logic                     accept_input, accept_output;
56
57      // Control FSM
58      mult_control #(WIDTH) ctrl (.*);
59
60      // Input A conditional handshake
61      for (genvar i = 0 ; i < WIDTH/2 ; i++) begin
62         discard discard_a (.a(a_next[i]), .en(!accept_input), .q(a_r[i]), .*);
63         hold hold_a (.a(a[i]), .en(accept_input), .q(a_r[i]), .*);
64      end
65
66      always_ff @(posedge clk or negedge reset)
67        if (!reset)
68          a_next <= '0;
69        else
70          a_next <= a_r;
71
72      // Input B conditional handshake
73      discard discard_acc (.a(acc[0]), .en(!accept_input), .q(acc_r[0]), .*);
74      condlo null_acc (.en(accept_input), .q(acc_r[0]), .*);
75      for (genvar i = 1 ; i < WIDTH/2+1 ; i++) begin
76         discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
77         hold hold_b (.a(b[i-1]), .en(accept_input), .q(acc_r[i]), .*);
78      end
79      for (genvar i = WIDTH/2+1 ; i < WIDTH+1 ; i++) begin
80         discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
81         condlo null_acc (.en(accept_input), .q(acc_r[i]), .*);
82      end
83
84      // Computation loop
85      always_comb
```

```
86        begin
87          case (acc_r[1:0])
88            2'b01 : a_comp <= a_r;
89            2'b10 : a_comp <= ~(a_r)+1;
90            default: a_comp <= '0;
91          endcase
92        end
93
94      assign upper = acc_r[WIDTH:WIDTH/2+1] + a_comp;
95
96
97      always_ff @(posedge clk or negedge reset)
98        if (!reset)
99          acc <= '0;
100       else
101         acc <= {upper[WIDTH/2-1], upper, acc_r[WIDTH/2:1]};;
102
103     // Output Conditional Handshake
104     for (genvar i = 0 ; i < WIDTH ; i++)
105       discard discard_out (.a(acc[i+1]), .en(accept_output), .q(result[i]), .*);
106
107 endmodule // mult
```

Listing B.8: rtl/mult4.sv

```
1  module mult_control #(WIDTH)
2      (input logic  clk, reset,
3       output logic accept_output, accept_input
4       );
5      logic [$clog2(WIDTH-1)-1:0] counter, counter_next;
6      typedef enum              bit[1:0] {LOAD, OPERATE, UNLOAD} state_t;
7      state_t                   ps, ns;
8
9      // Control FSM
10     always_ff @(posedge clk or negedge reset)
11       if (!reset)
12         ps <= LOAD;
13       else
14         ps <= ns;
15
16     always_ff @(posedge clk)
17       case (ps)
18         LOAD    : ns <= OPERATE;
19         OPERATE :
20           if (!counter)
21             ns <= UNLOAD;
22           else
23             ns <= OPERATE;
```

```
24          UNLOAD : ns <= LOAD;
25          default : ns <= ps;
26        endcase // case (ps)
27
28      always_comb begin
29          accept_input <= ps == LOAD ? 1'b1 : 1'b0;
30          accept_output <= ps == UNLOAD ? 1'b1 : 1'b0;
31      end
32
33      // Counter
34      always_ff @(posedge clk or negedge reset)
35        if (!reset)
36          counter <= (WIDTH/2)-2;
37        else
38          counter <= counter_next;
39
40      always_ff @(posedge clk)
41        if (ps == OPERATE)
42          counter_next <= counter-1;
43        else
44          counter_next <= (WIDTH/2)-2;
45
46  endmodule // mult_control
47
48  module mult4 #(WIDTH=64)
49      (input logic clk, reset,
50       input logic [WIDTH/2-1:0] a, b,
51       output logic [WIDTH-1:0]  result);
52      logic [WIDTH/2-1:0]       a_r, b_r, a_next;
53      logic [WIDTH/2-1:0]       upper, a_comp;
54      logic [WIDTH:0]           acc, acc_r, acc_next;
55      logic                     accept_input, accept_output;
56
57      // Control FSM
58      mult_control #(WIDTH) ctrl (.*);
59
60      // Input A conditional handshake
61      for (genvar i = 0 ; i < WIDTH/2 ; i++) begin
62          discard discard_a (.a(a_next[i]), .en(!accept_input), .q(a_r[i]), .*);
63          hold hold_a (.a(a[i]), .en(accept_input), .q(a_r[i]), .*);
64      end
65      always_ff @(posedge clk or negedge reset)
66        if (!reset)
67          a_next <= '0;
68        else
69          a_next <= a_r;
70
71      // Input B conditional handshake
```

```
72     discard discard_acc (.a(acc[0]), .en(!accept_input), .q(acc_r[0]), .*);
73     condlo null_acc (.en(accept_input), .q(acc_r[0]), .*);
74     for (genvar i = 1 ; i < WIDTH/2+1 ; i++) begin
75        discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
76        hold hold_b (.a(b[i-1]), .en(accept_input), .q(acc_r[i]), .*);
77     end
78     for (genvar i = WIDTH/2+1 ; i < WIDTH+1 ; i++) begin
79        discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
80        condlo null_acc (.en(accept_input), .q(acc_r[i]), .*);
81     end
82
83     // Computation loop
84     always_comb
85       begin
86         case (acc_r[1:0])
87           2'b01 : a_comp <= a_r;
88           2'b10 : a_comp <= ~(a_r)+1;
89           default: a_comp <= '0;
90         endcase
91       end
92
93     assign upper = acc_r[WIDTH:WIDTH/2+1] + a_comp;
94
95    always_ff @(posedge clk)
96      acc_next <= {upper[WIDTH/2-1], upper, acc_r[WIDTH/2:1]};;
97
98
99     always_ff @(posedge clk or negedge reset)
100      if (!reset)
101        acc <= '0;
102      else
103         acc <= acc_next;
104
105    // Output Conditional Handshake
106    for (genvar i = 0 ; i < WIDTH ; i++)
107      discard discard_out (.a(acc[i+1]), .en(accept_output), .q(result[i]), .*);
108
109 endmodule // mult
```

Listing B.9: rtl/mult5.sv

```
1  module mult_control #(WIDTH)
2     (input logic  clk, reset,
3      output logic accept_output, accept_input
4      );
5     logic [$clog2(WIDTH-1)-1:0] counter, counter_next;
6     typedef enum             bit[1:0] {LOAD, OPERATE, UNLOAD} state_t;
7     state_t                  ps, ns;
```

```
 8
 9     // Control FSM
10     always_ff @(posedge clk or negedge reset)
11       if (!reset)
12         ps <= LOAD;
13       else
14         ps <= ns;
15
16     always_ff @(posedge clk)
17       case (ps)
18         LOAD    : ns <= OPERATE;
19         OPERATE :
20           if (!counter)
21             ns <= UNLOAD;
22           else
23             ns <= OPERATE;
24         UNLOAD : ns <= LOAD;
25         default : ns <= ps;
26       endcase // case (ps)
27
28     always_comb begin
29        accept_input <= ps == LOAD ? 1'b1 : 1'b0;
30        accept_output <= ps == UNLOAD ? 1'b1 : 1'b0;
31     end
32
33     // Counter
34     always_ff @(posedge clk or negedge reset)
35       if (!reset)
36         counter <= (WIDTH/2)-2;
37       else
38         counter <= counter_next;
39
40     always_ff @(posedge clk)
41       if (ps == OPERATE)
42         counter_next <= counter-1;
43       else
44         counter_next <= (WIDTH/2)-2;
45
46 endmodule // mult_control
47
48 module mult5 #(WIDTH=64)
49     (input logic clk, reset,
50      input logic [WIDTH/2-1:0] a, b,
51      output logic [WIDTH-1:0]  result);
52     logic [WIDTH/2-1:0]       a_r, b_r, a_next;
53     logic [WIDTH/2-1:0]       upper, a_comp;
54     logic [WIDTH:0]           acc, acc_r, acc_next[2];
55     logic                     accept_input, accept_output;
```

```
56
57      // Control FSM
58      mult_control #(WIDTH) ctrl (.*);
59
60      // Input A conditional handshake
61      for (genvar i = 0 ; i < WIDTH/2 ; i++) begin
62         discard discard_a (.a(a_next[i]), .en(!accept_input), .q(a_r[i]), .*);
63         hold hold_a (.a(a[i]), .en(accept_input), .q(a_r[i]), .*);
64      end
65      always_ff @(posedge clk or negedge reset)
66        if (!reset)
67          a_next <= '0;
68        else
69          a_next <= a_r;
70
71      // Input B conditional handshake
72      discard discard_acc (.a(acc[0]), .en(!accept_input), .q(acc_r[0]), .*);
73      condlo null_acc (.en(accept_input), .q(acc_r[0]), .*);
74      for (genvar i = 1 ; i < WIDTH/2+1 ; i++) begin
75         discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
76         hold hold_b (.a(b[i-1]), .en(accept_input), .q(acc_r[i]), .*);
77      end
78      for (genvar i = WIDTH/2+1 ; i < WIDTH+1 ; i++) begin
79         discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
80         condlo null_acc (.en(accept_input), .q(acc_r[i]), .*);
81      end
82
83      // Computation loop
84      always_comb
85        begin
86          case (acc_r[1:0])
87            2'b01 : a_comp <= a_r;
88            2'b10 : a_comp <= ~(a_r)+1;
89            default: a_comp <= '0;
90          endcase
91        end
92
93      assign upper = acc_r[WIDTH:WIDTH/2+1] + a_comp;
94
95     always_ff @(posedge clk) begin
96       acc_next[1] <= {upper[WIDTH/2-1], upper, acc_r[WIDTH/2:1]};
97       acc_next[0] <= acc_next[1];
98     end
99
100
101     always_ff @(posedge clk or negedge reset)
102       if (!reset)
103         acc <= '0;
```

```
104       else
105          acc <= acc_next[0];
106
107     // Output Conditional Handshake
108     for (genvar i = 0 ; i < WIDTH ; i++)
109       discard discard_out (.a(acc[i+1]), .en(accept_output), .q(result[i]), .*);
110
111 endmodule // mult
```

Listing B.10: rtl/mult6.sv

```
 1 module mult_control #(WIDTH)
 2     (input logic  clk, reset,
 3      output logic accept_output, accept_input
 4      );
 5     logic [$clog2(WIDTH-1)-1:0] counter, counter_next;
 6     typedef enum              bit[1:0] {LOAD, OPERATE, UNLOAD} state_t;
 7     state_t                   ps, ns;
 8
 9     // Control FSM
10     always_ff @(posedge clk or negedge reset)
11       if (!reset)
12         ps <= LOAD;
13       else
14         ps <= ns;
15
16     always_ff @(posedge clk)
17       case (ps)
18         LOAD    : ns <= OPERATE;
19         OPERATE :
20           if (!counter)
21             ns <= UNLOAD;
22           else
23             ns <= OPERATE;
24         UNLOAD : ns <= LOAD;
25         default : ns <= ps;
26       endcase // case (ps)
27
28     always_comb begin
29        accept_input <= ps == LOAD ? 1'b1 : 1'b0;
30        accept_output <= ps == UNLOAD ? 1'b1 : 1'b0;
31     end
32
33     // Counter
34     always_ff @(posedge clk or negedge reset)
35       if (!reset)
36         counter <= (WIDTH/2)-2;
37       else
```

```
38          counter <= counter_next;
39
40      always_ff @(posedge clk)
41        if (ps == OPERATE)
42          counter_next <= counter-1;
43        else
44          counter_next <= (WIDTH/2)-2;
45
46  endmodule // mult_control
47
48  module mult6 #(WIDTH=64)
49      (input logic clk, reset,
50       input logic [WIDTH/2-1:0] a, b,
51       output logic [WIDTH-1:0]  result);
52      logic [WIDTH/2-1:0]       a_r, b_r, a_next;
53      logic [WIDTH/2-1:0]       upper, a_comp;
54      logic [WIDTH:0]           acc, acc_r, acc_next[3];
55      logic                     accept_input, accept_output;
56
57      // Control FSM
58      mult_control #(WIDTH) ctrl (.*);
59
60      // Input A conditional handshake
61      for (genvar i = 0 ; i < WIDTH/2 ; i++) begin
62         discard discard_a (.a(a_next[i]), .en(!accept_input), .q(a_r[i]), .*);
63         hold hold_a (.a(a[i]), .en(accept_input), .q(a_r[i]), .*);
64      end
65      always_ff @(posedge clk or negedge reset)
66        if (!reset)
67          a_next <= '0;
68        else
69          a_next <= a_r;
70
71      // Input B conditional handshake
72      discard discard_acc (.a(acc[0]), .en(!accept_input), .q(acc_r[0]), .*);
73      condlo null_acc (.en(accept_input), .q(acc_r[0]), .*);
74      for (genvar i = 1 ; i < WIDTH/2+1 ; i++) begin
75         discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
76         hold hold_b (.a(b[i-1]), .en(accept_input), .q(acc_r[i]), .*);
77      end
78      for (genvar i = WIDTH/2+1 ; i < WIDTH+1 ; i++) begin
79         discard discard_acc (.a(acc[i]), .en(!accept_input), .q(acc_r[i]), .*);
80         condlo null_acc (.en(accept_input), .q(acc_r[i]), .*);
81      end
82
83      // Computation loop
84      always_comb
85        begin
```

```
86          case (acc_r[1:0])
87            2'b01 : a_comp <= a_r;
88            2'b10 : a_comp <= ~(a_r)+1;
89            default: a_comp <= '0;
90          endcase
91        end
92
93      assign upper = acc_r[WIDTH:WIDTH/2+1] + a_comp;
94
95     always_ff @(posedge clk) begin
96       acc_next[2] <= {upper[WIDTH/2-1], upper, acc_r[WIDTH/2:1]};
97       acc_next[1] <= acc_next[2];
98       acc_next[0] <= acc_next[1];
99     end
100
101
102     always_ff @(posedge clk or negedge reset)
103       if (!reset)
104         acc <= '0;
105       else
106          acc <= acc_next[0];
107
108     // Output Conditional Handshake
109     for (genvar i = 0 ; i < WIDTH ; i++)
110       discard discard_out (.a(acc[i+1]), .en(accept_output), .q(result[i]), .*);
111
112 endmodule // mult
```

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br