TU WIEN Informatics

Comparison of QDI Adders

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Oliver Haschke, BSc
Matrikelnummer 01636305

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Florian Huemer, BSc

Wien, 23. Oktober 2024

Oliver Haschke	Andreas Steininger

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

TU WIEN Informatics

Comparison of QDI Adders

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Oliver Haschke, BSc
Registration Number 01636305

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Florian Huemer, BSc

Vienna, October 23, 2024

Oliver Haschke	Andreas Steininger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Erklärung zur Verfassung der Arbeit

Oliver Haschke, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Oktober 2024

Oliver Haschke

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich möchte an dieser Stelle meinen herzlichen Dank an all jenen aussprechen, die zur Fertigstellung dieser Arbeit beigetragen haben.

Allen voran möchte ich mich bei Andreas Steininger und Florian Huemer für die Gelegenheit zu dieser Arbeit und die Betreuung ebenjener bedanken. Ihre Anleitung bei der Arbeit, Hilfe, sowohl mit dem theoretischen Inhalt der Arbeit, wie auch der praktischen Implementierung und den Eigenheiten der Simulationsumgebung, und auch die regelmäßigen Besprechungen haben es mir ermöglicht die Arbeit erfolgreich abzuschließen.

Auch meiner Familie, im speziellen meinen Eltern, Heinz und Andrea, möchte ich dafür danken, dass sie mir die Chance gegeben haben einen Universitätsabschluss anzustreben, und mich auch während der gesamten Studienzeit dabei unterstützt haben.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Binäre Addierer spielen in der digitalen Datenverarbeitung eine zentrale Rolle, weswegen sehr viel Aufwand in ihre Optimierung gesteckt wurde, vor allem was die Verarbeitungszeit angeht. Diese Optimierungen beziehen sich jedoch zum klar überwiegenden Teil auf synchrone Logik.

Asynchrone Logik bietet im Vergleich dazu bereits eine implizite Optimierungen, da sie, im Gegensatz zur synchronen Logik, wo die Verarbeitungszeit fix auf das ungünstigste denkbare Szenario ausgelegt werden muss, sich auf die gegebenen Bedingungen jeweils mögliche - und daher im Durchschnitt kürzere - Verarbeitungszeit adaptieren kann. Andererseits muss asynchrone Logik dafür andere Abstriche machen, zum Beispiel wird zusätzliche Logik in der Form der Completion Detection (CD) benötigt, und die verwendeten Codierungen können ebenfalls zu einem Mehraufwand an Logik führen.

In dieser Arbeit wurden die Leistung, der Platzbedarf und die Komplexität, gemessen an der Anzahl der Signalleitungen im Design, für eine ausgewählte Gruppe von Addieren quantitativ analysiert, und zwar jeweils in synchroner wie auch in asynchroner Realisierung. Die zugrundeliegenden Simulationsergebnisse wurden mit State of the Art Simulationtools erlangt. Die dafür benötigten Designs wurden auf Basis einer offen verfügbaren Library in VHDL implementiert. Ausführliche Vergleiche der genannten Charakteristiken der Addierer, welche die Stärken und Schwächen der unterschiedlichen Implementationen zeigen, wurden dann anhand dieser Ergebnisse aufgestellt.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Abstract

Binary adders are a core component of most digital processing circuitry, and thus much effort has been put forward to optimize them as much as possible, particularly regarding the performance, as measured by the latency of the adder circuitry. However, all of these optimizations are done within the context of a synchronous logic design.

Asynchronous logic, in comparison, already has the built-in performance gain of achieving average-case performance, rather than the worst-case performance that synchronous logic is limited by. On the other hand, asynchronous logic requires additional circuitry, such as the Completion Detection (CD), and incurs overheads due to the used encoding for example.

This thesis quantitatively analyzes the performance, area usage, and complexity, expressed as the number of nets in the design, of a select group of adder designs, each in a synchronous and an asynchronous variant. These results were obtained by way of simulation using state-of-the-art simulation tools. The designs themselves were implemented in VHDL on the basis of an open cell library. Based on these results extensive comparisons of the noted characteristics were made, which highlight the relative strengths and weaknesses of the implementations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

1.1 Introduction and Background

In his 1996 masters thesis [Lyn96] Thomas W. Lynch noted that there had been over 700 papers related to addition since the 1960s. He also noted the most common aspects of these papers, such as the model used for a logic element or function, or the configuration of the adder. While those adder designs are largely still in use today, the breadth of the aspects of binary addition which are discussed in research papers has increased. In particular modern papers focus on a key aspect that was considered uncommon back then, and that is asynchronous logic.

As with many other applications of asynchronous logic the theoretical performance gain is immediately obvious, as the worst-case performance that all synchronous logic components are bounded by is instead shifted to an average-case performance.

Of course things are never this simple and asynchronous logic comes with both benefits and drawbacks of its own. In particular less common and thus less optimized logic components, such as the Muller C gate or threshold gates must be utilized to meet the requirements that a given asynchronous design style imposes. Furthermore, modern design tools are not built for asynchronous logic, which also hampers the creation of optimized asynchronous circuits.

However, as this introduction already noted there is an abundance of research, much of which is dedicated to finding the most optimal adder designs for various different scenarios and applications. Some of these have been optimized for processing speed, which will be quantified as the delay of the adder circuit and which will be obtained trough simulation, or for the area that the adder occupies when implemented on the substrate.

And with this we come to the main motivation for this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1.2 Motivation

As mentioned in the previous section, there have been many advancements made for binary adders to improve the performance, reduce the needed area or reduce their power consumption. More recent papers focus on implementing half-adder blocks, full-adder blocks and even a full ripple-carry adder in asynchronous logic, utilizing and comparing some of the different, commonly proposed asynchronous design styles.

In principle it makes sense to focus on the ripple-carry adder in particular, since it exhibits the biggest difference between the average- and the worst-case performance. This is further supported by the findings of Von Neumann [VNT63], which note that given uniformly distributed random operands the average length of the longest carry chain is proportional to $log(n)$, where n is the width of the adder in bits. This must be contrasted against the theoretical worst-case performance of the ripple-carry adder which has, depending on the exact configuration, either a longest possible carry chain of length n if subtraction utilizing twos-complement is supported, or $n-1$ if the carry is generated at the first bit-position and then propagated through the remainder of the adder.

Thus, many techniques which seek to improve the performance of the adder focus on reducing the theoretical, longest possible carry chain, which would then allow the given adder to be operated at higher clock frequencies. However, asynchronous implementations already purport average-case performance. Since the improvements for synchronous adder designs were focused on improving the worst-case in particular it remains to be seen if these same techniques can also improve an asynchronous design, which already achieves average-case performance. Furthermore, it remains to be seen whether the cost of these improvements, judged for example by the additional required area, is worthwhile when compared with the asynchronous ripple-carry adder.

1.3 Preliminaries

Before further discussing the contents specific to this thesis the main relevant concepts for this subject matter will be introduced.

1.3.1 Critical path

An important concept for the evaluation in particular of synchronous logic is the critical path. The critical path is the longest possible signal path through the circuit and in the case of synchronous logic its length serves as a lower bound for the clock period, given that any signal must be able to traverse the critical path within one clock period.

In contrast an asynchronous circuit has a data-dependent critical path, meaning that the critical path can be different for different data that is processed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1.3.2 Group Carries

Unlike the ripple-carry adder, which simply calculates each carry as part of the single-bit binary addition and then passes this carry on to the next single-bit adder block, i.e., a Full Adder (FA), modern designs achieve some level of parallelism for the calculation of the sum of the binary addition by calculating the carries ahead of time and as much as possible in parallel.

Of the adders which calculate the carries ahead of time the most basic one is simply called the Carry-Lookahead Adder (CLA). The CLA calculates the single-bit-position carry-generate (g_i), which indicates that this bit-position will generate a carry-out signal ($c_i = 1$), and carry-propagate (p_i), which indicates that this bit-position will generate a carry-out signal only if it receives a carry-in signal ($c_i = c_{i-1}$), signals ahead of time and then passes them on to the adder blocks that calculate the sum for the given bit-position. The generate and propagate signals for the bit at the ith bit-position are calculated as follows:

$$\begin{aligned} g_i &= a_i \wedge b_i \\ p_i &= a_i \oplus b_i \end{aligned} \tag{1.1}$$

where a_i and b_i are the values of the operands at the ith bit-position.

Based on this the carry that is generated at a given bit-position i can be expressed as follows:

$$c_i = g_i \vee (p_i \wedge c_{i-1}) \tag{1.2}$$

This recursive definition can be unrolled to calculate the carry for any bit-position of the addition. However, this creates significant problems regarding the fan-out of the previous gates, in particular the gates or storage elements which supply the least significant bits of the two operands, a_0 and b_0. This is because these two signals will occur in the calculations of all of the carries. Furthermore, for wider adders the logic depth for the calculation of the carry at the most significant bit can also be quite deep.

Thus, to relieve the previous logic or buffer stage and to achieve better parallelism the concept of the group carries is introduced. The group carry signals $G_{i:j}$ and $P_{i:j}$ represent that a group of bit-positions from i down to j will generate a carry at its end, the ith position, or propagate the incoming carry, c_{j-1} to the end, c_i, respectively.

The definition for these group carries is given in Equation (1.3), with the base case given in Equation (1.4). Both of these definitions are taken from Section 10.2.2.2 of [WH11].

$$\begin{aligned} G_{i:j} &= G_{i:k} \vee P_{i:k} \wedge G_{k-1:j} \\ P_{i:j} &= P_{i:k} \wedge P_{k-1:j} \end{aligned} \tag{1.3}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\begin{aligned} G_{i:i} &= G_i = g_i \\ P_{i:i} &= P_i = p_i \end{aligned} \tag{1.4}$$

As the papers which describe such adders show, smaller groups of bits can be calculated in parallel and the groups can then be combined to form the carry signals for larger groups of bits. This is usually accomplished using a tree structure, hence the adders utilizing this techniques are commonly referred to as Parallel Prefix Tree Adders (PPTAs).

1.3.3 Asynchronous Logic

Asynchronous logic plays an important part in this thesis and its main draw, as already mentioned, is the shift from worst-case performance as necessitated by synchronous logic to average-case performance facilitated in practice by some form of handshaking protocol.

This handshaking can be realized in a number of ways, but in general it is based on a request and an acknowledge signal, see Figure 1.1. The request signal signals to the data sink that new data is available for processing, whereas the acknowledge signal notifies the data source that the sink is ready for new data. As the design styles that will be discussed later will show, the specific signals can also be implicit in the data, for example by utilizing an encoding for the data from which the sink can infer that the transmitted data is complete.

The handshaking can be realized in a 2-phase or a 4-phase manner. For the 2-phase protocols each transition of the request signal signals the availability of new data from the source to the sink. The following transition of the acknowledge signal then notifies the source that the sink is ready for new data. In contrast the 4-phase protocols are based on the states of the two signals and not on the transitions. This means that for example a "1" or "true" on the request signal indicates the readiness of the data, and a "true" on the acknowledge signal indicates that the data has been consumed. Naturally, both signals then have to return to zero, "0" or "false", so that they can indicate their respective function again for the next data cycle. Thus, the 2-phase protocols are referred to as Non-Return-to-Zero (NRZ), whereas the 4-phase protocols are referred to as Return-to-Zero (RZ). The phase of the 4-phase protocol where both signals have returned to their idle state is then referred to as a reset or null phase.

While this thesis will only utilize a RZ protocol, which will be introduced later, for completions sake it will be mentioned here that Return-to-One (RO) protocols also exist, which work analogously to the RZ protocols, where instead of to "0" or "false" they return to "1" or "true" for their reset or null phase, and thus indicate their respective function, data readiness and data consumption, when they switch to "0" or "false".

1.3.3.1 Bundled Data (BD)

For the Bundled Data (BD) design style [Sut89], as the name suggests, the handshaking signals, request and acknowledge, are "bundled" together with the data. This means that data is put onto the data signal lines which can also lead through a possible combinational

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

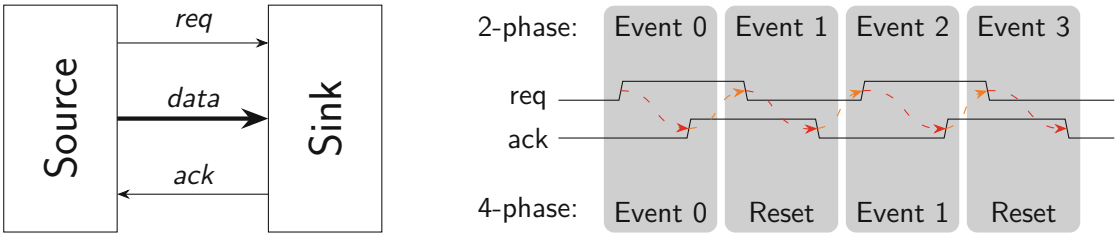

Figure 1.1: Asynchronous handshaking protocols

logic circuit, before arriving at the data sink, which in most cases is the next buffer stage within a processing pipeline. The request signal is then put onto the signal line such that it arrives at the sink only after the data at its inputs is complete and has settled. This implies a timing requirement that is referred to as delay matching, as the specific design implementation has to guarantee this timing assumption by matching the delay of the request line to the delay of the data. This means the delay on the request line has to be matched to the longest possible delay that the data can experience. However, to improve performance variable delays can be used which then adapt to the specific data that is currently being processed.

1.3.3.2 Delay-Insensitive (DI) Logic

Delay-Insensitive (DI) designs rely on a very weak timing assumption in which all wire and gate delays are only assumed to be both positive and also bounded [vdS85]. As long as these two requirements can be guaranteed a circuit implemented as a DI design can be guaranteed to work regardless of the exact value that any delay assumes. However, it has been shown that this timing assumption restricts this class of designs significantly, in particular, because the selection of possible gates to construct a DI circuit is limited to inverters and C gates only [Mar90].

1.3.3.3 Quasi Delay-Insensitive (QDI) Logic

In contrast to the DI designs the class of Quasi Delay-Insensitive designs is expanded by the introduction of the concept of *isochronic forks* [Mar86, Mar90]. This means that for some selected wire forks within the design it must be guaranteed that the signal transitions occur at the same time at all endpoints of the fork, for Figure 1.2 this means that $\delta_2 = \delta_3$. All other delays, in particular the gate delays, are as they would be in a DI design. The introduction of this single relaxation to the otherwise Delay-Insensitive designs allows that arbitrary circuits can be constructed.

1.3.3.4 Encoding

Both DI and QDI designs can encode the request signal directly into the transmitted data utilizing a variety of different styles, although this is only possible if the chosen code satisfies certain properties [Ver88]. One of the simplest encodings that can be used

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

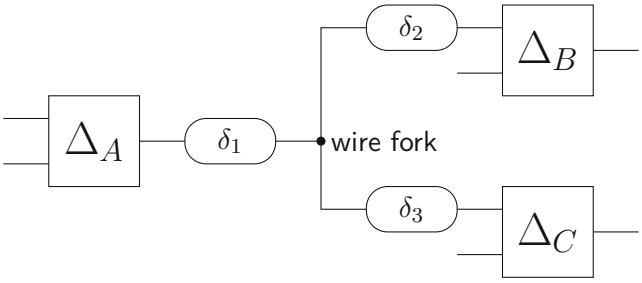

Figure 1.2: Asynchronous circuit model

for this purpose, and also the one this thesis will use for its design implementation, is the dual-rail encoding, where a single logical signal is mapped to two physical wires, referred to as rails, which represent the two possible truth values of the logical signal, i.e., the logical signal x, which can be either "true" or "false" is mapped to the dual-rail signal $x = (x.T, x.F)$, where $x.T$ being "true" represents that x is "true" and $x.F$ being "true" represents x is "false". Such a dual-rail encoding can thus be realized as a RZ or RO encoding, as described in the beginning of this section. However, as previously mentioned, this thesis will utilize the RZ variant.

1.3.3.5 Completion Detection (CD)

Both the RZ (RO) and the NRZ encodings then allow a different component of the design, the so called completion detector, to detect whether data at the inputs is complete. This process is referred to as the Completion Detection (CD). Thus, in place of an explicit request signal, which may experience a race condition versus the data, these designs utilize an implicit request signal that is implied by the data.

As mentioned before, for the 4-phase encoding this process necessitates a null phase, which is either represented by all "false" (all zeros) or all "true" (all ones) on the data lines, to reset the Completion Detection. Otherwise a single transition could lead from one valid data word to the next, and thus be detected as a new valid data word even though the remaining signal transitions have not yet arrived. An example for such a CD for a RZ implementation is given in Figure 1.3.

This further necessitates that the last output of a combinational DI or QDI circuit may only transition after all of its inputs have arrived. In practice it is, however, difficult or costly to always guarantee this directly within the logic design. Thus, it can be advantageous to add an explicit signal, most commonly referred to as a done signal, to the design which guarantees this requirement.

Some of the asynchronous designs within the class of the QDI designs have thus decided to move the CD, which can be a large and comparatively slow sub circuit of the design, off of the data path.

Certain designs can possibly benefit from moving the CD off of the data path, as the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

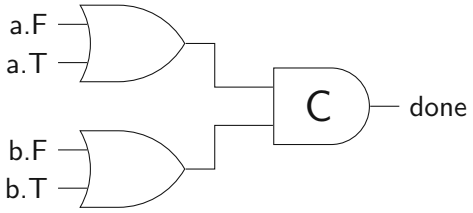

Figure 1.3: CD for a RZ design

explicit CD that is separated from the implementation of the actual logic function could in practice be shared, at least partially, with the Completion Detection of the preceding buffer stage. Furthermore, the done signal of the combinational circuit is used together with the CD of the following buffer stage to generate the acknowledge signal of this buffer, this means there is some possible performance gain as the two CDs now run at least partially in parallel as opposed to fully sequentially. These considerations will become relevant again later when the obtained delay values for the various adder designs will be discussed.

1.4 Research Question

This thesis seeks to answer two related questions regarding the design and implementation of binary addition circuits in asynchronous logic. First, *(a)* it should be evaluated whether asynchronous designs can at all benefit from the techniques that were developed to improve the performance of synchronous binary addition, in particular the parallel-prefix-tree method will be evaluated for this purpose. Second, *(b)* these asynchronous implementations should be compared against an asynchronous implementation of the RCA and be evaluated regarding the performance and area usage relative to the asynchronous RCA.

1.4.1 Asynchronous Adder Implementation

This part of the thesis focuses on the implementation of the candidate adder architectures in an asynchronous, particularly a QDI design style, and their subsequent evaluation through simulation. This process can be subdivided into three main components:

- Selection of candidate adder architectures
  For this purpose a total of 4 architectures that purport some differing benefits and drawbacks, and exhibit various levels of logic complexity, will be identified.
- Implementation as QDI designs
  The selected adders will be implemented as NCL with explicit CD (NCLX) designs.
- Evaluation through simulation
  The finished designs will then be evaluated in both performance and area usage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The performance will be judged both in terms of the delay that the given adder exhibits, from the application of the operands at the inputs until the correct result is available at the outputs, and also through static analysis of the signal paths in the design. The area will be reported by the design tool.

Based on this methodology the following research question can be answered:

RQ1 Can the same patterns of relative performance improvements that are present in synchronous adder architectures also be observed in asynchronous implementations when the same design techniques are applied?

1.4.2 Comparison with the RCA

As previously mentioned the theoretical improvement of the performance of a RCA when implemented as an asynchronous design, due to the shift from worst-case to average-case performance, is well supported in the research. However, it remains to be seen whether any possible performance gain of the asynchronous adder implementations is worth the potential cost in terms of circuit complexity and area usage.

While the matter of area usage can be answered in a straightforward manner, since the total area usage will be reported by the design tool, the matter of circuit complexity comes with some additional considerations. The additional complexity that the various adder architectures have compared to the RCA has a sizable impact when translated to the asynchronous designs.

Based on these considerations this thesis will answer the following additional question regarding the comparison of the various asynchronous adder designs with the RCA:

RQ2 What is the cost of the more complex adder architectures when compared to the RCA, as expressed through area usage and circuit complexity?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Related Work

This chapter will focus on introducing the related work which has already been done on the topics of adder comparison, particularly the comparison of synchronous and asynchronous adders, whether about comparing the same base architecture implemented in both design styles, or comparing novel architectures with established ones.

2.1 Performance Comparison of Asynchronous Adders

The 1994 paper [FP94] by Franklin *et al.* compares various well known adder designs. It gives an overview of the statistical distribution of instructions performed in a typical RISC processor (DLX) and finds that 72% of instructions carry out an addition or subtraction in the data path. Based on this fact it can be reasoned that the adder has a disproportionate influence on the overall system performance, as it is one of the most frequently utilized modules of the processor.

Due to the work from [VNT63] we know that the worst-case, that is a carry bit propagating through a chain of length n for an n-bit addition is unlikely to happen. In fact the mean length of a carry propagation for such an addition is bounded by $log_2(n) - 0.5$ from above. Of particular interest are the strictly synchronous techniques that achieve a maximum delay bounded by $\mathcal{O}(log_2(n))$, such as the Conditional Sum Adder (CSA). As the paper states, while work on the maximum delay has been extensive, research on the average delay of the adders is comparatively quite scarce. The paper itself focuses on a variety of "hybrid" designs which are larger than the Ripple-Carry Adder (RCA), yet purport to have a faster average speed than strictly synchronous adders, which may also be larger. The adders are simulated and their speeds compared within the context of a simple asynchronous RISC processor.

The paper identifies four key advantages that asynchronous systems may have over clocked ones:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1. As clocked systems increase in size, clock skew becomes an issue and limits clock rates, this delay is not present in asynchronous systems, although other delays are.
2. Hierarchical modular design techniques are ill suited to deal with global constraints such as clock distribution.
3. Asynchronous systems may require less power, as currently "unused" modules perform no switching, as compared to clocked systems.
4. The clock in clocked systems is based on the worst-case timing, see Subsection 1.3.1, whereas asynchronous systems are primarily governed by average delays.

Furthermore, three key issues with asynchronous designs are listed:

1. If *dual-rail encoding*, see Subsubsection 1.3.3.4, is utilized, along with Completion Detection, see Subsubsection 1.3.3.5, then this requires extra chip area and routing.
2. Completion Detection and handshaking, see Subsection 1.3.3, cause an additional logic overhead and thus additional delays, although these overheads may be comparable to the clocking overhead of synchronous systems.
3. Synchronous systems are generally considered easier to design due to the superior tool support.

The paper models the speed of execution of the adders as *cycle times* where the *cycle time* is generally composed of a *computation time*, t_{comp} and a *synchronization time*, t_{sync}. For clocked designs this resolves to the worst-case computation delay, $t_{worst-case-comp}$, as well as the clocking delay, that is the clock skew and latching delay, $t_{clocking}$. For the asynchronous case the average-case computation time is used, and a factor h, such that $h > 1$, that models the synchronization time of an asynchronous circuit, which is generally greater than the clocking time of a synchronous circuit, is introduced. This results in the following formulas:

$$Cycle\ time_{clk} = t_{worst-case-comp} + t_{clocking} \tag{2.1}$$

$$Cycle\ time_{asyn} = t_{average-case-comp} + h \cdot t_{clocking} \tag{2.2}$$

The average-case computation time can further be expressed from the worst-case computation time by introducing the factors i and d, such that $i, d \leq 1$, which model the instruction dependent and data dependent parts of the computation time. Thus, $t_{average-case-comp} = i \cdot d \cdot t_{worst-case-comp}$. Furthermore, clocked designs must budget for the worst-case in terms of tolerances and other environmental parameters, whereas

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

asynchronous designs can take advantage of the nominal conditions. This is expressed as a parameter e, such that $e < 1$. Thus, we finally get:

$$Cycle\ time_{asyn} = e \cdot (i \cdot d \cdot t_{worst-case-comp} + h \cdot t_{clocking}) \tag{2.3}$$

Since only the data dependent parameter directly relies on the adder design it is the one that the paper focuses on.

The paper classifies the adders into 3 categories, *serial*, *tree* or *hybrid*, based on their high level design structure, and selects a total of 6 candidate adders for evaluation. The selected adders are:

- **Ripple-Carry Adder (RCA)**: Has the traditional *serial* structure, which leads to a small size but large worst-case delay.
- **Conditional Sum Adder (CSA)**: Performs in parallel one addition which assumes the carry-in is 0 and a separate addition where the carry-in is assumed to be 1. The results are then combined in a *tree* structure based on the actual value of the carry-in.
- **Completion Detection Conditional Sum Adder (CDA)**: A modified version of the CSA which has more of an asynchronous flavor by detecting the availability of *true sum* at each level of computation and latching it, ending the computation and thus often avoiding the full tree delay. Its mean can be as small as $\mathcal{O}(log_2(log_2(n))$, but it requires extra logic to detect the *true sum* and route it from any arbitrary level to the output latch.
- **Carry-Lookahead Adder (CLA)**: Is considered in a *hybrid* structure, where both P and G, see Subsection 1.3.2, are calculated with basic boolean functions for each block of bits and then rippled through said block to calculate the sum.
- **Carry-Skip adder (SKP)**: Is also considered in a *hybrid* structure, where P is obtained through a boolean function, whereas G is obtained by rippling through the block, with an assumed carry-in of 0.
- **Carry-Select adder (SEL)**: Is also considered in a *hybrid* structure, where both P and G are obtained by rippling through the block. For P a carry-in of 1 is assumed, whereas for G a carry-in of 0 is assumed. Conditional sums are also generated in this process and the correct sum is simply selected when the *true* carry is known.

For the purpose of simulation the adder implementations were assumed to employ Differential Cascode Voltage Switch Logic (DCVSL). The simulation itself is implemented in a hierarchical way, first the delay values for basic cells were obtained for each possible combination of inputs for the cell. These delay values were fed into a gate level simulation

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

for each adder, where each gate is one such basic cell, which looked up the delay value from a table based on the combination of inputs for the cell.

The CDA had the overall best performance due to its small mean of $\mathcal{O}(log_2(log_2(n))$, but the overhead for detecting and routing the *true sum* is unique to it, and so regarding the fairness of the comparison it was not considered for the overall systems performance. Overall, the 3 *hybrid* designs outperformed both the *serial* and *tree* structures. Among the *hybrid* designs the SKP was worst due to its reliance on rippling for generating P and G as well as the sum. Both CLA and SEL were very close as either used rippling for either the generation of P and G or the calculation of the sum.

Finally, the adders are considered within the context of an asynchronous pipelined DLX machine. The architecture is assumed to have 5 pipeline stages:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execution (EX)
4. Memory access (MA)
5. Write back (WB)

For simplicity of the simulation the pipeline is assumed to have no stalls and that there is a 100% success of branch prediction. Furthermore, it is assumed that there is an instruction prefetching buffer from which 75% of instructions are fetched, to relieve the bottleneck in the IF stage. The delays for each stage are assumed to be 3 ns except when a cache access is involved, in which case the delay is assumed to be 10 ns. The delay distributions of the adders are taken from the simulation. The architecture is supplied with a sequence of 100 000 instructions, randomly selected based on a distribution from [PH90]. The results are evaluated in terms of MIPS, million of instructions per second, with the SEL being the clear winner in the comparison. It outperforms an architecture utilizing the RCA by 17% for 32 bit, and by 23% for 64 bit. However, it is of note that the SEL is also 2.5 times larger than the RCA, but cost of the adder is assumed to be small compared with the cost of the overall processor. Finally, under these conditions it can be said that a comparable clocked processor, operating under the same assumptions would require a clock cycle time of 10 ns and thus be outperformed by the asynchronous version. However, this comparison only took into account instruction dependent and data dependent parameters, environmental parameters and synchronization times would have to also be considered for a more accurate comparison.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2.2 An Evaluation of Asynchronous Addition

In his 1996 paper [Kin96] David Kinniment aims to clearly show why asynchronous adder designs fail to achieve their purported, impressive theoretical performance gain, as compared to synchronous designs, in practice, and to discuss the reasons for this.

According to Kinniment the Completion Detection, see Subsubsection 1.3.3.5, is the essential feature of a self timed adder. The design by Gilchrist *et al.* [GPW55] utilizes *dual-rail* encoding, see Subsubsection 1.3.3.4, for the carry paths. The dual carries, C^0 and C^1, are preset to a logic 0 and when one of them reaches a logic 1 in every stage, then the output of an AND gate signals completion. However, if the n-input AND gate is implemented in CMOS then for typical values of n, such as 8, 16, 32 or 64, the area of the gate, as well as its delay will be significant. One alternative is to precharge the carry paths with a logic 1 and detect the presence of a logic 0 using an n-input *NOR* gate, but its delay still increases linearly with n. Kinniment proposes as a faster alternative to build the n-input AND gate as a tree structure.

For the purpose of comparing the adders both the area and delay are considered, and both are expressed in "gate equivalents" of 2-input gates. To this end the basic gates, such as AND, OR, *NAND* and *NOR*, are all counted as 1 gate equivalent, both for area and delay. For this a CMOS implementation style which is flexible enough to produce for example *NAND* and *NOR* gates with similar characteristics is assumed. Additionally, AND-OR-Inverter structures and tree structures of *NAND* or *NOR* gates are assumed to have 80% of the delay of 2 levels of *NAND* gates. Furthermore, gates with a fan-in greater than 4 are assumed to have a delay of $log_4(fan\text{-}in)$ 2-input gate equivalents and occupy an area of $1.33 \cdot log_4(fan\text{-}in)$ gates. Similarly, extra delay and area was added to the comparison as necessary to account for a fan-out greater than 4.

In total three adder designs are considered for this evaluation:

1. **Simple parallel adder (SPA)**: What Kinniment calls SPA is based on his description simply a RCA. It requires 7 gates per bit and incurs a total delay of $2 + 2n$.

2. **Asynchronous adder (ASY)**: is the asynchronous design by Gilchrist *et al.* which was also evaluated previously by Sklansky in [Skl60], where it was called **IDA**. The ASY requires 13.33 gates per bit to implement and incurs a total delay of $2 + 2.5 \cdot log_2(n)$.

3. **Conditional Sum Adder (CSA)**: Is the same adder design as already described in Section 2.1. It requires $3n \cdot (2.22 + 1.22log_2(n+1))$ gates to implement and incurs a total delay of $2 + 2log_2(n+1)$ if $n + 1 \leq 8$, else the total delay is $3.5 + 0.25log_2(n+1)^2 + 0.75log_2(n+1)$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

It should be noted that Kinniments numbers for the required gates and the incurred delay differ from Sklanskys for the ASY and the CSA due to Kinniments revised assumptions regarding the area and delay cost for fan-in and fan-out greater than 4.

Plotting of these area and delay costs over the number of bits shows that the typical CSA operation is slightly faster than the typical ASY operation for 64 bits or less, yet the CSA has a significantly higher area cost. However, Kinniment also notes that this only holds true for random data, and that the ASY's performance can be significantly degraded in a typical application due to its data dependence.

When it comes to real-world applications Kinniment notes three major factors for consideration:

1. The effect of the hardware used to generate the completion signal.
2. The nature of the data that is being used.
3. The implementation technology.

As Kinniment goes on to show, using a *wired-AND* in place of his proposed gate tree would significantly degrade the performance, even though it might map particularly nicely to the chosen technology. On the other hand, if the input operands are random in nature, with an equal chance of generating or propagating a carry at every stage, than the maximum average carry chain for a 32 bit addition is between 4 and 5 [Rei60]. However, results also showed that the overall average for address and data calculations is between 9 and 10 [Gar93]. When considering a split of 18% of operations that are address calculations of the form $(a - a)$, with a result of 0 and a carry propagation chain of length n, then the average length of a carry chain can be considered partly dependent on the word length of the adder. This split of 18% carry chains of length n and 82% carry chains of length $log_2(n) - 0.5$ is showcased by Kinniment. The result of this is that the ASY incurs a significant loss of performance compared to the CSA yet it has a comparable area-time product.

Based on these results it is stated that there is no clear advantage to using asynchronous designs over the conventional synchronous CSA, as there is also little cost effectiveness for the ASY. A further comparison with the results obtained by Franklin [FP94] and Garside [Gar93] is given. To obtain the average values for this comparison two sets of 100 operand pairs were chosen, one random, and one with the previously mentioned characteristics. The chosen technology for this simulation was such that the delay of a 2-input *NAND* gate comes out to 0.42 ns, whereas Franklins values correspond to a *NAND* gate delay of 0.7 ns. The simulations were carried out for a word length of 32 bit. The following table, Table 2.1, gives an abbreviated overview of the results obtained by Kinniment, showing the delays as gate equivalents.

As can be seen the asynchronous adders loose out to the worst-case of the synchronous CSA when considering the non random case.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

32 bit Adder	Average delay (random)	Average delay (non random)	Worst case delay	Basic gate delay
SPA	N/A	N/A	66	0.42
Optimized SPA	N/A	N/A	55	0.42
ASY, Garside	17.2	19.2	40.1	0.42
ASY, Franklin	13.5	24.4	74	0.70
CSA, Franklin	N/A	N/A	11.3	0.70
CSA, Kinniment	N/A	N/A	13.5	N/A
ASY, Kinniment	14.5	24.4	66	N/A

Table 2.1: Abbreviated results from [Kin96]

Finally, Kinniment considers the application in a micropipelined system. Of note is the high possible variance in computation time of the asynchronous adders and its effect on the overall throughput of such a pipeline. A possible further performance loss comes from the consideration that in, for example, a two stage pipeline both stages must be ready for the data transfer. If either stage takes a long time then transfer is held up and the overall throuput degrades. To illustrate this Kinniment has chosen a theoretical two stage pipeline with one stage being the asynchronous adder and the other stage having an assumed average delay matching that of the asynchronous adders, but with a deviation of 1 gate delay. The delay values for the adders and hence also the other pipeline stage are computed from data given by [Gar93], which comes out to an average of 24.36 gate delays for the adder. The end results is that the adder can not improve the overall throughput of this theoretical pipeline due to the more rigid timing of the other stage, whereas it can degrade the overall throughput of the pipeline whenever it is coming close to its worst-case timing. The final result is that the theoretical pipeline using the ASY has an average throughput of one transfer every 33.39 gate delays. In comparison a pipeline using two identical stages with an average delay of 24.36 gate delays and a deviation of 1 would achieve one transfer every 25.38 gate delays.

2.3 Asynchronous Parallel Prefix Computation

In their 1998 paper Rajit Manohar and José A. Tierno present an asynchronous solution to the parallel prefix problem [MT98]. While the prefix problem can be used to solve a variety of different problems efficiently, Manohar and Tierno use their devised method to construct an asynchronous adder with $\mathcal{O}(log\ log\ n)$ average-case latency.

The prefix problem is stated as such:
Let $\otimes$ be an associative operation, then for given $x_1, x_2, ..., x_n$, the $y_1, y_2, ..., y_n$ should be computed such that $y_k = x_1 \otimes x_2 \otimes ... \otimes x_k$, for $1 \leq k \leq n$ [LF80].

The solution proposed in the paper is a tree structure which exploits the associativity of $\otimes$ to simply perform one $\otimes$ to compute y_n given $x_1 \otimes x_2 \otimes ... \otimes x_{\lfloor \frac{n}{2} \rfloor}$ and $x_{\lceil \frac{n}{2} \rceil} \otimes x_{\lceil \frac{n}{2} \rceil + 1} \otimes ... \otimes x_n$.

The processing nodes resulting from this approach are then arranged in a tree structure. They are also further augmented with additional input and output channels to send

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Adder type	Worst case delay	Average case delay (uniform)	Average case delay (workload)
SPA-32	55	N/A	N/A
CSA-32	11.3	N/A	N/A
CSA-64	14.1	N/A	N/A
ASY-32	40.1	17.2	19.2
PA1-32	14.0	13.2	13.8
PA2-32	14.2	10.4	11.2
PA2-64	17.5	11.5	N/A

Table 2.2: Adder delay results from [MT98]

the required prefixes up and down through the tree. The process is further augmented with appropriate acknowledge signals and buffering to permit pipelining of the prefix computation.

The entire tree is finally augmented with a serial solution to the prefix problem. Assuming that $x \otimes a = a$ for some a and any value of x, this serial solution can be fast depending on the data. Thus, every node is modified such that it also serially receives the prefix from its neighboring node and passes it on as well, on any given level of the tree. Since the serial processing is designed such that it produces a fast output $y = a$ for $x = a$ based on the assumption above, this is expected to improve the average-case performance.

This general solution to the prefix problem is then specifically used to implement an asynchronous adder. The resulting delays for these adders are given in Table 2.2, where PA1 is the implementation with the pipelining only, and PA2 has the added serial computation. The "-32/64" denotes the width of the given adder. The results for the other adders are taken from [Kin96] and are given as the number of gate delays. The results are given for both a uniform distribution of inputs and a more realistic workload for 32 bit adders taken from [Gar93].

As the data shows the PA2 adder implementation outperforms the CSA from [Kin96], with the expectation that the gap increases with increasing adder width. The CSA's strong performance for small n is noted, however, and it is theorized that a better asynchronous adder could be constructed by combining the carry select technique with the prefix computation. Both the PA1 and PA2 have larger areas than the CSA due to the overheads incurred from pipelining and the generation of the acknowledge signals.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 3

Implementation and Simulation

3.1 Selection of Adder Architectures

As mentioned previously in Section 1.4 the selection of the adder architectures plays an important part in this thesis, as we want to get a good overview of different performance improvement techniques at various levels of complexity and with different benefits and drawbacks. To keep the workload at a manageable level it was decided that four adder architectures should be investigated in this thesis.

3.1.1 Ripple-Carry Adder (RCA)

The simplest adder architecture that we will investigate further is the Ripple-Carry Adder (RCA). The RCA for a given width of n bits is implemented by chaining n Full Adder (FA) blocks together. This is achieved by connecting the carry-out of the previous FA to the carry-in of the subsequent FA.

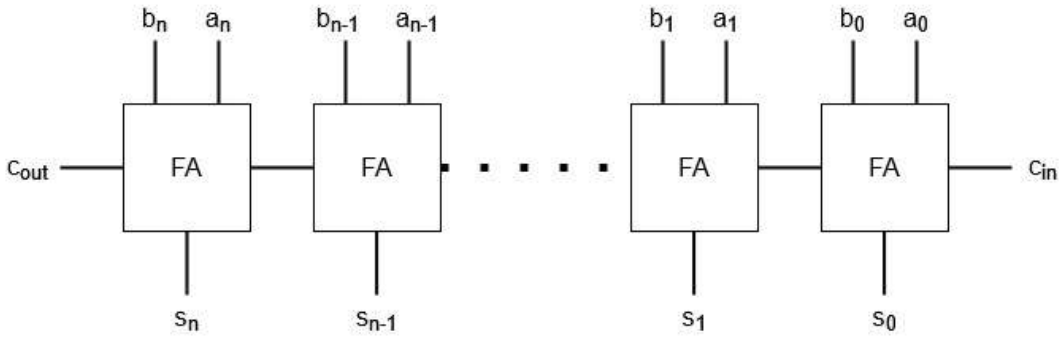

Figure 3.1: Basic block diagram of a RCA with carry-in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

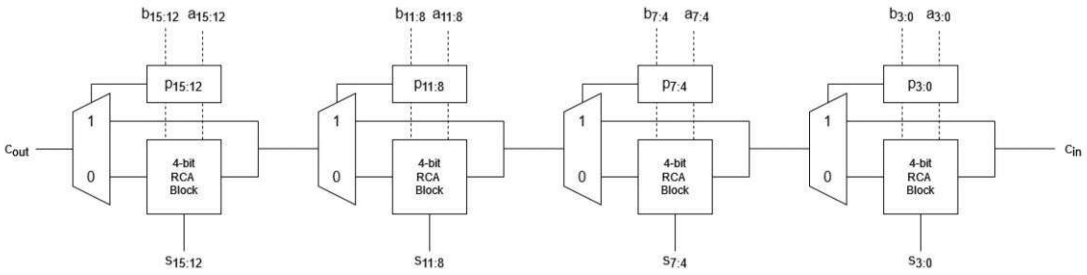

Figure 3.2: Block diagram of a 16 bit CSKA with carry-in

The main draw of the RCA has already been outlined in this thesis, but to reiterate, it has the worst worst-case performance among the adders, namely that its critical path is directly proportional to n ($\mathcal{O}(n)$) where n is the width of the adder in bits. Thus, we expect the most benefit when implementing this adder in asynchronous logic.

$$t_{crit} = n \cdot t_{FA} \tag{3.1}$$

Equation (3.1) gives the delay of the critical path for the RCA where n is the width of the adder in bits and t_{FA} is the delay of a FA.

As stated in the second part of the research question, in Section 1.4, we will use the RCA as a benchmark for the benefits and drawbacks of the other adder architectures.

3.1.2 Carry-Skip Adder (CSKA)

The next adder architecture chosen for our consideration is the Carry-Skip Adder (CSKA). It is created by modifying a RCA with additional "skip"-logic which will allow the carry to skip a certain amount of bit-positions. It should be noted here that the "skip"-part and the other common name, carry-bypass adder, are both slight misnomers as the carry still ripples through the "skipped" FAs, but it can also advance to the next block of FAs to possibly start calculation of the respective sum-bits there almost in parallel.

This skipping of the blocks of FAs is achieved utilizing a skip-logic which calculates all the propagate signals for the given block of bits ahead of time. The combined group propagate signal then controls a Multiplexer (MUX) which "forwards" the carry-in of this block directly to the next block. Thus, the next block already calculates the results after one delay of the MUX rather than having to wait for the delay of m FAs.

Thus, the critical path of this adder is as follows:

$$t_{crit} = 2 \cdot m \cdot t_{FA} + (k-1) \cdot t_{MUX} \tag{3.2}$$

where m is the number of bits in one group and k is the number of groups that the adder is split into. It should be noted here that this formula should mainly illustrate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

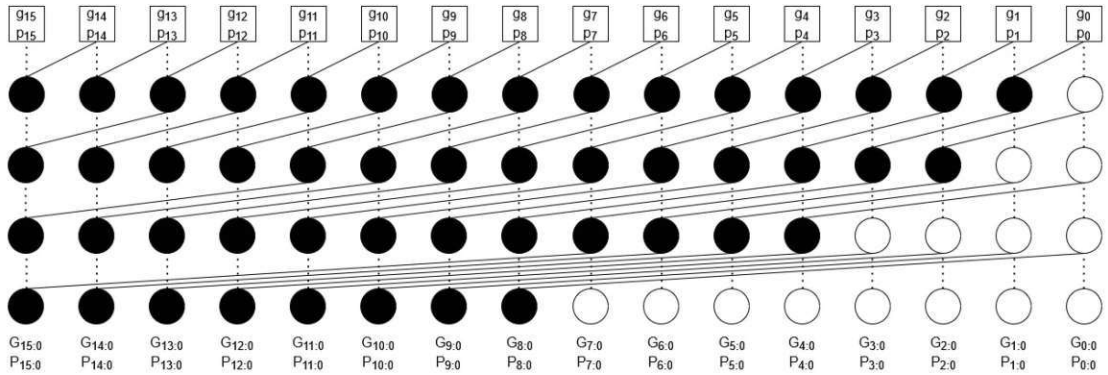

Figure 3.3: Prefix tree of a 16 bit KSA

the significant shortening of the critical path when compared to the RCA. An analysis which takes a more fine-grained view of this critical path, that considers the individual gates which form the FA and the MUX will produce an overall slightly different looking formula.

Finally, it should be noted that the CSKA was chosen for this thesis as it produces a substantial improvement when implemented in synchronous logic, when compared to the RCA, but at a minimal logic overhead in terms of additional gates or wiring needed. While there is a significant amount of research regarding the optimal configuration of a CSKA for some given parameters, this thesis will implement a very simple version of the CSKA with a fixed group size of 4 bits per group and only a single level of skip-logic.

3.1.3 Kogge-Stone Adder (KSA)

The Kogge-Stone Adder (KSA) is based on a paper by Kogge and Stone which introduces a general algorithm for solving recurrence problems [KS73]. The adder architecture itself is based on the carry-lookahead adder which was discussed in Subsection 1.3.2. However, rather than simply unrolling the recursive equation for the respective c_i the algorithm by Kogge and Stone introduces the blueprint for a tree structure that calculates the group carries in a parallel manner.

The group carry signals are calculated based on the definition given in Equations (1.3) and (1.4).

An example of this prefix tree is given in Figure 3.3, where the filled, black nodes represent a processing node which implements the logic according to the previously mentioned equation, and the empty, white nodes represent a simple throughput where the output is equal to the input.

The KSA comes with some important benefits that make it worthwhile for this thesis, namely it achieves a constant fan-out of 2 at every node and it achieves very good performance due to its high degree of parallelism and its low logic depth. The logic depth for the prefix tree of a KSA is $log_2(n)$ where n is the width of the adder in bits.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

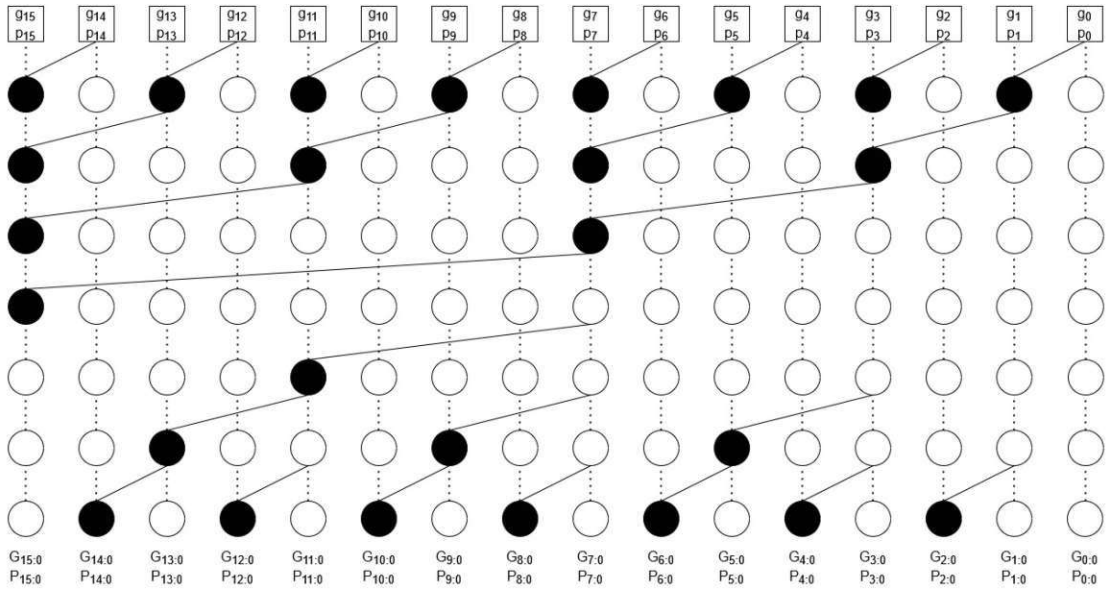

Figure 3.4: Prefix tree of a 16 bit BKA

$$t_{crit} = log_2(n) \cdot t_{GC} + t_{pg} + t_{sum} \tag{3.3}$$

Equation (3.3) gives the critical path for the KSA where t_{GC} is the delay of one processing node, t_{pg} is the delay of the logic block the calculates the initial g_i and p_i and t_{sum} gives the delay of the logic block which calculates the sum.

3.1.4 Brent-Kung Adder (BKA)

The Brent-Kung Adder (BKA) is the final adder architecture that will be considered for this thesis. Like the KSA it is a parallel prefix tree adder, but it structures its prefix tree using what Brent and Kung refer to as a "regular layout" [BK82]. This regular layout improves the area utilization of the adder and contains fewer processing nodes than the prefix tree of the KSA.

An example of this prefix tree is given in Figure 3.4. As can be seen the improved area utilization and reduction in the number of processing nodes, which also helps with wire congestion, comes at the cost of increased depth of the tree, which directly impacts the performance of the adder. The prefix tree of the BKA has a depth of $2 \cdot log_2(n) - 1$ where n is the width of the adder in bits.

The BKA has some important differences to the KSA that make it a worthwhile candidate adder architecture for this thesis. It uses significantly fewer processing nodes in its prefix tree which greatly reduces area usage and its much sparser layout makes it easier to implement on a chip. However, these benefits come at the cost of performance when compared to the KSA.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$t_{crit} = (2 \cdot log_2(n) - 2) \cdot t_{GC} + t_{pg} + t_{sum} \tag{3.4}$$

Equation (3.4) gives the critical path for the BKA with the t_{GC}, t_{pg} and t_{sum} being the same as for the KSA.

In general we suppose that $t_{GC} < t_{FA}$ and that $t_{pg} < t_{FA}$ since the processing node represented by t_{GC} introduces at most two basic gate delays and that the block calculating the initial generate and propagate signals g_i and p_i introduces only a single gate delay. Furthermore, $t_{sum} < t_{FA}$ since it also introduces only a single gate delay, whereas the FA introduces three gate delays. This argumentation assumes that the delays introduced by basic logic gates, such as AND, OR or XOR, are roughly equivalent, and the difference would not be significant for evaluating these critical paths.

3.2 Implementation

As mentioned in Subsection 1.4.1 the selected adders were implemented in NCLX design style for the asynchronous version. For the implementation an open, 45nm cell library which provides the basic logic gates (AND, OR, XOR) and function blocks (FA, MUX) that we needed was chosen. The designs were implemented in VHDL and compiled using the Synopsys Design Compiler (DC). This process, as well as the simulation process described in the next section, was automated using a simple shell-script. The script is configured with a list of architecture, a list of bit widths, and a simple selector to determine whether the synchronous or QDI versions, or both, should be compiled and simulated.

3.2.1 Synchronous Adder Implementation

The synchronous designs were implemented in a straightforward manner by instantiating and interconnecting the cells provided in the aforementioned library. No additional optimization by hand was done and the optimizations done automatically by DC were entirely disabled. This approach was due to two main reasons. First, it was discovered early in the implementation process that the DC, when not otherwise configured, "optimizes" the more complex adder designs into logically equivalent combinations of prefix tree and ripple-carry logic, which when simulated performed significantly worse than the simple ripple-carry adder.

For this reason it was deemed necessary to turn off any automatic optimizations performed by the tool. In addition, the asynchronous designs were not supposed to be optimized by the tool anyways, since we would require the logic to be exactly what we specified, so as to guarantee the QDI properties of the design. This required specifying a so called "dont_touch" attribute for all cells and nets within the design, so as to guarantee that the design remains exactly as specified. Furthermore, the boundary optimization which optimizes across sub-design boundaries was turned off as well.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
architecture arch of rca_adder is
   -- Components
   component FA_X1
      port (A, B, CI : in std_logic; CO, S : out std_logic);
   end component;
   -- Signals
   signal c_s : std_logic_vector(BIT_WIDTH downto 0);
begin
   c_s(0) <= c_in;

   -- generate adder from full adder blocks
   fa_gen : for i in 0 to (BIT_WIDTH-1)
   generate
      FA_inst : FA_X1
      port map
      (
         A => a(i),
         B => b(i),
         CI => c_s(i),
         S => s(i),
         CO => c_s(i+1)
      );
   end generate;

   c_out <= c_s(BIT_WIDTH);
end architecture;
```

Listing 1: VHDL architecture of a synchronous RCA

So to provide a better basis for the comparison and evaluation of the designs, and due to the aforementioned complications, it was decided to turn off all optimizations by the tool for the synchronous designs as well.

Listing 1 shows the implementation of the synchronous RCA in VHDL. Due to the FA being available as a cell in the library it is quite simply a matter of connecting one FA with the next, as well as connecting both of the carry-in and carry-out to the respective signals going into or out of the first or last FA, respectively.

3.2.2 NCLX Adder Implementation

To implement the NCLX adder designs it was necessary to first create NCLX versions of the logic gates and function blocks that were used in the designs from the available cells of the library. Additionally, an implementation of the Muller C gate from the library cells was also required.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

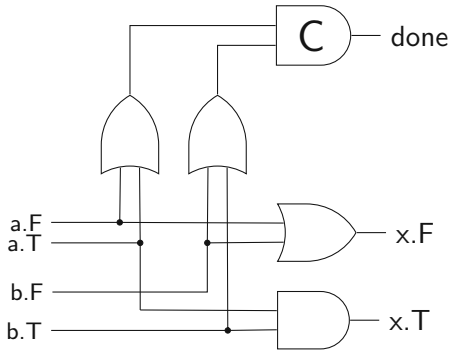

Figure 3.5: NCLX AND gate with explicit input CD

3.2.2.1 Basic gates

As previously mentioned, a possible advantage of the NCLX design style is the explicit CD, which means that there are no C gates on the data path. Thus, the NCLX versions of the logic gates and function blocks were also created without a CD or done signal.

Thus, the NCLX versions of the gates are essentially just dual-rail implementations of the respective logic gates. For example the AND gate shown in Figure 3.5 is essentially just one logic gate per output-rail which produces the required logic function for this rail. Thus, the NCLX AND gate has one OR gate to assert the "false" rail when either of the input "false" rails is asserted, and one AND gate which asserts the "true" rail when both of the input "true" rails are asserted. The NCLX OR gate works analogously.

3.2.2.2 XOR

The NCLX version of the XOR gate is slightly more complicated. It requires 4 AND gates to check for each of the possible input signal combinations and then 2 OR gates, one for each of the rails, which assert the respective rail based on the output of the AND gates. Since only one input combination can be applied to the XOR gate at any given time only one of the AND gates and therefore only one of the OR gates asserts its output.

This approach allows us to forgo unnecessary, duplicate CDs for example for the input signals, since rather than each NCLX gate individually checking whether the input signal has arrived, considering the *isochronic wire forks* we can check a given signal just once and assume that it has arrived at all of its endpoints.

3.2.2.3 FA & MUX

The slightly more complex function blocks, namely the FA and MUX, also needed to be implemented in NCLX. For the synchronous designs both of these blocks were available as cells in the library, but for the asynchronous design they were created from the NCLX logic gates, according to their logic function, as defined by a truth table for example.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

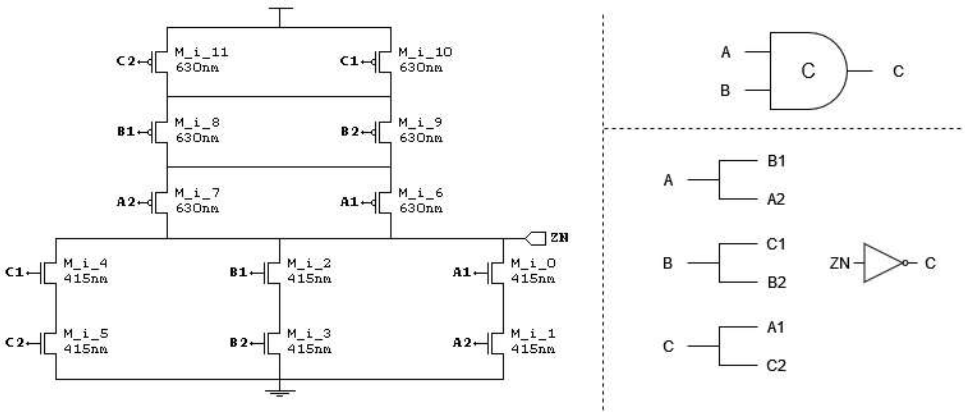

Figure 3.6: Muller C gate, left is the schematic of the AOI222 cell from the library, top right is the schematic symbol of the Muller C gate and bottom right shows how the logical pins of the C gate map to the actual pins of the AOI222 cell

3.2.2.4 Muller C gate

The Muller C gate was implemented using an AOI222 cell from the library, as well as an inverter cell, to create a storage loop that exhibits the desired hysteresis behavior. Figure 3.6 shows how the Muller C gate was created from the AOI222 cell. It should be noted that the output pin of the C gate, pin "C", is connected back to the inputs to form a combinational storage loop, which is needed to created the aforementioned hysteresis behavior. Furthermore, in practice an additional buffer cell was needed for the output of the C gate, since the Design Compiler refused to establish a connection between the internal signal and the actual output port of the C gate in certain cases otherwise. The individual C gates of the CD are then combined into a tree structure, as shown in Figure 3.7, and the individual done signals then combine into a final, single done signal for the combinational logic. As the figure shows two internal done signals can feed into one C gate, however, the intermediate results of these C gates have to be combined in an additional C gate as well. In general we can say that $n-1$ C gates are necessary for combining n done signals. The same would also be true and perhaps even more evident if we were to combine the done signals in a C gate chain, where at every stage the respective C gate combines the results of the previous stage with one new signal to form the new result. It should be noted here that for the tree structure and for $n : n > 1 \wedge n \neq 2^i : i \in \mathbf{N}$ there are multiple ways of constructing the tree structure, however, neither the required number of C gates nor the maximum depth and thus the longest path through the tree depend on the specific way that the tree is constructed. The advantage to using a tree structure is naturally that the longest path through the structure is much shorter as it has logarithmic rather than linear scaling.

As Figure 3.5 shows the actual logic behind the CD without the required C gate structure is rather simple. For each dual-rail signal one OR gate is needed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

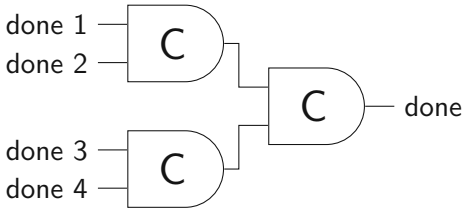

Figure 3.7: Muller C gate tree

3.2.2.5 Adder

The adders themselves are implemented by taking the existing implementation of the synchronous versions and replacing the occurrences of the library cells with the newly created NCLX versions of the same cells. Of course this also means any port or signal has to be replaced with a dual-rail version. Finally, a CD has to be added where required.

Listing 2 shows the QDI implementation of the RCA, which can be compared to the synchronous version shown in Listing 1. The logic part itself looks quite similar, as would be expected, given that it is the same logic, simply implemented in a dual-rail way. The main differences are that the FAs now produce an additional output, the done signal. Of course, there is also the top level CD, which combines all of the individual done signals.

3.3 Simulation

The implemented adder designs, both the synchronous and asynchronous versions, were simulated using ModelSim. An overview of the complete compilation and simulation process is given in Figure 3.8.

3.3.1 Simulation Setup

As described in the previous section, Section 3.2, the simulation process was automated in the same way, and utilizing the same script as the compilation process. This allows specifying which architectures, and whether either the synchronous or the asynchronous or both version should be compiled and then simulated, as well as the bit widths of the adders for which the specified designs should be simulated.

Additionally, it was also possible to specify the simulation mode for the smallest two widths, 8 bit and 16 bit. This means specifying whether the operands of the addition are selected randomly, based on a uniform distribution, exhaustively, meaning that every possible combination of operands is applied, or as a counter, meaning that the adder simply counts from 0 to the maximum for the given width. Simulation of the 8 bit wide adders supports all 3 modes, whereas 16 bit only supports the random and the counter mode, and the wider widths only support the random mode.

Overall, four common bit widths, namely 8 bit, 16 bit, 32 bit, and 64 bit, were chosen as possible values for the width of the adders. The limitation of only simulating for these 4

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
architecture arch of qdi_rca_adder is
   -- Signals
   signal c_s : dual_rail_vector_t(BIT_WIDTH downto 0);
   signal done_s : std_logic_vector(BIT_WIDTH-1 downto 0);
begin
   -- carry in
   c_s(0) <= c_in;
   -- generate adder from full adder blocks
   fa_gen : for i in 0 to (BIT_WIDTH-1)
   generate
      fa_inst : entity work.qdi_fa
      port map
      (
         a => a(i),
         b => b(i),
         c_in => c_s(i),
         s => s(i),
         c_out => c_s(i+1),
         done => done_s(i)
      );
   end generate;

   -- carry out
   c_out <= c_s(BIT_WIDTH);

   -- done signal
   done_tree : entity work.c_gate_tree
   generic map
   (
      SIGNALS => done_s'length
   )
   port map
   (
      input => done_s,
      done => done
   );
end architecture;
```

Listing 2: VHDL architecture of an asynchronous RCA

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

specific widths also comes at least in part from the required simulation time. Anecdotally we can say that simulating a given adder with a width of 64 bit for the same number of iterations, i.e., executed additions, takes longer than the simulations for the 3 other bit widths combined. Furthermore, having the 4 distinct bit widths already gives valuable data to judge how well the given architecture scales for larger additions.

3.3.2 Simulation Run

A simulation run of a given adder design and simulation configuration involves embedding the adder in a generic testbench which, based on the simulation mode, selects the operands and applies them to the adder. The testbench then reports the operands, as well as the start time, end time, and duration of the addition in a CSV file. For the duration only the time until the correct result is present at the outputs is counted. This is done in a loop for a specified number of iterations.

The random values for the operands are generated using the functionality provided by the "RandomPkg" from OSVVM. This also guarantees us that we get the same random operands for the different designs, so that they are properly comparable.

For the asynchronous adders the testbench also has to apply the null phase after each addition. Especially for the asynchronous adders, in addition to the duration of the addition, the time until the done signal transitioned to high, indicating that the data-phase is now complete, as well as the time until the null phase is processed and the time until the done signal transitioned to low, indicating that the null-phase is complete, were also reported in the CSV file.

This of course allows for a proper comparison between the worst-case of the synchronous and the full cycle time of the asynchronous adders.

Additionally, the synchronous adders were also reset after each performed addition, since the goal is a statistical evaluation of the individual performed additions, rather than of the theoretical throughput. This also had another positive effect, since especially for the smaller two bit-widths it was not uncommon to have two consecutive pairs of operands which produce the same result, resulting in a duration of 0 for that particular simulation iteration.

3.3.3 Area and Timing Analysis

Unlike the delay values obtained by simulation, the area usage of the adders is directly reported by DC using the simple "report_area" command. The area report contains a listing of the number of ports, nets, and cells, as well as a further breakdown of the types of cells, e.g. combinational or sequential. Additionally, the report also contains the total area usage of the design, as well as a breakdown based on the type of area, e.g. combinational, noncombinational or interconnect area. The report itself does not contain a unit for the area value, rather the unit has to be taken from the library to which the design is mapped. For the open library used for this thesis the area is given in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

square micro meters [µm²]. Furthermore, the library used by this thesis has an area value of 0 defined for the wire load models. Thus, the area report cannot calculate a total interconnect area for the design, instead the interconnect area and by extension the total area of the design remain as "undefined". Due to this circumstance the thesis will instead use the "total cell area" as the principal value for all of the comparisons. However, in the grand scheme of things this should not cause any significant difference, as we would expect the total interconnect area to be roughly proportional to the aforementioned reported numbers, i.e., number of ports or number of nets.

Similarly the timing, or more specifically the maximum timing path in the design, which is the critical path, can be reported directly by DC using the command "report_timing". The generated report contains a listing of the points in the design along this unconstrained maximum path, the increment that the given point adds and the cumulative length, measured in time, that the path has from the start to that point. The report ends with the "data arrival time" which for our unconstrained, maximum path is the time when the data arrives at the output, when starting from the input at time 0. Like the unit for the area the unit for the time values is also defined by the used library, in this case the timing is given in nanoseconds [ns]. Furthermore, it is possible to set the number of reported digits to control the output format for the reported values. This has no influence on the precision of the calculation, which is always done with the maximum possible precision that the operating system supports for floating point arithmetic. While the maximum timing path is not as important for the asynchronous designs, since they will be compared based on their average delay, as calculated from the simulation results, it is very important for the synchronous designs, since they will be compared primarily based on this reported value.

Special care had to be taken for CSKA since the timing analysis reports the longest signal path that it can find, even if this signal path can not logically occur for any input configuration. In this case this means that it chooses the path through the MUX which goes through the block of FAs rather than "skipping" the block every time, even if this implies logically that each block either generates or kills the carry-in at some bit-position within the block. This of course does not form the actual critical path, however, since it is theoretically an electrical signal connection that runs all the way through the adder it gets reported as the longest signal path. Thus, it was necessary to constrain the timing analysis to the path that was discussed in Subsection 3.1.2. However, it was possible to achieve this in an automated way given that the path has a very regular composition, going only through the first and last block and "skipping" the others.

3.3.4 Simulation Results

The results of the simulation are stored in a CSV file containing one row for each executed addition. These CSV files are then loaded into a Jupyter Notebook, a Python based tool with a Web-browser-based user interface, where they are processed.

This processing involves an evaluation of the statistical parameters of the given simulation

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

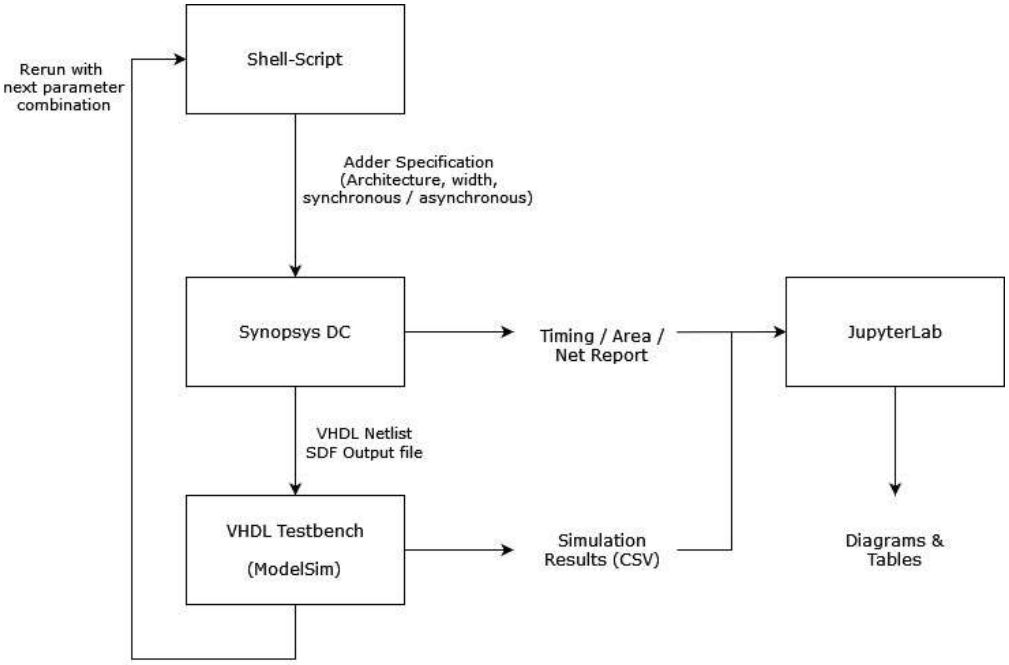

Figure 3.8: Block diagram of the simulation setup

series. In particular the average and the maximum for any given adder are of importance. Furthermore, the synchronous adders have the delay of their reported maximum timing path, which is the critical path of the design, as described in Subsection 3.3.3 also added to the processed results. These results are plotted as *box plots*, since this allows to show all of the relevant data in a very compact form. The *box plots* have the average, maximum and the reported worst-case path overlayed, to make comparison easier.

The key values for the performance, expressed as the delay of the adder, the area usage and the number of nets were also printed as bar graphs to allow an easy comparison of this key data. In addition to the total number of nets, the maximum and the average fanout for the architectures are also displayed as bar graphs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Results

Based on the aforementioned methodologies the architectures, both synchronous and asynchronous, will now be compared on their key metrics, as outlined previously.

4.1 Performance

As mentioned in Chapter 1 the key metric which we want to evaluate is the performance. For our purposes the performance will be evaluated as the delay from when the inputs are applied to the adder until the correct result is present at the outputs, as mentioned in Section 1.4. However, there are some additional considerations in place. Namely, we have to consider that synchronous adders are judged on their worst-case performance, since that is what determines the minimum clock period and conversely the maximum clock frequency of a synchronous design. On the other hand, we have to also consider the CD for the asynchronous adders. Even when the correct result is already present at the output the adder is formally speaking only finished when the CD indicates this. So for the synchronous adders the results regarding the delay were further enhanced with the worst-case path, as obtained from the design tool, as already described in Subsection 3.3.3. And for the asynchronous adders both the delay values with and without the CD are available for comparison.

First, we will compare the synchronous implementations in order to establish the relative performance gain of the more complex architectures against the RCA. For the plots in this section we used the 16 bit variant in any such cases where only the results for a specific width are plotted in a single plot. This is because the 16 bit variants in general provide the best visual clarity for the plots, since the difference between the average and the worst-case increases linearly with the width for the RCA. In comparison to the other architecture this leads to a stretching of the diagrams, as the RCA's worst-case data-point is significantly far removed from all other relevant data.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

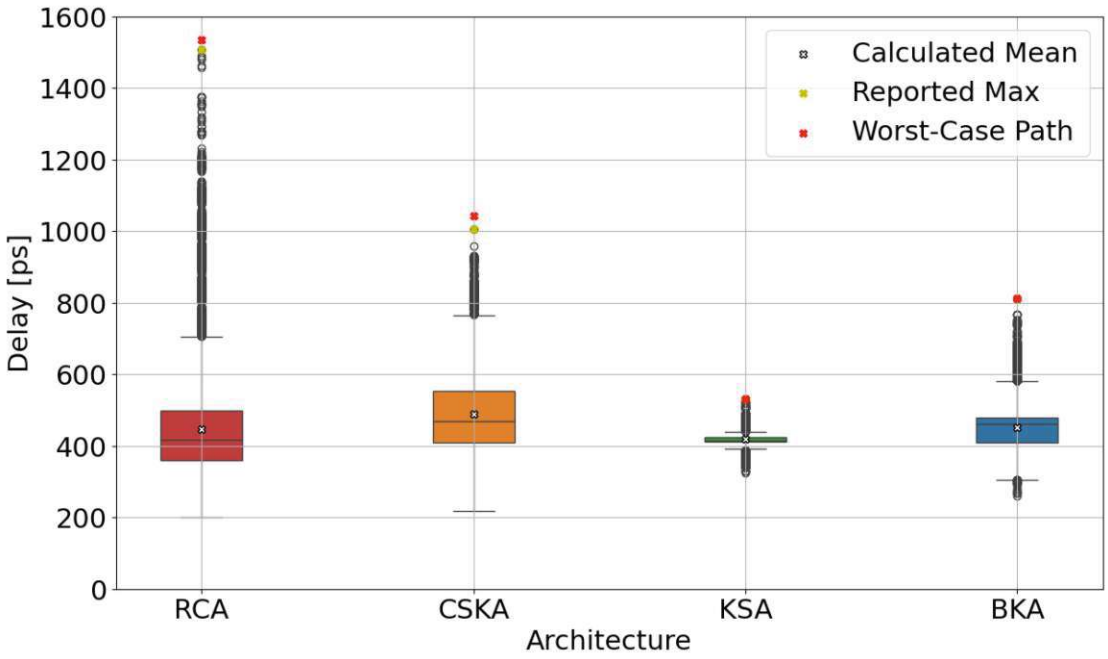

Figure 4.1: Delay results of the synchronous 16 bit adders

As can be seen in Figure 4.1, the average delay of all the architectures is very similar. Based on the already stated observations this is an expected outcome, since the primary focus of the improvements of the modern adder designs lies in reducing the critical path of the adder as much as possible through parallelism. We can see this in the figure as well, where the two parallel prefix adders, namely the KSA and BKA, have significantly reduced worst-case paths when compared to the CSKA and especially when compared to the RCA. Furthermore, we can also observe that the complex adders have their delay values more tightly grouped around the mean. This may seem obvious for the delay values which are larger than the mean. When the mean and the maximum or in this specific case the worst-case delay are closer together then the values in between are naturally more tightly grouped. However, for the parallel prefix adders that we have observed here this is also true for the delays that are less than the mean. Since the prefix tree has to be traversed in any case this means that a baseline delay is introduced even for the simplest additions. Thus, both the RCA and CSKA which have either no such overhead in the case of the RCA or only a very small overhead in the case of the CSKA can produce faster additions in very specific cases, than the KSA or BKA can.

When we now examine the delay results of the asynchronous adders, as shown in Figure 4.2, we can see that the results are almost the same. For these results the CD was not considered, this shows us that expanding the conventional single-rail logic into the dual-rail equivalent has, as expected, no noticeable impact on performance. Of course the main drawback of this design style so far is that singular standard library cells are

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

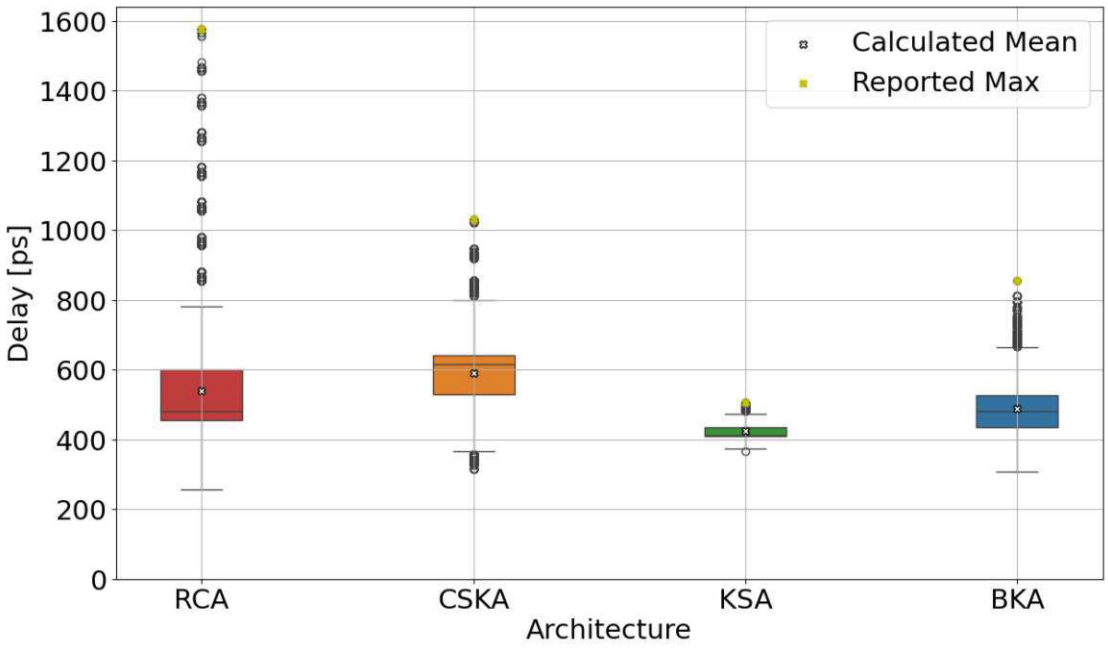

Figure 4.2: Delay results of the asynchronous 16 bit adders, without CD

replaced with multiple such cells to implement the same logic function. As mentioned in Subsection 3.2.2 some of the cells such as XOR or MUX require multiple cells to implement. Because these cells are not only in parallel, as is the case for the AND and OR for example, this means that certain paths through the designs became slightly longer when implemented in this way. This leads to the observable differences in the results between Figure 4.1 and Figure 4.2.

However, for the overall results we can make the same general observations that we already made for the synchronous results. So in the next step we will now also consider the CD and observe how this changes the results regarding the performance. The results are shown in Figure 4.3 and it is immediately obvious that the CD has a big impact on the overall performance of the adders. While we can observe that the delay values still show the same grouping patterns, in regards to how tightly the delays are grouped for each architecture, we can now also see that the CD takes a significant amount of time to produce the done signal. Since the CD has a different level of complexity for each architecture we can see that the results are not merely shifted upwards by a flat amount, instead the more complex architectures take significantly longer to produce their done signal than the less complex adders. This can also be seen in Figure 4.6, which shows the difference in time between when the data is available at the output and when the done signal is finally available. This is of course because the size of the CD increases with the number of intermediate signals in the design for which their arrival must be detected. With these results we can also see that, at least for this bit width, while the RCA still has the longest maximum delay it also now has the lowest average delay, which

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

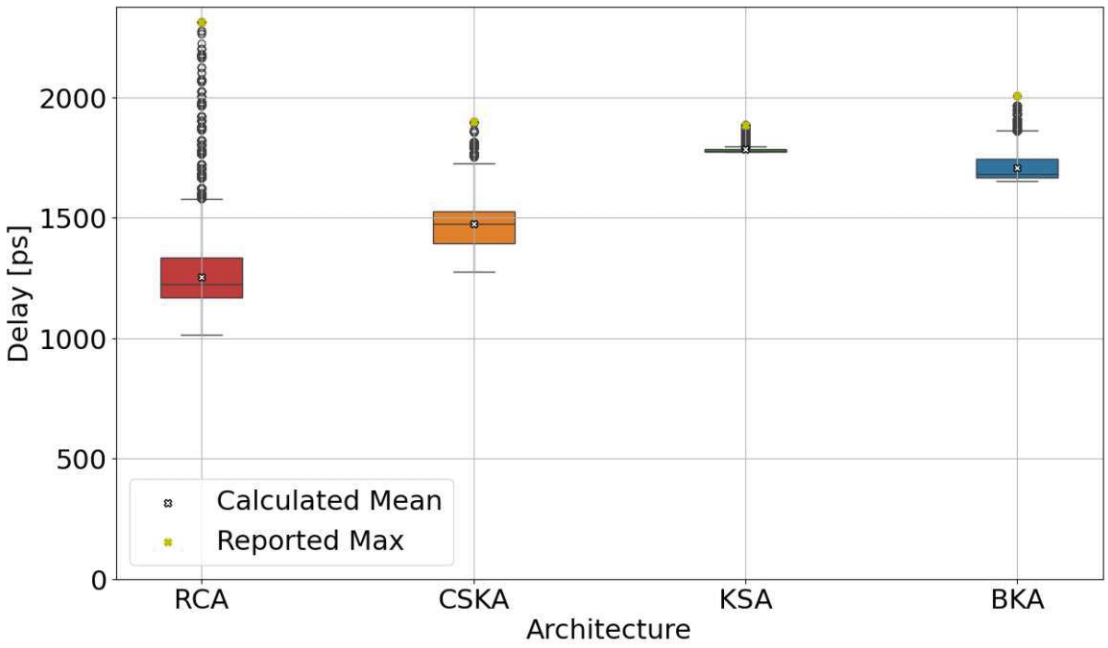

Figure 4.3: Delay results of the asynchronous 16 bit adders, with CD

is what we care about most in regards to the asynchronous adders.

Both Figure 4.2 and Figure 4.3 show the results for the data-phase of the calculations, but every such calculation also has a null-phase. To that end Figure 4.4 and Figure 4.5 show the results of the null-phase without and with the CD. The first thing that we can take note of is that all of the architectures are very quick at producing the "spacer" at its output. When we then consider the total delays for the null-phase as seen in Figure 4.5 we can conclude that nearly all of this time is spent solely waiting on the CD. This is further supported by Figure 4.7, which shows the difference between when the "spacer" is available at the output and when the done signal finally goes to '0'.

One explanation why the adders seem to have more overhead for "resetting" the done signal is that the CD runs partially in parallel with the actual data-path of the adders. Since the actual calculation or "forward"-path of the adder takes longer than producing the "spacer" or "reverse" path does, the pure overhead, where the desired result is already present and the time is purely spent waiting on the CD is thus perceived lower. Even though these results suggest that "resetting" the CD simply takes longer, this is not the whole truth. When comparing Figure 4.3 and Figure 4.5 we can see that mean time for the RCA is roughly the same in both directions, whereas for the other architectures it holds true that the "reverse" direction takes longer. The main result that supports the fact that the "reverse" direction is slower stems from the KSA and BKA. For both architectures the "reverse" direction is slower, even though producing the "spacer" is significantly faster than the actual addition for both adders. However, when we also

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

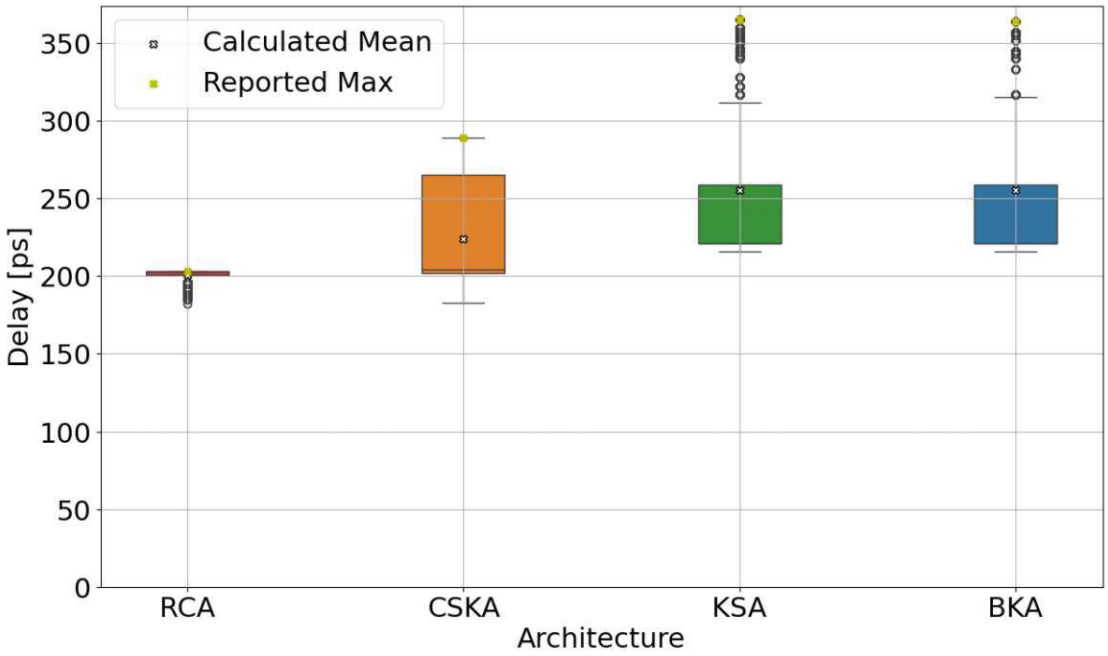

Figure 4.4: Delay results of the asynchronous 16 bit adders, without CD, null-phase

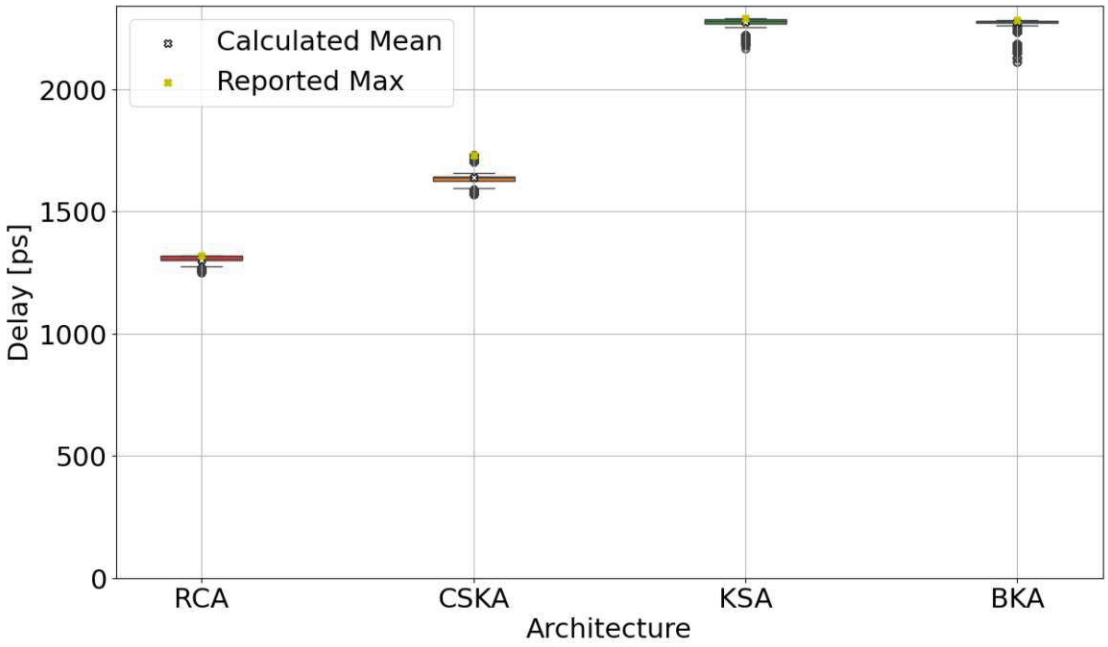

Figure 4.5: Delay results of the asynchronous 16 bit adders, with CD, null-phase

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

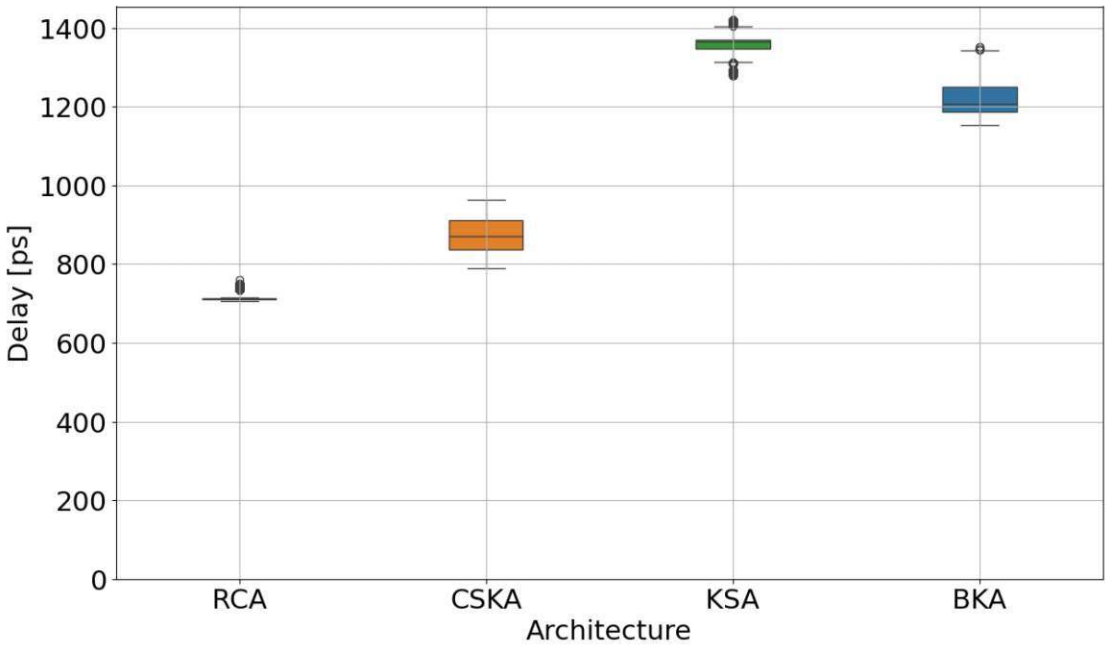

Figure 4.6: Delay overhead of the CD for the asynchronous 16 bit adders, data-phase

consider the aforementioned results of the RCA this suggests that this difference in the "forward" and "reverse" latency scales with the size of the CD. Overall, these results let us identify the CD as the clear bottleneck in regards to the performance of the asynchronous adders.

Finally, a full comparison of the cycle times of both the synchronous and asynchronous adders can be seen in Figure 4.8. For the synchronous adders the worst-case is shown. For the asynchronous adders on the other hand the full cycle, that is both the data- and null-phase are shown. Of course, as previously mentioned, it is the mean cycle time that is used for the asynchronous adders.

Despite the large overheads that were identified for the QDI adders it can be seen here that the QDI RCA starts outperforming its synchronous counterpart on average at a size of 32 bits. At a width of 64 bit **all** of the asynchronous adders outperform the synchronous RCA on average.

4.1.1 Conclusion

In summary we can say that the adders exhibit the same relative performance characteristics when implemented in NCLX as their synchronous counterparts, when considering the data-path only. However, when the CD is also considered then the more complex architectures experience a significant loss of performance as compared to the RCA. As outlined already in Subsection 3.2.2 this is because the CD consists of Muller C gates which are much more expensive in terms of performance. The more complex adders

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

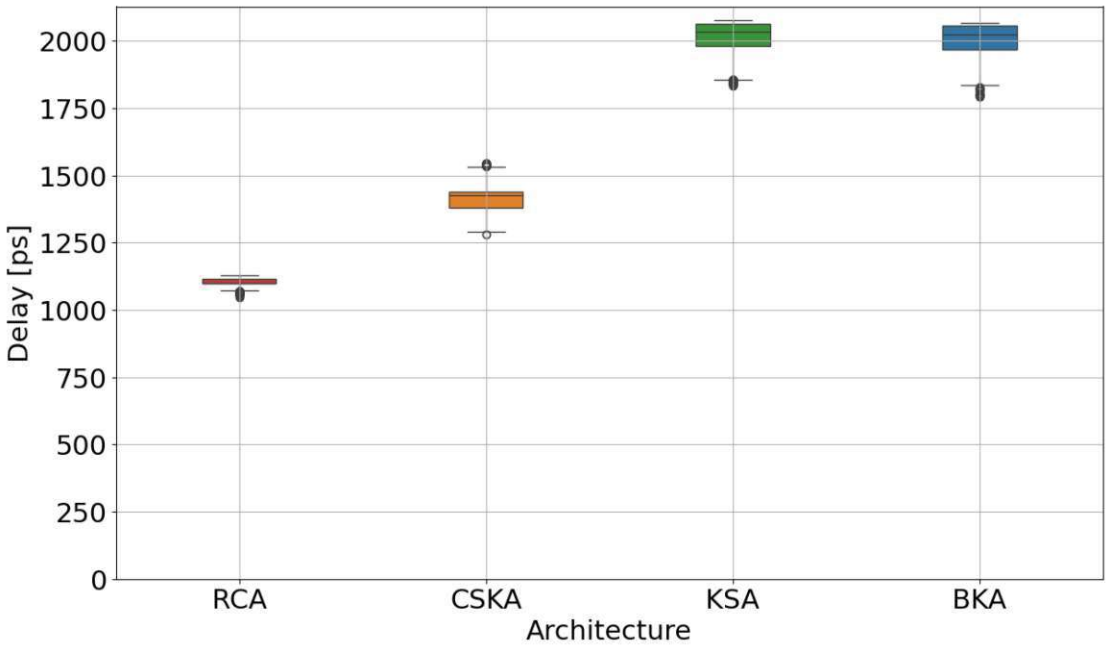

Figure 4.7: Delay overhead of the CD for the asynchronous 16 bit adders, null-phase

achieve their performance gain with higher degrees of parallelism as compared to the RCA or the CSKA, and this in turn is achieved by having more total gates working in parallel. This means that not only are the gates working more in parallel but there simply are also more gates in the design in general. But this is in this case also the downfall of these adders, what is gained on the data-path for the processing of the result is then lost with the need for a larger CD due to the increased number of intermediate signals. Furthermore, we naturally have to consider that the asynchronous adders require a "spacer" or null-phase in this design style, to separate to valid data-words. This not only halves the total throughput but as the results show the larger CDs seem to disproportionately struggle with producing a logic 0 at their output.

4.2 Area Usage

It is well known that the more complex adder architecture have significantly larger areas when compared to the RCA. As mentioned previously this fundamentally stems from the higher degree of parallelism that most of these adders achieve. This parallelism comes at the cost of additional logic that mostly operates on the aforementioned group carry signals, see Subsection 1.3.2. For a simple approximation we can view the logic that generates the initial single-bit group carries plus the logic that ultimately calculates the sum for each bit position as roughly equivalent to a single FA per bit.

This approximation leaves the entirety of the tree-like structure itself as pure overhead

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

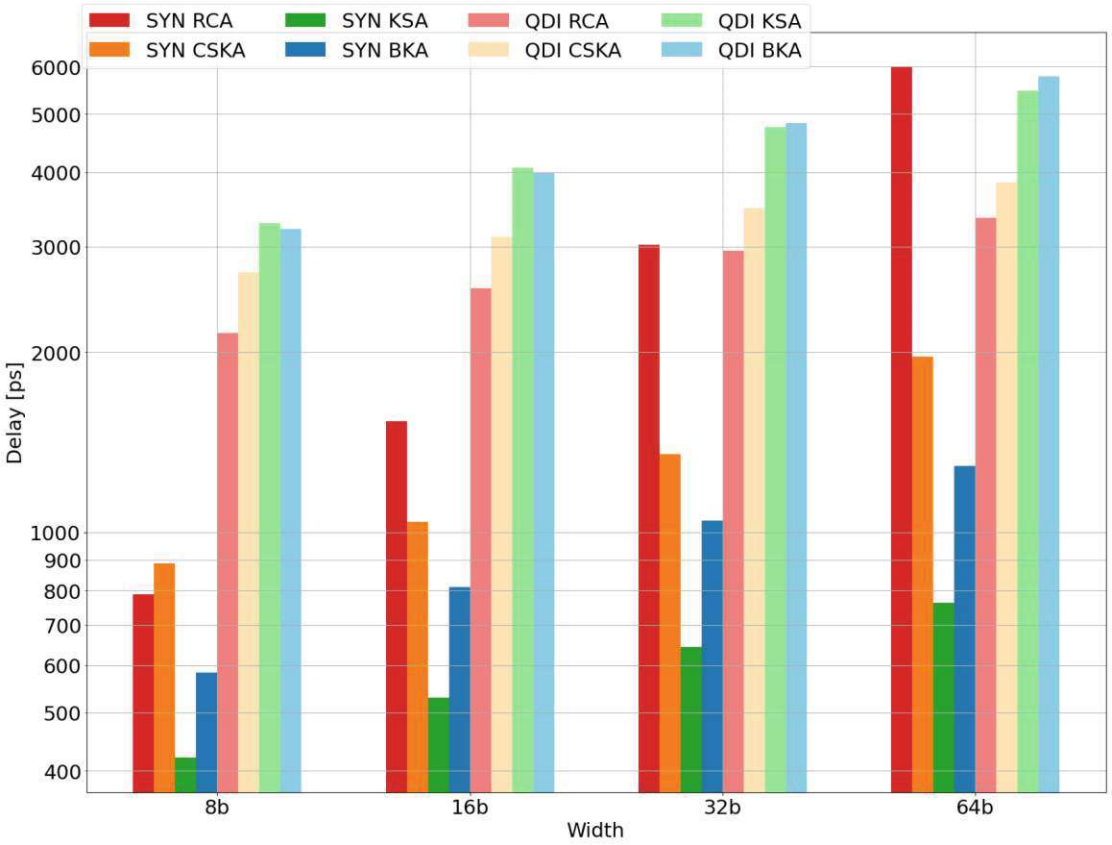

Figure 4.8: Cycle times

for any of the PPTAs. For the CSKA on the other hand the area overhead is much more immediately obvious, as it is simply an RCA with the additional bypass logic inserted into its structure. Additionally, the increased logic comes with an increase in interconnect which itself also occupies area of course.

In Figure 4.9 we can see a full comparison of the required area for both the synchronous and asynchronous adders. As can be seen and should be expected the QDI adders have a significantly increased area requirement, when compared to their synchronous counterparts. Most of this has already been outlined in Subsection 3.2.2, but naturally we can expect the area requirement of the asynchronous implementations to be at least twice that of the synchronous due to the expansion from single-rail to dual-rail. As also already stated in Subsection 3.2.2 some gates, such as the XOR, are more complicated to realize in dual-rail and therefore incur even more area overhead. Finally, the CD has to be considered. As stated multiple times throughout this thesis the CD consists mainly of Muller C gates, which are expensive both in terms of area and performance.

To further analyze the actual area overhead we can see the area usage in μm^2 for the synchronous adders in Table 4.1. Furthermore, Table 4.2 shows us the relative increase

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

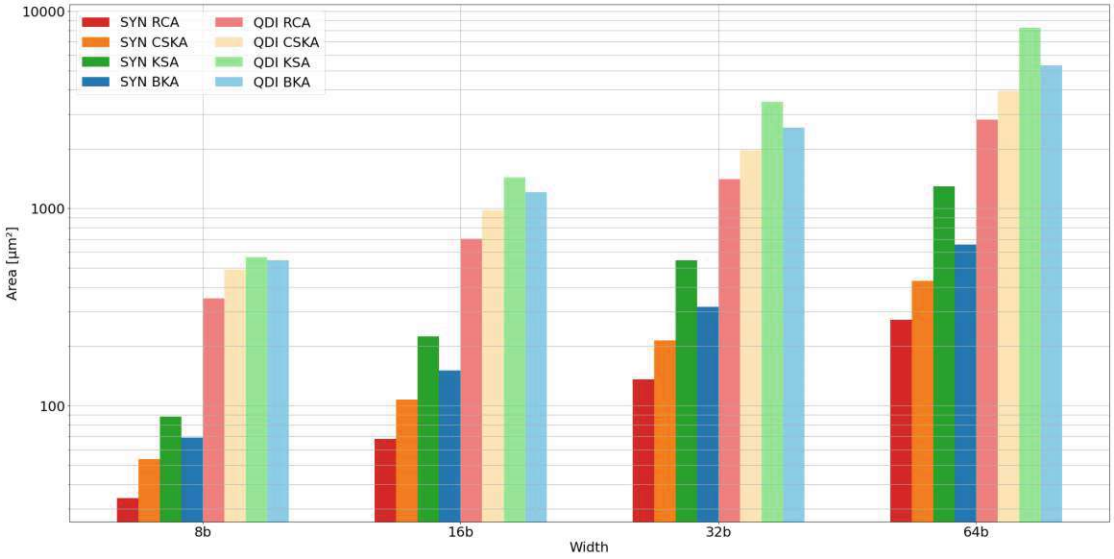

Figure 4.9: Area usage

Width	RCA	CSKA	KSA	BKA
8	34.048	53.732	88.312	69.16
16	68.096	107.464	224.504	151.088
32	136.192	214.927	547.960	318.136
64	272.384	429.856	1297.016	655.424

Table 4.1: Area usage [μm^2] of the synchronous adders

Width	CSKA	KSA	BKA
8	57.81%	159.38%	103.13%
16	57.81%	229.69%	121.88%
32	57.81%	302.34%	133.59%
64	57.81%	376.17%	140.63%

Table 4.2: Area usage increase of the synchronous adders relative to the RCA

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Width	RCA	CSKA	KSA	BKA
$8 \rightarrow 16$	100%	100%	154.22%	118.46%
$16 \rightarrow 32$	100%	100%	144.08%	110.56%
$32 \rightarrow 64$	100%	100%	136.70%	106.02%

Table 4.3: Area scaling of the synchronous adders

Width	RCA	CSKA	KSA	BKA
8	349.790	489.706	567.910	547.694
16	703.038	982.870	1434.006	1210.300
32	1409.534	1969.198	3485.398	2564.506
64	2822.526	3941.854	8226.582	5301.912

Table 4.4: Area usage [μm^2] of the asynchronous adders

Width	RCA	CSKA	KSA	BKA
8	927.34%	811.39%	543.07%	691.92%
16	932.42%	814.60%	538.74%	701.06%
32	934.96%	816.22%	536.07%	706.10%
64	936.23%	817.02%	534.27%	708.93%

Table 4.5: Area overhead of the QDI adders in [%] relative to their synchronous counterparts

in area usage of the synchronous adders in %, relative to the synchronous RCA. As we can see the area overhead of the bypass logic of the CSKA is relatively the same at every width. In contrast to this both PPTAs grow relatively larger with increasing width as compared to the RCA. In the case of the KSA it starts out at $\sim$ 2.6 times the size of the RCA at 8 bits and grows to $\sim$ 4.8 times the size of the RCA at 64 bit. The BKA on the other hand has a more reasonable scaling, starting out at roughly twice the size of the RCA and growing to $\sim$ 2.4 times the size at 64 bit.

Table 4.3 shows how the area of synchronous adders scales with the width. Due to their designs both the RCA and CSKA grow perfectly linearly, which can be seen here as the area simply doubles at double the width. For the PPTAs the relative overhead incurred by the respective tree structure is most prominent at the smallest width and then relatively decreases with increasing width. For the KSA the structure can be seen in Figure 3.3 and for the BKA it can be seen in Figure 3.4.

As Figure 4.9 already showed, and as Table 4.4 now confirms, the asynchronous adders require significantly more area to implement. Table 4.5 shows this same fact but expressed as a percentage. This means that, for example, the 16 bit wide asynchronous RCA has 932.42% more area than its synchronous counterpart. From Table 4.5 we can see that the asynchronous RCA is roughly 10.3 times the size of its synchronous counterpart at every width, although the relative area overhead slowly increases with increasing width. This

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Width	CSKA	KSA	BKA
8	40.00%	62.36%	56.58%
16	39.80%	103.97%	72.15%
32	39.71%	147.27%	81.94%
64	39.66%	191.46%	87.84%

Table 4.6: Area usage increase of the asynchronous adders relative to the RCA

Width	RCA	CSKA	KSA	BKA
$8 \rightarrow 16$	100.99%	100.71%	152.51%	120.98%
$16 \rightarrow 32$	100.49%	100.35%	143.05%	111.89%
$32 \rightarrow 64$	100.25%	100.18%	136.03%	106.74%

Table 4.7: Area scaling of the asynchronous adders

is because the CD is organized in a tree structure and this tree would require slightly more than double the total nodes to combine twice the amount of done signals. When these results are then compared to the CSKA, KSA and BKA we notice that the area overhead from switching to a QDI implementation is relatively smaller. This is because the synchronous RCA constitutes only of interconnected FAs which exist as a standard cell in the library. A single such FA has an area of $4.256\mu m^2$ whereas building the same functionality from 2 XOR gates, 2 AND gates and 1 OR gate would yield a total area of $6.916\mu m^2$. However, this exact thing happens when the RCA is implemented in NCLX, since the NCLX FA is simply built from the NCLX versions of the underlying logic gates, as described in Subsection 3.2.2. The same principle also applies to the CSKA which also uses FAs from the library in the synchronous case. The PPTAs instead have the carry generation and sum logic already split in the synchronous case and thus don't use any FAs to begin with, therefore the relative area overhead when implemented in NCLX is lower.

Similarly to the synchronous case, Table 4.7 shows the scaling of the asynchronous adders relative to the width. The first thing to take notice of is that both the RCA and CSKA don't exactly double in size for twice the width, but increase slightly more. Furthermore, this increase also decreases slightly with increasing width, this is, as stated previously, because the tree structure of the CD becomes more efficient for bigger widths. All three of RCA, CSKA and BKA have slightly worse scaling in the asynchronous case when compared to the synchronous case, meaning that the relative increase in adder area relative to the increase in adder width is a higher percentage for these three architectures for the asynchronous case. The KSA, however, scales slightly better in the asynchronous case than in the synchronous case. However, the overall differences in scaling are very minor for all four adders when comparing the synchronous and asynchronous cases.

Finally, we can see in Table 4.6 the comparison of the asynchronous adders to the asynchronous RCA. The NCLX adders incur much less area overhead when compared

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

to the NCLX RCA, as compared to the synchronous case. As stated previously, this is mainly because the synchronous RCA, and the CSKA to a lesser extent, benefit from the optimized FA library cell, whereas their asynchronous implementations have to build the FA from the basic logic gates and are thus already much less area optimized when compared to the PPTAs, than in the synchronous case.

4.2.1 Conclusion

As would be expected, the RCA is the most area efficient implementation. In the synchronous case the CSKA has a fixed overhead and both it and the RCA scale linearly with the width, requiring double the area at double the width. The synchronous PPTAs in comparison not only require significantly more area from the outset, but also scale worse, more than doubling their respective area requirements at double the width. Since the tree structure of the PPTAs grows more efficient with growing size, the relative increase with the width slowly decreases. Overall, the main advantage of the RCA, and to a lesser extent the CSKA, is the optimized FA cell from the library that they can make use of.

The asynchronous adders in comparison are of course much larger than their synchronous counterparts. The difference is greatest for the RCA and least for the KSA. Since their is no compact, optimized cell for an asynchronous FA the RCA suffers from having the FAs expanded to their basic logic gates, which then are also expanded to dual-rail and require a CD. However, this also means that the relative advantage regarding the area that the RCA has in the synchronous case is significantly less in the asynchronous case.

4.3 Number of Nets

The number of nets in the designs is a complicated topic. Due to the issues outlined in Section 3.2 and more specifically in Subsection 3.2.1 there is a mismatch between the number of nets that the tool reports and the number of nets we would expect based on a schematic of the design. Due to the issues with the optimization the tool has not optimized away or combined any nets. Thus, we have a higher then expected number of nets for each design other then the RCA. The total number of nets per adder and for each bit width is shown in Figure 4.10.

For the following equations n will be the width of the adder, additionally the formulas only consider $n : n = 2^i,\ i \in \mathbb{N}$.

The tables in the following subsections will provided a validation of the given formulas versus the reported number of nets for each design. The main purpose is to show that the formulas yield the correct number for the examined bit widths, rather than show the difference between what is expected and what is reported. As the following explanations will show, the formulas already account for the aforementioned difficulties and limitations of the design tool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

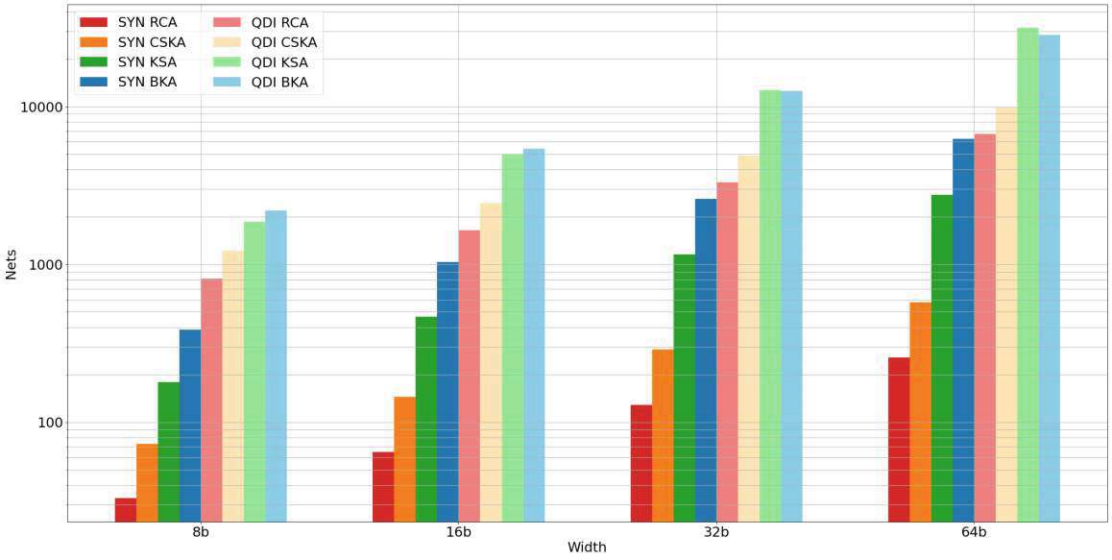

Figure 4.10: Number of nets

Width	Reported	Calculated
8	33	33
16	65	65
32	129	129
64	257	257

Table 4.8: Nets of the synchronous RCA

4.3.1 Synchronous

4.3.1.1 RCA

The number of nets for the synchronous RCA is given in Equation (4.1). The N_{ports} gives the ports of the adder, $3 \cdot n$ as there are two operands and the sum, and $+2$ for the carry-in and carry-out. The $N_{carries}$ gives the number of internal carry-signals that connect the individual FAs.

$$\begin{aligned} N_{ports} &= 3 \cdot n + 2 \\ N_{carries} &= n - 1 \\ N_{RCA} &= N_{ports} + N_{carries} \end{aligned} \tag{4.1}$$

4.3.1.2 CSKA

Equation (4.2) gives the number of nets for the synchronous CSKA. N_{ports} gives the number of ports of the design. It is of course the same as for the RCA.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Width	Reported	Calculated
8	73	73
16	145	145
32	289	289
64	577	577

Table 4.9: Nets of the synchronous CSKA

n_{blocks} is the number of ripple-carry-blocks that comprise the adder. In our case the block size was fixed at 4 bit and since we only considered adder widths that are multiples of 4 this leaves us with an expression that evaluates to a whole number of blocks for our use cases. N_{block} in turn gives the number of nets in each such block. Each block has $3 \cdot 4 + 2$ ports, which are the 4 bit of the two operands, as well as the 4 bit of the sum and the carry-in/out. Furthermore, each block also contains 4 internal carry-signals, as well as the 4 propagate-signals from the skip-logic, and finally the select signal that controls the MUX. Finally, the blocks themselves are connected with $n_{blocks} - 1$ carry-signals.

$$\begin{aligned} N_{ports} &= 3 \cdot n + 2 \\ n_{blocks} &= n/4 \\ N_{block} &= 3 \cdot 4 + 2 + 4 + 4 + 1 \\ N_{carries} &= (n_{blocks} - 1) \\ N_{CSKA} &= N_{ports} + n_{blocks} \cdot N_{block} + N_{carries} \end{aligned} \tag{4.2}$$

4.3.1.3 KSA

For the KSA determining the number of nets is more complicated. The formula for the number of nets in the design is given in Equation (4.3). The layout of the prefix tree is illustrated in Figure 3.3, but the actual implementation is done in terms of stages, where each horizontal row in the tree structure is one stage in the design.

n_{stages} gives the number of stages for a given adder width. The total number of nodes is shown in the equation as n_{nodes}, due to the varying amount of nodes per stage it is easier to express the total over the whole design. Similarly it is easier to express the total amount of nets across all stages, rather than an amount per stage, which is given as N_{stages}. For N_{stages} each processing node contributes three signals, which are the 2 inputs and 1 internal signal. Furthermore, each of the "empty" white nodes in the prefix tree is simply a connection, that is the outputs of such a node are simply equal to its inputs. However, due to aforementioned limitations with the design tool each such nodes still contributes 2 nets to the total, since it has 2 inputs/outputs. Thus, at each stage $2 \cdot n$ nets are added in total for all the outputs of the nodes. Additionally, $n - 1$ carry-signals, which are the outputs of the prefix tree are needed for the sum computation. It is of note here that the nth carry-signal is already counted as it is also the carry-out and in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Width	Reported	Calculated
8	180	180
16	468	468
32	1156	1156
64	2756	2756

Table 4.10: Nets of the synchronous KSA

such instances where these nets are equivalent and on the same hierarchical level of the design, the design compiler was able to combine them in spite of the aforementioned constraints. It is for the same reason that the "empty" nodes in the tree only contribute two nets per node rather than 4. Finally, N_{group} nets, the group carry signals, are needed as interconnection between the stages, since on each stage of the tree both the calculated group-generate and group-propagate signal have to be passed on to the next stage.

$$\begin{aligned} N_{ports} &= 3 \cdot n + 2 \\ n_{stages} &= log_2(n) \\ n_{nodes} &= n \cdot (n_{stages} - 1) + 1 \\ N_{stages} &= n_{stages} \cdot 2 \cdot n + 3 \cdot n_{nodes} \\ N_{group} &= n_{stages} \cdot 2 \cdot n \\ N_{carries} &= n - 1 \\ N_{KSA} &= N_{ports} + N_{carries} + N_{stages} + N_{group} \end{aligned} \tag{4.3}$$

4.3.1.4 BKA

Finally, the equation for the total nets of the BKA is shown in Equation (4.4). Similarly to the KSA the BKA was also implemented in stages, see Figure 3.4, where each horizontal row in the prefix tree is one stage. However, unlike the KSA, the overall prefix tree is essentially made of two smaller trees, where one is appended to the other. Because of this determining the total number of nets in the design is more complicated.

N_{ports}, $N_{carries}$ and N_{group} are the same as for the KSA. Although for N_{group} it should be noted that this only means that it is defined the same way, the actual number is of course different, since n_{stages} is different. n_{stages} gives the total number of stages in the tree, as described in Subsection 3.1.4. The signals connecting the stages are combined into N_{stages}, in total 4 nets per bit and stage are needed, for both the carry generate and carry propagate signals, for both input and output.

The number of nodes in a tree of size n is recursively defined as $n_{nodes}(n)$. For $n = 1$ no processing node is needed as the required group carry signals for a single bit group are equal to the single bit signals that pass into the tree, see Subsection 1.3.2. The initial tree needs $n - 1$ nodes to calculate the group carry signals for half of the bit positions, and then a similar tree for size $\frac{n}{2}$ is needed to calculate the remaining half.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Width	Reported	Calculated
8	180	180
16	468	468
32	1156	1156
64	2756	2756

Table 4.11: Nets of the synchronous BKA

Finally, each node contributes 2 nets for its inputs and one net for the internal signal, as well as 2 nets for the outputs. As with the KSA, for the "empty" nodes only one net per input/output pair is needed, thus we elect to count the inputs only for the processing nodes, and the output for all nodes. The total number of nets needed across all nodes is given as N_{nodes}. If all of the above is put together then we arrive at the total number of nets for the complete adder, given in Equation (4.4) as N_{BKA}.

$$\begin{aligned}
N_{ports} &= 3 \cdot n + 2 \\
N_{carries} &= n - 1 \\
N_{group} &= n_{stages} \cdot 2 \cdot n \\
n_{stages} &= 2 \cdot log_2(n) - 1 \\
N_{stages} &= n_{stages} \cdot 4 \cdot n \\
n_{nodes}(1) &= 0 \\
n_{nodes}(n) &= (n-1) + n_{nodes}(\frac{n}{2}) \\
N_{nodes} &= 3 \cdot n_{nodes} + n_{stages} \cdot 2 \cdot n \\
N_{BKA} &= N_{ports} + N_{carries} + N_{stages} + N_{group} + N_{nodes}
\end{aligned} \tag{4.4}$$

4.3.2 Asynchronous

4.3.2.1 CD

Before we look at any specific adder we will look at the CD. The CDs are organized in tree structures to improve the performance, this is because in theory the path through a binary tree should be of length $log_2(n)$ for n input signals, which correspond to the input done signals.

$$\begin{aligned}
N_{c-gate} &= 5 \\
N_{ports}(x) &= x + 1 \\
N_{CD}(1) &= 1 \\
N_{CD}(2) &= N_{c-gate} + N_{ports}(2) = 8 \\
N_{CD}(n) &= N_{CD}(\lfloor (n+1)/2 \rfloor) + N_{CD}(\lfloor \frac{n}{2} \rfloor) + 2 + N_{c-gate} + N_{ports}(n)
\end{aligned} \tag{4.5}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Equation (4.5) shows the recursive formula for the number of nets in a given C gate tree. First, each C gate requires 5 nets. Subsubsection 3.2.2.4 describes the implementation of the Muller C gate. From this we can see that of the 5 total nets, 3 are for the ports of the gate, and 2 are for connecting the inverter from the output of the AOI structure to the output port. Furthermore, each recursively defined branch of the tree has N_{ports} depending on the number of input signals to that branch, plus one output. The $N_{CD}(1)$ and $N_{CD}(2)$ are the stopping points of the recursion, where the leafs of the tree are generated. In the case of a branch containing only one signal, which happens for example when there are 3 input signals to the parent branch, then no Muller C gate is needed. In the case of 2 input signals to the leaf node a single C gate is instantiated which together with the required ports requires a total of 8 nets. Finally, the formula recursively generates branches each dealing with half of the input signals, in case of an odd number of signals the extra signal is added to the left child branch.

4.3.2.2 FA

Subsubsection 3.2.2.3 describes the creation of the NCLX FA. In particular 2 AND gates, 2 XOR gates and one OR gate are required to build one such FA. The resulting number of nets is given in Equation (4.6).

$$\begin{aligned}
N_{ports} &= 5 \cdot 2 + 1 = 11 \\
N_{xor} &= 2 \cdot 3 = 6 \\
N_{and} &= 2 \cdot 2 = 4 \\
N_{intermediate} &= 2 \\
N_{done} &= 3 + 3 = 6 \\
N_{FA} &= N_{ports} + N_{xor} + N_{and} + N_{intermediate} + N_{done} + N_{CD}(6) = 83
\end{aligned} \tag{4.6}$$

First, we have the ports, as given by N_{ports}, in total there are 11 ports, which are the 5 dual-rail signals, A, B, C_{in}, S and C_{out}, as well as the done signal of the FA itself. Next we have the 2 XOR gates, for simplicity the nets of both XORs were combined into one value. In total the asynchronous XORs contribute 8 nets, which are 2 times 3 nets for the outputs of the 3 synchronous AND gates that form the first level of the XOR, with each AND gate corresponding to one input combination. The fourth gate is not counted here because it can be shared with the asynchronous AND gates that we will discuss next. Next are the asynchronous AND gates, which contribute a total of 4 nets. This is simply because each asynchronous AND gate contributes an intermediate dual-rail signal, hence $2 \cdot 2$ nets. Then we also count the single intermediate signal coming out of the first asynchronous XOR gate which naturally contributes 2 nets. Finally, we have a total of 6 internal done signals coming out of the synchronous OR gates that detect whether a dual-rail signal has transitioned. Of course 3 of these are for the input-CD of the 3 inputs and 3 are for the 3 intermediate signals, 2 from the asynchronous AND gates and 1 from the first asynchronous XOR gate. These internal done signals go into a C gate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Width	Reported	Calculated
8	811	811
16	1643	1643
32	3323	3323
64	6715	6715

Table 4.12: Nets of the asynchronous RCA

tree which itself needs 54 nets to deal with the 6 signals, as the previous subsection, Subsubsection 4.3.2.1, showed. In total this leaves us with 83 nets for a single NCLX FA.

4.3.2.3 RCA

For the asynchronous RCA Equation (4.7) gives us the number of nets depending on the width n. First, an asynchronous n bit RCA needs N_{ports} nets for its ports. These are from the 2 n bit input vectors, A and B, and the n bit output vector S, as well as the 2 dual-rail ports, C_{in} and C_{out}. Additionally, the RCA itself of course also produces a done signal. Next we have the internal carry signals that interconnect the individual FAs, given as $N_{carries}$. These are dual-rail signals of course, so they contribute twice as many nets. The RCA of course consists of n FAs, which as a result contribute $n \cdot N_{FA}$ nets in total to the adder. Finally, for the n FAs n internal done signals are needed, which are collected in a C gate tree, which itself then contributes $N_{CD}(n)$ nets. Table 4.12 confirms that the number of nets calculated using this formula matches the numbers reported by Synopsys Design Compiler.

$$\begin{aligned} N_{ports} &= 3 \cdot n \cdot 2 + 2 \cdot 2 + 1 \\ N_{carries} &= (n-1) \cdot 2 \\ N_{done} &= n \\ N_{FA} &= N_{ports} + N_{carries} + n \cdot N_{FA} + N_{done} + N_{CD}(n) \end{aligned} \tag{4.7}$$

4.3.2.4 CSKA

The asynchronous CSKA is significantly more complicated than its synchronous counterpart regarding the total number of nets it requires. First, we look at the number of ports of the adder itself, which is the same as for the asynchronous RCA. Next we have the number of nets for the asynchronous versions of the XOR, AND and MUX. For the AND and XOR it has to be mentioned that the number is different than for the NCLX FA, as described in Subsubsection 4.3.2.2. This is because of previously mentioned optimization regarding the shared synchronous AND gate in the implementation of the FA. Since the asynchronous gates were simply instantiated here for the CSKA and due to the previously mentioned limitations regarding the optimizations that the tool can perform the total number of nets for the XOR and AND gates combined is higher here since it also includes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the ports of these gates themselves, which was not the case in the section of the FA. In total each XOR contributes 3 times 2 nets for its ports plus 4 internal signals for a total of 10 nets. Similarly the AND contributes 3 times 2 nets for its ports with no internal signals for a total of 6 nets. And lastly the MUX contributes 4 times 2 nets for its ports, with 4 internal signals as well, for a total of 12 nets. Next we look at the nets of each 4 bit block. First, each such block needs a total of 29 nets for its ports, which is detailed in N_{ports}^{block}, where we can see that we have 3 4-bit dual-rail ports, for A, B and S respectively, as well as the 2 single-bit dual-rail ports for C_{in} and C_{out}, and finally the done signal output for the block. In addition to the N_{ports}^{block} each block also contains 4 FAs, which is given as $4 \cdot N_{FA}$. Furthermore, each block needs 4 XOR gates, 3 AND gates and 1 MUX to create the skip logic. The 4 XORs and 3 AND gates each also produce a single dual-rail output that has to considered. Finally, the internal CD of the block combines a total of 11 done signals, 4 of which come from the 4 FAs, 4 of which come from the 4 XORs, and the last 3 of which come from the 3 AND gates. This of course also adds the $N_{CD}(11)$ nets from the C gate tree itself. All of this is combined in N_{block}, n_{blocks} such blocks are then needed to form the CSKA. In the end the total number of nets for the entire CSKA is given as N_{CSKA}, which combines its number of ports, N_{ports}, the n_{blocks} blocks of N_{block} nets each, and $N_{carries}$ carry signals that connect these blocks. Of course the individual done signals, which in total contribute N_{done} nets, of each block are collected in a C gate tree, which contributes $N_{CD}(n_{blocks})$ nets. All of this is described in Equation (4.8) and then validated in Table 4.13.

$$
\begin{aligned}
N_{ports} &= 3 \cdot n \cdot 2 + 2 \cdot 2 + 1 \\
n_{blocks} &= n/4 \\
N_{xor} &= 3 \cdot 2 + 4 \\
N_{and} &= 3 \cdot 2 \\
N_{mux} &= 4 \cdot 2 + 4 \\
N_{ports}^{block} &= 3 \cdot 4 \cdot 2 + 2 \cdot 2 + 1 \\
N_{block} &= N_{ports}^{block} + 4 \cdot N_{FA} + 4 \cdot (N_{xor} + 2) + 3 \cdot (N_{and} + 2) + \\
&\quad 4 \cdot 2 + N_{mux} + 11 + N_{CD}(11) \\
N_{carries} &= (n_{blocks} - 1) \cdot 2 \\
N_{done} &= n_{blocks} \\
N_{CSKA} &= N_{ports} + n_{blocks} \cdot N_{block} + N_{carries} + N_{done} + N_{CD}(n_{blocks})
\end{aligned}
\tag{4.8}
$$

4.3.2.5 KSA

The asynchronous KSA is quite complex, which is reflected in the number of nets, as described in Equation (4.9). The number of stages, n_{stages}, is the same as for the synchronous version, the explanation for which is given in Subsubsection 4.3.1.3. The same is also true for the total number of processing nodes over all the stages, n_{nodes}, which is described in the same section. The number of ports for the adder, n_{ports}, is the same as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Width	Reported	Calculated
8	1221	1221
16	2451	2451
32	4915	4915
64	9851	9851

Table 4.13: Nets of the asynchronous CSKA

for all the other asynchronous adders, and the number of nets for an asynchronous AND or XOR gate is of course also the same as it was for the previous asynchronous adders, the numbers are given as N_{and} and N_{xor} respectively. The NCLX KSA is the first of the asynchronous designs that also uses an asynchronous OR gate, its number of nets is given as N_{or}, unsurprisingly it is the same as for the AND gate. The first new component, that is tracked separately, is the "preprocessing" logic, which computes the single bit carry propagate and carry generate signals for all n inputs. This "preprocessing" logic consists, as the formulas in Subsection 1.3.2 describe, of an asynchronous AND gate for the generate signal, and an asynchronous XOR gate for the propagate signal. In total this logic requires N_{GP} nets. At the output of the prefix tree, which will be discussed a bit later, are the group carry signals. However, for the sum only the group generate signals are relevant. These are listed as $N_{carries}$, and as usual for dual-rail signals require twice as many nets. The computation of the actual sum itself is achieved with n XOR gates, the nets of which are listed as N_{sum}. Also similarly to the synchronous KSA, there are the group carry signals, that connect the individual stages of the tree, for these a total of N_{group} nets are necessary. This number is similar to the number for the synchronous KSA, except that there are twice as many nets for the dual-rail implementation of course. However, the final $\frac{n}{2}$ propagate signals, which are unused and therefore not counted in the synchronous version, have to be checked by the CD and therefore are now also counted. Finally, the overall design combines N_{done} done signals in its CD, which of course also contribute a net each to the total. These N_{done} signals are the results of the input CD for A, B and C_{in}, as well as for the $\frac{n}{2}$ group generate and group propagate signals that are produced by the last stage of the tree. The inputs have to be checked as they are then fed into the preprocessing logic, and the final outputs of the tree have to be checked as they are then fed to the sum logic. The intermediate group carry signals, as well as the output of the preprocessing logic, however, need not be checked since each stage of the tree has its own input CD. Naturally, an appropriately sized CD is needed for these N_{done} done signals, the total number of nets for it is expressed as $N_{cd}(N_{done})$.

Now we still need to discuss the stages of the prefix tree. As with the synchronous version it is easier to discuss the total over all stages, rather than each individual stage. The resulting number is given in the equation as N_{stages}. First, the total ports over all stages are given as the first term in N_{stages}. Because the white, non-filled nodes in the tree contain no logic they also only contribute one net for each input/output pair of ports, rather than two. Since there are a total of n_{nodes} processing nodes in the tree, we can

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

express the total number of nets for the ports over all stages as $2 \cdot n_{nodes} \cdot 2 + 2 \cdot n_{stages} \cdot n \cdot 2$. This is because we have two pairs of input/output signals per bit, but only the ones for the processing nodes should be counted twice. Of course, since these are dual-rail signals they have to be counted twice. Additionally, every stage outputs its own done signals, which is the final term, n_{stages} within this first term. Next is the actual logic within the processing nodes. It consists of 2 AND and one OR gate, which require $2 \cdot N_{and} + N_{or}$ nets, as well as one internal dual-rail signal, of course n_{nodes} of these are required in total. This yields a total of 3 done signals per processing node, for a total of $3 \cdot n_{nodes}$ done signals over all stages.

This is where it gets complicated. The number of nets for the CDs have to be tracked per stages and then summed, because $N_{CD}(a) + N_{CD}(b) \neq N_{CD}(a + b)$. This is expressed in the term $N_{stages-CD}$. The number of nodes in an individual stag can be expressed as $n - 2^i$ for $i \in [0, n_{stages})$ and $i \in \mathcal{N}$. As stated, each node has to check 3 signals for the generation of the done signal, thus the CD contributes a total of $N_{CD}(3 \cdot (n - 2^i))$ nets for each stage. Of course, this is then simply summed over all stages to give the total amount of nets for the CDs of the stages.

If we put all of the above together then we get the total number of nets for the entire asynchronous KSA in Equation (4.9) as N_{KSA}. The calculated values and the values reported by DC for the considered widths can be found in Table 4.14.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Width	Reported	Calculated
8	1866	1866
16	4995	4995
32	12778	12778
64	31595	31595

Table 4.14: Nets of the asynchronous KSA

$$
\begin{aligned}
n_{stages} &= log_2(n) \\
n_{nodes} &= n \cdot (n_{stages} - 1) + 1 \\
N_{ports} &= 3 \cdot n \cdot 2 + 2 \cdot 2 + 1 \\
N_{xor} &= 3 \cdot 2 + 4 \\
N_{and} &= 3 \cdot 2 \\
N_{or} &= 3 \cdot 2 \\
N_{GP} &= n \cdot (N_{and} + N_{xor}) \\
N_{carries} &= (n-1) \cdot 2 \\
N_{sum} &= n \cdot N_{xor} \\
N_{group} &= 2 \cdot n_{stages} \cdot n \cdot 2 + \frac{n}{2} \cdot 2 \\
N_{done} &= 2 \cdot n + 2 \cdot \frac{n}{2} + n_{stages} + 1 \\
N_{stages-CD} &= \sum_{i=0}^{n_{stages}-1} N_{CD}(3 \cdot (n - 2^i)) \\
N_{stages} &= ((2 \cdot n_{nodes} + 2 \cdot n_{stages} \cdot n) \cdot 2 + n_{stages}) \\
&\quad + n_{nodes} \cdot (2 \cdot N_{and} + N_{or} + 2) \\
&\quad + 3 \cdot n_{nodes} + N_{stages-CD} \\
N_{KSA} &= N_{ports} + N_{GP} + N_{carries} + N_{sum} \\
&\quad + N_{stages} + N_{group} + N_{done} + N_{CD}(N_{done})
\end{aligned}
\tag{4.9}
$$

4.3.2.6 BKA

As with the synchronous version, the asynchronous BKA is the most complex design, at least regarding the number of nets. Equation (4.10) describes the number of nets within the design, broken apart in the usual format of this section. Due to the differences in design, as can be seen by comparing Figure 3.3 with Figure 3.4, which are the prefix trees of the KSA and BKA respectively, the BKA has almost double the stages of the KSA, which is reflected in n_{stages}. The number of ports, n_{ports}, as well as the number of nets of the asynchronous gates, e.g. AND, OR and XOR, are of course the same as before. Furthermore, the number of nets for the "preprocessing", i.e., the base case of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

group carry signals, N_{GP}, the number of nets for calculating the sum from the group carries, N_{sum}, the interconnect of the prefix tree, N_{group}, and the amount of done signal, N_{done}, are all the same as for the KSA, although it has to be said that, since n_{stages} is different, these numbers are only the same in definition, but not in value.

The main differences are of course to be found within the prefix tree. N_{stages} gives the total number of nets across all stages, but for simplicity we will go through it line by line. The term of the first line gives the ports for each stage across all stages, each stage has the 2 n bit wide group carry signals as both input and output, as well as the done signal output of each stage. The next term gives us the signals that are tracked per node. First, we start with the input ports. As has been mentioned previously, the way that the signals are defined and optimized by the tool is not always straightforward, so in general the tool declares the input side signals of the group carries within a stage only for those bit positions that contain a node, for the other positions the input and output signals are combined, and thus counted as only one net. Thus, we have the 2 input signals and 1 intermediate signal per node, which of course contribute 2 nets, since they are dual rail signals. Then we have the nets contributed by the logic within each node, namely 2 AND and 1 OR gate. Finally, each node requires 5 nets for its done signal, which are the 4 inputs per node and the intermediate signal. Next we have the output ports of each stage, as stated any bit position in each stage that contains no processing node combines the inputs and outputs into one net per signal. Of course, each stage contains its own CD. The total number of nets for the individual CDs across all stages is given as $N_{stages-CD}$. Unlike the KSA, it is not quite so easy to express the number of nodes in each stage, thus the formula for $N_{stage-modes}(i)$ gives the number of nodes in stage i. As stated before, there are 5 signals in each node for the CD, so $N_{CD}(5 \cdot N_{stage-nodes}(i))$ gives the number of nets for the CD for stage i.

In the end, as with any other adder so far, it is simply a matter of putting all of this together to arrive at the total number of nets for the BKA, which is given as N_{BKA}. The comparison, showing that the formula matches the reported values, is given in Table 4.15.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Width	Reported	Calculated
8	2204	2204
16	5403	5403
32	12609	12609
64	28550	28550

Table 4.15: Nets of the asynchronous BKA

$$
\begin{aligned}
n_{stages} &= 2 \cdot log_2(n) - 1 \\
n_{nodes}(1) &= 0 \\
n_{nodes}(n) &= (n-1) + n_{nodes}(\frac{n}{2}) \\
N_{ports} &= 3 \cdot n \cdot 2 + 2 \cdot 2 + 1 \\
N_{xor} &= 3 \cdot 2 + 4 \\
N_{and} &= 3 \cdot 2 \\
N_{or} &= 3 \cdot 2 \\
N_{carries} &= (n-1) \cdot 2 \\
N_{GP} &= n \cdot (N_{and} + N_{xor}) \\
N_{sum} &= n \cdot N_{xor} \\
N_{group} &= 2 \cdot n_{stages} \cdot n \cdot 2 + \frac{n}{2} \cdot 2 \\
N_{done} &= 2 \cdot n + 2 \cdot \frac{n}{2} + n_{stages} + 1 \\
N_{stage-nodes}(i) &= \begin{cases} \frac{n}{2^{i+1}}, & \text{if } i < log_2(n) \\ \lfloor \frac{n-2}{2^{2log_2(n)-1-i}} \rfloor, & \text{otherwise} \end{cases} \\
N_{stages-CD} &= \sum_{i=0}^{n_{stages}-1} N_{CD}(5 \cdot N_{stage-nodes}(i)) \\
N_{stages} &= n_{stages} \cdot (4 \cdot n \cdot 2 + 1) \\
&\quad + n_{nodes} \cdot (3 \cdot 2 + 2 \cdot N_{and} + N_{or} + 5) \\
&\quad + n_{stages} \cdot 2 \cdot n \cdot 2 \\
&\quad + N_{stages-CD} \\
N_{BKA} &= N_{ports} + N_{carries} + N_{GP} \\
&\quad + N_{sum} + N_{group} + N_{done} \\
&\quad + N_{stages} + N_{CD}(N_{done})
\end{aligned}
\tag{4.10}
$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Conclusion

As the previous chapter showed, the quantitative analysis and comparison of the synchronous and asynchronous implementations along three main axis, namely performance, area usage and number of nets, shows that the asynchronous adders cannot compete with the most optimized synchronous designs, yet they are also more complex, as expressed in the number of nets, and require significantly more area to implement.

The NCLX RCA emerges as the clear winner among the QDI implementations and is the only asynchronous design that can outperform its synchronous counterpart in terms of performance. However, it requires roughly 10 times as much area to implement without accounting for the area cost of any interconnect. A quick look at the number of nets reveals that this area overhead may be a rather low estimate given the serious difference in the number of nets, and thus in complexity and possible area cost of the interconnect.

An important contribution of this work is that it clearly identifies the CD as the main bottleneck for performance and as one of the main contributors to the high area usage.

With all of that said, it should still be noted that all of the asynchronous implementations outperform the synchronous RCA at the highest examined bit width. This shows that one of the main draws of asynchronous logic, the average-case performance, is not without any merit. Furthermore, it can be stated here that, while not a focus of this work, asynchronous logic has other draws as well, particularly in regards to robustness and fault tolerance.

If we turn our attention to applications that require robustness rather than speed, then the asynchronous implementations become contenders again. The synchronous adders would usual require additional functionality to achieve the desired level of robustness, such as a triple modular redundant implementation for example. The asynchronous adders in comparison can achieve such robustness inherently. Thus, the results tell us that the QDI RCA could be a viable alternative to the synchronous versions in any such

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

cases were the focus lies on characteristics other than performance or area usage, given that it is faster and also more robust at the same time.

5.1 Future Work

As stated in the previous section, the CD is the main bottleneck regarding the performance of the asynchronous adders. The CD for all of the asynchronous implementations was realized in the form of a tree structure to achieve the desired hysteresis behavior for more than two signals at a time, as discussed in Subsubsection 3.2.2.4. Thus, both the C gate itself as well as the C gate tree as a whole are therefore immediate candidates for further optimization. In particular, the effects of combining a tree structure and a chain, or simply having an unbalanced tree, could be explored. This is because the simple tree structure has all of the input signals either at the same depth or at most one level closer to the root, if the tree is only partially filled on its final level. However, this does not at all reflect in any way the actual logic and the order in which these input signals would logically arrive.

The NCLX FA is a good example of this. It has a total of 6 signals for which completion must be detected, namely the 3 single bit inputs (A, B, C_{in}) and a further 3 intermediate signals (the result of $a \wedge b$, the result of $a \oplus b$ and the result of $C_{in} \wedge (a \oplus b)$). These intermediate signals are all at different logic depths within the combinational logic, and thus it would be reasonable to modify the structure of the CD such that their position within an unbalanced tree would reflect the expected timing of these signals. Listing 2 which shows the actual implementation of an asynchronous RCA also shows this quite well. As can be seen all of the individual done signals are combined into a tree, and thus are all at the same depth, even though the expectation would be that they become available in sequence, starting from the FA for the **LSB** and ending with the FA for the **MSB**. Figure 5.1 gives an example of what such an optimization may look like. While the path for the input CD signals is now longer the path for the intermediate signals that we expect to be generated last is now much shorter. In the simple implementation the signal which would have been expected to arrive last would still have to traverse the entire tree, while it is now the case that the signal expected to arrive last would have to go through one expensive gate only. The viability and possible effectiveness of such an optimization could be explored for all instances of the CD in the various architectures.

Another case that may be worthwhile for further consideration is the embedding of the adders in a pipeline. The expectation here is that the delay overhead caused by the CD is more negligible because the done signals for the logic need only be ready by the time that the next buffer stage has latched its new input. Furthermore, the preceding buffer stage already performs an input CD and therefore this part of the CD for the combinational logic may be omitted. Finally, depending on the characteristics of the surrounding pipeline stages, it might be of use to explore the influence that the asynchronous adders average-case performance has on the throughput of the pipeline.

To this end both of the works as summarized in Section 2.1 and Section 2.2, respectively

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

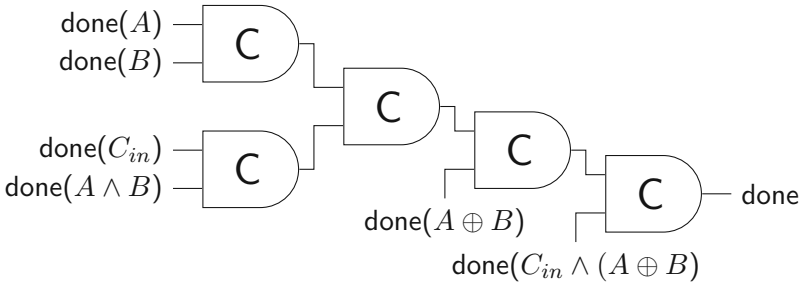

Figure 5.1: Modified C gate tree for the NCLX FA

show that the pipeline configuration is arguably more important than the adders themselves. In both works it is not the adder within the pipeline that informs the overall result. For [FP94] it is the memory which, as the overall slowest component, informs the timing of the pipeline. For the synchronous adders this of course means that regardless of their performance the clock period is fixed such that it accommodates this comparatively much slower component. This of course illustrates one of the often purported advantages of asynchronous logic, namely the average-case timing, and the benefits that can thus be gained in any scenario where the memory would not be involved. In the second paper [Kin96] on the other hand, it is the comparatively rigid timing of the second pipeline stage that eliminates all possible advantages that the asynchronous adders might have. Both of these works show that the configuration of the pipeline and the timing of the surrounding stages play an important part in analyzing the overall throughput. Therefore testing the asynchronous adders in a variety of different pipeline configurations and comparing the results with a synchronous implementation might show the overall impact that an asynchronous implementation can have on the performance of the whole pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

List of Tables

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Bibliography

[BK82] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, 1982.

[FP94] Mark A. Franklin and Tienyo Pan. Performance comparison of asynchronous adders. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC 1994, Salt Lake City, UT, USA, November 3-5, 1994*, pages 117–125. IEEE, 1994.

[Gar93] Jim D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In Stephen B. Furber and Martyn Edwards, editors, *Asynchronous Design Methodologies, Proceedings of the IFIP WG10.5 Working Conference on Asynchronous Design Methodologies, Manchester, UK, 31 March - 2 April, 1993*, volume A-28 of *IFIP Transactions*, pages 181–192. North-Holland, 1993.

[GPW55] Bruce Gilchrist, James H. Pomerene, and S. Y. Wong. Fast carry logic for digital computers. *IRE Trans. Electron. Comput.*, 4(4):133–136, 1955.

[Kin96] D. J. Kinniment. An evaluation of asynchronous addition. *IEEE Trans. Very Large Scale Integr. Syst.*, 4(1):137–140, 1996.

[KS73] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, 22(8):786–793, 1973.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[Lyn96] Thomas Walker Lynch. Binary adders. 1996.

[Mar86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Comput.*, 1(4):226–234, 1986.

[Mar90] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, AUSCRYPT '90, page 263–278, Cambridge, MA, USA, 1990. MIT Press.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[MT98] Rajit Manohar and José A. Tierno. Asynchronous parallel prefix computation. *IEEE Trans. Computers*, 47(11):1244–1252, 1998.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[Rei60] George W. Reitwiesner. The determination of carry propagation length for binary addition. *IRE Trans. Electron. Comput.*, 9(1):35–38, 1960.

[Skl60] Jack Sklansky. An evaluation of several two-summand binary adders. *IRE Trans. Electron. Comput.*, 9(2):213–226, 1960.

[Sut89] Ivan E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, 1989.

[vdS85] Jan L. A. van de Snepscheut. *Trace Theory and VLSJ Design*, volume 200 of *Lecture Notes in Computer Science*. Springer, 1985.

[Ver88] Tom Verhoeff. Delay-insensitive codes - an overview. *Distributed Comput.*, 3(1):1–8, 1988.

[VNT63] J. Von Neumann and A.H. Taub. *Collected Works*. Number Bd. 1. Pergamon Press, 1963.

[WH11] N.H.E. Weste and D.M. Harris. *Integrated Circuit Design*. Pearson, 2011.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.