Universität Bremen

Application of Alternative Number Formats for Digital Signal Processing

An Evaluation on Sets Of Real Numbers

Dem Fachbereich Physik, Elektrotechnik und Informationstechnik
der Universität Bremen

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

genehmigte Dissertation

von

M.Sc. Moritz Bärthel

Referent:	Professor Dr.-Ing. Steffen Paul
Korreferent:	Professor Dr.-Ing. Guillermo Payá Vayá
Prüfer:	Professor Dr.-Ing. Alberto Garcia-Ortiz
Prüfer:	Professor Dr.-Ing. Jochen Rust

Eingereicht am: 21.12.2023

Tag des Promotionskolloquiums: 12.02.2024

Abstract

The encoding and manipulation of binary numbers in digital systems is a major design challenge for modern high-performance computer arithmetic and digital signal processing. Besides unsigned and signed integer representation, fixed point and floating point are the two state-of-the-art choices for implementing real number arithmetic in digital systems. In the past decades a few other approaches were proposed, but none achieved to provide a general-purpose alternative to the established fixed and floating point. In recent years, however, machine learning and other applications with requirements for processing huge amounts of data increased the demands for alternative number representations in order to efficiently handle these new challenges. A further motivation for the development of digital number format alternatives are inefficiencies and inaccuracies within the IEEE standard for floating point, which are tackled by the universal number (unum) format, proposed in 2015.

This thesis deals with the Sets Of Real Numbers (SORN) format, which is a derivative of the unum approach. The SORN format represents the entire real numbers with a dedicated set of exact values and intervals, and implements arithmetic operations with pre-computed lookup tables, realized with simple Boolean logic circuits. This approach leads to low-complex and low-latency hardware designs, while the represented precision is rather low, compared to standard formats. Due to these properties, SORNs are well-suited for constraining high-dimensional optimization problems by means of preprocessing, as well as for implementing threshold-based algorithms.
This thesis presents implementations of SORN arithmetic operations on register-transfer level and evaluates on different SORN datatype representations, as well as optimizations such as the introduction of fused SORN arithmetic. In order to facilitate a design space exploration with SORNs, an automated design flow is presented, which provides complete SORN datapaths for different algorithms and applications. By using this design flow, SORN arithmetic is applied within edge detection for image processing and as preprocessing for detection algorithms in wireless MIMO communication. The presented results show that the proposed SORN approach achieves an improvement of the hardware measures for the respective designs, while providing similar algorithmic performance as the state-of-the-art implementations with standard formats.

Kurzfassung

Die Kodierung und Verarbeitung von Binärzahlen in digitalen Systemen stellt eine große Herausforderung für moderne und hoch-performante Computerarithmetik und digitale Signalverarbeitung dar. Neben der Darstellung von positiven und negativen Ganzzahlen sind Festkomma- und Gleitkommadarstellungen die beiden meist genutzten Ansätze für die Implementierung von arithmetischen Operationen mit reellen Zahlen in digitalen Systemen. In den vergangenen Jahrzehnten wurden einige weitere Verfahren vorgestellt, jedoch stellte keines eine ernsthafte Konkurrenz zu den etablierten Fest- und Gleitkommaformaten dar. Innerhalb der letzten Jahre haben jedoch das maschinelle Lernen und andere Anwendungen mit enormem Datendurchsatz die Nachfrage nach alternativen Zahlendarstellungen stark gesteigert, um diese neuen Herausforderungen effizient bewältigen zu können. Eine weitere Motivation für die Entwicklung alternativer digitaler Zahlenformate sind Ineffizienz und Ungenauigkeiten innerhalb des IEEE-Standards für Gleitkomma-Arithmetik, die durch das 2015 vorgestellte universal number (unum)-Format ausgeräumt werden sollen.

Diese Dissertation beschäftigt sich mit dem Sets Of Real Numbers (SORN)-Format, einer Weiterentwicklung des unum-Ansatzes. Das SORN-Format repräsentiert alle reellen Zahlen mit einem dedizierten Satz von exakten Werten und Intervallen und implementiert arithmetische Operationen mit vorberechneten Tabellen, die mit einfachen booleschen Logikschaltungen realisiert werden. Dieser Ansatz führt zu Hardware-Designs mit geringer Laufzeit und Komplexität, wobei die darstellbare Genauigkeit im Vergleich zu Standardformaten geringer ausfällt. Aufgrund dieser Eigenschaften eignen sich SORNs für die Vereinfachung hochdimensionaler Optimierungsprobleme durch Vorverarbeitung, sowie für die Umsetzung von Schwellwert-basierten Algorithmen.
In dieser Dissertation werden Implementierungen von arithmetischen Operationen mit SORNs auf Register-Transfer-Ebene vorgestellt und für verschiedene SORN Datentypen ausgewertet, sowie Optimierungen wie die Einführung von Fused-SORN-Arithmetik untersucht. Zur Evaluierung des Entwurfsraums mit SORNs wird ein automatisiertes Entwurfswerkzeug vorgestellt, das vollständige SORN-Datenpfade für verschiedene Algorithmen und Anwendungen umsetzt. Mit Hilfe dieses Entwurfswerkzeuges wird die SORN-Arithmetik bei Kantenerkennung in der Bildverarbeitung sowie als Vorverarbeitung für Detektionsverfahren in der drahtlosen MIMO-Kommunikation eingesetzt. Die vorgestellten Ergebnisse zeigen, dass für alle SORN-Implementierungen eine Verbesserung der Hardware-Parameter der jeweiligen Designs erreicht werden kann, wobei ein vergleichbares algorithmisches Verhalten wie bei Referenz-Implementierungen mit Standardformaten erzielt wird.

Acknowledgements

The thesis presented here is the result of my research work at the Institute of Electrodynamics and Microelectronics, Department of Communication Electronics (ITEM.me), at the University of Bremen. First of all, I would like to thank Professor Steffen Paul for providing me with the opportunity to write this thesis and for supervising my research with numerous valuable suggestions in terms of content and methodology. I am also very thankful for the great amount of self-responsibility he offered me for implementing my ideas. Further, I would like to thank Professor Guillermo Payá Vayá for serving as a reviewer, as well as Professor Alberto Garcia-Ortiz and Professor Jochen Rust for serving as examiners.
During his time at ITEM, Jochen Rust was also the one who introduced me to the world of unums and SORNs, while I was working on my master's thesis. I am very grateful to him, since this set the basis for my research, which eventually led to this thesis. I also would like to thank all my other colleagues at ITEM, who contributed to this work in various ways and made me enjoy my daily work. On behalf of all of them I would like to mention and thank Nils Hülsmeier, who shared my enthusiasm for SORNs and with whom I had a lot of fruitful discussions on the topic, and Alexandra Avenius, who supported me with numerous organizational, language-related and off-topic matters. I am also very grateful to both Nils and Alexandra for proof-reading my thesis.
Further, I would like to thank the students who supported my work with their bachelor's and master's theses, and would like to name Simon Knobbe, Chen Yuxing and Jan Schäfer on behalf of all of them.
Finally, I would like to express my deepest gratitude to my family and friends who supported me all those years and helped me becoming the person I am. Above all, I am infinitely grateful to Wiebke for her love, patience and her continued support in every conceivable way.

Moritz Bärthel
February 2024

Contents

1 Introduction

One of the first things a student learns about in a digital design course is the immense growth of the semiconductor industry within the second half of the last century, and how this development is described by Moore's law. In 1965 Gordon Moore postulated that the number of transistors in an integrated circuit (IC) would roughly double every two years [Moo06], a forecast that should hold for the next 50 years. This massive increase of components per device was the main driver for an improving computing performance of different microprocessor generations, and became possible because the size of single transistors could be reduced from micrometers to a few nanometers. During the last decade, however, Moore's law seems to have come to an end, due to fundamental physical limits like the size of an atom, which eventually limits the minimal size of a single transistor [Cro16, Wal16, Bla16]. In order to preserve the outcome of Moore's law and further improve the performance of microprocessors, new approaches have to be considered, both on the technological or physical level [RK22], as well as on the algorithmic or arithmetic level [Sou21]. The latter includes the field of computer arithmetic and digital number formats, which, since the beginning of digital computers, is dominated by solely two approaches to encode and manipulate numbers in digital systems: integer or fixed point (FxD) arithmetic on one hand, and floating point (FP) arithmetic on the other. Digital computer arithmetic contains a huge innovation potential, which is why a trend towards more alternative approaches could be observed in recent years. This includes formats proposed many years ago, like the Residue Number System (RNS) and stochastic computing (SC), which are now applied for quantum-resistant cryptography [MS20], as well as new approaches like posits and adaptions of the existing floating point formats for applications in machine learning [Joh18] or radio astronomy [Gun23].

To get an overview of the role of alternative number formats within the computer arithmetic research community, figure 1.1 breaks down the publications on alternative number formats other than integer, fixed and standard floating point at the IEEE International Symposium on Computer Arithmetic (ARITH), starting from 1985, when the first floating point standard was released by the IEEE [IEEE85]. Even though one single conference cannot reflect all the research going on in this field, the ARITH is considered the leading conference for computer arithmetic and gives a good overview of the most important topics. Figure 1.1 depicts the percentage of publications on alternative formats per issue, taken from in total 715 publications, with a mean of about 30 papers per issue. Until 2015, the symposium was held every two years, and annually since then. The total share of publications on alternative number formats at ARITH over all considered issues is about 20%, which shows that alternative arithmetic does not play a significant role within this research community. A major part of the considered publications deals with modular arithmetic like the RNS and Galois field (GF)/finite field (FF)

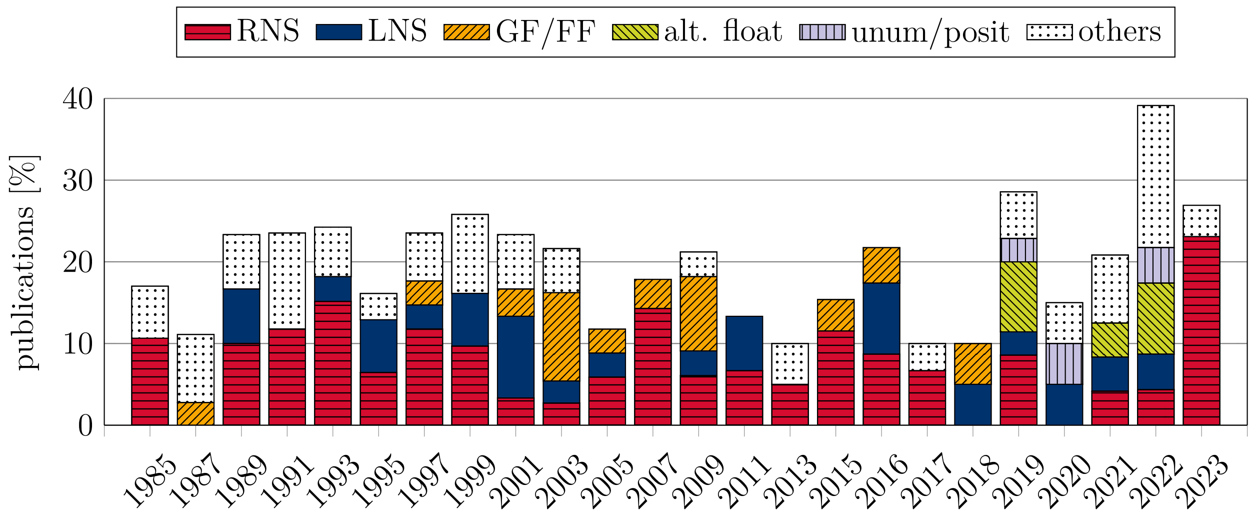


Figure 1.1.: Percentage of publications on alternative number formats at ARITH symposium per issue, taken from 715 publications accessed via [ACS] and [IEEE].

arithmetic, which are mainly used in cryptography applications. Another group of publications considers a Logarithmic Number System (LNS) in order to ease the hardware implementation of multiplication and division for standard floating point. Since the late 2010s machine learning is the major emerging topic in most research fields related to digital systems, which also shows its impact at ARITH. For machine learning implementations, quantization and energy-efficient number formats are of high importance [RSL+21], which is reflected by ARITH publications on alternative floating point formats in recent years.

Another part of this trend on alternative formats for machine learning is the posit format, which is a derivative of the universal number (unum) format, proposed by John Gustafson in 2015. The unum format is an alternative floating point approach which includes interval arithmetic (IA), and targets the improvement of inefficiencies and inaccuracies within the IEEE floating point standard [Gus15]. The approach was presented as a keynote at ARITH 2015, leading to a panel discussion on the advantages and disadvantages of both unums and floats in the next years edition between Gustafson and William Kahan, one of the founders of the IEEE floating point standard [GK16]. In 2019 the first unum related publication was presented at ARITH, followed by a few more posit publications in the following years.

Since the ARITH community does not show enough interest in alternative number format research in general, and the unum approach in particular, Gustafson started his own Conference for Next Generation Arithmetic (CoNGA) in 2018 [CoN18], which deals with all kinds of alternative computer arithmetic approaches, including unums and its two derivatives: posit and Sets Of Real Numbers (SORN). The latter is a low-complex and low-precision version of the interval-based unums, and will be the main topic of this thesis.

Among other alternative digital number formats like posit, LNS, RNS and standard floating point adaptions, SORNs are part of a recent trend towards more diverse and application-specific number representation in current and future computer arithmetic and digital signal processing [CRR+21, Sou21].

1.1. Contributions

The Sets Of Real Numbers (SORN) number format is a derivation of the original unum format and was first proposed by John Gustafson in 2016 [Gus16], also referred to as type-II unums. At the time of writing this, and to the best of the authors knowledge, there exists no publications on SORN arithmetic other than the original and the authors publications presented in this thesis. The primary contributions of this work therefore are the implementation, evaluation, optimization and application of the SORN number format. This includes a comprehensive mathematical formulation of the original approach, as well as adaptions of the SORN datatype structure, the development of an automated SORN design flow, evaluations on the hardware complexity of basic SORN arithmetic components, and the introduction of fused SORN arithmetic. Further contributions are the application of the SORN number format for a SORN based edge detection implementation used for image processing, and within a preprocessing component for symbol detection in a wireless MIMO communication system, implemented for two different detectors. An overview of the described contributions is shown in figure 1.2. A list of publications is given in the following.

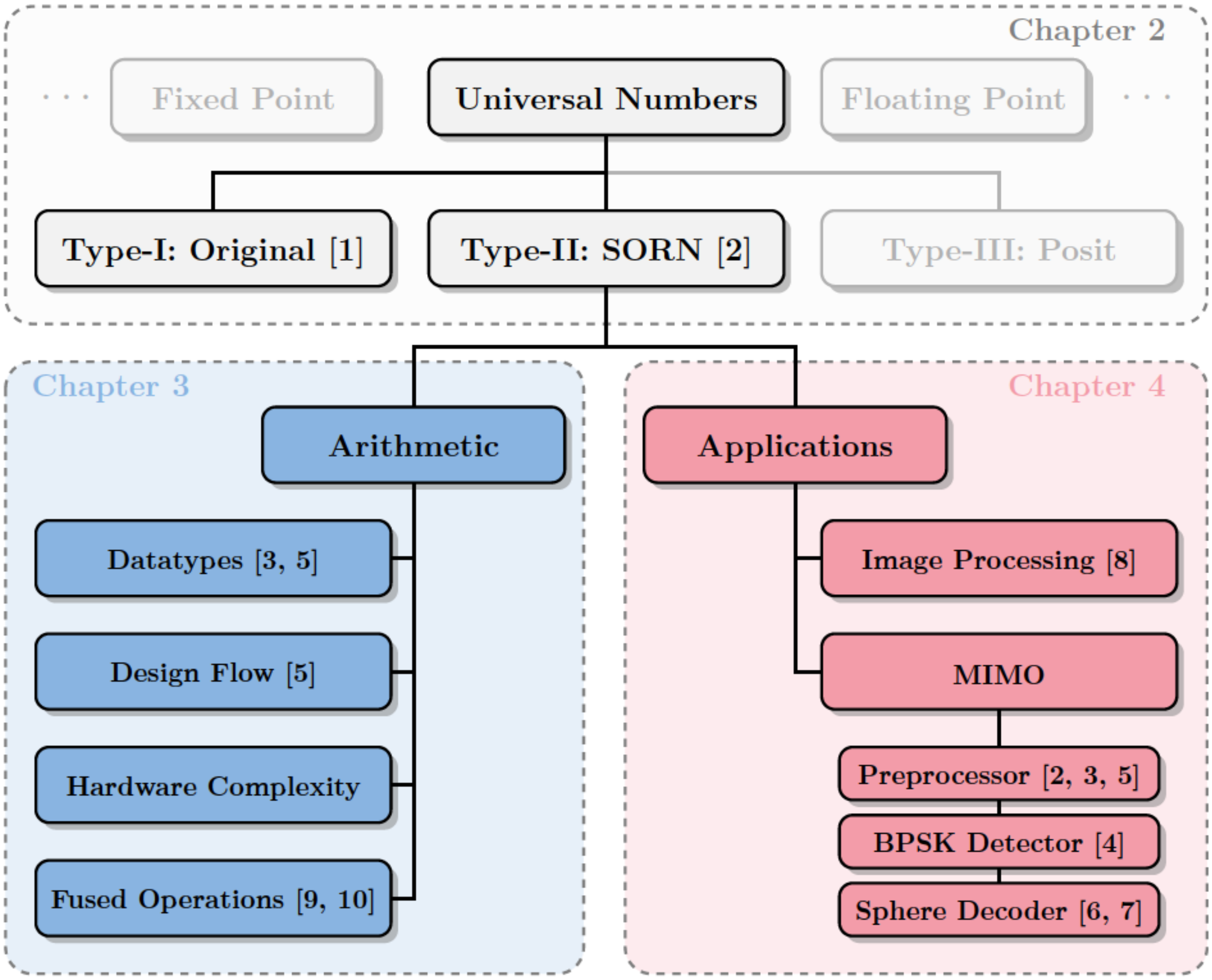


Figure 1.2.: Primary contributions of the thesis with chapter organization.

List of Publications

The primary contributions of this thesis are based on the following peer-reviewed publications:

[1] M. Bärthel, J. Rust, and S. Paul. Hardware Implementation of Basic Arithmetics and Elementary Functions for Unum Computing. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 125–129, Oct 2018.

[2] M. Bärthel, P. Seidel, J. Rust, and S. Paul. SORN Arithmetic for MIMO Symbol Detection - Exploration of the Type-2 Unum Format. In *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*. IEEE, 23-26 June 2019.

[3] M. Bärthel, J. Rust, and S. Paul. Application-Specific Analysis of Different SORN Datatypes for Unum Type-2-Based Arithmetic. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[4] M. Bärthel, J. Rust, and S. Paul. Combining Fixed-Point and SORN Arithmetic in a MIMO BPSK-Symbol Detection Architecture. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 181–184, 2020.

[5] J. Rust, M. Bärthel, P. Seidel, and S. Paul. A Hardware Generator for SORN Arithmetic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4842–4853, 2020.

[6] S. Knobbe, M. Bärthel, S. Paul, and J. Rust. Complexity Reduction for Sphere Decoding using Unum-Type-II-Based SORN-Arithmetic. In *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, 2020.

[7] M. Bärthel, S. Knobbe, J. Rust, and S. Paul. Hardware Implementation of a Latency-Reduced Sphere Decoder With SORN Preprocessing. *IEEE Access*, 9:91387–91401, 2021.

[8] M. Bärthel, N. Hülsmeier, J. Rust, and S. Paul. On the Implementation of Edge Detection Algorithms with SORN Arithmetic. In *Next Generation Arithmetic: Third International Conference, CoNGA 2022, Singapore, March 1–3, 2022, Revised Selected Papers*, pages 1–13. Springer, 2022.

[9] M. Bärthel, J. Rust, J. Gustafson, and S. Paul. Improving the Precision of SORN Arithmetic by Introducing Fused Operations. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 258–262, 2022.

[10] M. Bärthel, C. Yuxing, N. Hülsmeier, J. Rust, and S. Paul. Fused Three-Input SORN Arithmetic. In *Next Generation Arithmetic: 4th International Conference, CoNGA 2023, Singapore, March 1-2, 2023, Proceedings*, pages 101–113. Springer, 2023.

1.2. Outline

The outline of the thesis is structured according to the main contributions, as shown in figure 1.2. Chapter 2 introduces state-of-the-art (SOTA) approaches for digital arithmetic and number formats including the standard representations for integers, fixed and IEEE floating point, as well as alternative number formats like RNS or LNS, interval arithmetic and the unum format with its type-III version of posits. Chapter 3 introduces the type-II unum and SORN approach, and discusses datatypes, an automated design flow, hardware complexity and fused operations for SORNs. Chapter 4 presents applications for SORN arithmetic, namely edge detection for image processing and symbol detection in wireless MIMO communication. Chapter 5 summarizes and concludes the thesis and gives an outlook on possible future work.

2 Digital Arithmetic and Number Formats

According to the Oxford Dictionary, a number is *"a word or symbol that represents an amount or a quantity"* [HLB+20]. The amount, quantity or value of something can be displayed or written in many different ways. Just as we can speak or write words and text in different languages, numbers can be represented in different notations and with different symbols, the numerals. A numeral is *"a sign or symbol that represents a number"* [HLB+20]. In the history of humankind, various numeral systems have been used to express numbers and perform calculations, before the Arabic numerals were established, as we use them today. Some prominent historical examples, some of which are still used today, are the unary representation, also called tally marks 卌||, the Maya numerals $\overset{\bullet\bullet}{\overline{}}$, and the roman numerals XII (here all representing the value 7) [Ifr00].
Nowadays, the Arabic numerals {0 1 2 3 4 5 6 7 8 9} are used to represent numbers within a decimal number system, where a number consists of several digits, each with a value 0-9. In an integer number, every digit is multiplied with 10^n, where n is the position of the digit within the number, starting with $n = 0$ for the rightmost digit and increasing by 1 towards the left. For rational numbers with a fractional part, containing a value between 0 and 1, a separator sign, usually a dot ".", is used. Every digit to the right of the separator is also multiplied with 10^n, starting with $n = -1$ for the leftmost digit and decreasing towards the right, as depicted in figure 2.1.

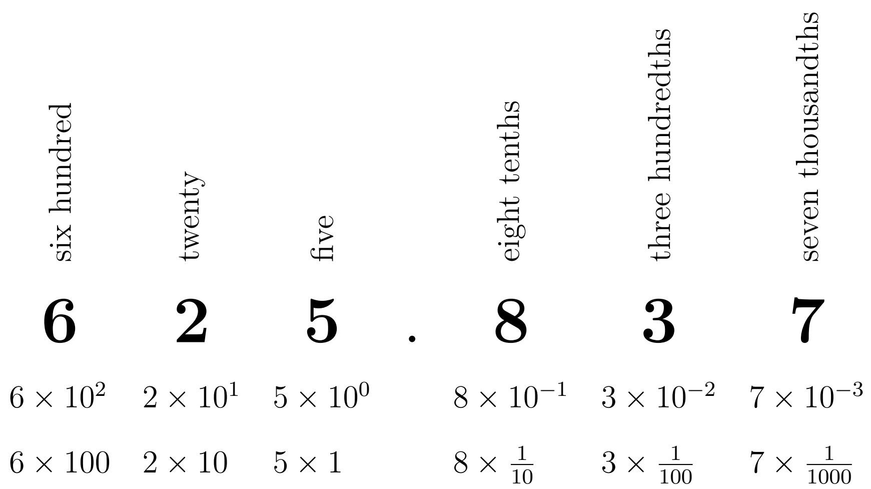

six hundred
twenty
five
eight tenths
three hundredths
seven thousandths
6 2 5 . 8 3 7
6×10^2 2×10^1 5×10^0 8×10^{-1} 3×10^{-2} 7×10^{-3}
6×100 2×10 5×1 $8 \times \frac{1}{10}$ $3 \times \frac{1}{100}$ $7 \times \frac{1}{1000}$

Figure 2.1.: Visualization of the decimal number system.

In mathematical terms, this can be written with the general formula for positional systems

$$v_B = \sum_{n=-M}^{N-1} d_n B^n \tag{2.1}$$

with the represented value v, the digits d_n with $0 \leq d_n < B$ and the base B, also called radix. For the decimal number system, $B = 10$ and $d_n \in \{0, \ldots, 9\}$. M and N are the number of digits for the fractional and integer part, respectively [BSMM08]. Note that in this notation, the position index starts with the rightmost digit $n = -M$ and increases towards the left until $n = N - 1$. The index $_B$ can be used with the actual value to indicate the base of the current number system, for example 625.837_{10}, sometimes also $625.837|_{10}$. This notation is mostly applied when different bases are used within one equation or algorithm. If only one base is used, the base index is usually omitted. Throughout this work, the described index notation is mainly used to distinguish the numerals 0 and 1 used in decimal representation (0_{10} and 1_{10}), or in binary representation (0_2 and 1_2), as introduced below.

The decimal number system, as we use it today, started to appear in Europe somewhere between the 10th and 13th century anno Domini (AD) [Ifr00]. After that, it took several more centuries before it became established as the matter of course with which we use it today in everyday life. Then, in the late 1930th and early 1940th, a new way of representing numbers became important with the development of the first digital computers. These computers were built with vacuum tubes, which were later replaced by transistors, nowadays still the main component of modern digital computing systems [O'R08]. Both vacuum tubes and transistors behave like switches, able to distinguish between solely two different states: off or on, low or high voltage, 0 or 1. Due to this behavior, computers are not suited to use decimal arithmetic. Instead, the binary number system is applied.

Like the other number systems mentioned above, the binary system is much older than the first computers. One famous mathematician who contributed to arithmetic with binary numbers was Leibniz in the early 18th century [O'R08]. Nevertheless, with the invention of digital computers, this approach using only two different numerals became more important than ever. The binary and decimal number systems are both positional systems applying to equation (2.1), the only differences are the base B and the digits d_n. For binary representation, $B = 2$ and the digits $d_n \in \{0, 1\}$ are also called bits. How to encode and represent binary numbers, for example as integer or real values, and how to perform arithmetic operations with them, is the discipline of digital number formats and computer arithmetic, located within computer science. This topic will be explained in detail in this chapter.

As a general remark, it has to be mentioned that all the different number systems discussed so far are clearly distinguishable because of their outer forms and the utilized numerals. Now, entering the topic of digital number formats, all encoded in binary representation, this distinction is not so obvious anymore. The outer form of all the different formats, that will be covered in the following, is a bit string composed of zeros and ones. Whether the bit string encodes an integer or fixed point value, with or without a sign, and at which position the radix point is located (if there is one), is only visible from the system configuration and type definitions.

Further, it has to be mentioned that the implementation of arithmetic operations for different number formats, using transistors in electronic circuits, for example on field-programmable gate arrays (FPGAs), with integrated circuits (ICs) or application-specific integrated circuits (ASICs), respectively, is based on the theory of digital design. This includes Boolean Logic and Algebra, as well as the principles of combinatorial and sequential logic design [HH13]. These topics will not be covered here, a basic knowledge is assumed in the following.

2.1. Integer Format

In mathematics, the most elementary set of numbers are the natural or whole numbers $\mathbb{N}$, defined either without zero as $\mathbb{N} = \{1, 2, 3, \dots\}$, also referred to as positive integers [JJ43], or with zero $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ [BSMM08]. These are the most intuitive numbers that are commonly used in everyday life. When negative values (and zero) are also considered, the set is called integers $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$. For computer arithmetic, these two sets build the most basic number format, the integer format. The natural numbers $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$ are hereby referred to as unsigned integers, whereas the set $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ is called signed integers.

Unsigned The unsigned integers are built straight forward following the definition of a positional number system in equation (2.1) with base $B = 2$ and digits $d_n \in \{0, 1\}$, now called bits b_n

$$v_2 = \sum_{n=0}^{N-1} b_n 2^n \tag{2.2}$$

with the total number of bits N. The index $n = 0$ corresponds to the rightmost bit, the least significant bit (LSB), and $n = N - 1$ to the leftmost bit, the most significant bit (MSB). Table 2.1 shows the encoding of unsigned integers with up to $N = 3$ bits and the respective decimal equivalents [HH13]. The minimum and maximum values in this representation are given in table 2.2.

Table 2.1.: Unsigned integer encoding for up to $N = 3$ bits.

unsigned integer			**decimal**
$N = 1$	$N = 2$	$N = 3$	
0	00	000	0
1	01	001	1
	10	010	2
	11	011	3
		100	4
		101	5
		110	6
		111	7

Table 2.2.: Minimum and maximum values for unsigned and signed integer formats with N bits.

encoding	**min value**		**max value**	
	decimal	binary	decimal	binary
unsigned	0	$00\dots0$	2^N-1	$11\dots1$
signed-magnitude	$-2^{N-1}+1$	$11\dots1$	$2^{N-1}-1$	$01\dots1$
one's complement	$-2^{N-1}+1$	$10\dots0$	$2^{N-1}-1$	$01\dots1$
two's complement	-2^{N-1}	$10\dots0$	$2^{N-1}-1$	$01\dots1$

Signed For the representation of signed integer values $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$, three different encoding schemes can be used: the signed-magnitude, the one's complement and the two's complement encoding.

Signed-Magnitude Comparable to the decimal numbers, where an extra numeral is used to indicate that a number is negative, the signed-magnitude encoding reserves the MSB to represent the sign of a number. This bit is called the sign bit. Zero hereby indicates a positive, one a negative number. The remaining $N-1$ bits encode the absolute value of the number as an unsigned integer. The minimum and maximum value in this representation are given in table 2.2. Table 2.3 shows the encoding of signed-magnitude integers with $N=3$ bits and the respective decimal equivalents [Far04]. The drawback of signed-magnitude encoding is that it is not compatible with the rules of binary arithmetic, which will be discussed in section 2.1.1. In addition, the encoding contains redundancy with a negative and positive zero value [HH13].

One's Complement For creating the one's complement encoding of a negative number, its unsigned binary value is complemented, meaning that all bits are flipped to the opposite. Zeros become ones and vice versa. Positive values are not changed, their encoding is the same as for unsigned or signed-magnitude encoding. The minimum and maximum value in this representation are given in table 2.2. Table 2.3 shows the encoding of one's complement integers with $N=3$ bits [Far04]. One's complement numbers can be used with binary arithmetic (see section 2.1.1), but they don't solve the problem of redundant zero values.

Table 2.3.: Signed integer encoding for $N=3$ bits.

signed magnitude	**one's complement**	**two's complement**	**decimal**
000	000	000	0
001	001	001	1
010	010	010	2
011	011	011	3
100	111	-	-0
101	110	111	-1
110	101	110	-2
111	100	101	-3
-	-	100	-4

Two's Complement The two's complement representation is the most commonly used encoding for negative integers, because it applies to binary arithmetic and eliminates the redundancy of two zeros. As for the previous representations, positive values remain unchanged. For negative values, in a first step the unsigned integer value of a number is inverted, just as for one's complement. In a second step, a one is added to the inverted value. The following example shows the two's complement encoding of -5_{10} using $N = 4$ bits:

$$\begin{array}{rl} 0101 & \text{(unsigned value)} \\ \hline 1010 & \text{(inverted/one's complement)} \\ +\ 0001 & (\text{add } 1_2) \\ \hline 1011 & \text{(two's complement)} \end{array} \tag{2.3}$$

The minimum and maximum value in two's complement representation are given in table 2.2. Table 2.3 shows the encoding of two's complement integers with $N = 3$ bits [Far04]. An alternative way of interpreting two's complement numbers is to treat them as unsigned integers except the MSB, which is given a weight -2^{N-1}, instead of 2^{N-1} [HH13]. In this case the MSB acts as a weighted sign bit. Equation 2.2 can be rewritten in order to represent this behavior:

$$v_{\text{twosComp}} = -b_{N-1}2^{N-1} + \sum_{n=0}^{N-2} b_n 2^n \tag{2.4}$$

2.1.1. Addition

Arithmetic with binary numbers, encoded as unsigned and signed integers, derives from the concepts of decimal arithmetic, since both representations are positional number systems and differ solely in base and digits [Far04]. For decimal addition, the digits of two numbers are added positional wise, starting with the rightmost digit at position $n = 0$. When the result of this addition can not be displayed with a single digit, the first resulting digit contributes to the overall result, whereas the second digit is stored as so-called *carry* and contributes to the addition of the next two digits at position $n = 1$ [Far04]. This process is continued until the rightmost position $n = N - 1$. Due to the carry, it can happen that the result value requires an additional digit at position $n = N$. An example of this process is given in figure 2.2a.
Binary addition with unsigned integers follows the decimal approach, performing a bit wise addition, starting with the LSB at position $n = 0$. The addition of two 1_2-bits produces a 0_2 in the result and a 1_2 in the carry, which is applied to the next position. Figure 2.2b shows the binary addition of the two unsigned 4-bit values $a = 0110_2 = 6_{10}$ and $b = 0011_2 = 3_{10}$.

$$\begin{array}{rl} 5173 & \\ +\ \ 7854 & \\ 111 & \text{(carry)} \\ \hline 13027 & \end{array}$$

(a) decimal addition

$$\begin{array}{rl} 0110 & (= 6_{10}) \\ +\ \ 0011 & (= 3_{10}) \\ 11 & \text{(carry)} \\ \hline 1001 & (= 9_{10}) \end{array}$$

(b) binary addition

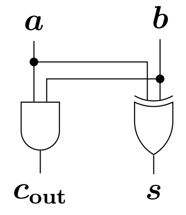
Figure 2.2.: Concepts for decimal and binary addition.

Carry Out In decimal arithmetic, an extra digit at position $n = N$ due to the carry, as shown in figure 2.2a, is simply added to the result value without any further considerations. In digital systems, however, the storage size of a binary number is limited to a fixed bitwidth N. If the addition of two fixed width unsigned integer numbers produces a carry that propagates to the N-th position, this is signaled by a carry output, because the result value requires more bits than the operands [Par10]. In a naive implementation, if the extra bit would be discarded, this would lead to a wrong result.

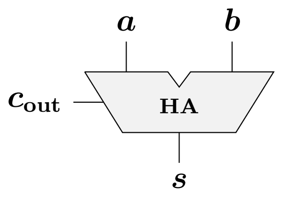
Implementation The functionality of binary addition for a single bit can be implemented with the so-called half adder (HA). This building block produces the sum s for two inputs a and b, along with the carry output c_{out}. The 1-bit half adder can be implemented with one AND and one XOR gate. Figure 2.3 shows the truth table, as well as the gate level and block level design. For adding values with more than one bit, single adder blocks can be connected, each processing the resulting bit for one position n. In order to handle carry values from previous stages, the full adder (FA) block can be used. It contains an additional carry input and can be implemented with two half adders and one OR gate. Figure 2.4 shows the truth table, gate and block level design of a full adder. The serial connection of one half adder, followed by $N - 1$ full adders is the simplest way of implementing an N-bit adder, the so-called *ripple-carry adder* (RCA), shown in figure 2.5. Other architectures like *carry-lookahead* or *prefix* can be used to implement adders with different properties, such as lower latency [HH13].

a	b	c_{out}	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) truth table



(b) gate level

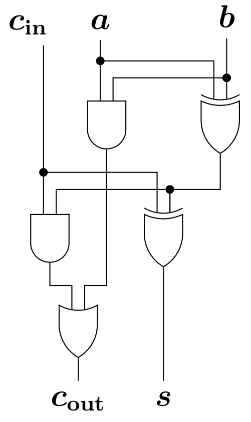


(c) block level

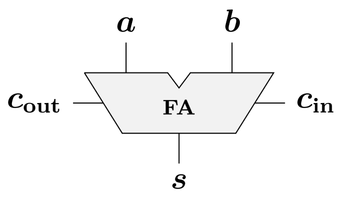
Figure 2.3.: Truth table, gate and block level architecture for a 1-bit half adder.

c_{in}	a	b	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) truth table



(b) gate level



(c) block level

Figure 2.4.: Truth table, gate and block level architecture for a 1-bit full adder.

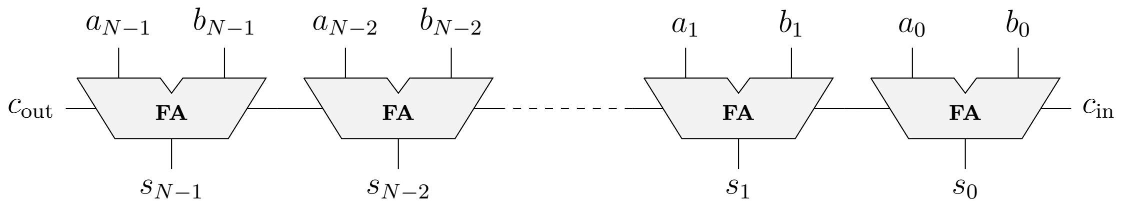


Figure 2.5.: Architecture of an N bit ripple-carry adder [HH13].

Signed Addition Signed addition or the subtraction of one binary integer from another can be realized by combining the unsigned integer addition with two's complement encoding. The operation $a - b$ can be rewritten as $a + (-b)$, where $(-b)$ is the negative two's complement encoding of b. Figure 2.6a shows the computation of $a-b$ with $a = 0110_2 = 6_{10}$, $b = 0011_2 = 3_{10}$, and $-b = 1101_2 = -3_{10}$, respectively. In contrast to the addition of two unsigned numbers, for signed addition the carry output has no significance [Par10]. In the depicted example, the output is correct only if the carry output is discarded.
Signed addition can be implemented with the described adder architecture without any changes. If a dedicated subtraction operation is to be implemented, the addition block can be used with a negated second input b and a carry input $c_{\text{in}} = 1_2$ for the LSB ($n = 0$), in order to obtain the negated two's complement encoding of the second input b [HH13].

Overflow When adding two negative values, i.e. subtracting a positive from a negative value, a so-called *overflow* is possible, as shown in figure 2.6b for $a - b$ with $a = 1010_2 = -6_{10}$, $b = 0011_2 = 3_{10}$, and $-b = 1101_2 = -3_{10}$, respectively. An overflow is detected by the XOR operation between the two leftmost caries and indicates that the computed output is incorrect [Par10], as depicted in the example.

$$\begin{array}{rrl} & 0110 & (= 6_{10}) \\ + & 1101 & (= -3_{10}) \\ & 11 & \text{(carry)} \\ \hline & 0011 & (= 3_{10}) \end{array}$$

(a) no overflow

$$\begin{array}{rrl} & 1010 & (= -6_{10}) \\ + & 1101 & (= -3_{10}) \\ & 10 & \text{(carry)} \\ \hline & 0111 & (= 7_{10}) \end{array}$$

(b) overflow

Figure 2.6.: Binary subtraction, realized as addition of two's complement encoded numbers, without and with overflow, detected by an XOR of the two leftmost carries.

2.1.2. Multiplication

Similar to addition, binary multiplication can be derived from the decimal approach. Figure 2.7a shows an example of decimal multiplication. The multiplicand $a = 3751_{10}$ is multiplied by each individual digit of the multiplier $b = 6042_{10}$. The resulting partial products are aligned to the position n of the current multipliers digits and then summed up, leading to the result of the multiplication. The same approach can be used for binary multiplication, as shown in figure 2.7b for two 4-bit numbers. The multiplicand $a = 0101_2$ is multiplied by each bit of the multiplier $b = 1101_2$. Since this is a multiplication with 0_2 or 1_2, the partial products are

$$\begin{array}{lrl}
 & 3751 & \\
\times & 6042 & \\
\hline
 & 7502 & \\
 & 15004 & \\
 & 0 & \\
+ & 22506 & \\
 & 1 & \text{(carry)} \\
\hline
 & 22663542 &
\end{array}$$

(a) decimal multiplication

$$\begin{array}{lrl}
 & 0101 & (= 5_{10}) \\
\times & 1101 & (= 13_{10}) \\
\hline
 & 0101 & \\
 & 0000 & \\
 & 0101 & \\
+ & 0101 & \\
 & 1111 & \text{(carry)} \\
\hline
 & 1000001 & (= 65_{10})
\end{array}$$

(b) binary multiplication

Figure 2.7.: Concepts for decimal and binary multiplication.

either the multiplicand or all-zero. Following the decimal approach, all partial products are aligned to their respective position n and then summed up with binary addition [Man02]. Figure 2.8 shows the process of unsigned binary multiplication for the individual bits of multiplicand and multiplier for a 4×4 bit multiplication. A general $N \times N$ multiplier produces an output p with a width of $2N$ for two N-bit inputs a and b. For digital implementation, the

$$\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
+ & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

Figure 2.8.: Partial products for unsigned multiplication.

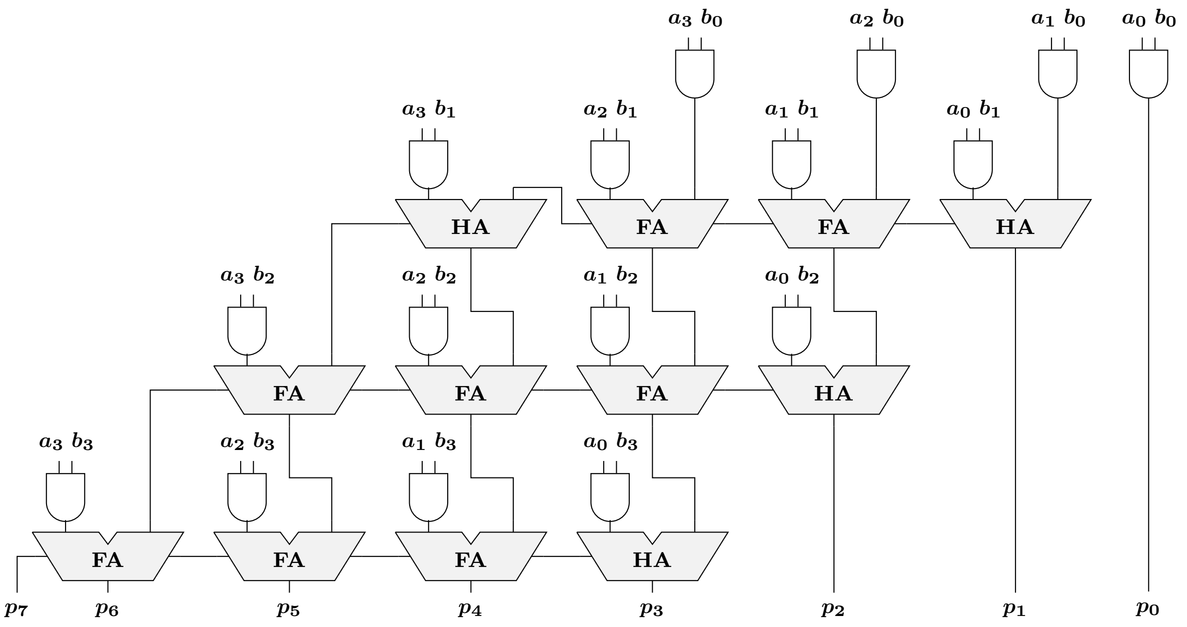


Figure 2.9.: Architecture of a 4-bit unsigned integer array multiplier.

multiplication of two single bits, required to obtain the partial products, can be realized with an AND gate. The addition of those partial products is implemented with half and full adders. Figure 2.9 shows the architecture of a 4×4 array multiplier for unsigned integer numbers [HH13]. The N^2 partial products, obtained by AND gates, are summed up with $N-1$ stages of N-bit adders, resulting in a $2N$-bit output.

Two's Complement Multiplication For signed multiplication with two's complement numbers, some adjustments have to be made in the partial product scheme and the unsigned architecture:

1. If the multiplicand a is negative, sign extension has to be carried for every partial product. This is done by copying the MSB of every partial product to the positions $N-1 < n < 2N$.
2. If the multiplier b is negative, the last partial product has to be two's complemented and sign extended. The two's complement encoding can be achieved by negating the bits of the partial product and adding a 1_2 to the carry in the lowest position.
3. Possible overflow at positions $n \geq 2N$ is discarded.

These adjustments, especially the sign extension, can be simplified using a two's complement trick which is described in [EL04]. This simplification leads to a reduced partial product scheme, displayed in figure 2.10, which realizes all of the above mentioned adjustments [EL04, DBS06]. Figure 2.11 shows the architecture of a 4-bit two's complement multiplier that implements the adapted partial product scheme. All differences to the unsigned scheme and architecture are displayed in red. The negated partial product bits are implemented with NAND gates instead of ANDs, and the adder in the top row at position $n = 4$ is now a full adder which takes the additional 1_2 as input. The 1_2 in the bottom row would require another half adder at position $n = 7$ to add up with the carry from $n = 6$. Since the carry output in position $n = 7$ is not required, this additional half adder can be simplified with a NOT gate [Pir96].
Note that the here presented array multiplier with ripple-carry adder stages is only one of various multiplier architectures that can be found in the literature [Pir96, EL04, DBS06].

$$
\begin{array}{ccccccccc}
 & & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & \times & b_3 & b_2 & b_1 & b_0 \\
\cline{5-9}
 & & & & 1 & \overline{a_3b_0} & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & \overline{a_3b_1} & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & \overline{a_3b_2} & a_2b_2 & a_1b_2 & a_0b_2 & & \\
+ & 1 & a_3b_3 & \overline{a_2b_3} & \overline{a_1b_3} & \overline{a_0b_3} & & & \\
\hline
 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Figure 2.10.: Partial products for the multiplication of two operands a and b in two's complement. Both a and b can be either positive or negative.

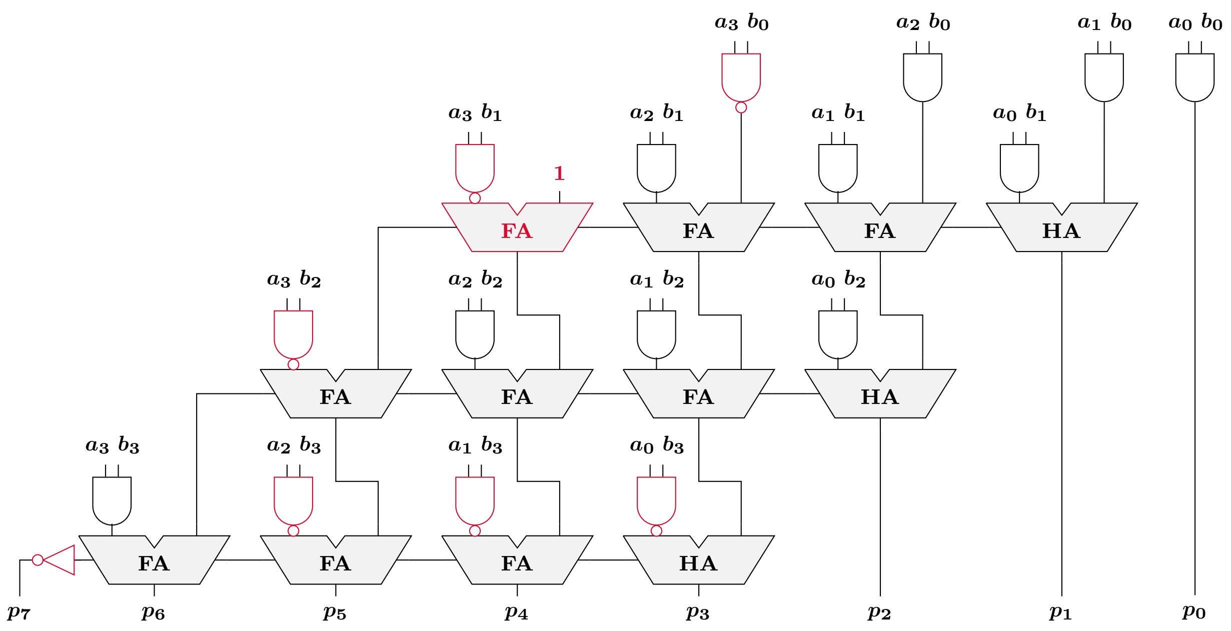


Figure 2.11.: Architecture of a 4-bit two's complement integer array multiplier.

2.1.3. Division

Comparable to the previously discussed operations addition, subtraction and multiplication, the concept of division for binary numbers can be derived from decimal division. The general approach for the division of two integers can be stated as

$$a \div b = \frac{a}{b} = q + \frac{r}{b} \tag{2.5}$$

with the dividend a divided by the divisor b. The result can be written as the combination of the integer quotient q and the remainder r, which displays the fractional part of the result, if existing. In figure 2.12a the approach of decimal division is shown with an example. The digits of the dividend $a = 7369$ are divided by the divisor $b = 5$ separately, starting with the leftmost digit $a_3 = 7$. The quotient of the first division $q_3 = 1$ is written to the result. The remainder is obtained by $a_3 - q_3 b = 2$. Then the next digit a_2 is pulled down right to the remainder.

$$\begin{array}{l}
7369 \div 5 = 1473 + \frac{4}{5} \\
\underline{-5} \\
23 \\
\underline{-20} \\
36 \\
\underline{-35} \\
19 \\
\underline{-15} \\
4
\end{array}$$

(a) decimal division

$$\begin{array}{l}
1101 \div 0010 = 0110 + \frac{0001}{0010} \\
\underline{-0} \\
11 \\
\underline{-10} \\
10 \\
\underline{-10} \\
01 \\
\underline{-\ \ 0} \\
1
\end{array}$$

(b) binary division

Figure 2.12.: Concepts for decimal and binary division.

This new number is again divided by b, resulting in the next quotient digit q_2. This process is repeated until the last digit a_0 was used. The remainder of this last iteration is the remainder r of the final result.
For binary division, a similar approach can be used. The only difference is that the quotient bits are not obtained by dividing the current remainder by b, but by a comparison. Figure 2.12b shows an example using two 4-bit unsigned integer values. Again, the bits of the dividend $a = 1101$ are processed separately, starting with the MSB a_3. Since $a_3 < b$, the first quotient bit is $q_3 = 0$. The remainder is again obtained by $a_3 - q_3 b$, then the next bit a_2 is pulled down to the right of the remainder. This new value is now greater than b, leading to $q_2 = 1$. The process is continued until the last quotient bit q_0 is obtained. A generalization of this binary division process can be formulated with the following algorithm [HH13].

Division Algorithm The division $a \div b$ for two N-bit integer numbers a and b can be described with the following iterative algorithm, requiring N iteration steps which are counted downwards by the iteration index $i = N - 1 \dots 0$. The partial remainder r' is initialized with zero: $r'(N-1) = 0_{10} = 00\dots00_2$. After initialization, the following steps are executed per iteration. Note that the round bracket notation $r'(i)$ indicates the (binary) value of r' for the current iteration i, whereas the lower case index notation q_i represents the i-th bit of the value q. The application of the division algorithm for the values used in the previous example from figure 2.12b is shown in table 2.4.

1. The remainder of the current iteration $r(i)$ is obtained by shifting $r'(i)$ left by one position and appending the bit a_i to the right as the new LSB.

$$r(i) = [r'(i) << 1, a_i] \tag{2.6}$$

2. The value of b is subtracted from the current remainder $r(i)$, using binary subtraction, and stored as the difference $d(i)$.

$$d(i) = r(i) - b \tag{2.7}$$

3. The quotient bit q_i is obtained by negating the MSB of the difference $d(i)$, which is encoded as two's complement value.

$$q_i = \overline{d}_{N-1}(i) \tag{2.8}$$

4. Finally, the partial remainder for the next iteration $r'(i-1)$ is set, depending on the current quotient bit.

$$r'(i-1) = \begin{cases} r(i) & \text{if } q_i = 0 \\ d(i) & \text{if } q_i = 1 \end{cases} \tag{2.9}$$

After the last iteration $i = 0$, the computed partial remainder $r'(-1)$ is the remainder of the overall result r.

Table 2.4.: Application of the division algorithm for 4-bit binary division $a \div b$ with $a = 1101$ and $b = 0010$.

i	$r'(i)$	$r(i)$	$d(i)$	q_i
3	0000	0001	1111	0
2	0001	0011	0001	1
1	0001	0010	0000	1
0	0000	0001	1111	0
−1	0001			

Implementation A possible hardware architecture for a division of two 4-bit integer values is the array divider shown in figure 2.13. It implements the algorithm described in equations (2.6)-(2.9) with $N = 4$ rows, each realizing one iteration i. Every row consists of an N-bit adder, calculating the difference $d(i)$, and $N - 1$ multiplexers which compute the next $r'(i - 1)$. The final stage contains one additional multiplexer (MUX) to obtain the final r. In total, the array divider consists of N^2 full adders, $N^2 - N + 1$ multiplexers, and N NOT gates [HH13, Pir96]. Similar to multiplication, many approaches for implementing division can be used. One stage of the presented array divider can, for example, be used together with a register file to implement a sequential divider, calculating one quotient bit per clock cycle. Other architectures for implementing the division operation with quotient and remainder can be found in the literature [EL04], including approaches considering signed integers [DBS06].

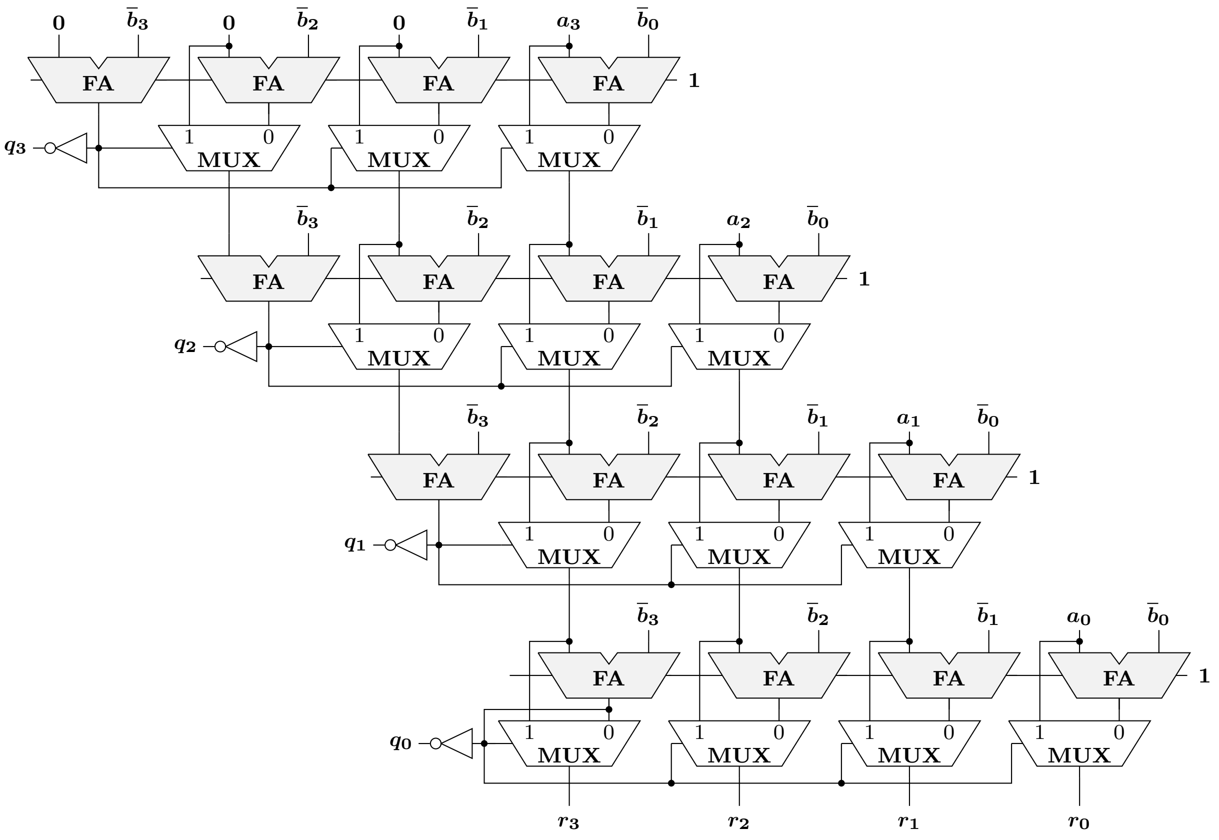


Figure 2.13.: Architecture of a 4-bit integer array divider.

2.2. Fixed Point Formats

The natural numbers $\mathbb{N}$ and integers $\mathbb{Z}$ can be considered as subsets to the rational numbers $\mathbb{Q} = \{x \,|\, x = \frac{p}{q} \text{ with } p, q \in \mathbb{Z} \text{ and } q \neq 0\}$ [BSMM08]. The rationals include all numbers with an integer and a fractional part that can be expressed as the quotient of two integers. In decimal representation, rational numbers are encoded with digits for the integer and the fractional part, separated by a separator sign, usually a dot, called decimal or radix point. The digits right to the separator are weighted with negative exponents as defined in equation (2.1) and shown in figure 2.1.

For binary encoded rationals, again the same positional approach can be used. The integer part of the number is represented with N bits, followed by the fractional part with M bits. In between is the implicit radix or binary point at a fixed position, leading to the formats name: FxD. The formula for positional notation (2.1) can be adapted for fixed point numbers [EL04]:

$$v_{\text{FxD}} = \sum_{n=-M}^{N-1} b_n 2^n \tag{2.10}$$

The integer bits $n = 0 \dots N-1$ are interpreted with positive powers of 2, the fractional bits $n = -M \dots -1$ are multiplied with negative powers of 2. The radix point is located between the bits b_0 and b_{-1}. In the following equation an example is given for an 8-bit unsigned fixed point number with $N = M = 4$:

$$1001.1101_2 = 2^3 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4} = 9.8125_{10} \tag{2.11}$$

Signed fixed point values can be encoded with the two's complement scheme in the same way as integers (see section 2.1). Table 2.5 shows the minimum and maximum values for a given unsigned and two's complement fixed point configuration.

The radix point in the binary value in equation (2.11) is shown for clarification, in a digital implementation the separator position is not encoded in a FxD value. Therefore the applied format has to be noted in the system documentation. The Q-format $Q_{N.M}$ is sometimes used as a declaration of the applied fixed point format to specify the number of integer and fractional bits, respectively [Obe07]. According to this notation, the example value from equation (2.11) is formatted as $Q_{4.4}$ and could also be written as $1001.1101_{Q4.4}$ or $10011101_{Q4.4}$, respectively. In addition to the position of the radix point, it has to be declared or documented whether the applied Q-format is used as unsigned or two's complement encoding and if the respective sign bit is included in the number of integer bits N [PU20].

Table 2.5.: Minimum and maximum values for unsigned and signed fixed point formats with N integer and M fraction bits.

encoding	**min value**		**max value**	
	decimal	binary	decimal	binary
unsigned	0	$00\dots0$	$2^N - 2^{-M}$	$11\dots1$
two's complement	-2^{N-1}	$10\dots0$	$2^{N-1} - 2^{-M}$	$01\dots1$

2.2.1. Basic Arithmetic

Since fixed point numbers are based on the same positional approach as binary integers, the basic arithmetic operations addition, subtraction, multiplication and division can be carried out with the same approaches that were discussed for integer values in sections 2.1.1-2.1.3. However, because some bits of a fixed point value are interpreted with negative powers instead of positive ones, the operations have to keep track of the radix point. For addition and subtraction, the only condition is that the fractional parts of the inputs are aligned, leading to the same fraction width in the result. For multiplication, the number of fractional bits in the result is the sum of the fraction widths of both inputs. The same holds for the integer part. For division, the number of fraction bits in the result is the dividend fraction width subtracted by the divisor fraction width. The number of integer bits in the result is the width of the dividends integer part subtracted by the fraction width of the divisor [PU20].
Figure 2.14 shows an exemplary subtraction of two 8-bit numbers, interpreted as integers and fixed point values in $Q_{4.4}$ format. Concerning hardware implementation, integer and FxD numbers can share the same architectures for arithmetic operations. The arithmetic logic unit (ALU) within a processor, for example, can be used for both integer and fixed point operations. The radix point location for fixed point operations can be tracked in software.

$$\begin{array}{rr} & 00101101 \\ + & 11001011 \\ & 1111 \\ \hline & 11111000 \end{array}$$

(a) binary

$$\begin{array}{rr} & 45 \\ + & -53 \\ \hline & -8 \end{array}$$

(b) integer

$$\begin{array}{rr} & 2.8125 \\ + & -3.3125 \\ \hline & -0.5000 \end{array}$$

(c) $Q_{4.4}$

Figure 2.14.: Subtraction of two 8-bit binary numbers using two's complement encoding, interpreted as integers and fixed point values in $Q_{4.4}$ format.

2.2.2. Iterative Approaches

The algorithms for the basic arithmetic operations discussed so far have a deterministic computing time and they produce guaranteed accurate results, except for rounding errors and precision limitations. Another group of algorithms that can be used to compute certain arithmetic operations and functions are approaches which approximate the result within multiple iterations, starting from an initial value. One of these algorithms is the Newton-Raphson method, which can be used to compute the reciprocal of a number, the square root or the reciprocal square root [EL04]. Other prominent examples are the Goldschmidt division algorithm and the Coordinate Rotation Digital Computer (CORDIC) algorithm to compute trigonometric functions [DBS06].

Newton-Raphson Algorithm The Newton-Raphson method is a general approach to determine the root of a function $f(x)$ by using the first derivative $f'(x)$. With the iterative algorithm

$$x(i+1) = x(i) - \frac{f(x(i))}{f'(x(i))} \tag{2.12}$$

and a suitable start value $x(0)$, as well as a sufficient number of iterations i, the root can be approximated [EL04]. In order to use this approach to calculate the reciprocal of a value b, for example to realize a division as a multiplication of the dividend with the reciprocal of the divisor b, the corresponding function can be defined as $f(x) = \frac{1}{x} - b$ with $f'(x) = -\frac{1}{x^2}$. This leads to the iterative algorithm

$$x(i+1) = x(i)\,(2 - bx(i)) \tag{2.13}$$

which approximates $x(i+1) = \frac{1}{b}$ for a sufficient number of iterations [AOS09]. Table 2.6 shows the application of the Newton-Raphson method to calculate the reciprocal of $b = 7$ using a $Q_{4.8}$ fixed point format with a start value $x(0) = 0.25$.
The convergence of the Newton-Raphson method is quadratic. Mathematically spoken, $\epsilon(i+1) = \epsilon(i)^2$ with the relative error of the current iteration $\epsilon(i) = 1 - bx(i)$ for the reciprocal method. In order to guarantee convergence, this leads to the following condition for the start value $x(0)$ [EL04]:

$$|\epsilon(0)| = |1 - bx(0)| < 1 \tag{2.14}$$

In general, the Newton-Raphson method can be used to find the root of any function $f(x)$. In addition to the reciprocal, another prominent use case in the context of implementing arithmetic functions is the square root (sqrt) function, or reciprocal square root, respectively. For the square root $\sqrt{s} = s^{1/2}$, the function $f(x)$ is set to $f(x) = x^2 - s$ with $f'(x) = 2x$, leading to the update equation

$$x(i+1) = \frac{1}{2}\left(x(i) + \frac{s}{x(i)}\right) \tag{2.15}$$

which approximates $x(i+1) = \sqrt{s}$ by using a division operation in every iteration. In order to avoid the division, the reciprocal square root $\frac{1}{\sqrt{s}}$ can be calculated with $f(x) = \frac{1}{x^2} - s$ and $f'(x) = -\frac{2}{x^3}$, leading to

$$x(i+1) = \frac{x(i)}{2}\left(3 - sx(i)^2\right). \tag{2.16}$$

Table 2.6.: Application of the Newton-Raphson algorithm to calculate the reciprocal of $b = 7$ $\frac{1}{b} = \frac{1}{7} = 0.142857\ldots$ using the $Q_{4.8}$ format.

$\boldsymbol{i}$	0	1	2	3	4	5
$\boldsymbol{x(i)}$	0.25000000	0.06250000	0.09765625	0.12890625	0.14062500	0.14453125

The result $x(i+1) = \frac{1}{\sqrt{s}}$ can be multiplied with s to obtain $\sqrt{s}$. According to the convergence criteria for the reciprocal method described above, the convergence of the Newton-Raphson method for square root and reciprocal square root can be achieved as follows:

$$\left|1 - \frac{x(0)^2}{s}\right| < 1 \tag{2.17}$$

$$\left|1 - sx(0)^2\right| < 1 \tag{2.18}$$

Possible architectures to implement the Newton-Raphson method for division or (reciprocal) square root can be found in [EL04] and [AOS09].

Goldschmidt Algorithm The Goldschmidt algorithm is an alternative approach to Newton-Raphson for calculating the result of a division $q = \frac{a}{b}$. Here both the dividend a and the divisor b are iteratively multiplied with a factor $\gamma(i)$ reading as

$$q = \frac{a \prod_{i=0}^{N} \gamma(i)}{b \prod_{i=0}^{N} \gamma(i)} \tag{2.19}$$

until $b \prod_{i=0}^{N} \gamma(i) \approx 1$ and $a \prod_{i=0}^{N} \gamma(i) \approx q$. The update equations of the iterative algorithm can be written as

$$\begin{aligned} \gamma(i+1) &= 2 - b(i) \\ a(i+1) &= a(i)\gamma(i+1) = a(i)(2 - b(i)) \\ b(i+1) &= b(i)\gamma(i+1) = b(i)(2 - b(i)) \end{aligned} \tag{2.20}$$

with the initial values $a(0) = a$ and $b(0) = b$. To ensure convergence, the inputs have to be normalized in order to fulfill $0 < b < 1$ [DBS06]. Table 2.7 shows the application of the Goldschmidt division $\frac{a}{b} = \frac{0.3}{0.7}$ using a $Q_{4.8}$ fixed point format. Note that $a = 0.3$ and $b = 0.7$ can not be represented exactly with the $Q_{4.8}$ format. The $Q_{4.8}$ equivalents are given as start values for $i = 0$ in Table 2.7.
Similar to Newton-Raphson, the Goldschmidt algorithm can also be modified in order to calculate the (reciprocal) square root. Details, along with possible hardware architectures for both division and square root can be found in [EL04] and [AOS09].

CORDIC Algorithm Different from Newton-Raphson or Goldschmidt, which require addition/subtraction and multiplication operations, the CORDIC algorithm is based on shift and addition/subtraction operations only. Originally developed to compute trigonometric functions,

Table 2.7.: Application of the Goldschmidt algorithm to calculate the division $\frac{a}{b} = \frac{0.3}{0.7} = 0.428571\ldots$ using the $Q_{4.8}$ format.

$\boldsymbol{i}$	0	1	2	3
$\boldsymbol{a(i)}$	0.30078125	0.39062500	0.42578125	0.42968750
$\boldsymbol{b(i)}$	0.69921875	0.91015625	0.99218750	1.00000000

the algorithm can be configured in six different operation modes to also compute multiplication, division and hyperbolic functions [Beu08]. The unified CORDIC algorithm can be written as

$$\begin{aligned} x(i+1) &= x(i) - m\,\sigma(i)\,y(i)\,2^{-\delta(i)} \\ y(i+1) &= y(i) + \sigma(i)\,x(i)\,2^{-\delta(i)} \\ z(i+1) &= z(i) - \sigma(i)\,\theta(i) \end{aligned} \tag{2.21}$$

with the hyperbolic, linear and cyclic operation modes $m \in \{-1, 0, 1\}$ and the rotation and vectoring option controlled by the parameter $t \in \{0, 1\}$. The iteration series $\delta(i)$, the sign $\sigma(i)$ and the angle $\theta(i)$ can be written as follows [Bär18]:

$$\delta(i) = \begin{cases} i & \text{with } i \in \{0, 1, ..., N-1\} & \text{if } \ m \geq 0 \\ i-k & \text{with } 3^{k+1} + 2k - 1 \leq 2i \ \text{ and } \ i \in \{1, 2, ..., N\} & \text{if } \ m = -1 \end{cases} \tag{2.22}$$

$$\sigma(i) = \begin{cases} \operatorname{sgn}(z(i)) & \text{if } \ t = 0 \\ -\operatorname{sgn}(y(i)) & \text{if } \ t = 1 \end{cases} \tag{2.23}$$

$$\theta(i) = \begin{cases} 2^{-i} & \text{if } \ m = 0 \\ \arctan\left(2^{-\delta(i)}\right) & \text{if } \ m = 1 \\ \operatorname{artanh}\left(2^{-\delta(i)}\right) & \text{if } \ m = -1 \end{cases} \tag{2.24}$$

The final result of a CORDIC computation has to be multiplied with a scale factor

$$K = \begin{cases} 1 & \text{if } \ m = 0 \\ 0.6072529 & \text{if } \ m = 1 \\ 1.2074971 & \text{if } \ m = -1 \end{cases} \tag{2.25}$$

which depends on the operation mode. Additionally, CORDIC computations are subject to certain convergence criteria, which are summarized in table 2.8.
A detailed derivation of the unified CORDIC equations, the scale factors and convergence criteria, as well as possible hardware implementations can be found in the original work [Wal71], summarized in German in [Beu08] or English in [Bär18], respectively. Table 2.9 shows the

Table 2.8.: Convergence criteria of the CORDIC algorithm for the different operation modes.

	vectoring $(t = 1)$	**rotation** $(t = 0)$
linear $(m = 0)$	$\frac{y(0)}{x(0)} \leq 2$	$z(0) \leq 2$
cyclic $(m = 1)$	$\arctan\left(\frac{y(0)}{x(0)}\right) \leq 1.743$	$z(0) \leq 1.743$
hyperbolic $(m = -1)$	$\operatorname{arctanh}\left(\frac{y(1)}{x(1)}\right) \leq 1.118$	$z(1) \leq 1.118$

different functions realized with the CORDIC algorithm in its different operations modes, controlled by the parameters m and t. Note that the hyperbolic mode $m = -1$ starts with an iteration index $i = 1$ and the start values are $x(1)$, $y(1)$ and $z(1)$, respectively. Table 2.10 shows the application of the CORDIC algorithm in cyclic rotation mode to calculate the sin() and cos() functions using a $Q_{4.8}$ fixed point format. The results computed by the CORDIC in cyclic mode are $\frac{1}{K}\sin()$ and $\frac{1}{K}\cos()$, respectively. The final results have to be multiplied with the scale factor K from equation (2.25).

Table 2.9.: Computable functions with the CORDIC algorithm for the different operation modes [Bär18].

	vectoring ($t = 1$)	**rotation** ($t = 0$)
linear	$x(N) = x(0)$	$x(N) = x(0)$
($m = 0$)	$z(N) = z(0) + \frac{y(0)}{x(0)}$	$y(N) = y(0) + x(0)\,z(0)$
cyclic	$x(N) = \sqrt{x(0)^2 + y(0)^2}$	$x(N) = x(0)\cos(z(0)) - y(0)\sin(z(0))$
($m = 1$)	$z(N) = z(0) + \arctan\left(\frac{y(0)}{x(0)}\right)$	$y(N) = y(0)\cos(z(0)) + x(0)\sin(z(0))$
hyperbolic	$x(N) = \sqrt{x(1)^2 - y(1)^2}$	$x(N) = x(1)\cosh(z(1)) + y(1)\sinh(z(1))$
($m = -1$)	$z(N) = z(1) + \operatorname{artanh}\left(\frac{y(1)}{x(1)}\right)$	$y(N) = y(1)\cosh(z(1)) + x(1)\sinh(z(1))$

Table 2.10.: Application of the CORDIC algorithm in cyclic rotation mode to calculate $x(N) = \frac{1}{K}\cos(0.5) = 1.4451\ldots$ and $y(N) = \frac{1}{K}\sin(0.5) = 0.7894\ldots$ using the $Q_{4.8}$ format.

$\boldsymbol{i}$	0	1	2	3	4
$\boldsymbol{x(i)}$	1.00000000	1.00000000	1.50000000	1.37500000	1.48437500
$\boldsymbol{y(i)}$	0.00000000	1.00000000	0.50000000	0.87500000	0.70312500
$\boldsymbol{z(i)}$	0.50000000	−0.28515625	0.17968750	−0.06640625	0.05859375
$\boldsymbol{i}$	5	6	7	8	9
$\boldsymbol{x(i)}$	1.44140625	1.46484375	1.45312500	1.44531250	1.44140625
$\boldsymbol{y(i)}$	0.79687500	0.75000000	0.77343750	0.78515625	0.78906250
$\boldsymbol{z(i)}$	−0.00390625	0.02734375	0.01171875	0.00390625	0.00000000

2.3. Floating Point Formats

The real numbers $\mathbb{R}$ are a superset to the so far discussed natural numbers $\mathbb{N}$, integers $\mathbb{Z}$ and rationals $\mathbb{Q}$ [BSMM08], represented with integer and fixed point formats, respectively. The floating point (FP) format is often referred to as the digital representation of real numbers, even though one of the properties of the reals is to be an infinite set [BSMM08], which can never be fulfilled with the implementation of finite precision. Floating point numbers can therefore be considered as a finite subset of the real numbers $\mathbb{R}$ [EL04].
As the name indicates, and in contrast to fixed point formats, floating point numbers are not limited to a fixed position of the radix point. Instead of a positional encoding, numbers are encoded in scientific notation with a significant or mantissa m, a radix or base B, and an exponent e [MBdD$^+$18, HH13]:

$$\pm\, m \times B^e \tag{2.26}$$

In a digital system, the implementation of floating point numbers can be realized in many ways, i.e. with different bitwidths for exponent and mantissa, different exception cases, etc. In fact, the first computers with floating point arithmetic showed various versions of the format [MBdD$^+$18, Kah81]. In order to standardize a floating point format which leads to comparable results on different computers, in 1985 the Institute of Electrical and Electronics Engineers (IEEE) published the "*IEEE* 754 *Standard for Binary Floating Point Arithmetic*" [IEEE85]. The encoding defined in this standard with its two revisions from 2008 [IEEE08] and 2019 [IEEE19] is still the most commonly used state-of-the-art encoding for floating point numbers and will be discussed in detail in the following section.

2.3.1. IEEE 754 Floats

The IEEE 754 Standard specifies formats for decimal and binary encoding of floating point data in order to guarantee identical results independent from the implementation, and whether it is done in software or hardware. On top of the encoding, also conversion and arithmetic operations are specified, as well as rounding behavior and exception case handling. In the following, the binary floating point encoding from the recent 2019 revision of IEEE 754 Standard is discussed [IEEE19]. Additional explanations are taken from [MBdD$^+$18], [HH13] and [Bär18].
According to IEEE 754, a binary floating point value is encoded with three different fields, namely the sign bit S, the w bit biased exponent E, and the t bit mantissa, here called trailing significant T. The encoding is shown in Figure 2.15, both E and T are interpreted as unsigned

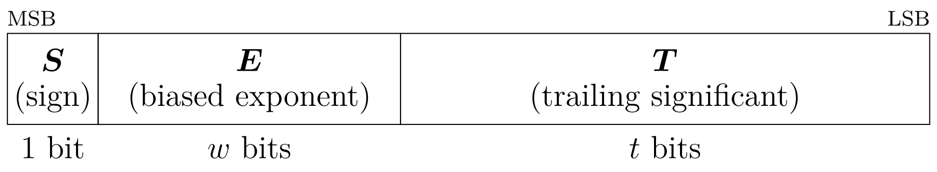


Figure 2.15.: Encoding of binary floating point value according to IEEE 754 [IEEE19].

integers. The bits from the biased exponent field E are assumed to represent the true exponent of the floating point number e from Equation (2.26) added to a bias, reading as

$$E = e + \text{bias} \tag{2.27}$$

with the bias representing the maximum exponent value e_{max} for a given exponent size w

$$\text{bias} = e_{max} = 2^{w-1} - 1\ . \tag{2.28}$$

The true exponent e is calculated as

$$e = \begin{cases} E - \text{bias} & \text{if } \ E > 0_{10} \\ 1 - \text{bias} & \text{otherwise}\ . \end{cases} \tag{2.29}$$

If $E > 0_{10}$ the encoded value is considered a *normal* float, and a *subnormal* float for $E = 0_{10}$. This affects the representation of the mantissa m from Equation (2.26), encoded by the trailing significant T. For both cases the value of T is normalized by 2^t to represent a number $< 1_{10}$. If $E > 0_{10}$ (normal float), a 1_{10} is added to shift the mantissa value to $1 \leq m < 2$. This extra 1_{10} is sometimes referred to as the hidden bit h, which has the binary value 1_2 for the normal case and 0_2 for subnormals. With this definition, the trailing significant or mantissa can be interpreted as follows:

$$m = h + \frac{T}{2^t} = \begin{cases} 1 + \frac{T}{2^t} & \text{if } \ E > 0_{10} \ \ \text{(normal)} \\ 0 + \frac{T}{2^t} & \text{if } \ E = 0_{10} \ \ \text{(subnormal)} \end{cases} \tag{2.30}$$

Special Values The IEEE 745 Standard describes the encoding of three kinds of special values, namely zero, infinity and Not a Number (NaN). The first two can occur as both positive and negative, NaN is distinguished between *quiet* and *signaling* NaN. A zero is encoded with $E = 0_{10}$ and $T = 0_{10}$. For infinity $E = 2^w - 1$, meaning that all bits of the biased exponent are set to 1_2, and $T = 0_{10}$. A NaN is represented by $E = 2^w - 1$ and $T \neq 0_{10}$. If the first (leftmost) bit of T is a 1_2, the NaN is quiet (qNaN), otherwise it is signaling (sNaN).

With these definitions, the general case description to obtain the value v_{FP} of a binary encoded floating point number according to IEEE 754 can be written as follows:

$$v_{FP} = (-1)^S \times \begin{cases} 0 & \text{if } E = 0_{10} \quad\quad \text{and } T = 0_{10} \\ 2^{2-2^{w-1}} \times (0 + \frac{T}{2^t}) & \text{if } E = 0_{10} \quad\quad \text{and } T > 0_{10} \\ 2^{E+1-2^{w-1}} \times (1 + \frac{T}{2^t}) & \text{if } 1 \leq E \leq 2^w - 2 \\ \infty & \text{if } E = 2^w - 1 \text{ and } T = 0_{10} \\ \text{qNaN} & \text{if } E = 2^w - 1 \text{ and } T \neq 0_{10} \text{ with } T_{\text{MSB}} = 1_2 \\ \text{sNaN} & \text{if } E = 2^w - 1 \text{ and } T \neq 0_{10} \text{ with } T_{\text{MSB}} = 0_2 \end{cases} \tag{2.31}$$

Precision The precision of a floating point number $p = 1 + t$ is defined by the number of bits in the significant, including the hidden bit. In order to provide different precision for

Table 2.11.: Parameter of the binary floating point encoding according to IEEE 754 [IEEE19].

name	**binary16** (half)	**binary32** (single)	**binary64** (double)	**binary128** (quad)
total bits	16	32	64	128
exponent bits w	5	8	11	15
significant bits t	10	23	52	112
precision p	11	24	52	113
bias	15	127	1023	16383

different applications, the IEEE 754 specifies four different binary floating point formats with a total bitwidth of 16, 32, 64 and 128, respectively. The original version from 1985 includes the 32 bit format as single and the 64 bit format as double precision. In the later version also 16 and 128 bit formats were specified, sometimes called half and quad or quadruple precision. Table 2.11 lists the four different formats with the respective exponent and significant bitwidths, precision and the bias values.

Rounding When the value to be encoded or the result of an arithmetic operation requires more precision than available in the current format, the value is to be rounded and the *inexact* exception has to be signaled. The standard specifies one default and three user-selectable directed rounding attributes mandatory for any IEEE 754 compliant implementation:

- roundTiesToEven (default): "[...] the floating-point number nearest to the infinitely precise result shall be delivered; if the two nearest floating-point numbers bracketing an unrepresentable infinitely precise result are equally near, the one with an even least significant digit shall be delivered; if that is not possible, the one larger in magnitude shall be delivered [...]" [IEEE19].
- roundTowardPositive: The next greater floating point value closest to the infinitely precise result shall be delivered. $+\infty$ is a possible result of this operation.
- roundTowardNegative: The next lower floating point value closest to the infinitely precise result shall be delivered. $-\infty$ is a possible result of this operation.
- roundTowardZero: The floating point value closest to the infinitely precise result and with no greater magnitude shall be delivered.

Over- & Underflow When the magnitude of a result with unbounded exponent range is greater than the largest number within the destination floating point format and is rounded according to the current rounding mode, an overflow has to be signaled. This means that not only returned $\pm\infty$ values can produce overflow. If, for example, the unbounded exponent range result is larger than the formats largest value but is rounded to this value via roundTowardNegative or roundTowardZero, this is also considered as an (positive) overflow [IEEE19, MBdD$^+$18]. The same holds for negative overflows.
For underflow, a similar definition is given [IEEE19]: An exception is signaled when a non-zero

result with unbounded exponent range lies strictly between $\pm 2^{e_{min}}$ with $e_{min} = 1 - e_{max}$ and is rounded according to the applied rounding mode. The standard also contains a second, slightly different definition which leads to different results only in a very few cases. This is discussed in [MBdD+18].

2.3.2. Floating Point Arithmetic

The IEEE 754 Standard specifies various operations to manipulate floating point numbers and defines which operations are mandatory for an IEEE 754 compliant implementations of floats. Some of these operations are listed in the following (non-exhaustive list):

- Conversion operations to and from integer and between different floating point formats.
- Comparison operations less than, equal, greater than and unordered.
- Mandatory arithmetic operations addition, subtraction, multiplication, fused multiply-add (FMA), division and square root, all with correct rounding.
- Recommended arithmetic operations/functions like $\exp(x)$, $\log(x)$, hypot(x), x^y, $\sin(x)$, $\cos(x)$, $\tan(x)$, ...

The Standard only names the operations and specifies the required behavior, not the actual implementation. Possible algorithms, as well as software and hardware implementations for the different arithmetic operations can be found in textbooks like [EL04], [DBS06] or [MBdD+18]. Note that the iterative algorithms from section 2.2.2 like Newton-Raphson or Goldschmidt can also be used with floats [MBdD+18]. In the following, the algorithms and possible implementations for floating point addition/subtraction and multiplication are discussed further.

2.3.3. Addition/Subtraction

Since floating point number representation is based on scientific notation, the methods for performing arithmetic operations with floats can be derived accordingly. Let $x = 3.125 \times 10^2$ and $y = 1.850 \times 10^4$ be two numbers in scientific/decimal floating point representation. In order to perform the addition $z = x + y$, the exponents of both numbers have to be aligned, before the mantissas can be added. In the given example, the radix point in x has to be shifted left by two positions to align the exponent, then the addition can be performed:

$$\begin{array}{rr} & 0.03125 \times 10^4 \\ + & 1.85000 \times 10^4 \\ \hline & 1.88125 \times 10^4 \end{array} \tag{2.32}$$

A similar approach is used for binary floats. The general algorithm for addition and subtraction is discussed in the following [EL04, MBdD+18].

Algorithm Let x and y be the two operands formatted as binary floating point values (S_x, E_x, T_x) and (S_y, E_y, T_y), respectively. The result of the addition/subtraction operation

$z = x \pm y$ is formatted accordingly as (S_z, E_z, T_z). Note that during the following steps, the significants are considered including the respective hidden bit in all performed operations, including shifts.

1. **Exponent Comparison:** In a first step the exponents of both operands are compared. If $E_x < E_y$ the operands x and y are swapped to ensure $E_x \geq E_y$. The output exponent is set to be $E_z = E_x$.
2. **Significant Alignment:** The significant of the operand with smaller exponent y is aligned by shifting T_y right by the exponent difference $E_x - E_y$.
3. **Significant Addition/Subtraction:** The significants T_x and T_y are added/subtracted, according to their respective signs S_x and S_y. This signed addition determines the significant T_z and sign S_z of the result.
4. **Normalization:** During the signed addition in the previous step it can happen that either (a) a carry out is produced in the MSB or (b) the result contains leading zeros due to subtraction. In both cases the result has to be adjusted by either (a) shifting T_z right by one position and incrementing E_z by one, or (b) shifting T_z left by the number of leading zeros and decrementing E_z accordingly.
5. **Rounding:** Due to the possible shifts in the alignment and normalization steps the result significant T_z might be subject to rounding according to the specified rounding mode. In order to guarantee a correct rounding, the intermediate resulting significant T_z is computed with three extra bits right to the LSB, two guard bits, sometimes separated into guard and round bit, and a sticky bit.
   - The (first) guard bit is used to store the LSB shifted out during the normalization step when a carry out is produced during addition.
   - The round bit is required if a subtraction is performed and the exponents of both operands differ by more than 1_{10}. In this case a left shift by one position can be necessary to normalize the result. Then the guard bit becomes the LSB and the round bit becomes the guard bit.
   - The sticky bit is used whenever bits are shifted out to the right, mostly during the alignment step. It contains the result of an OR operation of all shifted out bits.

   These three bits are used to apply the selected rounding mode (see section 2.3.1). More details can be found in [EL04], [MBdD$^+$18], [DBS06] and [Gol91].
6. **Exceptions:** During the operation, five different exceptions might be detected:
   - An overflow occurs in the normalization step (a) if the exponent to be incremented E_z is already at its maximum value.
   - During case (b) an underflow can occur when E_z is too small to be decremented by the number of leading zeros.
   - When the result of the significant addition is zero, the resulting exponent has to be set to $E_z = 0$.
   - The result is inexact if rounding is applied.
   - If one or both operands are NaN, the output is also set to NaN.

Implementation A hardware implementation of a floating point adder realizes the steps of the algorithm discussed above. It consists of five main blocks:

- The exponent comparison which computes the difference of the two operand exponents with a subtractor.
- The significant alignment that utilizes a shifter according to the exponent difference.
- The significant addition/subtraction whose main component is a signed adder.
- The normalization which is composed of a leading zero or leading one detector and a shifter.
- The rounding block with an adder and combinatorial logic for evaluating the guard bits.

On top of these blocks, the exception cases have to be detected and processed. Different possible implementation schemes for floating point adders can be found in figures 2.16 and 2.17, [EL04], [MBdD$^+$18] and [DBS06], respectively. The main conceptional difference is between the so-called single-path and dual-path architecture. The single-path approach, depicted in figure 2.16, performs the above discussed algorithm in a straight forward way including the two shift operations for alignment and normalization in one path. However, it can never happen that large shifts in both directions are required within one operation: For exponent differences ≤ 1 the required right shift during the alignment is maximum one position, while

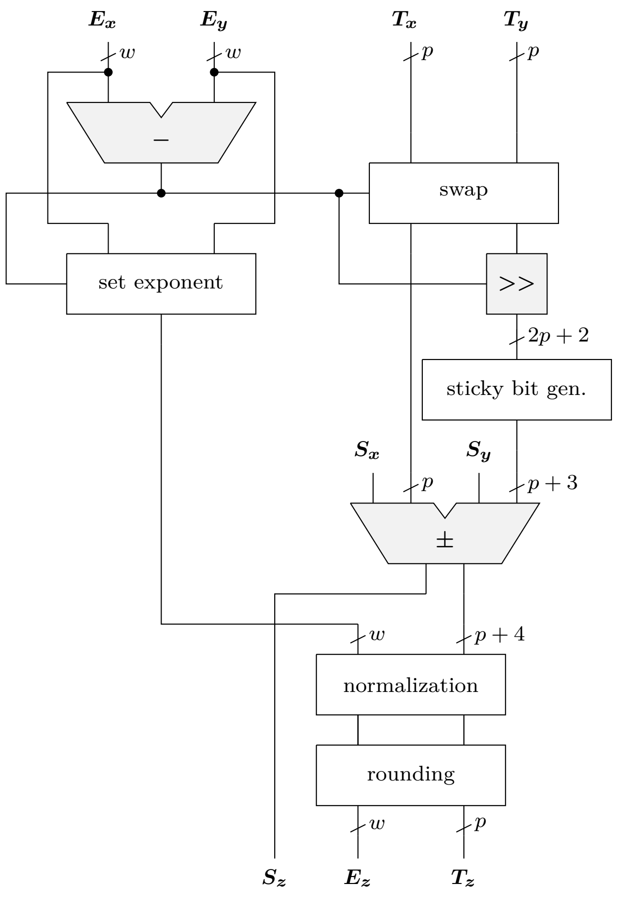


Figure 2.16.: Architecture of a single-path floating point adder/subtractor without exception case handling.

the left shift during the normalization can be larger. This is called the *close* path. For the *far* path with exponent differences > 1, on the other hand, the alignment right shift can be large, but the normalization left shift is again maximum one position. By separating these two paths with the dual-path design, shown in figure 2.17, the critical path delay can be improved [EL04, MBdD+18].

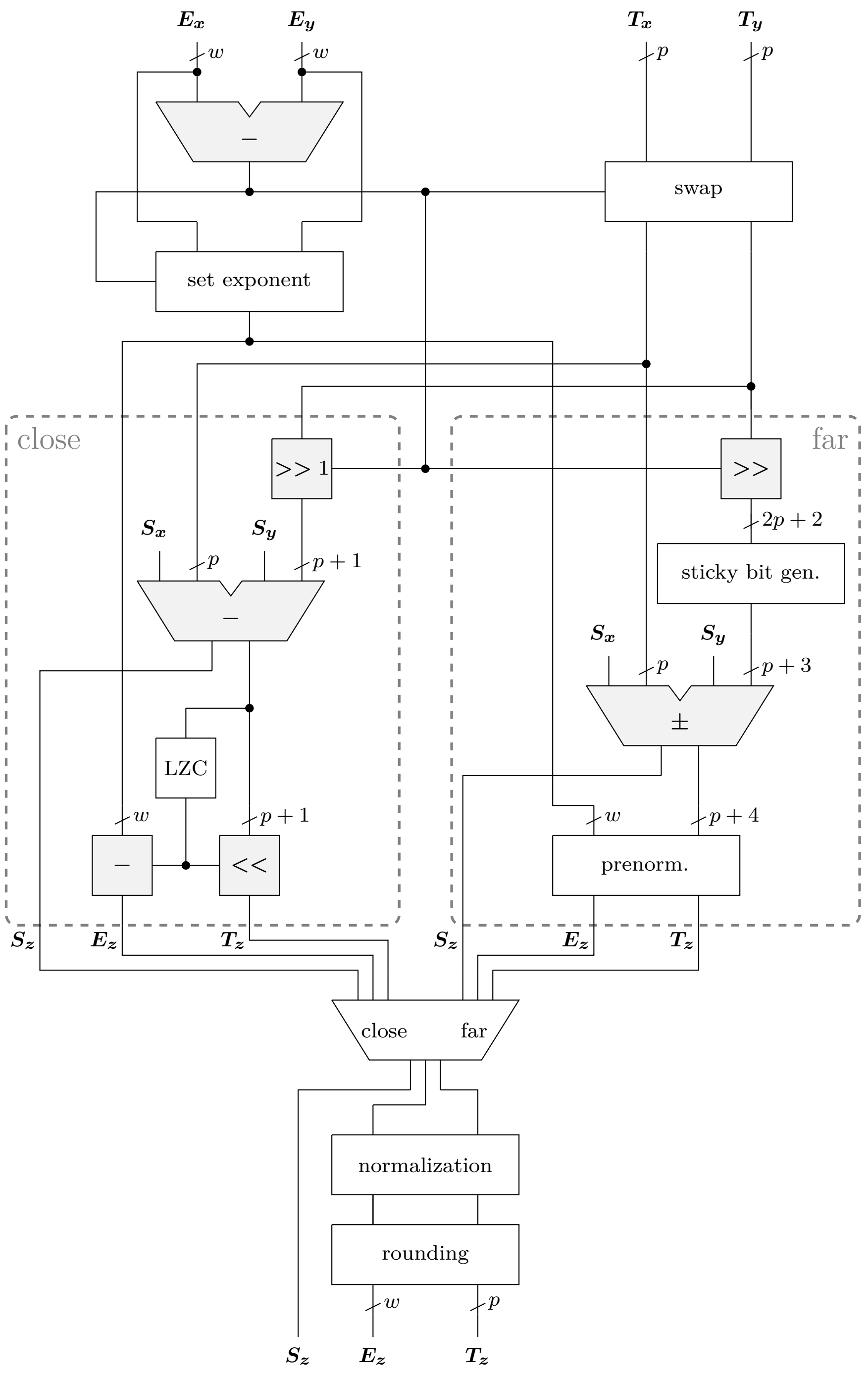


Figure 2.17.: Architecture of a dual-path floating point adder/subtractor without exception case handling [MBdD+18].

2.3.4. Multiplication

Multiplication of two floating point numbers is less complicated than addition. Considering again two numbers $x = 3.125 \times 10^2$ and $y = 1.850 \times 10^4$ in scientific notation, their product $z = x \times y$ can be computed by simply multiplying the mantissas and adding the exponents:

$$\left(3.125 \times 10^2\right) \times \left(1.850 \times 10^4\right) = (3.125 \times 1.850) \times 10^{2+4} = 5.78125 \times 10^6 \tag{2.33}$$

For binary floats this approach is followed straight forward. In the following the general algorithm for multiplication is discussed [EL04, MBdD+18].

Algorithm Let x and y be the two operands formatted as binary floating point values (S_x, E_x, T_x) and (S_y, E_y, T_y), respectively. The result of the multiplication operation $z = x \times y$ is formatted accordingly as (S_z, E_z, T_z). Note that during the following steps, the significants are considered including the respective hidden bit in all performed operations.

1. **Significant Multiplication:** The two significants T_x and T_y are multiplied, resulting in a product with doubled bitwidth $2p$. The lower half of the result is reduced to a guard bit and a sticky bit combining all discarded bits with an OR operation.

2. **Exponent Addition:** The two biased operand exponents E_x and E_y are added and the bias (equation (2.28)) is subtracted to obtain the resulting exponent: $E_z = E_x + E_y - \text{bias}$.

3. **Sign:** The output sign is obtained by an XOR operation of the operand signs: $S_z = S_x \,\text{XOR}\, S_y$.

4. **Normalization:** The multiplication of the significants might produce a result $2 \leq T_z < 4$ because the operand significants are between 1_{10} and 2_{10}. In this case the output has to be normalized by shifting the significant T_z right by one position and incrementing the exponent E_z by 1_{10}. The LSB of T_z becomes the guard bit and the sticky bit needs to be recomputed by an OR of the previous sticky and guard bit.

5. **Rounding:** Similar to addition/subtraction, rounding is performed according to the specified rounding method (section 2.3.1) based on the to guard and sticky bit.

6. **Exceptions:** During the operation, five different exceptions might be detected:
   - An overflow may occur if the resulting exponent is too large to represent after the exponent addition.
   - An underflow may occur if the resulting exponent is too small to represent after the exponent addition, due to the included bias subtraction.
   - The output is set to zero if at least one of the operands is zero.
   - The result is inexact if rounding is applied.
   - The result is NaN if one or both operands are NaN, or if the operands are zero and infinity.

A special case that is not considered in the presented algorithm is the multiplication of subnormal floats with $E = 0$. If one or both operands are subnormal they either have to be normalized before multiplication, which might require an internal extension of the exponent size, or the result of the significant multiplication has to be normalized with left shifts. This is discussed in detail in [MBdD$^+$18].

Implementation A hardware implementation of a floating point multiplier realizes the steps of the algorithm discussed above. It consists of five main blocks:

- The exponent addition which utilizes an adders and a subtractor.
- The significant multiplication which is composed of an $p \times p$ multiplier.
- The sign computation which uses a single XOR gate.
- The normalization that consists of a one bit right shifter and an incrementer.
- The rounding block with an adder and combinatorial logic for evaluating the guard and sticky bit.

On top of these blocks, the exception cases have to be detected and processed. Despite the straight forward implementation, which is discussed here and depicted in figure 2.18, various alternative approaches exists. Since the critical path delay is set by the significant multiplier, the most prominent approach is to implement a carry-save multiplier to reduce the latency of the overall design [EL04, MBdD$^+$18, DBS06].

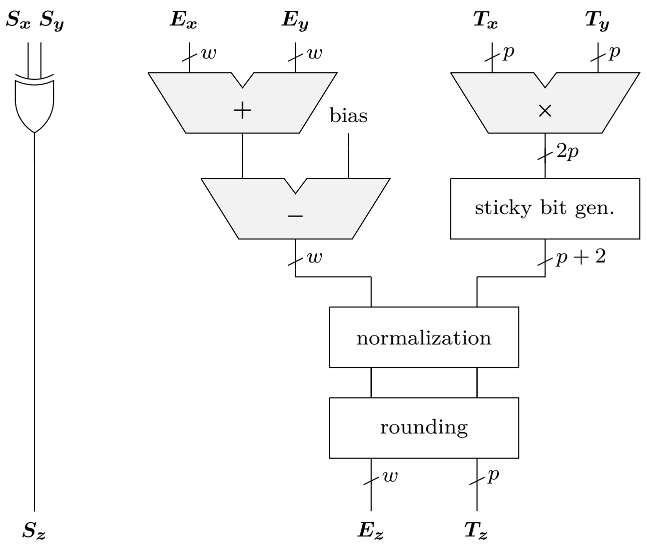


Figure 2.18.: Architecture of a floating point multiplier without subnormal and exception case handling.

2.3.5. Alternative Floating Point Formats

The IEEE 754 standard was released in 1985 in order to harmonize the various existing floating point formats and to ensure compatibility and portability between different systems. During that time, the standard served its purpose well, whereas nowadays the demands for more flexibility and application specificity increase (again). Two strong drivers of this trend are neural networks (NNs) with demands for low computational complexity and power consumption [Joh18], as well as high-performance computing (HPC) where data movement and memory bandwidth are identified as the system bottlenecks [LLH18]. Both applications require datatypes different from the IEEE standard, either in terms of bitwidth, dynamic range or efficiency of arithmetic circuits. Due to the demands of these and other applications, the ecosystem of floating point datatypes is growing again.
One prominent example is the so-called brain float *bfloat16* format from Google, a 16 bit floating point representation designed for machine learning applications [WK19]. The difference to the IEEE 16 bit half precision format is the number of exponent and fraction bits. While half precision consists of 5 exponent and 10 significant bits, bfloat16 has 8 exponent bits like the IEEE 32 bit single precision datatype, and 7 significant bits. This gives bfloats the same dynamic range as single precision numbers and simplifies conversions between both formats while sacrificing some precision.
A similar approach is followed for the *DLFloat* format developed by IBM, especially for deep learning (DL) applications [AMF$^+$19]. It can be seen as an intermediate approach between half precision and bfloat, as it also consists of 16 bit, 6 for the exponent and 9 for the significant. DLFloats further do not utilize subnormal numbers and combine the NaN and infinity exception into one bit pattern, which both increases the dynamic range and reduces the hardware complexity of the corresponding arithmetic circuits.
Another, more disruptive approach is the posit number format which belongs to the unums. This format will be discussed in detail in section 2.6.2. Posits are a floating point format consisting of a 2 bit exponent, and two variable length fields: the significant, and a second, exponent-like scaling factor called *regime* which enables a tapered accuracy and a higher dynamic range, compared to IEEE floats [GY17, Pos22]. The encoding of the discussed formats is shown in figure 2.19. Comparisons of IEEE floats, bfloats, DLFloats, posits and other intermediate FP formats, especially for machine learning applications, can be found in the literature, for example [Joh18], [LLH18], [WK19], [AMF$^+$19], [RSL$^+$21] or [DSTH$^+$23].

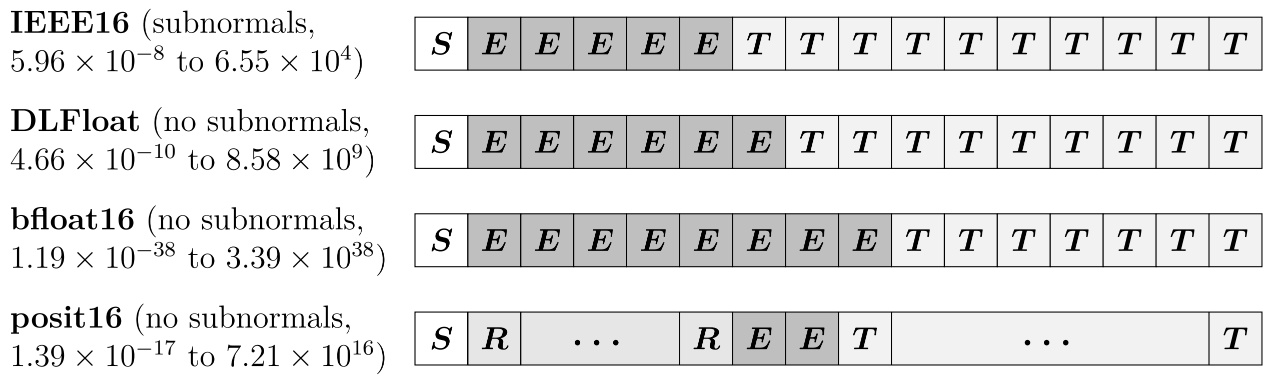

IEEE16 (subnormals, 5.96×10^{-8} to 6.55×10^{4})
S E E E E E T T T T T T T T T T
DLFloat (no subnormals, 4.66×10^{-10} to 8.58×10^{9})
S E E E E E E T T T T T T T T T
bfloat16 (no subnormals, 1.19×10^{-38} to 3.39×10^{38})
S E E E E E E E E T T T T T T T
posit16 (no subnormals, 1.39×10^{-17} to 7.21×10^{16})
S R ... R E E T ... T

Figure 2.19.: Encoding of different 16 bit floating point formats.

2.4. Application Specific Number Formats

The discussed integer, fixed and (IEEE) floating point number representations can be considered the standard or classic formats for computer number representation. Apart from these standard ones, and the already discussed floating point variations, there exists a variety of other approaches, either for complete, stand-alone number formats, or for those with intermediate purpose to be used in combination with the standard representations [Par03]. Since these formats are used for dedicated approaches in special applications and are usually not implemented in general-purpose designs, they can be summarized under the term *application specific*, sometimes also called *non-conventional* number formats. One example for such an approach are the slash number systems with fixed- or floating-slash, which, in simplified terms, utilize two separate values for nominator and denominator to represent a number in fractional form [MK85]. Other approaches, which will be discussed in more detail in the following, are the carry-save format, the Residue Number System (RNS), the Logarithmic Number System (LNS) and stochastic computing.

2.4.1. Carry-Save Format

The ripple-carry adder structure discussed in section 2.1.1, as well as the carry-lookahead and prefix adder, are summarized as carry propagate adders (CPAs) because the carry from the LSB position has to propagate from right to left to determine the final result. This carry propagation defines the delay of the respective adder design [HH13]. When more than two operands have to be added, for example the partial products within a multiplier, a carry-save adder (CSA) can be used to improve the critical path delay of the design. During this process, the intermediate results are converted into the carry-save representation which consists of an N-bit number s to store the sum bits of the result, and a second N-bit number c to store the carry bits. This carry-save representation with $2N$ bit can be interpreted as a redundant number representation, since the result of a three operand addition with N-bit operands could generally be stored with $N+2$ bit [Pir96].
When further operands need to be added, this can be done in a similar manner, using s and c as operands, and producing another result in carry-save form. Note that c has to be applied to the addition of positions 1 to N, rather than 0 to $N-1$, because the carry of a result contributes to the next higher position [Par10]. Figure 2.20 shows an N-bit CSA with three operands x, y and z consisting of N full adders, which produce an N-bit sum s and an N-bit carry output c. Because the carry bits are saved rather than propagated, the delay of such a CSA is equal to the delay of only one full adder. For multi-operand additions, multiple

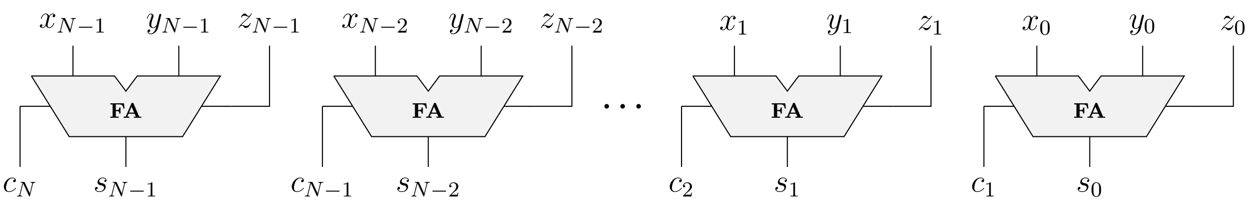


Figure 2.20.: Architecture of an N bit carry-save adder [Pir96].

CSAs with one CPA at the end, required to convert the carry-save formatted result back to the standard integer representation, can significantly reduce the overall delay. For an array multiplier, as discussed in section 2.1.2, the critical path delay can be improved by 33.3% when using a CSA instead of a ripple-carry adder for the partial products [DBS06].

2.4.2. Residue Number System

The Residue Number System (RNS) is based on a rather old approach, derived by a Chinese mathematician in the first century AD [JSR18]. The general idea is to convert a (large) integer number x into a set of N small integers $\{x_1, \dots, x_N\}$, which represent the remainders or residues for a division of x by a set of moduli $\{m_1, \dots, m_N\}$. These remainders are obtained by the modulo operation:

$$\{x_1, \dots, x_N\} = x \bmod \{m_1, \dots, m_N\} \qquad \text{with } \ 0 \leq x_i < m_i \tag{2.34}$$

The moduli $\{m_1, \dots, m_N\}$ have to be pairwise coprime, or relatively prime, meaning that the greatest common divisor of two moduli m_i and m_j is 1 [BSMM08]. The product of all moduli

$$M = \prod_{i=1}^{N} m_i \tag{2.35}$$

defines the maximum integer value that can be represented uniquely with the defined set of moduli, leading to $0 \leq x \leq M - 1$. When applying a complement encoding, also negative numbers can be represented within the range $-\frac{M}{2} \leq x \leq \frac{M}{2} - 1$ for an even M and $-\frac{M-1}{2} \leq x \leq \frac{M-1}{2}$ for an odd M [Par10, JSR18]. In the negative case, the remainders are obtained as follows:

$$x_i = \begin{cases} x \bmod m_i & \text{if } \ x \geq 0 \\ m_i - |x| \bmod m_i & \text{if } \ x < 0 \end{cases} \tag{2.36}$$

The arithmetic operations addition, subtraction and multiplication can be applied to the remainders directly. For two integer values x and y in residual form $\{x_1, \dots, x_N\}$ and $\{y_1, \dots, y_N\}$, the result $\{z_1, \dots, z_N\}$ is obtained by

$$z_i = (x_i \circ y_i) \bmod m_i \tag{2.37}$$

where $\circ$ denotes the respective operation. Other operations like division or comparison, however, are more complicated with residual numbers [SJJT86].
The (back)conversion from a number in residual form $\{x_1, \dots, x_N\}$ to its integer form x can be done using the Chinese Remainder Theorem (CRT) according to

$$x = \left(\sum_{i=1}^{N} \hat{m}_i \left(\hat{m}_i^{-1} x_i \bmod m_i \right) \right) \bmod M \tag{2.38}$$

with $\hat{m}_i = \frac{M}{m_i}$ and $\hat{m}_i^{-1}$ being the multiplicative inverse of $\hat{m}_i$, obtained by solving the following congruence relation [SJJT86]:

$$\hat{m}_i \hat{m}_i^{-1} \equiv 1 \pmod{m_i} \tag{2.39}$$

Figure 2.21 shows an example of applying the RNS to two integer numbers, performing an addition operation in residual form and converting the result back using the CRT.
In terms of hardware implementation, the RNS enables reduced storage size when numbers are stored in residual form, and speeds up the computation of additions and multiplications, because the required adder architectures become much smaller, and the operations on the different moduli can be carried out in parallel [JSR18]. Current applications for RNS are cryptography [MASC17] and deep learning [RTAF21].

System: $\{m_1, m_2, m_3\} = \{3, 5, 7\}$ $\quad M = 3 \times 5 \times 7 = 105$

Operands:

$x = 11$

$$x_1 = 11 \bmod 3 = 2$$
$$x_2 = 11 \bmod 5 = 1$$
$$x_3 = 11 \bmod 7 = 4$$

$y = 13$

$$y_1 = 13 \bmod 3 = 1$$
$$y_2 = 13 \bmod 5 = 3$$
$$y_3 = 13 \bmod 7 = 6$$

Addition:

$z = x + y$

$$z_1 = (2 + 1) \bmod 3 = 0$$
$$z_2 = (1 + 3) \bmod 5 = 4$$
$$z_3 = (4 + 6) \bmod 7 = 3$$

Conversion:

$$\hat{m}_1 = \frac{M}{3} = 35 \qquad 35\,\hat{m}_1^{-1} \equiv 1 \pmod{3} \quad \Rightarrow \hat{m}_1^{-1} = 2$$
$$\hat{m}_2 = \frac{M}{5} = 21 \qquad 21\,\hat{m}_2^{-1} \equiv 1 \pmod{5} \quad \Rightarrow \hat{m}_2^{-1} = 1$$
$$\hat{m}_3 = \frac{M}{7} = 15 \qquad 15\,\hat{m}_3^{-1} \equiv 1 \pmod{7} \quad \Rightarrow \hat{m}_3^{-1} = 1$$

$$\begin{aligned} z &= \left(35 \times \underbrace{(2 \times 0) \bmod 3}_{=0} + 21 \times \underbrace{(1 \times 4) \bmod 5}_{=4} + 15 \times \underbrace{(1 \times 3) \bmod 7}_{=3} \right) \bmod 105 \\ &= (0 + 84 + 45) \bmod 105 \\ &= 24 \end{aligned}$$

Figure 2.21.: Example for applying the RNS to two integers x and y, performing the addition in residual form and converting the result back to integer using the CRT.

2.4.3. Logarithmic Number System

Another prominent example for a number format is the Logarithmic Number System (LNS), which was developed during the 1970s as an alternative for the standard floating point format [CP13]. The approach is to encode numbers by taking their logarithms to a base B, usually with $B = 2$ for digital systems. In a logarithmic representation, a number $x = \pm 2^{L_x}$ is encoded by a sign bit and the logarithm of its absolute value

$$L_x = \log_2(|x|) \tag{2.40}$$

which is stored as a fixed point value with an integer and fractional part. Since for $0 < x < 1$ the logarithm is negative, L_x must either be encoded in two's complement, or a bias needs to be added to ensure that L_x is always positive [Par10]. Some implementations also include extra bits to encode special cases like zero, NaN or infinity [DdD03]. The benefit of representing numbers by their logarithms is that multiplication and division operations are simplified to addition and subtraction, respectively:

$$\begin{aligned} \log_2(|x| \times |y|) &= L_x + L_y \\ \log_2(|x| \div |y|) &= L_x - L_y \end{aligned} \tag{2.41}$$

The output sign is obtained with an XOR operation of the input signs [Par10]. The downside of the approach, on the other hand, is that addition and subtraction in logarithmic representation are more complex:

$$\begin{aligned} \log_2(x + y) &= L_x + \log_2\left(1 + 2^{L_y - L_x}\right) \\ \log_2(x - y) &= L_x + \log_2\left(1 - 2^{L_y - L_x}\right) \end{aligned} \tag{2.42}$$

with $|x| \geq |y|$, or $L_x > L_y$, respectively [DdD03, CP13]. The main challenge for implementing a LNS is therefore to handle the non-linear function $\log_2\left(1 \pm 2^d\right)$ with $d = L_y - L_x < 0$. Approaches for this are either the usage of lookup tables (LUTs) [SA75], or the approximation using Taylor-series [CSK⁺08]. Another method for implementing addition and subtraction in LNS architectures is to convert from logarithmic to an internal fixed point format, for example using Mitchell's Method [Mit62], or piecewise function approximation [RLP13]. The operation is then performed with a conventional adder, before the result is converted back to its logarithmic form.

LNS arithmetic thus can be beneficial for algorithms with a high amount of multiplication and/or division operations, and when addition and subtraction can be implemented without performance losses, compared to standard formats [CP13]. Applications for LNS arithmetic reach from general-purpose processors [CSK$^+$08] and digital signal processing [RLP13] to current scientific topics like neural networks [ACJ20] and quantum computing [Arn22].

2.4.4. Stochastic Computing

All the number representations discussed so far are positional systems where every bit encodes a dedicated value or weight. Most of them are non-redundant or contain only a small amount of redundancy, which means they are very efficient from an information-per-bit perspective. However, such an encoding can also have disadvantages, especially when it comes to reliability and fault-tolerance of a system. One big challenge in this domain are so-called *soft errors*, visible as temporarily bit flips, which can be caused by ionized radiation [DM03]. If a fixed point number is effected by such a single bit flip, a huge numerical error can result if the MSB or one of the higher order bits is flipped. The same holds for a floating point number when flipping the sign bit or one of the exponent bits. Common techniques to deal with this problem are the usage of error-correction codes or the implementation of redundant operation blocks [QLR+11].

An alternative concept is to use a non-positional and redundant number format to encode numerical values, where a bit flip in any position introduces only a small numerical error. This approach is called stochastic computing (SC). Instead of a positional encoding, numbers are represented as probabilities p of zeros and ones in a bit stream [AH13]. The following equation shows the encoding of a number x with a random 8-bit stochastic number (SN), where the probability of ones p_x encodes the decimal value of x:

$$x = 0.375_{10} = 0.011_{Q1.3} \mathrel{\hat{=}} 01001100|_{p_x=\frac{3}{8}} \tag{2.43}$$

While a flip of the MSB in the fixed point encoding would lead to a value $1.011_{Q1.3} = 1.375_{10}$ (unsigned) with a numerical error $\epsilon = 1$, flipping any of the bits in the stochastic encoding would introduce a numerical error of only $\epsilon = \pm\frac{1}{8}$.

Arithmetic operations in this stochastic format can be implemented with simple logic gates. A multiplication of two probabilities $p_x \times p_y$ can be realized by an AND operation of the two stochastic numbers. However, this only holds when the two numbers are uncorrelated [AH13]. Another condition for using SC is that input values might have to be scaled or normalized, because, since they encode probabilities, stochastic numbers can only represent values in $[0, 1]$ [AQH18]. Due to this limitation, the implementation of a stochastic adder, realized with a MUX, shifts the addition results to $[0, 1]$, while they would normally be in $[0, 2]$ [AH13]. Concepts for other arithmetic operations and functions with stochastic numbers, as well as conversions and the bipolar encoding for representing negative values are discussed in [AH13] and [AQH18], respectively.

SC provides a high fault tolerance and low-complex arithmetic operations, but, at the same time, a quite low precision and low latency when implementing the stochastic operations in a serial manner [QLR+11]. These advantages and limitations define possible applications, which include neural networks and image and signal processing [AQH18].

2.5. Interval Arithmetic

Traditional fixed point or floating point arithmetic and all other approaches that implement real numbers with finite precision single values are subject to rounding, which leads to potentially incorrect results. For most applications these small inaccuracies can be tolerated as they do not influence the behavior of the overall system in a critical way. For applications where rounding error propagation is a critical point, however, as well as for scientific computing, interval arithmetic (IA) is a way to guarantee correct solutions and to ensure reliable computing. The general approach is to represent a value x not by a finite precision and potentially rounded single value, but by an interval X which is composed of two values for the lower and upper bound and encompasses the actual value of x [Gom09]. Such an interval can be defined as

$$X = \{[\underline{x}, \overline{x}] \mid \underline{x}, \overline{x} \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\} \tag{2.44}$$

with the lower bound $\underline{x}$ and the upper bound $\overline{x}$ [Rum10]. This notation is compliant to the IEEE standard for interval arithmetic which specifies IA with IEEE floating point bounds [IEEE18]. The notation with square brackets $[\underline{x}, \overline{x}]$ denotes that the endpoints $\underline{x}$ and $\overline{x}$ are included in the respective interval, whereas round brackets $(\underline{x}, \overline{x})$ exclude them. Combinations of square and round brackets for representing half-open intervals are also possible [BSMM08]. An alternative representation for excluding interval bounds is $]\underline{x}, \overline{x}[$ [Gom09]. In this work round brackets will be used to denote excluded or open interval bounds.
For the four standard operations $\circ \in \{+, -, \times, \div\}$ arithmetic with two intervals X and Y with $0 \notin Y$ for division is defined as follows [Rum10]:

$$X \circ Y = [\min(\underline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \underline{y}, \overline{x} \circ \overline{y}), \max(\underline{x} \circ \underline{y}, \underline{x} \circ \overline{y}, \overline{x} \circ \underline{y}, \overline{x} \circ \overline{y})] \tag{2.45}$$

This general formula can be simplified for the operations addition and subtraction reading as

$$X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \tag{2.46}$$
$$X - Y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \tag{2.47}$$

while for multiplication, equation (2.45) can be specified as follows [KK06, NSWvG12]:

$$X \times Y = \begin{cases} [\underline{x} \times \underline{y}, \overline{x} \times \overline{y}] & \text{if } \underline{x} \geq 0 \text{ and } \underline{y} \geq 0 \\ [\overline{x} \times \underline{y}, \overline{x} \times \overline{y}] & \text{if } \underline{x} \geq 0 \text{ and } \underline{y} < 0 \leq \overline{y} \\ [\overline{x} \times \underline{y}, \underline{x} \times \overline{y}] & \text{if } \underline{x} \geq 0 \text{ and } \overline{y} < 0 \\ \\ [\underline{x} \times \overline{y}, \overline{x} \times \overline{y}] & \text{if } \underline{x} < 0 \leq \overline{x} \text{ and } \underline{y} \geq 0 \\ [\min(\underline{x} \times \overline{y}, \overline{x} \times \underline{y}), \max(\underline{x} \times \underline{y}, \overline{x} \times \overline{y})] & \text{if } \underline{x} < 0 \leq \overline{x} \text{ and } \underline{y} < 0 \leq \overline{y} \\ [\overline{x} \times \underline{y}, \underline{x} \times \underline{y}] & \text{if } \underline{x} < 0 \leq \overline{x} \text{ and } \overline{y} < 0 \\ \\ [\underline{x} \times \overline{y}, \overline{x} \times \underline{y}] & \text{if } \underline{x} < 0 \text{ and } \underline{y} \geq 0 \\ [\underline{x} \times \overline{y}, \underline{x} \times \underline{y}] & \text{if } \underline{x} < 0 \text{ and } \underline{y} < 0 \leq \overline{y} \\ [\overline{x} \times \overline{y}, \underline{x} \times \underline{y}] & \text{if } \underline{x} < 0 \text{ and } \overline{y} < 0 \end{cases} \tag{2.48}$$

Accordingly, equation (2.45) can be specified for division [KK06, NSWvG12]:

$$X \div Y = \begin{cases} [\underline{x} \div \overline{y}, \overline{x} \div \underline{y}] & \text{if } \underline{x} \geq 0 \text{ and } \underline{y} > 0 \\ [\overline{x} \div \overline{y}, \underline{x} \div \underline{y}] & \text{if } \underline{x} \geq 0 \text{ and } \overline{y} < 0 \\ \\ [\underline{x} \div \underline{y}, \overline{x} \div \underline{y}] & \text{if } \underline{x} < 0 \leq \overline{x} \text{ and } \underline{y} > 0 \\ [\overline{x} \div \overline{y}, \underline{x} \div \overline{y}] & \text{if } \underline{x} < 0 \leq \overline{x} \text{ and } \overline{y} < 0 \\ \\ [\underline{x} \div \underline{y}, \overline{x} \div \overline{y}] & \text{if } \overline{x} < 0 \text{ and } \underline{y} > 0 \\ [\overline{x} \div \underline{y}, \underline{x} \div \overline{y}] & \text{if } \overline{x} < 0 \text{ and } \overline{y} < 0 \end{cases} \quad \text{with } 0 \notin Y \tag{2.49}$$

$$X \div Y = \begin{cases} [\mathrm{NaN}, \mathrm{NaN}] & \text{if } \underline{x} > 0 \text{ and } \underline{y} = \overline{y} = 0 \\ [-\infty, \underline{x} \div \underline{y}] & \text{if } \underline{x} > 0 \text{ and } \underline{y} < \overline{y} = 0 \\ [-\infty, \underline{x} \div \underline{y}] \cup [\underline{x} \div \overline{y}, +\infty] & \text{if } \underline{x} > 0 \text{ and } \underline{y} < 0 < \overline{y} \\ [\underline{x} \div \overline{y}, +\infty] & \text{if } \underline{x} > 0 \text{ and } 0 = \underline{y} < \overline{y} \\ \\ [-\infty, +\infty] & \text{if } \underline{x} \leq 0 \leq \overline{x} \\ \\ [\mathrm{NaN}, \mathrm{NaN}] & \text{if } \overline{x} < 0 \text{ and } \underline{y} = \overline{y} = 0 \\ [\overline{x} \div \underline{y}, +\infty] & \text{if } \overline{x} < 0 \text{ and } \underline{y} < \overline{y} = 0 \\ [-\infty, \overline{x} \div \overline{y}] \cup [\overline{x} \div \underline{y}, +\infty] & \text{if } \overline{x} < 0 \text{ and } \underline{y} < 0 < \overline{y} \\ [-\infty, \overline{x} \div \overline{y}] & \text{if } \overline{x} < 0 \text{ and } 0 = \underline{y} < \overline{y} \end{cases} \quad \text{with } 0 \in Y \tag{2.50}$$

Note that in an actual implementation, for the case resulting in the union of two intervals with $\underline{y} < 0 < \overline{y}$, the result can be either represented as $[-\infty, +\infty]$ [NSWvG12], or with an improper interval where $\underline{x} > \overline{x}$ [KK06].
Other arithmetic operations, comparisons and elementary functions with intervals are discussed in [KK06], [Kul09], [Gom09], [Rum10] and [MKC09]. A commonly used class of elementary functions are monotonic functions. When considering such a monotonic increasing or decreasing function f with an interval input X, the output can be determined as follows [MKC09]:

$$f(X) = [\min(f(\underline{x}), f(\overline{x})), \max(f(\underline{x}), f(\overline{x}))] \tag{2.51}$$

Since there are many elementary functions which are not or only piece-wise monotonic, the given definitions need to be adapted for certain functions. A prominent example is the exponential or polynomial function $f(X) = X^n$. The corresponding output interval can be calculated as follows [MKC09, Gom09]:

$$X^n = \begin{cases} [\underline{x}^n, \overline{x}^n] & \text{if } \underline{x} \geq 0 \text{ or } n \text{ is odd} \\ [\overline{x}^n, \underline{x}^n] & \text{if } \overline{x} < 0 \text{ and } n \text{ is even} \\ [0, \max(\underline{x}^n, \overline{x}^n)] & \text{if } 0 \in X \text{ and } n \text{ is even} \end{cases} \tag{2.52}$$

Accuracy/Overestimation The correctness of a numerical computation is usually denoted by the term *accuracy*. For non-interval representations, the applied metric is the distance of a computed result to the actual correct value, obtained with infinite precision. For standard encodings like fixed point or floating point, the accuracy of arithmetic operations is limited by rounding, under- and overflow, and the precision of the applied number format [Gol91]. For interval arithmetic a different definition of the term *accuracy* is used. Since the infinite-precision result of a computation is always included in an interval, the width or diameter $\overline{x} - \underline{x}$ is used as metric for the accuracy, which is also called *overestimation* in this context [Rum10].

The Dependency Problem One of the main challenges when dealing with interval arithmetic is the so-called *dependency problem*, which can arise when an interval variable occurs multiple times in an equation. If these occurrences are handled as independent inputs with the above defined arithmetic operations for a straightforward IA implementation, an unnecessary growth of the output interval results [Rum10, Gus15]. This behavior can be illustrated by evaluating the function $f(X) = 2X + X^2$ for an interval input $X = [-1, 1]$. A naive implementation handling both occurrences of X as independent inputs would lead to the following result:

$$\begin{aligned} f(X) &= 2X + X^2 \\ &= 2\,[-1,1] + [-1,1]^2 \\ &= [-2,2] + [0,1] \\ &= [-2,3] \end{aligned} \tag{2.53}$$

This computation assumes that the actual value of x, represented by the interval X, can be -1 and 0 at the same time, which results in an overestimation. A more accurate result, by means of a tighter output interval, can be achieved when the dependency of the two inputs is taken into account. One option is to evaluate the complete equation either for the lower or the upper bound of X in order to determine the output bounds:

$$\begin{aligned} f(X) &= [\min\left(f(\underline{x}), f(\overline{x})\right), \max\left(f(\underline{x}), f(\overline{x})\right)] \\ &= [\min\left(f(-1), f(1)\right), \max\left(f(-1), f(1)\right)] \\ &= [\min\left(-1, 3)\right), \max\left(-1, 3)\right)] \\ &= [-1, 3] \end{aligned} \tag{2.54}$$

This approach, however, requires a tracking of the dependency of the input variables. Alternatively, the function $f(X)$ can be rewritten in order to eliminate the double occurrence of the same variable and then be processed in a straightforward way:

$$\begin{aligned} f(X) &= 2X + X^2 \\ &= (X+1)^2 - 1 \\ &= ([-1,1]+1)^2 - 1 \\ &= ([0,2])^2 - 1 \\ &= [0,4] - 1 \\ &= [-1,3] \end{aligned} \tag{2.55}$$

In both cases, the overestimated range $[-2, -1)$ is removed from the output interval.

Implementation An interval number is usually implemented in hardware using two separate fixed point or floating point numbers for the two interval bounds. The corresponding arithmetic is rather slow when running on a standard processor as software implementation, because both interval bounds and special cases are computed subsequently [GEOA06]. Therefore, interval arithmetic is either implemented with two parallel ALUs using fixed point numbers [GEOA06] or with two parallel floating point units (FPUs) [KK06, NSWvG12] to compute the lower and upper bound of a result simultaneously. In addition, the implementation contains a control structure for selecting the operands and handling special cases, as well as respective rounding modules. As a general rule, when applying the given formulas for interval arithmetic with finite precision numbers, the lower bound result is always rounded down (towards zero), the upper bound is always rounded up (towards infinity).
Figure 2.22 shows the general structure of an interval arithmetic unit in a simplified form without rounding and special case handling. Comparable architectures can be found in [GEOA06] for fixed point interval arithmetic, as well as in [KK06] and [NSWvG12] for floating point IA. According to figure 2.22, the operands of an interval operation are selected with multiplexers and then processed with two parallel arithmetic units. These two operation units realize the four basic operations $\circ \in \{+, -, \times, \div\}$ as introduced in equations (2.46) - (2.50). Depending on the operation, as well as the target latency, the required circuitry of the modules varies. While for addition and subtraction one addition/subtraction module per bound is required, for the multiplication operation with lowest possible latency in total four multipliers, as well as comparison operations and additional logic have to be implemented to realize equation (2.48). For the division operation from equations (2.49) and (2.50) two division modules and some additional logic is required [NSWvG12].
Due to the parallel processing, the IA implementation of addition and subtraction can compete with the speed of a standard single-value operation, at the cost of an approximately doubled hardware area and an accordingly increased power consumption. The described modules required to realize the interval multiplication also more than double the area of a single-valued

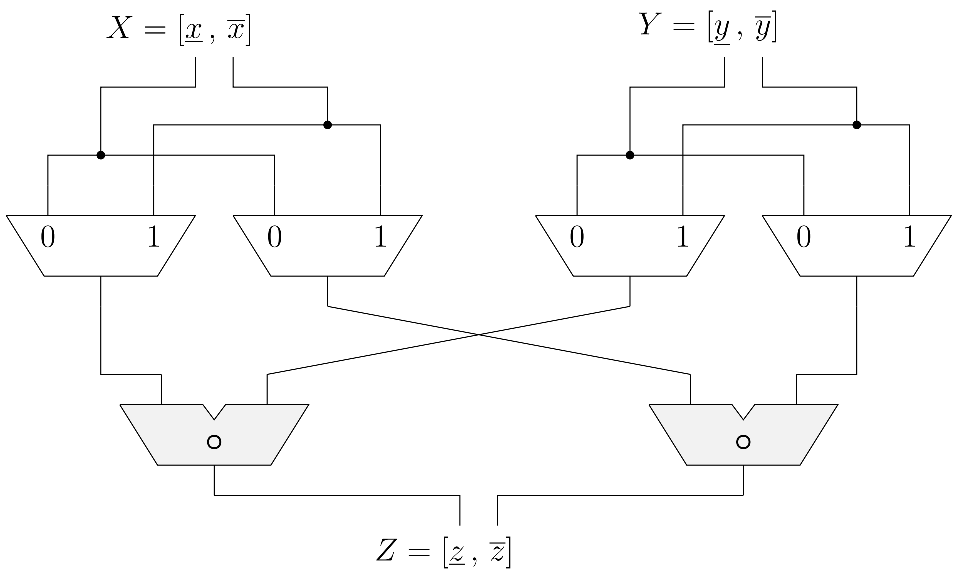

$X = [\underline{x}\,, \overline{x}]$
$Y = [\underline{y}\,, \overline{y}]$
0
1
0
1
0
1
0
1
$\circ$
$\circ$
$Z = [\underline{z}\,, \overline{z}]$

Figure 2.22.: Simplified general structure for an interval arithmetic unit with operand selection and two parallel operation units for $\circ \in \{+, -, \times, \div\}$ [KK06].

multiplication. The expected latency increase is below a factor 1.5 when using four parallel multiplication modules [NSWvG12]. Finally, the additional hardware to realize the interval division again approximately doubles the complexity of single-valued division. Since the division operation in hardware is very slow in general, the additional latency due to the extra logic is negligible [NSWvG12].

At this point the state-of-the-art is left behind and new approaches for digital number representation are discussed. All the encodings presented so far have been published decades ago and are well known concepts in computer arithmetic, whereas the following universal numbers are a relatively new approach. The universal number concept is separated into three distinct number representations, referred to as type-I, type-II and type-III. Since the main contribution of this thesis does not target all of these approaches, but solely the type-II derivative and the corresponding Sets Of Real Numbers, first the general universal number approach with the derived type-I and type-III encodings will be discussed in this chapter in the following sections. The type-II format is then introduced in the next chapter 3.

2.6. Universal Numbers

"*Fewer bits. Better answers.*" is the claim John L. Gustafson made in 2015 [Gus15] when he introduced a new number format, the universal numbers (unums). The proposed format is a new approach for a floating point format tackling the inaccuracy and inefficiency of IEEE floats with an interval-based structure and variable exponent and significant widths. The format will be discussed in detail in the following section 2.6.1, according to [Gus15], [1] and [Bär18].
In the next year, an even more "*radical approach*" for a number format was presented with Sets Of Real Numbers (SORN) [Gus16]. As an extension of the original unum idea, this approach leaves the floating point idea behind and builds a fixed-width, interval-based format with low precision, targeting a regular, very low complex and very fast arithmetic. The remainder of this work deals with SORN arithmetic and its applications. The original SORN approach from [Gus16] is discussed in section 3.1, followed by extensions and adaptions of this initial approach in the rest of chapter 3. Applications for SORN arithmetic are presented in chapter 4.
Since interval arithmetic and variable bitwidth numbers are concepts far away from the established floating point approach unums target to replace, a third, more standard-float-like format was presented in 2017, called *posits* [GY17]. This approach applies to single-valued rounding and fixed overall bitwidth, yet targets to succeed IEEE floats with a higher dynamic range and accuracy, as well as simpler hardware and exception case handling. The posit format will be discussed in section 2.6.2, following [GY17] and [Pos22].
In order to distinguish between the three approaches and since they are all developed from the original unum idea, they will be referred to as unum type-I (original), type-II (SORN) and type-III (posit) in the following.

2.6.1. Type-I: The Original Unums

The original type-I unums adapt the existing structure of IEEE floats presented in section 2.3.1 with a sign bit s, a biased exponent e, and a significant, here called fraction f. The bias is the same as for floats, given in equation (2.28). The concept of normal and subnormal numbers with a hidden bit $h = 1$ for $e > 0$ is also kept. One of the two big differences to floats is the exponent size es and the fraction size fs. These are variable parameters with a value ≥ 1. To be able to track the current exponent and fraction size of a unum at runtime, the values $es - 1$ and $fs - 1$ are encoded in two extra fields right to the fraction, as depicted in figure 2.23. With this approach unum values with different exponent and fraction sizes can be operated and stored, which, in theory, can reduce the memory bandwidth, compared to IEEE floats. The bitwidth of the fields storing the values $es - 1$ and $fs - 1$, however, is fixed. The width of the

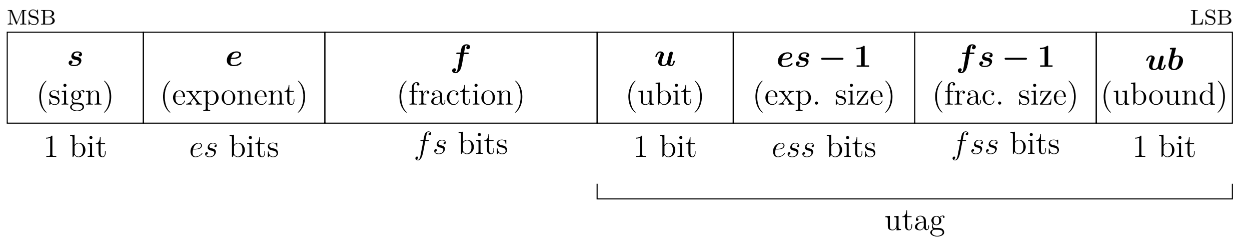


Figure 2.23.: Encoding of type-I unums [Gus15].

field storing the exponent size is ess, the width of the fraction size field is fss. Therefore, the exponent can have a width $1 \leq es \leq 2^{ess}$, and the fraction width is $1 \leq fs \leq 2^{fss}$, respectively. The values $\{ess, fss\}$ define the unum-environment. Operations between unums always have to be in the same environment. This is comparable to the precision of IEEE floats.
The second difference of unums compared to standard floats is the rounding behavior. Whenever a float value exceeds the available precision, it is rounded according to the specified rounding mode, as discussed in section 2.3.1. In contrast, unums are not rounded at all. Instead, when a value is between two representable unums, this is signaled by the so-called *uncertainty* bit or *ubit* u. When the ubit is set to 1_2, the unum represents the open interval between the value encoded with the current exponent and fraction field, and the value which is larger by one unit of least precsision (ulp), which would be a 1_2 added to the LSB of the fraction [Mul05]. The ubit is located right to the fraction, as depicted in figure 2.23. Together with the exponent and fraction size, as well as the later explained ubound bit, it forms the so-called *utag*, the self-descriptive part of a unum. The following example shows the ubit concept with two exact unum values a and c that differ by one ulp, and the inexact unum b in between:

$$\begin{aligned}
a &= 0\ 0011\ 010100\ 0\ 0011\ 0101 = 0.08203125_{10} \\
b &= 0\ 0011\ 010100\ 1\ 0011\ 0101 = (0.08203125_{10}, 0.0830078125_{10}) \\
c &= \underset{s}{0}\ \underset{e}{0011}\ \underset{f}{010101}\ \underset{u}{0}\ \underset{es-1}{0011}\ \underset{fs-1}{0101} = 0.0830078125_{10}
\end{aligned} \tag{2.56}$$

Considering special value encoding, unums avoid the problem of having $2^t - 1$ NaN encodings like IEEE floats do. There are only two NaN values, distinguished between *quiet* and *signaling* by the sign bit, while all other bits are 1_2. The same holds for positive and negative infinity, where all fields but the ubit have all 1_2 bits. However, there is some redundancy in the unum format. Since a zero is encoded by $e = f = 0_{10}$ which can be represented with different exponent and fraction widths, and the sign is irrelevant, there are $2^{ess+fss+1}$ possible zero representations. More details on special cases with unums can be found in [Gus15] and [Bär18], respectively.
In summary, adapting equation (2.31) for unums leads to the following general case unum representation:

$$v_{\text{unum}} = (-1)^s \times \begin{cases} 2^{2-2^{es-1}} \times (\frac{f}{2^{fs}}) & \text{if } \ e = 0_{10} \\ \infty & \text{if } \ \{e, f, es-1, fs-1\} = \text{all } 1_2 \text{ bits } \ \text{and } \ u = 0_2 \\ \text{NaN} & \text{if } \ \{e, f, es-1, fs-1, u\} = \text{all } 1_2 \text{ bits} \\ 2^{e+1-2^{es-1}} \times (1 + \frac{f}{2^{fs}}) & \text{otherwise} \end{cases} \tag{2.57}$$

Ubounds With the ubit concept unums can only represent ulp-wide intervals, yet arithmetic operations with an inexact unum might result in wider intervals. In order to be able to represent these, in [Gus15] *ubounds* are introduced. A ubound is composed of two single unums, one for the lower and one for the upper bound, comparable to interval arithmetic, as introduced in section 2.5. Since these lower and upper unum bounds can be open intervals themselves, such a ubound interval can have both included or excluded endpoints. In order to signal whether an exact or inexact unum represents a stand-alone value or is part of a ubound, the ubound bit is used. The ubound bit is located right to the field denoting the fraction size, as depicted in figure 2.23. More details on ubounds can be found in [Gus15].

Arithmetic Arithmetic with unums and ubounds is a combination of floating point and interval arithmetic. If two exact unums are involved in an operation, standard floating point arithmetic can be used, extended by algorithms which handle the variable size of exponents and fractions. If at least one of the inputs of an arithmetic operation is either an inexact single unum or a ubound, interval arithmetic rules have to be applied. Standard floating point and interval arithmetic are discussed in sections 2.3.2 - 2.3.4 and 2.5, respectively. Details on unum/ubound arithmetic can be found in [Gus15], [1] and [Bär18].

Implementation The main challenge for the hardware implementation of unum/ubound arithmetic are the variable exponent and fraction widths. Since hardware circuits can not be implemented with variable widths, most unum architectures are implemented with maximum size datapaths and use zero padding for values with non-maximum sizes [HZS+17, GMR+18, 1]. This, however, eliminates the advantage of possible savings due to small width unums. Another approach, introduced in [Gus15], is to not carry out arithmetic operations on numbers in unum format directly, but to use an internal, fixed-size and high-precision scratchpad layer, the so-called *general-*, *gbound-* or *g*-layer, along with appropriate conversion functions. This approach is implemented in [BDdD17], using a standard floating point format with some custom adaptions as internal *g*-layer format.
An overview of hardware implementations for unum arithmetic is given in table 2.12. Even though this list might be non-exhaustive, publications on unum type-I hardware implementations are limited. Possible reasons are the challenges introduced by the variable size properties which complicate implementations and somehow counteract the idea of improved efficiency. An intermediate approach is to use unums as a memory format only, where the advantages of small exponent and fractions sizes can be exploited. For arithmetic operations, the unums are converted to a fixed-size floating point format. In [BJC+19] this approach is implemented for variable precision floating point computing.

Table 2.12.: Implementations of type-I unums for different unum environments $\{ess, fss\}$ and hardware platforms.

	operations	**env.**	**HW platform**
[BDdD17]	$+$, $\times$, comparison	$\{4, 6\}$	ASIC
[HZS+17]	$+$, $\times$, FFT	N/A	FPGA
[GMR+18]	$+$, $-$	$\{4, 5\}$	ASIC (fabricated)
[1]	$+$, $-$, $\times 2$, $\div 2$, comparison, CORDIC	$\{3, 5\}$	ASIC, FPGA
[BDdD19]	ISA with $+$, $-$, $\times$, comparison	$\{4, 8\}$	ASIC, FPGA (RISC-V)

2.6.2. Type-III: Posits

Posits are meant to be a drop-in replacement for IEEE floats and are claimed to have several improvements over standard floating point, such as a higher dynamic range and accuracy, lower hardware complexity and less exception cases. They were first presented in 2017 [GY17], resulting from the experiences with the two previous unum versions. In 2022, a posit standard was ratified [Pos22]. Unlike type-I and type-II unums, posits are a single-valued floating point format with a fixed bitwidth, utilizing rounding rather than interval arithmetic, if a value can not be represented with the available precision. Like IEEE floats, posits have a sign bit S, as well as an exponent E and a fraction F, whose definitions, however, differ from standard floats. In addition, posits contain an extra field, called the *regime* R. All fields after the sign bit can have a variable size, while the total bitwidth of a posit number is fixed. The structure of a posit encoding with N bits is shown in figure 2.24, according to [Pos22]. The different fields of a posit are explained in the following:

- The sign bit S determines the sign of a posit. In contrast to IEEE floats, posits were introduced with a two's complement encoding in [GY17] in order to remove the redundancy of a negative zero. This means that if $S = 1_2$, the rest of the binary posit value is considered to be encoded in two's complement. In [Pos22] this behavior is expressed by encoding the implicit integer value represented by the sign bit as $1 - 3S$, leading to a hidden bit h that is either 1_{10} or -2_{10}. This covers the negation and addition of a 1_2 in case of a negative (two's complemented) fraction. In addition, the value of the sign bit is added to the exponent calculation, and the effective exponent is multiplied by $1 - 2S$. This covers the negation and addition of a 1_2 in case of a negative (two's complemented) exponent. The complete description for deriving a decimal posit value with this two's complement handling is given in equation (2.59).

- The regime field R consists of a variable number of k bits with same value, either 0_2 or 1_2, followed by one termination bit with opposite value $\overline{R_0}$. The decimal value of R is determined by the number bits k and by their value:

$$R = \begin{cases} -k & \text{if } R_0 = 0_2 \\ k-1 & \text{if } R_0 = 1_2 \end{cases} \tag{2.58}$$

  In the original proposal [GY17], R was interpreted as power of $2^{2^{es}}$, with the variable exponent size es, which was used to define the posit environment $\{N, es\}$, together with

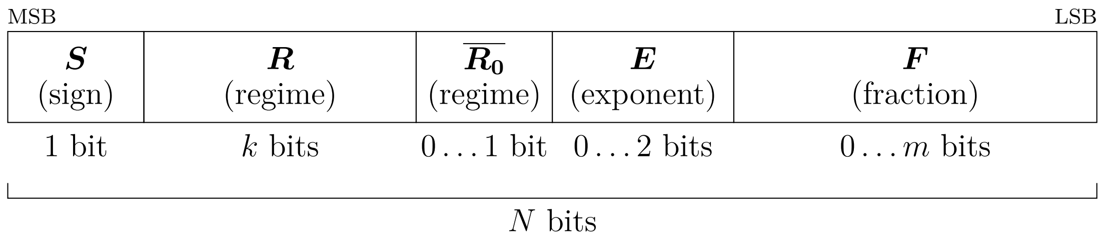

MSB
LSB
$\boldsymbol{S}$ (sign)
$\boldsymbol{R}$ (regime)
$\boldsymbol{\overline{R_0}}$ (regime)
$\boldsymbol{E}$ (exponent)
$\boldsymbol{F}$ (fraction)
1 bit
k bits
$0\ldots1$ bit
$0\ldots2$ bits
$0\ldots m$ bits
N bits

Figure 2.24.: Encoding of posits according to the posit standard [Pos22].

the posit size N. In [Pos22] the exponent size was fixed to 2. Therefore R scales with powers of 16.

- The exponent E is unbiased and interpreted as an unsigned integer, scaling with powers of 2. If there are any bits left after the regime encoding, the exponent can have up to 2 bits, according to [Pos22].
- If there are any bits left after regime and exponent, they encode the fraction F, which follows the same definition as for IEEE floats. Only the hidden bit is defined in a different way for negative numbers, as explained above.

There are two special values with posits, namely zero and Not a Real (NaR), which merges NaN and $\pm\infty$. A zero is encoded with all bits set to 0_2, for NaR the sign bit is 1_2 and all others bits are 0_2.
With all the discussed definitions, the general case representation of a posit value according to [Pos22] reads as follows:

$$v_{\text{posit}} = \begin{cases} 0 & \text{if } \{S, R, E, F\} = 0_{10} \\ \text{NaR} & \text{if } \{R, E, F\} = 0_{10} \text{ and } S = 1_{10} \\ \left((1-3S) + \frac{F}{2^m}\right) \times 2^{(1-2S)\times(4R+E+S)} & \text{otherwise} \end{cases} \tag{2.59}$$

In contrast to IEEE floating point, there is only one rounding mode for posits, which follows a *rounding-to-nearest* method and is defined in [Pos22]. Implementation of correct rounding for posits is more complicated than for standard floats, since, due to the variable field widths, it is not only the fraction that might be rounded, but also the exponent. In addition to rounding, the posit standard also specifies comparisons, arithmetic operations and elementary functions. Comparisons with posits are as simple as with integers, due to the two's complement encoding. Details on posit arithmetic operations can be found in the literature. In the following paragraph several publications related to hardware implementation for posit arithmetic are discussed.

Implementation For the hardware implementation of posits, handling the variable widths of regime, exponent and fraction leads to an overhead in control logic, compared to IEEE floats. The length of the regime has to be decoded first, before exponent and fraction positions can be determined [MDBB20]. Therefore most posit hardware implementations utilize a data extraction module before the actual arithmetic operations are performed. The exponent of a standard float implementation is replaced by a scale factor which represents the combination of regime and exponent of a posit. With this scale factor and the fraction part, the algorithms for performing arithmetic operations on posits are similar to those for standard floats [CGS+18, JS19, MDBB20]. Table 2.13 shows a collection of recent posit hardware implementations for different arithmetic operations and target platforms. Most of them include comparisons to IEEE FP hardware, as shown in table 2.14 for FPGA implementation. Since the original claim for simpler hardware with posits from [GY17] can mostly not be confirmed with the presented and further comparisons from the literature, there are also approaches for approximate computing with posits in order to reduce the hardware complexity [MDBB+22].

Table 2.13.: Posit hardware implementations (non-exhaustive list).

	operations	**bitwidth**	**HW platform**	**quire**
[**CGS$^+$18**]	$+$, $\times$	8, 16, 32	ASIC, FPGA	no
[**PM18**]	$+$, $-$, $\times$	10...32	FPGA	no
[**JS19**]	$+$, $-$, $\times$, $\div$	16, 32	ASIC, FPGA	no
[**UFdD19**]	$+$, $-$, $\times$	16, 32, 64	FPGA (HLS)	yes
[**MDBB20**]	$+$, $-$, $\times$	8, 16, 32	ASIC	no
[**MMDBB21**]	MAC	8, 16, 24, 32	ASIC	yes
[**TGRK21**]	ISA with FMA, $\div$, $\sqrt{}$	32	ASIC, FPGA (RISC-V)	no
[**MMB$^+$22**]	ISA with $+$, $-$, $\times$, $\div$, $\sqrt{}$, MAC	32	ASIC, FPGA (RISC-V)	yes
[**ZKAKP22**]	MAC	8	ASIC	no

Quire The definition of the posit format also includes a scratchpad layer, comparable to the gbound-layer for type-I unums. For posits, this internal format is called the *quire*, and is used for exact accumulations and fused operations. Posits can be converted to the high-precision, signed fixed point quire format to perform operations without intermediate rounding. Solely one single rounding step at the end of a series of operations is required, when the result is converted back from quire to posit format [Gus17]. The bitwidth and encoding of the quire datatype is defined in [Pos22]. Some of the presented hardware implementations from table 2.13 also include quire compatibility for fused operations [UFdD19, MMDBB21, MMB$^+$22].

Table 2.14.: Comparison of 32 bit float and posit implementations on FPGA.

	[**CGS$^+$18**]				[**UFdD19**]				[**MMB$^+$22**]	
device	Zynq-7000				Kintex-7				Kintex-7	
operation	add		mul		add		mul		ISA	
format	float	posit	float	posit	float	posit	float	posit	float	posit
LUT	1049	981	533	572	425	738	80	544	3726	11796
register	-	-	-	-	375	811	193	710	1008	2979
delay [ns]	41.58	40.03	29.05	33.02	2.69	2.66	2.20	2.42	19.82	19.82
DSP	-	-	4	4	0	0	3	4	-	-
cycles	-	-	-	-	14	22	7	21	-	-

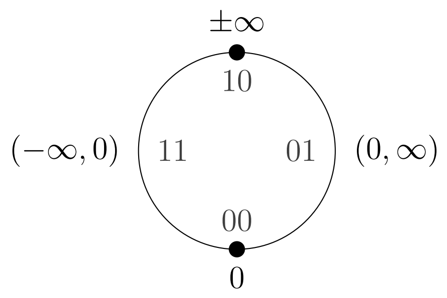
Accuracy Despite the claim of better hardware performance for posits, which is at least questionable considering the publications discussed above, the second argument for posits in [GY17] is a higher dynamic range and a better accuracy of posits compared to IEEE floats of same bitwidth. However, this higher accuracy does not apply for all representable values. While floats have a mostly constant accuracy for all representable values, posits have a so-called tapered accuracy, which is high for small values around 0_{10} (the *sweet spot* or *golden zone*) and decreases for increasing magnitudes [GY17].
Some of the presented publications on hardware implementation [CGS+18, TGRK21, MMB+22] also include accuracy benchmarks comparing posits and IEEE floats. In addition, there are also several software implementations comparing both formats, such as [LLH18] and [dDFMU19]. As a general conclusion the different works mainly confirm the higher dynamic range and the better accuracy of posits. The accuracy improvement is hereby caused by the extra fraction bits a posit value has, compared to a float value of same bitwidth. However, the improved accuracy is only valid within the posit sweet spot around 0_{10}, where the improvement can be up to multiple orders of magnitude [MMB+22]. Yet, there are also various applications and benchmarks (out of the sweet spot) where posits perform worse than floats [dDFMU19].

3 SORN: Sets Of Real Numbers

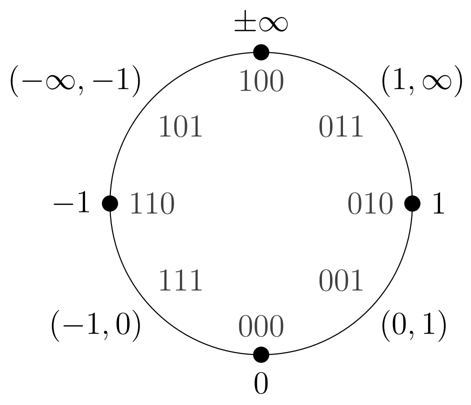
Sets Of Real Numbers (SORN) is a number representation which is derived from the second version of the universal numbers. This type-II unum approach was presented as "*A Radical Approach to Computation with Real Numbers*" [Gus16], because the format differs fundamentally from the commonly used approaches for fixed point, floating point or interval arithmetic. While type-I unums, posits or the Logarithmic Number System are adaptions of floating point, the SORN approach defines a totally different way of encoding and computing numbers, comparable to the dissimilarity of the Residue Number System or stochastic computing. In the following section 3.1, the initial unum type-II approach and the derived SORN arithmetic is presented according to [Gus16].
At the time of writing this, and to the best of the authors knowledge, for type-II unums and SORN arithmetic, no publications other than the original [Gus16] and the authors publications presented in this thesis exist, neither on hardware or software implementation, nor any other related topic. As discussed in section 1.1 and figure 1.2, respectively, the main contributions of this work are the implementation, evaluation, optimization and application of the SORN number format. After the introduction of the initial approach in section 3.1, the remainder of this chapter presents adaptions of the originally proposed SORN datatype structure in section 3.2, discusses the developed SORN design-flow in section 3.3, evaluates on the hardware complexity of basic SORN arithmetic components in section 3.4, and introduces fused SORN arithmetic in section 3.5. Applications of the SORN number format are discussed in chapter 4.

3.1. Type-II Unums and the SORN Approach

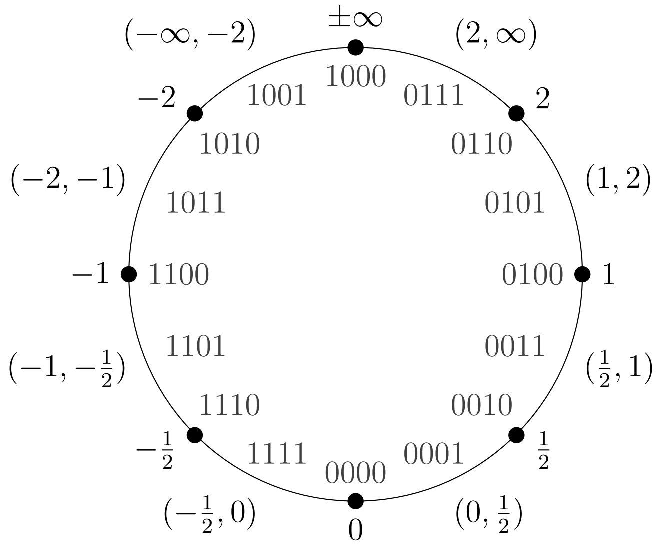
Type-I unums turn out to be very challenging to implement in hardware, as discussed in section 2.6.1. Following these experiences, the second unum version targets some aspects of an ideal number format, namely a regular and easy hardware implementation with equally fast arithmetic operations, no exception cases and no rounding errors. The last goal is achieved by the open interval style enabled by the ubit from type-I unums, which is therefore kept for type-II. The general idea of the second unum approach is to represent all real numbers in a closed form. Since an exact representation of every real number is not possible with finite precision, the ubit is used to denote open intervals. The real number line is interpreted as a circle, with both ends $-\infty$ and $+\infty$ merged together to a single value $\pm\infty$, assumed to be an exact value and the reciprocal of zero. This interpretation of the reals is depicted in figure 3.1a. In this very simple resolution, the reals are represented by 0, $\pm\infty$ and two open intervals encompassing all positive and all negative reals, respectively. The corresponding binary unum-II representation is composed of two bits with the LSB serving as the ubit and



(a) 2 bit unums



(b) 3 bit unums



(c) 4 bit unums

Figure 3.1.: Representations of type-II unums with the decimal values in the outer and the binary encoding in the inner circle [Gus16].

the MSB as the sign bit of a two's complement encoding. This format can now be extended by one bit, adding the exact values 1_{10} and -1_{10}, as well as the corresponding open intervals. This 3 bit unum-II representation is shown in figure 3.1b. Further extensions include another exact value, its negative version, and the positive and negative reciprocal. Figure 3.1c shows a 4 bit unum-II format with the additional exact values 2, −2, $^1/_2$ and $-^1/_2$, as well as the open intervals in between. Due to the preserved symmetry, the given representation is closed under negation and reciprocation. Negation follows the two's complement encoding by negating all bits and adding a 1_2. This corresponds to a reflection on the vertical axis. For a reciprocation, all bits but the sign bit are negated, and a 1_2 is added. This corresponds to a reflection on the horizontal axis.

The format is defined by the chosen exact values, which are called the *u-lattice* or *lattice values*. Except the endpoints zero and infinity and the mandatory value 1_{10}, any lattice values $> 1_{10}$ can be chosen. The example from figure 3.1c is composed of the lattice values $\{0, 1, 2, \pm\infty\}$, their reciprocals, negations and the open intervals in between. Instead of 2, any real value can be chosen to extent the lattice values, for example 10, 256, or even π. The number of lattice values defines the bitwidth (and precision) of the binary unum-II representation.

If algorithms for arithmetic operations would now be developed for this unum-II format, a traditional, type-I-like arithmetic would result. This algorithms would also require a solution to handle wider intervals, comparable to ubounds. In addition, different operations would require different implementation complexity. All this is avoided with a new number format and arithmetic: The Sets Of Real Numbers.

3.1.1. SORNs

To begin with, it has to be clarified what otherwise might lead to a misconception: **SORNs are not identical to type-II unums**, they are an own representation, derived from unum-II, but way more powerful. With the above defined unums, the real numbers $\mathbb{R}$ are represented with a set of exact values and intervals, for example $\{\pm\infty, (-\infty, 0), 0, (0, \infty)\}$ or $\{\pm\infty, (-\infty, -1), -1, (-1, 0), 0, (0, 1), 1, (1, \infty)\}$. Every element of these sets can be considered as a subset of the reals. The SORN format represents the power set of the given unum type-II representation.

Definition 3.1.1. A power set of a set $\mathcal{S}$ includes all possible subsets of $\mathcal{S}$. Given a set $\mathcal{S} = \{a, b, c\}$, the power set of $\mathcal{S}$ is $\{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$ [BSMM08].

In other words, a value in SORN format can not only represent one of the given subsets, as the unum format does, but any combination of them. Therefore the representation is called *Sets Of Real Numbers*, with emphasis on the plural *Sets*.
The number of bits for the binary SORN encoding equals $2^{\text{unum bits}}$, or the number of elements in the unum set. Every SORN bit encodes the absence (0_2) or presence (1_2) of a unum value. The MSB is hereby used to represent $\pm\infty$, followed by the remaining elements sorted from negative to positive towards the LSB. In table 3.1 the encoding for 4 bit SORNs is given, corresponding to the 2 bit unums from figure 3.1a. Accordingly, the 3 and 4 bit unums in figures 3.1b and 3.1c can be represented with 8 and 16 SORN bits, respectively. Table 3.2 shows some examples for 8 and 16 bit SORNs.

Table 3.1.: Encoding of 4 bit SORNs.

SORN	decimal	SORN	decimal
0000	$\varnothing$	1000	$\pm\infty$
0001	$(0, \infty)$	1001	$(0, \infty]$
0010	0	1010	$0 \cup \pm\infty$
0011	$[0, \infty)$	1011	$[0, \infty]$
0100	$(-\infty, 0)$	1100	$[-\infty, 0)$
0101	$(-\infty, 0) \cup (0, \infty)$	1101	$[-\infty, 0) \cup (0, \infty]$
0110	$(-\infty, 0]$	1110	$[-\infty, 0]$
0111	$(-\infty, \infty)$	1111	$[-\infty, \infty]$

Table 3.2.: Examples of 8 and 16 bit SORN encodings ($N = 3$ and $N = 4$).

8 bit SORN	decimal	16 bit SORN	decimal
00011100	$(-1, 1)$	0000111111100000	$[-1, 1/2]$
00001111	$[0, \infty)$	1000000001111111	$(0, \infty]$
00110000	$[-1, 0)$	0011100000001110	$[-2, -1] \cup [1, 2]$
10000001	$(1, \infty]$	0000000000000110	$(1, 2]$

The encoded set of lattice values and intervals will be called the SORN *datatype* in the following. Details on SORN datatypes are discussed in section 3.2.

3.1.2. LUT-based Arithmetic

The implementation of arithmetic operations between two or more SORN operands relies on table lookup. For any arithmetic operation and a designated SORN datatype, a lookup table (LUT) can be derived, containing the results for every possible combination of the two (or more) input SORNs. The entries of that table are obtained following the rules of interval arithmetic, as presented in section 2.5. Table 3.3 shows the LUT for the addition of two 8 bit SORNs, encoded according to the datatype from figure 3.1b. This table can now be used to perform additions via table lookup, but only for SORN values with a single 1_2 bit, also called *one-hot* encoding. For operations on SORNs with multiple 1_2 bits, the LUT has to be called multiple times, providing the result for every one-hot input combination. These intermediate results are then combined with an OR operation to obtain the final result. The following equation (3.1) shows this behavior by adding the SORN values 00001110 and 00000100, which correspond to the decimal intervals $[0, 1]$ and $(0, 1)$:

$$\begin{array}{rl}
 & \underset{[0,1]}{00001110} + \underset{(0,1)}{00000100} \\
\hline
 & \underset{0}{00001000} + \underset{(0,1)}{00000100} = \underset{(0,1)}{00000100} \\
\text{OR} & \underset{(0,1)}{00000100} + \underset{(0,1)}{00000100} = \underset{(0,\infty)}{00000111} \\
\text{OR} & \underset{1}{00000010} + \underset{(0,1)}{00000100} = \underset{(1,\infty)}{00000011} \\
\hline
 & \qquad\qquad\qquad\qquad = \underset{(0,\infty)}{00000111}
\end{array} \tag{3.1}$$

Table 3.3.: LUT for the addition of two 8 bit SORNs.

$\boldsymbol{x+y}$	$\boldsymbol{\pm\infty}$ 10000000	$\boldsymbol{(-\infty,-1)}$ 01000000	$\boldsymbol{-1}$ 00100000	$\boldsymbol{(-1,0)}$ 00010000	$\boldsymbol{0}$ 00001000	$\boldsymbol{(0,1)}$ 00000100	$\boldsymbol{1}$ 00000010	$\boldsymbol{(1,\infty)}$ 00000001
$\boldsymbol{\pm\infty}$ 10000000	11111111	10000000	10000000	10000000	10000000	10000000	10000000	10000000
$\boldsymbol{(-\infty,-1)}$ 01000000	10000000	01000000	01000000	01000000	01000000	01110000	01110000	01111111
$\boldsymbol{-1}$ 00100000	10000000	01000000	01000000	01000000	00100000	00010000	00001000	00000111
$\boldsymbol{(-1,0)}$ 00010000	10000000	01000000	01000000	01110000	00010000	00011100	00000100	00000111
$\boldsymbol{0}$ 00001000	10000000	01000000	00100000	00010000	00001000	00000100	00000010	00000001
$\boldsymbol{(0,1)}$ 00000100	10000000	01110000	00010000	00011100	00000100	00000111	00000011	00000001
$\boldsymbol{1}$ 00000010	10000000	01110000	00001000	00000100	00000010	00000011	00000001	00000001
$\boldsymbol{(1,\infty)}$ 00000001	10000000	01111111	00000111	00000111	00000001	00000001	00000001	00000001

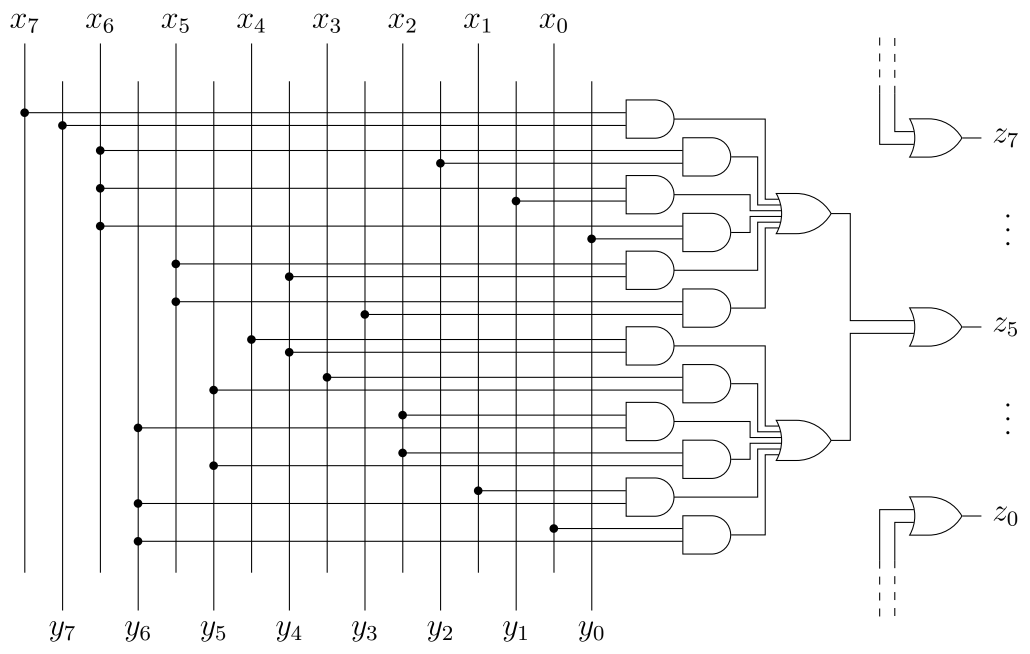


Figure 3.2.: ROM hardware circuit for bit z_5 of an 8 bit SORN addition $z = x + y$, according to table 3.3.

The hardware implementation of the SORN LUTs can be realized with (non-programmable) read-only memory (ROM) circuits, which consist of wired connections and basic logic gates. Figure 3.2 shows the logic circuit for obtaining the bit z_5 (exact -1_{10}) for the result of an 8 bit SORN addition $z = x + y$ according to table 3.3. The remaining bits of the result z are implemented in a similar way. According to [Gus16], the benefit of this kind of arithmetic is that a SORN LUT and its corresponding hardware circuit can be implemented for any operation, without a difference in chip area or circuit delay. Section 3.4 will evaluate on this hypothesis.

3.2. Datatypes

The set of lattice values and their distribution as exact values and intervals is called the SORN *datatype*. In contrast to positional number systems like integers, floats, type-I unums or posits, the decimal value represented by a certain SORN bit or bit string is not entirely fixed by the format itself, but can be varied by the choice of lattice values and interval distribution. According to [Gus16], the three mandatory lattice values $\{0, 1, \pm\infty\}$ and the general structure with reciprocals, negatives and open intervals is predetermined, but the further lattice values can be freely chosen, as described in section 3.1. This version of the SORN number format will be called the *original* or *unum-II based* SORN format in the remainder of this work. The following section 3.2.1 will introduce a formal mathematical description of this format, according to [5] and [8].
Since SORN arithmetic relies on LUTs which are created for every operation and datatype combination, there are possibilities to further adapt the formats structure and create suitable LUTs accordingly. Some drawbacks of the original format are discussed in section 3.2.2, which leads to a datatype structure without exact values and with half-open intervals. This structure is presented and mathematically formulated in section 3.2.3, according to [3] and [5]. The choice of lattice values and interval distributions throughout this work is discussed in section 3.2.4, further customized and intermediate SORN datatypes are proposed in section 3.2.5.
Evaluations on the differences in algorithmic performance between the unum-II based, the half-open and the custom datatypes will be presented in sections 4.2 - 4.4 for the application of symbol detection in wireless multiple-input and multiple-output (MIMO) communication. The hardware performance measures of the corresponding arithmetic LUTs for the different datatypes will be evaluated and compared in section 3.4.

3.2.1. Original Format

The original, unum-II based SORN format from [Gus16], presented in section 3.1, can be formulated as a general, mathematical definition, according to [5] and [8]. The format is defined by a set of N lattice values l_i with $i \in \{0, \ldots, N-1\}$ and $N > 2$. The special case $N = 2$ corresponds to the 4 bit SORNs with $l_i \in \{0, \pm\infty\}$, described in table 3.1. This case will not be covered by the following formal definition.
For a general datatype, the mandatory lattice values are $l_0 = 0$, $l_1 = 1$, $l_{N-1} = \pm\infty$, accompanied by further, user-selected lattice values $1 < l_i < \infty$, with $i \in \{2, \ldots, N-2\}$. The general lattice $\mathcal{L}$ is composed of

- the lattice values l_i,
- the reciprocals $1/l_i$ of all user-selected lattice values $1 < l_i < \infty$, and
- the negations of all lattice values and reciprocals, except for l_0 and l_{N-1}.

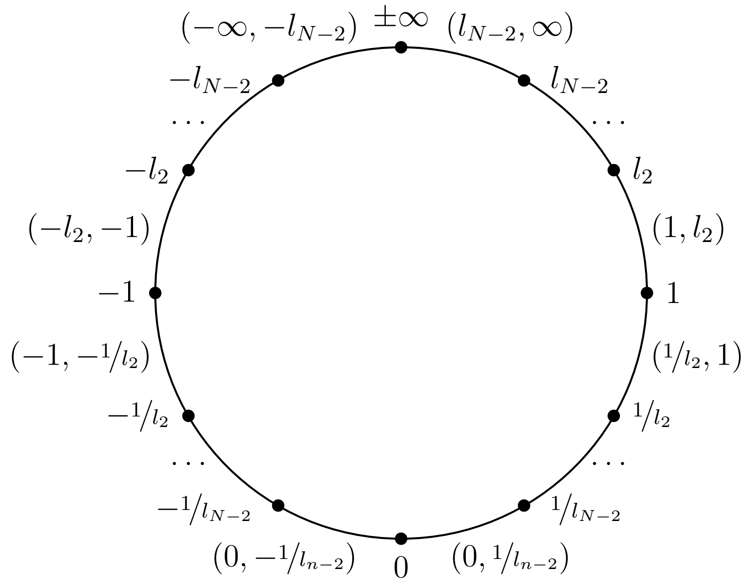


Figure 3.3.: General definition of the original, unum-II based SORN datatype.

This formulates the exact values within the lattice $\mathcal{L}$. In addition, every gap between two exact values is filled by an open interval with the two adjacent values as boundaries. With these definitions, the general lattice $\mathcal{L}$ reads as follows:

$$\begin{aligned}\mathcal{L} = \{ \pm\infty, (-\infty,-l_{N-2}), -l_{N-2}, \dots, -l_2, (-l_2,-1), -1, (-1,-1/l_2), -1/l_2, \dots, -1/l_{N-2},\\ (-1/l_{N-2},0), 0, (0,1/l_{N-2}), 1/l_{N-2}, \dots, 1/l_2, (1/l_2,1), 1, (1,l_2), l_2, \dots, l_{n-2}, (l_{N-2},\infty)\}\end{aligned} \tag{3.2}$$

This general lattice also defines the structure of the original, unum-II based SORN datatype, as depicted in figure 3.3. The bitwidth w_s of this datatype is $w_s = 8 \times (N-2)$, with $N > 2$. The representation given in figure 3.1c is an example for this definition with $N = 4$, $l_i \in \{0, 1, 2, \pm\infty\}$ and a SORN bitwidth $w_s = 8 \times (4-2) = 16$. The next larger SORN datatypes would contain $N = 5$ and $N = 6$ lattice values, resulting in $w_s = 24$ and $w_s = 32$ SORN bits, respectively.
Table 3.4 gives an overview of the original SORN datatypes implemented throughout this work. Note that in [4] and [6], the exact value for $\pm\infty$ is split in two bits for $-\infty$ and $+\infty$. This is done to align the bitwidth with the also implemented half-open datatypes in order to allow a fair comparison of the utilized hardware.

Table 3.4.: Overview of the implemented original SORN datatypes throughout this work.

label	**lattice values**	$\boldsymbol{N}$	$\boldsymbol{w_s}$	**adaptions**	**reference**
unum8	$\{0, 1, \pm\infty\}$	3	8	-	[2, 3, 5]
unum9	$\{0, 1, \pm\infty\}$	3	9	separate $\pm\infty$ bits	[4, 6]
unum16	$\{0, 1, 2, \pm\infty\}$	4	16	-	[2, 3, 5]
unum17	$\{0, 1, 2, \pm\infty\}$	4	17	separate $\pm\infty$ bits	[4, 6]
unum32	$\{0, 1, 2, 4, 8, \pm\infty\}$	6	32	-	[2]

3.2.2. Drawbacks of the Original Format

The original, unum-II based SORN datatype relies on horizontal symmetry with respect to the values -1_{10} and 1_{10} on the unum circle, in order to provide reciprocal closure. In addition, exact values and open intervals are distinguished by the ubit. These two properties, however, are only exploited in the binary unum encoding. When dealing with SORN values, the general method of arithmetic with pre-computed LUTs can be applied to any SORN datatype, independent of its internal structure. This gives the designer the freedom to adapt the existing datatype structure and to chose or evaluate the most-suitable SORN datatype for a specific application. The following drawbacks of the original datatype structure are target of improvement when adapting the SORN representation.

Precision The precision of the format is quite low and scales poorly. The insertion of a single lattice value l_i requires 8 extra bits for the SORN datatype: one for the value itself, the reciprocal, the negative, the negative reciprocal, and one new open interval for each of these four exact values.

Exact Values When the exact values within the SORN datatype do not match the application data, these SORN bits are barely ever addressed without one of the adjacent open intervals during arithmetic operations, which makes them de-facto-redundant. Even if one of the formats exact values matches some of the input data of a SORN algorithm, once an operation results in an interval SORN, further calculations with this value will always result in intervals, making the formats other exact values de-facto-redundant. Table 3.5 lists the occurrence of all exact results except zero for the two original datatypes *unum8* and *unum16* for two-operand addition and multiplication. The table includes two scenarios, where either only one-hot inputs or all possible SORN input combinations are applied. A valid SORN input is hereby considered as a bit string with a single 1_2 bit (one-hot) or consecutive 1_2 bits. Bit strings like 10011010_2 are not considered, because they usually do not occur when applying straight forward SORN arithmetic (even though they are valid SORN values in theory).

Table 3.5.: Exact result count within the original SORN datatypes.

OP	datatype	inputs	No.	No. exact results						
			cases	$\pm\infty$	$+1$	-1	$+2$	-2	$+\frac{1}{2}$	$-\frac{1}{2}$
$+$	*unum8*	one-hot	64	14	2	2	-	-	-	-
$+$	*unum8*	all possible	1296	56	2	2	-	-	-	-
$\times$	*unum8*	one-hot	64	13	2	2	-	-	-	-
$\times$	*unum8*	all possible	1296	31	2	2	-	-	-	-
$+$	*unum16*	one-hot	256	30	5	5	3	3	4	4
$+$	*unum16*	all possible	18496	240	5	5	3	3	4	4
$\times$	*unum16*	one-hot	256	29	6	6	4	4	4	4
$\times$	*unum16*	all possible	18496	127	6	6	4	4	4	4

With the number of SORN bits w_s, the number of possible one-hot inputs for two-input addition and multiplication is w_s^2. When considering all possible SORN inputs, the number of input cases is $1/4\,(w_s^2 + w_s)^2$, derived from the formula for triangle numbers (Gaußsche Summenformel) [BSMM08].
According to table 3.5, for one-hot inputs and *unum8*, about 22% (addition) and 20% (multiplication) of the cases result in exact $\pm\infty$, for *unum16* its about 11% (addition and multiplication). When considering all possible SORN inputs, the percentage of exact $\pm\infty$ results drops to 4% (addition) and 2% (multiplication) for *unum8*. For *unum16* its 1% (addition) and 0.7% (multiplication). The other exact result occurrences are (even more) negligible, which is why they are removed with the half-open SORN datatypes.

3.2.3. Half-Open Intervals

Both issues discussed in the previous section can be improved with a SORN representation that dismisses all exact values, except for zero, and uses half-open intervals instead. This removes the described de-facto-redundancy, improves the information-per-bit and therefore also the precision scaling. Removing the mandatory lattice value 1_{10} and the requirement to accompany every introduced lattice value with a reciprocal further increases the flexibility of a designer to increase the precision or improve the dynamic range without adding too much unwanted complexity.
This described SORN representation will be called *half-open* in the remainder of this work. As introduced in [3] and [5], this general SORN datatype is composed of an exact zero framed by half-open intervals towards negative and positive infinity, which include their respective lower/upper bound. The adapted lattice reads as

$$\mathcal{L} = \{[-\infty, -l_{N-2}), [-l_{N-2}, -l_{N-3}), \ldots, [-l_1, 0), 0, (0, l_1], \ldots, (l_{N-3}, l_{N-2}], (l_{N-2}, \infty]\} \qquad (3.3)$$

and consists of N lattice values l_i with $i \in \{0, \ldots, N-1\}$, $N > 1$, $l_0 = 0$ and $l_{N-1} = \infty$. The structure is depicted in figure 3.4. The corresponding SORN bitwidth is $w_s = 2N - 1$, since any introduced lattice value l_i with $0 < l_i < \infty$ only introduces two extra bits each representing one half-open interval.
An overview of several half-open datatypes implemented throughout this work is given in table

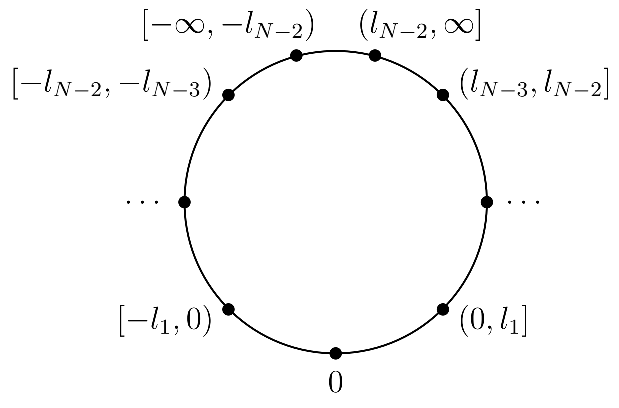


Figure 3.4.: General definition of the half-open SORN datatype.

3.6. The distribution of the lattice values l_i within the representation is chosen to be either linear or logarithmic, as indicated by the label $<$*lin/log*,$w_s$$>|^{<max\ value>}$. In order to distinguish between representations of same bitwidth and value distribution, the respective maximum lattice value $l_{N-2} < \infty$ is included in the label as well. From the specified lattice values in the table, the datatype can be constructed according to equation (3.3). In the following equations (3.4) and (3.5), the two datatypes *lin11*$|^1$ and *log11*$|^1$ from table 3.6 are given as examples:

$$\begin{aligned} lin11|^1 = \{&[-\infty,-1),[-1,-3/4),[-3/4,-1/2),[-1/2,-1/4),[-1/4,0),\\ &0,(0,1/4],(1/4,1/2],(1/2,3/4],(3/4,1],(1,\infty]\} \end{aligned} \tag{3.4}$$

$$\begin{aligned} log11|^1 = \{&[-\infty,-1),[-1,-1/2),[-1/2,-1/4),[-1/4,-1/8),[-1/8,0),\\ &0,(0,1/8],(1/8,1/4],(1/4,1/2],(1/2,1],(1,\infty]\} \end{aligned} \tag{3.5}$$

Table 3.6.: Overview of the implemented half-open SORN datatypes throughout this work.

label	**lattice values l_i**	N	w_s	**reference**
log5$\|^1$	$\{0,1,\infty\}$	3	5	-
lin7$\|^3$	$\{0,3/2,3,\infty\}$	4	7	[10]
lin7$\|^{100}$	$\{0,50,100,\infty\}$	4	7	[10]
log7$\|^1$	$\{0,1/2,1,\infty\}$	4	7	[10]
log7$\|^{64}$	$\{0,32,64,\infty\}$	4	7	[10]
lin9$\|^1$	$\{0,1/3,2/3,1,\infty\}$	5	9	[5, 7, 10]
lin9$\|^{120}$	$\{0,40,80,120,\infty\}$	5	9	[10]
log9$\|^1$	$\{0,1/4,1/2,1,\infty\}$	5	9	[5]
log9$\|^2$	$\{0,1/2,1,2,\infty\}$	5	9	[4, 10]
log9$\|^{128}$	$\{0,32,64,128,\infty\}$	5	9	[10]
lin11$\|^1$	$\{0,1/4,1/2,3/4,1,\infty\}$	6	11	[3, 5, 7, 10]
lin11$\|^{200}$	$\{0,50,100,150,200,\infty\}$	6	11	[10]
log11$\|^1$	$\{0,1/8,1/4,1/2,1,\infty\}$	6	11	[5]
log11$\|^2$	$\{0,1/4,1/2,1,2,\infty\}$	6	11	[10]
log11$\|^{256}$	$\{0,32,64,128,256,\infty\}$	6	11	[10]
lin13$\|^1$	$\{0,1/5,2/5,3/5,4/5,1,\infty\}$	7	13	[5, 9]
lin13$\|^2$	$\{0,1/4,1/2,3/4,1,2,\infty\}$	7	13	[3, 7]
log13$\|^1$	$\{0,1/16,1/8,1/4,1/2,1,\infty\}$	7	13	[5]
lin15$\|^2$	$\{0,1/4,1/2,3/4,1,3/2,2,\infty\}$	8	15	[7]
lin15$\|^{300}$	$\{0,50,100,150,200,250,300,\infty\}$	8	15	[8]
log15$\|^{512}$	$\{0,16,32,64,128,256,512,\infty\}$	8	15	[8]
log17$\|^8$	$\{0,1/8,1/4,1/2,1,2,4,8,\infty\}$	9	17	-
lin19$\|^2$	$\{0,1/4,1/2,3/4,1,5/4,3/2,7/4,2,\infty\}$	10	19	-

3.2.4. Choice of Lattice Values and Distribution

The choice of lattice values and their distributions within the SORN datatypes listed in table 3.6 might seem quite random and raise the question how SORN datatypes in general, and the presented datatypes in particular, are set up. From a general hardware architects point of view, the datatype is always set up to match a specific application and its data, just like it is done for standard formats like fixed or floating point, where the format and its bitwidth are chosen according to the application or system requirements. For the SORN format, however, there are more parameters to set. The usual approach for setting up a SORN datatype is to start with the dynamic range required for the target application, which is given by the maximum non-infinity lattice value. Next, the precision by means of the number of intervals between zero and the maximum lattice value is set, which defines the bit width of the SORN datatype. In addition, the interval distribution has to be specified, which can be linear, logarithmic or any other distribution. The given parameters of dynamic range, precision and distribution can be varied in order to evaluate on the complexity-performance trade-off within the design space of the target application. For any application, the best suited SORN datatype can be evaluated with the automated design flow presented in section 3.3.
Within this work, the *log* based datatypes like *log5*$|^1$, *log9*$|^2$ or *log17*$|^8$ are chosen to match the lattice values and interval distribution of the original unum-based datatypes, in order to ease a fair comparison between both approaches. In addition, the linear distributed datatypes with a similar value range are introduced to compare against the *log* based approach. In the remainder of this work, those datatypes presented in table 3.6 with a small value range ≤ 3 are used for a MIMO symbol detection application which will be presented in section 4.2. Those datatypes with large value ranges > 100 are used for an image processing application, presented in section 4.1. A further customization of the SORN datatypes for specific applications is discussed in the next section 3.2.5.
In addition to the described, explorative SORN datatype design approach followed throughout this work, also more theoretically determined datatypes are applicable. One possible approach is to take into account the statistics of a certain application and to match the interval distribution

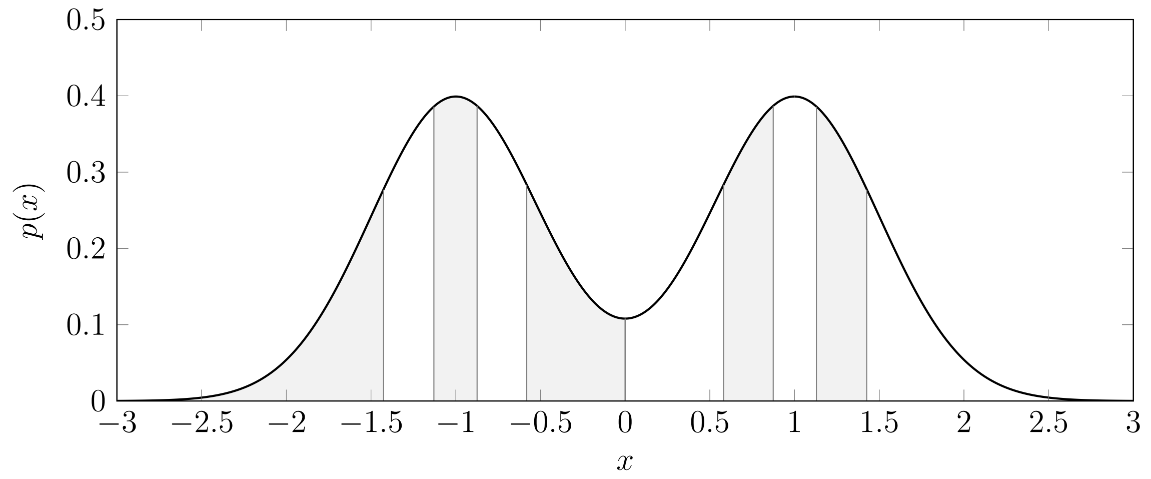


Figure 3.5.: Probability density function of a noisy, normal distributed BPSK signal, segmented into intervals $[\underline{x}, \overline{x}]$ with equal probabilities $P(\underline{x} \leq x \leq \overline{x}) = \int_{\underline{x}}^{\overline{x}} p(x)dx$.

of the SORN datatype accordingly. As a brief outlook to section 4.2, the value distribution for a received signal within a wireless communication system using a binary phase-shift keying (BPSK) modulation is considered as an example. The transmitted signal of either -1_{10} or 1_{10} is disturbed by noise during the transmission, resulting in a normal distribution of the received values, as depicted in figure 3.5. In order to process the received signal with SORN arithmetic, the datatype could be chosen to contain intervals with an equal probability $P(\underline{x} \leq x \leq \overline{x}) = \int_{\underline{x}}^{\overline{x}} p(x)dx$. A possible interval distribution is depicted in figure 3.5 and would lead to the following 13 bit SORN datatype:

$$\begin{aligned} \mathit{stat13}|^{3} = \{&[-\infty,-3),[-3,-1.426),[-1.426,-1.129),[-1.129,-0.874),\\ &[-0.874,-0.581),[-0.581,0),0,(0,0.581],(0.581,0.874],\\ &(0.874,1.129],(1.129,1.426],(1.426,3],(3,\infty]\} \end{aligned} \tag{3.6}$$

According to the notation for SORN datatypes introduced above, this statistically determined datatype can be labeled as $\mathit{stat13}|^{3}$. In sections 4.2.5 and 4.2.7 evaluations on a SORN preprocessor for symbol detection in a wireless MIMO system using a BPSK modulation are presented, which take into account the $\mathit{stat13}|^{3}$ datatype in figures 4.12a, 4.12b and 4.18a. The presented results show that for this particular application and datatype, the statistic approach does not improve the logarithmic and linear based distributions applied in the remainder of this work.

3.2.5. Custom Datatypes

As discussed in the previous section, the structure of a SORN datatype and the utilized lattice values can be chosen and evaluated according to a given application. Even though the half-open datatype structure from equation (3.3) targets to improve the properties of the original datatype, further adaptions might be beneficial for certain applications. In general, it is possible to customize a SORN datatype in any way. The presented half-open structure can be seen as a customization of the original one, but it can be further adapted. In the following, the adaptions made to the general half-open datatype structure throughout this work are discussed. These customized half-open SORN datatypes are listed in table 3.7. The datatype labels $<\!\mathit{lin/log}, w_s\!>|^{<\mathit{max\ value}>}_{<\mathit{adaptions}>}$ extend the general half-open labels by the adaptions made to the general structure.

Custom Value Distribution Some of the linear labeled datatypes from table 3.6 are actually not strictly linear. The datatypes $\mathit{lin13}|^{2}$ and $\mathit{lin15}|^{2}$ have different lattice value distributions for $0 \leq l_i \leq 1$ and $1 \leq l_i \leq 2$. Both distributions can, however, be considered linear, wherefore the respective datatypes are labeled accordingly. The same holds for the *lin15* and *lin17* datatypes from table 3.7.

Exact Values The general half-open datatype structure removes exact values in order to avoid redundancy when these values are not used without adjacent intervals. For some applications, however, it can be beneficial to re-introduce single exact values that match the

Table 3.7.: Overview of the implemented custom half-open SORN datatypes throughout this work.

label	lattice values l_i	N	w_s	adaptions	ref.
$lin6\|_{nz,nn}^{250}$	$\{0, 50, 100, \ldots, 250, \infty\}$	7	6	no exact zero, no negatives	[8]
$log10\|_{nn}^{256}$	$\{0, 2, 4, 8, 16, \ldots, 256, \infty\}$	10	10	no negatives	[8]
$lin11\|_{nz,nn}^{250}$	$\{0, 25, 50, 75, \ldots, 250, \infty\}$	12	11	no exact zero, no negatives	[8]
$lin13\|_{e1/2}^{1}$	$\{0, 1/4, 1/2, 3/4, 1, \infty\}$	6	13	exact value $1/2$	[3]
$lin15\|_{e1/2}^{2}$	$\{0, 1/4, 1/2, 3/4, 1, 2, \infty\}$	7	15	exact value $1/2$	[3]
$lin17\|_{e1/2}^{3}$	$\{0, 1/4, 1/2, 3/4, 1, 2, 3, \infty\}$	8	17	exact value $1/2$	[3]
$lin17\|_{e1/2}^{2}$	$\{0, 1/4, 1/2, 3/4, 1, 3/2, 2, \infty\}$	8	17	exact value $1/2$	[3]
$lin17\|_{e1}^{2}$	$\{0, 1/4, 1/2, 3/4, 1, 3/2, 2, \infty\}$	8	17	exact value 1	[4]
$lin17\|_{e1/\sqrt{2}}^{2}$	$\{0, 1/4, 1/2, 1/\sqrt{2}, 1, 3/2, 2, \infty\}$	8	17	exact value $1/\sqrt{2}$	[7]

application data. The respective datatypes in table 3.7 are labeled $< \cdots > |_{e<exact\ value>}$. The following equation (3.7) gives the datatype $lin13|_{e1/2}^{1}$ from table 3.7 as an example:

$$\begin{aligned} lin13|_{e1/2}^{1} = \{&[-\infty, -1), [-1, -3/4), [-3/4, -1/2), -1/2, (-1/2, -1/4), [-1/4, 0), \\ &0, (0, 1/4], (1/4, 1/2), 1/2, (1/2, 3/4], (3/4, 1], (1, \infty]\} \end{aligned} \tag{3.7}$$

No Negatives or Exact Zero Both the original and the general half-open SORN datatype structures provide a representation for all real numbers between $-\infty$ and $+\infty$. For some applications, such a coverage is not necessary and can be reduced in order to save bitwidth and hardware complexity. There are many applications where negative numbers are not required, for example when representing the pixel values of an image. For such an application, the SORN bitwidth can be almost halved by removing all negative intervals. In the datatype label this is indicated with $< \cdots > |_{nn}$ for *no negatives*. In a similar manner, some applications do not require an exact zero value. This adaption is indicated with $< \cdots > |_{nz}$ for *no exact zero*. Combinations of these attributes are also possible. Equation (3.8) gives the datatype $lin6|_{nz,nn}^{250}$ from table 3.7 as an example:

$$lin6|_{nz,nn}^{250} = \{[0, 50], (50, 100], (100, 150], (150, 200], (200, 250], (250, \infty]\} \tag{3.8}$$

3.3. Automated Design-Flow

As introduced in the previous section, SORNs are a highly flexible number format. In contrast to standard formats like integer, fixed point or floating point, bitwidth is not the only adjustable parameter. Starting from the original SORN version, the choice of lattice values offers a certain amount of flexibility when designing a SORN datatype for a specific application. Applying the half-open or customized datatypes enables an even higher level of flexibility, not only by the choice of lattice values, but also by their distribution, possible single exacts and present or absent zero, infinity or negative values. The overall goal of the SORN design process is to find the best-suited SORN datatype for a specific algorithm, which delivers meaningful results while maintaining a reasonable amount of hardware resources. Performing this design space exploration can be a tedious task, especially when all the different designs have to be implemented manually. In order to perform this task in a reasonable amount of design time, an automation of the process is indispensable.

Design-Flow The general SORN design flow, which is developed and used throughout this work, is depicted in figure 3.6 and will be presented in the following.

1. As for any digital design process, the starting point is a specified **algorithm** for a dedicated application. In addition, a **datatype** has to be set up, which should be tailored towards the algorithm or application and its input and output data, for example by setting an appropriate dynamic range or matching regular input values.
2. The next step is to create the SORN arithmetic **LUTs** for the chosen datatype and all the operations and functions that are required for the specified algorithm.
3. The LUTs are implemented on register-transfer level (RTL) using AND and OR gates, as described in section 3.1.2 and shown in figure 3.2. This results in single **RTL modules** for every required operation.
4. According to the algorithm, the RTL modules for the different arithmetic operations are connected to form the entire SORN **datapath**.

By performing the described steps, a single SORN design can be created for the specified algorithm. Repeating the design process and evaluating the resulting datapaths regarding algorithmic and hardware performance for different SORN datatypes leads to an explored design space and eventually to the best-suited configuration for the specified application problem.

Automation The described SORN design space exploration may require a lot of effort when all the required steps of LUT, RTL and datapath generation have to be carried out manually for various datatypes. In order to facilitate this design process, a python tool for the automated generation of SORN datapaths was created. This "*SORN Hardware Generator*" [5] is an open source python tool, available on GitHub [Bär19]. The tool delivers not only the SORN arithmetic LUTs and respective RTL modules per datatype and operation, but also connects these modules to a complete SORN datapath, including an automated pipeline register insertion. The target algorithm, the SORN datatype and the pipeline configuration

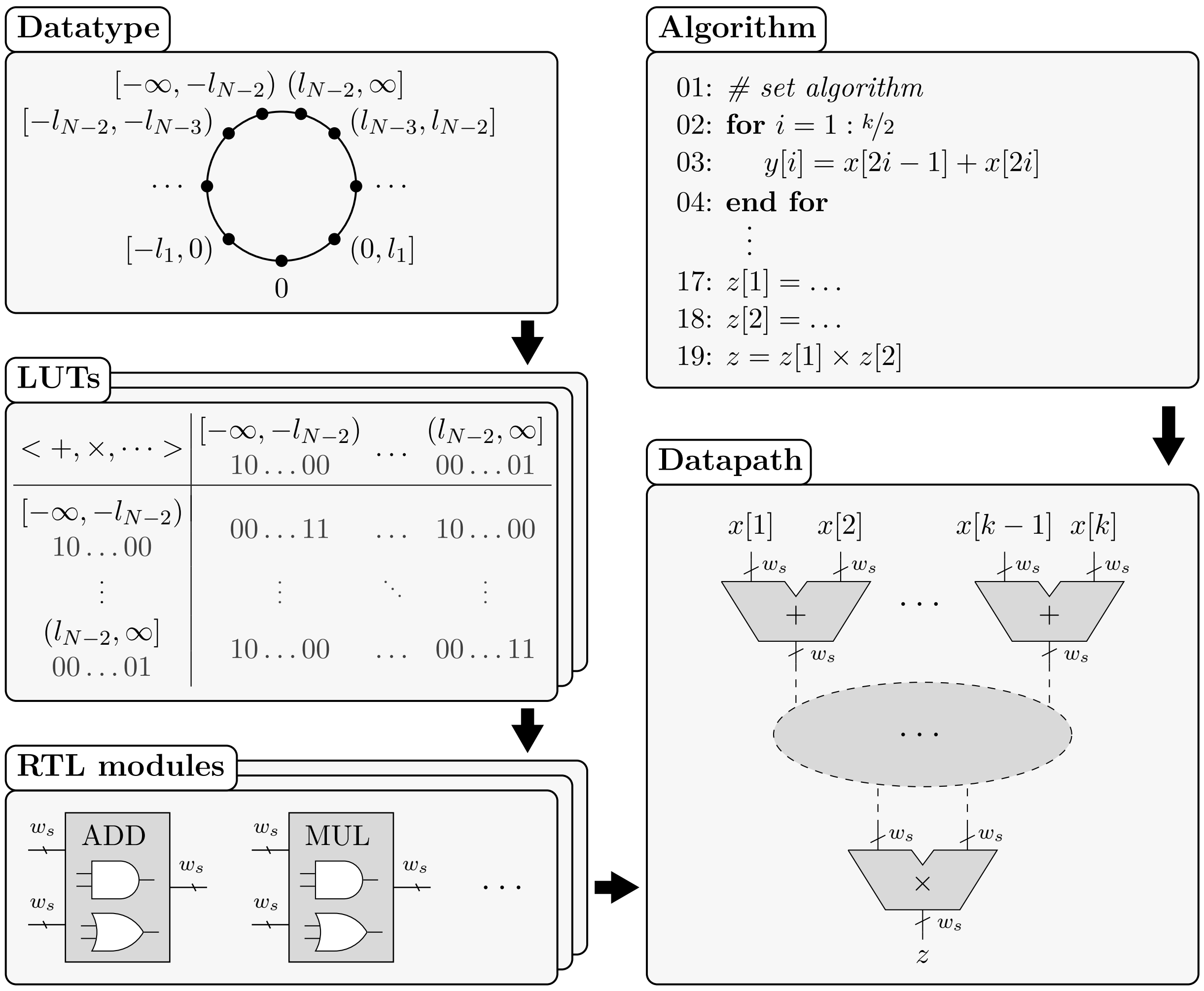


Figure 3.6.: Proposed SORN design flow, starting from the datatype and algorithm specification to create the required arithmetic LUTs and respective RTL modules, and eventually set up the entire datapath.

together form the input specification for the tool, which will be discussed in the following section 3.3.1. A detailed explanation of the SORN LUT generation and the template-based synthesis of the corresponding RTL descriptions is given in sections 3.3.2 and 3.3.3, respectively. Implementation details on the final step, the automated generation of the entire datapath, are not discussed, since this is not a primarily contribution of the author. Details on this part can be found in [5].

3.3.1. Algorithm and Datatype Specification

The input of the SORN Hardware Generator tool is a specification file which includes

1. the name of the top level RTL design,
2. the SORN datatype,
3. the number of required pipeline registers and
4. the targeted arithmetic algorithm.

An example for such an input file is given in listing 3.1. The top level name is indicated with the keyword `@name`, followed by the SORN datatype, indicated by the keyword `@datatype`. The configuration offers three different kinds of datatypes: The first configuration is a half-open datatype with a linear lattice value distribution, as introduced in section 3.2.3. The definition reads as follows:

```
['lin', '[<start>,<stop>,<step>]','zero', 'negatives', 'infinity']
```

The first keyword introduces the distribution, followed by the definition of lattice values. The `<start>` value sets the smallest lattice value (typically zero), the `<stop>` value the largest non-infinity one. The value of `<step>` defines the step size between the start and stop values. Following the lattice value definition, the three options for an exact zero, negative values and an infinity lattice value are set. An inclusion of the respective keyword sets the corresponding option. Any combination of the keywords is possible. The example in listing 3.1 sets a datatype with linear distributed values between zero and one with a step size of 0.25, an exact zero and an infinity value. This corresponds to the $lin11|^1$ datatype from table 3.6 and equation (3.4), but without negative values.
The second possible datatype configuration is a half-open datatype with a logarithmic lattice value distribution, also introduced in section 3.2.3. The definition

```
['log', '[<start>,<stop>]','zero', 'negatives', 'infinity']
```

is similar to the linear one, the only difference is that the `<step>` parameter is not required. The datatype is built with lattice values between $2^{\texttt{<start>}}$ and $2^{\texttt{<stop>}}$, followed by the three optional parameters which serve the same purpose as for the linear distribution.
Finally, the third possible datatype configuration is fully manual and reads as follows:

```
['man', '{<element1>;<element2>;...}']
```

With this configuration, every element of the datatype can be defined as exact value, open or half-open interval with a customized lattice value spacing, using the respective round or square brackets. All regular linear or logarithmic datatypes can also be specified with this manual

Listing 3.1 Specification file for the SORN Hardware Generator tool [Bär19, 5].

```
# 1/ set file name
@name top_level

# 2/ set datatype
@datatype ['lin', '[0,0.25,1]', 'zero', 'infinity']

# 3/ set number of pipeline stages
@pipeline 1

# 4/ set equation(s)
z1 = x1 + x2
z2 = x3 + x4
z  = z1 * z2
```

option. Further, this configuration can also be used to define the customized datatypes from table 3.7, the original datatypes from table 3.4, or any other SORN datatype.
The third specification in the input file is the number of required pipeline registers to be inserted in the datapath. This option is set by the keyword `@pipeline` which can be followed by any integer value ≥ 0. The pipeline stages are automatically placed between two SORN LUT blocks in the top level architecture, targeting an equal distribution within the datapath.
The fourth and last specification is the targeted arithmetic algorithm. The algorithm is specified with single equations, where every input, intermediate value and output is given as a labeled variable. Since the equations are interpreted with the `numpy` package in python, all the arithmetic operations and functions available in `numpy` can be used. The tool automatically detects those variables corresponding to an input and output. These are added to the entity of the final RTL top level architecture. For every equation in the specification file, a submodule is created and instantiated in the top level.

3.3.2. LUT Generation

For every arithmetic operation and function from the algorithm specified in the input file, a SORN LUT object is created within the tool. The inputs of this LUT generation process are the number of operands, the operation or function, and the SORN datatype. In a first step, the decimal LUT is created and stored with floating point values. The inputs of the LUT are the elements of the specified datatype, which can be either exact values, open, half-open or closed intervals. Internally, exact values are handled as closed intervals with the same lower and upper boundary $\underline{x} = \overline{x}$, in order to handle calculations with both exact and interval inputs. The results for the LUT are obtained by following the rules of interval arithmetic, which are discussed in section 2.5. In particular, two-input operations are implemented according to equation (2.45), single-input operations according to equation (2.51). Therefore, the tool is limited to monotonic single-input functions. These functions, however, can also be piece-wise monotonic, as long as the interval boundaries wherein they are monotonic, are also present as lattice values in the defined datatype. Considering the function $f(X) = X^2$ with the interval input X, defined according to equation (2.44), f can be considered monotonic decreasing for $\overline{x} \leq 0$ and monotonic increasing for $\underline{x} \geq 0$. Consequently, as long as 0_{10} is a lattice value within the datatype, the function can be evaluated correctly.
All previously discussed interval arithmetic operations are defined for closed intervals. Since the presented SORN datatypes from section 3.2 can contain any combination of closed, open or half-open intervals, the *open/closed* boundary conditions for the LUT outputs also have to be considered when calculating the respective results as discussed before. Therefore, whenever a LUT output is calculated according to equation (2.45) or (2.51), the corresponding boundary condition is also taken into account. For single-input operations, the input condition can be simply forwarded to the output. For two-input operations, the output boundary is *open* if at least one of the operand boundaries is *open*. This corresponds to a Boolean OR operation between the operand boundaries, where the *open* condition is encoded as a 1_2 and the *closed* condition as a 0_2.
Once the LUT is set up with float intervals and corresponding interval boundaries, every entry is converted into binary SORN representation. This is done by iterating over all elements of

the defined SORN datatype. If the current float output interval entirely or partly matches with an element of the datatype, the respective SORN bit is set to 1_2. After all elements of the datatype are iterated, the SORN result is stored in a binary LUT. This float-to-SORN conversion includes a certain rounding, since the float results are mapped onto the predefined intervals of the SORN datatype. For example, consider the $lin11|^1$ datatype from equation (3.4) and the single-input operation $f(X) = X^2$. When creating the float LUT entry for the datatype element $X = (1/2, 3/4]$, the output is

$$f\left((1/2, 3/4]\right) = (1/2, 3/4]^2 = (1/4, 9/16] \tag{3.9}$$

which is not an element of the SORN datatype. When converting this result to a binary SORN value, the output yields a SORN result with a rounded upper boundary:

$$00000001100_2 = (1/4, 3/4] \tag{3.10}$$

After all entries of the float LUT are converted to binary SORN representation with the described method, the LUT is structured like the one depicted in table 3.3 and can be implemented on RTL.

3.3.3. RTL Implementation

The RTL implementation of the previously defined SORN LUTs per arithmetic operation is realized with the help of predefined VHDL templates, which are customized according to the respective operation. These templates include generic building blocks like an entity, architecture, signal declaration and assignment, etc., and are customized with the user-defined parameters like the operation name and the bitwidth of the datatype. The main part to be customized by the tool, however, is the implementation of the functionality by means of the bit assignments to encode the binary SORN outputs of the LUT. These bit assignments are realized with AND and OR operations, in order to describe a logic circuit with the structure shown in figure 3.2. Such a circuit does not only consider one-hot inputs, as a single instance of the LUT does, but every possible combination of input bits.

For the creation of this logic-level description, the tool iterates over all the w_s bits in the SORN datatype, and for every bit a loop over both operands x and y is performed. Whenever the LUT output for the current operand combinations has a 1_2 at the position w_s, an AND of the two corresponding operand bits is added to the VHDL template. Every instance of (x_i AND y_j) is separated with an OR. This algorithm is described in listing 3.2 with python-style pseudo code. As an example, the generated VHDL description for the bit z_5 from table 3.3 and figure 3.2 would read as follows:

```
result(5) <= (x(0) AND y(6)) OR (x(1) AND y(6)) OR ... OR (x(7) AND y(7));
```

Such a line is created for every bit of the output and added to the VHDL template. The result of this process is a VHDL-based RTL description per required arithmetic operation, specified in the arithmetic algorithm, as described in section 3.3.1. These arithmetic SORN building blocks are then instantiated within a top level module to form the entire datapath, realizing the specified algorithm. Details on this last part of the SORN Hardware Generator tool are described in [5].

Listing 3.2 Python-style pseudo code for the template-based creation of VHDL files for implementing the SORN LUTs [Bär19].

```
    # loop over bits in datatype
    for bit in range(0,len(SORNdatatype)):
       vhdlSTR = "result(" + str(bit) + ") <= "
       # loop over values in operand 0
       for op0 in range(0,len(SORNdatatype)):
          # loop over values in operand 1
          for op1 in range(0,len(SORNdatatype)):
             currentSORNresult = LUT.resultSORN[op0][op1]
             # write resulting assignment
             if currentSORNresult[bit] == 1:
                if not isFirstAssignment:
                   vhdlSTR = vhdlSTR + "OR "
                vhdlSTR = vhdlSTR + "(x(" + str(op0) + ") AND y(" +
                 ↪  str(op1) + ")) "
          # write end of line
          if op0 == len(SORNdatatype)-1:
             vhdlSTR = vhdlSTR + ";\n"
```

3.4. Hardware Complexity

One of the main motivations for the development of type-II unums and SORNs is the difficult design process and the high hardware complexity when implementing type-I unums [Gus16]. Therefore some properties are aimed for when developing the SORN format, namely a high energy efficiency, a high speedup compared to legacy formats, and a regular arithmetic structure, leading to a simplified design process and equally complex circuits. The presented automated design flow for SORN arithmetic including the generation of dedicated RTL descriptions for different SORN operations contributes to such a simplification of the design process. However, the choice of the SORN datatype is still up to the designer and has to be evaluated according to a specific application. The different SORN applications presented in chapter 4 therefore include evaluations on architectures utilizing different SORN datatypes, as well as comparisons to reference implementations.
This section discusses an application-independent evaluation of the hardware complexity of SORN modules for basic arithmetic operations, namely addition and multiplication. On top of comparing these two operations, various SORN datatypes are evaluated against each other, as well as against integer/fixed point reference modules. The following datatype classes are considered in this evaluation:

- **Unum-based SORN:** The evaluation includes circuits for the unum-based SORN datatypes *unum8* - *unum17* from table 3.4. The circuits for *unum8* and *unum16* are implemented as parallel instances of the respective one-hot SORN LUTs whose outputs are combined with OR gates in order to allow multiple-hot inputs, as proposed in [Gus16] and described in section 3.1.2. The circuits for *unum9* and *unum17* are implemented with the SORN Hardware Generator described in section 3.3.

- **Half-open SORN:** The evaluation further includes circuits for various half-open and custom SORN datatypes from tables 3.6 and 3.7 with widths of $5, 9, 11, 13$ and 17 bit, all implemented with the SORN Hardware Generator.

- **Integer/fixed point:** In order to compare the SORN modules against a reference design, the evaluation includes circuits for integer/fixed point addition and multiplication for all utilized SORN bitwidths $5, 8, 9, 11, 13, 16$ and 17. Addition is implemented as ripple-carry adder. For the implemented array multipliers, the output is rounded to the respective input bitwidth in order to make a fair comparison to the SORN modules.

All implemented RTL modules were synthesized for a 28 nm CMOS SOI technology from STMicroelectronics (STM) using Genus Synthesis Solution software from Cadence. The target frequency is set to 1 GHz. The synthesis results for area utilization, critical path delay and power consumption are presented in figures 3.7 and 3.8 for the addition and multiplication modules, respectively. These results will be analyzed in detail in the following. The exact results can also be found in the appendix B.1 in table B.1. Note that some of the SORN modules for different datatypes but same bitwidths show equal results, because their circuits are identical.

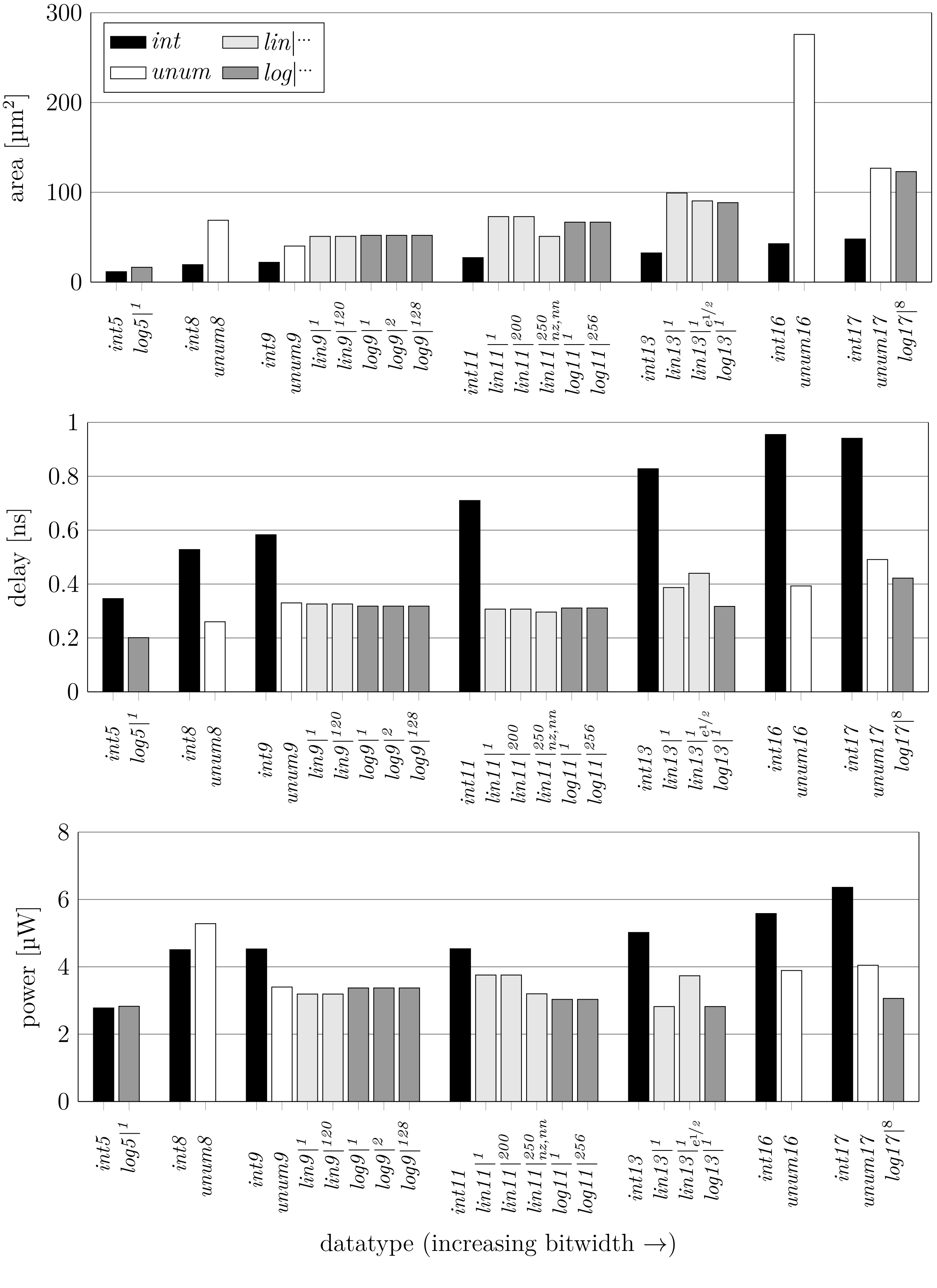


Figure 3.7.: Synthesis results for **addition** components without pipeline stages for $f = 1\,\text{GHz}$ and CMOS 28 nm technology.

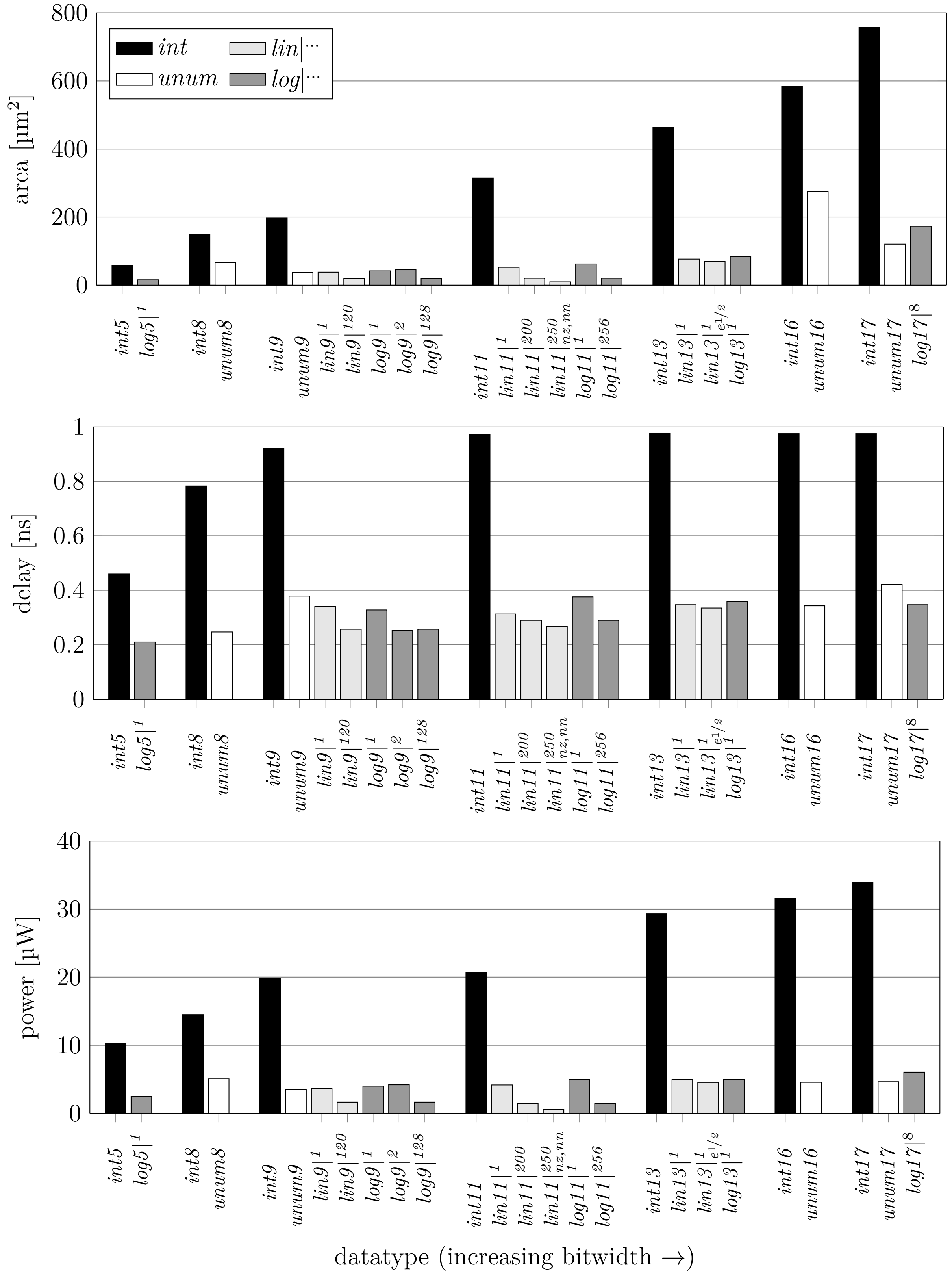


Figure 3.8.: Synthesis results for **multiplication** components without pipeline stages for $f = 1\,\text{GHz}$ and CMOS 28 nm technology.

3.4.1. SORN vs. Integer/FxD

Before comparing the hardware performance of the different SORN datatypes against each other, it is essential to investigate whether the SORN format in general can compete with or even outperform classical number formats in terms of hardware measures, as it was aimed for. Therefore the implemented SORN modules are compared against a standard signed integer arithmetic, which is also capable of handling fixed point numbers. Among the standard formats, integers are considered the least complex.
Evaluating on the synthesis results displayed in figures 3.7 and 3.8, it can be generalized that SORNs outperform integers in all but one category, which is the area utilization of the addition components. Here the respective integer design is always smaller than the SORN design of the same bitwidth, on average the SORN modules more than double the integer area utilization. However, this larger area does not lead to a longer critical path or a higher power consumption of the SORN addition modules, compared to integer. On the contrary, except for 5 and 8 bit, all SORN components show a lower power consumption than the integer reference of same bitwidth. With increasing bitwidth, this difference between SORN and integer also increases towards halving the integer power consumption for the 17 bit case. For the critical path delay, the gap between SORN and integer is even larger. For small bitwidths, the SORN components are nearly twice as fast as the integer ones, which increases even more towards 17 bit.
Considering the multiplication modules, the gap between integer and SORN is even larger. While the SORN modules show a moderate area increase for growing bitwidths, the integer area utilization grows much more rapidly. For 5 bit, the integer area compares to SORN by a factor of almost 4, which increases even more for higher bitwidths. A similar behavior can be observed for the power consumption, ranging from a factor of 4 to almost 7 when comparing integer to the worst SORN module per bitwidth. Finally, the critical path delay for the integer multiplication is between 2 and 3 times longer than for the corresponding SORN module.
It has to be noted that this evaluation considers only the hardware aspects of the presented designs. From an algorithmic point of view, a bitwise comparison between integer and SORN arithmetic blocks is hardly fair, because the two formats follow completely different approaches and differ in attributes like precision, dynamic range, or interval and set representation. In chapter 4 application specific SORN implementations are therefore compared to reference designs on both hardware and algorithmic performance to determine their suitability.

3.4.2. SORN vs. SORN

After showing how SORN hardware behaves in comparison with integers, it is interesting to evaluate the performance among the various SORN datatypes. This comparison is manifold and therefore split into the following subgroups.

Unum-based vs. unum-based To begin with, the datatypes *unum8* and *unum16*, implemented as parallel one-hot LUTs, are compared against *unum9* and *unum17*, implemented as pure logic gates with the SORN Hardware Generator. For both addition and multiplication, the area utilization of *unum8/16* is higher than for *unum9/17*, by an average factor of 2. For the path delay, an opposite behavior can be observed. Here the *unum9/17* datatypes show an average delay increase of factor 1.3 compared to *unum8/16*. For the power consumption, only the 8 and 9 bit datatypes differ noticeable, by a factor around 1.5 in favor for *unum9*. For the higher bitwidths, only a very small difference of around 2% - 4% is visible. Considering all three measures combined, it can be stated that the *unum9/17* datatypes perform better than *unum8/16*. In detail, the area-power-timing (APT) product of *unum8/16* increases by a factor between 1.7 and 2, compared to *unum9/17*. This shows that the SORN logic for multiple-hot inputs is more efficient than the parallel one-hot LUT implementation.

Unum-based vs. half-open The next step in the evaluation is to compare the unum-based with the half-open SORN datatypes. For this comparison, two approaches can be followed:

1. **Same bitwidth:** The more intuitive way is to compare unum-based against half-open linear and logarithmic datatypes with the same bitwidth, considering those datatypes containing lattice values within the same range. When comparing *unum9* with the different *lin/log9* datatypes for addition, leaving out $lin9|^{120}$ and $log9|^{128}$, it can be observed that the area utilization is about 20% lower for the unum-based module, while delay and power consumption are almost equal. For 9 bit multiplication, all three measures are also on a similar level, with some small outliers in both directions, resulting from the different half-open modules.
   For 17 bit addition, the half-open datatype $log17|^{8}$ shows slightly better results than *unum17*, with the highest difference in the power consumption, which is reduced by about 25%. For 17 bit multiplication, *unum17* has a 30% lower area and 25% lower power consumption than $log17|^{8}$, but also an about 20% longer path delay. In general, in can be concluded that unum-based and half-open datatypes of same bitwidth show a comparable hardware performance with some outliers in both directions.

2. **Same lattice values:** One could argue that comparing unum-based and half-open datatypes with the same bitwidth is not fair, because the half-open datatypes have a higher dynamic range and precision, because they leave out exact values and utilize more lattice values at the same bitwidth. The second approach is therefore to compare those datatypes that utilize the same lattice values. In this evaluation, this would be *unum8/9* vs. $log5|^{1}$ and *unum16/17* vs $log9|^{2}$. Consequently, the $log17|^{8}$ datatype would actually compare to *unum32/33*, which was not considered here.
   Following this approach for comparison, the half-open datatypes easily outperform the unum-based ones in all measures, especially for the area utilization, which is more than halved for all configurations.

Half-open vs. half-open Figures 3.7 and 3.8 cover different linear, logarithmic and custom half-open datatypes per bitwidth. Differences among these will be analyzed in the following, separated into three categories:

1. **Linear vs. logarithmic:** Both linear and logarithmic datatypes are implemented for 9, 11 and 13 bit. The evaluations show no results that indicate one of the two options to be advantageous in terms of hardware performance. For multiplication, it even turns out that the circuits for $lin9|^{120}$ and $log9|^{128}$ are identical, as well as the circuits for $lin11|^{200}$ and $log11|^{256}$.

2. **Small vs. large lattice values:** For addition, the choice of small or large lattice values does not have an influence on the hardware performance. For multiplication, however, those datatypes with large lattice values show a much better hardware performance for all measures. The reason is that for multiplication with large values, most results utilize solely the infinity interval which simplifies the circuit, whereas multiplication with values around 1_{10} stays in that range and utilizes more SORN bits.

3. **Lin/log vs. custom:** Considering the two custom datatypes, it can be observed that an exact value as in $lin13|^{1}_{e^{1/2}}$ does not have a major impact on the hardware performance, whereas leaving out negative values as in $lin11|^{250}_{nz,nn}$ drastically improves the hardware performance for multiplication.

Addition vs. Multiplication Finally, the initial goal of equally complex arithmetic operations with SORNs needs to be evaluated. Analyzing the results from figures 3.7 and 3.8, as well as table B.1, it can be observed that for the unum-based datatypes, not equal but similar results are obtained, even for both implementations methods. For the half-open datatypes, this does not apply. Except for 17 bit, all half-open multiplication modules show a lower area than the addition ones, especially noticeable for datatypes with large lattice values, as discussed above. In contrast, for datatypes with small lattice values the power consumption is noticeably higher for multiplication.
In conclusion, the hardware performance of the operation blocks created with the SORN Hardware Generator is highly datatype and operation dependent and does not provide equally complex operation blocks.

3.5. Fused SORN Arithmetic

For traditional number formats like floating point, the main reason for accuracy degradation during computation is the accumulation of rounding errors. The term *accuracy* can hereby be defined as the distance of a computed result to the actual correct value, obtained with infinite precision. Despite rounding, also under- and overflow, as well as the applied precision affect the accuracy of a single-valued number system like floats [Gol91]. In most cases, when sufficient precision is applied, the accumulated rounding error is rather small compared to the computed values and does not have a large impact on the outcome of a computation. However, due to some cases where actual accidents happened because of rounding errors [Szp13], and to address the accuracy requirements of some applications like scientific computing, the reduction of rounding errors for floating point arithmetic is still a major issue. A commonly used approach to achieve this reduction is the implementation of fused operations, where multiple arithmetic operations are mapped into a joint function block. A prominent example for a fused floating point operation is the fused multiply-add (FMA) or multiply-accumulate (MAC), which is included in the IEEE-754 standard [IEEE08]. For such a fused operation, rounding is performed only at the end, omitting the rounding of the intermediate result and reducing the accumulated error. The following definition of a fused operation will be used throughout this work, independent of the number system.

Definition 3.5.1. A fused operation is an expression containing two or more mathematical operations that is evaluated exactly and converted back to the machine-representable form only at the end [9].

For SORN arithmetic, accuracy degradation during computation is a challenge as well, but with another definition of the term *accuracy*. As described in section 2.5, for interval arithmetic number formats, the distance of the two interval bounds, also called the diameter of the interval, is used as a measure for accuracy. For SORNs, the width of a represented interval can be measured by the number of consecutive 1_2 bits. The most accurate case is a one-hot SORN value, representing only one interval from the specified datatype, whereas the least accurate case with all bits set to 1_2 usually represents the unspecific interval $[-\infty, \infty]$ (if infinity and negative values are included in the datatype). Complex SORN computations can accumulate to such wide intervals which lack of usability. Consequently, improving the accuracy of SORN arithmetic operations means reducing the interval growth during computation. As with traditional formats like floats, this can be achieved with fused arithmetic, where multiple operations are combined in one SORN LUT without intermediate quantization, according to the definition 3.5.1 given above. The following sections will introduce and evaluate fused SORN arithmetic for single-, two- and three-input operations.
From an implementation point of view, SORNs are highly suited for fused arithmetic, since the software-defined SORN LUT generation can be easily adapted for more complex operations as well. For the process described in section 3.3.2, the arithmetic operation has to be changed from a single operation like $x \times y$ to a more complex function like $(x \times y)^2$. Since the LUT entries are computed as floats before they are converted to SORN, the general structure of the LUT and the resulting RTL description do not change. With some further adaptions to the tool, this process can also be used to generate SORN LUTs for three-input fused operations.

3.5.1. Single- and Two-Input Fused Operations

The previously mentioned process of growing SORN intervals can be best explained with an example [9]. Consider the operation $(x \times y)^2$, composed of two-input multiplication followed by a single-input square operation. Tables 3.8a and b show the SORN LUTs for multiplication and squaring, using a 5 bit half-open SORN datatype with values between 0_{10} and 1_{10}, which is an extract from the *lin13*$|^1$ datatype from table 3.6. With this LUTs, the datapath of the squared multiplication operation can be set up. The upper half of figure 3.9 shows this non-fused operation for the two input values x and y:

$$\begin{aligned} x &= 00010_2 = (3/5, 4/5] \\ y &= 00001_2 = (4/5, 1] \end{aligned} \tag{3.11}$$

After multiplication, the intermediate SORN result is quantized from the true mathematical product $x \times y = (12/25, 4/5]$ to the SORN representable interval $(2/5, 4/5]$, including an overestimation of $(2/5, 12/25]$. The two intervals from the intermediate result $(2/5, 3/5]$ and $(3/5, 4/5]$ are

Table 3.8.: LUTs for the SORN operations (a) multiplication and (b) square, fused to (c) the LUT for the square of two multiplied operands [9]. All LUTs use a 5-bit SORN datatype which is extracted from the *lin13*$|^1$ datatype from table 3.6.

$x \times y$		$(0, \frac{1}{5}]$	$(\frac{1}{5}, \frac{2}{5}]$	$(\frac{2}{5}, \frac{3}{5}]$	$(\frac{3}{5}, \frac{4}{5}]$	$(\frac{4}{5}, 1]$
		10000	01000	00100	00010	00001
$(0, \frac{1}{5}]$	10000	10000	10000	10000	10000	10000
$(\frac{1}{5}, \frac{2}{5}]$	01000	10000	10000	11000	11000	11000
$(\frac{2}{5}, \frac{3}{5}]$	00100	10000	11000	11000	01100	01100
$(\frac{3}{5}, \frac{4}{5}]$	00010	10000	11000	01100	01110	00110
$(\frac{4}{5}, 1]$	00001	10000	11000	01100	00110	00011

(a)

x^2
10000
10000
11000
01110
00011

(b)

$(x \times y)^2$		$(0, \frac{1}{5}]$	$(\frac{1}{5}, \frac{2}{5}]$	$(\frac{2}{5}, \frac{3}{5}]$	$(\frac{3}{5}, \frac{4}{5}]$	$(\frac{4}{5}, 1]$
		10000	01000	00100	00010	00001
$(0, \frac{1}{5}]$	10000	10000	10000	10000	10000	10000
$(\frac{1}{5}, \frac{2}{5}]$	01000	10000	10000	10000	10000	10000
$(\frac{2}{5}, \frac{3}{5}]$	00100	10000	10000	10000	11000	11000
$(\frac{3}{5}, \frac{4}{5}]$	00010	10000	10000	11000	11100	01110
$(\frac{4}{5}, 1]$	00001	10000	10000	11000	01110	00111

(c)

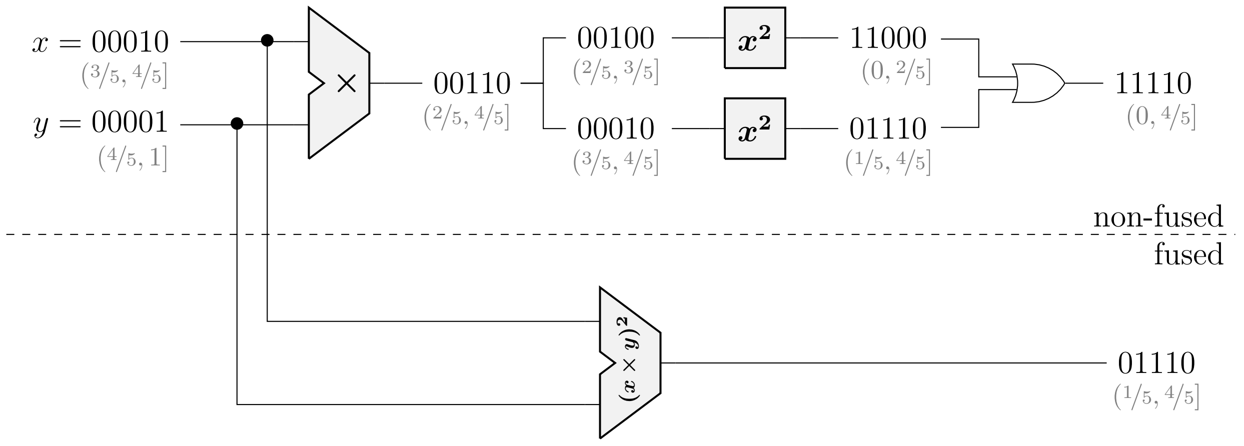


Figure 3.9.: Non-fused and fused SORN squared multiplication example for the 5-bit SORN datatype from table 3.8.

squared individually, followed by an OR operation to compute the final result $(0\,, 4/5]$. Since the overestimation from the multiplication is propagated, this result includes the interval $(0\,, 1/5]$, which would not be required to represent the true result $(144/625, 16/25]$ with the available SORN datatype.
Table 3.8c shows the SORN LUT for the fused operation $(x \times y)^2$ with the same 5 bit datatype. The fused version of the above described example is depicted in the lower half of figure 3.9. Both show that when using a fused SORN operation, the result is $(1/5, 4/5]$ and does not contain the unnecessary interval $(0\,, 1/5]$, therefore increasing the accuracy. This behavior can be observed for in total 4 of the possible 25 one-hot input cases for the given operation and datatype.
In the following, this approach of fusing single- and two-input SORN operations is evaluated for a set of basic operations, as well as for the more complex hypot and swish functions, and two polynomials of second and third order. As a measure for accuracy, the interval or output bitwidth is used, which means the number of consecutive 1_2 in the SORN result. In addition, all evaluations also take into account the hardware results of the synthesized non-fused and fused designs, using 28 nm CMOS SOI technology from STM.

Basic Operations In order to evaluate this fused approach on a quantitative basis, the twelve functions depicted in table 3.9 are implemented and evaluated as both non-fused and fused version for the *lin13|¹* SORN datatype from table 3.6. Each operation is composed of either two-input addition (1-6) or multiplication (7-12), combined with one of the single-input functions square, square root or the exponential function. An even index represents single-input

Table 3.9.: Basic operations evaluated for two-input fused and non-fused implementations [9].

1	**2**	**3**	**4**	**5**	**6**
$(x+y)^2$	x^2+y	$\sqrt{x+y}$	$\sqrt{x}+y$	e^{x+y}	e^x+y
7	**8**	**9**	**10**	**11**	**12**
$(x \times y)^2$	$x^2 \times y$	$\sqrt{x \times y}$	$\sqrt{x} \times y$	$e^{x \times y}$	$e^x \times y$

followed by two-input operations, an odd index represents the opposite. The results of this evaluation are depicted in figure 3.10. The synthesis results are given as APT product, the accuracy as mean interval width of the output SORN. The mean is hereby obtained from the results of all possible 13^2 one-hot input combinations during RTL simulation. Note that for the square-root-based operations 3, 4, 9 and 10, only zero or positive input values were considered. The primary goal of introducing fused operations is to improve the output accuracy by reducing the interval width. Considering the results from figure 3.10, it can be observed that all evaluated fused designs show an accuracy improvement over the respective non-fused design, except for the operation with index 3, where the accuracy is equal for both designs. The highest reductions of the interval width are achieved for the operations with indices 6 (22%), 9 (24%) and 12 (28%), respectively. For all operations that include addition (indices 1-6), the mean improvement is 6.2%, for the multiplication operations (indices 7-12) the mean improvement is 11.8%. Taking into account those operations where a two-input operation is followed by a single-input (odd indices), the mean improvement is 7.1%. For the opposite operation order with even indices, the mean improvement is 10.9%.
Considering the hardware results, even higher improvements can be observed. The highest are achieved for the operations with indices 7 (88%) and 11 (77%), respectively. However, there are also two cases where the APT product is larger for the fused implementation, compared to the non-fused version. In particular, for the operations with indices 6 and 12, the fused APT

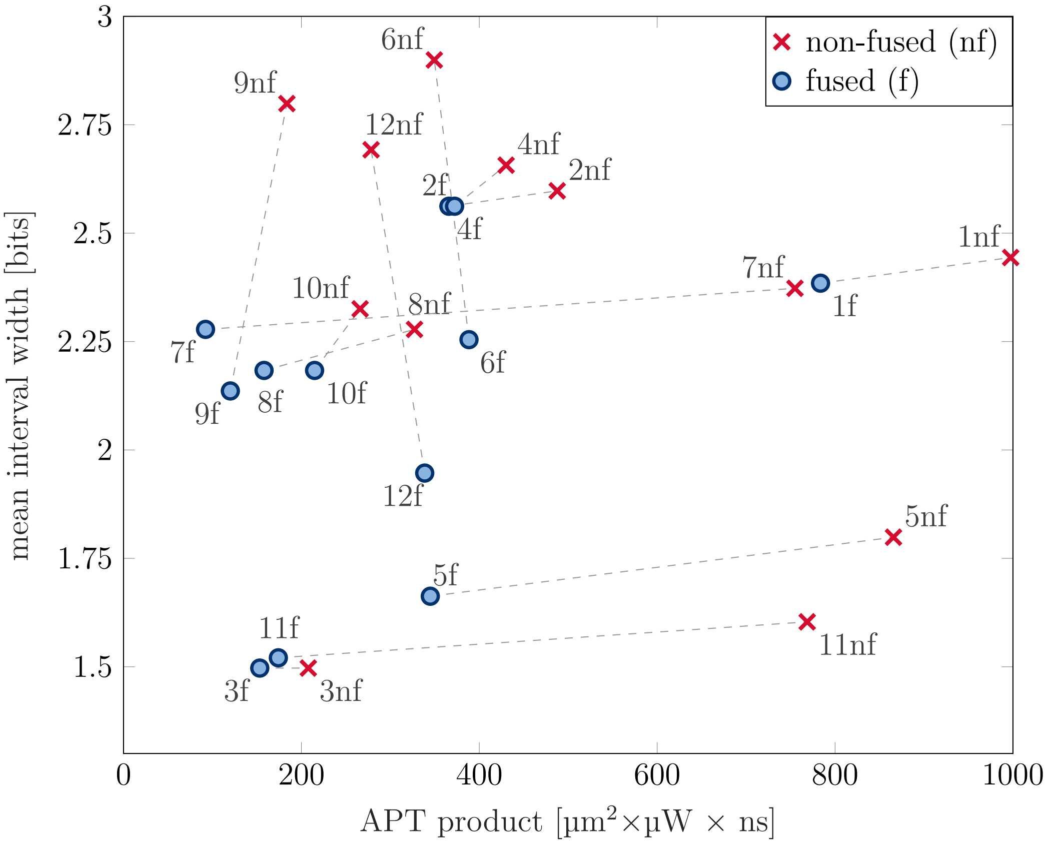


Figure 3.10.: APT product vs. mean interval width for the basic operations from Tab. 3.9, implemented as non-fused (nf) and fused (f) version for the *lin13*$|^1$ datatype from table 3.6 [9].

product is larger by 11% and 21%, respectively. Yet, for all operations that include addition (indices 1-6), still a mean APT product improvement of 22.5% can be achieved for the fused designs. For the multiplication operations (indices 7-12) the improvement is even higher with 41.7%. Comparing again two-input operations followed by a single-input (odd indices) with the opposite order (even indices), the mean improvements are 51.2% and 13%, respectively. As a general conclusion, it can be stated that all fused designs achieve an improved performance for at least one of the evaluation measures, in most of the cases even for both. In addition, the multiplication-based operations show higher improvements than the addition-based ones for both evaluation measures.

Hypot and Swish Function In order to consider functions that combine more than two arithmetic operations, the two-input hypot and the single-input swish functions

$$\text{hypot}(x, y) = \sqrt{x^2 + y^2} \tag{3.12}$$

$$\text{swish}(x) = \frac{x}{1 + e^{-x}} \tag{3.13}$$

are evaluated in a similar manner as the basic operations before. The hypot function is widely used in signal processing applications like the computation of 2-dimensional Euclidean distances or in image processing [SB11], and is even included in the 2008 version of the IEEE-754 standard [IEEE08]. The swish function is used as activation function in deep neural networks [RZL17]. Both functions are composed of one two-input and three single-input operations, which are fused into one block, as depicted in figure 3.11.

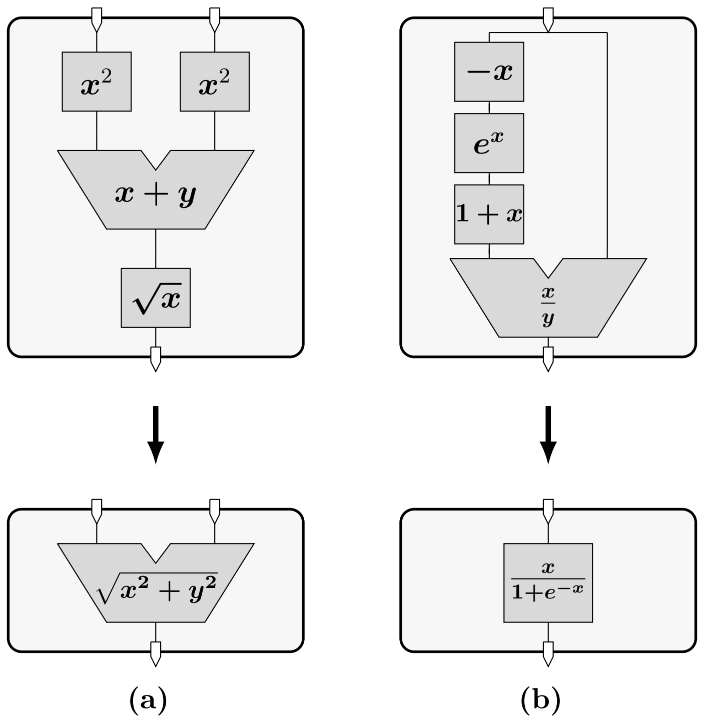


Figure 3.11.: Non-fused (top) and fused (bottom) block diagrams of the RTL designs for the (a) hypot and (b) swish function [9].

Table 3.10.: Mean output interval width of the SORN result and 28 nm CMOS synthesis results for fused and non-fused two-input hypot and swish functions [9].

			hypot(x,y)		**swish(x)**	
			non-f.	fused	non-f.	fused
mean int. width		[bit]	2.610	1.544	4.077	1.923
CMOS	**area**	[µm^2]	36.067	26.765	22.358	22.522
CMOS	**power**	[µW]	11.168	6.900	4.985	4.510
CMOS	**timing**	[ns]	0.327	0.309	0.433	0.247
CMOS	**APT product**		131.714	57.066	48.260	25.089

Similar to the basic operations, the two functions are evaluated for both accuracy and hardware performance. For the accuracy the mean output interval width is obtained from RTL simulations for all possible one-hot inputs for the applied *lin13|1* datatype, which are 13^2 different inputs for the hypot and 13 for the swish function. The results are given in table 3.10, together with the hardware measures, obtained from a 28 nm CMOS synthesis. Similar to the basic operation evaluation, it can be observed that the fused implementations outperform their non-fused counterparts in both accuracy and hardware performance. In detail, the fused hypot function shows an accuracy and APT product improvement of 41% and 57%, respectively. For the fused swish function, the improvements are 53% for the accuracy and 48% for the APT product.

Polynomials Polynomial functions are another interesting use case for fused SORN arithmetic since they are often used as benchmarks for performance evaluations and state-of-the-art comparisons for different computing environments, for example scientific extensions for floating point [HHKR12], or type-I and type-III unums [Gus15, Gus17]. With SORNs, solving polynomial functions, i.e. determining their roots, differs from standard algorithms with classical number formats because of the limited precision of SORN environments. SORNs can be used in a preprocessing step to determine possible solution candidates or start values for optimization algorithms, which are subsequently evaluated in another number format. This approach is also used for the SORN MIMO symbol detection which will be discussed in chapter 4. Essentially, for determining the roots of a polynomial using SORN arithmetic, the binary SORN output is calculated for every possible one-hot input value from the chosen datatype. Those input values leading to a result containing the zero bit set to 1_2 are considered as solution candidates.
In the following two different polynomial benchmarks with one variable and orders 2 and 3 are considered:

$$\begin{aligned} p_2(x) &= 2\,x^2 + 0.5\,x - 0.25 && \text{(roots: } -0.5,\ 0.25) \\ p_3(x) &= x^3 - 1.5\,x^2 + 0.75\,x - 0.125 && \text{(root: } 0.5) \end{aligned} \tag{3.14}$$

The polynomials are implemented as both non-fused and fused design for the *lin19|2* datatype from table 3.6. Table 3.11 shows the mean output interval width and number of solution candidates after SORN processing for both polynomials from Eq. (3.14), as well as their

Table 3.11.: Mean output interval width of the SORN result, number of solution candidates, and 28 nm CMOS synthesis results for second and third order polynomials.

			$p_2(x)$		$p_3(x)$	
			non-f.	fused	non-f.	fused
mean int. width [bit]			4.684	3.158	8.211	5.579
solution candidates			5	3	11	7
CMOS	**area**	[µm^2]	66.749	63.158	296.534	239.578
CMOS	**power**	[µW]	9.315	7.598	38.150	25.465
CMOS	**timing**	[ns]	0.397	0.387	0.786	0.737
CMOS	**APT product**		246.842	185.711	8891.839	4496.329

respective 28 nm CMOS synthesis results. For the second order polynomial $p_2(x)$ the output interval width is reduced by 33%, the number of solution candidates improves from 5 to 3, and the APT product is reduced by 25%. For the third order polynomial $p_3(x)$ the output interval width reduction is by 32%, the number of solution candidates improves from 11 to 7, and the APT product is almost halved (49% reduction).

3.5.2. Three-Input Fused Operations

The results from the previous section show that fused SORN arithmetic can improve both the accuracy and the hardware performance for single- and two-input SORN operations. Consequently, this approach is to be evaluated for three-input operations as well. The most widely used fused operation in digital signal processing with standard formats is the three-input fused multiply-add (FMA) operation $(x \times y) + z$. Figure 3.12 shows an example for this

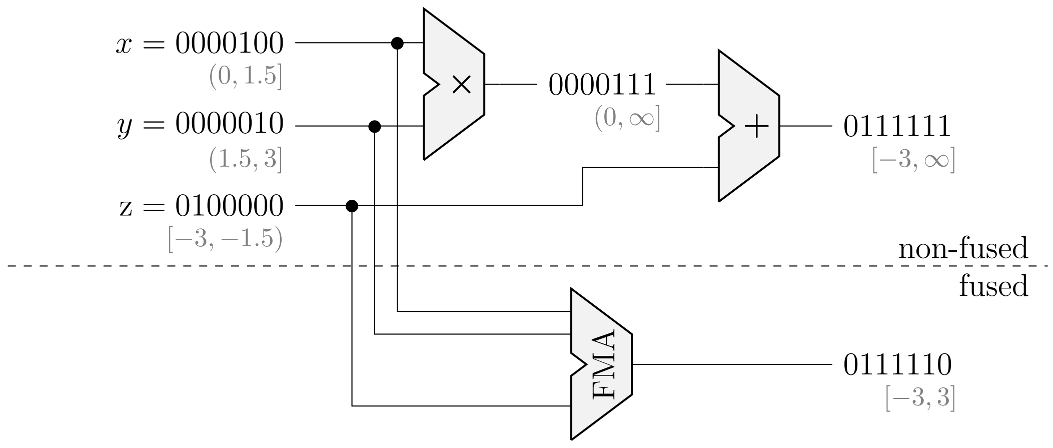


Figure 3.12.: Non-fused and fused SORN multiply-add example for the *lin7*$|^3$ datatype from table 3.6 [10].

operation carried out with SORN arithmetic using the *lin7*$|^3$ datatype from table 3.6 for both non-fused and fused implementations with the following inputs:

$$\begin{aligned} x &= 0000100_2 = (0, 1.5] \\ y &= 0000010_2 = (1.5, 3] \\ z &= 0100000_2 = [-3, -1.5) \end{aligned} \tag{3.15}$$

The upper half of figure 3.12 shows the non-fused case where the intermediate true multiplication result $(0, 4.5]$ is quantized to the SORN interval $(0, \infty]$, before z is added, leading to the final result $[-3, \infty]$. The lower half of figure 3.12 shows that when the intermediate quantization can be avoided with the fused operation, the true output $[-3, 3]$ can be achieved, which corresponds to an improvement of the output accuracy.
In the following, the three-input fused SORN approach is evaluated for three different arithmetic operations, namely three-input addition $x + y + z$, multiplication $x \times y \times z$, and multiply-add (MA) $(x \times y) + z$, each implemented as non-fused and fused version. The evaluation includes designs for in total twelve different SORN datatypes per operation. In detail, four 7, 9 and 11 bit datatypes from table 3.6 are used, each with either a linear or logarithmic value distribution and with small or large lattice values. The evaluation covers accuracy and hardware results individually, as well as in a combined analysis.

Accuracy Results The accuracy results in terms of the mean output interval width per operation and datatype are given in figure 3.13. The results are obtained from RTL simulations

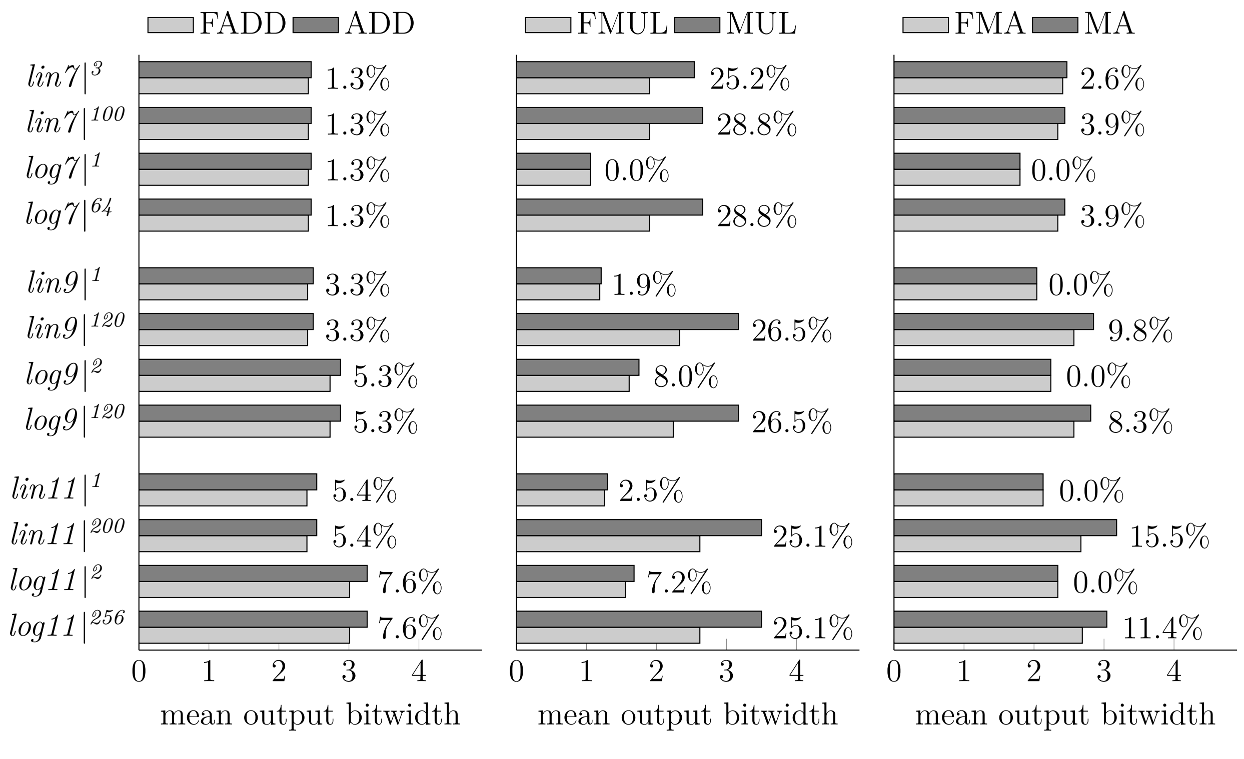


Figure 3.13.: Mean output bitwidths for non-fused and fused three-input addition, multiplication and multiply-add, evaluated for one-hot input values. The output bitwidth reduction for fused operations per datatype is given in % [10].

for all possible one-hot input combinations without infinity intervals, leading to $(w_s - 2)^3$ possible input combinations per datatype. The graph also includes the reduction of the interval width in % for every fused designs, compared to its non-fused equivalent.
The highest improvements are achieved for multiplication with up to 28.8% reduction, followed by multiply-add with up to 15.5%. The improvement for the fused three-input addition with up to 7.6% is rather moderate. Except for multiplication, the improvements scale with the bitwidth of the SORN datatype. Another interesting observation is that for all operations with at least one multiplication, the improvement for datatypes with larger lattice values is much higher. One multiplication and five multiply-add cases even show no improvement, all for datatypes with small lattice values up to 1_{10} or 2_{10}. The reason is that multiplication with values $\leq 1_{10}$ stays in that range and does not lead to infinity intervals, which is why the small-value datatypes show a lower interval width in general. Since there are less infinity interval cases that can be avoided with fused operations, the improvement is also smaller.

Hardware Results Figure 3.14 shows the area utilization and power consumption for a 28 nm CMOS synthesis of all fused and non-fused designs for a target frequency of $f = 1$ GHz.

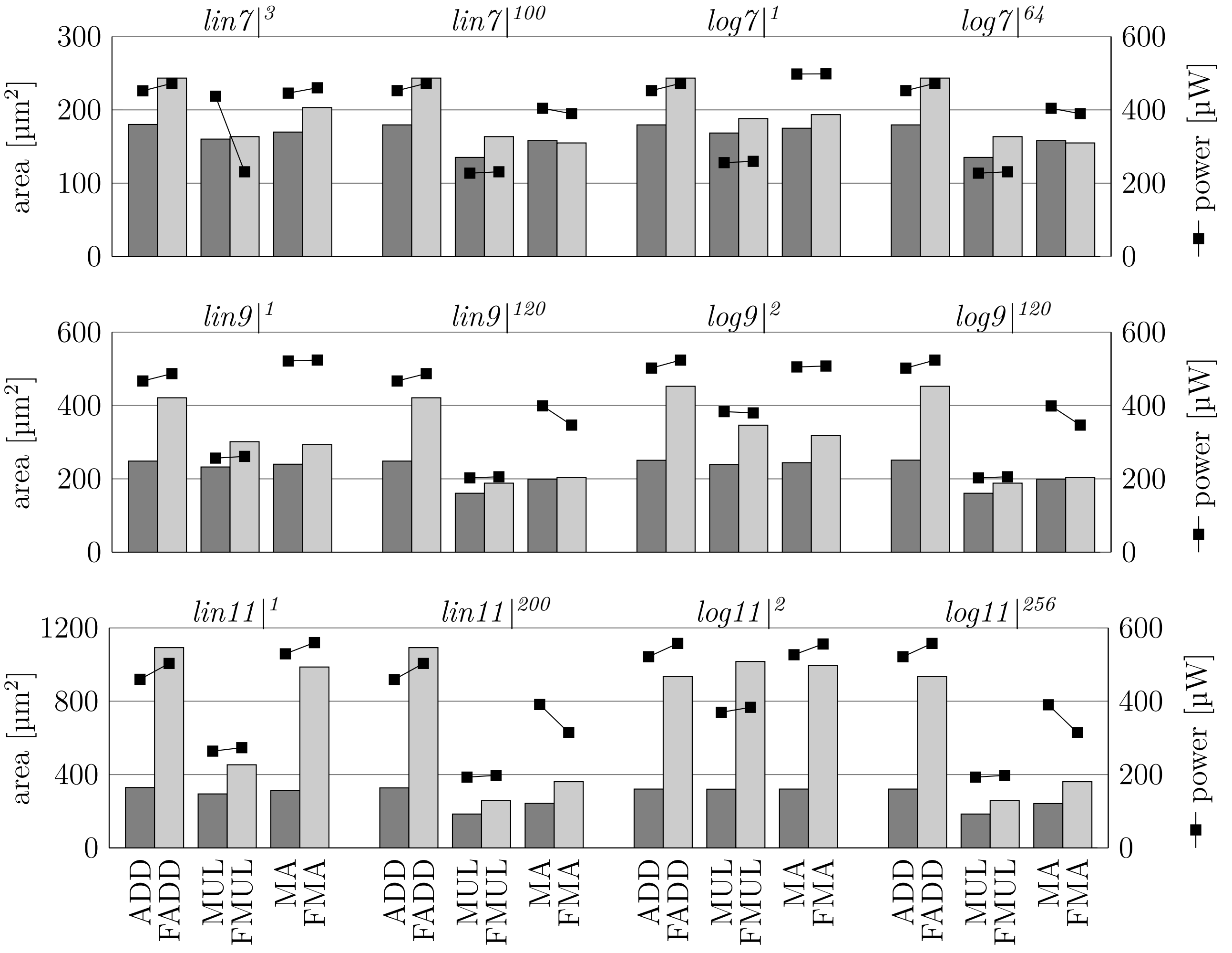


Figure 3.14.: Synthesis results for non-fused and fused three-input addition, multiplication and multiply-add, for 28 nm CMOS with $f = 1$ GHz [10].

The path delay is not shown because the fused path delay is equal to the non-fused one for 2/3 of the evaluated cases. For the remaining 1/3, the fused operations show slightly higher delays. All delay results are included in the APT product which is used for the joint accuracy and hardware evaluation in the next paragraph.
The fused three-input SORN LUTs show a higher area utilization than the non-fused combination of two two-dimensional SORN LUTs. This is true for every evaluated operation and SORN datatype except for the $lin7|^{100}$ and $log7|^{64}$ datatypes in combination with the multiply-add operation, where a small area reduction for the fused designs can be observed. In particular, for fused addition, the area increases between 35% and 230%, for fused multiplication between 2% and 217%. For FMA the area utilization reaches from a decrease of 2% to an increase of 215%. The mean area increase for the fused designs over the 12 different datatypes is 101% for addition, 43% for multiplication, and 51% for multiply-add.
Considering the power consumption, more fused designs show an improvement over their non-fused equivalents. For all datatypes with large lattice values in combination with the multiply-add operation a reduction of the power consumption can be achieved, which is 12% on average. Additionally, datatypes $lin7|^{3}$ and $log9|^{2}$ with the multiplication operation achieve a reduction of 47% and 1%, respectively.
All implemented designs were also synthesized for an FPGA platform. Details and results can be found in the appendix B.2 and figure B.1. This evaluation mostly confirms the results discussed above, especially concerning the higher area utilization for the fused designs.

Accuracy vs. Hardware Results In contrast to the two-input fused operations evaluated in section 3.5.1, where both accuracy and hardware could be improved, the above evaluated results for three-input fused operations show that while the output interval accuracy is also mostly improved by the fused designs, the hardware complexity is mostly increased. In order to visualize and rate the trade-off between both measures, a joint evaluation is discussed in the following. According to [10], an accuracy ratio r_{acc} is introduced, which compares the mean output width of fused and non-fused designs for every operation and datatype, based on the presented simulation results:

$$r_{acc} = \frac{\text{fused mean out width}}{\text{non-fused mean out width}} \tag{3.16}$$

In a similar manner, the APT product ratio r_{APT} compares the hardware performance of fused vs. non-fused, utilizing the combined APT results given in [$\mu m^2 \times \mu W \times ns$] from the presented CMOS synthesis:

$$r_{APT} = \frac{\text{fused APT } [\mu m^2 \times \mu W \times ns]}{\text{non-fused APT } [\mu m^2 \times \mu W \times ns]} \tag{3.17}$$

Figure 3.15 plots both ratios against each other. An improvement for the fused over the non-fused design is indicated by a ratio < 1. Cases where the percentage degradation of one measure is compensated by the other are represented by balanced ratios $r_{acc} + r_{APT} = 2$. Some designs showing an $r_{APT} > 2$ are depicted in the small subplot which is simply an extension of the main plot window.
Roughly 3/4 of the evaluated cases neither achieve an $r_{APT} < 1$ nor balance ratios. However, for four FMA designs with 7 and 9 bit, as well as for one fused multiplication with 7 bit,

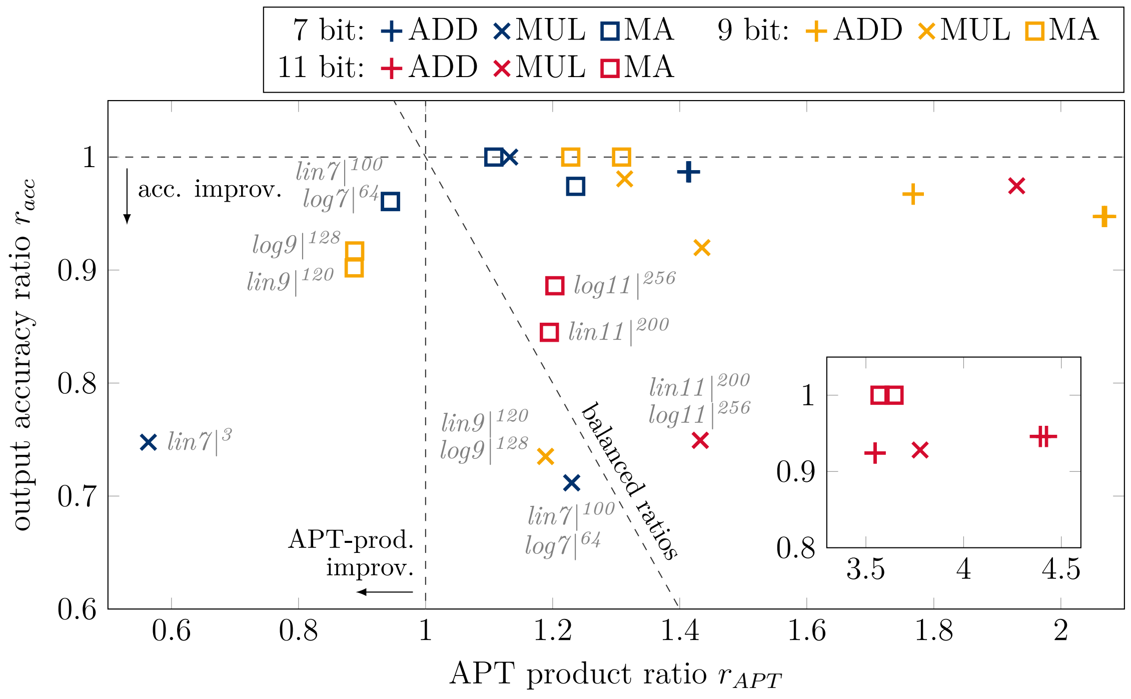


Figure 3.15.: Output accuracy ratio r_{acc} vs. APT product ratio r_{APT} for fused three-input addition, multiplication and multiply-add [10]. A ratio < 1 indicates an improvement for the fused design over the non-fused. Balanced ratios are achieved with $r_{acc} + r_{APT} = 2$.

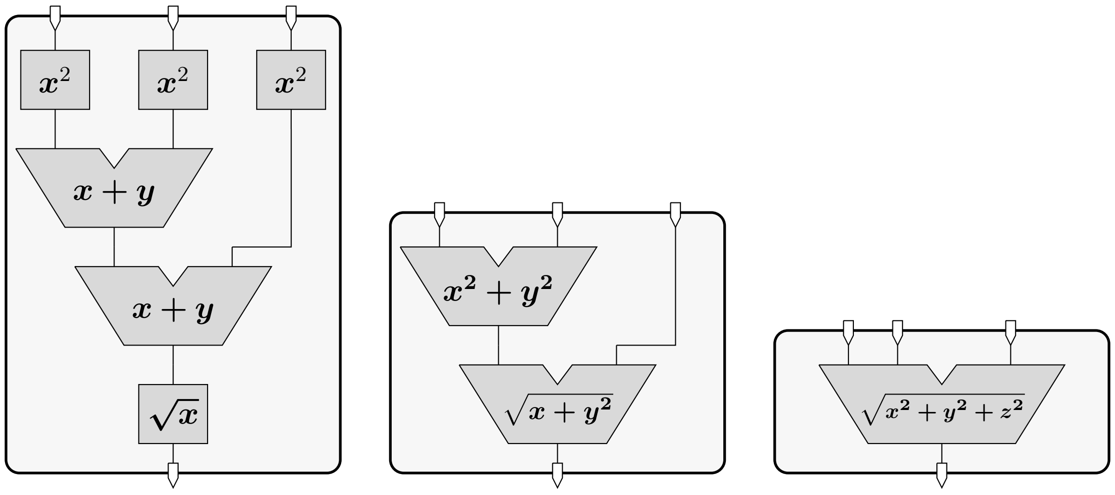
an improvement for both ratios can be achieved. Further, two 7 bit and two 9 bit fused multiplication designs achieve results better than balanced ratios. In addition, for the datatypes *lin11*$|^{200}$ and *log11*$|^{256}$ the designs for multiplication and multiply-add are close to balanced ratios.
In general, it can be concluded that the multiplication and multiply-add operations with 7 and 9 bit datatypes show a high improvement potential for using three-input fused operations, whereas addition and the 11 bit datatypes are mostly not worth considering.

3.5.3. Two- vs. Three-Input Fused Hypot Function

The two previous sections 3.5.1 and 3.5.2 showed separate evaluations for fused operations with up to either two or three inputs. While the results for two inputs showed mainly improved hardware measures, designs with three inputs showed both better and worse results. In order to compare both approaches, this section evaluates on the previously introduced hypot function, but with three inputs

$$\text{hypot}(x, y, z) = \sqrt{x^2 + y^2 + z^2} \tag{3.18}$$

and compares the implementations of a non-fused design, one which fuses operations up to two inputs, and a design that fused the complete function into a single block with three inputs. These three different designs are shown in figure 3.16. All three designs were implemented for the twelve SORN datatypes utilized in the previous section 3.5.2 and synthesized for 28 nm CMOS technology with $f = 1\,\text{GHz}$. The hardware results are again summarized as the APT



(a) non-fused **(b)** two-input fused **(c)** three-input fused

Figure 3.16.: Block diagrams for the three-input hypot function as non-fused, two-input fused and three-input fused designs [10].

product ratio r_{APT}, introduced in equation (3.17). The accuracy is also obtained through RTL simulations as described in section 3.5.2 and summarized as accuracy ratio r_{acc}, introduced in equation (3.16). The ratios for the two-input and the three-input fused implementations both are computed against the respective non-fused design. The results of this evaluation are given in figure 3.17. The two subplots split the r_{APT}-axis into parts with different resolutions.

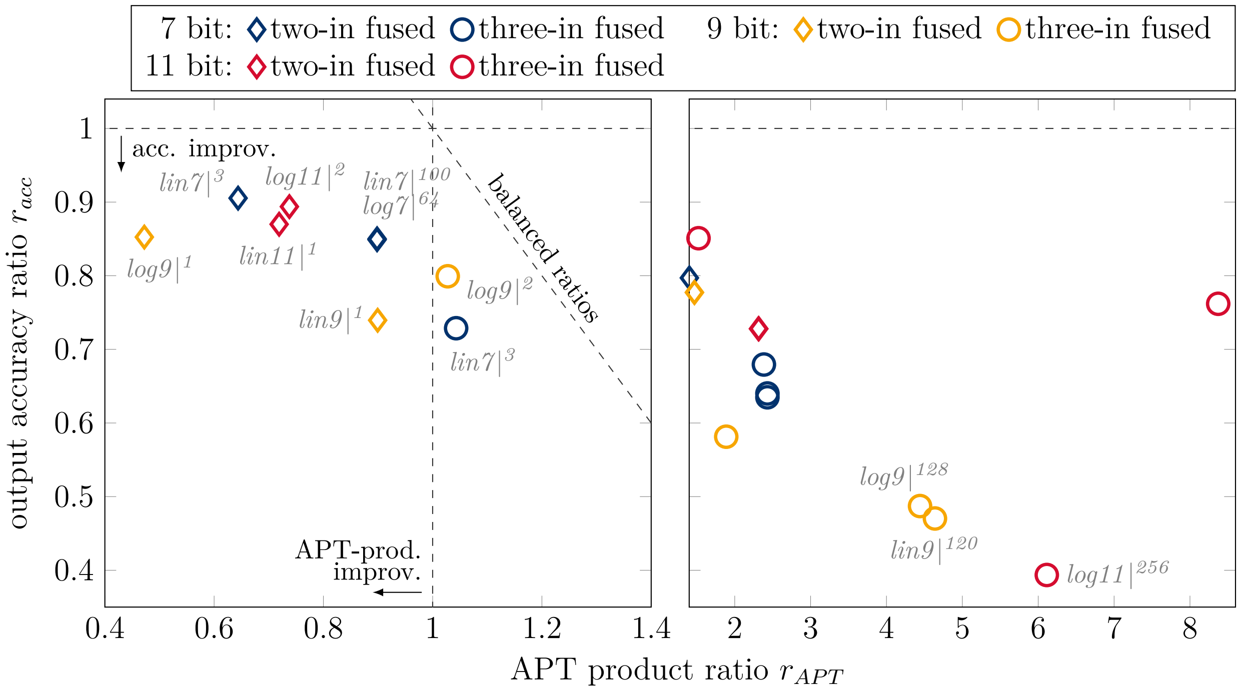


Figure 3.17.: Output accuracy ratio r_{acc} vs. APT product ratio r_{APT} for two-input fused and three-input fused hypot function [10]. A ratio < 1 indicates an improvement for the fused design over the non-fused. Balanced ratios are achieved with $r_{acc} + r_{APT} = 2$.

The main result from this evaluation is that only two-input fused designs achieve an $r_{APT} < 1$. Only two of the three-input designs achieve a result better than balanced ratios with an $r_{APT} \approx 1$. In contrast to the results for fused three-input multiplication and FMA from the previous section, for the three-input hypot function only datatypes with small lattice values show improvements for both ratios. The $lin7|^{100}$ datatype is the only exception for this observation. Another interesting observation is that some of the three-input designs show an accuracy improvement of up to 60%, yet accompanied by a large $r_{APT} >> 1$.

3.6. Summary

This chapter introduced the basic concept of the type-II unum format and the derived Sets Of Real Numbers (SORN) representation, according to its proposal from [Gus16]. After extending this original, unum-based SORN format by a formal mathematical description, and discussing possible drawbacks of this original version, alternative half-open and custom SORN datatypes were proposed. In order to facilitate the implementation of SORN datapaths with different datatypes and to provide a basis for a design space exploration, an automated design-flow for RTL implementation through the open-source SORN Hardware Generator tool was presented. With the help of this tool, the hardware complexity of basic arithmetic SORN components was evaluated for different SORN datatypes and against standard integer/fixed point operations. The results showed that SORNs easily outperform integer designs of same bitwidth. When comparing the proposed half-open SORN datatypes against the original unum-based ones for the same lattice values, the half-open designs showed major improvements. Finally, the concept of fused operations from standard floating point arithmetic was adapted for SORNs and evaluated for single-, two- and three-input operations. Evaluations on both accuracy and hardware performance showed that fused SORN operations for up to two inputs almost always improve both measures at once. For three-input fused SORN operations, the accuracy is also mostly improved, while the hardware improvement is not always achieved and highly datatype and operation depended.

4 SORN Applications

When categorizing the Sets Of Real Numbers format among all the other existing approaches for representing real numbers in digital systems, which are discussed in chapter 2, SORNs can be considered as an application specific rather than a general purpose format. The main reason is the utilized interval arithmetic, where result values have to be interpreted differently than in standard, single-valued arithmetic. This complicates a head-to-head comparison for the performance of SORNs versus legacy formats like fixed or floating point. In section 3.4 it is shown that SORNs can achieve a better hardware performance compared to standard integer or fixed point formats. However, SORNs still have to prove that they can compete with the algorithmic performance of these standard formats as well. While in the previous chapter 3 the SORN format and the general arithmetic concept with the corresponding datatypes are presented, this chapter therefore deals with suitable applications for SORNs. These use cases facilitate evaluations and comparisons with standard formats, taking into account not only the hardware but also the algorithmic performance.
Another aspect is the evaluation of different SORN datatypes, which also has been carried out solely for hardware measures in sections 3.4 and 3.5 so far. Here as well, an evaluation with respect to the algorithmic performance is facilitated through the following applications.

In [Gus16] a possible use case for the SORN format is given by a twelve dimensional non-linear system of equations, taken from a robotics application. With standard formats, such a system of equations is usually solved with an iterative optimization algorithm relying on a suitable starting value. With a fast SORN implementation, it becomes feasible to compute the entire solution space by means of an exhaustive search. Due to the low precision of SORNs, this does not lead to a single solution for the system of equations, but it reduces the amount of possible solutions and provides a space for the starting value of a subsequent optimization algorithm executed in a standard format. In this scenario, the SORN computation can be seen as a preprocessing step to reduce the overall runtime of the optimization.
In the following, different use cases for SORNs will be presented, where the implemented algorithms follow two different general approaches:

1. SORNs are used in an application where precision and output accuracy are not critical, for example when the results are used for a threshold-based decision.

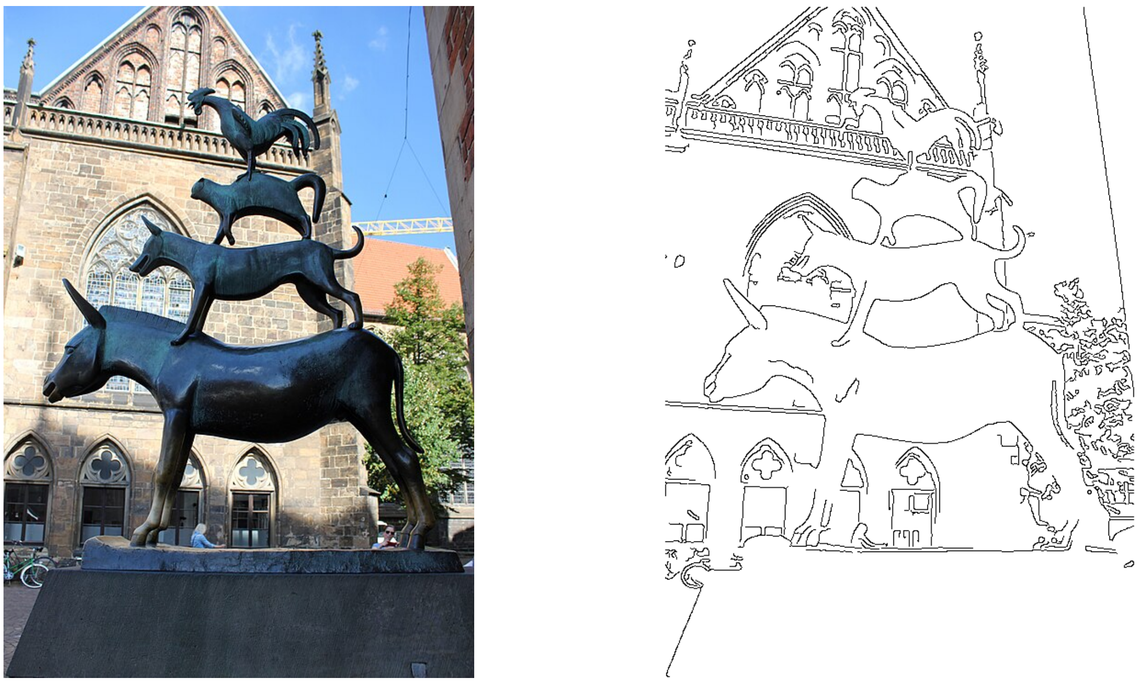
2. SORNs are used in a preprocessing step to reduce the solution space of an optimization problem as shown in the example from [Gus16].

These two general approaches are applied in two different application areas: For the first approach, SORNs are used in image processing, in particular for an implementation of the

Sobel algorithm used for edge detection. The edge detection problem and the SORN realization are presented in section 4.1. For the second approach, SORNs are used in a wireless MIMO communication system, which is introduced in section 4.2, together with a SORN preprocessor for MIMO symbol detection. This preprocessor is used within a BPSK detector presented in section 4.3, and for improving a state-of-the-art sphere decoder (SD), discussed in section 4.4.

4.1. SORN Edge Detection for Image Processing

The detection of edges within images is a central problem statement in the field of image processing. An edge, also called contour, can be defined as a region of an image, where discontinuities or distinct changes in color or brightness of two or more adjacent pixels can be detected [SB11, AMFM11]. These edges can then be used to separate the image into different segments or to identify certain objects. Use cases of edge detection are, for example, fingerprint recognition [CWHY08], the classification of clouds via satellite images [DT13] or road lane and object detection for autonomous driving [BGLP15, YYW21].
An edge detection algorithm can consist of multiple processing steps which target different aspects and improvements of the edge result. One of the most widely used approaches is the Canny edge detection algorithm [Can86], which applies some of these different processing steps. The edge detection result for applying the Canny algorithm on an example image is shown in figure 4.1.



(a) Original Image **(b)** Canny Edge Detection

Figure 4.1.: Result of an edge detection example using the canny algorithm (b) on the original image (a) [Ima]. The original image (a) is licensed under the *Creative Commons Attribution-Share Alike 3.0 Unported* license [Lic] and is depicted without changes in (a).

The Canny edge detection algorithm computes the following steps [Can86, SB11, Yux23]:

1. Gaussian smoothing to reduce the effect of noise within the image.
2. Gradient magnitude/intensity $G(x, y)$ and direction $\theta(x, y)$ calculation per pixel (x, y), by use of the Sobel operator.
3. Non-maximum suppression to thin the calculated edges by removing those edges without a maximum intensity in edge direction.
4. Double thresholding to identify weak and strong edges and delete those weak edges not connected to a strong edge, in order to form complete edges and remove single edge artifacts.

The Canny algorithm can be seen as an advanced approach with the pre- and post-processing steps 1 and 3, while the actual edge detection is performed by the Sobel operator in step 2 and the threshold decision in step 4, which, in a more simple approach, can also be implemented with a single threshold. This Sobel operator with single thresholding is introduced in the subsequent section 4.1.1, followed by a SORN implementation in section 4.1.2, as well as an algorithmic and hardware evaluation in sections 4.1.3 and 4.1.4, respectively.

4.1.1. Sobel Operator

The Sobel operator or Sobel filter, also called Sobel-Feldman operator [Sob14], basically consists of two 3×3 filter kernel matrices used to approximate the first derivative of the image gradient in x and y direction, respectively [SB11]. The input of this process is a grayscale version of the original image $\mathbf{A} \in \mathbb{N}_0^{N_x \times N_y}$ with N_x and N_y as the number of pixels in horizontal and vertical direction. The pixels of this grayscale image are typically stored as 8 bit integer values $A(x, y) \in \{0, \dots, 255\}$. The derivatives in x (horizontal) and y (vertical) direction, G_x and G_y, are calculated per pixel (x, y) by performing a discrete convolution of the filter kernels $\boldsymbol{G}_{\text{Sobel},x}$ and $\boldsymbol{G}_{\text{Sobel},y}$ with a 3×3 slice of the input matrix that has the current pixel (x, y) as center element. The discrete convolution operation $*$ is defined as follows for 3×3 matrices [SB11]:

$$\begin{aligned} \boldsymbol{G} * \boldsymbol{A} &= \begin{bmatrix} G_{1,1} & G_{1,2} & G_{1,3} \\ G_{2,1} & G_{2,2} & G_{2,3} \\ G_{3,1} & G_{3,2} & G_{3,3} \end{bmatrix} * \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \\ &= \sum_{i=1}^{3} \sum_{j=1}^{3} G_{4-i,4-j} A_{i,j} \\ &= G_{3,3}A_{1,1} + G_{3,2}A_{1,2} + G_{3,1}A_{1,3} + G_{2,3}A_{2,1} + G_{2,2}A_{2,2} + G_{2,1}A_{2,3} + \\ &\quad G_{1,3}A_{3,1} + G_{1,2}A_{3,2} + G_{1,1}A_{3,3} \end{aligned} \tag{4.1}$$

The process of parsing the input matrix $\boldsymbol{A}$ with the filter kernels and calculating G_x and G_y for every pixel (x, y) is shown in figure 4.2 for a 5×5 input image. The 3×3 slice of the input matrix used for this process reads as

$$\mathbf{A}_{3\times3}(x, y) = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} = \begin{bmatrix} A(x-1, y+1) & A(x, y+1) & A(x+1, y+1) \\ A(x-1, y) & A(x, y) & A(x+1, y) \\ A(x-1, y-1) & A(x, y-1) & A(x+1, y-1) \end{bmatrix} \tag{4.2}$$

and is convolved with the filter kernels in order to obtain G_x and G_y:

$$G_x(x,y) = \underbrace{\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}}_{G_{\text{Sobel},x}} * \mathbf{A}_{3\times 3}(x,y) \qquad = A_{1,3} - A_{1,1} + A_{3,3} - A_{3,1} + 2A_{2,3} - 2A_{2,1} \tag{4.3}$$

$$G_y(x,y) = \underbrace{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}}_{G_{\text{Sobel},y}} * \mathbf{A}_{3\times 3}(x,y) \qquad = A_{3,1} - A_{1,1} + A_{3,3} - A_{1,3} + 2A_{3,2} - 2A_{1,2} \tag{4.4}$$

After computing the gradients in x and y direction, the 2-dimensional image gradient magnitude or intensity can be calculated:

$$G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \tag{4.5}$$

For a simple edge detector, this result can be compared to a pre-defined threshold T in order to determine whether the current pixel is an edge. The output of this process is a binary image indicating edges with a 1_2 and no edge pixels with a 0_2.

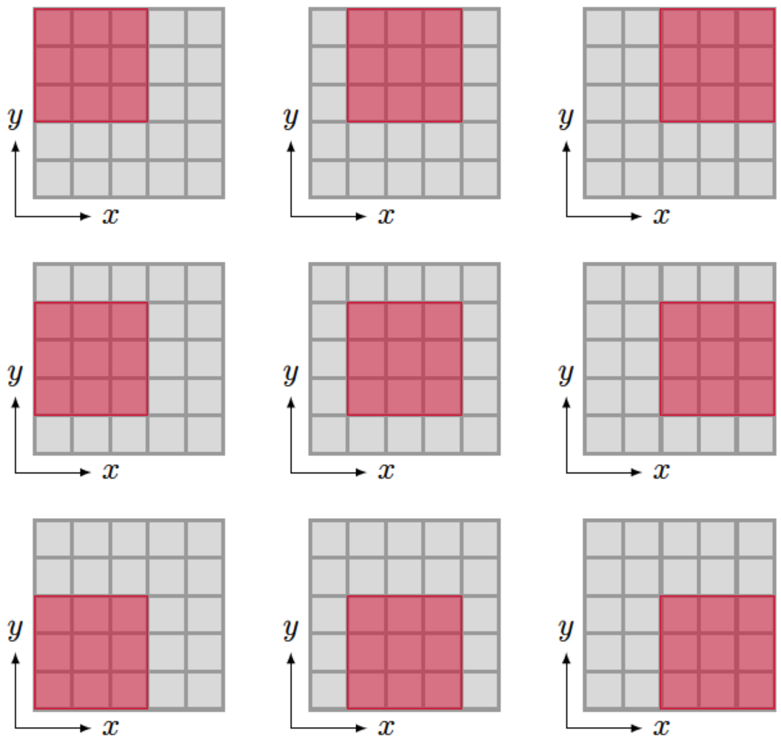


Figure 4.2.: Visualization of parsing a 5×5 matrix $\boldsymbol{A}$ with a 3×3 kernel $\boldsymbol{G}$.

For more complex approaches like the Canny detector, also the direction of the gradient is taken into account, which is computed based on the gradients in x and y direction:

$$\theta(x,y) = \text{atan2}(G_y(x,y), G_x(x,y)) = \begin{cases} \arctan\left(\frac{G_y}{G_x}\right) & \text{if } G_x > 0 \\ \arctan\left(\frac{G_y}{G_x}\right) + \pi & \text{if } G_x < 0 \text{ and } G_y \geq 0 \\ \arctan\left(\frac{G_y}{G_x}\right) - \pi & \text{if } G_x < 0 \text{ and } G_y < 0 \\ +\frac{\pi}{2} & \text{if } G_x = 0 \text{ and } G_y > 0 \\ -\frac{\pi}{2} & \text{if } G_x = 0 \text{ and } G_y < 0 \\ \text{undefined} & \text{if } G_x = 0 \text{ and } G_y = 0 \end{cases} \tag{4.6}$$

Comparable edge detection operators which mainly differ by the applied matrix kernels are the Prewitt, Roberts or Laplace operators [SB11].

4.1.2. SORN Implementation

The Sobel operator is usually implemented in standard integer or fixed point format, for example as hardware accelerator on an FPGA [BAPA16, KSH23]. Since the final step of the algorithm is a comparison of the gradient magnitude $G(x, y)$ with a predefined threshold T, resulting in a binary image, precision is no major concern, as long as the threshold decision can be applied with a sufficient accuracy. The bitwidth for implementing this algorithm is determined by the required dynamic range, as the input data of the grayscale image and most of the required arithmetic operations can be implemented with integers. When using SORN arithmetic instead, the dynamic range of the applied datatype can be aligned to the problem, not only by increasing the bitwidth, but also by decreasing the precision and maintaining a small bitwidth. The SORN datatypes applied in the following are $\textit{lin6}|^{250}_{nz,nn}$, $\textit{log10}|^{256}_{nn}$, $\textit{lin11}|^{250}_{nz,nn}$, $\textit{log15}|^{512}$ and $\textit{lin15}|^{300}$ from tables 3.6 and 3.7, all representing a high dynamic range with different resolutions and bitwidths. In order to improve the hardware performance of a Sobel edge detection, implementations of the Sobel operator for the different SORN datatypes are discussed and compared to an integer reference implementation. The implementation details are discussed in the following. The algorithmic performance and synthesis results are evaluated in sections 4.1.3 and 4.1.4, respectively.
Three different Sobel implementations are considered: a full integer reference design, a hybrid approach combining integer and SORN arithmetic, and a full SORN design. The block diagram for all designs is given in figure 4.3, indicating which number format is used for the respective subblock. The integer-to-SORN conversions required for the hybrid and full SORN designs are not shown. These modules are implemented as LUTs, realized with simple logic circuits.

Integer Reference Design The inputs of the reference design are the pixel values of the input image $\boldsymbol{A}_{3\times3}$, formatted as 8 bit unsigned integer values. These inputs are converted to a signed representation before performing the convolution with additions and subtractions, according to equations (4.3) and (4.4). The outputs of this convolution G_x and G_y are 10 bit signed integers. In order to compute the 2-dimensional image gradient magnitude G, the intermediate results G_x and G_y are squared and added, resulting in a 20 bit unsigned integer

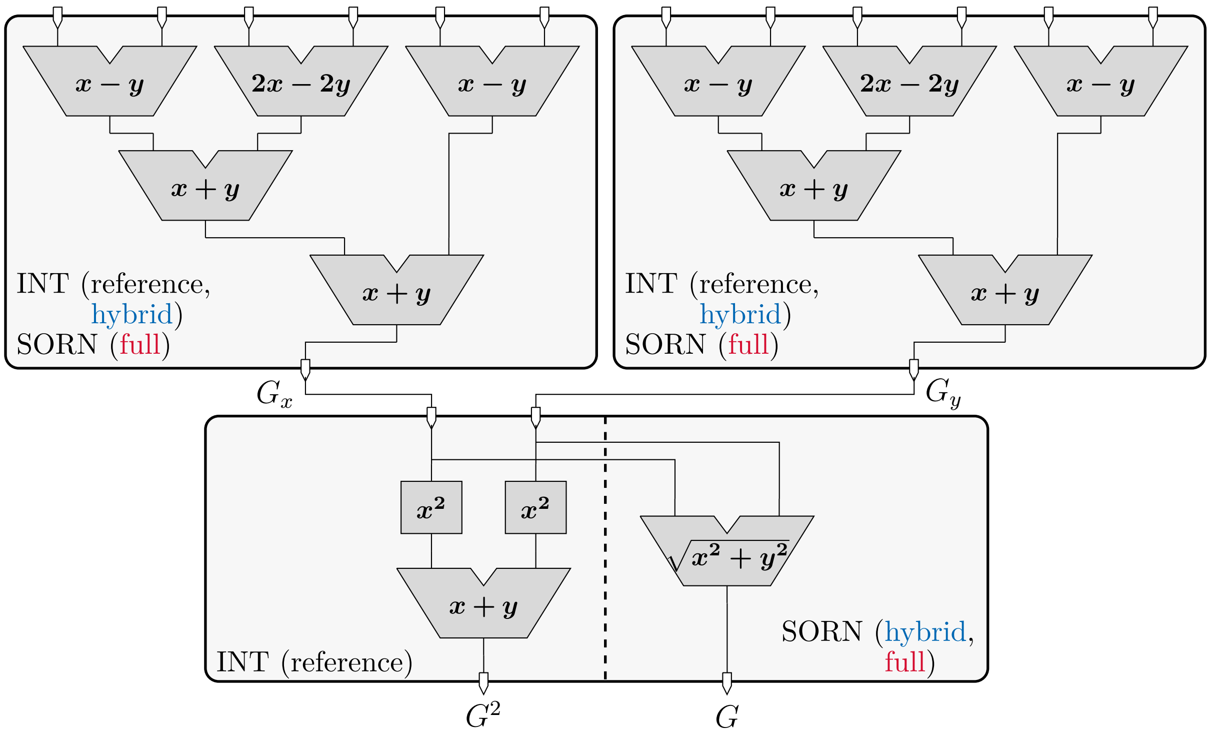


Figure 4.3.: Block diagram for the three different Sobel implementations: All integer as reference implementation, integer convolution with SORN hypot for the hybrid approach, and SORN convolution and hypot for the full SORN approach [8].

value. To reduce the complexity of the design, the required square root is omitted, and the squared gradient magnitude G^2 is compared to a squared threshold T^2 instead.

Hybrid Integer-SORN Design The hybrid design implements the convolution with integers in the same way as in the reference design, before G_x and G_y are converted to SORN representation in order to compute the 2-dimensional image gradient magnitude G. Since G_x and G_y are squared in the subsequent operation, solely positive values have to be taken into account. Therefore the absolute values are considered for SORN conversion, and the implemented datatypes $lin6|^{250}_{nz,nn}$, $log10|^{256}_{nn}$ and $lin11|^{250}_{nz,nn}$ all represent positive values only. The required hypot function for computing the 2-dimensional gradient magnitude is implemented with the SORN Hardware Generator from section 3.3 as fused SORN module, as introduced in section 3.5.1. Since the result G is in SORN representation, the threshold T has to be selected as one of the intervals from the implemented SORN datatype.

Full SORN Design For the full SORN design, the pixel values from the input image $\boldsymbol{A}_{3\times3}$ are converted to SORN representation, before both convolutions and the hypot function are carried out with SORN arithmetic. Since for the convolutions also negative values have to be taken into account, here the implemented SORN datatypes $log15|^{512}$ and $lin15|^{300}$ also cover negative values. As for the hybrid design, the threshold T has to be selected as one of the intervals from the implemented SORN datatype.

4.1.3. Algorithmic Performance

In figure 4.4a a grayscale test image showing a highway is depicted, which is used for the application of road lane detection for autonomous driving [Gat19b]. Figures 4.4b - 4.4d show the Sobel edge detection results for the given test image for the integer reference implementation, a hybrid and a full SORN implementation with SORN datatypes $lin11|^{250}_{nz,nn}$ and $lin15|^{300}$, respectively. The threshold T for any edge detection problem is not a fixed value, but has to be chosen depending on the required level of detail in the edge result. For the reference design, a threshold $T = 250$ was chosen, which leads to a reasonable detection of the road lane shape, whereas most of the visible cars and the shape of the surrounding landscape can not be detected in their entirety. In order to attempt a fair comparison, for the SORN designs comparable threshold intervals are chosen. For the datatype utilized in the hybrid SORN approach, the corresponding threshold interval is $T = (250, \infty]$. For the full SORN approach, however, thresholds near zero have to be chosen, and the resulting image has to be inverted afterwards, in order to achieve a comparable result. For the edge detection in figure 4.4d the threshold interval $T = (0, 50]$ was used for this purpose.
From a visual comparison it can be stated that all three approaches achieve to detect the road lanes properly, while some minor differences can be observed in the detection of the cars on the road, and for the shape of the landscape. A visual comparison, however, is not

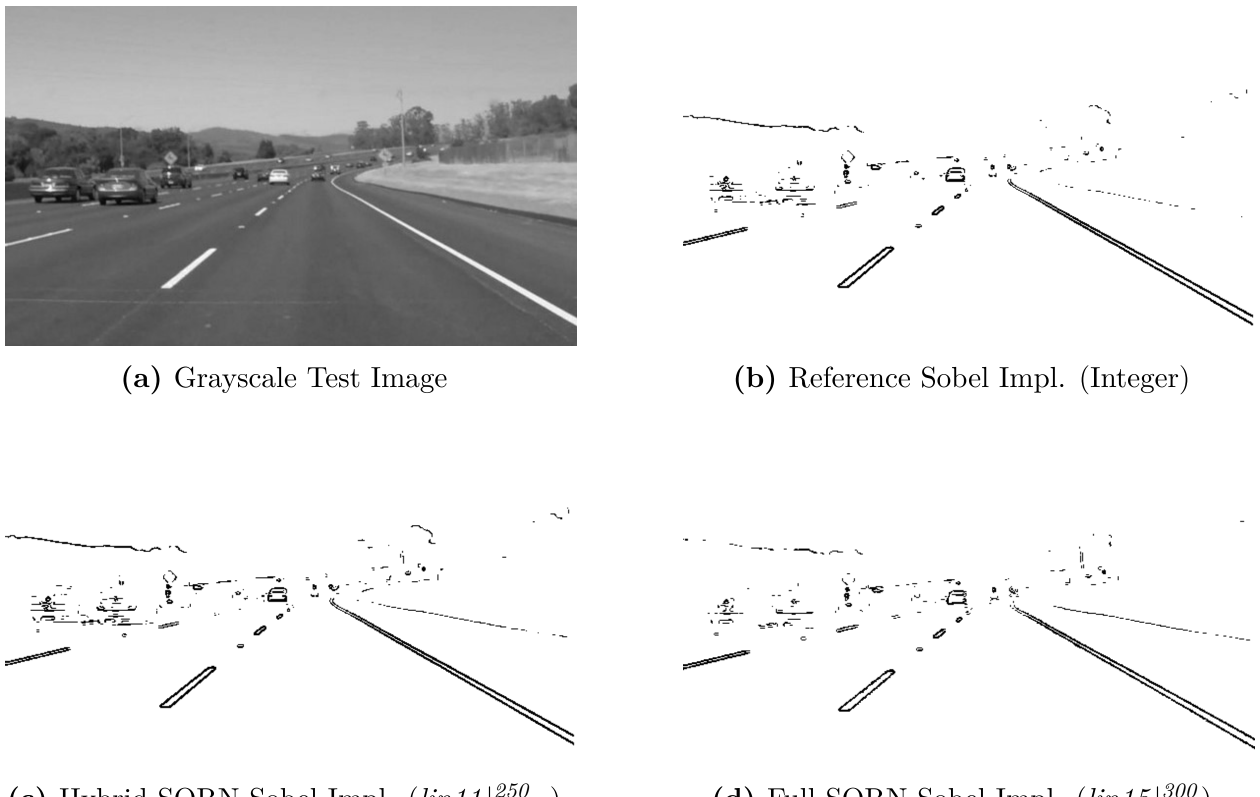

(a) Grayscale Test Image
(b) Reference Sobel Impl. (Integer)
(c) Hybrid SORN Sobel Impl. ($lin11|^{250}_{nz,nn}$)
(d) Full SORN Sobel Impl. ($lin15|^{300}$)

Figure 4.4.: Highway image (a) in grayscale [Gat19b, Gat19a], and with Sobel edge detection results from (b) an integer reference implementation with threshold $T = 250$, (c) a hybrid-SORN implementation with datatype $lin11|^{250}_{nz,nn}$ and threshold interval $T = (250, \infty]$, and (d) the negated result for a full-SORN implementation with datatype $lin15|^{300}$ and threshold interval $T = (0, 50]$ [8].

sufficient to evaluate the performance of the different approaches. Unfortunately, finding appropriate measures to evaluate the performance of different edge detection approaches is an open problem in the field of image processing. A comprehensive study on various error and performance metrics for edge detection from [LMDBB13] indicates that no general-purpose solutions exists to this problem. Therefore, in there following different evaluations on the algorithmic performance of the SORN Sobel edge detection designs are carried out.
Despite the visual comparison, the most intuitive evaluation approach is a numerical comparison of the hybrid and full SORN designs with the integer reference. For this purpose, the normalized absolute error (NAE) is introduced, which counts the number of pixels differing between the edge detection results of the reference and the respective SORN design, normalized by the total number of pixels:

$$\mathrm{NAE} = \frac{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} (\boldsymbol{E}_{\mathrm{INT}}(x,y) \neq \boldsymbol{E}_{\mathrm{SORN}}(x,y))}{N_x N_y} \tag{4.7}$$

$\boldsymbol{E}_{\mathrm{INT/SORN}} \in \{0,1\}^{N_x \times N_y}$ hereby are the respective edge detection results as binary images with dimension $N_x \times N_y$. Applied to the edge detections depicted in figure 4.4, this leads to the following results:

$$\mathrm{NAE}|_{\mathrm{hybridSORN},\mathit{lin11}|^{\mathit{250}}_{\mathit{nz,nn}}} = 0.0181 \tag{4.8}$$

$$\mathrm{NAE}|_{\mathrm{fullSORN},\mathit{lin15}|^{\mathit{300}}} = 0.0287 \tag{4.9}$$

These results show that the differences between reference and SORN designs are below 3%, which matches the results of the visual comparison, but it can not draw a conclusion whether the SORN results are better, worse or just different to the integer result. In addition, solely one test image was considered so far, which is not enough for a comprehensive evaluation. In the following, evaluations on the Berkeley Segmentation Data Set (BSDS) 500 from [AMFM11] are carried out, which is a set of images for the performance evaluation of contour detection and image segmentation algorithms, consisting of images of humans, animals, objects and landscapes.

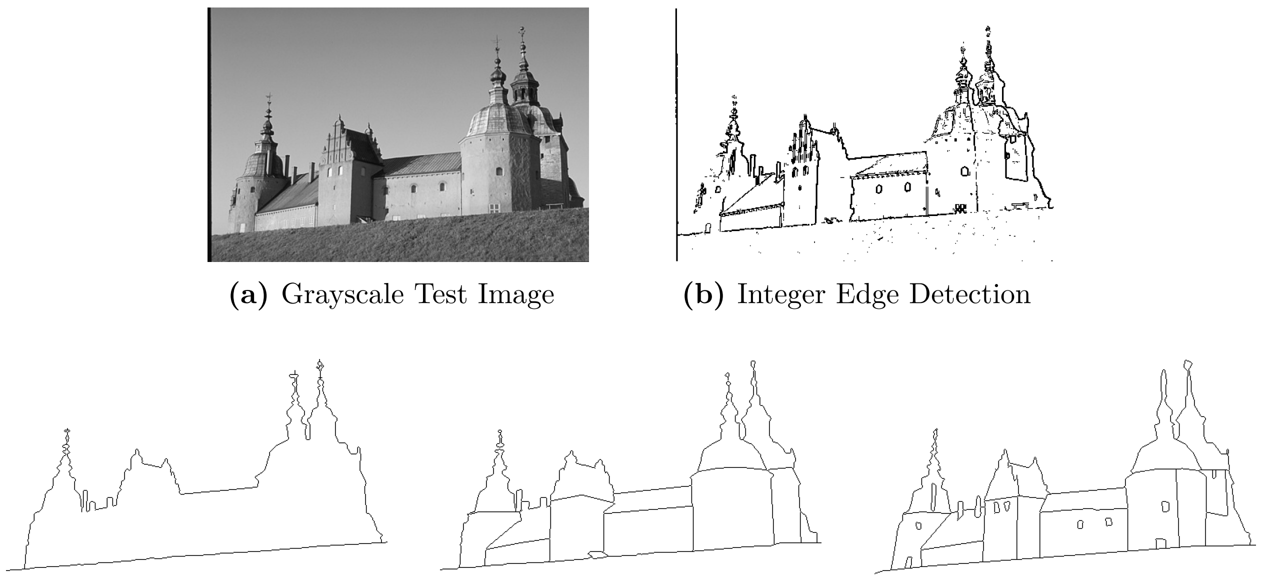
MNAE for SORN and Integer Designs with BSDS 500 The BSDS 500 contains 200 test images for the purpose of edge detection evaluation. These test images are used to evaluate on the differences between integer and SORN implementations. The applied metric is the mean normalized absolute error (MNAE), which takes the mean of all 200 NAEs per design. For this evaluation, all implemented SORN datatypes are considered, as well as two different thresholds per design. For the hybrid designs, the two rightmost SORN intervals with indices $T = w_s - 1$ and $T = w_s$ are considered as thresholds, for the full SORN designs the two intervals closest to zero are used. Since the edge results for the full SORN approach are negated, these close-to-zero thresholds can be considered as equivalent thresholds $T_e = w_s - 1$ and $T_e = w_s$. For the reference design, matching integer thresholds are used in order to allow a fair comparison. A matching threshold hereby means $T_{\mathrm{INT}} = 200$ for $T_{\mathrm{SORN}} = (200, 250]$, $T_{\mathrm{INT}} = 250$ for $T_{\mathrm{SORN}} = (250, \infty]$, etc.
The results of the described evaluation are given in table 4.1. It can be observed that for both the hybrid and full SORN approach, the linear datatypes show a lower difference to the integer

Table 4.1.: The mean normalized absolute error (MNAE) between SORN and reference integer implementation for 200 test images from BSDS500 [8] .

SORN datatype		**hybrid SORN**				**full SORN**	
		$lin6\|^{250}_{nz,nn}$	$log10\|^{256}_{nn}$	$lin11\|^{250}_{nz,nn}$		$log15\|^{512}$	$lin15\|^{300}$
MNAE	$T = w_s$	0.0659	0.1200	0.0598	$T_e = w_s$	0.1396	0.0667
	$T = w_s - 1$	0.1167	0.2323	0.0852	$T_e = w_s - 1$	-	0.0673

reference than the logarithmic distributed ones. In addition, the threshold value $T = w_s$ shows the best performance. In general, differences to the reference design below 7% can be achieved, which again shows that the SORN approach achieves a similar performance, but can still not be rated as better or worse than the integer design. For this purpose, a third, independent reference is required, which is given in the BSDS 500 with the *ground truth* solution.

Ground Truth Comparison with BSDS 500 The ground truth (GT) solutions are human made edge detections from different human subjects for the test images within the BSDS 500 [AMFM11], which include the main aspects of an image from the subjects point of view. In figure 4.5 three of these GT solutions are shown for a grayscale test image from BSDS 500, together with the integer reference edge detection result. For the test images from the dataset, in total 6 different GT solutions per test image are available. To compare the performance of the proposed SORN edge detection designs with the integer reference, again the mean



(a) Grayscale Test Image
(b) Integer Edge Detection
(c) Ground Truth 1
(d) Ground Truth 2
(e) Ground Truth 3

Figure 4.5.: Grayscale test image from BSDS 500 [AMFM11] with integer edge detection result and different ground truth solutions.

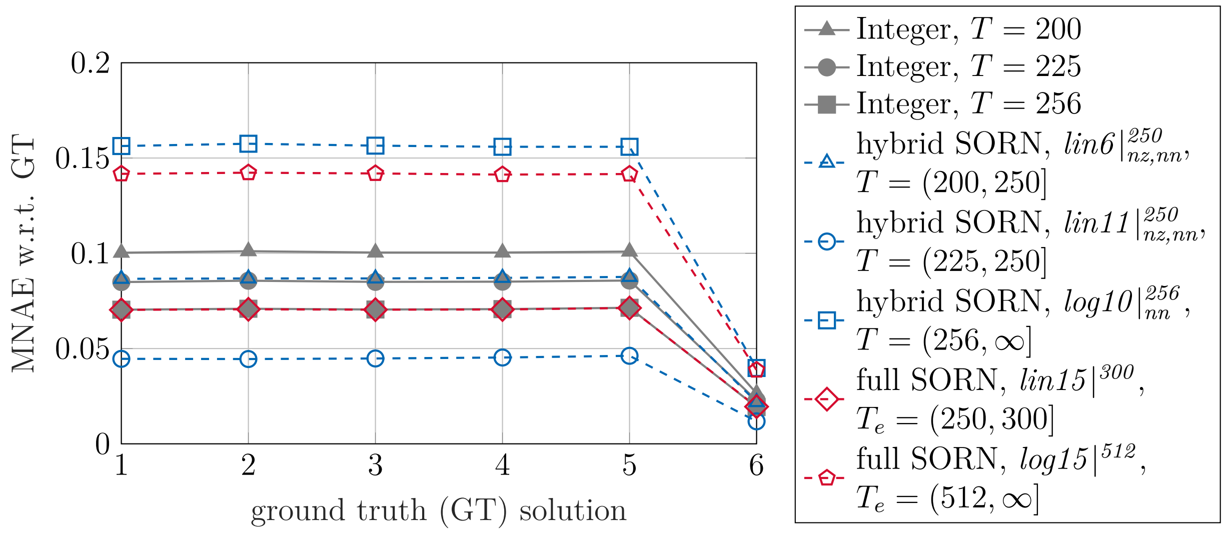


Figure 4.6.: The mean normalized absolute error (MNAE) w.r.t. 6 ground truth solutions for the different integer and SORN Sobel implementations over 200 test images from BSDS500 [8].

normalized absolute error (MNAE) is considered, but this time with respect to the different GT solutions:

$$\text{MNAE} = \frac{\sum_{i=1}^{N_i} \left(\frac{\sum_{x=1}^{N_x} \sum_{y=1}^{N_y} (\boldsymbol{GT}_i(x,y) \neq \boldsymbol{E}_i(x,y))}{N_x N_y} \right)}{N_i} \tag{4.10}$$

$\boldsymbol{E}_i$ hereby is the respective integer, hybrid or full SORN edge result, $\boldsymbol{GT}_i$ the ground truth solution and N_i the number of test images. The MNAE with respect to GT is shown in figure 4.6 for the five different SORN implementations, each with that threshold value leading to the lowest error, and for the integer reference design with three different threshold values corresponding to the SORN thresholds. As for the previous evaluation, it can be observed that the linear distributed SORN datatypes lead to a lower error metric than the logarithmic ones. Compared to the integer references, similar or even better results can be achieved for the SORN designs, especially for the hybrid approach. As discussed before, and also mentioned in [LMDBB13], the lower difference to GT does not necessarily indicate a better performance for any edge detection application, but it shows that the SORN approach achieves at least similar results than the integer reference and can be used as a replacement.

4.1.4. Synthesis Results

The synthesis results for a 28 nm CMOS technology from STM are given in table 4.2. All integer and SORN designs were synthesized without pipeline registers for a target frequency of 1 GHz and the respective maximum frequency. For 1 GHz, all SORN designs achieve a lower area and power consumption compared to the integer reference. The area reduction is higher for the hybrid designs with up to 45% reduction, whereas the power consumption is reduced by all SORN designs on a similar level, with up to 44% reduction. In addition, all SORN designs achieve a higher maximum frequency than the integer reference while still maintaining a lower

Table 4.2.: CMOS STM 28 nm technology synthesis results for all implemented integer and SORN Sobel designs [8].

		Integer	**hybrid SORN**			**full SORN**	
			$lin6\|^{250}_{nz,nn}$	$log10\|^{256}_{nn}$	$lin11\|^{250}_{nz,nn}$	$log15\|^{512}$	$lin15\|^{300}$
target freq.	[MHz]	1000	1000	1000	1000	1000	1000
runtime	[ns]	0.962	0.958	0.962	0.962	0.961	0.962
area	[µm^2]	1153.987	638.765	693.110	733.421	989.808	1132.282
power	[µW]	550.337	329.210	349.387	349.964	309.294	324.075
max. freq.	[MHz]	1263	1681	1603	1605	1661	1715
runtime	[ns]	0.792	0.595	0.624	0.623	0.602	0.583
area	[µm^2]	2087.165	1100.294	1157.251	1245.706	1661.213	2017.642
power	[µW]	1979.710	757.566	774.962	838.914	403.631	413.465

area and power consumption. For the respective maximum frequency, again the area is mostly reduced by the hybrid approach with up to 47%, while the reduction in power consumption is the highest for the full SORN approach with up to 80%. Similar results can be achieved when implementing the different designs on an FPGA, as shown in the appendix B.3 with the synthesis results for an Artix-7 FPGA in table B.2.
Together with the algorithmic performance evaluation from the previous section, it can be concluded that the SORN Sobel approach achieves similar edge detection results compared to a state-of-the-art integer implementation, while improving all hardware parameters complexity, maximum frequency and power consumption.

4.2. SORN Preprocessor for Wireless MIMO Communication

For the edge detection approach discussed in the previous section, SORNs were used within an application where decisions are made based on predefined thresholds. Hereby precision and accuracy of the final result are no major issue, as long as a sufficient accuracy for threshold comparison is provided. In the remainder of this chapter, a second general use case is discussed, where SORNs are used in a preprocessing step in order to reduce the solution set for an optimization problem. The application is symbol detection for wireless multiple-input and multiple-output (MIMO) communication.

In wireless broadband telecommunication systems, signals are transmitted between antennas through a wireless, time-variant channel using radio waves [Spe21]. A widely used technique to improve single antenna systems with one transmit and receive antenna are so-called MIMO systems which use multiple antennas at both ends. Since multiple transmit antennas send different pieces of information over the wireless channel within the same frequency band and time frame, a higher spectral efficiency can be achieved, i.e. a higher data rate per frequency bandwidth, measured in bits per second per Hertz [RG08]. The price of this improvement are higher hardware costs for the radio frequency (RF) components, as well as an increased complexity of the baseband signal processing at both transmitter and receiver [RPL+13]. The first wireless communication standards to include the MIMO approach were the *IEEE 802.11n-2009* standard (Wi-Fi 4) [IEEE09], as well as the Evolved High Speed Packet Access (HSPA+) and the long-term evolution (LTE) standard [DJPM09].

One way of implementing the MIMO approach is to use a single transmitter and receiver, each with multiple antennas. This is called point-to-point MIMO. In the following, a different scenario with multiple non-cooperative single-antenna clients, communicating with a multiple-antenna basestation is considered. This approach is called multi-user (MU) MIMO [RPL+13]. In figure 4.7 such a scenario for the upload of N_C clients to a basestation with N_B antennas is shown. The baseband signal processing for this upload scenario performed at the receiving basestation will be discussed in the subsequent section 4.2.1, followed by the introduction of the SORN preprocessor approach in section 4.2.2, its implementation in section 4.2.3, and evaluations on the algorithmic and hardware performance of the preprocessor within a MIMO system in sections 4.2.4 - 4.2.8.

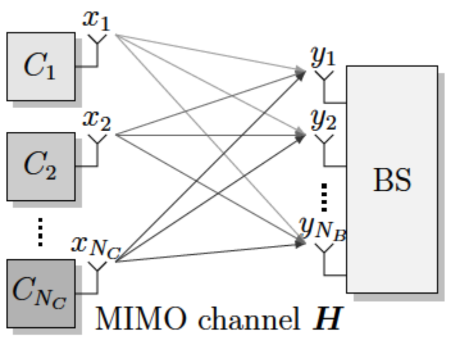


Figure 4.7.: Multiple client upload scenario for a wireless MIMO transmission with N_C transmit and N_B receive antennas [7].

4.2.1. MIMO Symbol Detection

All clients C_i simultaneously transmit digital modulated data x_i, which contains fixed elements from a finite symbol alphabet $\mathcal{S}$, corresponding to the chosen digital modulation. All symbols in $\mathcal{S}$ have an identical *a priori* probability. Depending on the modulation bitwidth m, the cardinality of the symbol alphabet is $|\mathcal{S}| = 2^m$. The most common modulation schemes are phase-shift keying (PSK) and quadrature amplitude modulation (QAM). Figure 4.8 shows the constellation diagrams of some basic PSK and QAM schemes. All symbols are normalized to a signal power $\sigma_{\mathcal{S}}^2 = 1$ [Kam04].

The transmitted symbols x_i form the transmit vector $\boldsymbol{x} \in \mathcal{S}^{N_C}$, which can be either real- or complex-valued, depending on the modulation. The vector $\boldsymbol{x}$ is transmitted over the wireless channel, which is modeled by the channel matrix $\boldsymbol{H} \in \mathbb{C}^{N_B \times N_C}$. This matrix is obtained by channel estimation techniques, which will not be discussed here. In the following, perfect channel state information (CSI) is assumed. The channel is considered as a flat fading, Rayleigh distributed channel with additive white Gaussian noise with variance σ_N^2 and zero mean, modeled by the noise vector $\boldsymbol{n} \in \mathbb{C}^{N_B}$. The entries of both the channel matrix and the noise vector are symmetric complex Gaussian, independently and identically distributed (i.i.d.) [BHEHZ16, RPL+13].

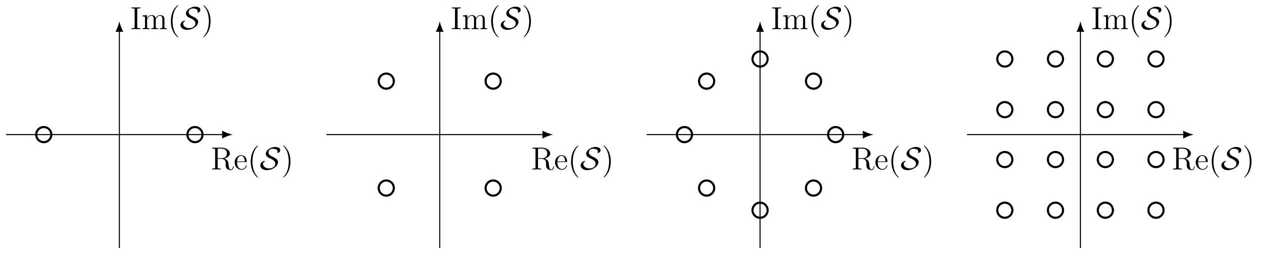
Every antenna on the basestation receives a complex-valued signal y_i, which together form the receive vector $\boldsymbol{y} \in \mathbb{C}^{N_B}$. The relation between transmitted and received signals can be described by the following linear system of equations [Lar09, YH15]:

$$\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x} + \boldsymbol{n} \tag{4.11}$$

At the basestation, the signal processing task is to calculate the estimate of the transmit symbol vector $\hat{\boldsymbol{x}} \in \mathcal{S}^{N_C}$ by solving the maximum likelihood estimation (MLE) problem:

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{x} \in \mathcal{S}^{N_C}}{\operatorname{argmin}} \left\| \boldsymbol{y} - \boldsymbol{H}\boldsymbol{x} \right\|_2 \tag{4.12}$$

The MLE problem is known to be non-deterministic polynomial-time (NP)-hard [YH15], which means that the complexity of a solving algorithm grows exponentially with the number of symbols in $\mathcal{S}$. Therefore an exhaustive search approach becomes impractical for higher order modulations. Instead, linear detection approaches like zero forcing (ZF) [CW07] or minimum mean square error (MMSE) [SFS11], all well as non-linear methods such as lattice reduction (LR) [WF03], soft interference cancellation (SIC) [YH15] or the tree-search based



(a) 2-PSK **(b)** 4-PSK/QAM **(c)** 8-PSK **(d)** 16-QAM

Figure 4.8.: Constellation diagrams for PSK and QAM modulations with $m = 1 \ldots 4$ bit.

sphere decoder (SD) [Lar09] are used to solve equation (4.12) in a maximum likelihood (ML) sense. The sphere decoder will be used as a reference ML detector in the following and will be discussed in detail in the subsequent section.
In the remainder of this work, a quadratic MIMO system with $N_C = N_B = N$ is assumed.

4.2.1.1. Sphere Decoding

The MLE problem from equation (4.12) is a least squares problem with a discrete search space, since $\boldsymbol{x} \in \mathcal{S}^N$ consists of symbols from a finite digital modulation alphabet $\mathcal{S}$. In order to avoid calculating the norm $\|\boldsymbol{y} - \boldsymbol{Hx}\|_2$ for every possible symbol vector $\boldsymbol{x}$ by means of an exhaustive search, a sphere decoder only takes into account those solutions that lie within a sphere around the received vector $\boldsymbol{y} \in \mathcal{S}^N$ and satisfy

$$\|\boldsymbol{y} - \boldsymbol{Hx}\|_2 \leq r \tag{4.13}$$

with the sphere radius r [HV05]. However, since the sphere is N-dimensional, and the ML solution should not be excluded, the computational effort to determine those $\boldsymbol{x}$ that fulfill both equations (4.13) and (4.12) is not yet reduced [BBW+05]. By using a QR decomposition (QRD) of the channel matrix $\boldsymbol{H} \in \mathbb{C}^{N\times N}$, equation (4.12) can be rewritten as

$$\hat{\boldsymbol{x}} = \underset{x\in\mathcal{S}^N}{\operatorname{argmin}} \|\underbrace{\boldsymbol{Q}^{\boldsymbol{H}}\boldsymbol{y}}_{\tilde{\boldsymbol{y}}} - \boldsymbol{Rx}\|_2 \tag{4.14}$$

with the orthogonal matrix $\boldsymbol{Q} \in \mathbb{C}^{N\times N}$ and the upper triangular matrix $\boldsymbol{R} \in \mathbb{C}^{N\times N}$ [Lar09, BBW+05]. $\boldsymbol{Q}^{\boldsymbol{H}}$ denotes the Hermitian transposition of $\boldsymbol{Q}$, also called conjugate transpose. Details on the QRD can be found in the appendix A.1 and in [GVL96].
The squared norm from the rewritten MLE problem (4.14) can now be defined element-wise [7]:

$$\|\tilde{\boldsymbol{y}} - \boldsymbol{Rx}\|_2^2 = \sum_{j=1}^{N} \left| \tilde{y}_j - \sum_{i=j}^{N} (R_{ji}x_i) \right|^2 \tag{4.15}$$

Because $\boldsymbol{R}$ is upper triangular, the problem can be reduced to a single dimension by calculating the last element $i = j = N$ first, before successively obtaining the complete norm. This transforms the problem into a tree search with a tree that traverses the iterations j from equation (4.15) in an inverse manner, starting with $j = N$, which corresponds to the tree level $l = 1$. Such a tree is shown in figure 4.9 with a dimension $N = 3$ and a symbol alphabet with $|\mathcal{S}| = 2$. From equation (4.15) the recursive error $e(l)$ at every tree level l can be defined [7, BBW+05]:

$$e(l) = \left| \tilde{y}_{N-l+1} - \sum_{i=N-l+1}^{N} R_{N-l+1,i}\, x_i \right|^2 + e(l-1) \tag{4.16}$$

Starting from the root node, the error at the first level $l = 1$ with $e(0) := 0$ is calculated as

$$e(1) = |\tilde{y}_N - R_N\, x_N|^2 \tag{4.17}$$

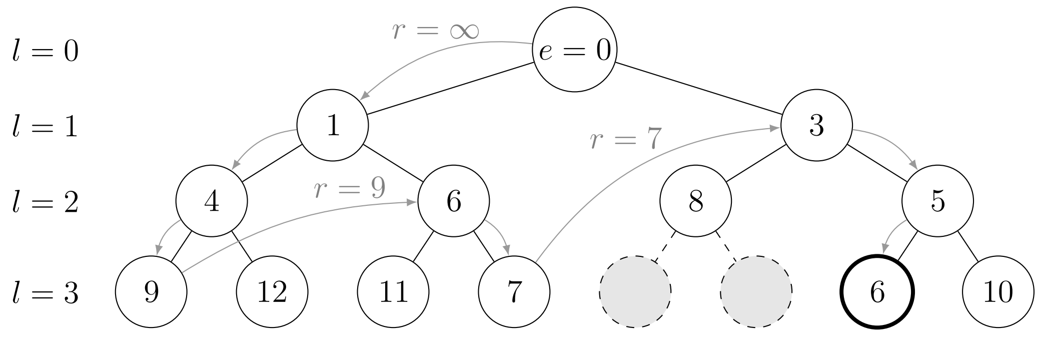


Figure 4.9.: Illustration of a Schnorr-Euchner SD algorithm with pruning, for a system with $N = 3$ and $|\mathcal{S}| = 2$. Each node contains the accumulated error metric $e(l)$ for the respective path. The adaptive radius r is adjusted each time the bottom level is reached [7].

for both branches with the respective x_N. Both results are compared to the radius, which is initially set to $r = \infty$. According to Schnorr-Euchner (SE) [SE94], the path with lower error metric is followed first, if both fulfill $e(l) < r$. This process is repeated for the lower tree levels until the bottom level is reached and $e(N)$ represents a complete N-dimensional norm for one symbol vector $\boldsymbol{x}$. This is called a depth-width or depth-first search [BHEHZ16]. A sphere decoder with pruning adjusts the radius to $r = e(N)$ every time the bottom tree level is reached. Then the algorithm continues with the next, not yet evaluated branch at a higher level until $e(l) > r$ or the bottom level is reached again, and the radius is adjusted accordingly. This process is repeated until there are no more branches to evaluate. The determined bottom level node with the lowest error metric defines the estimated symbol vector $\hat{\boldsymbol{x}}$.

4.2.2. SORN Preprocessor Approach

The number of possible symbol vectors $\boldsymbol{x} \in \mathcal{S}^N$ to solve the MLE problem (4.12) is obtained from the modulation bitwidth m and the MIMO system size N:

$$|\mathcal{S}|^N = (2^m)^N \tag{4.18}$$

The straight forward approach of computing the required norm from equation (4.12) for all possible $(2^m)^N$ symbol vectors is usually too complex and time consuming, which is why detection algorithms like ZF, SD or others are required. As shown in section 3.4.1, however, SORN arithmetic is not only less complex, but also a 2-3 times faster than standard integer or fixed point arithmetic, which makes an exhaustive search evaluation of problem (4.12) feasible. Similar to the twelve-dimensional robotics problem from [Gus16], a computation of the norm $\|\boldsymbol{y} - \boldsymbol{H}\boldsymbol{x}\|_2$ with any reasonable-length SORN datatype for all possible symbol vectors $\boldsymbol{x}$ does not lead to a single solution, because the precision of the SORN format is not high enough. The norm for any $\boldsymbol{x}_i \in \mathcal{S}^N$ will be a SORN value with one or multiple 1_2 bits, representing an interval $\geq 0_{10}$. Two different symbol vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ can hereby lead to the same norm in SORN representation. Consequently, a threshold is introduced to determine which of the SORN computed norms are considered small enough to lead to a ML solution for problem (4.12). This is done by determining that SORN interval bit which is closest to 0_{10} among all solutions. Consider the following example showing the SORN computed norms for three different symbol vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$ using the *lin9*$|^1$ SORN datatype from table 3.6:

$$\begin{aligned}\|\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x}_1\|_2 &= 000001110_2 = (0,1]\\ \|\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x}_2\|_2 &= 000001100_2 = (0,{}^2\!/\!_3]\\ \|\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x}_3\|_2 &= 000000011_2 = ({}^2\!/\!_3,\infty]\end{aligned} \tag{4.19}$$

Both $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ would be considered as a valid solution because they produce norms close to 0_{10}, whereas $\boldsymbol{x}_3$ would be marked as a non-valid solution. Depending on the chosen SORN datatype, the threshold interval does not necessarily have to be the open zero interval, yet it is in most of the evaluated cases discussed below.
This computation of the norm for all possible symbol vectors $\boldsymbol{x} \in \mathcal{S}^N$ with SORN arithmetic leads to a set of possible estimation results $\hat{\boldsymbol{x}} \in \mathcal{R} \subset \mathcal{S}^N$ with $|\mathcal{R}| \leq |\mathcal{S}|^N$. $\mathcal{R}$ is the set of remaining solutions after SORN processing [7]. Since $|\mathcal{R}| > 1_{10}$, the described process is considered as a preprocessing step within a MIMO detection to reduce the solution space for a state-of-the-art detector implemented in a standard format like fixed point.

4.2.3. Implementation

The norm within the MLE problem (4.12) requires the computation of a square root, which can be avoided by using the squared norm $\|\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x}\|_2^2$ instead, without affecting the result of the optimization. The general block diagram of the SORN preprocessor is depicted in figure 4.10a. In the first stage, the complex matrix-vector multiplication $\boldsymbol{H}\boldsymbol{x}$ is carried out, which requires $4N^2$ multiplications as well as $4N^2 - 2N$ additions for both the real and the imaginary part, and results in a first intermediate vector $(\boldsymbol{H}\boldsymbol{x}) \in \mathbb{C}^N$. The elements of this vector are obtained by the following equations for $i = 1 \dots N$ [5]:

$$\begin{aligned}\operatorname{Re}(\boldsymbol{H}\boldsymbol{x})_i &= \sum_{j=1}^{N} (\operatorname{Re}(H_{ij}) \operatorname{Re}(x_j) - \operatorname{Im}(H_{ij}) \operatorname{Im}(x_j))\\ \operatorname{Im}(\boldsymbol{H}\boldsymbol{x})_i &= \sum_{j=1}^{N} (\operatorname{Re}(H_{ij}) \operatorname{Im}(x_j) + \operatorname{Im}(H_{ij}) \operatorname{Re}(x_j))\end{aligned} \tag{4.20}$$

The second stage subtracts the first intermediate vector $(\boldsymbol{H}\boldsymbol{x})$ from $\boldsymbol{y}$ using $2N$ subtractions, leading to the second intermediate vector $(\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x}) \in \mathbb{C}^N$. The elements of this vector are obtained by the following equations for $i = 1 \dots N$:

$$\begin{aligned}\operatorname{Re}(\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x})_i &= \operatorname{Re}(y_i) - \operatorname{Re}(\boldsymbol{H}\boldsymbol{x})_i\\ \operatorname{Im}(\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x})_i &= \operatorname{Im}(y_i) - \operatorname{Im}(\boldsymbol{H}\boldsymbol{x})_i\end{aligned} \tag{4.21}$$

In the final stage, the squared norm is calculated using $2N$ square operations and $2N-1$ additions:

$$\|\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x}\|_2^2 = \sum_{i=1}^{N} \left(\operatorname{Re}(\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x})_i^2 + \operatorname{Im}(\boldsymbol{y}-\boldsymbol{H}\boldsymbol{x})_i^2\right) \tag{4.22}$$

The described architecture can be implemented using the SORN Hardware Generator presented

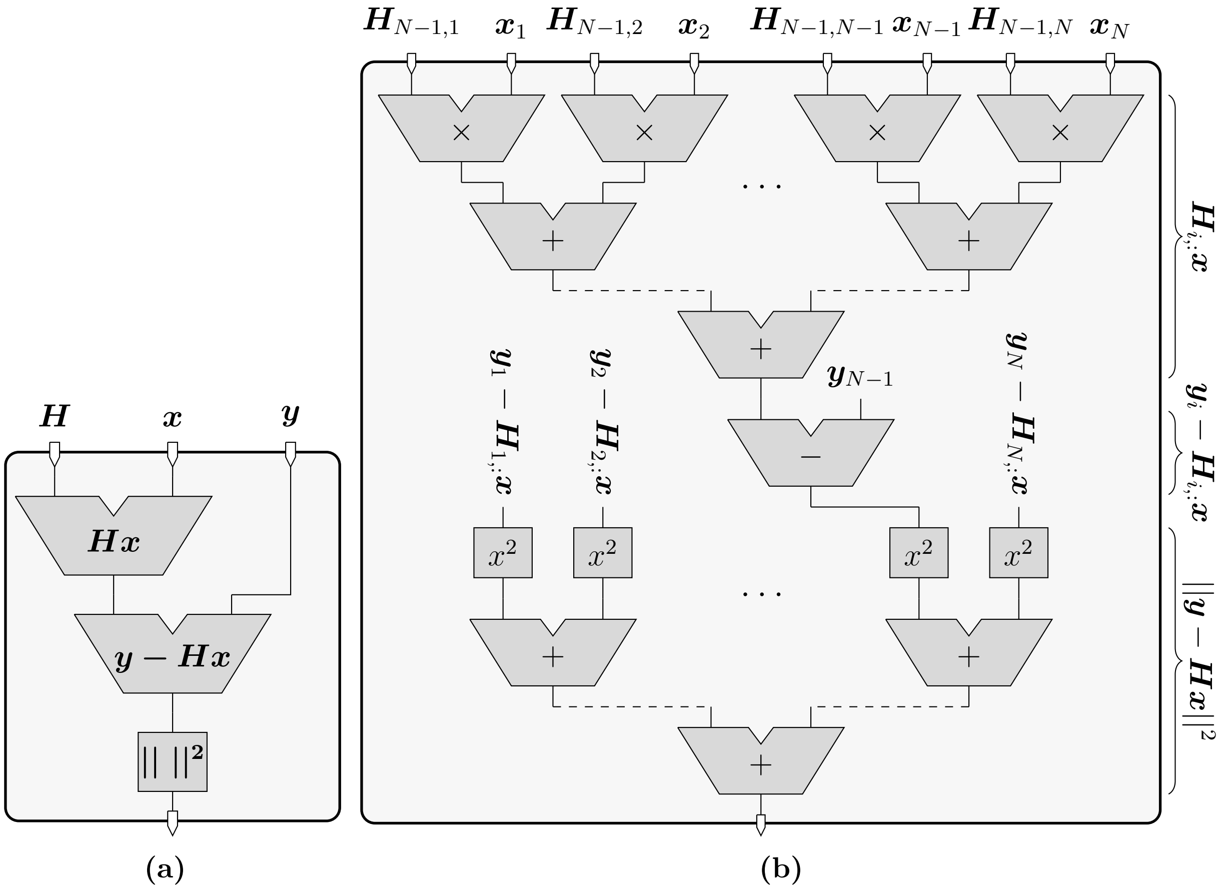

$\boldsymbol{H}$
$\boldsymbol{x}$
$\boldsymbol{y}$
$\boldsymbol{Hx}$
$\boldsymbol{y}-\boldsymbol{Hx}$
$|| \; ||^2$
(a)
$\boldsymbol{H}_{N-1,1}$
$\boldsymbol{x}_1$
$\boldsymbol{H}_{N-1,2}$
$\boldsymbol{x}_2$
$\boldsymbol{H}_{N-1,N-1}$
$\boldsymbol{x}_{N-1}$
$\boldsymbol{H}_{N-1,N}$
$\boldsymbol{x}_N$
$\times$
$+$
$-$
$\boldsymbol{y}_{N-1}$
$\boldsymbol{y}_1-\boldsymbol{H}_{1,:}\boldsymbol{x}$
$\boldsymbol{y}_2-\boldsymbol{H}_{2,:}\boldsymbol{x}$
$\boldsymbol{y}_N-\boldsymbol{H}_{N,:}\boldsymbol{x}$
x^2
$\boldsymbol{H}_{i,:}\boldsymbol{x}$
$\boldsymbol{y}_i-\boldsymbol{H}_{i,:}\boldsymbol{x}$
$||\boldsymbol{y}-\boldsymbol{Hx}||^2$
(b)

Figure 4.10.: Datapath structure for the SORN preprocessor without pipeline registers as (a) general block diagram and (b) realization for real-valued system.

in section 3.3. Listing A.1 in the appendix A.2 gives the corresponding specification file to generate a SORN MIMO preprocessor with $N = 4$ and one pipeline stage for the *lin13*$|^1$ datatype from table 3.6.

If a real-valued MIMO system is considered, the implementation of the preprocessor reduces to N^2 multiplications and $N^2 - N$ additions for the first stage, N subtractions for the second stage, and N square operations as well as $N - 1$ additions for the third stage. The implementation of a SORN preprocessor for a real-valued system is depicted in figure 4.10b.

4.2.4. Hardware Results

Table 4.3 gives the synthesis results for a complex-valued SORN preprocessor from figure 4.10a with two pipeline stages, for a 2×2 and 4×4 MIMO system with different SORN datatypes from table 3.6. The utilized technology is a general-purpose 90 nm process from Taiwan Semiconductor Manufacturing Company Limited (TSMC). The complexity by means of the area utilization is given in kilo gate equivalents (GEs), a technology independent measure where the total area in µm^2 is normalized by the area of a 2-input NAND gate with lowest driver strength. The throughput is given in mega operations (OPs) per second and is obtained from the maximum frequency f and the number of required clock cycles per detection C, which in turn is composed of the number of possible solutions from equation (4.19) and the number of pipeline stages N_{pipe}:

$$\text{throughput}\ \left[\frac{\text{MOPs}}{s}\right] = \frac{f}{C} = \frac{f}{(2^M)^N + N_{\text{pipe}}} \tag{4.23}$$

One operation is hereby considered as the computation of all possible solutions $\boldsymbol{x} \in \mathcal{S}^N$ per received vector $\boldsymbol{y}$, resulting in the reduced solution set $\mathcal{R}$, which can be used for further processing. The applied modulations are a 4-PSK [2, 5], also called quadrature phase-shift keying (QPSK), and a 4-QAM [3], with a bitwidth $m = 2$, respectively.
Solely linear SORN datatypes are considered, since the algorithmic evaluations from the subsequent sections will show that they perform better than the logarithmic ones for the given application. For those datatypes with an exact 0.5 value, the complex PSK/QAM symbols x_i are scaled to $\pm 0.5 \pm \mathrm{j}\, 0.5$ before converting to SORN representation.
Comparing the results for those designs with unum-based and half-open linear SORN datatypes, it can be observed that for both 2×2 and 4×4 cases, the unum-based designs achieve a slightly lower complexity than the linear datatypes of similar bitwidth. This also holds for the energy for the 4×4 case comparing *unum8* and *lin9*$|^{1}$. For all other measures, however, both the 8 and 16 bit unum-bases designs are outperformed by the linear ones with similar bitwidth, yet not by multiple orders of magnitude. These results are consistent with the evaluation for basic arithmetic operations from section 3.4.
When comparing the results for the linear-based designs of different bitwidth w_s, it can be observed that with an increasing w_s also the complexity and energy increase, while frequency and throughput decrease, as expected. For different datatypes with same bitwidth $w_s = 13$ and $w_s = 17$, however, small differences in the results can be observed, caused by the slightly different logic of the respective circuits. This behavior was also observed in section 3.4.
In order to compare the hardware performance of the SORN preprocessor (only 4×4) to state-of-the-art (SOTA) designs, table 4.3 also includes the hardware measures for sphere decoder and QR decomposition designs from the literature. Some of those results are given for another technology node, therefore the given results are normalized to 90 nm in order to allow a fair comparison. The first observation is that the SORN preprocessor designs can run at much higher frequencies than the SD and QRDs, mostly more than one order of magnitude. Despite the high frequency, the energy is also much lower for the SORN designs, also about one order of magnitude on average. The complexity is at least similar to the SOTA designs, depending on the implementation it can also be up to an order of magnitude lower for the preprocessor. The throughput of the SOTA designs varies, therefore the SORN preprocessor throughput can be lower, similar or higher, but all within one order of magnitude.
This comparison shows that the SORN preprocessor could be incorporated in a SOTA detector without limiting the throughput or adding a disproportionate amount of complexity or power consumption. Two complete detector designs applying the SORN preprocessor are presented in sections 4.3 and 4.4.

Table 4.3.: Synthesis results for the SORN preprocessor with different datatypes, for a 2×2 and 4×4 complex-valued MIMO system with QPSK modulation and two pipeline stages, for 90 nm CMOS technology. For comparison with state-of-the-art architectures, the hardware measures for reference SD and QRD designs are given. To allow a fair comparison, results for a different technology are normalized to 90 nm $^{(c-d)}$ [5].

	datatype	**complexity**	**frequency**	**energy**	**throughput**
		[kGE]	[GHz]	$[\frac{\mu W}{MHz}]$	$[\frac{MOP}{s}]$
		2×2 SORN preprocessor			
[2]$^{(a)}$	*unum8*	4.80	2.02	12.08	112.22
[2]$^{(a)}$	*unum16*	18.29	1.69	44.12	93.89
[5]	$\mathit{lin9}\|^{1}$	5.48	2.19	8.09	121.67
[5]	$\mathit{lin13}\|^{1}$	10.72	1.95	14.63	108.33
		4×4 SORN preprocessor			
[2]$^{(a)}$	*unum8*	17.15	1.58	48.30	6.12
[2]$^{(a)}$	*unum16*	64.98	1.25	190.85	4.84
[5]	$\mathit{lin9}\|^{1}$	26.30	1.90	54.28	7.36
[3]	$\mathit{lin11}\|^{1}$	37.23	1.74	35.38	6.74
[5]	$\mathit{lin13}\|^{1}$	49.07	1.62	65.90	6.28
[3]	$\mathit{lin13}\|^{2}$	48.92	1.57	45.47	6.09
[3]	$\mathit{lin13}\|^{1}_{e1/2}$	42.56	1.69	37.41	6.55
[3]	$\mathit{lin15}\|^{2}_{e1/2}$	59.13	1.60	48.36	6.20
[3]	$\mathit{lin17}\|^{3}_{e1/2}$	80.73	1.50	64.86	5.81
[3]	$\mathit{lin17}\|^{2}_{e1/2}$	89.18	1.52	68.69	5.89
		4×4 SD			
[ROP11]	-	39.53$^{(b)}$	0.471	34.10$^{(c)}$	2.82$^{(d)}$
[BWA^{+}12]	-	872$^{(b)}$	0.135	497$^{(c)}$	23.35$^{(d)}$
[YTC^{+}13]	-	153.9	0.109	264.49	20
		4×4 QRD			
[RLP13]	-	22.38$^{(b)}$	0.129	4403.10$^{(c)}$	0.65$^{(d)}$
[RLP15]	-	71.75$^{(b)}$	0.192	213.16$^{(c)}$	1.25$^{(d)}$
[HCW15]	-	452	0.143	654.13	35.75

$^{(a)}$ In [2] the results are given for one pipeline stage, here for two stages.

$^{(b)}$ *Normalized by:* area $\times$ $(\text{area}_{\text{NAND2}})^{-1}$

$^{(c)}$ *Normalized by:* energy $\times \left(\frac{\text{technology}}{90\,\text{nm}}\right)^{-1} \times \left(\frac{0.9}{V_{DD}}\right)^{2}$

$^{(d)}$ *Normalized by:* throughput $\times \frac{\text{technology}}{90\,\text{nm}}$

4.2.5. Reducing the Solution Set

As described in section 4.2.2, the result of the SORN preprocessor is a reduced solution set $\mathcal{R}$ that contains $|\mathcal{R}|$ possible estimation results $\hat{\boldsymbol{x}} \in \mathcal{R}$. In order to evaluate on the size of $\mathcal{R}$, the metric for the mean (number of) remaining solutions (MRS) is introduced as

$$\text{MRS} = \overline{|\mathcal{R}|} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} |\mathcal{R}|_i \tag{4.24}$$

which gives the arithmetic mean of $|\mathcal{R}|$ over N_{test} MIMO detections. In relation to the number of possible solutions, the MRS value can be given as a percentage:

$$\text{MRS}\,[\%] = \frac{\overline{|\mathcal{R}|}}{|\mathcal{S}|^N} \times 100\% = \frac{\overline{|\mathcal{R}|}}{(2^m)^N} \times 100\% \tag{4.25}$$

Figure 4.11 gives the MRS in [%] over the PSK modulation bitwidth m for a 2×2 and 4×4 MIMO system for different SORN datatypes. The linear and logarithmic half-open datatypes outperform the unum-based ones by far, even the 9 bit *log*/*lin* datatypes show a higher reduction of the solution set than the 16 bit unum-based one. As a general observation among the half-open datatypes, it can be stated that the higher the bitwidth, the higher the reduction. In addition, the linear datatypes show a better performance than the logarithmic ones. For all datatypes, the MRS is mostly stable over the modulation bitwidth m. For the respective best configurations, reductions of the solution set by more than 80% can be achieved. A second evaluation in figure 4.12 gives the MRS in % over the signal-to-noise ratio (SNR). The SNR values hereby represent the mean SNR over all receive antennas in decibel (dB). The evaluation covers a real-valued system with BPSK modulation ($m = 1$) in figures 4.12a and 4.12b, and a complex-valued system with QPSK modulation ($m = 2$) in figures 4.12c

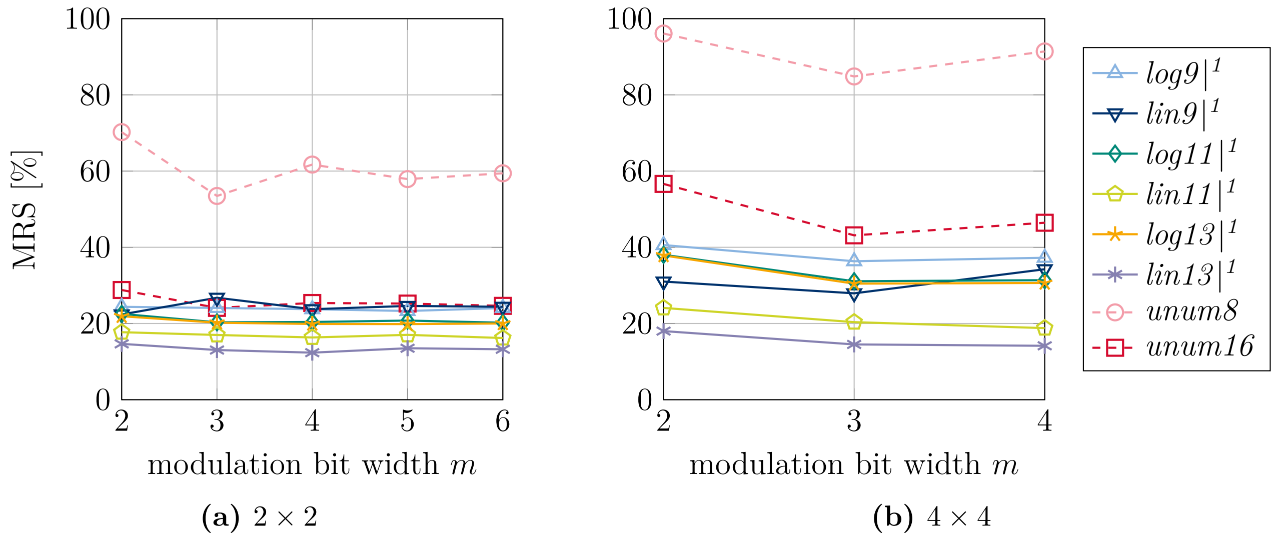
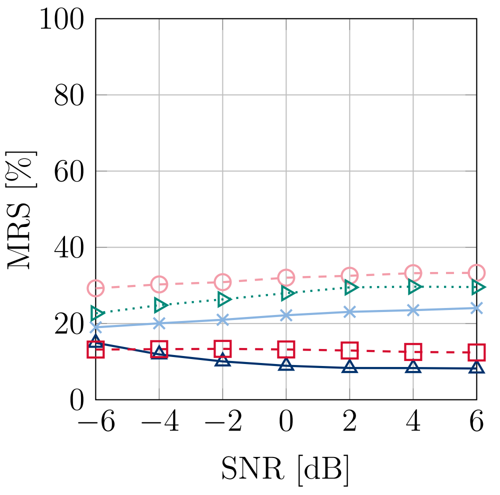
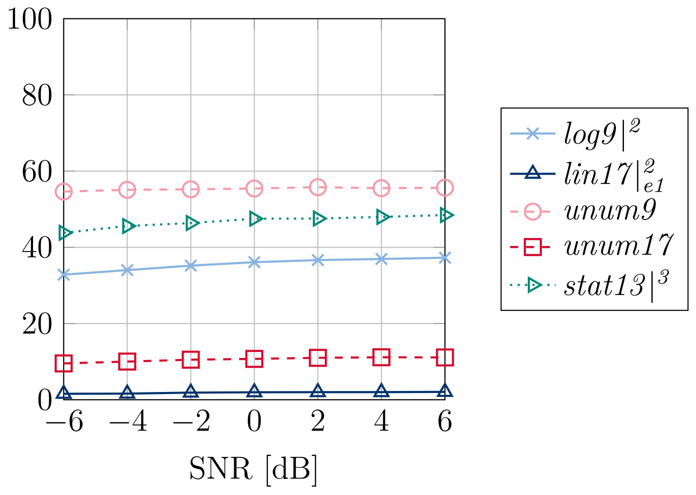


Figure 4.11.: The mean (number of) remaining solutions (MRS) for the MLE problem (4.12) in % after SORN preprocessing over modulation bitwidth m, for a complex-valued MIMO system with PSK modulation, a system size of (a) $N = 2$ and (b) $N = 4$, SNR = 30 dB, $N_{\text{test}} = 10^3$ and different SORN datatypes [2, 5].

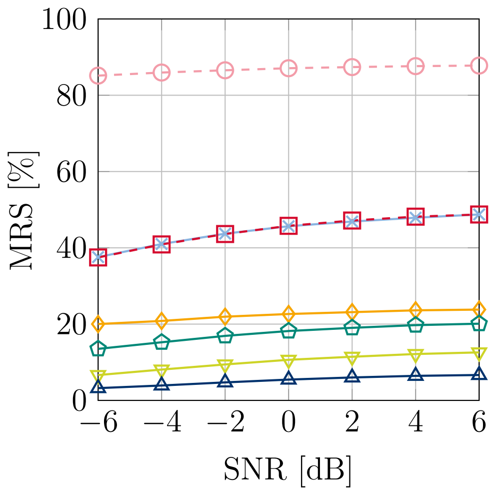
and 4.12d, both for $N = 4$ and $N = 8$, respectively. As in the previous evaluation, the half-open SORN datatypes outperform the unum-based ones for all cases. Especially for the complex-valued system it can be observed that the $log9|^2$ datatype shows the exact same results as the almost doubled-bitwidth *unum17* datatype, because they share the same lattice values. For the real-valued system this does not apply, because here the exact 1_{10} within the unum-based datatype matches the input data of the $\pm 1_{10}$ BPSK symbols. However, the 17 bit linear datatype, which also includes the exact 1_{10}, shows an even better performance as it does not include the other exact values from *unum17*.
Comparing the different half-open *lin* and *log* datatypes again shows that the higher the bitwidth, the higher the reduction. For the best configuration $lin17|^2_{e1}$ for the real-valued, and



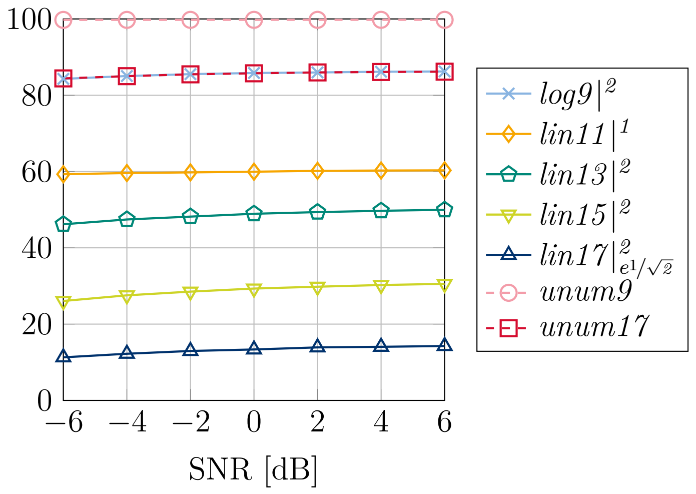
(a) Real-valued 4×4 MIMO system with BPSK modulation [4].



(b) Real-valued 8 × 8 MIMO system with BPSK modulation.



(c) Complex-valued 4 × 4 MIMO system with QPSK modulation [6, 7].



(d) Complex-valued 8×8 MIMO system with QPSK modulation [6, 7].

Figure 4.12.: The mean (number of) remaining solutions (MRS) for the MLE problem (4.12) in % after SORN preprocessing over SNR, for a real- and complex-valued MIMO system with BPSK and QPSK modulation, respectively, a system size of $N = 4$ and $N = 8$, and different SORN datatypes.

lin17$|^2_{e^1/\sqrt{2}}$ for the complex-valued system, reductions of up to 92% (4×4 real), 98% (8×8 real), 93% (4×4 complex), and 86% (8×8 complex) can be achieved. All datatypes show only small variations over different SNRs, some are almost constant.
The results for the real-valued system in figures 4.12a and 4.12b also include the statistically determined datatype discussed in section 3.2.4. The results show that for the given application scenario, this datatype approach does not lead to better results than the linear and logarithmic distributed datatypes.

4.2.6. Properties of the Remaining Solutions

After SORN preprocessing, the set of remaining solutions $\mathcal{R}$ is subject to further processing, for example with a state-of-the-art sphere decoder as will be discussed in sections 4.2.7 and 4.4. Before that step is taken, some properties of $\mathcal{R}$ are evaluated, which could influence the further processing.

Discarding the Maximum Likelihood Solution When the SORN preprocessor calculates the remaining solutions with a fixed threshold interval, it can happen that the ML solution, i.e. the solution that would be given from a maximum likelihood detector, is discarded by the preprocessor and not included in $\mathcal{R}$. Figure 4.13 shows the percentage of test cases where the ML solution is discarded by the SORN preprocessor, over the mean received SNR for a complex-valued 4×4 and 8×8 system, in reference to the MRS figures 4.12c and 4.12d. In comparison to the MRS results, it can be observed that those datatypes which lead to a high reduction of the solution set also have a higher probability of discarding the ML solution. However, while the MRS results are almost constant over SNR, the probability of discarding the ML solution is highly SNR depended and only noticeable for low SNR values. In addition,

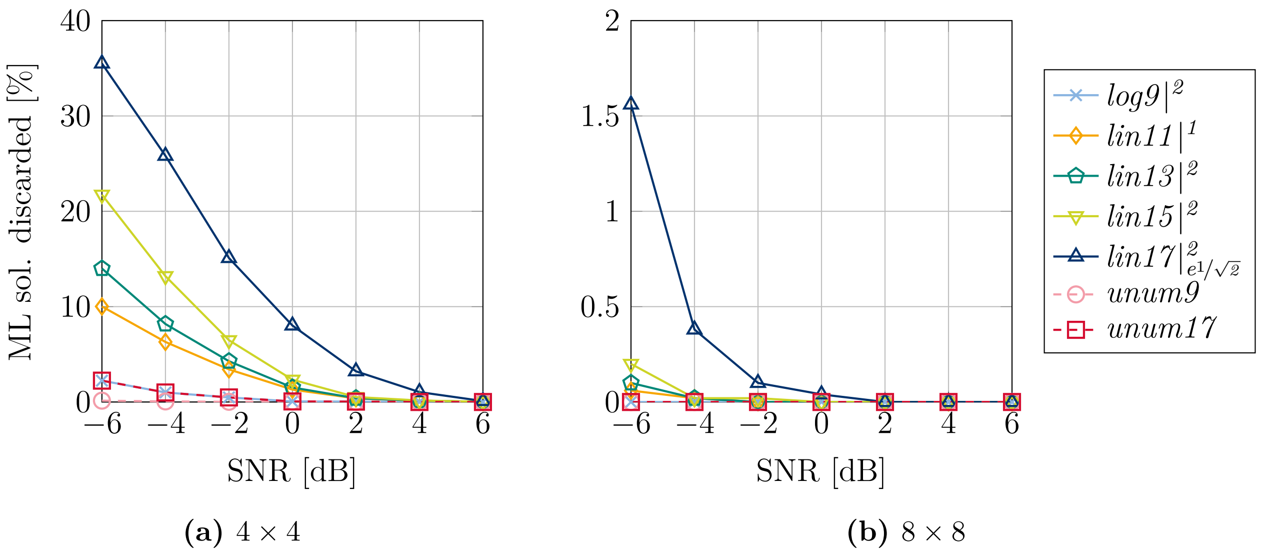


Figure 4.13.: Percentage of cases where the ML solution is discarded by the SORN preprocessor over SNR, for a complex-valued MIMO system with QPSK modulation, a system size of (a) $N = 4$ and (b) $N = 8$, and different SORN datatypes [7, Kno20].

only for the 4×4 case non-negligible probabilities occur, for the 8×8 case the probability is mostly below 0.5%. The reason for the higher discarding probability in the 4×4 case is the lower percentage of MRS, compared to 8×8.

Scatter Plots and Angle of Remaining Solutions In order to rate the suitability of the remaining solutions for further processing, one possible evaluation metric is the position of the estimated solutions $\hat{\boldsymbol{x}} \in \mathcal{R}$ within the complex plane. This position, or the respective angle when using a PSK modulation, are the main decision criteria for the demodulation process at the receiving basestation. Therefore the arithmetic mean of the remaining solutions per detection

$$\overline{\hat{\boldsymbol{x}}} = \frac{1}{|\mathcal{R}|} \sum_{i=1}^{|\mathcal{R}|} \hat{\boldsymbol{x}}_i \tag{4.26}$$

can be used as estimated solution for the MLE problem. In figure 4.14 the position of this mean estimation after SORN preprocessing is shown for $N_{\text{test}} = 10^3$ transmissions for the symbol $x_1 = {}^1\!/\!\sqrt{2} + \mathrm{j}\,{}^1\!/\!\sqrt{2}$ for a complex-valued 4×4 system with QPSK modulation using the

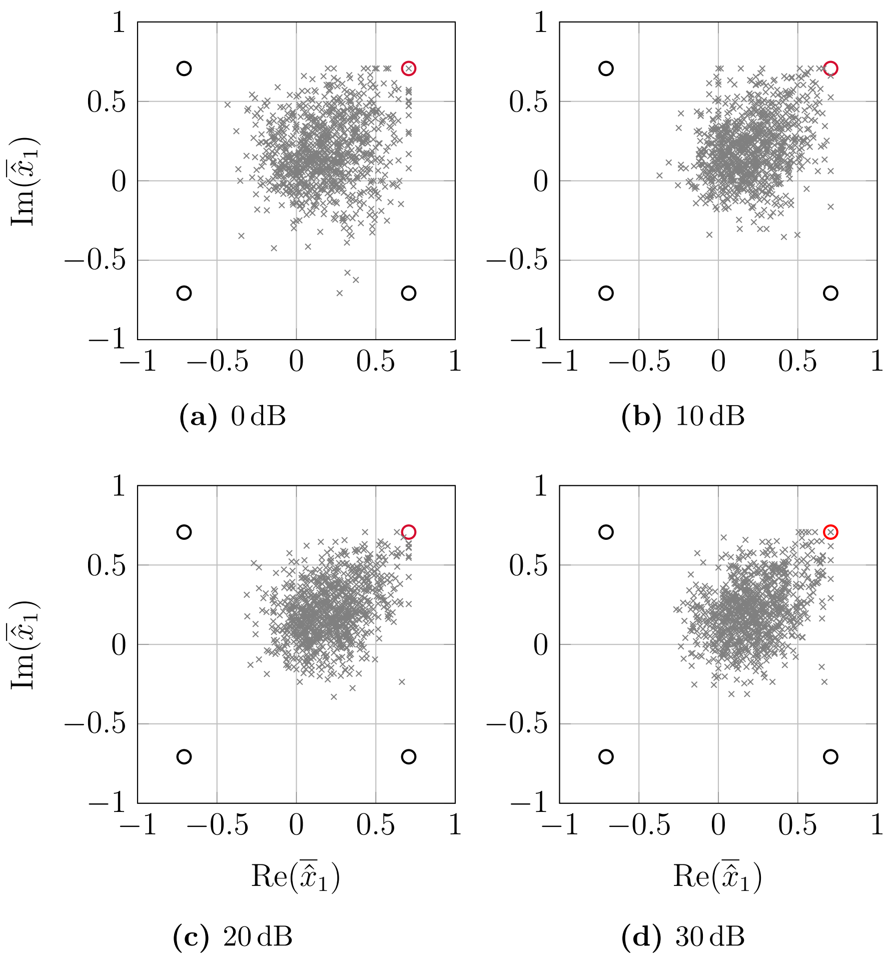
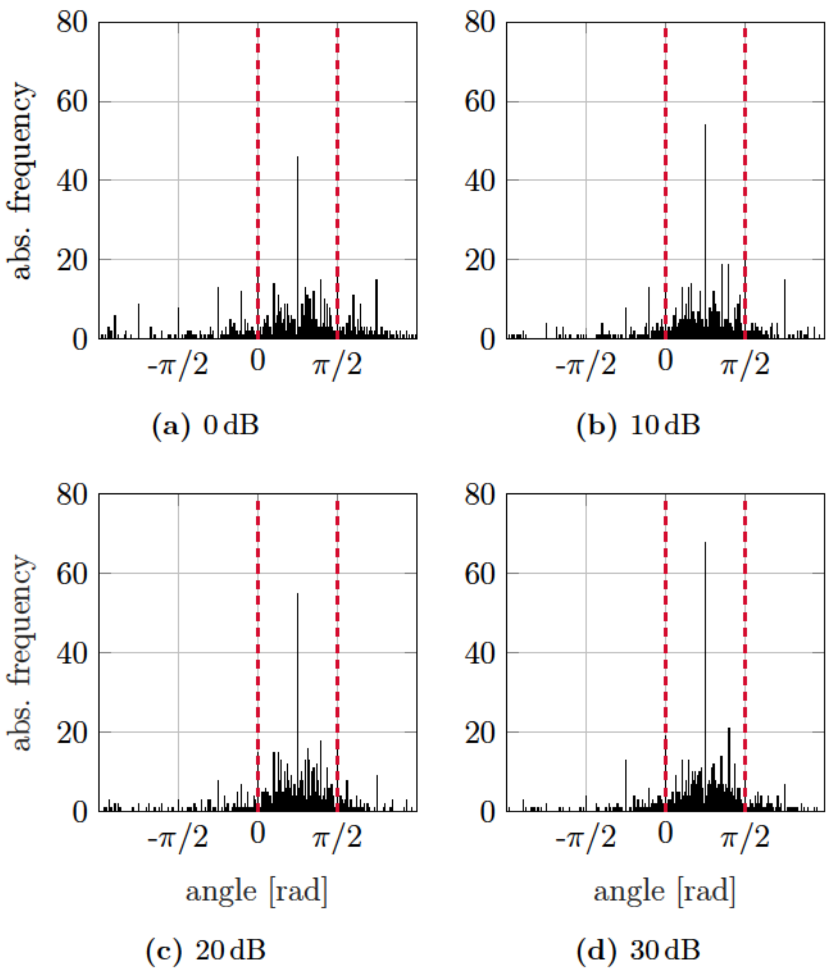


Figure 4.14.: Scatter plot for the arithmetic mean $\overline{\hat{x}}_1$ of the remaining solutions after SORN preprocessing for 10^3 transmissions of the symbol $x_1 = {}^1\!/\!\sqrt{2} + \mathrm{j}\,{}^1\!/\!\sqrt{2}$, for $N = 4$ with QPSK, the *lin13*$|^1$ SORN datatype and different SNR values [5].

lin13$|^1$ datatype and different SNR values. Figure 4.15 shows the heuristic of the corresponding angles $\angle\overline{\hat{x}}_1$ for the same test setup. The results for a similar evaluation with an 8-PSK and transmitted symbol $x_2 = {}^2\!/_{3\sqrt{2}} + \mathrm{j}\,{}^1\!/_{2\sqrt{2}}$ are given in the appendix B.5 in figures B.3 and B.4, respectively. The heuristics of the respective angle for both the QPSK and 8-PSK evaluation show the shape of a normal distribution with the angle of the correct estimation $\angle x_1 = \frac{\pi}{4}$ and $\angle x_2 = \frac{\pi}{8}$ as mean value. The graphs also show the decision boundaries that would be used for a demodulation. For the QPSK case in figure 4.15 with an SNR of 0 dB, in about 63% of the cases the angle $\angle\overline{\hat{x}}_1$ lies within the decision boundaries and would lead to a correct detection. For an SNR of 30 dB, this probability increases to 75%. For the 8-PSK case in figure B.4, the results decrease to 40% (0 dB) and 54% (30 dB). This evaluation shows that such a SORN preprocessor with the given configuration and SORN datatype would not be sufficient as a stand-alone detector, as will also be shown with a bit error rate (BER) analysis in section 4.2.7. In order to improve the preprocessing and the suitability of the remaining solutions for the detection process, figure 4.16 shows the position of the mean estimation $\overline{\hat{x}}_4$ for $N_{\text{test}} = 10^3$ transmissions for the symbol $x_4 = -{}^1\!/_{\sqrt{2}} + \mathrm{j}\,{}^1\!/_{\sqrt{2}}$ and a complex-valued 4×4 system with 4-QAM modulation for different SORN datatypes and a fixed SNR of 10 dB. The corresponding



(a) 0 dB (b) 10 dB (c) 20 dB (d) 30 dB

Figure 4.15.: Heuristic for the angle of the arithmetic mean $\angle\overline{\hat{x}}_1$ of the remaining solutions after SORN preprocessing for 10^3 transmissions of the symbol $x_1 = {}^1\!/_{\sqrt{2}} + \mathrm{j}\,{}^1\!/_{\sqrt{2}}$, for $N = 4$ with QPSK, the *lin13*$|^1$ datatype and different SNR values [5].

heuristic of the angles $\angle\overline{\tilde{x}}_4$ is given in figure 4.17. Some of the datatypes utilize an exact 0.5 value, used to match the complex QAM symbols x_i, which are scaled to $\pm 0.5 \pm \mathrm{j}\, 0.5$ before converting to SORN representation. The influence of this exact value can be seen in the scatter plots from figure 4.16, where for all datatypes with an exact 0.5 results $\overline{\tilde{x}}_4$ with a correct real and/or imaginary part are visible. The higher the SORN bitwidth, and therefore the precision, the more such results appear. In addition, an increasing amount of results with a correct angle $\angle\overline{\tilde{x}}_4$ but incorrect absolute value $|\overline{\tilde{x}}_4|$ are visible.
This latter observation is even better visible when considering the heuristic of the angles $\angle\overline{\tilde{x}}_4$ from figure 4.17. Here the increase of angles within the decision boundaries, as well as with a correct angle $\angle x_4 = \frac{3\pi}{4}$ can be observed for an increasing SORN bitwidth. In detail, the percentage of cases where the angle $\angle\overline{\tilde{x}}_4$ lies within the decision boundaries are 73.2% (*lin11*$|^{1}$), 77.9% (*lin13*$|^{2}$), 80.3% (*lin13*$|^{1}_{e1/2}$), 85.6% (*lin15*$|^{2}_{e1/2}$), 85.7% (*lin17*$|^{3}_{e1/2}$) and 84.7% (*lin17*$|^{2}_{e1/2}$). This verifies that the exact 0.5 value within the datatypes improves the results, which, however, might still be insufficient to serve as a stand-alone detector. This will be evaluated in the next section 4.2.7.

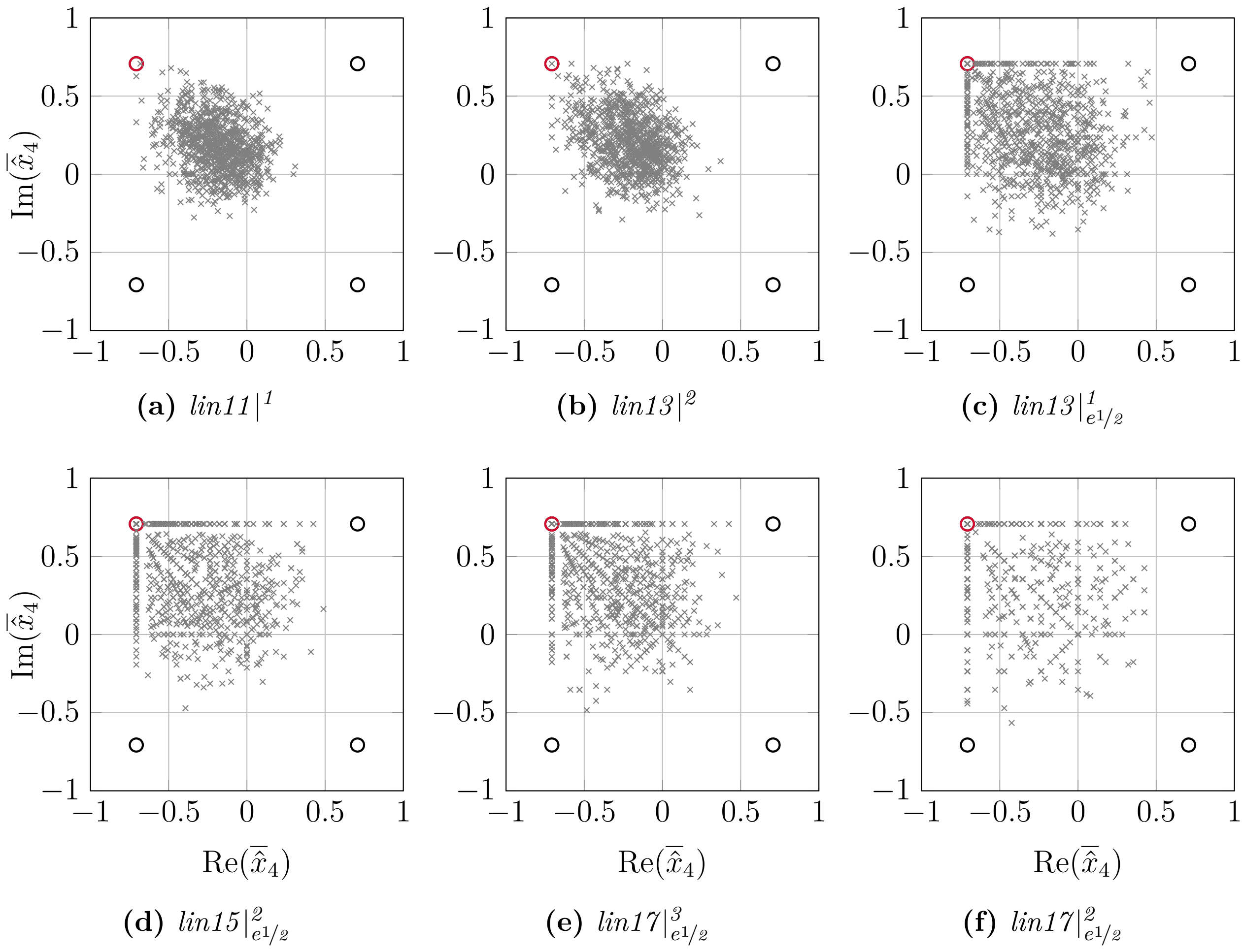


Figure 4.16.: Scatter plot for the arithmetic mean $\overline{\tilde{x}}_4$ of the remaining solutions after SORN preprocessing for 10^3 transmissions of the symbol $x_4 = -1/\sqrt{2} + \mathrm{j}\, 1/\sqrt{2}$, for $N = 4$ with 4-QAM and SNR $= 10\,\mathrm{dB}$, for different SORN datatypes [3].

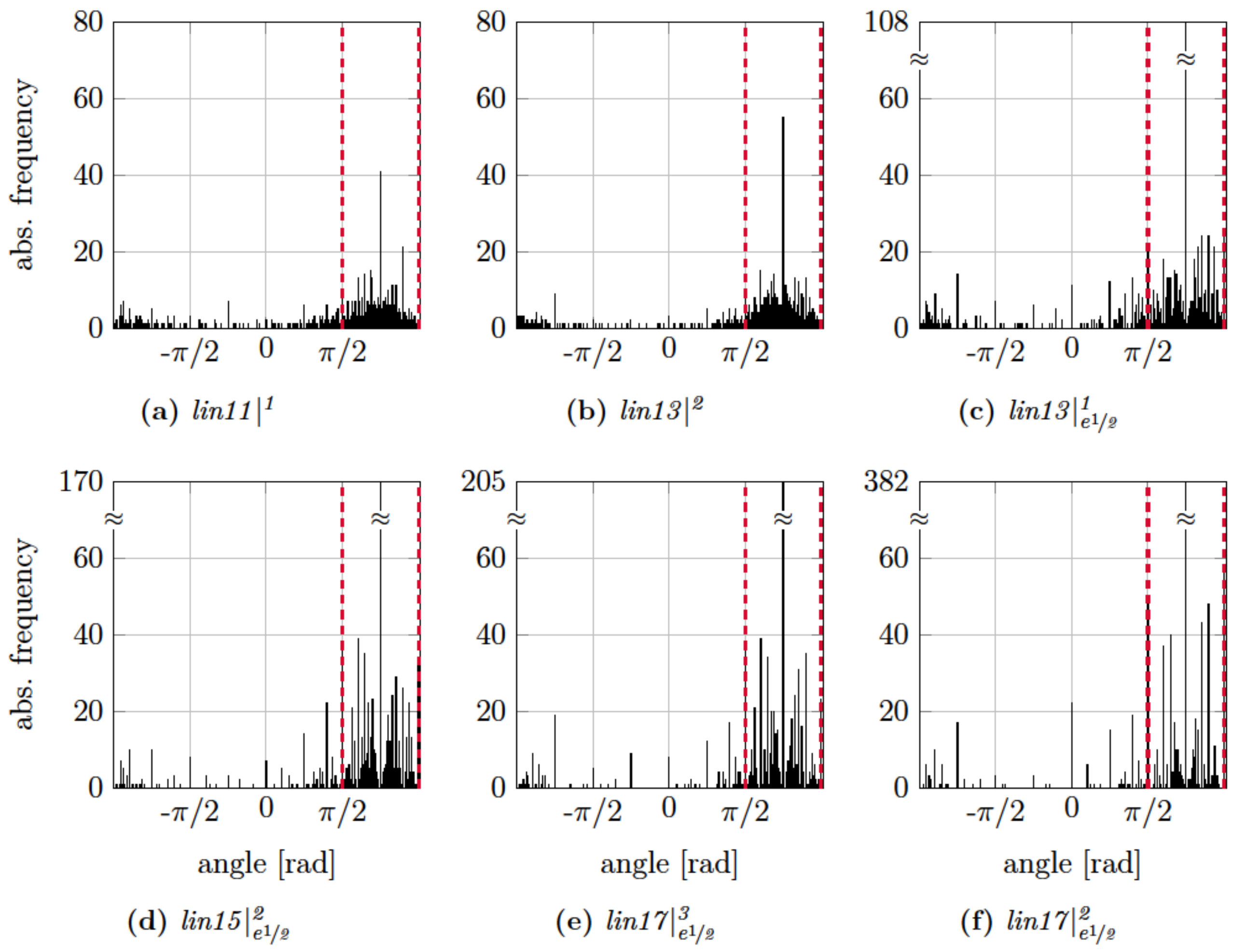
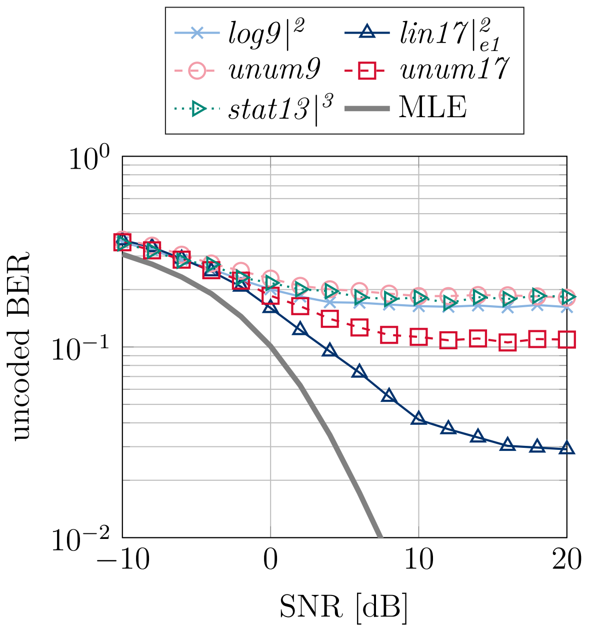


Figure 4.17.: Heuristic for the angle of the arithmetic mean $\angle\overline{\hat{x}}_4$ of the remaining solutions after SORN preprocessing for 10^3 transmissions of the symbol $x_4 = -1/\sqrt{2}+\mathrm{j}\,1/\sqrt{2}$, for $N = 4$ with 4-QAM and SNR = 10 dB, for different SORN datatypes [3].

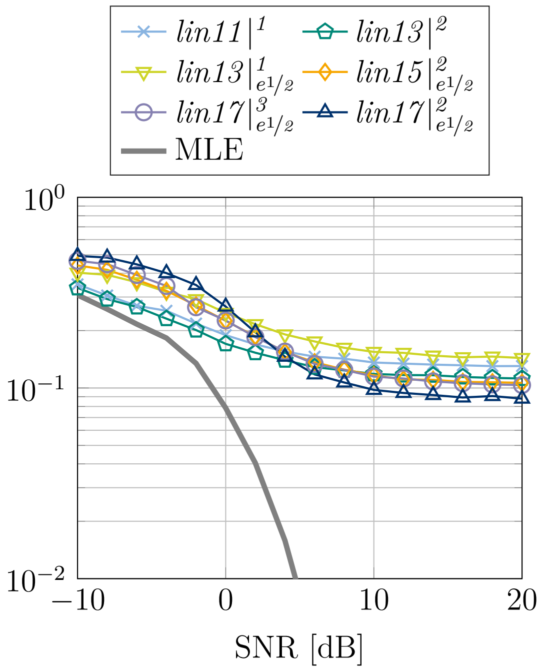
4.2.7. Detection with Reduced Solution Set

After calculating the remaining solutions $\mathcal{R}$ and evaluating their properties in the previous sections, the next step is to compute a single estimated result $\hat{x}$ based upon this preprocessing. In the following, two detection approaches using the preprocessing result are discussed, before two more complex approaches are presented in sections 4.3 and 4.4.

Stand-Alone SORN Detector As already mentioned in the previous section, the most simple solution would be to use the mean of the remaining solutions $\overline{\hat{x}}$ from equation (4.26) as stand-alone solution. In figure 4.18 the bit error rate (BER) for such a detector is shown for a real-valued BPSK and a complex-valued 4-QAM system, both with $N = 4$, and for different SORN datatypes. For comparison, the maximum likelihood estimation (MLE) of a 64 bit floating point SD is given. It is obvious that such a stand-alone SORN detector does not deliver sufficient BER performance, neither for a real-valued nor for a complex-valued system, and independent from the SORN datatype. It is worth mentioning, however, that the linear 17 bit datatype again outperforms the unum-based one by far for the real-valued system and SNR



(a) Real-valued MIMO system with BPSK [4].



(b) Complex-valued MIMO system with 4-QAM [3].

Figure 4.18.: Uncoded BER of the stand-alone SORN preprocessor with result $\bar{\hat{\boldsymbol{x}}}$ for an (a) real-valued and (b) complex-valued MIMO system with $N = 4$, BPSK and 4-QAM, respectively.

values > 0 dB. For the complex-valued system, the 17 bit datatypes show the best performance only for SNR values > 5 dB. A possible reason is the higher probability to discard the ML solution for low SNRs, as discussed in section 4.2.6.
The results for the real-valued system in figure 4.18a again include the statistically determined datatype discussed in section 3.2.4. The results support those from the evaluation in section 4.2.5 and show that this datatype approach does not lead to better results than the linear and logarithmic distributed datatypes.

Sphere Decoder Initialization Since the stand-alone solution was proven insufficient, another approach is to use the SORN solution to obtain an initial radius r for a state-of-the-art SD, which is then processed in a standard format like floating point. For this evaluation, a SD with pruning is considered, which adjusts its radius r throughout the tree-search, as described in section 4.2.1.1. The MLE solution hereby corresponds to a maximum initial radius. For comparison, the evaluation also includes other, manually defined initial radii $r = \{4, 8, 16\}$. The initial radius based on the SORN preprocessing is obtained by computing the squared norm from equation (4.15) with a SORN estimate $\hat{\boldsymbol{x}}$. This SORN estimate is hereby either the mean of the remaining solutions $\bar{\hat{\boldsymbol{x}}}$ according to equation (4.26), or the result of a majority vote among the remaining solutions in $\mathcal{R}$.
Figure 4.19b shows the BER of a SD with the different discussed initial radii for a 4×4 MIMO system with 8-PSK modulation. As a measure for estimating the required computing time of the algorithm, the mean number of visited nodes for one tree-search is given in figure 4.19a. The

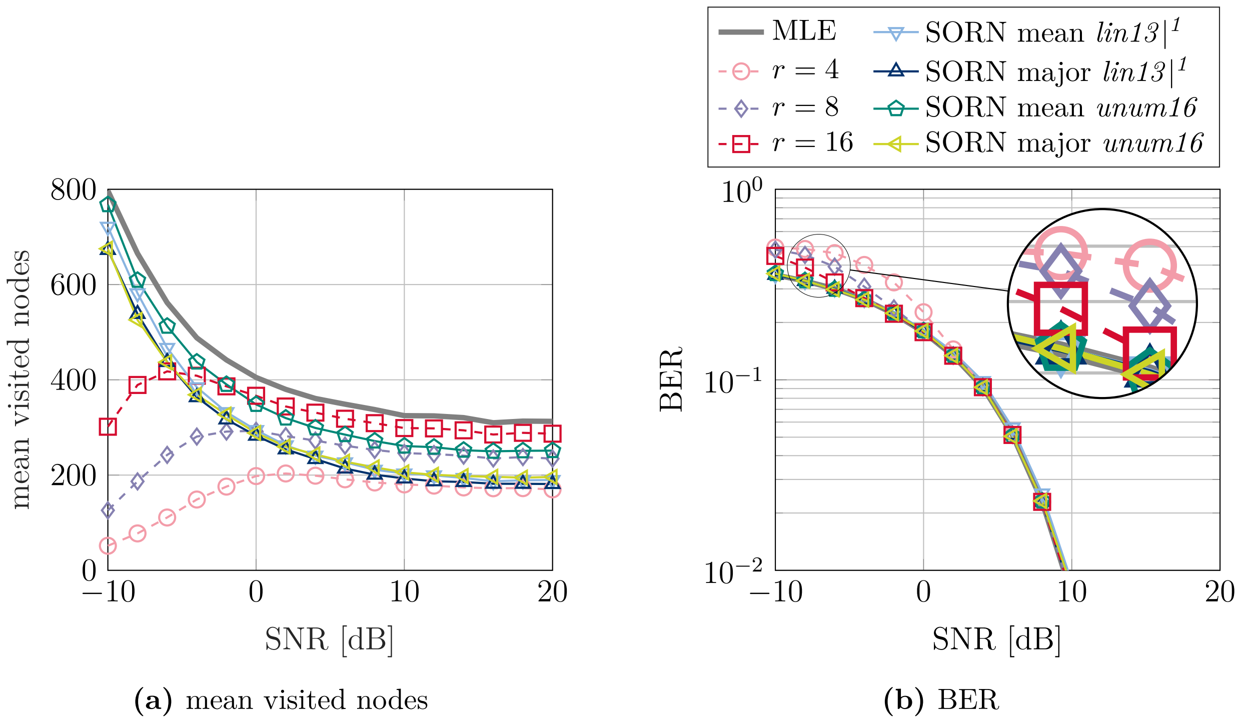


Figure 4.19.: (a) mean visited nodes and (b) BER for a sphere decoder initialized with different radii r for a 4×4 MIMO system with 8-PSK [2, 5]. The MLE solution corresponds to an initial radius $r = \infty$. The radii for the SORN solutions are obtained from the arithmetic mean and majority vote after SORN preprocessing.

first remarkable observation is that all SORN-based SDs achieve a quasi-ML BER-performance, while they reduce the number of visited nodes, compared to the MLE solution. The decoders with manually defined initial radii show a worse BER-performance for low SNRs, while they also show the lowest number of visited nodes in this SNR range. For those SNR values where they achieve a quasi-ML performance, their number of visited nodes is similar or even worse than for the SORN-based decoders. Overall, the SORN-based approach shows the better BER and a similar or better visited-nodes-performance. Comparing the different SORN-based detectors, it can be noted that the majority vote solution outperforms the mean approach, and that the linear datatype shows slightly better results than the unum-based one.
From an implementation point of view, the presented approach can reduce the latency of the SD as it reduces the number of visited nodes. It has to be considered, however, that the required SORN preprocessing also adds a certain computing time. Depending on the implementation strategy, if the preprocessor can run in parallel with the required QRD, the reduced latency can be fully exploited.

4.2.8. Datatype Considerations

Throughout the evaluation of the SORN MIMO preprocessor in the previous sections, different unum-based and half-open linear and logarithmic SORN datatypes from tables 3.4, 3.6 and 3.7 have been considered. The hardware results from section 4.2.4 showed that the unum-based designs have a slightly lower complexity than the linear half-open ones with similar bitwidth. For the other metrics energy, maximum frequency and throughput, however, the half-open

datatypes perform much better than the unum-based ones. On top of that, the algorithmic evaluations from sections 4.2.5 - 4.2.7 not only showed a better performance for the half-open over unum-based datatypes for similar bitwidths, but also for half-open datatypes with much smaller bitwidths. In detail, for the MRS evaluation in section 4.2.5 it was shown that the half-open $\mathit{log9}|^{2}$ datatype achieves the exact same results as the *unum17* datatype which requires almost twice the bitwidth. Together with the hardware evaluation of basic arithmetic components in section 3.4 it can be concluded that the half-open SORN datatypes outperform the unum-based ones in both hardware and algorithmic performance.
Considering the half-open SORN datatypes, the evaluations in section 4.2.6 showed that the introduction of single exact values matching the application data can improve the algorithmic performance. Concerning the difference between linear and logarithmic distributed half-open datatypes, section 4.2.5 showed better results for the linear ones, which, however, might be application specific and not a general conclusion.
In order to evaluate on the SORN interval properties of the different datatypes not only from a result point of view, but also with an emphasis on the arithmetic behavior, in figure 4.20 different interval properties of intermediate results within a 4×4 SORN preprocessor are given for different linear half-open datatypes. In detail, for the architecture from figure 4.10a the intermediate results after the matrix-vector multiplication $\boldsymbol{Hx}$ (figure 4.20a) and after subtraction $\boldsymbol{y} - \boldsymbol{Hx}$ (figure 4.20b) are considered. The analyzed interval properties are

I) intervals with the value $[-\infty, \infty]$, i.e. SORN values with all bits set to 1_2,

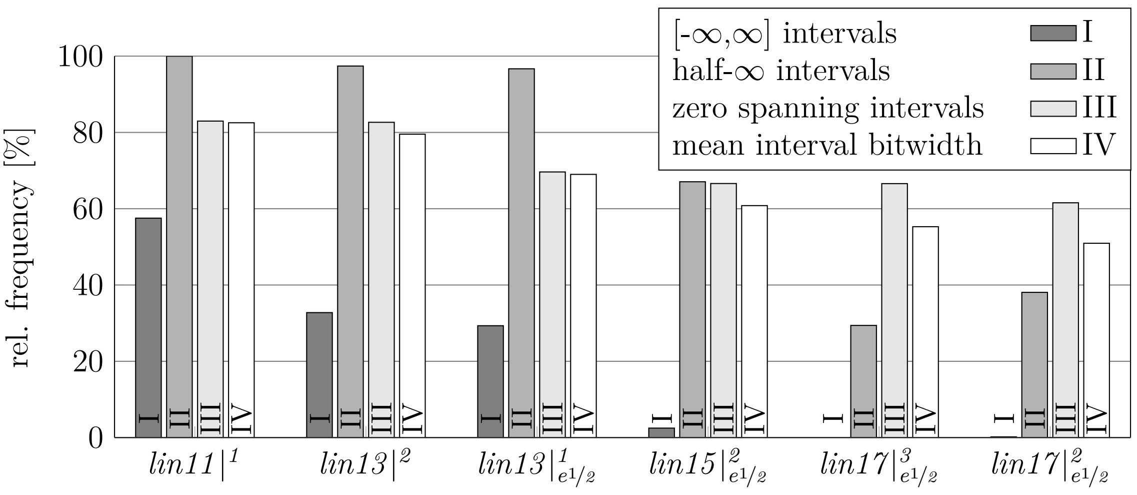
II) intervals with at least one ∞-boundary,

III) intervals that span 0_{10} and cover both negative and positive values, and

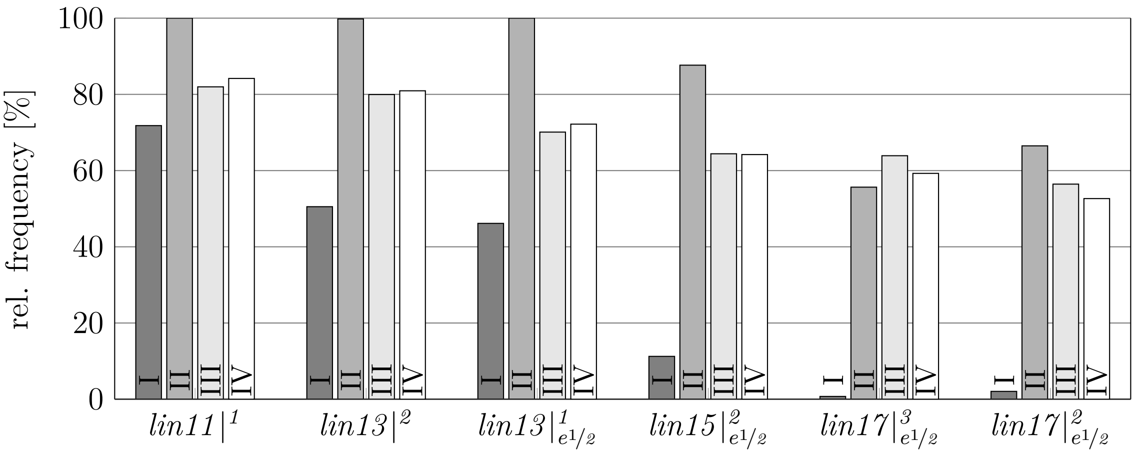
IV) the mean interval bitwidth of the SORN values by means of bits set to 1_2, normalized by the total SORN bitwidth w_s.

For the first two metrics I) and II) it can be observed that the occurrence of $[-\infty, \infty]$ or half-∞ intervals can mainly be reduced by increasing the maximum lattice value of the SORN datatype. The transition from datatypes $\mathit{lin11}|^{1}$ to $\mathit{lin13}|^{2}$, $\mathit{lin13}|^{1}_{e^{1/2}}$ to $\mathit{lin15}|^{2}_{e^{1/2}}$, $\mathit{lin15}|^{2}_{e^{1/2}}$ to $\mathit{lin17}|^{3}_{e^{1/2}}$, and $\mathit{lin17}|^{2}_{e^{1/2}}$ to $\mathit{lin17}|^{3}_{e^{1/2}}$ all increase the maximum lattice value and reduce the occurrence of ∞-intervals. Considering those SORN intervals that span 0_{10} with the third metric III), it can be stated that the occurrence of this interval property can be reduced by increasing the precision of the datatype, either by introducing an exact value, as in the transition from $\mathit{lin13}|^{2}$ to $\mathit{lin13}|^{1}_{e^{1/2}}$, or by representing the same value range with more intervals, as in the transition from $\mathit{lin17}|^{3}_{e^{1/2}}$ to $\mathit{lin17}|^{2}_{e^{1/2}}$. For the mean interval bitwidth as last metric IV), both a larger maximum lattice value, as well as a higher precision improve the mean interval bitwidth. The precision, however, is more important, as can be observed for the 13 and 17 bit datatypes. Here the respective datatypes with a higher precision, not the ones with a larger maximum lattice value show a lower mean interval bitwidth.
From this evaluation it can be concluded that in general datatypes with a higher bitwidth improve arithmetic behavior, no matter if the higher bitwidth datatype introduces a higher dynamic range with a larger maximum lattice value, or a higher precision. For datatypes with the same bitwidth, that improve either the dynamic range or the precision, however, no general conclusion can be drawn. The presented evaluation shows that ∞-intervals can be avoided

with a larger maximum lattice value, whereas zero-spanning intervals and the mean interval bitwidth can be reduced with a higher precision. When returning to the result point of view, for the MRS metric discussed in section 4.2.5, a higher precision is more important to improve the results, as can be seen for the transitions from datatype $lin13|^{2}$ to $lin13|^{1}_{e^{1/2}}$ and $lin17|^{3}_{e^{1/2}}$ to $lin17|^{2}_{e^{1/2}}$ in figure B.2 in the appendix B.4. Regarding the angle of $\bar{\hat{\boldsymbol{x}}}$ which is used as a decision criterion in section 4.2.6, the 13 bit datatype with a higher precision shows the better performance, whereas for the 17 bit datatypes the one with a lager dynamic range leads to better results. Finally, for the BER evaluation in section 4.2.7 and figure 4.18b, the opposite can be observed: here the 13 bit datatype with a higher maximum lattice value shows better results for all SNRs, whereas the 17 bit datatype with a higher precision performs better, but only for high SNR values. In total, for this particular datatype property, no general conclusion can be drawn and the optimal datatype has to be chosen depending on the most important performance metric for the given application.



(a) Stage $\boldsymbol{Hx}$



(b) Stage $\boldsymbol{y} - \boldsymbol{Hx}$

Figure 4.20.: SORN interval properties of different datatypes within the MIMO preprocessor datapath after (a) matrix-vector multiplication $\boldsymbol{Hx}$ and (b) vector subtraction $\boldsymbol{y} - \boldsymbol{Hx}$, respectively, for $N = 4$ and SNR $= 10\,\text{dB}$ [3].

4.3. Hybrid SORN/FxD BPSK Detector

Among the variety of MIMO detectors in the literature, as described in section 4.2.1, the main trade-off is between hardware complexity and performance in terms of BER and data rate/throughput, respectively. The two SORN based-approaches presented in section 4.2.7 mark two ends of this spectrum: the stand-alone SORN detector with a very low complexity but poor BER performance, and the SORN-initialized 64 bit floating point sphere decoder which achieves ML performance. In the following, an intermediate approach is presented for a real-valued 4×4 MIMO system with BPSK modulation ($m = 1$). The single-bit BPSK modulation does not provide very high data rates, but due to its simplicity, leads to a low hardware complexity and robustness against harsh channel conditions.
For this approach, the remaining solutions after SORN preprocessing $\hat{\boldsymbol{x}} \in \mathcal{R}$ are evaluated by a fixed point (FxD) solver with bitwidth $w_f = 16$ in signed $Q_{6.10}$ format. The FxD solver, like the SORN preprocessor, computes the squared norm from the MLE problem (4.12). The number of possible solutions $\boldsymbol{x} \in \mathcal{S}^N$ in the given scenario is $|\mathcal{S}^N| = (2^m)^N = 2^4 = 16$. The SORN preprocessor reduces this number, as described in section 4.2.2 and 4.2.5. The mean reduction is given in % in figure 4.12a for different SORN datatypes and SNRs. The corresponding absolute MRS value $\overline{|\mathcal{R}|}$ varies between 4.7 and 5.3 for *unum9*, between 3 and 3.9 for *log9*$|^{2}$, between 1.3 and 2.4 for *lin17*$|^{2}_{e1}$, and is nearly constant at 2 for *unum17*. These are mean values, however, and the maximum peak values can be higher. From a hardware point of view, a varying number of remaining solutions $|\mathcal{R}|$ per detection leads to a non-deterministic runtime for an iterative implementation, or to a high complexity with a highly unbalanced circuit utilization for a parallel implementation. In figure 4.21 the BER performance for computing a dedicated number of solutions $\hat{\boldsymbol{x}} \in \mathcal{R}_{\mathrm{FxD}} \subseteq \mathcal{R}$ after SORN preprocessing with a FxD solver is given for $|\mathcal{R}_{\mathrm{FxD}}| = \{2, \ldots, 8\}$ and different SORN datatypes over the mean received SNR. The graphs also include the maximum likelihood estimation, obtained from a 64 bit floating point sphere decoder with maximum radius, and a solution for $|\mathcal{R}_{\mathrm{FxD}}| = |\mathcal{R}|$.
The best BER for processing the maximum number of remaining solutions $|\mathcal{R}_{\mathrm{FxD}}| = |\mathcal{R}|$ is achieved for the 9 bit datatypes. The reason is that these datatypes show a higher number of remaining solutions compared to the 17 bit ones, as shown in figure 4.12a. According to the evaluations from section 4.2.6, datatypes with higher MRS values also have a higher probability of including the ML solution, which leads to a better BER performance. When considering a smaller, fixed number of remaining solutions, every increase of $|\mathcal{R}_{\mathrm{FxD}}|$ leads to a better BER performance for the 9 bit datatypes, whereas for the 17 bit datatypes a saturation at $|\mathcal{R}_{\mathrm{FxD}}| = 6$ for *unum17* and $|\mathcal{R}_{\mathrm{FxD}}| = 4$ for *lin17*$|^{2}_{e1}$ can be observed. Also worth mentioning is that with a minimum number of $|\mathcal{R}_{\mathrm{FxD}}| = 2$ a near-optimum performance can be achieved for the *lin17*$|^{2}_{e1}$ datatype. Near-optimum is hereby related to the $|\mathcal{R}_{\mathrm{FxD}}| = |\mathcal{R}|$ rather than the ML solution, which, however, are quite close for low SNRs. This evaluation shows that a detector which computes solely two remaining solutions after *lin17*$|^{2}_{e1}$ SORN preprocessing with 16 bit fixed point can achieve a reasonable BER performance, which is much better than for the stand-alone SORN preprocessor with the same datatype, shown in figure 4.18a. The hardware implementation for this approach will be evaluated in the following.

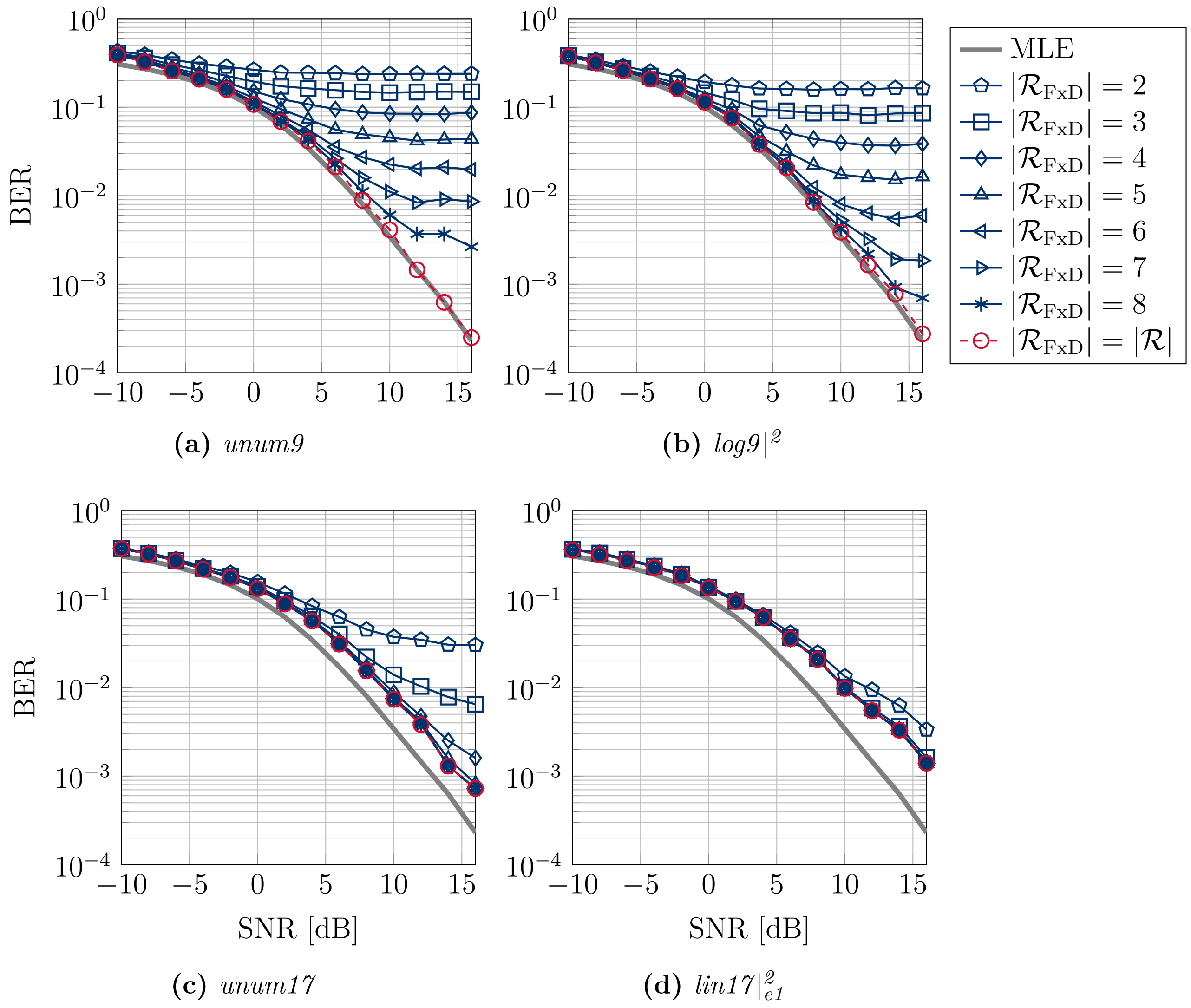


Figure 4.21.: BER for a detector applying the respective SORN datatype (a) - (d) and considering $|\mathcal{R}_{\text{FxD}}|$ of the remaining solutions for FxD processing, for a real-valued 4×4 MIMO system with BPSK modulation [4].

4.3.1. Hardware Architecture

A simplified block diagram for the hardware architecture of the proposed detector is given in figure 4.22. The two main components are the SORN preprocessor and the FxD solver, both implemented in a tree structure according to figure 4.10b. The SORN preprocessor has one pipeline stage, for the FxD solver different designs without and with one pipeline stage are implemented. The SORN bitwidth is $w_s = 17$, the FxD bitwidth is $w_f = 16$, formatted as signed $Q_{6.10}$ format. All in- and outputs are also $Q_{6.10}$ formatted. The design uses two different clock signals, realized with a clock divider that creates a SORN clock which is 4 times faster than the FxD/control clock. In figure 4.22 the fast clock is indicated with a star symbol $\star$. The control path is implemented with a finite-state machine (FSM), executing the following states:

1. In the first state, the inputs are converted from FxD to SORN format with a simple, single-input LUT, realized with logic gates.

2. In the second state the preprocessor is executed for all $2^N = 16$ possible solutions $\boldsymbol{x}$, and the results are written to the register file. Both happens at SORN frequency, corresponding to in total 4 clock cycles at control frequency.

3. The third state determines the two remaining solutions by comparing all results with the threshold interval. If more than two solutions fulfill the threshold condition, the first two are selected for FxD processing.

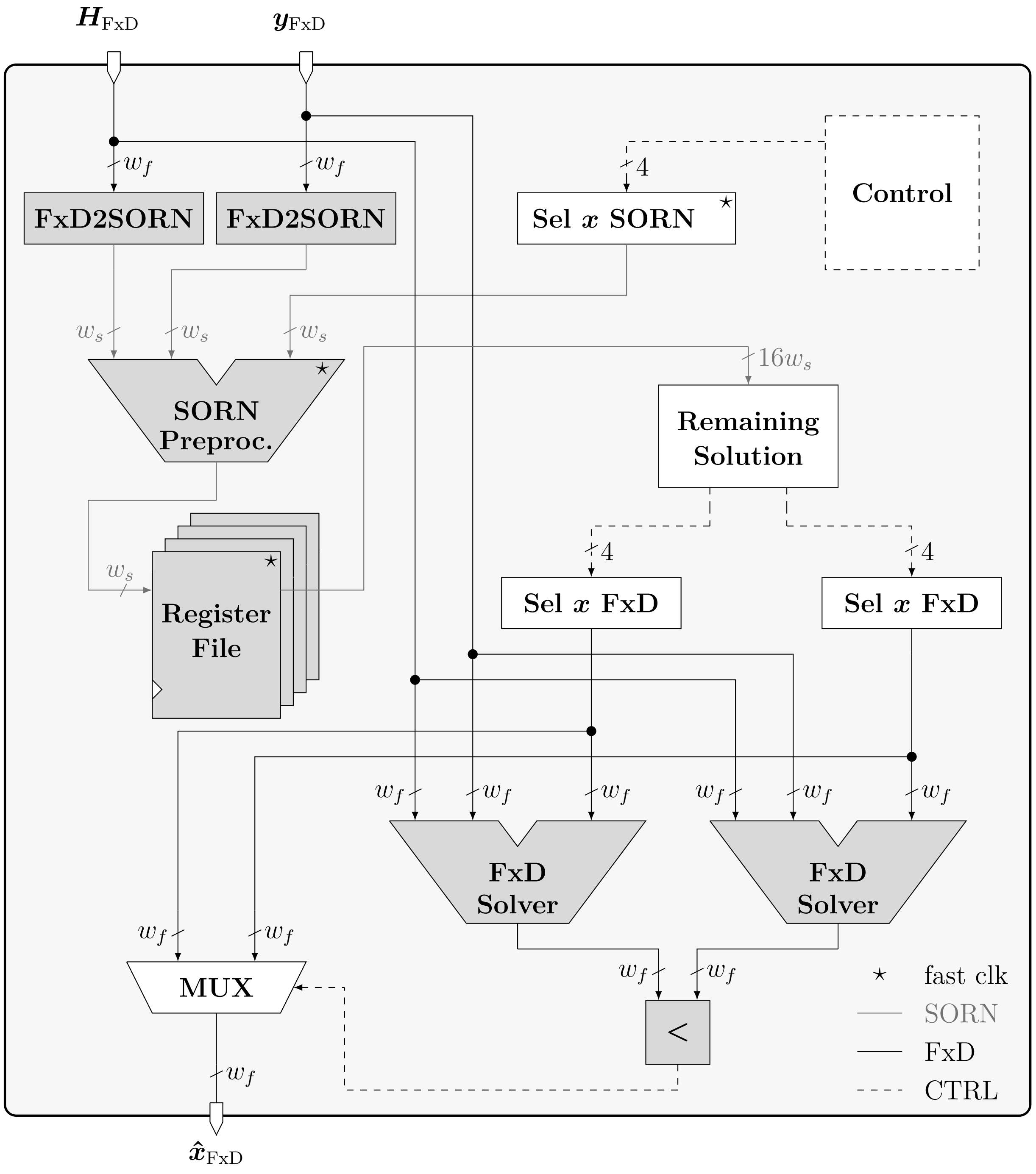


Figure 4.22.: Datapath structure (simplified) of the proposed hybrid SORN/FxD BPSK detector for a real-valued 4×4 MIMO system. The SORN preprocessor uses a faster clock signal than the FxD and control path.

4. In the fourth state, the two determined remaining solutions are processed with the FxD solvers and the results are compared to determine the final output.

For the case $|\mathcal{R}| = 1$, the respective result in FxD format is directly passed to the output. The architecture requires in total 7 clock cycles at control frequency if the FxD solver is implemented without a pipeline stage, one for each of the states 1, 3 and 4, and 4 cycles for the preprocessor state 2.
In addition to the described architecture, an alternative design with one FxD solver is implemented, which computes both remaining solutions iteratively and contains an additional registers file. Due to the removed second FxD solver, this design is less complex, but requires one more clock cycle. For comparison with an all-FxD approach, two additional detectors are implemented, composed of 4 and 2 FxD solvers, respectively, which process all 16 possible solutions iteratively in fixed point only. All intermediate results are stored in a register file and compared in order to determine the smallest norm.

4.3.2. Synthesis Results

The results for a 90 nm CMOS technology synthesis of the two different hybrid SORN/FxD detector designs are presented in table 4.4. The first design (a) is the one depicted in figure 4.22 with the SORN preprocessor and two FxD solvers, the second design (b) is composed of the SORN preprocessor and one FxD solver. For both designs, different frequency and FxD pipeline stage combinations are implemented, labeled with S1 - S6. The results for the all-FxD designs are given in table 4.5. Design (c) is composed of four parallel FxD solvers, design (d) contains two. All FxD solvers are implemented with one pipeline stage and for different frequencies, labeled with F1 - F4.

Hybrid SORN/FxD Detector The different frequency/pipeline configurations lead to different runtimes, which are given as number of required clock cycles at the respective lower control frequency, and in nanoseconds. As mentioned in the previous section, the architecture with one FxD solver (b) requires one more cycle than the one with two FxD solvers (a), resulting in an average runtime gain of 11.6% for approach (a). Considering the total area, which is given in μm^2 and kGE, approach (b) requires approximately 28.9% less area on average, compared to approach (a). Since (a) utilizes more area than (b), also the power consumption is higher, as well as the APT product which measures the trade-off between all three parameters area, power consumption and runtime, and is given in [$\mu m^2 \times \mu W \times ms$].
Considering the area distribution of the proposed detectors, a general outcome of this evaluation is that the SORN preprocessor shows a lower complexity than the FxD solver. For approach (a) with two FxD solvers, each requires 27.8% to 30.2% of the total area, while the SORN preprocessor requires between 20.2% and 24.9%, depending on the frequency and pipeline configuration. For approach (b), the SORN preprocessor requires between 28.3% and 34.0% of the total area, while the FxD solver requires between 38.7% and 43.6%.
The area demands of the component for FxD to SORN conversion are independent from the configuration. Over all different configurations for approach (a) and (b), the (in total 20) conversion components together require an area of 2800 μm^2, which is a part of 3.0% to 4.5% of the respective total area.

Table 4.4.: Synthesis results for the hybrid SORN/FxD BPSK detector with 1 SORN preprocessor and (a) 2 or (b) 1 FxD solver for 90 nm CMOS technology, implemented for the $lin17|^{2}_{e1}$ SORN datatype and a signed $Q_{6.10}$ format [4].

parameter		(a) **SORN + 2 FxD solvers**			(b) **SORN + 1 FxD solver**		
config label		S1	S2	S3	S4	S5	S6
freq [MHz]	FxD+CTRL	200	200	250	200	200	250
	SORN	800	800	1000	800	800	1000
pipeline stages	FxD	0	1	1	0	1	1
runtime	[cycles]	7	8	8	8	9	9
	[ns]	35	40	32	40	45	36
total area	[µm^2]	93337	87576	93377	64958	62372	67695
	[kGE]	33070	31029	33084	23015	22099	23985
SORN area$^{(a)}$	[µm^2]	18871	18276	23211	18393	18225	22992
	[%]	20.2	20.9	24.9	28.3	29.2	34.0
FxD area$^{(a)}$	[µm^2]	28225	25608	25985	28296	25793	26215
	[%]	30.2	29.2	27.8	43.6	41.4	38.7
power	[µW]	7997	8703	10683	7686	8060	9873
APT product	[µm^2×µW× ms]	26.1	30.5	31.9	20.0	22.6	24.1

$^{(a)}$area of the SORN preprocessor or one single FxD solver

Comparison to all-FxD Design When comparing the two SORN-based designs (a) and (b) with the all-FxD designs (c) and (d), it can be observed that the best runtime is achieved for design F2, but this also comes with the highest complexity. The design with the lowest overall complexity is the SORN design S5, which is also faster than the lowest complex FxD design F3. Architecture F3 also shows the lowest power consumption, while S3 shows the highest. In general it can be stated that the hybrid SORN/FxD approach achieves hardware results on a similar level to the all-FxD approach. When considering the APT product as a combined measure, the SORN design S4 even achieves the best overall performance while still being 20% faster and nearly equally complex than the all-FxD design with lowest APT product F3.

Table 4.5.: Synthesis results for the reference FxD BPSK detector with (c) 4 or (d) 2 FxD solvers for 90 nm CMOS technology, implemented with a signed $Q_{6.10}$ format [4].

parameter		(c) **4 FxD solvers**		(d) **2 FxD solvers**	
config label		F1	F2	F3	F4
freq	[MHz]	200	250	200	250
runtime	[cycles]	6	6	10	10
	[ns]	30	24	50	40
total area	[µm^2]	111933	117414	64603	67603
	[kGE]	39659	41601	22889	23952
FxD area[a]	[µm^2]	24329	25695	24754	26241
	[%]	21.7	21.9	38.3	38.8
power	[µW]	8812	10215	6381	8282
APT product	[µm^2×µW×ms]	29.6	28.8	20.6	22.4

[a] area of one single FxD solver

4.4. SORN Sphere Decoder

The hybrid SORN/FxD BPSK detector discussed in the previous section is an example for a low-complex and sub-optimal detector, which does not achieve ML performance. In contrast, the standard sphere decoder algorithm is able to achieve this ML performance, but at the cost of a higher complexity. From a hardware point of view, especially the latency is challenging for a SD implementation, since it is SNR depended an non-deterministic. In order to improve the suitability of the algorithm for hardware implementation, there exist various approaches to adapt the standard SD towards a lower and deterministic latency. Some of these approaches will be discussed in section 4.4.5. The SD initialized with a SORN preprocessing result, presented in section 4.2.7, can also be seen as one of these approaches, as it was shown to reduce the number of visited nodes, which is a measure for the latency of the algorithm. In the following, the SORN SD approach is presented, which also utilizes the preprocessing introduced in section 4.2 in order to reduce the SD latency, yet not by calculating an initial radius, but by removing dedicated nodes from the search tree. In the following, this reduction, as well as a subsequent permutation of the search tree are described in section 4.4.1. The hardware implementation of the complete SD design with SORN and QRD preprocessing is presented in section 4.4.2, as well as RTL and CMOS synthesis results in sections 4.4.3 and 4.4.4, respectively. Comparisons to state-of-the-art approaches which also target a reduction of the SD latency are discussed in sections 4.4.5 and 4.4.6.

4.4.1. Tree Reduction and Permutation

As introduced in section 4.2.2, the SORN preprocessor results in a reduced solution set $\mathcal{R}$ for the MLE problem (4.12). This set is now used to reduce the complexity of the sphere decoder search tree. As described in section 4.2.1.1 and depicted in figure 4.9, the search tree contains every possible transmit vector $\boldsymbol{x} \in \mathcal{S}^N$, which correspond to the respective bottom level tree nodes. Those nodes corresponding to the solutions $\boldsymbol{x} \notin \mathcal{R}$ which are discarded by the preprocessor can be removed from the tree. Figure 4.23a shows the example of a tree with $N = 3$ and BPSK, where two of the bottom level nodes are removed. The intention of this approach is to reduce the number of visited nodes and therefore the latency of the algorithm. Even though the described SD with pruning does not iterate over all nodes within the tree, a lower number of total nodes can still be expected to lead to a lower number of visited nodes. This behavior is evaluated for a 4×4 and 8×8 complex-valued MIMO system with QPSK

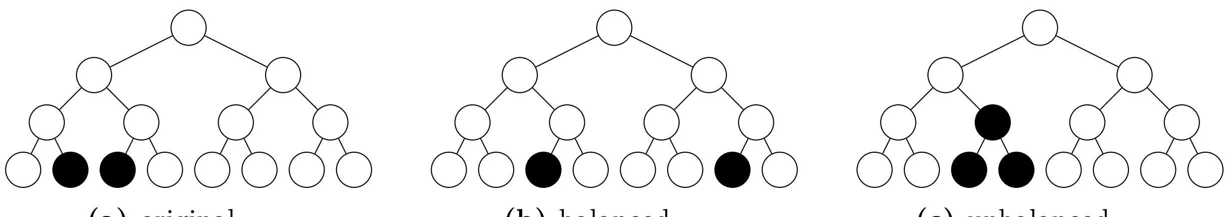
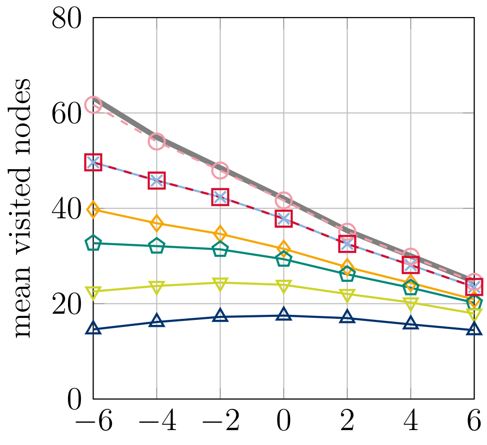

(a) original **(b)** balanced **(c)** unbalanced

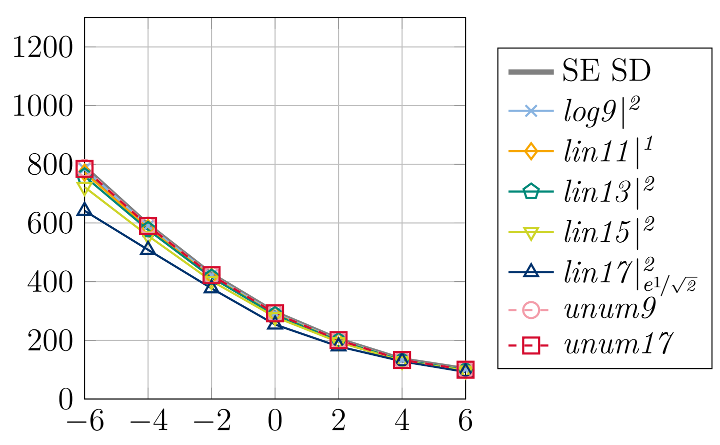
Figure 4.23.: Permutation of the sphere decoder search tree: (a) original tree with deleted nodes (black) resulting from the SORN preprocessing, and permuted tree for a (b) balanced and (c) unbalanced node ratio [6, 7, Kno20].

modulation over the SNR in figures 4.24a and 4.24b, respectively. For the SORN preprocessor different datatypes are applied. It can be observed that the mean number of visited nodes can be drastically reduced by up to 75% for the 4×4 case. This reduction, however, is highly datatype and SNR dependent. For the 8×8 case, the maximum reduction with less than 20% is much lower.
In order to further improve the latency reduction introduced by the deleted nodes, a permutation of the search tree can be applied to fully exploit the reduction. By virtually permuting the order of transmit antennas, the remaining solution set $\mathcal{R}$, as well as the SD search tree are also permuted, the latter by interchanging the tree levels l. Different permutations hereby lead to different shapes of the search trees. In the following, two opposite permutations are evaluated, leading either to a balanced or unbalanced subtree size. The two deleted nodes in the tree without permutation in figure 4.23a are both in the same level-1-subtree, but in different level-2-subtrees. A balanced permutation leads to equally sized subtrees starting from level 1, as shown in figure 4.23b, whereas for the unbalanced permutation in figure 4.23c both deleted nodes are shifted into the same level-2-subtree which facilitates the deletion of one additional node. The corresponding simulation results for a 4×4 and 8×8 QPSK MIMO system with different SORN datatypes for the preprocessor are shown in figures 4.24c and 4.24d for the balanced, and figures 4.24e and 4.24f for the unbalanced permutation, respectively. It can be observed that the balanced permutation leads to a deterioration of the visited nodes, rather than to an improvement, compared to the results without any permutation. For the 4×4 system some and for 8×8 all datatypes lead to a performance which is even worse than for the standard SD without any deleted nodes. The unbalanced permutation, on the other hand, further improves the results with no permutation leading to a reduction of the visited nodes by up 82% for 4×4 and up to 55% for the 8×8 case.

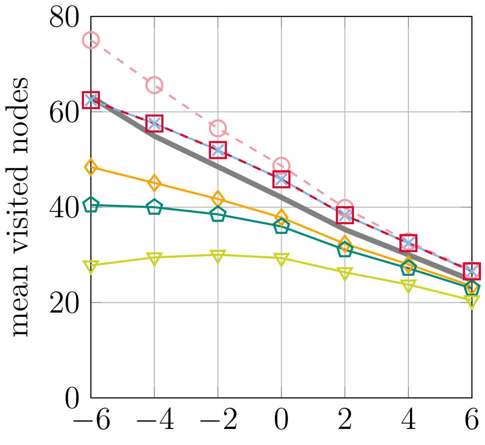
Determining the Permutation The balanced and unbalanced permutation used for the evaluations in figure 4.24 are determined by a sorting algorithm. This algorithm computes the permutation order $\boldsymbol{p} \in \mathbb{N}^N$, which gives the new virtual order of the transmit antennas. The main approach is to count the number of remaining solutions in every branch per tree level, and to compute the standard deviation of this value as a measure for the subtree size. By iterating over different permutations, this standard deviation is either minimized for a balanced or maximized for an unbalanced ratio. The computation of the standard deviation is hereby simplified in order to reduce the complexity for hardware implementation. The algorithm is given in code A.1 in appendix A.3. Since this is not a main contribution of the author, the approach is not discussed in more detail here. A detailed derivation and further explanations can be found in [6] and [Kno20].
In the given references, also an approximate version of the sorting algorithm is presented, which further reduces the complexity with respect to its hardware implementation. In [6] it is shown that this approximate permutation leads to a similar visited nodes performance as the exact algorithm. Due to its simplicity, this approximate algorithm, which is given in code A.2 in appendix A.3, is used for the hardware implementation presented in the following section 4.4.2.



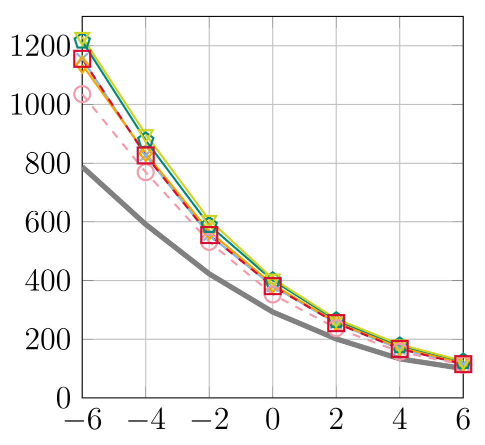
(a) no permutation 4×4



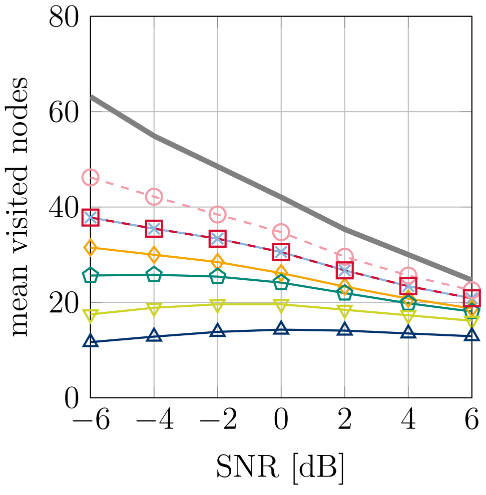
(b) no permutation 8×8



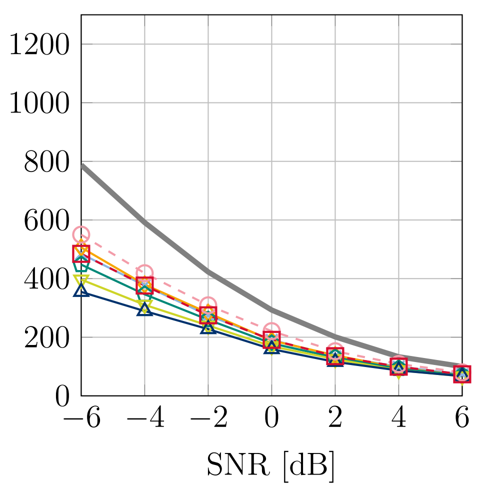
(c) bal. permutation 4×4



(d) bal. permutation 8×8



(e) unbal. permutation 4×4



(f) unbal. permutation 8×8

Figure 4.24.: Mean visited nodes of a standard Schnorr-Euchner (SE)-SD, and a SE SD with deleted nodes after SORN preprocessing, without and with balanced and unbalanced permutation, for a 4×4 and 8×8 complex-valued MIMO system with QPSK [6, 7, Kno20].

4.4.2. Implementation

In the following, the RTL implementation of the complete SORN SD approach is discussed for a 4×4 MIMO system with QPSK modulation. The top level architecture is shown in figure 4.25 for a 16 bit FxD datapath. The design consists of five main modules:

- **SORN:** A module computing the SORN preprocessing as described in section 4.2.3.
- **SORT:** The implementation of the approximate sorting algorithm to compute the unbalanced permutation for the SD search tree, as described in section 4.4.1. The permutation is applied to the channel matrix $\boldsymbol{H}$ and the SORN preprocessing result, as well as to inverse the permutation of the final SD result.
- **QRD:** A QR decomposition of the permuted channel matrix $\boldsymbol{H}$, required as preprocessing for the sphere decoder, as described in sections 4.2.1.1 and A.1.
- **MVM:** The implementation of the matrix-vector-multiplication (MVM) $\tilde{\boldsymbol{y}} = \boldsymbol{Q}^{\boldsymbol{H}}\boldsymbol{y}$ from equation (4.14), required for the sphere decoder, as described in section 4.2.1.1.
- **SD:** The implementation of a sphere decoder with a reduced search tree, according to the SORN preprocessing result. The inputs $\tilde{\boldsymbol{y}}$ and $\boldsymbol{R}$ result from the permuted and decomposed channel matrix $\boldsymbol{H}$.

In addition to the five main modules, the design contains three further submodules to apply the permutation to the channel matrix $\boldsymbol{H}$ and the SORN result, and the inverse permutation to the SD result $\hat{\boldsymbol{x}}$. The SORN preprocessing and the sorting algorithm are implemented for a frequency of 1 GHz. A frequency divider is used to create a second clock of 100 MHz, which drives the QRD, MVM and SD modules. The detailed implementation of the five main modules is described in the following. In order to provide a comparison to a state-of-the-art design, also a standard SE SD is implemented together with the required QRD, MVM preprocessing, but without a reduced search tree and a permutation. The differences between both SD implementations are discussed in the respective paragraph.

SORN Preprocessor The architecture of the SORN preprocessor is shown in figure 4.26. This module computes the squared norm $\|\boldsymbol{y} - \boldsymbol{H}\boldsymbol{x}\|_2^2$ according to equation (4.12) for every possible symbol vector $\boldsymbol{x} \in \mathcal{S}^N$ in SORN format, as described in section 4.2.2. For the given 4×4 case with QPSK, the number of possible solutions is $|\mathcal{S}|^N = (2^m)^N = 256$. The implemented preprocessor contains two such SORN solvers to compute the norm in parallel. Both solvers are implemented as described in section 4.2.3 with three pipeline stages for the different SORN datatypes $lin9|^1$, $lin11|^1$, $lin13|^2$, $lin15|^2$ and $lin17|^2_{e1/\sqrt{2}}$. Before fed into the SORN solvers, the FxD inputs $\boldsymbol{H}$ and $\boldsymbol{y}$ are converted to the respective SORN representation. The possible solutions $\boldsymbol{x}$ are selected according to a submodule which counts the respective indexes. All calculated norms are stored in a register file and evaluated by another submodule, which determines the reduced solution set $\mathcal{R}$, as described in section 4.2.2. The output of the preprocessor is a 256 bit signal where each bit represents the inclusion (1_2) or exclusion (0_2) of a respective solution $\boldsymbol{x}$ in $\mathcal{R}$.
The number of required clock cycles for the SORN preprocessor can be determined from the

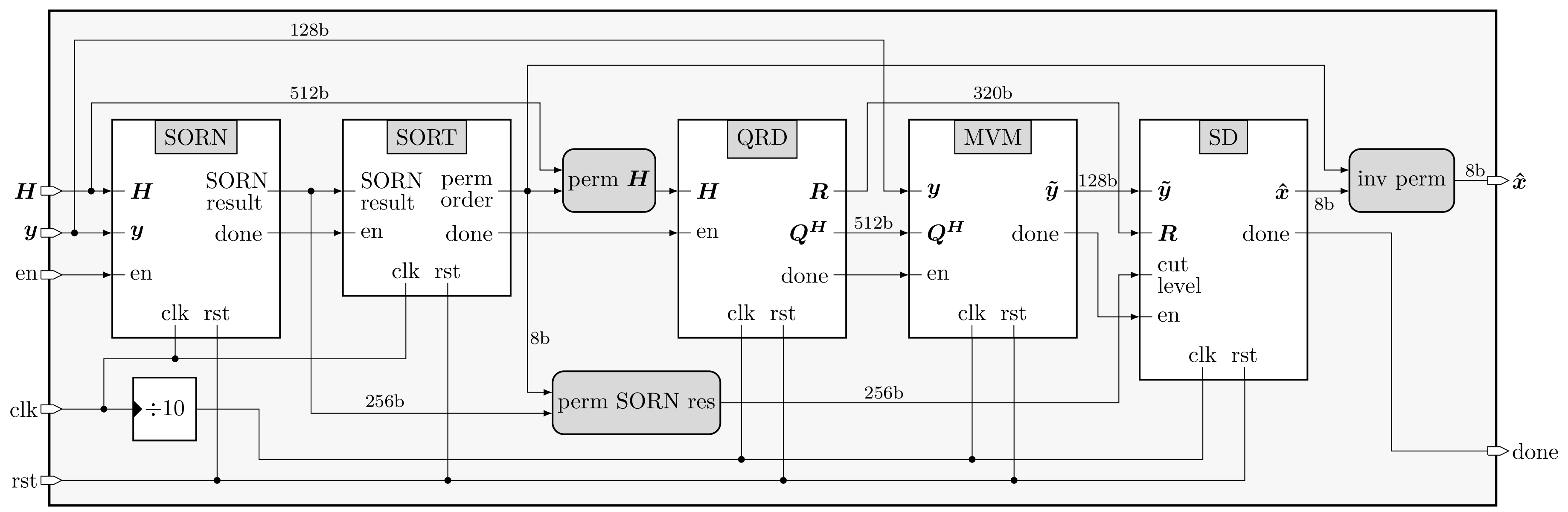


Figure 4.25.: Top level architecture of the proposed SORN SD for 4×4 MIMO and a 16 bit FxD datapath: The SORN preprocessor (SORN) and sorting unit (SORT) are running with the fast clock signal, the other modules QRD, matrix-vector-multiplication (MVM) and SD with a 10× lower clock frequency, provided by the frequency divider in the top level design [7].

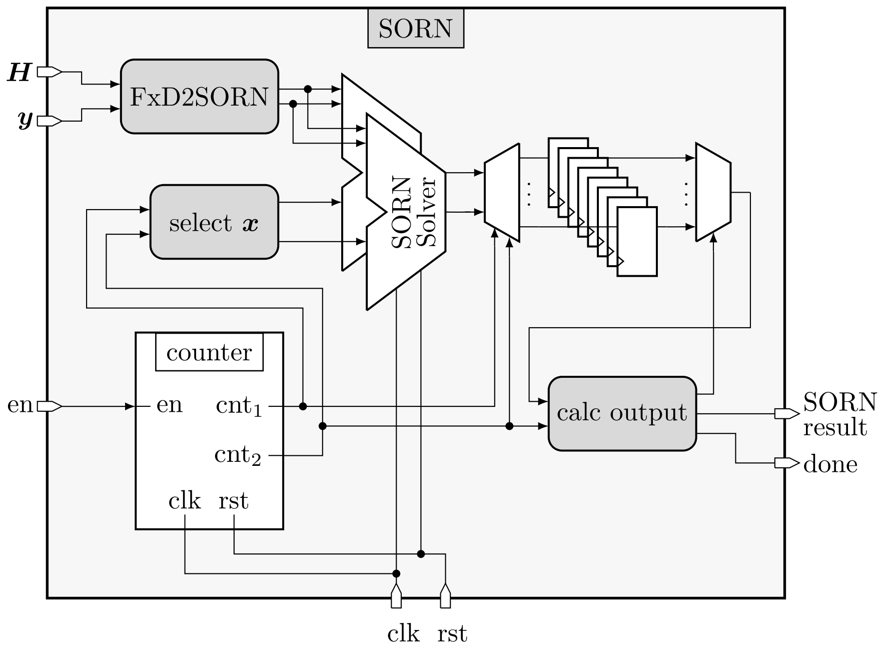


Figure 4.26.: SORN preprocessing unit performing an exhaustive search for the MLE problem (4.12) with two parallel SORN solvers to determine the remaining solutions [7].

256 possible solutions processed with two parallel SORN solvers with three pipeline stages. This leads to 131 cycles at SORN frequency 1 GHz, corresponding to $C_{\text{SORN}} = 13.1$ cycles at 100 MHz.

Sorting Module The sorting module shown in figure 4.27 implements the approximate sorting algorithm given in code A.2 in appendix A.3, as described in section 4.4.1. The output gives the permutation order to achieve an unbalanced node ratio in the reduced SD tree. The input of the sorting module is the 256 bit signal from the preprocessor, where each bit represents the in- or exclusion of one symbol vector $\boldsymbol{x}$. This is shown in figure 4.28 with all 256 possible symbol vectors in the first four rows and the SORN preprocessing result in the bottom row. For the sorting process, the preprocessing result for one QPSK symbol per row 1 - 4 is mapped to a 64 bit signal. In figure 4.28 this is shown for the symbol $\frac{1}{\sqrt{2}}(1 + \text{j})$. For the first row, every fourth bit of the SORN result is mapped, for the second row the first four bits, then bits 17 to 20, and so on. For the last row, the first 64 bits are used. This results in 16 different combinations for the 64 bit mapped signal, which are controlled by an FSM. The mapped signal is passed to a counter which determines the number of 1_2 bits in the signal. This value is then added to the result from the previous iteration, or to 0_{10} for the first iteration. The sum is squared and passed to the feedback loop. After every fourth iteration the accumulated result corresponds to the approximated standard deviation of one row. This value is passed to the last block which sorts the calculated standard deviations per row in descending order to obtain the final permutation.
The sorting module requires in total 19 clock cycles at SORN frequency 1 GHz, corresponding to $C_{\text{SORT}} = 1.9$ cycles at 100 MHz. This number includes 16 cycles for the different mapping combinations, one for initializing the FSM and two for the final sorting process.

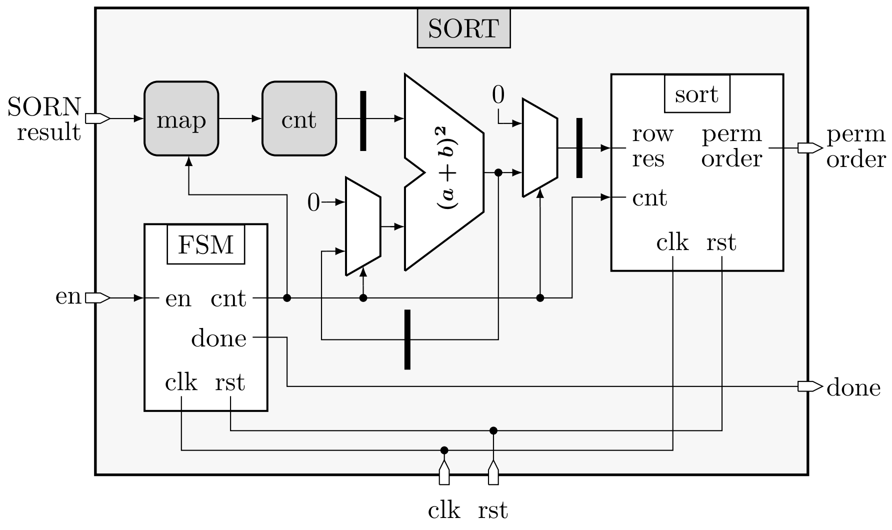


Figure 4.27.: Sorting unit calculating the permutation order for the channel matrix and an unbalanced SD search tree based on the SORN preprocessing [7].

QR decomposition The QRD module is used to decompose the channel matrix $\boldsymbol{H}$ into an orthogonal matrix $\boldsymbol{Q} \in \mathbb{C}^{N\times N}$ and an upper triangular matrix $\boldsymbol{R} \in \mathbb{C}^{N\times N}$, as described in the appendix A.1. The applied method is a complex Givens Rotation which successively generates zero-elements below the main diagonal of the input matrix $\boldsymbol{H}$ by multiplying with a complex rotation matrix. The QRD is implemented iteratively and produces one zero entry per iteration. For the given 4×4 system size this leads to 6 global iterations. In every iteration the elements of the rotation matrix c and s are calculated according to equation (A.2), before the rotation is applied. The required inverse square root is implemented with an iterative Newton-Raphson algorithm with three iterations, as described in section 2.2.2. The Givens Rotation is realized by computing the (sub-)matrix multiplication shown in equation A.1. Therefore two different submodule types are implemented, realizing either one of the equations for R'_{ik} and R'_{jk}:

$$\begin{aligned} R'_{ik} &= c\,R_{ik} + s\,R_{jk} \\ R'_{jk} &= -s^*\,R_{ik} + c^*\,R_{jk} \end{aligned} \quad \text{for } k \in \{i,\dots,N\}, \quad i < j \tag{4.27}$$

In total 11 such submodules and two additional complex multiplications are required to compute all new entries of $\boldsymbol{Q}$ and $\boldsymbol{R}$ per iteration in parallel, as well as a submodule to compute c and

$$\frac{1}{\sqrt{2}} \left[\begin{array}{cccccccccccccc} 1+\mathrm{j} & -1+\mathrm{j} & -1-\mathrm{j} & 1-\mathrm{j} & 1+\mathrm{j} & \cdots & 1-\mathrm{j} & 1+\mathrm{j} & \cdots & 1-\mathrm{j} & 1+\mathrm{j} & \cdots & 1-\mathrm{j} \\ 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & -1+\mathrm{j} & \cdots & 1-\mathrm{j} & 1+\mathrm{j} & \cdots & 1-\mathrm{j} & 1+\mathrm{j} & \cdots & 1-\mathrm{j} \\ 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & \cdots & 1+\mathrm{j} & -1+\mathrm{j} & \cdots & 1-\mathrm{j} & 1+\mathrm{j} & \cdots & 1-\mathrm{j} \\ 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & 1+\mathrm{j} & \cdots & 1+\mathrm{j} & 1+\mathrm{j} & \cdots & 1+\mathrm{j} & -1+\mathrm{j} & \cdots & 1-\mathrm{j} \\ \hline 1 & 0 & 1 & 0 & 0 & \cdots & 0 & 1 & \cdots & 0 & 1 & \cdots & 0 \end{array}\right]$$

Figure 4.28.: Representation of the SORN preprocessing result (bottom row) and the corresponding symbol vectors $\boldsymbol{x}$ in rows 1 - 4. Per row, the SORN result bits for the symbol $\frac{1}{\sqrt{2}}(1+j)$ are mapped to the next stage [7].

s. In addition, registers to store all intermediate values of c, s, $\boldsymbol{Q}$ and $\boldsymbol{R}$ are implemented. Each of the 6 global iterations requires 16 clock cycles, 14 to generate the rotation parameters and 2 to apply the rotation. One additional cycle is required for initialization, resulting in a total number of $C_{\text{QRD}} = 97$ clock cycles at 100 MHz.

Matrix-Vector-Multiplication The MVM module implements the matrix-vector-multiplication $\tilde{\boldsymbol{y}} = \boldsymbol{Q}^{\boldsymbol{H}}\boldsymbol{y}$ from equation (4.14) using 16 complex multiplications and 14 complex additions/subtractions. These arithmetic blocks are organized in a tree structure with one pipeline stage. Since the four rows of the output vector are calculated after each other, together with the pipeline stage in total $C_{\text{MVM}} = 5$ clock cycles at 100 MHz are required.

Sphere Decoder The dataflow diagram of the implemented standard SE SD and SORN-reduced SD is shown in figure 4.29. The standard SE SD will be described first, then the adaptions for the SORN-based approach will be discussed. The module implements the SE SD algorithm with pruning as described in section 4.2.1.1. The control path is composed of an FSM that manages the transitions between the different nodes and tree levels. A register file contains all required parameters: the current tree level l, the node counter c_{node}, the global radius r and the calculated error metrics $\boldsymbol{e}(1,1)\ldots\boldsymbol{e}(2^m,2^m)$ per node. The datapath is composed of multiplications, additions and a square block in order to compute the error metric per node according to equation (4.19). Beginning from the first level, the error metric is calculated for every node in the current level (and subtree). After processing all of these 2^m errors, they are sorted and compared to the global radius r. The path with the lowest error metric that satisfies $e(c_{\text{node}},l) < r$ is followed. When the bottom level is reached and the final error of the current path fulfills $e(c_{\text{node}},l) < r$, the global radius is adapted. The node counter c_{node} is reset and the algorithm continues with the next node in a higher level, that shows the lowest remaining error metric and fulfills $e(c_{\text{node}},l) < r$, until no such node is left.
For the SORN-reduced SD, an additional component is required to calculate those nodes that can be removed from the search tree. The input of this component is the 256 bit result from the SORN preprocessor, where every bit corresponds to one possible solution vector $\boldsymbol{x}$, equivalent to the bottom level nodes of the search tree. The removed bottom tree level nodes can therefore be taken directly from the preprocessing result. For the removed nodes in higher tree levels, the bits from the preprocessing result are connected by a 2^m-dimensional OR-gate as

$$\mathcal{N}_l(i) = \bigvee_{j=2^m \times i}^{2^m(i+1)-1} \mathcal{N}_{l+1}(j) \tag{4.28}$$

with $\mathcal{N}_l(i)$ being the node at level $l \in \{1,\ldots,N-1\}$ and $i \in \{0,\ldots,(2^m)^l - 1\}$, the bottom level nodes $\mathcal{N}_N(i)$, $\bigvee$ as logical OR and the modulation bitwidth m. Due to the described removal of nodes within the search tree, the presented SD implementation has to be adjusted. In figure 4.29, all adaptions for the SORN-reduced SD are indicated with dashed lines/blocks. The removed nodes lead to a new parameter $c_{\max}$, which gives the number of valid nodes per current tree level (and subtree) and requires an additional register. This value is recalculated every time the algorithm jumps to another level. In addition, the node counter c_{node} can not be simply incremented anymore, but has to be determined according to the removed nodes. Both implemented versions of the SD require one clock cycle to initialize the controlling FSM,

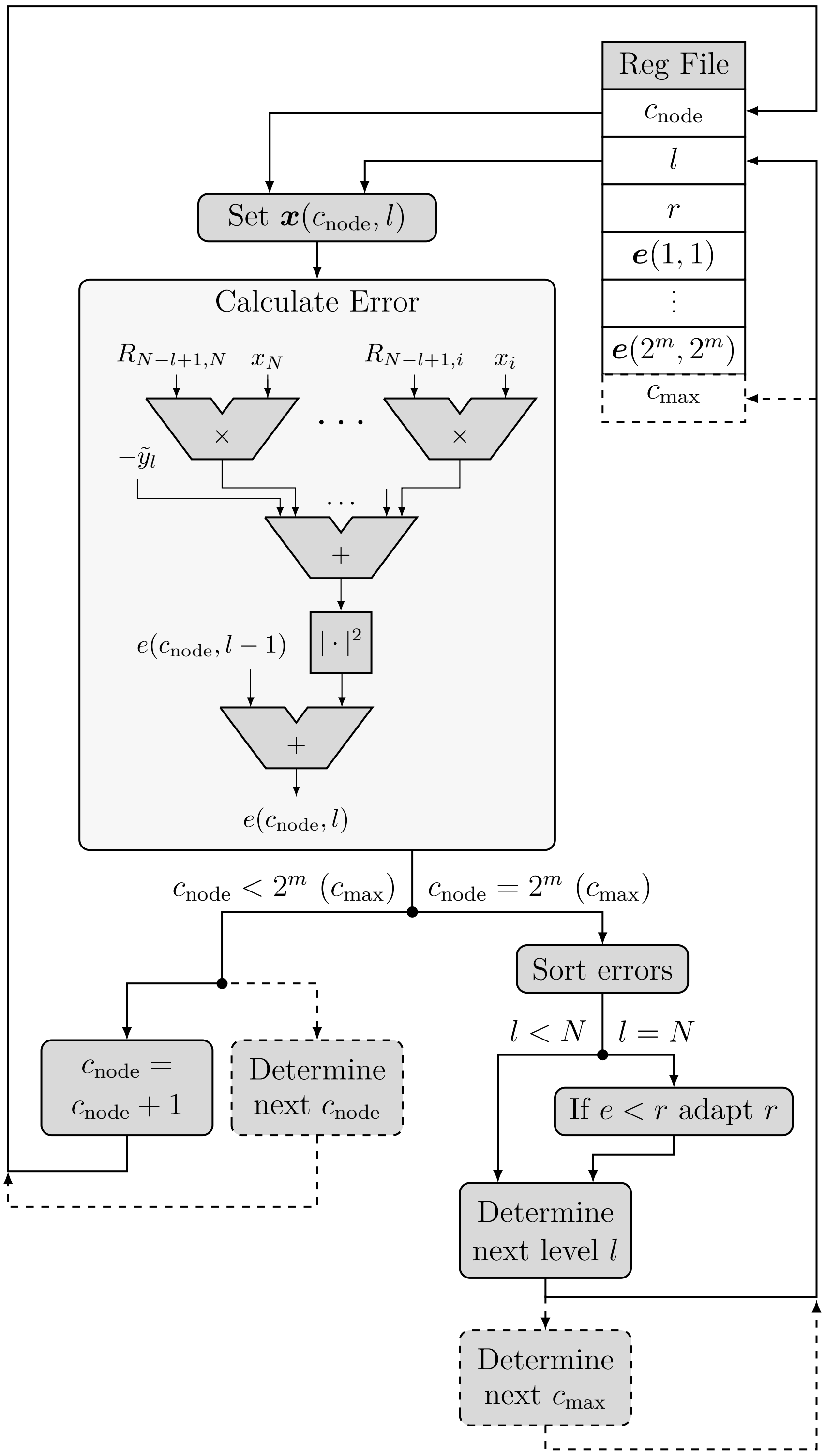
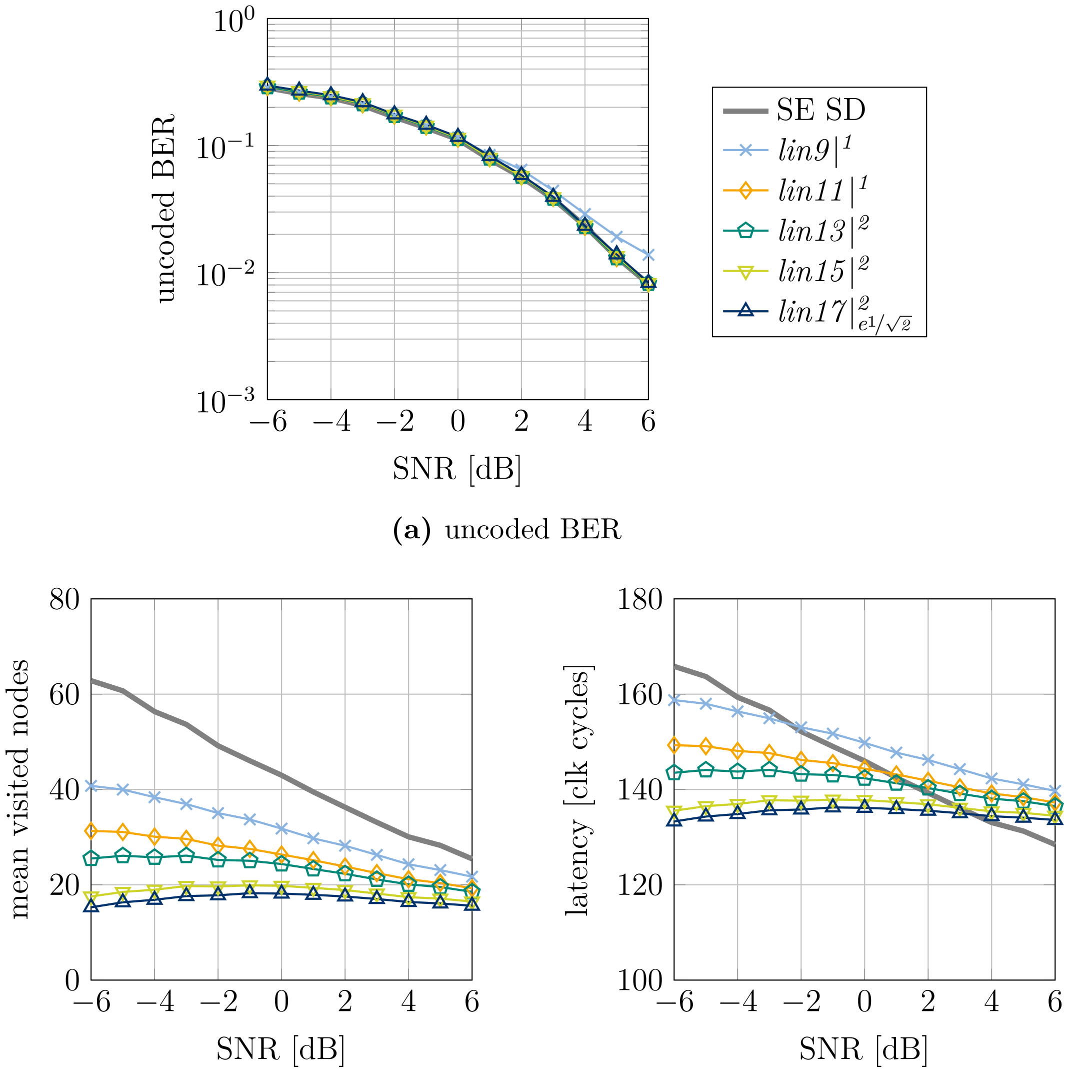


Figure 4.29.: Behavior of the standard SE SD with pruning, according to section 4.2.1.1. Additional steps and adaptions made for the SORN-reduced SD are displayed with dashed lines/blocks [7].

and one cycle per visited node. The number of visited nodes and therefore the total number of required clock cycles for the SD is non-deterministic and will be evaluated in section 4.4.3.

4.4.3. RTL-Simulation

To evaluate on the performance of the implemented designs, figure 4.30 shows the resulting BER, mean number of visited nodes and required latency in clock cycles for the reference SE SD and the SORN SD implementations with a 16 bit FxD format and different SORN datatypes and SNRs. Implementations for a 32 bit FxD format show the same results. Except for the *lin9*$|^{1}$ datatype and high SNR values, all SORN designs achieve the same ML performance as the reference SE SD. The visited nodes results in figure 4.30b confirm the software evaluation



(a) uncoded BER

(b) visited nodes

(c) latency

Figure 4.30.: RTL simulation results for (a) uncoded BER, (b) mean visited nodes and (c) latency in required clock cycles; all for the hardware implementation of the SORN-based SD with preprocessing and QRD, and the standard SE SD with QRD in 16 bit FxD format, for a 4×4 complex-valued MIMO with QPSK [7].

from figure 4.24e: the SORN-based designs achieve a high reduction of the visited nodes, compared to the reference SE SD. These reductions are highly datatype dependent. For the higher bitwidth SORN datatypes, the mean number of visited nodes is also almost constant over SNR, whereas for the SE SD the number decreases for increasing SNR. The visited nodes, however, only show the improvement for the SD itself, and do not take into account the increased preprocessing effort introduced by the SORN components. Therefore figure 4.30c shows the required latency of both top level designs, including the QRD and MVM for both and additionally the SORN preprocessor and sorting module for the SORN SD. The number of required clock cycles C for a complete detection for both SD designs at 100 MHz are given in the following. Note that the SORN preprocessor and the sorting module run with 1 GHz, which is why their respective number of required clock cycles at 100 MHz is not an integer value:

$$C_{SE-SD} = \underbrace{C_{QRD}}_{=97} + \underbrace{C_{MVM}}_{=5} + C_{SD} = 102 + C_{SD} \tag{4.29}$$

$$C_{SORN-SD} = \underbrace{C_{SORN}}_{=13.1} + \underbrace{C_{SORT}}_{=1.9} + \underbrace{C_{QRD}}_{=97} + \underbrace{C_{MVM}}_{=5} + C_{SD} = 117 + C_{SD} \tag{4.30}$$

Due to the high frequency of the SORN components, the introduced latency overhead of the additional preprocessing is less than 15% of the latency required for QRD and MVM. From the results in figure 4.30c it can be seen that this additional latency is smaller than the achieved gain in terms of less visited nodes. In total all SORN designs show a latency improvement for a certain SNR range $< 0_{10}$, with a reduction of up to 20%. The 17 bit SORN design even shows an improvement until an SNR of 3 dB.

4.4.4. Synthesis Results

All implemented designs were synthesized for a 28 nm CMOS technology from STM, the results are given in table 4.6. The FxD part of both the SE and SORN SD was implemented for 16 and 32 bit, the SORN part for datatypes with 9 to 17 bit. All SORN components were synthesized for a frequency of 1 GHz, all FxD components for 100 MHz. For the SORN SD designs with a 16 bit FxD path, the total area increases by up to 12%, comparing the design with 9 bit SORN datatype and one with 17 bit. For the 32 bit FxD SORN SD designs, this increase is much lower with up to 4%. A similar behavior can be observed for the energy, with an increase of up to 17% and 8%, respectively. Considering the area of the different submodules, it can be observed that the QRD is the largest with 50% to 55% for 16 bit FxD, and 67% to 70% for 32 bit FxD. The SORN preprocessor requires only between 10% and 19% of the total design for 16 bit FxD, and between 4% and 8% for 32 bit FxD. The sorting module is even smaller with 9% and 3.5%, respectively. The SD itself requires about 7% and 8%.
When comparing the SORN with the reference SE SD, a total area increase between 41% and 58% can be observed for the 16 bit FxD designs, and between 14% and 18% for 32 bit FxD. The energy increase is between 57% and 83%, and 18% and 27%, respectively, mainly caused by the high frequency used for the SORN components.

Table 4.6.: Synthesis results for the SORN-based SD with preprocessing and QRD, and the standard SE SD with QRD, all for a 4 × 4 complex-valued MIMO system with QPSK, synthesized for 28 nm CMOS technology [7]. The SORN frequency for all designs is 1 GHz, and 100 MHz for the FxD and control paths.

	config.	bitwidth		total area		partial area [%]					energy
		SORN	FxD	[µm^2]	[kGE]	SORN	SORT	QRD	MVM	SD	[µW/MHz]
SORN SD	C1	9	16	154234	315	10.27	9.83	55.18	8.55	7.50	32.77
	C2	11	16	159200	325	13.22	9.53	53.36	8.26	7.20	34.15
	C3	13	16	165509	338	16.07	9.16	51.90	7.97	6.92	35.50
	C4	15	16	170298	348	18.46	8.90	50.46	7.76	6.73	36.60
	C5	17	16	172419	352	19.27	8.79	49.93	7.66	6.68	38.23
	C6	9	32	424054	866	3.83	3.58	69.76	10.71	8.43	70.63
	C7	11	32	430309	879	4.99	3.53	69.12	10.54	8.43	72.52
	C8	13	32	432969	884	6.27	3.50	67.79	10.38	8.59	73.38
	C9	15	32	442277	903	7.21	3.47	67.62	10.17	8.14	75.26
	C10	17	32	441186	901	7.68	3.44	67.29	10.19	8.05	76.25
SE SD	C11	-	16	109165	223	-	-	77.12	11.89	9.95	20.93
	C12	-	32	372897	762	-	-	78.28	11.92	9.06	59.87

4.4.5. Comparison to SOTA Complexity Reduction Approaches

The SORN SD is not the first approach targeting an improvement of the standard SD. Various versions of the algorithm have been proposed in the past, mainly attempting a reduction or fixation of the number of visited nodes per detection. An overview of many of these state-of-the-art approaches with corresponding hardware implementations can be found in [BHEHZ16]. In the following, some of these approaches are compared to the presented SORN SD in terms of visited nodes performance and additional preprocessing effort. The mean number of visited nodes for the different approaches is given in figure 4.31 for a 4 × 4 and 8 × 8 QPSK MIMO system. The additional preprocessing effort is compared in table 4.7 in terms of required floating point, integer or SORN operations. The hardware results of the implemented SORN SD and QRD are compared to state-of-the-art architectures in section 4.4.6.

SQRD Comparable to the SORN SD approach, the sorted QR-decomposition (SQRD) algorithm targets to decrease the number of visited nodes by virtually permuting the antenna order, resulting in a permutation of the channel matrix $\boldsymbol{H}$. The permutation in this approach is based on the norm of the column vectors of $\boldsymbol{H}$ [WRB$^+$02]. This kind of permutation leads to a minimized number of wrong decisions in the first tree levels [SBB08]. From figure 4.31 it can be seen that the SQRD approach leads to a reduced number of visited nodes, compared to the SE SD with standard QRD. The SORN approach, however, shows mostly better results, especially for the 4×4 case. From an implementation point of view, the SQRD replaces the QRD component, leading to a complexity increase of 90% for an FPGA implementation from [CM14], or to an area and latency increase of 40% and 7% for a CMOS implementation in [NGW10].

K-Best Both the K-Best and the FSD approach fix the number of visited nodes per detection and lead to a deterministic latency of the SD. In contrast to the depth-first (DF) SE SD, which evaluates a complete tree path, before going back to higher tree levels to evaluate alternatives, the K-Best detector follows a breadth-first approach. Starting from the root node, in every tree level the k best candidates with lowest error metric $e(l)$ are determined and their paths are followed in parallel, while the remaining nodes are discarded [KCCW02]. This approach leads to a deterministic latency, and the corresponding hardware implementations can be parallelized and pipelined. Depending on the choice of the k parameter, however, the approach also leads to a degradation of the BER performance [BHEHZ16]. The number of visited nodes for the K-Best approach is

$$N_{\text{visitedNodes,K-Best}} = \begin{cases} 2^m + 2^m k(N-1) & k \leq 2^m \\ 2^m + (2^m)^2 + 2^m k(N-2) & 2^m < k \leq (2^m)^2 \end{cases} \tag{4.31}$$

with the modulation bitwidth m. In figure 4.31 the visited nodes are shown for a parameter $k_{4\times4} = 2$ and $k_{8\times8} = 8$, leading to a constant but mainly worse performance compared to the 17 bit SORN approach. When choosing $k_{4\times4} = 4$ and $k_{8\times8} = 16$, the number of visited nodes is equivalent to the FSD algorithm, which will be discussed in the following. For the K-Best detector no additional preprocessing is required. The hardware performance will be discussed in section 4.4.6.

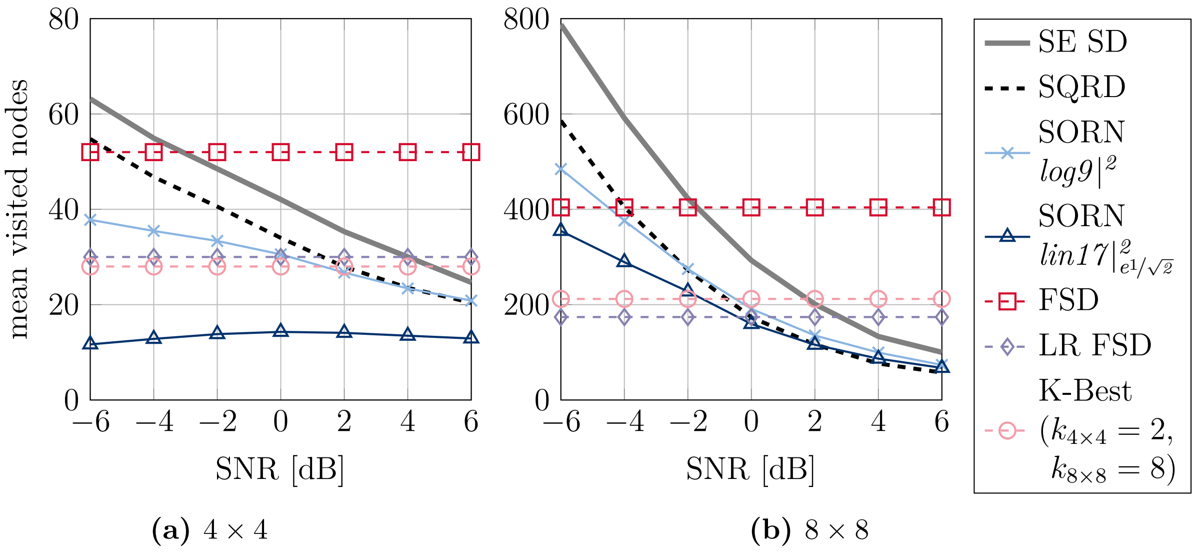


Figure 4.31.: Mean visited nodes of a standard SE SD, with SQRD, K-Best, FSD and LR FSD, and permuted tree after SORN preprocessing; for a (a) 4×4 and (b) 8×8 complex-valued MIMO system with QPSK [7]. (The K-Best, FSD and LR FSD results are obtained analytically.)

FSD In contrast to the K-Best approach, for the fixed-complexity SD (FSD) there are two different parameters fixing the number of considered nodes at different tree levels. In the full expansion stage all possible nodes are considered, whereas in the single expansion stage only one path per subtree is followed [BT08]. For a 4×4 system the first tree level, for 8×8 the first two levels are considered as full expansion, all remaining stages as single expansion. According to [BT08] and [FRG+09], this leads to the following number of visited nodes for the respective system size:

$$\begin{aligned} N_{\text{visitedNodes,FSD},4\times4} &= 2^m + (N-1)(2^m)^2 \\ N_{\text{visitedNodes,FSD},8\times8} &= 2^m + (2^m)^2 + (N-2)(2^m)^3 \end{aligned} \tag{4.32}$$

From figure 4.31 it can be seen that the number of visited nodes for this approach is mostly higher than for all the other approaches. The FSD, however, achieves a quasi-ML BER performance, in contrast to K-Best. Therefore an additional preprocessing step is required, which leads again to a permutation of the channel matrix $\boldsymbol{H}$ in a way that signals with a high noise amplification are detected in the full expansion stage and signals with a low noise amplification during the single expansion [BT08]. The complexity of this preprocessing is given in table 4.7 in terms of floating point operations (FLOPs). A comparison to the complexity of the SORN preprocessor is hardly fair because it requires only SORN and integer operations. When considering the QRD as a reference, however, which requires $37.3N^3$ FLOPs for a $N \times N$ matrix according to [GVL96, KPLK14], it can be shown that for $N = 4$ the FSD preprocessing requires 27% more FLOPs than a QRD. In comparison, from table 4.6 it can be seen that the SORN preprocessing requires at most 56% of the QRD area. In addition, in section 4.4.3 it was shown that the required latency of the SORN preprocessor requires less than 16% of the QRD latency.

LR FSD Even though K-Best and FSD are already adaptions of the standard SD approach, in the literature exist various further adaptions of these approaches. One of these is the lattice-reduced FSD (LR FSD) which targets to reduce the search tree within the full expansion stage of the FSD [KPLK14], comparable to what the K-Best approach does. During the full expansion, not all but a reduced number $k_{\text{LR}} \leq 2^m$ nodes are considered for further

Table 4.7.: Comparison of the preprocessing effort for the SORN-based and FSD algorithms supplementary to a QRD and $\boldsymbol{Q^H y}$ [7].

algorithm	**BER**	**preprocessing**	**operations** ($N \times N$ MIMO)	
FSD [KPLK14]	quasi-ML	FSD ordering of $\boldsymbol{H}$	$10N^4 + 8N^3 - 9N - 9$	FLOPs
LR FSD	sub-optimal	complex LR of $\boldsymbol{H}$	$59.7N^3$	
[KPLK14]		FSD ordering of $\boldsymbol{H}$	$10N^4 + 8N^3 - 9N - 9$	FLOPs
		ZF estimate	$18N^3 + 6N^2 + N$	
SORN SD	quasi-ML	SORN ex. search	$(2^m)^N(8N^2 + 4N - 1)$	SORN OPs
		permutation of $\boldsymbol{H}$	$2N2^m + 0.5N^2 - 1.5N$	Integer OPs

processing. This number is determined through another preprocessing step which performs a lattice reduction (LR) of the channel matrix $\boldsymbol{H}$. The visited nodes performance of this approach, as shown in figure 4.31, is comparable to the K-Best approach, and therefore mostly worse compared to the 17 bit SORN approach. The number of visited nodes is obtained with equation (4.32) and $k_{\mathrm{LR}} = 3$. The additional preprocessing effort of this approach is also listed in table 4.7 and is even higher than for the FSD.

4.4.6. Comparison to SOTA Hardware

The implemented SORN and SE SD are compared to reference designs of different SDs and comparable approaches in table 4.8. All designs were implemented for 4×4 MIMO with different modulations, as indicated. The results for the implemented SORN and SE SD are given without QRD and MVM preprocessing to allow a fair comparison, since the reference designs do not include these modules either. The SORN preprocessor and sorting module, however, are included in the results. The power consumption of the different designs is normalized to a 65 nm technology with 1.2 V supply voltage (V_{dd}) [BHEHZ16] to facilitate a comparison:

$$\text{norm. power} = \text{power} \times \left(\frac{1.2\,\mathrm{V}}{V_{dd}}\right)^2 \times \left(\frac{65\,\mathrm{nm}}{\text{tech.}}\right) \tag{4.33}$$

The throughput is obtained as

$$\text{throughput} = \frac{N \times m}{C} \times f \quad [\text{bit/s}] \tag{4.34}$$

with the MIMO dimension N, the modulation number m, the number of required clock cycles C and the clock frequency f [BBW+05].
Even though a normalized power, as well as gate equivalents are used, the comparisons can never be totally accurate, since different technologies, modulations and/or SNRs are used. However, comparing the implemented reference SE SD with the literature SDs from [YTC+13], [ROP11] and [GNJ17], a good throughput-area ratio, as well as a low power consumption can be achieved. For the SORN SD the throughput is further improved, while the power and area increase is at a moderate level. For the reference design [GNJ17] the throughput is about an order of magnitude higher than for the SORN SD, but it has to be noted that the throughput in [GNJ17] is given for an SNR of 12 dB, whereas the implemented designs are evaluated at 0 dB. In comparison with the fixed-complexity approaches K-Best [PSSG10] and FSD [LLN12], the achieved throughput of the implemented SORN and SE SD approaches is about two orders of magnitude lower. On the other hand, these designs show a higher area and a much higher power consumption, even though the results do not include the required preprocessing in addition to a QRD. Moreover, these designs do not achieve the same BER performance as the implemented SORN and SE SD. When comparing solely the SORN and SE SD, the area and power increase between both seems to be quite high, but it has to be considered that these results do not include the QRD module, which was shown to take the major part of both designs area and power consumption in section 4.4.4.
In table 4.9 the implemented QRD module is compared to reference implementations in order

Table 4.8.: Comparison Results for the implemented SE SD and SORN SD with SORN preprocessing (both without QRD and MVM) and reference architectures [7].

		[YTC⁺13]	**[ROP11]**	**[GNJ17]**	**[PSSG10]**	**[LLN12]**	**this work**	
detector		ASE SD	SD ASIP	DF SD	K-Best ($k = 10$)	imbal. FSD	SE SD	SORN SD
dimension		4×4	4×4	4×4	4×4	4×4	4×4	
modulation		QPSK	16 QAM	16 QAM	64 QAM	64 QAM	QPSK	
bitwidth		n.a.	22 (FP)	24	n.a.	n.a.	16	16/17
process/V_{dd}		90 nm/1.0 V[(a)]	65 nm/n.a.	45 nm/0.9 V	65 nm/1.3 V	65 nm/1.2 V	28 nm/0.9 V	
preprocessing		incl.	-	-	not incl.	not incl.	-	incl.
BER-performance		sub-optimal	quasi-ML	quasi-ML	sub-optimal	sub-optimal	quasi-ML	quasi-ML
area	[kGE]	153.9	30[(b)]	51.2	298	88.2	25	149
frequency	[MHz]	108.7	300	435	833	165	100	100/1000
power	[mW]	28.75	8.51	9.56	280	102.7	0.39	7.46
norm. power	[mW]	29.90	8.51	24.55	238.6	102.7	1.61	30.79
throughput	[Mbps]	40	15.59	275.86	2000	1980	18.2	23.5
@SNR	[dB]	0	n.a.	12	-	-	0	0

[(a)]Typical V_{dd} for TSMC 90 nm Std Cell Library

[(b)]Given area from [ROP11] (47832 µm^2) divided by the area of a NAND2X1 gate (1.6 µm^2 for TSMC 65 nm)

Table 4.9.: Comparison of the implemented and reference 4 × 4 complex QRD architectures [7].

		[HCW15]	**[TS17]**	**[GLS$^+$13]**	**this work**
algorithm		SVD/QRD	QRD/SQRD	QRD	QRD
bitwidth		n.a.	n.a.	13	16
process/V_{dd}		90 nm/1.0 V$^{(a)}$	90 nm/1.0 V$^{(a)}$	65 nm/1.0 V	28 nm/0.9 V
area	[kGE]	452	375	378	176
frequency	[MHz]	143	220	72	100
power	[mW]	93.54	140	127	1.53
norm. power	[mW]	97.28	145.6	182.9	6.31
throughput	[matr. / s]	35.75 M	44 M	72 M	1.03 M

$^{(a)}$Typical V_{dd} for TSMC 90 nm Std Cell Library

to rate its impact on the overall SORN SD design. The reference designs are implemented as systolic array [TS17, GLS$^+$13] or massively parallel CORDIC processors [HCW15] in order to enhance the throughput, while in this work the QRD is implemented iteratively, which leads to a much lower throughput. The high throughput of the reference designs is compensated by a high complexity and power consumption, which is much lower for the implemented QRD module. Since for the complete SORN SD design, the implemented QRD module requires at least half of the chip area and the highest amount of required clock cycles, as shown in section 4.4.4, a further increase of the area for a higher-throughput QRD would minimize the relative complexity overhead introduced by the SORN preprocessor. In addition, a lower latency of the QRD would increase the impact of the visited-nodes-reduction introduced by the SORN SD and lead to a higher overall latency reduction compared to the SE SD.

4.5. Summary

In this chapter the Sets Of Real Numbers (SORN) format was evaluated in terms of its algorithmic performance for dedicated applications. For the first application, SORNs were considered to implement the Sobel operator used for edge detection in image processing. It was shown that for this threshold-based algorithm the state-of-the-art integer arithmetic datapath can be replaced with SORN arithmetic, which provides a similar algorithmic performance while improving all hardware measures. For the second application, SORNs were applied within a wireless MIMO communication system. In particular, a SORN preprocessor was introduced, which targets to reduce the search space of a MIMO symbol detection in order to reduce the latency of such an algorithm. It was shown that this approach can drastically reduce the number of possible solutions for a detection, depending on multiple parameters like the SORN datatype, the number of transmit and receive antennas, the applied modulation, or the signal-to-noise ratio. Evaluations on hardware measures showed that such a SORN preprocessor can be incorporated in a state-of-the-art detector without limiting the throughput or adding a disproportionate amount of complexity or power consumption. The presented SORN preprocessor was further applied within a hybrid SORN/FxD BPSK detector for a real-valued MIMO system, and within a sphere decoder architecture with additional QRD preprocessing for a complex-valued system. For both approaches it was shown how SORNs can improve the state-of-the-art, either by improving the latency of the detector, and/or by reducing the hardware complexity and power consumption.

5 Conclusion and Outlook

The SORN number format is a derivative of the universal numbers and targets to perform low-complex and fast arithmetic operations in hardware via lookup tables. This thesis presents the hardware implementation of the original SORN approach and its evaluation with an automated design flow tool for design space exploration. The primary contributions comprise the SORN design flow, optimizations of the original approach such as adaptions of the SORN datatypes and the introduction of fused arithmetic, as well as the application of SORNs for different use cases. The primary contributions and findings of this thesis can be summarized as follows.

SORN Datatypes The originally proposed SORN datatypes were derived from the unum representation and consist of exact values and open intervals with lattice values that are included together with their reciprocals and negative equivalents. This very regular and rather strict structure misses flexibility, even though the arithmetic concept with pre-computed LUTs could generally be used with any datatype, when abandoning unum compatibility. A further issue is the combination of exact values with open intervals in the original datatype, which introduces a certain redundancy when used within SORN arithmetic, as discussed in section 3.2.2. Due to these drawbacks, within the thesis a new SORN datatype structure with half-open intervals and a less strict lattice value structure and distribution was proposed. Both datatype approaches were compared regarding their algorithmic and hardware performance. Evaluations for the SORN MIMO preprocessor in sections 4.2 - 4.4 show that half-open datatypes outperform the unum-based ones, while mostly requiring less bitwidth. For the hardware implementation of the SORN preprocessor in section 4.2.4, again those architectures with half-open datatypes show better results when comparing equal bitwidths, except for the area, which was slightly lower for the original approach. Considering the hardware performance of basic addition and multiplication operations in section 3.4.2, both approaches show similar results when comparing same bitwidths, but a much better performance for the half-open approach when comparing datatypes with the same lattice values, i.e. the same precision and dynamic range. In general, it can be concluded that the proposed half-open SORN datatypes show a significant improvement over the original approach, improving both the algorithmic and hardware performance at the same time.

Automated SORN Design Flow The proposed SORN datatype structure is highly flexible, not only in terms of bitwidth, but also concerning the choice of lattice values and their distribution. Since the SORN arithmetic LUTs have to be recomputed for every datatype and operation, an automation of this process is required in order to perform a design space exploration of different SORN implementations. The automated design flow presented in section 3.3 includes an open source SORN hardware generator tool which not only provides arithmetic operation blocks for different operations and user defined datatypes, but also complete datapaths to implement and evaluate entire algorithms with SORN arithmetic.

Fused SORN Arithmetic Since the SORN arithmetic LUTs are pre-computed with the automated hardware generator tool, they can be adapted for different operations, including a combination of multiple basic arithmetic operations, so-called fused operations. This approach was incorporated into the automated SORN design flow and evaluated for different basic and complex fused operations for up to three inputs. The main advantage of fused SORN operations is the improvement of the output accuracy, which, for a SORN computation, can be measured by the width of the output intervals. The presented evaluations in section 3.5 show that fused SORN operations improve the output accuracy in nearly all cases, compared to standard, non-fused SORN operations. In addition, fused SORN operations for up to two inputs also mostly improve the hardware performance, while for three-input fused architectures this improvement is highly operation and datatype dependent.

SORN Applications In order to evaluate on the suitability of SORN arithmetic for digital signal processing, in this thesis SORNs were applied to use cases for image processing and symbol detection in wireless MIMO communication. The considered Sobel operator is a threshold-based algorithm for edge detection within image processing. The presented evaluations from section 4.1 show that a SORN implementation of the algorithm can replace a state-of-the-art integer version, providing a similar algorithmic performance while improving all hardware measures. For the use case of symbol detection in wireless MIMO communication, a SORN preprocessor was presented in section 4.2, which can be used to constrain the solution space for symbol detection. The preprocessor was used to implement a hybrid SORN/FxD BPSK detector for a real-valued MIMO system in section 4.3, and within a sphere decoder with additional QRD preprocessing for a complex-valued system in section 4.4. For both approaches it was shown that SORNs can improve the respective state-of-the-art detectors, either by improving the latency or by reducing hardware complexity and power consumption.

The SORN number format with its corresponding LUT-based arithmetic provides a low-complex, low-energy and fast-computing approach for implementing arithmetic operations, outperforming state-of-the-art number system implementations of similar bitwidths. Since the interval-based approach is different to standard formats and comprises a lower precision, SORNs are not suitable as a general drop-in replacement for these standard formats. However, this thesis shows how the SORN approach can be exploited for improving the implementation of threshold-based algorithms, as well as for efficiently constraining large optimization problems by means of preprocessing. With the presented open-source tool for automated design space exploration, SORN implementations can be easily evaluated.

5.1. Outlook

The presented thesis mainly focuses on the SORN arithmetic concept with pre-computed LUTs, different datatypes and fused operations on one hand, and on suitable application scenarios on the other. A third aspect is the automated design flow which makes evaluations on the first two topics feasible. Following the research topics presented here, possible future work on SORNs is threefold. The first aspect is the arithmetic concept, especially the determination of suitable SORN datatypes. Throughout this work the implemented datatypes are somehow application-specific, with dynamic range, precision and sometimes also single exact values matched to the pixel values for edge detection or the MIMO detection problem. However, with their mostly regular and symmetric value distribution, they are still rather generic. As described in section 3.2.4, there are other possibilities to further match a SORN datatype to a specific application, for example by exploiting the statistics of the application data and deriving a suitable datatype accordingly. Even though in sections 4.2.5 and 4.2.7 it was evaluated that for the given application the statistically determined datatype does not lead to better results, the general idea is still worth considering for future work.
For such an approach, the automated design flow needs to be updated in order to perform an evaluation of the application data and automatically set up a SORN datatype that is compatible with the existing design flow. This contributes to the second aspect of possible future work, which is improving the design flow and the SORN hardware generator tool. Besides the statistically determined datatype generation, other implementation techniques could be considered. The current tool provides fully connected and parallel implementations of the user defined algorithms, which can be pipelined as required. Instead, also an iterative approach could be considered, as well as a resource-sharing between different submodules or even different operation blocks. A further possible extension is the automated combination of SORN and integer or fixed point datapaths including conversion functions to exploit hybrid SORN designs.
A third topic for future research are further applications for SORN arithmetic. This can include other threshold-based and constrainable optimization algorithms, as well as further suitable arithmetic problems. The currently most emerging research field is machine learning, where SORNs can be applied to classification algorithms such as support vector machines [HBRP21, HBRP23], or the k-nearest neighbor approach [HBK+22]. Due to the very high number of operations and the resulting complexity, the hardware implementation of neural networks is also a promising application field for SORN arithmetic.

A Supplementary Algorithms & Program Code

A.1. QR Decomposition with Givens Rotation

The QR decomposition is a method in linear algebra to split any complex-valued matrix $\boldsymbol{A} \in \mathbb{C}^{N \times N}$ into an orthogonal matrix $\boldsymbol{Q} \in \mathbb{C}^{N \times N}$ and an upper triangular matrix $\boldsymbol{R} \in \mathbb{C}^{N \times N}$ and can be used to solve full rank least squares problems [GVL96]. For computing the decomposition orthogonal transformations like Gram-Schmidt, Householder Reflection or Givens Rotation can be applied. The latter will be discussed in the following.
The approach of the Givens Rotation is to successively generate zero-elements below the main diagonal of the input matrix $\boldsymbol{A}$ by multiplying with a complex rotation matrix. After an initialization of $\boldsymbol{R} = \boldsymbol{A}$, the decomposition is performed by iteratively multiplying the rotation matrix with $\boldsymbol{R}$ [7]:

$$\begin{bmatrix} 1 \dots & 0 & 0 & \dots 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 \dots & c & s & \dots 0 \\ 0 \dots & -s^* & c^* & \dots 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 \dots & 0 & 0 & \dots 1 \end{bmatrix} \begin{bmatrix} R_{11} & \dots & R_{1i} & R_{1j} & \dots & R_{1N} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & R_{ii} & R_{ij} & \dots & R_{iN} \\ 0 & \dots & R_{ji} & R_{jj} & \dots & R_{jN} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & R_{Ni} & R_{Nj} & \dots & R_{NN} \end{bmatrix} = \begin{bmatrix} R_{11} & \dots & R_{1i} & R_{1j} & \dots & R_{1N} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & R'_{ii} & R'_{ij} & \dots & R'_{iN} \\ 0 & \dots & 0 & R'_{jj} & \dots & R'_{jN} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & R_{Ni} & R_{Nj} & \dots & R_{NN} \end{bmatrix} \tag{A.1}$$

The rotation matrix is composed of the elements c and s, as well as their complex conjugates c^* and s^*:

$$c = \frac{R^*_{ii}}{\sqrt{|R^*_{ii}|^2 + |R^*_{ji}|^2}} \qquad s = \frac{R^*_{ji}}{\sqrt{|R^*_{ii}|^2 + |R^*_{ji}|^2}} \tag{A.2}$$

The rotation matrix creates a zero entry in $\boldsymbol{R}$ at position $\{j, i\}$. After every iteration the rotation matrix has to be recomputed according to the updated $\boldsymbol{R}$. The $\mathbf{Q}$ matrix is the product of all intermediate rotation matrices [GVL96].

A.2. Specification Files for the SORN Hardware Generator

Listing A.1 Specification file for the SORN Hardware Generator tool to generate a SORN MIMO preprocessor with $N = 4$ and one pipeline stage for a *lin13*$|^1$ datatype.

```
@Name MIMO_solver_N4
@datatype ["lin","[0,1,1/5]","Zero","infty","negative"]
@pipeline 1

################## Function Definition: ###################
### H*s
Hs0_re = (((H00_re * s0_re) - (H00_im*s0_im)) + ((H01_re * s1_re) -
 ↪  (H01_im * s1_im))) + (((H02_re*s2_re) - (H02_im*s2_im)) +
 ↪  ((H03_re * s3_re) - (H03_im * s3_im)))
Hs1_re = (((H10_re * s0_re) - (H10_im*s0_im)) + ((H11_re * s1_re) -
 ↪  (H11_im * s1_im))) + (((H12_re*s2_re) - (H12_im*s2_im)) +
 ↪  ((H13_re * s3_re) - (H13_im * s3_im)))
Hs2_re = (((H20_re * s0_re) - (H20_im*s0_im)) + ((H21_re * s1_re) -
 ↪  (H21_im * s1_im))) + (((H22_re*s2_re) - (H22_im*s2_im)) +
 ↪  ((H23_re * s3_re) - (H23_im * s3_im)))
Hs3_re = (((H30_re * s0_re) - (H30_im*s0_im)) + ((H31_re * s1_re) -
 ↪  (H31_im * s1_im))) + (((H32_re*s2_re) - (H32_im*s2_im)) +
 ↪  ((H33_re * s3_re) - (H33_im * s3_im)))
Hs0_im = (((H00_re * s0_im) + (H00_im*s0_re)) + ((H01_re * s1_im) +
 ↪  (H01_im * s1_re))) + (((H02_re*s2_im) + (H02_im*s2_re)) +
 ↪  ((H03_re * s3_im) + (H03_im * s3_re)))
Hs1_im = (((H10_re * s0_im) + (H10_im*s0_re)) + ((H11_re * s1_im) +
 ↪  (H11_im * s1_re))) + (((H12_re*s2_im) + (H12_im*s2_re)) +
 ↪  ((H13_re * s3_im) + (H13_im * s3_re)))
Hs2_im = (((H20_re * s0_im) + (H20_im*s0_re)) + ((H21_re * s1_im) +
 ↪  (H21_im * s1_re))) + (((H22_re*s2_im) + (H22_im*s2_re)) +
 ↪  ((H23_re * s3_im) + (H23_im * s3_re)))
Hs3_im = (((H30_re * s0_im) + (H30_im*s0_re)) + ((H31_re * s1_im) +
 ↪  (H31_im * s1_re))) + (((H32_re*s2_im) + (H32_im*s2_re)) +
 ↪  ((H33_re * s3_im) + (H33_im * s3_re)))

### y-H*s
y_Hs0_re = y0_re - Hs0_re
y_Hs1_re = y1_re - Hs1_re
y_Hs2_re = y2_re - Hs2_re
y_Hs3_re = y3_re - Hs3_re
y_Hs0_im = y0_im - Hs0_im
y_Hs1_im = y1_im - Hs1_im
y_Hs2_im = y2_im - Hs2_im
y_Hs3_im = y3_im - Hs3_im

### ||y-H*s||^2
squared_norm = ((y_Hs0_re**2 + y_Hs0_im**2) + (y_Hs1_re**2 +
 ↪  y_Hs1_im**2)) + ((y_Hs2_re**2 + y_Hs2_im**2) + (y_Hs3_re**2 +
 ↪  y_Hs3_im**2))
```

A.3. Sorting Algorithm for the SORN SD

The exact and approximate versions of the sorting algorithm to determine the permutation order $\boldsymbol{p} \in \mathbb{N}^N$, which gives the new virtual order of the transmit antennas are given in code A.1 and A.2, respectively. The inputs of the algorithm are the matrix with all remaining symbol vectors $\boldsymbol{\mathcal{R}} \in \mathbb{C}^{N\times|\mathcal{R}|}$ and the symbol vectors per tree level l, $\boldsymbol{x}_l(k) \in \mathbb{C}^l$, with $k = 1, \dots, (2^m)^l$.

Code A.1 Sorting algorithm to compute the permutation order $\boldsymbol{p}$ (exact version) [6].

```
Input: $\boldsymbol{\mathcal{R}}, \boldsymbol{x}_l$    ▷ matrix with all remaining symbol vectors, vector of symbol combinations
Output: $\boldsymbol{p} \in \mathbb{N}^N$    ▷ permutation order
1: for $l = 1, \dots, N-1$ do    ▷ determine permutation elements
2:     Set $\boldsymbol{C} \in \mathbb{N}^{(N-l+1)\times(2^m)^l}$ as
3:     $c_{lik} := \sum_\kappa \left(\boldsymbol{\mathcal{R}}_{\{\underbrace{p_N, \dots p_{N-l+2}}_{\text{not existent for } l=1}, i\}\kappa} = \boldsymbol{x}_l(k)\right)$    ▷ count symbol vectors $\boldsymbol{x}_l(\kappa)$
4:     for $i = 1, \dots, N$ do
5:         Set $\boldsymbol{T}_l \in \mathbb{N}^N$ with $T_{li} := \sum_{k=1}^{2^m} c_{lik}^2$    ▷ compute squared sum
6:     end for
7:     $p_{N-l+1} = \arg\max_i(T_{li})$    ▷ use max for unbalanced and min for balanced branches
8: end for
```

Code A.2 Sorting algorithm to compute the permutation order $\boldsymbol{p}$ (approximate version) [6].

```
Input: $\boldsymbol{\mathcal{R}}, \boldsymbol{x}_l$    ▷ matrix with all remaining symbol vectors, vector of symbol combinations
Output: $\boldsymbol{p} \in \mathbb{N}^N$    ▷ permutation order
1: Set $\boldsymbol{C} \in \mathbb{N}^{N\times 2^m}$ as $c_{ik} := \sum_\kappa \left(\boldsymbol{\mathcal{R}}_\kappa(i) = \boldsymbol{x}_1(k)\right)$
2: Set $\boldsymbol{T} \in \mathbb{N}^N$ with $T_i := \sum_{j=1}^{2^m} c_{ij}^2$
3: $\boldsymbol{p} = \arg\mathrm{sort}_i(T_i)$    ▷ use *ascend* for balanced and *descend* sorting for unbalanced branches
```

B Supplementary Results

B.1. SORN & Integer Addition/Multiplication CMOS Synthesis Results

For the evaluation of basic addition and multiplication SORN components in section 3.4, table B.1 shows the detailed results for the graphs 3.7 and 3.8.

Table B.1.: Synthesis results for addition and multiplication components without pipeline stages for $f = 1\text{GHz}$ and CMOS 28 nm technology.

datatype	addition			multiplication		
	area [µm^2]	**delay** [ns]	**power** [nW]	**area** [µm^2]	**delay** [ns]	**power** [nW]
	integer/fixed point					
int5	11.587	0.346	2777.889	56.794	0.461	10297.013
int8	19.421	0.528	4507.542	148.022	0.783	14480.892
int9	22.032	0.583	4529.504	197.146	0.921	19869.390
int11	27.254	0.710	4536.213	314.813	0.973	20735.092
int13	32.477	0.828	5019.608	463.814	0.978	29296.747
int16	42.758	0.955	5582.112	583.930	0.975	31596.278
int17	47.981	0.941	6359.625	757.085	0.975	33943.750
	original SORN					
unum8	68.870	0.260	5281.106	66.749	0.247	5105.532
unum9	40.147	0.330	3399.655	37.699	0.379	3545.094
unum16	275.808	0.393	3890.240	274.829	0.343	4566.554
unum17	126.806	0.491	4045.557	120.442	0.422	4641.333
	half-open SORN					
$\mathit{log5}\|^{1}$	16.483	0.201	2828.047	15.667	0.210	2477.759
$\mathit{lin9}\|^{1}$	50.918	0.326	3191.194	38.189	0.341	3638.113
$\mathit{lin9}\|^{120}$	50.918	0.326	3191.194	18.768	0.257	1654.762
$\mathit{log9}\|^{1}$	52.061	0.318	3371.655	41.942	0.328	4001.991
$\mathit{log9}\|^{2}$	52.061	0.318	3371.655	45.043	0.253	4193.180
$\mathit{log9}\|^{128}$	52.061	0.318	3371.655	18.768	0.257	1654.762
$\mathit{lin11}\|^{1}$	72.950	0.307	3756.115	52.387	0.313	4171.617
$\mathit{lin11}\|^{200}$	72.950	0.307	3756.115	20.074	0.290	1466.160
$\mathit{lin11}\|^{250}_{nz,nn}$	50.918	0.296	3197.559	9.629	0.268	601.157
$\mathit{log11}\|^{1}$	66.749	0.311	3032.919	62.342	0.376	4956.896
$\mathit{log11}\|^{256}$	66.749	0.311	3032.919	20.074	0.290	1466.160
$\mathit{lin13}\|^{1}$	99.226	0.387	2819.989	76.541	0.347	5009.221
$\mathit{lin13}\|^{1}_{e1/2}$	90.413	0.440	3733.622	70.176	0.335	4552.063
$\mathit{log13}\|^{1}$	88.454	0.317	3135.989	83.395	0.358	4974.518
$\mathit{log17}\|^{8}$	123.053	0.422	3061.809	172.666	0.347	6049.132

B.2. FPGA Synthesis Results for Three-Input Fused Operations

Figure B.1 shows the FPGA synthesis results including LUT utilization and path delay for the evaluation of three-input fused operations from section 3.5.2. The target platform is a ZYNQ-7 ZC702 Evaluation Board from Xilinx. For all synthesized designs, no digital signal processors (DSPs) or Block RAMs (BRAMs) are utilized. The power consumption is not shown because the power of the designs is negligible compared to that of the FPGA and no differences between non-fused and fused can be observed.

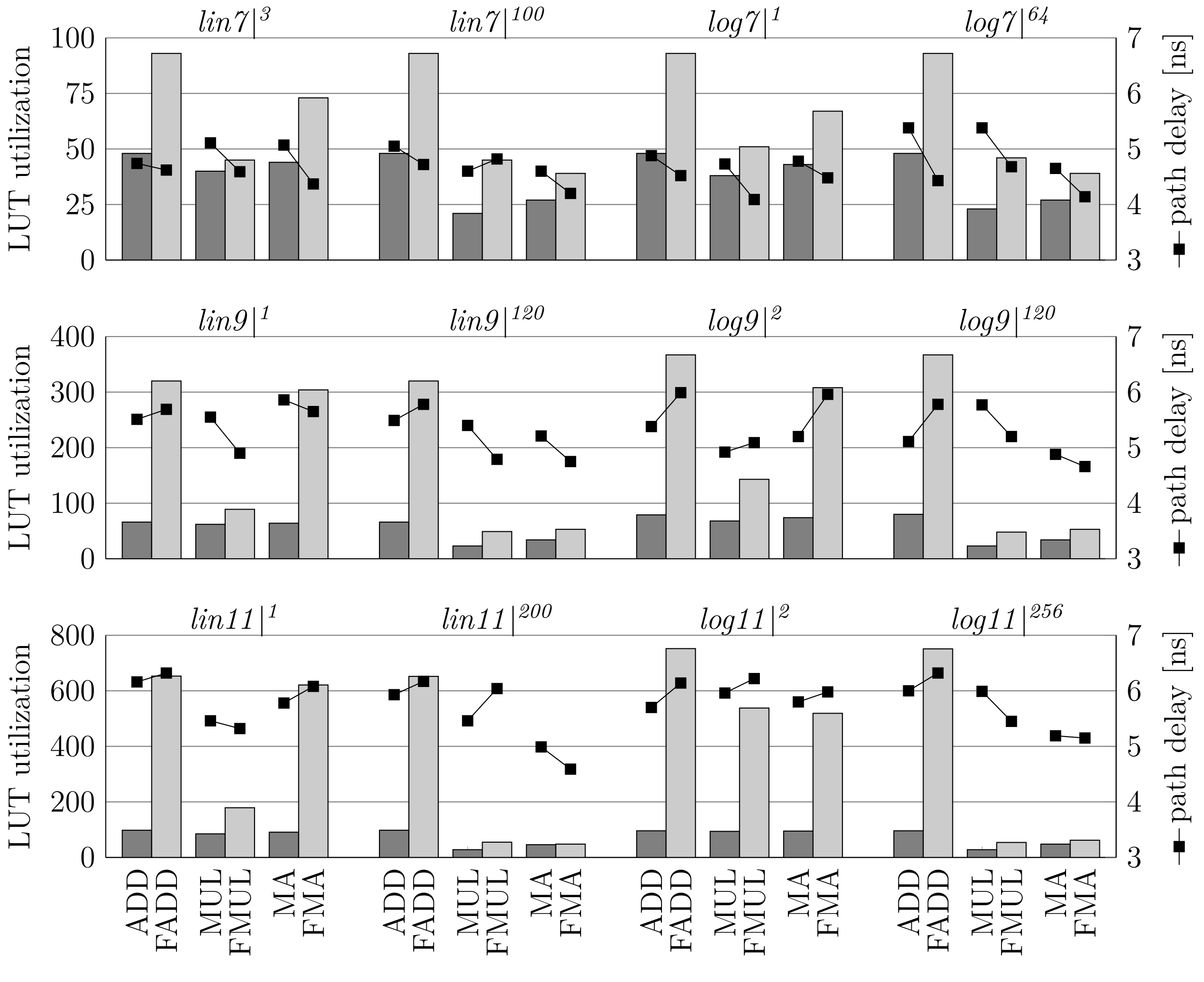


Figure B.1.: FPGA Synthesis results for non-fused and fused three-input addition, multiplication and multiply-add, for a ZYNQ-7 ZC702 Evaluation Board (xc7z020clg484-1) with $f = 150\,\text{MHz}$ [10].

B.3. FPGA Synthesis Results for SORN Sobel Implementations

In table B.2 the synthesis results for an Artix-7 AC701 FPGA from Xilinx are given for the integer and SORN Sobel implementations presented in section 4.1.2 for a target frequency of 100MHz. The worst negative slack (WNS) shows the required runtime margin with

$$\text{runtime} = \frac{1}{\text{frequency}} - \text{WNS} \tag{B.1}$$

indicating that the target frequency was met, when the WNS value is positive.

Table B.2.: Synthesis results without DSPs for an Artix-7 AC701 FPGA (xc7a200tfbg676-2) [8].

		Integer	**hybrid SORN**			**full SORN**	
			$lin6\|^{250}_{nz,nn}$	$log10\|^{256}_{nn}$	$lin11\|^{250}_{nz,nn}$	$log15\|^{512}$	$lin15\|^{300}$
target freq.	[MHz]	100	100	100	100	100	100
WNS	[ns]	−1.487	0.554	0.492	−0.173	−0.466	−1.042
max freq.	[MHz]	87.055	105.865	105.175	98.299	95.548	90.563
LUTs		457	148	207	219	597	712
total power	[W]	0.145	0.136	0.137	0.138	0.140	0.147

B.4. MRS for Linear Half-Open SORN Datatypes

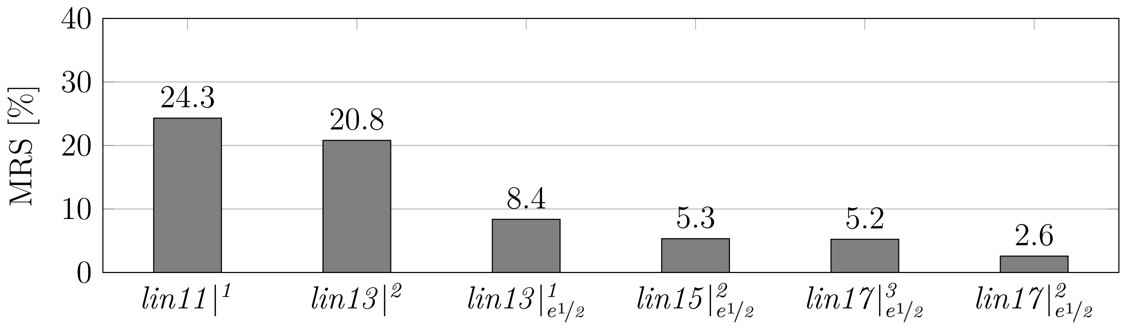
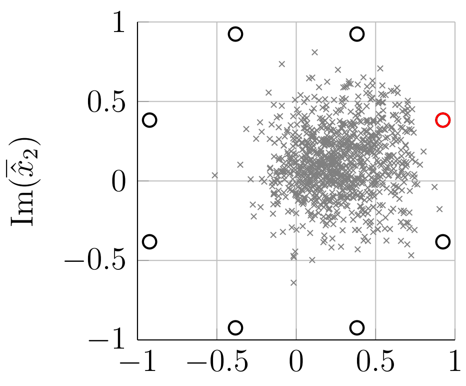


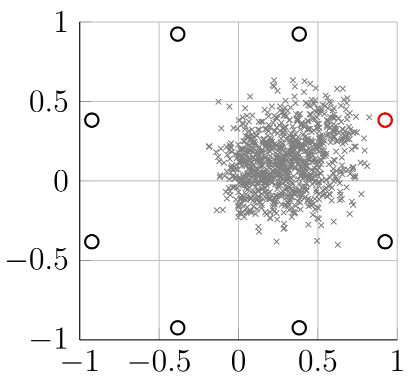
Figure B.2.: The mean (number of) remaining solutions (MRS) for the MLE problem (4.12) in % after SORN preprocessing for an SNR of 10 dB, for a complex-valued 4×4 MIMO system with 4-QAM modulation, and different SORN datatypes [3].

B.5. Scatter and Angle Plots for SORN MIMO Preprocessor with 8-PSK

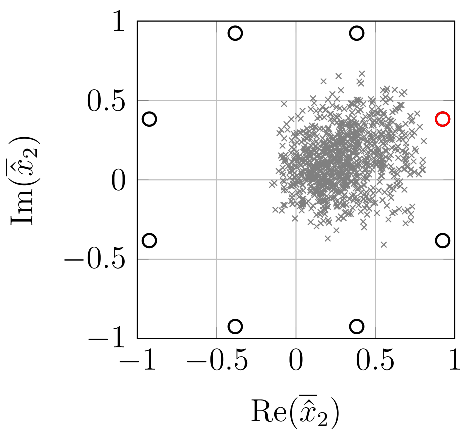
In figure B.3 the position of the mean estimation after SORN preprocessing according to equation (4.26) from section 4.2.6 is shown for $N_{\text{test}} = 10^3$ transmissions for the symbol $x_2 = {}^2\!/_{3\sqrt{2}} + \mathrm{j}\,{}^1\!/_{2\sqrt{2}}$ for a complex-valued 4×4 system with 8-PSK modulation using the *lin13*$|^1$ datatype and different SNR values. Figure B.4 shows the heuristic of the corresponding angles $\angle\bar{\hat{x}}_2$ for the same test setup.



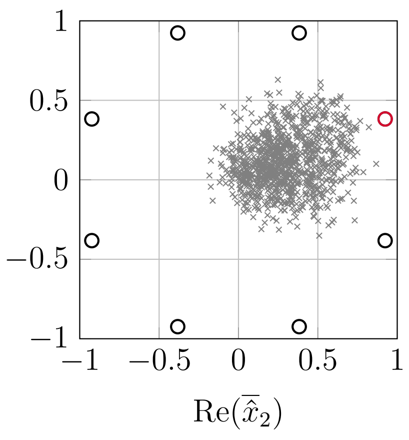
(a) 0 dB



(b) 10 dB

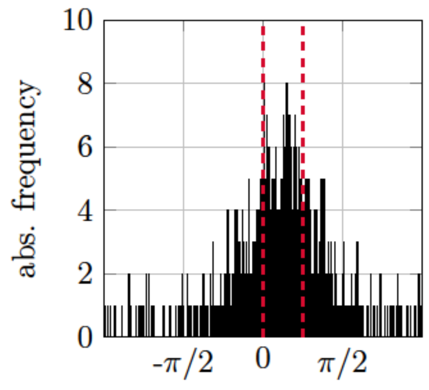


(c) 20 dB

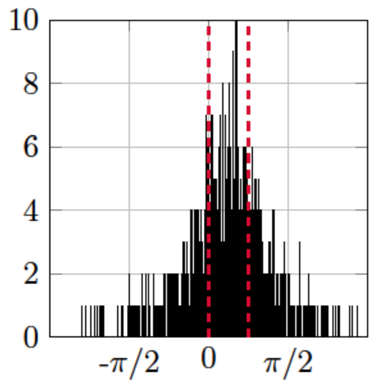


(d) 30 dB

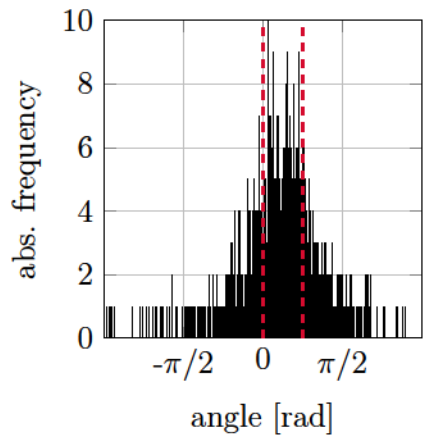
Figure B.3.: Scatter plot for the arithmetic mean $\bar{\hat{x}}_2$ of the remaining solutions after SORN preprocessing for 10^3 transmissions of the symbol $x_2 = {}^2\!/_{3\sqrt{2}} + \mathrm{j}\,{}^1\!/_{2\sqrt{2}}$, for $N = 4$ with 8-PSK, the *lin13*$|^1$ SORN datatype and different SNR values [5].



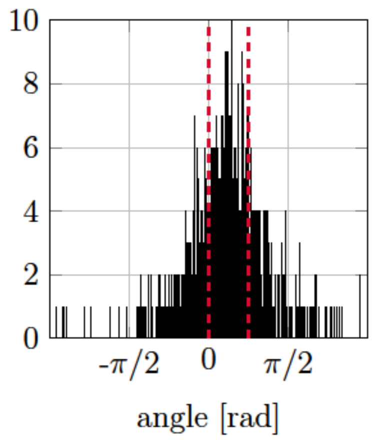
(a) 0 dB



(b) 10 dB



(c) 20 dB



(d) 30 dB

Figure B.4.: Heuristic for the angle of the arithmetic mean $\angle\overline{\hat{x}}_2$ of the remaining solutions after SORN preprocessing for 10^3 transmissions of the symbol $x_2 = {}^{2}/_{3\sqrt{2}} + \mathrm{j}\,{}^{1}/_{2\sqrt{2}}$, for $N = 4$ with 8-PSK, the *lin13*$|^{1}$ datatype and different SNR values [5].

List of Acronyms

AD	anno Domini
ALU	arithmetic logic unit
APT	area-power-timing
ARITH	IEEE International Symposium on Computer Arithmetic
ASE	admissible set elimination
ASIC	application-specific integrated circuit
ASIP	application-specific instruction-set processor
BER	bit error rate
BSDS	Berkeley Segmentation Data Set
BPSK	binary phase-shift keying
BRAM	Block RAM
CMOS	complementary metal-oxide-semiconductor
CORDIC	Coordinate Rotation Digital Computer
CoNGA	Conference for Next Generation Arithmetic
CPA	carry propagate adder
CRT	Chinese Remainder Theorem
CSA	carry-save adder
CSI	channel state information
dB	decibel
DF	depth-first
DL	deep learning
DSP	digital signal processor
FA	full adder
FF	finite field
FFT	fast Fourier transform
FMA	fused multiply-add
FLOP	floating point operation
FP	floating point
FPGA	field-programmable gate array
FPU	floating point unit
FSD	fixed-complexity SD
FSM	finite-state machine
FxD	fixed point
GE	gate equivalent
GF	Galois field
GT	ground truth
HA	half adder
HLS	high-level synthesis
HPC	high-performance computing
HSPA+	Evolved High Speed Packet Access
HW	hardware
IA	interval arithmetic
IC	integrated circuit
IEEE	Institute of Electrical and Electronics Engineers
ISA	instruction set architecture
LNS	Logarithmic Number System
LR	lattice reduction

LR FSD	lattice-reduced FSD
LSB	least significant bit
LTE	long-term evolution
LUT	lookup table
MA	multiply-add
MAC	multiply-accumulate
MIMO	multiple-input and multiple-output
ML	maximum likelihood
MLE	maximum likelihood estimation
MMSE	minimum mean square error
MNAE	mean normalized absolute error
MRS	mean (number of) remaining solutions
MSB	most significant bit
MU	multi-user
MUX	multiplexer
MVM	matrix-vector-multiplication
NaN	Not a Number
NaR	Not a Real
NAE	normalized absolute error
NN	neural network
NP	non-deterministic polynomial-time
PSK	phase-shift keying
OP	operation
QAM	quadrature amplitude modulation
QPSK	quadrature phase-shift keying
QRD	QR decomposition
RAM	random-access memory
RCA	ripple-carry adder
RF	radio frequency
RISC	Reduced Instruction Set Computer
RNS	Residue Number System
ROM	read-only memory
RTL	register-transfer level
SC	stochastic computing
SD	sphere decoder
SE	Schnorr-Euchner
SIC	soft interference cancellation
SN	stochastic number
SNR	signal-to-noise ratio
SOI	silicon on insulator
SORN	Sets Of Real Numbers
SOTA	state-of-the-art
SQRD	sorted QR-decomposition
sqrt	square root
STM	STMicroelectronics
SVD	singular value decomposition
TSMC	Taiwan Semiconductor Manufacturing Company Limited
ulp	unit of least precsision
unum	universal number
VHDL	Very High Speed Integrated Circuit Hardware Description Langauge
WNS	worst negative slack
ZF	zero forcing

List of Figures

List of Tables

Bibliography

The following lists contain the authors publications in chronological and the references in alphabetical order.

Publications

[1] M. Bärthel, J. Rust, and S. Paul. Hardware Implementation of Basic Arithmetics and Elementary Functions for Unum Computing. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 125–129, Oct 2018.

[2] M. Bärthel, P. Seidel, J. Rust, and S. Paul. SORN Arithmetic for MIMO Symbol Detection - Exploration of the Type-2 Unum Format. In *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*. IEEE, 23-26 June 2019.

[3] M. Bärthel, J. Rust, and S. Paul. Application-Specific Analysis of Different SORN Datatypes for Unum Type-2-Based Arithmetic. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[4] M. Bärthel, J. Rust, and S. Paul. Combining Fixed-Point and SORN Arithmetic in a MIMO BPSK-Symbol Detection Architecture. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 181–184, 2020.

[5] J. Rust, M. Bärthel, P. Seidel, and S. Paul. A Hardware Generator for SORN Arithmetic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4842–4853, 2020.

[6] S. Knobbe, M. Bärthel, S. Paul, and J. Rust. Complexity Reduction for Sphere Decoding using Unum-Type-II-Based SORN-Arithmetic. In *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, 2020.

[7] M. Bärthel, S. Knobbe, J. Rust, and S. Paul. Hardware Implementation of a Latency-Reduced Sphere Decoder With SORN Preprocessing. *IEEE Access*, 9:91387–91401, 2021.

[8] M. Bärthel, N. Hülsmeier, J. Rust, and S. Paul. On the Implementation of Edge Detection Algorithms with SORN Arithmetic. In *Next Generation Arithmetic: Third International Conference, CoNGA 2022, Singapore, March 1–3, 2022, Revised Selected Papers*, pages 1–13. Springer, 2022.

[9] M. Bärthel, J. Rust, J. Gustafson, and S. Paul. Improving the Precision of SORN Arithmetic by Introducing Fused Operations. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 258–262, 2022.

[10] M. Bärthel, C. Yuxing, N. Hülsmeier, J. Rust, and S. Paul. Fused Three-Input SORN Arithmetic. In *Next Generation Arithmetic: 4th International Conference, CoNGA 2023, Singapore, March 1-2, 2023, Proceedings*, pages 101–113. Springer, 2023.

References

[ACJ20] Mark Arnold, Ed Chester, and Corey Johnson. Training Neural Nets using only an Approximate Tableless LNS ALU. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 69–72, 2020.

[ACS] Proceedings of the IEEE Symposium on Computer Arithmetic 1969-2019. `http://www.acsel-lab.com/arithmetic/`, Accessed: 2023-11-09.

[AH13] Armin Alaghi and John P. Hayes. Survey of Stochastic Computing. *ACM Transactions on Embedded Computing Systems*, 12(2s), may 2013.

[AMF$^+$19] Ankur Agrawal, Silvia M. Mueller, Bruce M. Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 92–95, 2019.

[AMFM11] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour Detection and Hierarchical Image Segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(5):898–916, May 2011.

[AOS09] S. Aslan, E. Oruklu, and J. Saniie. Realization of area efficient QR factorization using unified division, square root, and inverse square root hardware. In *2009 IEEE International Conference on Electro/Information Technology*, pages 245–250, 2009.

[AQH18] Armin Alaghi, Weikang Qian, and John P. Hayes. The Promise and Challenge of Stochastic Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1515–1531, 2018.

[Arn22] Mark G. Arnold. Towards Quantum Logarithm Number Systems. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 76–83, 2022.

[BAPA16] Abdelkader Ben Amara, Edwige Pissaloux, and Mohamed Atri. Sobel edge detection system design and integration on an FPGA based HD video streaming architecture. In *2016 11th International Design & Test Symposium (IDT)*, pages 160–164, 2016.

[Bär18] Moritz Bärthel. Architecture and Hardware Implementation of Basic Arithmetics and Elementary Functions for UNUM Computing. Master's thesis, University of Bremen, Bremen, Germany, march 2018.

[Bär19] Moritz Bärthel. A Hardware Generator for SORN Arithmetic. GitHub repository, `https://github.com/mbaerthel/sorngen`, 2019.

[BBW$^+$05] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, and H. Bolcskei. VLSI implementation of MIMO detection using the sphere decoding algorithm. *IEEE Journal of Solid-State Circuits*, 40(7):1566–1577, 2005.

[BDdD17] Andrea Bocco, Yves Durand, and Florent de Dinechin. Hardware support for UNUM floating point arithmetic. In *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, pages 93–96, 2017.

[BDdD19] Andrea Bocco, Yves Durand, and Florent de Dinechin. SMURF: Scalar Multiple-Precision Unum Risc-V Floating-Point Accelerator for Scientific Computing. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, CoNGA'19, New York, NY, USA, 2019. Association for Computing Machinery.

[Beu08] Marcel Beuler. CORDIC-Algorithmus zur Auswertung elementarer Funktionen in Hardware. Technical report, Fachhochschule Gießen-Friedberg, 2008.

[BGLP15] Farid Bounini, Denis Gingras, Vincent Lapointe, and Herve Pollart. Autonomous Vehicle and Real Time Road Lanes Detection and Tracking. In *2015 IEEE Vehicle Power and Propulsion Conference (VPPC)*, pages 1–6, 2015.

[BHEHZ16] Ibrahim A. Bello, Basel Halak, Mohammed El-Hajjar, and Mark Zwolinski. A Survey of VLSI Implementations of Tree Search Algorithms for MIMO Detection. *Circuits, Systems, and Signal Processing*, 35(10):3644–3674, dec 2016.

[BJC$^+$19] Andrea Bocco, Tiago T. Jost, Albert Cohen, Florent de Dinechin, Yves Durand, and Christian Fabre. Byte-Aware Floating-point Operations through a UNUM Computing Unit. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 323–328, 2019.

[Bla16] Steve Blank. What the GlobalFoundries' Retreat Really Means - Things will never be the same for consumer devices. *IEEE Spectrum*, 2016. `https://spectrum.ieee.org/what-globalfoundries-retreat-really-means`, Accessed: 2023-11-13.

[BSMM08] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, 7th edition, 2008.

[BT08] L. G. Barbero and J. S. Thompson. Fixing the Complexity of the Sphere Decoder for MIMO Detection. *IEEE Transactions on Wireless Communications*, 7(6):2131–2142, 2008.

[BWA+12] F. Borlenghi, E. M. Witte, G. Ascheid, H. Meyr, and A. Burg. A 2.78 mm^2 65 nm CMOS gigabit MIMO iterative detection and decoding receiver. In *2012 Proceedings of the ESSCIRC (ESSCIRC)*, pages 65–68, Sep. 2012.

[Can86] John Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[CGS+18] Rohit Chaurasiya, John L. Gustafson, Rahul Shrestha, Jonathan Neudorfer, Sangeeth Nambiar, Kaustav Niyogi, Farhad Merchant, and Rainer Leupers. Parameterized Posit Arithmetic Hardware Generator. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 334–341, 2018.

[CM14] A. Chauhan and R. Mehra. Analysis of QR Decomposition for MIMO Systems. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 69–73, 2014.

[CoN18] *CoNGA '18: Proceedings of the Conference for Next Generation Arithmetic*, New York, NY, USA, 2018. Association for Computing Machinery.

[CP13] Manik Chugh and Behrooz Parhami. Logarithmic Arithmetic as an Alternative to Floating-Point: A Review. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 1139–1143, 2013.

[Cro16] Tim Cross. After Moore's Law. *The Economist Technology Quarterly*, 2016. `https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law`, Accessed: 2023-11-13.

[CRR+21] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, Sergio Saponara, and Benoit Dupont de Dinechin. Novel Arithmetics in Deep Neural Networks Signal Processing for Autonomous Driving: Challenges and Opportunities. *IEEE Signal Processing Magazine*, 38(1):97–110, 2021.

[CSK+08] John N. Coleman, Christopher I. Softley, Jiri Kadlec, Rudolf Matousek, Milan Tichy, Z. Pohl, Antonin Hermanek, and Nico F. Benschop. The European Logarithmic Microprocesor. *IEEE Transactions on Computers*, 57(4):532–546, 2008.

[CW07] Chiung-Jang Chen and Li-Chun Wang. Performance Analysis of Scheduling in Multiuser MIMO Systems with Zero-Forcing Receivers. *IEEE Journal on Selected Areas in Communications*, 25(7), 2007.

[CWHY08] Wei Cui, Guoliang Wu, Rongjin Hua, and Hao Yang. The research of edge detection algorithm for Fingerprint images. In *2008 World automation congress*, pages 1–5. IEEE, 2008.

[DBS06] Jean-Pierre Deschamps, Gery J. A. Bioul, and Gustavo D. Sutter. *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems.* Wiley-Interscience, USA, 2006.

[DdD03] J. Detrey and F. de Dinechin. A VHDL Library of LNS Operators. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 2227–2231 Vol.2, 2003.

[dDFMU19] Florent de Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. Posits: The Good, the Bad and the Ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, CoNGA'19, New York, NY, USA, 2019. Association for Computing Machinery.

[DJPM09] Erik Dahlman, Ylva Jading, Stefan Parkvall, and Hideshi Murai. 3G Radio Access Evolution - HSPA and LTE for Mobile Broadband. *IEICE Transactions on Communications*, E92.B(5):1432–1440, 2009.

[DM03] Paul E. Dodd and Lloyd W. Massengill. Basic Mechanisms and Modeling of Single-Event Upset in Digital Microelectronics. *IEEE Transactions on Nuclear Science*, 50(3):583–602, 2003.

[DSTH$^+$23] Himeshi De Silva, Hongshi Tan, Nhut-Min Ho, John L. Gustafson, and Weng-Fai Wong. Towards a Better 16-Bit Number Representation for Training Neural Networks. In *Proceedings of the Conference for Next Generation Arithmetic*, 2023.

[DT13] Jules R. Dim and Tamio Takamura. Alternative approach for satellite cloud classification: edge gradient application. *Advances in Meteorology*, 2013.

[EL04] Milos D. Ercegovac and Tomas Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[Far04] Hassan A. Farhat. *Digital Design and Computer Organization*. CRC Press, 2004.

[FRG$^+$09] J. Fink, S. Roger, A. Gonzalez, V. Almenar, and V. M. Garciay. Complexity assessment of sphere decoding methods for MIMO detection. In *2009 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 9–14, 2009.

[Gat19a] Ioannis Gatopoulos. Autonomous Car. GitHub repository, `https://github.com/ioangatop/AutonomousCar`, 2019.

[Gat19b] Ioannis Gatopoulos. Line Detection: Make an Autonomous Car see Road Lines. *towards data science*, 2019. `https://towardsdatascience.com/line-detection-make-an-autonomous-car-see-road-lines-e3ed984952c`, Accessed: 2021-12-07.

[GEOA06] Ruchir Gupte, William Edmonson, Senanu Ocloo, and Winser Alexander. Pipelined ALU for Signal Processing to Implement Interval Arithmetic. In *2006 IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 95–100, 2006.

[GK16] John L. Gustafson and William Kahan. The Great Debate @ARITH23. `https://www.youtube.com/watch?v=LZAeZBVAzVw`, Accessed: 2023-11-13, 2016.

[GLS$^+$13] R. Gangarajaiah, L. Liu, M. Stala, P. Nilsson, and O. Edfors. A high-speed QR decomposition processor for carrier-aggregated LTE-A downlink systems. In *2013 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, 2013.

[GMR$^+$18] Florian Glaser, Stefan Mach, Abbas Rahimi, Frank K. Gürkaynak, Qiuting Huang, and Luca Benini. An 826 MOPS, 210uW/MHz Unum ALU in 65 nm. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[GNJ17] Georgios Georgis, Konstantinos Nikitopoulos, and Kyle Jamieson. Geosphere: An exact depth-first sphere decoder architecture scalable to very dense constellations. *IEEE Access*, 5:4233–4249, 2017.

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[Gom09] Interval Arithmetic. In Abel J. P. Gomes, Irina Voiculescu, Joaquim Jorge, Brian Wyvill, and Callum Galbraith, editors, *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*, pages 89–115. Springer London, London, 2009.

[Gun23] Thushara Kanchana Gunaratne. Evaluation of the Use of Low Precision Floating-Point Arithmetic for Applications in Radio Astronomy. In *Next Generation Arithmetic: 4th International Conference, CoNGA 2023, Singapore, March 1-2, 2023, Proceedings*. Springer, 2023.

[Gus15] John L. Gustafson. *The end of error: Unum computing*. Chapman & Hall / CRC computational science series. CRC Press, Boca Raton, 2015.

[Gus16] John L. Gustafson. A Radical Approach to Computation with Real Numbers. *Supercomputing Frontiers and Innovations*, 3(2), 2016.

[Gus17] John L. Gustafson. Posit Arithmetic - Mathematica Notebook describing the posit number system. Technical report, 2017. `https://posithub.org/docs/Posits4.pdf`, Accessed: 2023-10-12.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, USA, 1996.

[GY17] John L. Gustafson and Isaac T. Yonemoto. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4(2), 2017.

[HBK$^+$22] Nils Hülsmeier, Moritz Bärthel, Ludwig Karsthof, Jochen Rust, and Steffen Paul. Hybrid SORN Implementation of k-Nearest Neighbor Algorithm on FPGA. In *2022 20th IEEE Interregional NEWCAS Conference (NEWCAS)*, pages 163–167, 2022.

[HBRP21] Nils Hülsmeier, Moritz Bärthel, Jochen Rust, and Steffen Paul. SORN-based Cascade Support Vector Machine. In *2020 28th European Signal Processing Conference (EUSIPCO)*, pages 1507–1511. IEEE, 2021.

[HBRP23] Nils Hülsmeier, Moritz Bärthel, Jochen Rust, and Steffen Paul. Hybrid SORN Hardware Accelerator for Support Vector Machines. In *Next Generation Arithmetic: 4th International Conference, CoNGA 2023, Singapore, March 1-2, 2023, Proceedings*, pages 77–87. Springer, 2023.

[HCW15] Y. Hwang, K. Chen, and C. Wu. A high throughput unified SVD/QRD precoder design for MIMO OFDM systems. In *2015 IEEE International Conference on Digital Signal Processing (DSP)*, pages 1148–1151, 2015.

[HH13] David Money Harris and Sarah L. Harris. *Digital Design and Computer Architecture.* Morgan Kaufmann, Boston, 2nd edition, 2013.

[HHKR12] Rolf Hammer, Matthias Hocks, Ulrich Kulisch, and Dietmar Ratz. *C++ Toolbox for Verified Computing I: Basic Numerical Problems Theory, Algorithms, and Programs.* Springer Science & Business Media, 2012.

[HLB$^+$20] Albert Sydney Hornby, Diana Lea, Jennifer Bradbery, Victoria Bull, Leonie Hey, Stacey Bateman, Kallah Pridgeon, and Gary Leicester. *Oxford Advanced Learner's Dictionary of Current English.* Oxford University Press, Oxford, 10th edition, 2020.

[HV05] B. Hassibi and H. Vikalo. On the sphere-decoding algorithm I. Expected complexity. *IEEE Transactions on Signal Processing*, 53(8):2806–2818, 2005.

[HZS$^+$17] Junjie Hou, Yongxin Zhu, Yulan Shen, Mengjun Li, Han Wu, and Han Song. Addressing Inefficiency of Floating-Point Operations in Cloud Computing: Implementation and a Case Study of Variable Precision Computing. In *2017 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 168–175, 2017.

[IEEE] IEEE Xplore Digital Library. `https://ieeexplore.ieee.org/`, Accessed: 2023-11-09.

[IEEE85] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985.

[IEEE08] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[IEEE09] IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC)and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput. *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)*, pages 1–565, 2009.

[IEEE18] IEEE Standard for Interval Arithmetic (Simplified). *IEEE Std 1788.1-2017*, pages 1–38, 2018.

[IEEE19] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[Ifr00] Georges Ifrah. *The Universal History of Numbers: From Prehistory to the Invention of the Computer.* Wiley, 2000.

[Ima] Image: Bremer Stadtmusikanten. Wikpedia Commons, `https://commons.wikimedia.org/wiki/File:Bremer_Stadtmusikanten._Bremen._IMG_6897WI.jpg`. Accessed: 2023-10-12.

[JJ43] Glenn James and Robert Clarke James. *Mathematics Dictionary.* Digest Press, Van Nuys, California, revised edition, 1943.

[Joh18] Jeff Johnson. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, 2018.

[JS19] Manish Kumar Jaiswal and Hayden K.-H. So. PACoGen: A Hardware Posit Arithmetic Core Generator. *IEEE Access*, 7:74586–74601, 2019.

[JSR18] W. Kenneth Jenkins, Michael A. Soderstrand, and Chandrasekhar Radhakrishnan. Historical Patterns of Emerging Residue Number System Technologies During the Evolution of Computer Engineering and Digital Signal Processing. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2018.

[Kah81] William Kahan. Why do we need a floating-point arithmetic standard? Technical report, UC Berkeley, 1981. `http://www.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf`, Accessed: 2023-11-13.

[Kam04] Karl-Dirk Kammeyer. *Nachrichtenübertragung.* Vieweg + Teubner Verlag, 3rd edition, 2004.

[KCCW02] Kwan-wai Wong, Chi-ying Tsui, R. S. K. Cheng, and Wai-ho Mow. A VLSI architecture of a K-best lattice decoding algorithm for MIMO channels. In *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*, volume 3, pages III–III, 2002.

[KK06] Reinhard Kirchner and Ulrich W. Kulisch. Hardware Support for Interval Arithmetic. *Reliable Computing*, 12(3):225–237, 2006.

[Kno20] Simon Knobbe. Conceptual Design and Implementation of a Sphere Decoder with a SORN-Reduced Solution Set. Master's thesis, University of Bremen, Bremen, Germany, march 2020.

[KPLK14] H. Kim, J. Park, H. Lee, and J. Kim. Near-ML MIMO Detection Algorithm With LR-Aided Fixed-Complexity Tree Searching. *IEEE Communications Letters*, 18(12):2221–2224, 2014.

[KSH23] Ahmed S. Khalil, Mohamed Shalaby, and Emad Hegazi. An Enhanced System on Chip-Based Sobel Edge Detector. In *2023 International Telecommunications Conference (ITC-Egypt)*, pages 179–183, 2023.

[Kul09] Ulrich W. Kulisch. Complete Interval Arithmetic and Its Implementation on the Computer. In Annie Cuyt, Walter Krämer, Wolfram Luther, and Peter Markstein, editors, *Numerical Validation in Current Hardware Architectures*, pages 7–26, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Lar09] E. G. Larsson. MIMO Detection Methods: How They Work [Lecture Notes]. *IEEE Signal Processing Magazine*, 26(3):91–95, May 2009.

[Lic] CC BY-SA 3.0 DEED: Creative Commons Attribution-Share Alike 3.0 Unported. Creative Commons License, `https://creativecommons.org/licenses/by-sa/3.0/`. Accessed: 2023-10-12.

[LLH18] Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. Universal Coding of the Reals: Alternatives to IEEE Floating Point. In *Proceedings of the Conference for Next Generation Arithmetic*, pages 1–14, 2018.

[LLN12] L. Liu, J. Lofgren, and P. Nilsson. Area-Efficient Configurable High-Throughput Signal Detector Supporting Multiple MIMO Modes. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(9):2085–2096, 2012.

[LMDBB13] Carlos Lopez-Molina, Bernard De Baets, and Humberto Bustince. Quantitative error measures for edge detection. *Pattern Recognition*, 46(4):1125–1139, 2013.

[Man02] M. Morris Mano. *Digital Design*. Prentice-Hall, Upper Saddle River, NJ, 3rd edition, 2002.

[MASC17] Pedro Miguens Matutino, Juvenal Araújo, Leonel Sousa, and Ricardo Chaves. Pipelined FPGA coprocessor for Elliptic Curve Cryptography based on Residue Number System. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 261–268, 2017.

[MBdD$^+$18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2nd edition, 2018.

[MDBB20] Raul Murillo, Alberto A. Del Barrio, and Guillermo Botella. Customized Posit Adders and Multipliers using the FloPoCo Core Generator. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[MDBB$^+$22] Raul Murillo, Alberto A. Del Barrio, Guillermo Botella, Min Soo Kim, HyunJin Kim, and Nader Bagherzadeh. PLAM: A Posit Logarithm-Approximate Multiplier. *IEEE Transactions on Emerging Topics in Computing*, 10(4):2079–2085, 2022.

[Mit62] John N. Mitchell. Computer Multiplication and Division Using Binary Logarithms. *IRE Transactions on Electronic Computers*, EC-11(4):512–517, 1962.

[MK85] David W. Matula and Peter Kornerup. Finite Precision Rational Arithmetic: Slash Number Systems. *IEEE Transactions on Computers*, C-34(1):3–18, 1985.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis.* Society for Industrial and Applied Mathematics, 2009.

[MMB$^+$22] David Mallasén, Raul Murillo, Alberto A. Del Barrio, Guillermo Botella, Luis Piñuel, and Manuel Prieto-Matias. PERCIVAL: Open-Source Posit RISC-V Core With Quire Capability. *IEEE Transactions on Emerging Topics in Computing*, 10(3):1241–1252, 2022.

[MMDBB21] Raul Murillo, David Mallasén, Alberto A. Del Barrio, and Guillermo Botella. Energy-Efficient MAC Units for Fused Posit Arithmetic. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 138–145, 2021.

[Moo06] Gordon E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[MS20] Paulo Martins and Leonel Sousa. The Role of Non-Positional Arithmetic on Efficient Emerging Cryptographic Algorithms. *IEEE Access*, 8:59533–59549, 2020.

[Mul05] Jean-Michel Muller. On the definition of ulp(x). Research Report RR-5504, LIP RR-2005-09, INRIA, LIP, February 2005.

[NGW10] Gabriel Luca Nazar, Christina Gimmler, and Norbert Wehn. Implementation Comparisons of the QR Decomposition for MIMO Detection. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design*, SBCCI '10, page 210–214, New York, NY, USA, 2010. Association for Computing Machinery.

[NSWvG12] Marco Nehmeier, Stefan Siegel, and Jürgen Wolff von Gudenberg. Specification of hardware for interval arithmetic. *Computing*, 94:243–255, 2012.

[Obe07] Erick L. Oberstar. Fixed-Point Representation & Fractional Math Revison 1.2. *Oberstar Consulting*, 2007.

[O'R08] Gerard O'Regan. *A Brief History of Computing.* Springer, 3rd edition, 2008.

[Par03] Behrooz Parhami. Number Representation and Computer Arithmetic. In *Encyclopedia of Information Systems*, pages 317 – 333. Elsevier, 2003.

[Par10] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, Inc., USA, 2nd edition, 2010.

[Pir96] Peter Pirsch. *Architekturen der digitalen Signalverarbeitung.* B. G. Teubner Stuttgart, 1996.

[PM18] Artur Podobas and Satoshi Matsuoka. Hardware Implementation of POSITs and Their Application in FPGAs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 138–145, 2018.

[Pos22] Standard for Posit™ Arithmetic (2022). *Posit Working Group*, pages 1–12, 2022. `https://posithub.org/docs/posit_standard-2.pdf`, Accessed: 2023-11-13.

[PSSG10] D. Patel, V. Smolyakov, M. Shabany, and P. G. Gulak. VLSI implementation of a WiMAX/LTE compliant low-complexity high-throughput soft-output K-Best MIMO detector. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 593–596, 2010.

[PU20] Larry D. Pyeatt and William Ughetta. Chapter 8 - Non-integral mathematics. In Larry D. Pyeatt and William Ughetta, editors, *ARM 64-Bit Assembly Language*, pages 239–292. Newnes, 2020.

[QLR+11] Weikang Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja. An Architecture for Fault-Tolerant Computation with Stochastic Logic. *IEEE Transactions on Computers*, 60(1):93–105, 2011.

[RG08] Gallager Robert G. *Principles of Digital Communication.* Cambridge University Press, 2008.

[RK22] Marko Radosavljevic and Jack Kavalieros. Taking Moore's Law to New Heights: When transistors can't get any smaller, the only direction is up. *IEEE Spectrum*, 59(12):32–37, 2022.

[RLP13] Jochen Rust, Frank Ludwig, and Steffen Paul. Low Complexity QR-Decomposition Architecture using the Logarithmic Number System. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 97–102, 2013.

[RLP15] Jochen Rust, Frank Ludwig, and Steffen Paul. QR-Decomposition Architecture Based on Two-Variable Numeric Function Approximation. In *Design, Automation Test in Europe Conference Exhibition*, 2015.

[ROP11] J. Rust, C. Osewold, and S. Paul. Implementation of a Low Power Low Complexity ASIP for various Sphere Decoding Algorithms. In *17th European Wireless 2011 - Sustainable Wireless Technologies*, pages 1–6, April 2011.

[RPL+13] F. Rusek, D. Persson, B. K. Lau, E. G. Larsson, T. L. Marzetta, O. Edfors, and F. Tufvesson. Scaling Up MIMO: Opportunities and Challenges with Very Large Arrays. *IEEE Signal Processing Magazine*, 30(1):40–60, Jan 2013.

[RSL+21] Aleksandr Yu. Romanov, Alexander L. Stempkovsky, Ilia V. Lariushkin, Georgy E. Novoselov, Roman A. Solovyev, Vladimir A. Starykh, Irina I. Romanova, Dmitry V. Telpukhov, and Ilya A. Mkrtchan. Analysis of Posit and Bfloat Arithmetic of Real Numbers for Machine Learning. *IEEE Access*, 9:82318–82324, 2021.

[RTAF21] Arman Roohi, MohammadReza Taheri, Shaahin Angizi, and Deliang Fan. RNSiM: Efficient Deep Neural Network Accelerator Using Residue Number Systems. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[Rum10] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, 2017.

[SA75] E.E. Swartzlander and A.G. Alexopoulos. The Sign/Logarithm Number System. *IEEE Transactions on Computers*, C-24(12):1238–1242, 1975.

[SB11] Chris Solomon and Toby Breckon. *Fundamentals of Digital Image Processing: A practical approach with examples in Matlab.* John Wiley & Sons, 2011.

[SBB08] C. Studer, A. Burg, and H. Bolcskei. Soft-output sphere decoding: algorithms and VLSI implementation. *IEEE Journal on Selected Areas in Communications*, 26(2):290–300, 2008.

[SE94] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994.

[SFS11] Christoph Studer, Schekeb Fateh, and Dominik Seethaler. ASIC Implementation of Soft-Input Soft-Output MIMO Detection using MMSE Parallel Interference Cancellation. *IEEE Journal of Solid-State Circuits*, 46(7):1754–1765, 2011.

[SJJT86] Michael A. Soderstrand, W. Kenneth Jenkins, Graham A. Jullien, and Fred J. Taylor. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing.* IEEE press, 1986.

[Sob14] Irwin Sobel. An Isotropic 3x3 Image Gradient Operator. *Presentation at Stanford A.I. Project 1968*, 2014.

[Sou21] Leonel Sousa. Nonconventional Computer Arithmetic Circuits, Systems and Applications. *IEEE Circuits and Systems Magazine*, 21(1):6–40, 2021.

[Spe21] Joachim Speidel. *Introduction to Digital Communications.* Springer Cham, 2nd edition, 2021.

[Szp13] George Szpiro. Kleine Rundungsfehler mit katastrophalen Folgen. *Neue Züricher Zeitung NZZ*, october 2013. `https://www.nzz.ch/wissenschaft/hintergrund/kleine-rundungsfehler-mit-katastrophalen-folgen-ld.830064`, Accessed: 2023-08-02.

[TGRK21] Sugandha Tiwari, Neel Gala, Chester Rebeiro, and V. Kamakoti. PERI: A Configurable Posit Enabled RISC-V Core. *ACM Trans. Archit. Code Optim.*, 18(3), apr 2021.

[TS17] T. Tseng and C. Shen. The VLSI architecture of a highly efficient configurable pre-processor for MIMO detections. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–5, 2017.

[UFdD19] Yohann Uguen, Luc Forget, and Florent de Dinechin. Evaluating the Hardware Cost of the Posit Number System. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 106–113, 2019.

[Wal71] J.S. Walther. A Unified Algorithm for Elementary Functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, pages 379–385. ACM, 1971.

[Wal16] M. Mitchell Waldrop. The chips are down for Moore's law. *nature*, 2016. `https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338`, Accessed: 2023-11-13.

[WF03] Christoph Windpassinger and Robert FH Fischer. Low-Complexity Near-Maximum-Likelihood Detection and Precoding for MIMO Systems using Lattice Reduction. In *Information Theory Workshop, 2003. Proceedings. 2003 IEEE*, pages 345–348. IEEE, 2003.

[WK19] Shibo Wang and Pankaj Kanwar. BFloat16: The secret to high performance on Cloud TPUs. *Google Cloud*, 2019. `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus?hl=en`, Accessed: 2023-04-12.

[WRB$^+$02] Dirk Wübben, Jürgen Rinas, R. Boehnke, Volker Kuehn, and K. Kammeyer. Efficient Algorithm for Detecting Layered Space-Time Codes. In *4th International ITG Conference on Source and Channel Coding (SCC)*, January 2002.

[YH15] Shaoshi Yang and Lajos Hanzo. Fifty Years of MIMO Detection: The Road to Large-Scale MIMOs. *IEEE Communications Surveys & Tutorials*, 17(4):1941–1988, 2015.

[YTC$^+$13] K. Yang, S. Tsai, R. Chang, Y. Chen, and G. C. H. Chuang. VLSI implementation of a low complexity 4x4 MIMO sphere decoder with table enumeration. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2167–2170, 2013.

[Yux23] Chen Yuxing. Implementation Of Road Lane Detection Using SORN-based Edge Detection. Master's thesis, University of Bremen, Bremen, Germany, April 2023.

[YYW21] Xiaokun Yang, T. Andrew Yang, and Lei Wu. An Edge Detection IP of Low-Cost System on Chip for Autonomous Vehicles. In Hamid R. Arabnia, Ken Ferens, David de la Fuente, Elena B. Kozerenko, José Angel Olivas Varela, and Fernando G. Tinetti, editors, *Advances in Artificial Intelligence and Applied Cognitive Computing*, pages 775–786, Cham, 2021. Springer International Publishing.

[ZKAKP22] Mohamadreza Zolfagharinejad, Mehdi Kamal, Ali Afzali-Khusha, and Massoud Pedram. Posit Process Element for Using in Energy-Efficient DNN Accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(6):844–848, 2022.