Fachhochschule
Dortmund
University of Applied Sciences and Arts

Bachelorthesis

VHDL-Implementierung der Arkussinusfunktion und der Division von Festkommazahlen nach dem CORDIC Algorithmus

vorgelegt von

Sofiene Tijani

Angefertigt im Studiengang Elektrotechnik der FH Dortmund

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis
Zweitprüfer: Dipl.-Ing. Rolf Paulus

Abstract

Im Rahmen dieser Arbeit werden digitale Schaltungen für die Berechnung von Quotienten und die Auswertung der Arkussinusfunktion entworfen und implementiert. Da diese für die Entwicklung eines kompakten Winkelsensors benötigt werden , wird für die Realisierung der CORDIC-Algotihmus verwendet, welcher die Umsetzung und die Funktionsauswertung mit geringem Hardwareaufwand auf einem FPGA erlaubt. Für die beiden Operationen wird in VHDL jeweils ein Modul entworfen und simuliert und abschließend auf einem Testboard überprüft. Durch die Simulation und die Tests wird die korrekte Funktion des Entwurfs sowie dessen Genauigkeit bei der Berechnung über einen weiten Arbeitsbereich verifiziert.

The scope of this thesis is the design and implementation of processors for the computation of quotients and the inverse sine function. These are required for the development of a compact angle sensor. Hence, the CORDIC-algorithm is used for this task, because it allows the implementation of function evaluation with little hardware effort on an FPGA. For each of the two operations, a module is designed and simulated in VHDL and then checked on a test board. The simulation and the tests verify the correct function of the design and its accuracy in the calculation over a wide working range.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADC	**A**nalog **D**igital **C**onverter
CMOS	**C**omplementary **M**etal-**O**xide-**S**emiconductor
CORDIC	**Co**ordinate **R**otation **Di**gital **C**omputer
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
POLDI	**pol**arisierende Photo**di**oden
ROM	**R**ead **O**nly **M**emory
VHDL	**V**ery **H**igh Speed Integrated Circuit Hardware **D**escription **L**anguage
VLSI	**V**ery **L**arge **S**cale **I**ntegration

Symbolverzeichnis

Symbol	Bedeutung
d	Einscheidungsparameter
K	Grenzwert Skalierungsfaktor für unendlich viele Iterationen
k_n	Proportionalitätsfaktor für den n-ten Schritt
K_n	Skalierungsfaktor nach n Iterationen
r	Polarkoordinate – Abstand zum Ursprung
x	kartesische Koordinate
y	kartesische Koordinate
φ	Drehwinkel
ξ	Argument Arkusfunktion oder Areafunktion

1 Einleitung

Winkelsensoren bzw. Drehgeber sind wichtige Sensortypen zur Erfassung geometrischer Größen. Sie finden sich dort, wo Achsen drehen oder rotatorische in lineare Bewegung umgesetzt wird. Dies eröffnet einen weiten Anwendungsbereich, bspw. im Maschinen- und Anlagenbau, Lager- und Fördertechnik, der Automobilindustrie, Verpackungsmaschinen, Robotik und Automationstechnik etc.[1] Für die Realisierung können unterschiedliche physikalische Konzepte herangezogen werden. Unter diesen zeichnen sich optische Winkelsensoren im Allgemeinen durch eine genaue und berührungslose Messung aus. [1]

Eine Neuentwicklung auf diesem Feld ist der sogenannte POLDI-Sensor. Das Akronym POLDI steht für polarisierende Photodioden und beschreibt das zugrundeliegende Messkonzept, bei dem die Stellung eines Polarisationsfilters durch Photodioden erfasst wird. Dies ermöglicht einen ebenso genauen, wie kompakten und energieeffizienten Sensor.[2] Der Aufbau des POLDI-Sensors ist in Abbildung 1 schematisch dargestellt.

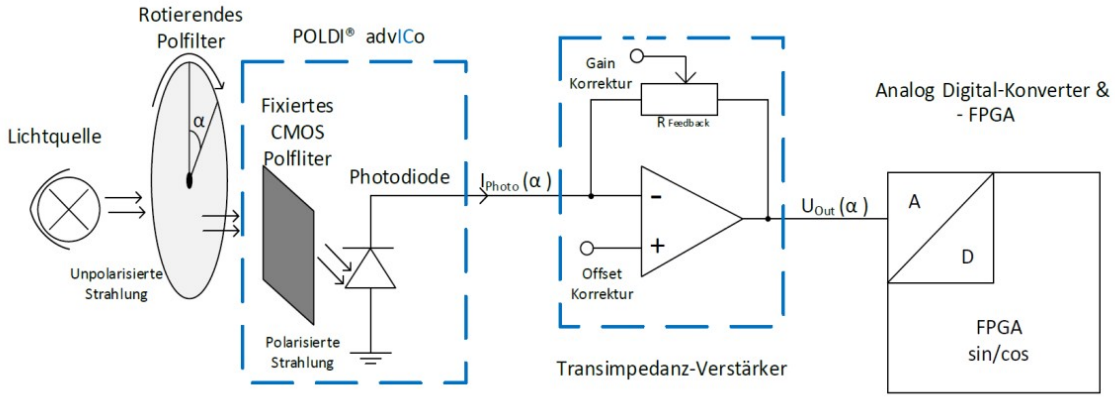

Abbildung 1: Aufbau des „POLDI“-Winkelsensors [3]

Das rotierende Polarisationsfilter ist mit dem rotierenden Messobjekt gekoppelt und polarisiert Licht, welches von einer Leuchtdiode emittiert wird. Somit entspricht der Winkel der Polarisation der Winkelposition des Objekts. Zur Erfassung werden vier Photodioden mit separaten CMOS-Polarisationsfiltern verwendet, wobei sich die Richtung dieser Filter jeweils um 45° unterscheidet. Die Intensität des Lichtes, welches auf die einzelnen Photodioden einfällt, und damit der Photostrom, den sie abgeben, wird durch die Stellung des rotierenden Polarisationsfilters bestimmt. Der Photostrom stellt das eigentliche Messsignal dar, welches mittels eines Transimpedanzverstärkers auf

die gewünschte Spannung und anschließend durch einen Analog-Digital-Wandler (ADC) transformiert und mithilfe eines Field Programmable Gate Array (FPGA) verarbeitet wird.

Mathematisch gesehen, erfordert die Verarbeitung im FPGA zur Bestimmung des Winkels verschiedene mathematische Operationen, wie Multiplikation, Division oder die Auswertung der trigonometrischen Funktionen und ihrer Umkehrfunktionen. Im Rahmen dieser Arbeit wird einer Teil der benötigten Funktionen für den FPGA implementiert und zwar die Berechnung von Quotienten und der Arkussinusfunktion.

Hierzu gibt es unterschiedliche Realisierungsmöglichkeiten. Schwerpunkt ist wegen der angestrebten Kompaktheit des POLDI-Sensors eine möglichst einfache und mit geringem Hardwareaufwand verbundene Lösung. Daher wird der CORDIC-Algorithmus für die Realisierung der oben genannten mathematische Operationen verwendet, da dieser die Umsetzung allein mithilfe von Addition/Subtraktion und Vierschiebeoperationen erlaubt. Ziel der Arbeit ist es somit, die Berechnung von Division und des Arkussinus mithilfe des CORDIC-Algortihmus in VHDL zu entwerfen und auf dem FPGA zu implementieren.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen des CORDIC-Algorithmus erläutert. Ausgangspunkt ist die Beschreibung des grundlegenden Ansatzes zur Drehung von Vektoren. Daran anschließend wird die Verallgemeinerung des Algorithmus auf andere Vektoroperationen erläutert. Danach wird erörtert, wie der Algorithmus genutzt werden kann, um diverse Funktionen zu berechnen. Den Abschluss der theoretischen Darstellung bildet eine Diskussion der Vor- und Nachteile des Algorithmus in der praktischen Anwendung.

2.1 Grundidee

Die Idee hinter dem CORDIC-Algorithmus besteht darin, die Phase einer komplexen Zahl durch Multiplikation mit einer Folge konstanter Werte zu „drehen“. Für diese Multiplikationen werden ausschließlich Faktoren verwendet, die eine Potenz der Zahl 2 sind. Dadurch lassen sich diese Multiplikationen in binärer Arithmetik mit Verschiebungen und Additionen ausdrücken, wodurch der Schaltungsaufwand für die Realisierung eines Multiplizierers vermieden wird.

Die Grundlage des CORDIC-Algorithmus ist die Rotation eines Vektors. Ausgangspunkt ist die Darstellung der komplexen Zahlen in einem zweidimensionalen kartesischen Koordinatensystem (Abbildung 2.1), wobei der Realteil auf der x-Achse und der Imaginärteil auf der y-Achse dargestellt wird.

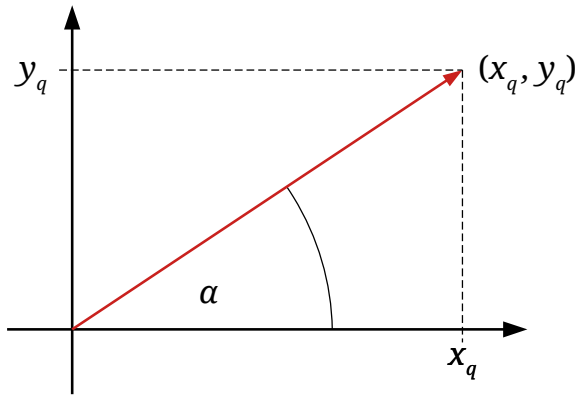

Abbildung 2.1: Kartesisches Koordinatensystem mit dem Punkt (x_q, y_q)

Ein Quellvektor (x_q, y_q) kann durch folgende Polarkoordinaten in diesem kartesischen Koordinatensystem wie folgt beschrieben werden:

$$x_q = r \cdot \cos(\alpha) \tag{2.1}$$

$$y_q = r \cdot \sin(\alpha) \quad . \tag{2.2}$$

Wird der Ortsvektor um den Winkel φ gedreht (Abbildung 2.2.), dann ergibt sich der neue Punkt (x_n, y_n) mit den Koordinaten:

$$x_n = r \cdot \cos(\alpha + \varphi) \tag{2.3}$$

$$y_n = r \cdot \sin(\alpha + \varphi) \quad . \tag{2.4}$$

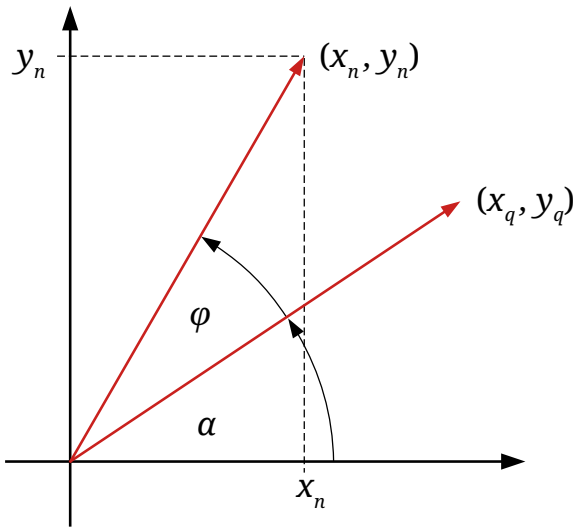

Abbildung 2.2.: Rotation eines Punktes um den Winkel φ

Durch Anwendung der Additionstheoreme für Sinus und Kosinus und Substitution der Definition der Ausgangskoordinaten nach Gleichung (2.1) und (2.2) folgt:[4] :

$$\begin{aligned} x_n &= r \cdot \cos(\alpha) \cdot \cos(\varphi) - r \cdot \sin(\alpha) \cdot \sin(\varphi) \\ &= x_q \cdot \cos(\varphi) - y_q \cdot \sin(\varphi) \end{aligned} \tag{2.5}$$

$$\begin{aligned} y_n &= r \cdot \cos(\alpha) \cdot \sin(\varphi) + r \cdot \sin(\alpha) \cdot \cos(\varphi) \\ &= x_q \cdot \sin(\varphi) + y_q \cdot \cos(\varphi) \end{aligned} \quad . \tag{2.6}$$

Das bedeutet, dass die Koordinaten nach der Drehung lassen sich durch eine einfach Linearkombination mithilfe der Winkelfunktionen sin(φ) und cos(φ) als Faktoren kalkulieren. Durch ein geschickte Wahl dieser Faktoren, kann die Berechnung der Drehung deutlich vereinfacht werden. Dies wird im folgenden Abschnitt erläutert.

2.2 Konzept und Grundgleichungen

Der CORDIC-Algorithmus wurde von Volder [5] einwickelt, um trigonometrische Funktionen zu berechnen. Durch Walther[6] wurde er um die Möglichkeit ergänzt, weitere elementare Funktionen wie Logarithmus, die Exponentialfunktion und die Quadratwurzel zu berechnen.

Ausgangspunkt für CORDIC-Algorithmen ist die zuvor erläuterte Berechnung der Rotation eines Quellvektors (x_q, y_q). Mithilfe der Definition für den Tangens kann der Sinus wie folgt substituiert werden:

$$\sin(\varphi) = \cos(\varphi) \cdot \tan(\varphi) \tag{2.7}$$

Durch Einsetzen in Gleichung (2.5) und (2.6) und anschließende Umformung ergibt sich:

$$x_n = \cos(\varphi)\left[x_q - y_q \cdot \tan(\varphi)\right] \tag{2.8}$$

$$y_n = \cos(\varphi)\left[y_q + x_q \cdot \tan(\varphi)\right] \quad . \tag{2.9}$$

Im nächsten Schritt wird der Tangens durch eine Potenz von 2 ersetzt, d. h.:

$$\tan(\varphi) = 2^{-i} \quad \text{mit} \quad i \in \mathbb{Z} \quad . \tag{2.10}$$

Diese Multiplikation mit eine Potenz von 2 entspricht bei der Berechnung im binären Zahlensystem das Hinzufügen bzw. Entfernen einer Bit Stelle. Infolgedessen entspricht diese Multiplikation bei der Auswertung durch einen digitalen Rechner einer einfachen Schiebeoperation. Zwar wird durch die Konvention nach Gleichung (2.10) die Anzahl der veränderbaren Winkel φ und somit die damit verbundenen elementaren Rotationen eingeschränkt. Allerdings kann eine beliebige Rotation durch eine Sequenz elementarer Rotationen mit hinreichender Genauigkeit erzeugt werden (siehe Abbildung 2.3.).

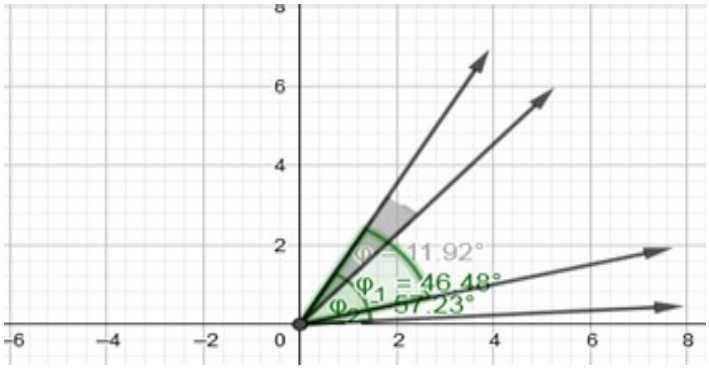

Abbildung 2.3.: Polarkoordinatensystem mit Vektoren des Winkels φ

Für jeden Teilschritt „i“ ist zu entscheiden, ob es sich um eine positive oder negative Komponente der Gesamtdrehung handelt. Das führt zu den folgenden Iterationen:

$$x_{(i+1)} = \cos\left(\varphi_i\right)\left[x_i - y_i \cdot d_i \cdot 2^{(-i)}\right] \tag{2.11}$$

$$y_{(i+1)} = \cos\left(\varphi_i\right)\left[y_i + x_i \cdot d_i \cdot 2^{(-i)}\right] \tag{2.12}$$

mit

$$d_i = \pm 1 \quad . \tag{2.13}$$

Allgemein gilt für einen beliebigen Winkel

$$\cos\left(\varphi\right) = \cos\ \left(-\varphi\right) \tag{2.14}$$

$$\varphi = \arctan\ \left(\tan\ \left(\varphi\right)\right) \quad . \tag{2.15}$$

Somit kann der Cosinus-Term in den Gleichungen (2.11) und (2.12) durch folgenden Ausdruck ersetzt werden:

$$\cos\left(\varphi_i\right) = \cos\ \left(\arctan\ \left(+2^{(-i)}\right)\right) = k_i \quad . \tag{2.16}$$

Durch in Einsetzen in Gleichung (2.11) und (2.12) folgt

$$x_{(i+1)} = k_i\left[x_i - y_i \cdot d_i \cdot 2^{(-i)}\right] \tag{2.17}$$

$$y_{(i+1)} = k_i\left[y_i + x_i \cdot d_i \cdot 2^{(-i)}\right] \quad . \tag{2.18}$$

Die im Laufe einer Iteration auftretenden Werte für k_i können zu einem Produkt zusammengefasst werden. Dies wird im Folgenden am Beispiel der ersten drei Schritte einer Iteration gezeigt[7]:

- Erster Iterationsschritt:

$$\begin{aligned} x_1 &= k_0\left[x_0 - y_0 \cdot d_0 \cdot 2^{(-0)}\right] = k_0\left[x_0 - y_0 \cdot d_0\right] \\ y_1 &= k_0\left[y_0 + x_0 \cdot d_0 \cdot 2^{(-0)}\right] = k_0\left[y_0 + x_0 \cdot d_0\right] \end{aligned} \tag{2.19}$$

- Zweiter Iterationsschritt ′

$$\begin{aligned} x_2 &= k_1\left[x_1 - y_1 \cdot d_{1i} \cdot 2^{(-1)}\right] = k_1 \cdot k_0\left[x_1' - y_1' \cdot d_1 \cdot 2^{(-1)}\right] \\ y_2 &= k_1\left[y_1 + x_1 \cdot d_1 \cdot 2^{(-1)}\right] = k_1 \cdot k_0\left[y_1' + x_1' \cdot d_1 \cdot 2^{(-1)}\right] \\ \text{mit} \quad x_1' &= \frac{x_1}{k_0} = x_0 - y_0 \cdot d_0 \quad \text{und} \quad y_1' = \frac{y_1}{k_0} = y_0 + x_0 \cdot d_0 \end{aligned} \tag{2.20}$$

- dritter Iterationsschritt

$$\begin{aligned} x_3 &= k_2\left[x_2 - y_2\cdot d_2\cdot 2^{(-2)}\right] = k_2\cdot k_1\cdot k_0\left[x_2^{l} - y_2^{l}\cdot d_2\cdot 2^{(-2)}\right] \\ y_2 &= k_2\left[y_2 + x_2\cdot d_2\cdot 2^{(-2)}\right] = k_2\cdot k_1\cdot k_0\left[y_2^{l} + x_2^{l}\cdot d_2\cdot 2^{(-2)}\right] \end{aligned} \tag{2.21}$$

$$\text{mit}\quad x_2^{l} = \frac{x_2}{k_1\cdot k_0} = x_1^{l} - y_1^{l}\cdot d_1\cdot 2^{(-1)}\cdot d_0 \quad\text{und}\quad y_1^{l} = \frac{y_2}{k_1\cdot k_0} = y_1^{l} + x_1^{l}\cdot d_1\cdot 2^{(-1)}$$

Daraus folgt die allgemeine Beziehung:

$$\begin{aligned} x_{i+1} &= \left(x_i^{l} - y_i^{l}\cdot d_i\cdot 2^{(-i)}\right)\prod_{k=o}^{i} k_k \\ y_{i+1} &= \left(y_i^{l} + x_i^{l}\cdot d_i\cdot 2^{(-i)}\right)\prod_{k=o}^{i} k_k \end{aligned} \quad . \tag{2.22}$$

Das Produkt der ersten N Werte für k_i

$$K_N = \prod_{i=o}^{N} k_i \tag{2.23}$$

wird auch als Skalierungsfaktor bezeichnet. Um Multiplikationen währende der Iteration zu vermeiden, wird der Skalierungsfaktor zunächst vernachlässigt. Es ergibt sich die vereinfachte Berechnungsvorschrift:

$$x_{(i+1)}^{l} = x_i^{l} - y_i^{l}\cdot d_i\cdot 2^{(-i)} \tag{2.24}$$

$$y_{(i+1)}^{l} = y_i^{l} + x_i^{l}\cdot d_i\cdot 2^{(-i)} \tag{2.25}$$

Nach N Iterationsschritten ergibt sich das folgende Zwischenergebnis als Näherungslösung

$$x_N^{l} \approx \frac{1}{K_{N-1}}\left[x_0\cdot\cos(\varphi) - y_0\cdot\sin(\varphi)\right] \tag{2.26}$$

$$y_N^{l} \approx \frac{1}{K_{N-1}}\left[y_0\cdot\cos(\varphi) + x_0\cdot\sin(\varphi)\right] \tag{2.27}$$

Bei diesem Ansatz müssen am Ende der Berechnung die Ergebnisse x^{l}_N und y^{l}_N nur noch einmalig mit K_{N-1} multipliziert werden, statt den jeweiligen Faktor k_i in jedem Schritt zu berechnen. Der Skalierungsfaktor K_N kann im voraus anhand der Anzahl der Iterationsschritte berechnet werden. Es gilt:[4]

$$K_N = \prod_{i=0}^{N}\cos\left(\arctan\left(2^{-i}\right)\right) = \prod_{i=0}^{N}\frac{1}{\sqrt{1+\tan^2\left(\arctan\left(2^{-i}\right)\right)}} = \prod_{i=0}^{N}\left(\frac{1}{\sqrt{1+2^{-(2i)}}}\right) \tag{2.28}$$

Dieser Wert kann als Konstante im System gespeichert werden. Für große Werte von N konvergiert der Skalierungsfaktor zum Wert[8, S. 134]:

$$K = \lim_{N \to \infty} K_n \approx 0{,}60725294 \tag{2.29}$$

Die numerischen Werte für den Skalierungsfaktor nähern sich für bereits für niedrige Anzahlen N rasch an diesem Grenzwert an. Tabelle 2.1 zeigt die ersten 10 Werte des Skalierungsfaktors und die relative Abweichung des Grenzwertes nach Gleichung (2.29) von diesen Werten.

Tabelle 2.1: Fehler durch die Approximation des Skalierungsfaktor mit dem Grenzwert

N	K_n	Abweichung K
1	0,632456	-0,0398%
2	0,613572	-1,0299%
3	0,608834	-0,2597%
4	0,607648	-0,0651%
5	0,607352	-0,0163%
6	0,607278	-0,0041%
7	0,607259	-0,0010%
8	0,607254	-0,0003%
9	0,607253	-0,0001%
10	0,607253	$-1{,}6 \cdot 10^{-5}$%

Es ist zu erkennen, dass bereits bei einer geringen Anzahl von Iterationen, die Abweichung vernachlässigbar ist. Somit kann in der Regel auf eine Berechnung des Skalierungsfaktors verzichtet werden und stattdessen in guter Näherung mit dem Grenzwert für $N \to \infty$ gerechnet werden.

Eine Gesamtrotation um den Winkel φ kann aus einer Folge von N Elementarrotationen mit der entsprechenden Richtung d_i zusammengesetzt werden:[4]

$$\varphi = \sum_{i=0}^{n} d_i \cdot \varphi_i \quad \text{mit} \quad \tan\left(\varphi_i\right) = 2^{-i} \quad \text{und} \quad d_i \in \left\{-1\ ,+1\right\} \tag{2.30}$$

Der Entscheidungsvektor $\vec{d} = \left(d_1, \ldots, d_N\right)^T$ mit den Vorzeichen der Einzelrotationen ist für jeden Gesamtrotationswinkel φ unterschiedlich. Für die vorherige Festlegung des Vektors, würde ein zusätzlicher Aufwand für Berechnung und Speicherung anfallen. Um dies zu vermeiden, wird er er während der Iteration bestimmt. Dazu wird die Hilfsvariable z_i eingeführt. Sie ist rekursiv durch den folgenden Ausdruck definiert [9, S. 122]:

$$z_{i+1} = z_i - d_i \cdot \varphi_i \tag{2.31}$$

und wird initialisiert mit dem Gesamtwinkel

$$z_0 = \varphi \quad . \tag{2.32}$$

Die Festlegung der Elementarentscheidungen für d_i , d. h. addieren oder subtrahieren für den nächsten Iterationsschritt wird anhand des Vorzeichens von z_i getroffen. Es gilt:

$$d_i = \begin{cases} +1 \text{ wenn } z_i < 0 \\ -1 \text{ wenn } z_i \geq 0 \end{cases} \tag{2.33}$$

Durch dieses Vorgehen wird implizit nach jedem Schritt die Gesamtdrehung $\sum d_i \cdot \varphi_i$ mit dem gewünschten Wert φ verglichen und der nächste Schritt in Richtung zur angestrebten Drehung festgelegt. Abbildung 2.4. veranschaulicht dies am Beispiel der ersten sieben Iterationen einer Rechnung.

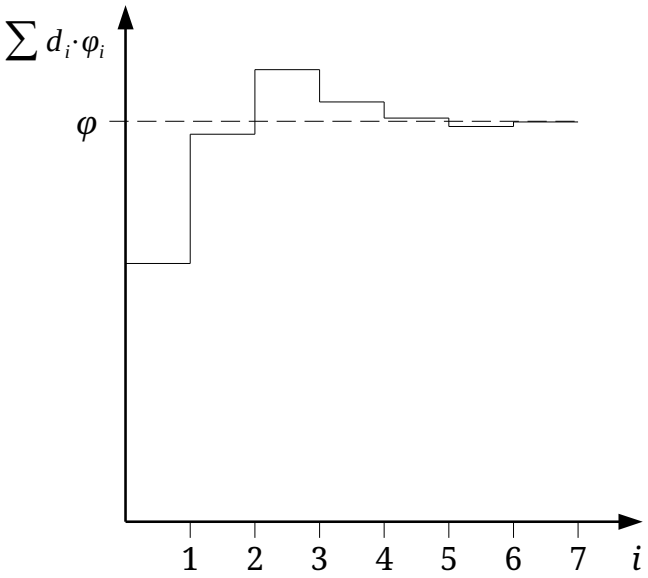

Abbildung 2.4.: Iteration der Teilwinkel zum Gesamtwinkel (in Anlehnung an [9, S. 122])

Ausgehend von diesen Überlegungen kann der CORDIC-Algorithmus wie folgt zusammengefasst werden. Für die Berechnung der Rotation eines Vektors ($x0$,$y0$) um den Winkel φ werden die folgenden Berechnungsvorschriften verwendet:

$$
\begin{aligned}
x'_{i+1} &= x'_i - y'_i \cdot d_i \cdot 2^{-i} \\
y'_{i+1} &= y'_i + x'_i \cdot d_i \cdot 2^{-i} \\
z_{i+1} &= z_i - d_i \cdot \arctan\left(2^{-i}\right) \\
&\text{mit} \\
d_i &= \begin{cases} +1 & \text{wenn } z_i < 0 \\ -1 & \text{wenn } z_i \geq 0 \end{cases}
\end{aligned}
\tag{2.34}
$$

und den Startwerten

$$
x'_0 = x_0 \ , \ y'_0 = y_0 \ , \ z_0 = \varphi \tag{2.35}
$$

2.3 Generalisierung

Der CORDIC-Algorithmus nach Gleichung (2.34) kann auch in Matrixform dargestellt werden:

$$
\begin{pmatrix} x'_{(i+1)} \\ y'_{(i+1)} \\ z_{(i+1)} \end{pmatrix} = \begin{pmatrix} 1 & -m \cdot d_i \cdot \varphi_i & 0 & 0 \\ d_i \cdot \varphi_i & 1 & 0 & 0 \\ 0 & 0 & 1 & -d_i \cdot \varphi_i \end{pmatrix} \cdot \begin{pmatrix} x'_i \\ y'_i \\ z_i \\ 1 \end{pmatrix} \tag{2.36}
$$

wobei hier gilt $m = 1$. Dieser Ansatz wurde von Walther [6] verallgemeinert, indem durch andere Werte von m lineare und hyperbolische Transformationen eingeschlossen werden. Es ergeben sich die folgenden Berechnungsvorschriften für die Iteration in den drei möglichen Fällen:

- **Zirkular (m = 1)**

$$
\begin{aligned}
x'_{i+1} &= x'_i - y'_i \cdot d_i \cdot 2^{-i} \\
y'_{i+1} &= y'_i + x'_i \cdot d_i \cdot 2^{-i} \\
z_{i+1} &= z_i - d_i \cdot \arctan\left(2^{-i}\right)
\end{aligned}
\tag{2.37}
$$

- **Hyperbolisch (m = -1)**

$$
\begin{aligned}
x'_{i+1} &= x'_i + y'_i \cdot d_i \cdot 2^{-i} \\
y'_{i+1} &= y'_i + x'_i \cdot d_i \cdot 2^{-i} \\
z_{i+1} &= z_i - d_i \cdot \operatorname{artanh}\left(2^{-i}\right)
\end{aligned}
\tag{2.38}
$$

- **Linear (m = 0)**

$$\begin{aligned} x_{i+1}^{l} &= x_0 \\ y_{i+1}^{l} &= y_i^{l} + x_0 \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot 2^{-i} \end{aligned} \tag{2.39}$$

In allen drei Fällen gilt für die Festlegung des Entscheidungsfaktors bei einer Rotation;

$$d_i = \begin{cases} -1 & \text{wenn} \quad z_i < 0 \\ +1 & \text{wenn} \quad z_i \geq 0 \end{cases} \tag{2.40}$$

Des Weiteren ist zu beachten, dass sich für die hyperbolische Variante in Gleichung (2.39) mit i = 0 der undefinierte Ausdruck artanh(1) ergibt. Daher beginnt für diesen Modus die Iteration bei i =1, während sie bei den beiden anderen Modi bei i = 0 startet. Unterschiede bestehen auch im effektiven Drehwinkel φ_i je Iterationsschritt. Daraus resultieren unterschiedliche Skalierungsfaktoren, welche in Tabelle 2.2 zusammengestellt sind.

Tabelle 2.2: Skalierungsfaktoren für den generalisierten CORDIC [4, S. 18]

Modus	Drehwinkel	Skalierungsfaktoren	
m	φ_i	K_n	$K = \lim_{n \to \infty} K_n$
1	$\arctan\left(2^{-i}\right)$	$\prod_{i=0}^{n} \cos\left(\arctan\left(2^{-i}\right)\right) = \prod_{i=0}^{n} \left(\frac{1}{\sqrt{1+2^{-2i}}}\right)$	$\approx 0{,}607253$
0	2^{-i}	$\prod_{i=0}^{n} 1 = 1$	$= 1$
-1	$\operatorname{artanh}\left(2^{-i}\right)$	$\prod_{i=1}^{n} \cosh\left(\operatorname{artanh}\left(2^{-i}\right)\right) = \prod_{i=1}^{n} \left(\frac{1}{\sqrt{1-2^{-2i}}}\right)$	$\approx 1{,}207497$

2.4 Anwendung des CORDIC-Algorithmus zu Berechnung von Funktionen

Der CORDIC-Algorithmus kann je nach Modus dazu verwendet werden, eine ganze Reihe an Funktionen zu berechnen. Dabei werden zwei Betriebsarten unterschieden: die Rotating-Berechnung und die Vectoring Berechnung. In der Rotating-Berechnung wird der Ergebnisvektor für eine vorgegebene Drehung des Startvektors bestimmt, während bei der Vectoring-Berechnung die Drehung bestimmt wird, um vom Startvektor auf einen vorgegebenen Zielvektor zu kommen. Auf diese Weise können im Zir-

kularmodus eineseits trigonometrische Funktionen und andererseits die inversen trigonometrischen Funktionen für vorgegeben Argumente ausgewertet werden. Dies wird in den beiden folgenden Abschnitten erläutert. Danach wird die Vorgehensweise bei der Berechnung von Funktionen, mit den beiden anderen Modi beschrieben.

2.4.1 Rotating-Berechnung im Zirkularmodus

Für diese Berechnung wird ein geeigneter Startvektor gewählt und um den Winkel φ gedreht. Aus dem Näherungsergebnis können dann nach einer hinreichend großen Anzahl von Iterationsschritten die Funktionswerte für φ abgeleitet werden.

Eine einfache Berechnung ergibt sich für den Einheitsvektor (x_0 , y_0) = (1 , 0) als Startvektor (siehe Abbildung 2.5.). Der Startwert für z entspricht dem Gesamtrotationswinkel, d. h. $z_0 = \varphi$.

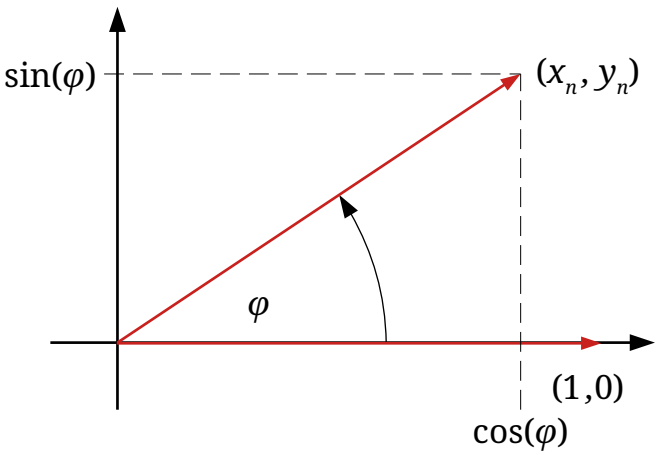

Abbildung 2.5.: Konzept der Rotating-Berechnung

Aus Gleichung (2.26) und (2.27) folgt für den Zirkular-Modus mit diesem Startvektor:

$$x_n' \approx \frac{1}{K_{n-1}} \cdot \cos(\varphi) \tag{2.41}$$

$$y_n' \approx \frac{1}{K_{n-1}} \cdot \sin(\varphi) \tag{2.42}$$

Somit können aus den Koordinaten nach n Iterationen direkt alle elementaren trigonometrischen Funktion des Drehwinkel φ ermittelt werden:

$$\sin(\varphi) \approx K_{n-1} \cdot y_n' \tag{2.43}$$

$$\cos(\varphi) \approx K_{n-1} \cdot x_n' \tag{2.44}$$

$$\tan(\varphi) \approx \frac{y_n'}{x_n'} \tag{2.45}$$

$$\cot(\varphi) \approx \frac{x_n'}{y_n'} \tag{2.46}$$

2.4.2 Vectoring-Berechnung im Zirkularmodus

Die Vectoring-Berechnung ist quasi eine Umkehrung der Rotating-Berechnung. Bei der Rotating wird ein Startvektor um eine definierten Winkel gedreht und anhand der Koordinaten des iterativ bestimmten Ergebnisvektor die trigonometrischen Funktion für den Winkel bestimmt. Im Gegensatz dazu wird bei der Vectoring-Methode Start- und Zielvektor vorgegeben und der dafür erforderliche Drehwinkel berechnet. Aus diesem ergeben sich dabei je nach Definition von Start-- und Zielvektor die unterschiedlichen inversen trigonometrischen Funktionen. Genauer gesagt wird unterschieden zwischen der Berechnung von Arkustangens bzw. Arkuskotangens einerseits und Arkussinus und Arkuskosinus andererseits.

Berechnung von Arkustangens und Arkuskotangens

Für diese Berechnung wird der Winkel eines vorzugebenden Startvektors bezüglich der x-Achse betrachtet. Durch den CORDIC-Algorithmus wird dieser gedreht bis er auf dieser Achse liegt (siehe Abbildung 2.6.).

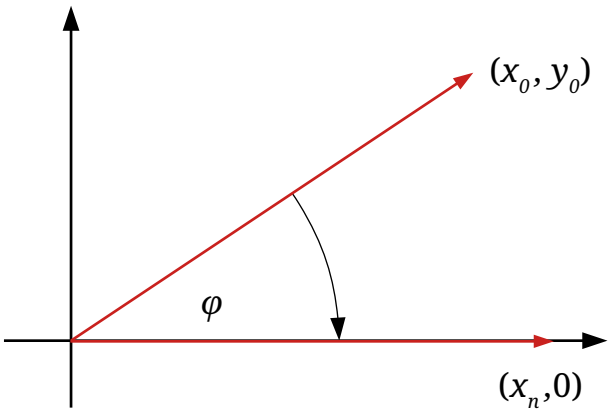

Abbildung 2.6.: Konzept der Vecoring-Berechnung für Arkustangens und Arkuskotangens

Diese vorgehensweise wird realisiert, indem die Entscheidungsbedingung in Gleichung (2.40) modifiziert wird. Für die Vectoring-Berechnung wird die Abweichung der y-Koordinate vom Zielwert 0 betrachtet, um über die Richtung der nächsten Drehung zu entscheiden, d.h. das Vorzeichen des Entscheidungsfaktors festzulegen:

$$d_i = \begin{cases} +1 & \text{wenn} \quad y_i' < 0 \\ 0 & \text{wenn} \quad y_i' = 0 \\ -1 & \text{wenn} \quad y_i' > 0 \end{cases} \tag{2.47}$$

Zusätzlich wird für den Startwert von z $z_0 = 0$ festgelegt. Die übrige Iterationsvorschrift nach Gleichung (2.37), (2.38) bzw. (2.39) bleibt unverändert. So wie die ursprüngliche Entscheidungsvorschrift in Gleichung (2.40) bewirkt, dass z_i sich dem Wert Null annähert, führt die Modifikation dazu, dass sich y_i sukzessive an Null annähert.

Im Zirkularmodus bewirkt der Algorithmus also, dass der Ausgangsvektor sukzessive mit der abnehmenden Schrittweite arctan(2^{-i}) gedreht wird, bis er auf der positiven x-Achse liegt. Die x-Koordinate am Ende der Iteration entspricht somit der Länge des Ausgangsvektors:

$$x_n' \approx \frac{1}{K_{n-1}} \sqrt{x_0^2 + y_0^2} \tag{2.48}$$

Der Winkel der Gesamtdrehung, die im Zuge der Iteration ausgeführt wurde, entspricht dabei dem Winkel zwischen dem Ausgangsvektor und der x-Achse. Somit ist der Tangens dieses Winkels gleich dem Verhältnis der y-Koordinate zur x-Koordinate des Startvektors.

Wie bereits erläutert wurde, summieren sich die Teildrehungen zum Gesamtwinkel. Durch Auswerten der Rekursion für z_i folgt somit unter Berücksichtigung des Startwertes $z_0 = 0$ und der Vorzeichenwahl gemäß Gleichung (2.47):

$$z_n \approx z_0 - \sum_{i=0}^{n-1} d_i \cdot \varphi_i \approx z_0 + \varphi \approx \varphi = \arctan\left(\frac{y_0}{x_0}\right) \tag{2.49}$$

Da der Kotangens das Verhältnis der x-Koordinate des Startvektors zu dessen y-Koordinate ist, gilt entsprechend für den Arkuskotangens:

$$z_n \approx \varphi = \operatorname{arccot}\left(\frac{x_0}{y_0}\right) \tag{2.50}$$

Arkussinus und Arkuskosinus

Für die Berechnung des Arkussinus und Arkuskosinus wird in umgekehrter Richtung vorgegangen, d. h. es wird ein Startvektor, der auf der x-Achse liegt, gedreht, bis eine vorgegebene Position erreicht wird. Ausgangspunkt für diese Überlegung ist, dass am

Ende der Iteration – nach der Drehung um den Winkel φ – die Koordinaten gegeben sind durch:

$$\begin{aligned} x_n &= |r| \cdot \cos(\varphi) \\ y_n &= |r| \cdot \sin(\varphi) \end{aligned} \tag{2.51}$$

Wird ein Einheitsvektor mit $|r| = 1$ gedreht, entspricht die y-Koordinate am Ende der Iteration dem Sinus des Drehwinkels. Sie entspricht somit dem Argument $\xi \in [-1\,;1]$, für das der Arkussinus bestimmt werden soll. Daher wird für die Berechnung ein Einheitsvektor von der x-Achse solange gedreht, bis seine y-Koordinate diesem Argument entspricht (siehe Abbildung 2.7.).

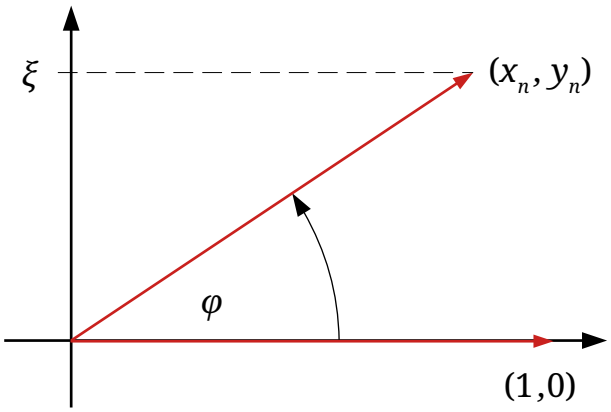

Abbildung 2.7.: Konzept der Vectoring-Berechnung für Arkussinus und Arkuskosinus

Diese Vorgehensweise wird realisiert, indem die Drehung entsprechend der Abweichung von y_i zu diesem Argument fortgesetzt wird. Somit gilt die Iterationsvorschrift:

$$\begin{aligned} x'_{i+1} &= x'_i - y'_i \cdot d_i \cdot 2^{-i} \\ y'_{i+1} &= y'_i + x'_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i + d_i \cdot \arctan\left(2^{-i}\right) \\ \text{mit} \quad d_i &= \begin{cases} -1 & \text{wenn } (\xi - y_i) < 0 \\ 0 & \text{wenn } (\xi - y_i) = 0 \\ +1 & \text{wenn } (\xi - y_i) > 0 \end{cases} \\ x_{ß} &= 1 \;,\; y_0 = z_0 = 0 \end{aligned} \tag{2.52}$$

Am Ende der Iteration ergibt sich:

$$\begin{aligned} y'_n &\approx \frac{\xi}{K_{n-1}} \\ z_n &\approx \varphi = \arcsin(\xi) \end{aligned} \tag{2.53}$$

Mithilfe dieser Iteration für den Arkussinus kann auch der Arkuskosinus berechnet werden. Für diese beiden inversen trigonometrischen Funktion gilt die Identität:

$$\arccos(\xi) = \frac{\pi}{2} - \arcsin(\xi) \tag{2.54}$$

Daher gilt auch:

$$\arccos(\xi) \approx \frac{\pi}{2} - z_n \tag{2.55}$$

Die Berechnung lässt sich vereinfachen, indem für x_0 der Startwert K_{n-1} verwendet wird. Die nachfolgende Abbildung 2.8. veranschaulicht die Berechnung von arcsin(0,9) über 10 Iterationsschritte.

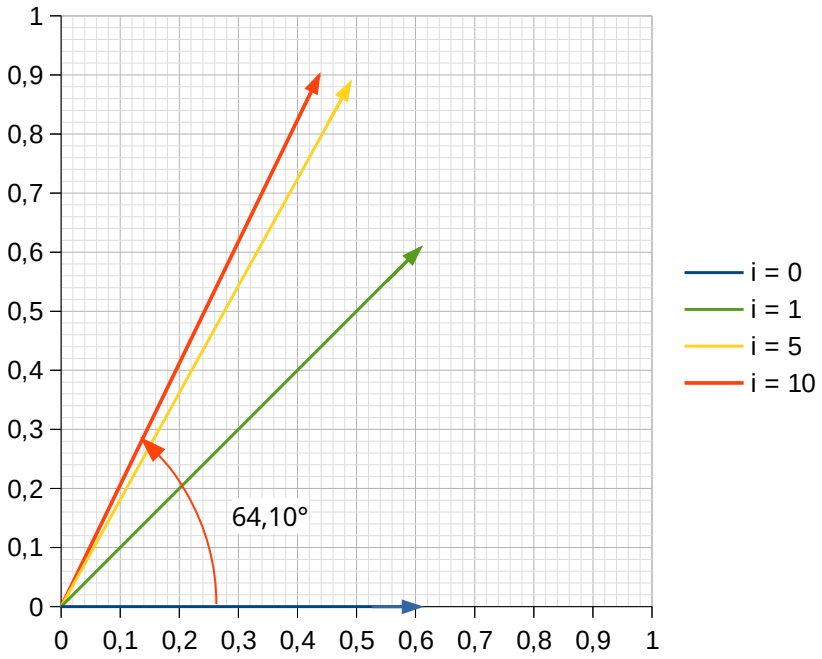

Abbildung 2.8.: Berechnung von arcsin(0,9)

Die einzelnen Werte der Berechnung können Tabelle A.1 im Anhang entnommen werden. Der exakte Wert beträgt arcsin(0,9)= 64,158072°.

2.4.3 Funktionsberechnungen im hyperbolischen Modus

Die in Kapitel 2.4.1 und 2.4.2 vorgestellten Vorgehensweisen gelten analog für die anderen Modi. Im hyperbolischen Modus erlaubt die Rotating-Berechnung ausgehend vom Startvektor (x_0 , y_0) = (1 , 0) die Bestimmung der hyperbolischen Funktionen. Analog zu den Gleichungen (2.43) bis (2.46) gilt:

$$\sinh(\varphi) \approx K_{n-1} \cdot y_n' \tag{2.56}$$

$$\cosh(\varphi) \approx K_{n-1} \cdot x_n' \tag{2.57}$$

$$\tanh(\varphi) \approx \frac{y_n'}{x_n'} \tag{2.58}$$

$$\coth(\varphi) \approx \frac{x_n'}{y_n'} \tag{2.59}$$

Aus der Definition der hyperbolischen Funktionen folgt:

$$\sinh(\varphi) + \cosh(\varphi) = \frac{1}{2}\left(e^{\varphi} - e^{-\varphi}\right) + \frac{1}{2}\left(e^{\varphi} + e^{-\varphi}\right) = e^{\varphi} \tag{2.60}$$

Somit kann zusätzlich die Exponentialfunktion berechnet werden.

$$e^{\varphi} \approx x_n + y_n \tag{2.61}$$

Wird durch die Vectoring-Berechnung im hyperbolischen Modus ein Startvektor (x_0, y_0) durch den CORDIC-Algorithmus in die Position (x_n, 0) gedreht kann der Areatangens hyperbolikus berechnet werden. Wegen des eingeschränkten Bildbereichs des Tangens hyperbolicus ($\tanh(\varphi)| < 1$), muss der Startvektor die Bedingung $|x_0| > |y_0|$ erfüllen. Unter Beachtung dieser Bedingung führt der Algorithmus zu:

$$x_n' \approx \frac{1}{K_{n-1}} \sqrt{x_0^2 - y_0^2} \tag{2.62}$$

Analog zu den Gleichungen (2.49) und (2.50) gilt am Ende der Berechnung:

$$z_n \approx \varphi = \operatorname{artanh}\left(\frac{y_0}{x_0}\right) \quad \text{bzw.} \quad z_n \approx \varphi = \operatorname{arcoth}\left(\frac{x_0}{y_0}\right) \tag{2.63}$$

Diese Form der Vectoring-Berechnung lässt sich auch für die Berechnung des natürlichen Logarithmus verwenden. Dazu wird der Tangens hyperbolicus mit logarithmischem Argument betrachtet:

$$\tanh\left(\frac{1}{2}\ln(\xi)\right) = \tanh\left(\ln\left(\sqrt{\xi}\right)\right) = \frac{e^{\ln(\sqrt{\xi})} - e^{-\ln(\sqrt{\xi})}}{e^{\ln(\sqrt{\xi})} + e^{-\ln(\sqrt{\xi})}} = \frac{\sqrt{\xi} - 1/\sqrt{\xi}}{\sqrt{\xi} + 1/\sqrt{\xi}} = \frac{\xi - 1}{\xi + 1} \tag{2.64}$$

Daraus folgt:

$$\ln(\xi) = 2 \cdot \operatorname{artanh}\left(\frac{\xi - 1}{\xi + 1}\right) \tag{2.65}$$

Nach Abschluss der Rechnung mit dem Startvektor von $x_0 = \xi+1$ und $y_0 = \xi-1$ gilt somit

$$2\,z_n \approx \ln(\xi) = 2 \cdot \operatorname{artanh}\left(\frac{y_0}{x_0}\right) \quad . \tag{2.66}$$

Durch eine geeignete Wahl des Startvektors kann außerdem mithilfe der Beziehung (2.62) eine Wurzel berechnet werden. Mit $x_0 = \xi + ¼$ und $y_0 = \xi - ¼$ ergibt sich:

$$\sqrt{x_0^2 - y_0^2} = \sqrt{\left(\xi + \frac{1}{4}\right)^2 - \left(\xi - \frac{1}{4}\right)^2} = \sqrt{\left(\xi^2 + \frac{\xi}{2} + \frac{1}{16}\right) - \left(\xi^2 - \frac{\xi}{2} + \frac{1}{16}\right)} = \sqrt{\xi} \quad . \tag{2.67}$$

Unter dieser Wahl des Startvektors gilt für die x-Koordinate am Ende der Iteration

$$x_n' \approx \frac{1}{K_{n-1}} \sqrt{x_0^2 - y_0^2} = \frac{\sqrt{\xi}}{K_{n-1}} \tag{2.68}$$

2.4.4 Funktionsberechnungen im linearen Modus

Die Rotating-Berechnung im linearen Modus ergibt ausgehend vom Startvektor (x_n, 0) am Ende der Iteration [4, S. 13]:

$$x_n' = x_0 \tag{2.69}$$

$$y_n' \approx y_0 + x_0 \cdot z_0 = x_0 \cdot z_0 \tag{2.70}$$

$$z_n = z_0 - \sum_{i=0}^{n} d_i \cdot 2^{-i} \approx 0 \tag{2.71}$$

Dabei ist z_0 nicht frei wählbar, da die Summe der Teilverschiebungen beschränkt ist. Es gilt:

$$\lim_{n \to \infty} \sum_{i=0}^{n} 2^{-i} = 2 \tag{2.72}$$

Selbst für den Fall, dass alle Teildrehungen dieselbe Richtung, d. h. alle d_i denselben Wert haben, kann die Gesamtverschiebung nicht größer sein als 2. Die Annäherung von z_n an Null nach Gleichung (2.71) ist also nur dann möglich, wenn die Bedingung

$$|z_0| < \sum_{i=0}^{\infty} 2^{-i} = 2 \tag{2.73}$$

erfüllt ist[4, S. 13]. Das heißt,dass sich durch den CORDIC-Algorithmus ein Produkt berechnen lässt, unter der Bedingung dass einer der beiden Faktoren betragsmäßig kleiner als 2 ist.

Die Vectoring-Berechnung mit einer Drehung des Startvektors (x_0 , y_0) in die Richtung (x_n, 0) ergibt im linearen Modus nach *n* Schritten:

$$x_n^{'} = x_0 \tag{2.74}$$

$$z_n = \sum_{i=0}^{n} d_i \cdot 2^{-i} \approx \frac{y_0}{x_0} \tag{2.75}$$

In dieser Variante erlaubt der CORDIC die Berechnung eines Quotienten. Die Beschränkung gemäß Gleichung (2.72) führt zu der Bedingung:

$$\left|\frac{y_0}{x_0}\right| < 2 \quad . \tag{2.76}$$

Die allgemeine Iterationsvorschrift für die Vectoring-Berechnung im Linear-Modus lautet:

$$\begin{aligned} x_{i+1}^{'} &= x_i^{'} = x_0 \\ y_{i+1}^{'} &= y_i^{'} + x_i^{'} \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot 2^{-i} \\ \text{mit} \quad d_i &= \begin{cases} +1 & \text{wenn} \quad y_i < 0 \\ 0 & \text{wenn} \quad y_i = 0 \\ -1 & \text{wenn} \quad y_i > 0 \end{cases} \\ x_0 &> 0 \ , \ z_0 = 0 \end{aligned} \tag{2.77}$$

2.5 Vor- und Nachteile des CORDIC-ALgorithmus

In der praktischen Nutzung bietet die Anwendung des CORDIC-Algorithmus spezifische Vorteile und Nachteile im Vergleich zu anderen Berechnungskonzepten. Als wichtigste Vorteile können genannt werden: [10, S. 11]

- Der CORDIC-Algorithmus verwendet – neben Additions- und Subtraktionsoperationen – Schiebeoperationen, welche die benötigten Multiplikationen abbilden. Dies ermöglicht ein einfaches VLSI-Design.
- Die Anforderungen an die Hardware und damit verbunden Kosten sind geringer als bei anderen Konzepten, weil für die Umsetzung des Algorithmus in einem CORDIC-Prozessor lediglich Schiebe-Register, Addierer und Look-up-tables (ROM) benötigt werden.
- Im Vergleich zu komplexeren Prozessoren, wie bspw. DSP-Mulitpliern, wird eine deutlich geringere Anzahl an Gates benötigt.
- Die Verarbeitungszeit ist relativ gering und in etwa vergleichbar mit einer herkömmlichen Divisions – oder Wurzelberechnung.

Wesentliche Nachteile in der Praxis sind:

- Die Genauigkeit des Algorithmus ist begrenzt. Hohe Anforderungen an die Genauigkeit erfordern daher eine große Anzahl an Iteration und dementsprechend hohe Berechnungszeiten.
- In einigen Designs kann es zu einem hohen Stromverbrauch kommen.
- Im Allgemeinen lassen sich mit Hardwaremultiplizieren und darauf basierenden Berechnungskonzepten, bspw. der Auswertung von Potenzreihen, kürzere Berechnungszeiten erreichen.

Ein CORDIC-Prozessor bietet sich daher vor allem dann an, wenn Multiplizierer vermieden oder die Anzahl der Gates minimiert werden sollen und gleichzeitg moderate Anforderungen an die Genauigkeit gestellt werden.

3 Entwurf in VHDL

In diesem Kapitel wird der Entwurf zweier VHDL Module zur Berechnung von Quotienten und der Arkussinusfunktion erläutert. Ausgehend von einer Formulierung der allgemeinen Anforderungen wird VHDL Queltext für die beiden Berechnungsmodule erläutert. Zusätzlich wird der Entwurf jeweils durch Simulationen verifiziert. Der Entwurf erfolgt mithilfe der Software Vivado von der Firma Xilinx.

3.1 Anforderungen

Wie bereits erwähnt soll mithilfe des FPGA unter Verwendung des CORDIC-Algorithmus zum einen die Berechnung von Divisionen und zum anderen die Berechnung des Arkussinus realisiert werden. Dabei sind einige Anforderungen zu beachten:

- In- und Output Signale sollen ausschließlich von Typ Std_Logic oder Std_logic_Vector sein.
  Diese nach dem Standard IEEE 1164 genormten Formate repräsentieren eine einzelne logische Variable bzw. ein Array logischer Variablen. Im Unterschied zum Bit-Format, das nur die Werte „0“ und „1“ kennt, kann eine Std_Logic bzw. eine Komponente eines Std-Logic-Vectors, weitere sieben Werte, wie bspw. „nicht initialisiert“ (uninitialized) oder „unbekannt“ (unknown) annehmen.[11, S. 4 f.] Ausschlaggebend für die Verwendung dieses Formates ist, dass es auch in der Praxis die übliche Form für die Ein- und Ausgabe-Ports an FPGA-Bausteinen darstellt.
- Keine Verwendung von Multiplizierern:
  Die Durchführung von Multiplikationen bedingt im Vergleich zu Additionen bzw. Subraktionen einen hohen Rechenaufwand. Hier wird der CORDIC-Algorithmus unter anderem deshalb verwendet, weil dieser Aufwand vermieden wird.
- Keine Verwendung von Gleitkommazahlen:
  Die Nutzung von Gleitkommazahlen erfordert komplexe Fließkommazahlenlogik auf dem FPGA. Um diese zu vermeiden, wird eine Festkommazahlendarstellung verwendet. Hierfür werden die Zahlen vor der Verarbeitung mit einer Potenz von 2 multipliziert, was auf binärer Ebene einer einfachen Stellenverschiebung entspricht, um Bitstellen für die Kodierung von Werten < 1 zu reservieren. Dies kann auf einfache Weise durch ein Schieberegister realisiert werden. Der zu verwendende Faktor ergibt sich dabei aus der Anzahl der Iterationsschritte im CORDIC-Algorithmus. Im letzten, n-ten Schritt wird 2^{-n} addiert bzw. subtrahiert.

- Flexibilität des Programms:
  Hierunter ist zu verstehen, dass die Anzahl der Iterationen leicht geändert werden kann. Die Berechnungsgenauigkeit des Algorithmus und der Zeitaufwand für dessen Durchführung wird über die Anzahl der Iterationen skaliert. Da zudem die Genauigkeit auch von der Art der Berechnungsaufgabe abhängt, sollte die Iterationsanzahl den jeweiligen Anforderungen entsprechend angepasst werden können.
- Nutzung eines Pipeline-Ansatzes:
  Die Berechnung ist als Pipeline-Prozess umzusetzen. Dabei handelt es sich um ein Konzept, das bei Prozessoren eingesetzt wird (siehe Abbildung 3.1.). Anstatt die Rechnung für verschiedene Werte nacheinander komplett auszuführen, werden die einzelnen Schritte der Operation für mehrere Werte parallel ausgeführt. Das bedeutet, dass sobald der erste Schritt des ersten Wertes ausgeführt worden ist, mit der ersten Iteration für die Berechnung des nächsten Wertes begonnen werden kann.

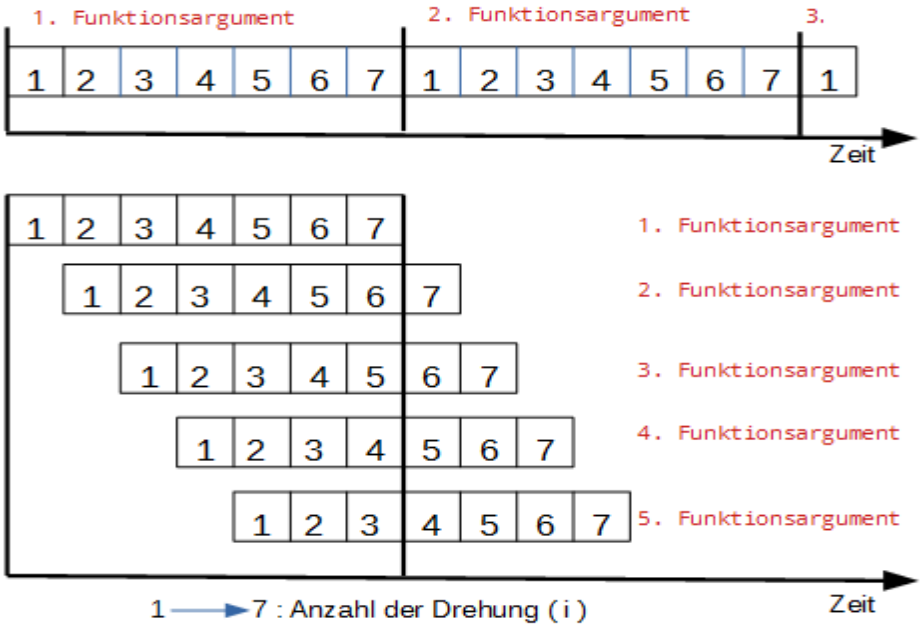

Abbildung 3.1.: Pipelining Beispiel

In diesem Fall ist eine Gesamtoperation die Berechnung eines Funktionswertes. Deren einzelne Schritte sind die Iterationsschritte nach dem CORDIC-Algorithmus. Die Latenz, d. h. der Zeitaufwand für die Berechnung eines Funktionswertes, ändert sich dabei nicht, weil sie durch die Taktzeit und die Anzahl der Iterationsschritte vorgegeben ist. Allerdings lässt sich durch das parallele Ausführen die Anzahl der Funktionsauswertungen pro Zeiteinheit erhöhen. Da ebenso viele Funktionsauswertungen parallel durchgeführt werden wie Iterationsschritte, beschränkt nur die Taktfrequenz des FPGA, die maximale Rate von Input- und Output-Werten.

3.2 Division-Berechnung in VHDL

Das Blockschaltbild für die Realisierung von Divisionsoperation mithilfe eines FPGAs ist in Abbildung 3.2. dargestellt. Die Grundlage für die Berechnung bildet die Vectoring-Berechnung des CORDIC-Algorithmus im Linearmodus, welche in Kapitel 2.4.4 vorgestellt wurde.

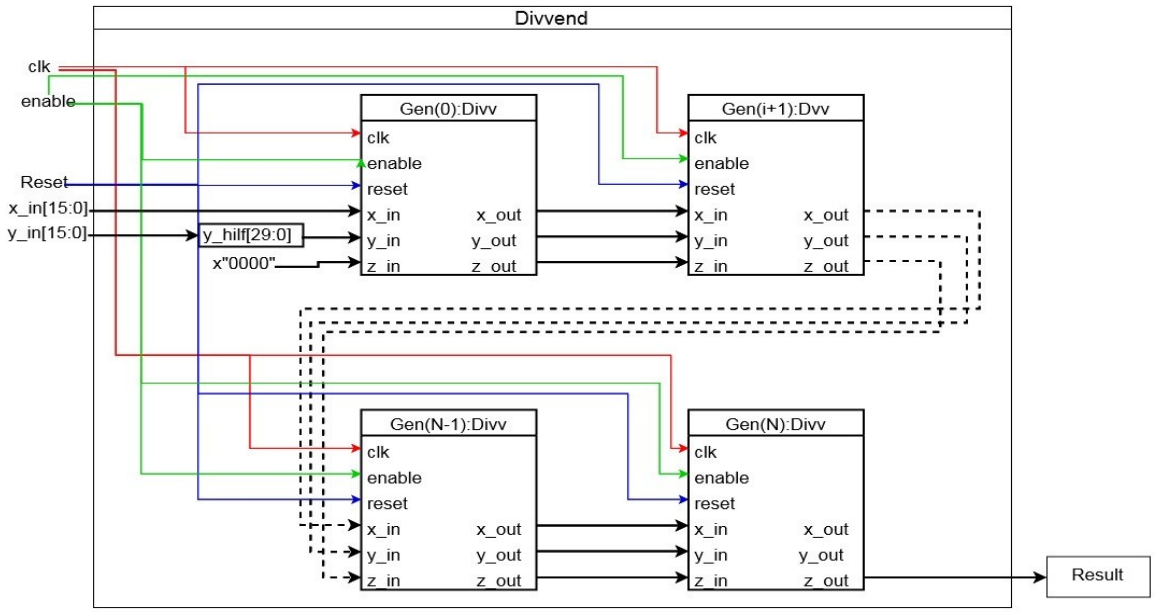

Abbildung 3.2.: Konzept für die Realisierung der Division

Die Berechnung der einzelnen Iterationsschritte erfolgt jeweils durch ein separates Modul. Jedes dieser Module erhält die Berechnungsergebnisse (x_{i-1}, y_{i-1}, z_{i-1}) des vorherigen Schrittes bzw. Moduls als Eingang und hat als Ausgang die Resultate (x_i, y_i, z_i) des betreffenden Iterationsschrittes.

In VHDL wird dieses Baustein-Konzept durch eine hierarchischen Struktur dargestellt. Die oberste Ebene ist das Modul „Divvend“, welches den Gesamtprozess der Iteration steuert. Darunter liegen die Module für die einzelnen Iterationsschritte. Aufgrund deren Gleichartigkeit braucht nur einsiges Modul (Bezeichnung „Divv“) entworfen werden, welches wiederholt instanziiert wird. Dies reduziert nicht nur den Implementierungsaufwand, sondern gewährleistet auch die einfache Anpassung der Anzahl der Iterationen. Im vorliegenden Fall wird die Kalkulation des Quotienten mit bis zu 15 Iterationen durchgeführt. Die Module für diese Berechnung werden in den beiden folgenden Abschnitten erläutert.

3.2.1 Submodul „Divv“

Die Implementierung des Moduls beginnt mit dessen Deklaration und der Beschreibung der verwendeten Signalen sowie der Definition der Ein- und Ausgänge (siehe Quellcode 3.1). Das Modul erhält vom Hauptmodul den generischen Integer-Parameter k1, welche für die Durchführung der Berechnungen benötigt wird, was im Zusammenhang mit der Beschreibung von Quellcode 3.4 erläutert wird.

```
33 entity divv is
34 generic(k1 : integer) ;
35 Port (  clk   : in  std_logic ;
36       reset  : in  std_logic ;
37       enable : in  std_logic ;
38        x_in  : in  std_logic_vector(15 downto 0) ;
39        y_in  : in  std_logic_vector(29 downto 0) ;
40        z_in  : in  std_logic_vector(15 downto 0) ;
41        x_out : out std_logic_vector(15 downto 0) ;
42        y_out : out std_logic_vector(29 downto 0) ;
43        z_out : out std_logic_vector(15 downto 0) );
```

Quellcode 3.1: Deklaration von Variablen und Ports für das Modul divv

Durch die Anweisung „port“ in den Zeilen 35 bis 43 werden die Ein- und Ausgänge des Moduls definiert, wobei der Port-Name, die Port Art (in oder out) sowie der Typ des Signals des Ports deklariert werden.

Die drei ersten Eingänge, (Zeile 33 bis 35 in Quellcode 3.1) sind Signale vom Typ std_logic. Diese dienen der Steuerung des Moduls und haben im einzelnen die folgenden Funktion:

- „clk“:: An diesem Eingang wird der Systemtakt übergeben
- „reset“: Über diesen Eingang erhält das Modul ein Signal zum Zurücksetzen der Berechnung (reset = 1). Dieses Signal wird vom Modul „Divvend“ erzeugt und an alle Module des Typs „Divv“ weitergegeben, sodass alle Module gleichzeitig zurückgesetzt werden.
- „enable“: Dieser Eingang dient der Aktivierung des Moduls. Auch dieses Signal wird zentral vom Modul „Divvend“ erzeugt, d. h. alle Module des Typs „Divv“ sind gleichzeitig aktiviert (enable = 1) oder deaktiviert (enable = 0).

Die anderen drei Eingänge und die drei Ausgänge (Zeile 36 bis 43 in Quellcode 3.1) dienen der Übertragung numerischer Werte und werden mit Signalen des Typs std_logic_vector in der Bit-Länge des jeweiligen numerischen Wertes belegt. Diese Länge ist für Ein- und Ausgang der jeweiligen Größe identisch. Die deklarierten Längen von 16 bzw. 30 Bit begründen sich wie folgt:

- „x_in“, „x_out“: Für die Größe *x* sind Ein- und Ausgangsignal identisch, weil diese Größe durch den Algorithmus nicht verändert wird. Daher entspricht die Signallänge der Länge des Anfangswertes, der auf 16 Bit festgelegt wurde. Obwohl die Größe sich nicht ändert, wird sie für jeden Iterationsschritt benötigt, weshalb sie jedes „Divv“-Modul als Eingang erhält.

- „y_in“, „y_out“: Die Größe y wird im Zuge der Iteration gegen Null reduziert. Da keine Gleitkommazahlen verwendet werden sollen, wird der 16-Bit-Startwert durch das Modul „Divvend“ mit 2^{14} multipliziert. Somit ist *y* auf der Berechnungsebene eine Fixkommazahl mit 14 Nachkommastellen.

- „z_in“, „z_out“: Die Größe z repräsentiert in dieser Berechnungsform die Summe der Verschiebungen (siehe Gleichung (2.73)). Die einzelnen Verschiebungen haben die Größe 2^{-i}, und besitzen im gewählten Fixkommaformat eine Länge von bis zu 14 Bit. Der Betrag der Summe dieser Verschiebungen ist stets kleiner als 2 (siehe Gleichung (2.72)), also maximal 15 Bit. Unter Berücksichtigung eines zusätzlichen Bits für das Vorzeichen ergibt sich somit eine Länge von 16 Bit für dieses Signal.

Theoretisch ist es möglich, die weiteren Berechnungen direkt mit den Eingangssignalen vom Typ std_logic_vector und den passenden Bibliotheken durchzuführen. Günstiger ist allerdings die Kalkulation mithilfe von Signalen der Integer-Typen aus der Bibliothek numeric_std. Dies erfordert die Umwandlung der entsprechenden Eingangssignale in interne Signale (siehe Quellcode 3.2).

```
55   x_in1 <= to_integer(signed(x_in)*2**14);
56   y_in1 <= to_integer(signed(y_in));
57   z_in1 <= to_integer(signed(z_in));
```

Quellcode 3.2: Umwandlung in interne Signale

Entsprechend den Konventionen dieser Bibliothek werden die Eingangsignale in Signale des Formats signed integer umgewandelt. Zusätzlich wird das Eingangssignal „x_in“ mit 2^{14} multipliziert, um es in dasselbe Fixkommazahlenformat zu konvertieren wie die beiden anderen Größen (Zeile 55 in Quellcode Quellcode 3.2) . Anstelle dieser Berechnung könnte auch das Ein- und Ausgangssignal mit 30 Bit Länge verwendet werden. Da aber *x* eine Ganzzahl ist, wären die 14 Bit mit den Nachkommastellen immer mit 0 belegt und somit ohne Informationsgehalt. Dadurch würde ein erheblicher Teil der Bitbreite bei der Übertragung des Signals verschwendet. Günstiger ist die dargestellte Operation im Modul, da sie nur eine einfache Stellenverschiebung beinhaltet.

Die Variablentypen werden beim Anlegen des Moduls nach Typ und Länge definiert (siehe Quellcode 3.3). Ein wichtiger Aspekt ist in diesem Kontext, dass die Bitlänge vor-

gegeben wird, da anderenfalls alle Variablen in einer Standardlänge und damit unter Umständen länger als erforderlich implementiert werden. Beim hier hier verwendeten Typ Integer ist das die Standardwortlänge 32 Bit. Den Zeilen 48 und 49 im Quellcode 3.3 ist zu entnehmen, dass zwei „Subtypes“ angelegt werden, welche die beiden auftretenden Längen von 16 Bit bzw. 30 Bit repräsentieren.

```
48  SUBTYPE S8 IS INTEGER RANGE   -2**15  TO (2**15)-1 ;
49  SUBTYPE S9 IS INTEGER RANGE   -2**29  TO (2**29)-1 ;
50
51  signal y1 ,x_in1 ,y_in1  : S9 :=0 ;
52  signal z1 ,z_in1 : s8 :=0  ;
```

Quellcode 3.3: Beschränkung der Länge der Variablen im Modul „Divv“

Durch den Befehl „signal“ in Zeile 51 und 52 wird den einzelnen Signalen ein Subtype zugewiesen und diese mit einem Startwert, hier 0, initialisiert. Dabei werden zusätzlich zu den Eingangssignalen die modulinternen Signale „y1“ und „z1“ definiert, welche die Berechnungsergebnisse erfassen, aus denen die Ausgangssignale erzeugt werden.

Für die Verarbeitung der Signale, d. h. die Durchführung der einzelnen Iterationsschritte des CORDIC-Algorithmus wird ein Prozess implementiert. Der Entwurf hierzu ist im Quellcode 3.4 dargestellt. Ein Prozess wird nur ausgeführt, wenn er aktiv ist. Die Aktivität des Prozesses wird, wie bereits erwähnt, bestimmt durch die drei Signale „clk“ und „reset“. Diese werden dem Prozess als sog. Sensitivity-Liste übergeben (siehe Zeile 58). Ändert sich einer dieser Eingänge wird der Prozess getriggert und die Werten der Signale entsprechend abgearbeitet.

```
58 process (clk , reset)
59   begin
60    if  (reset = '1') then
61           y1 <= 0;
62           z1 <= 0;
63        x_out <= x"0001" ;
64    elsif  (rising_edge(clk))   then
65      if (enable='1') then
66         if (y_in1> 0) then
67             if (x_in1 > 0)  then
68                y1 <= y_in1 - x_in1/k1 ;
69                z1 <= z_in1 + 2**14/k1 ;
70             elsif(x_in1 < 0) then
71                y1 <= y_in1 + x_in1/k1 ;
72                z1 <= z_in1 - 2**14/k1 ;
73             end if ;
74         else
75             if (x_in1 < 0)  then
76                y1 <= y_in1 - x_in1/k1 ;
77                z1 <= z_in1 + 2**14/k1 ;
78             elsif(x_in1 > 0) then
79                y1 <= y_in1 + x_in1/k1 ;
80                z1 <= z_in1 - 2**14/k1 ;
81             end if ;
82         end if;
83      x_out <= x_in ;
84      end if ;
85   end if ;
86 end process ;
```

Quellcode 3.4: Prozess für Durchführung eines Iterationsschrittes im Modul „DIVV“

Zunächst wird überprüft, ob die Berechnung zurückgesetzt werden soll, d. h. das Signal „reset“ den logischen Wert 1 hat, und die entsprechenden Zuweisungen vorgenommen (Zeile 60 bis 63). In diesem Fall werden die internen Signale y1 und z1 auf 0 gesetzt. Das Ausgangssignal für x sollte nie den Wert 0 annahmen, da es im Weiteren Verlauf als Divisor für eine Quotientenberechnung dient. Daher wird hier der Wert 1 zugewiesen. Durch die Notation „x0001“ in Zeile 63 wird dieser Wert als Hexadezimalzahl ausgedrückt. Dies dient dazu, die Eingabe zu kürzen, weil als Binärzahl eine Zahl mit 16 Stellen eingegeben werden müsste.

Die eigentliche Berechnung wird über das Taktsignal „clk“ aktiviert, welches periodisch zwischen den Werten 0 und 1 wechselt. Ausgelöst wird die Berechnung bei einer steigenden Flanke des Taktsignals. Dies wird in Zeile 64 durch die Auswertung der Bedingung „rising_edge(clk)“ überprüft. Die Ausführung der Berechnung ist außerdem an die Bedingung geknüpft, dass das Modul aktiv sein soll, d. h. das Signal „enable“ den logischen Wert 1 besitzt (Zeile 65). Anderenfalls werden die folgenden Berechnungen übersprungen, d. h. das Modul nimmt effektiv keine Änderungen an den Signalen vor.

Die Zeilen 66 bis 82 geben die Berechnung wieder. Nach der Iterationsvorschrift in Gleichung (2.77) bestimmt das Vorzeichen von „y_in1“ den Entscheidungsfaktor und somit die Berechnung von „y1“ und „z1“, wodurch sich eine Unterscheidung in zwei Fällen ergibt. In Abweichung von der Iterationsvorschrift, werden beliebige Vorzeichen für den x-Wert „xin1“ zugelassen. Für negative x-Werte ist das Vorzeichen der Änderungen der y- und z-Werte zu ändern, da sonst die gewünschte Verschiebung von y in Richtung 0 nicht realisiert wird. Somit ergeben sich jeweils zwei Unterfälle und insgesamt vier Fälle für die Durchführung der Berechnung. Gemäß der Iterationsvorschrift ist der Betrag der Änderung im i-ten Schritt $x_i \cdot 2^{-i}$ für die y-Werte bzw. 2^{-i} für die z-Werte. Unter Berücksichtigung der vorgenommenen Verschiebung um 2^{14} ergeben sich damit für das Modul i die Beträge $xin1 \cdot 2^{-i}$ bzw. 2^{14-i}. Da die Modulnummer i nicht als Variable vorhanden ist, wird dem Modul durch das Hauptmodul eine generische Variable k1 mit dem Wert 2^i übergeben. Mithilfe dieser Variablen berechnen sich durch „xin1/k1“ bzw. „2**14/k1“ die gewünschten Veränderungsbeträge für die Berechnung von „y1“ und „z1“.

Nach Abschluss der Berechnung sind die Ausgangssignale zu erzeugen. Der x-Wert wird im Zuge des Prozesses nicht verändert, sodass die Eingangsgröße weitergegeben werden kann (siehe Zeile 83). Da das Eingangssignal „xin“ bereits das richtige Format hat, wird es unmittelbar dem Ausgang „xout“ zugewiesen. Die Zuweisung wird innerhalb des Prozesses vorgenommen, weil anderenfalls ein Latch für die Zuweisung erzeugt wird. Das bedeutet, dass diese Zuweisung nicht mit der Flanke des Clock-Signals getaktet, sondern pegelgesteuert ist. Infolgedessen funktioniert die taktgesteuerte Pipeline nicht. Die weiteren Zuweisungen können außerhalb des Prozesses vorgenommen werden, da die verwendeten internen Signale „y1“ und „z1“ getaktet sind (siehe Quellcode 3.5).

```
87   y_out  <= std_logic_vector(to_signed(y1, y_out'length)) ;
88   z_out  <= std_logic_vector(to_signed(z1, z_out'length)) ;
```

Quellcode 3.5: Erzeugung der Ausgangssignale

Für die Weitergabe an das nächste Modul sind die berechneten Größen wieder in Signale des Typs std_logic_vector zu transformieren, was durch die dargestellten Zuweisungen erreicht wird.

3.2.2 Haupt-Modul „Divvend

Die Implementierung des Hauptmoduls divvend beginnt, wie beim Modul divv, mit der Deklaration der Signale sowie der Ein- und Ausgänge des Moduls (siehe Quellcode 3.6). Mit der generischen Variable „Drehung“ in Zeile 43 wird die Anzahl der Iterati-

onsschritte beschrieben. Da die Zählung der Iterationen bei 0 beginnt, wird hier durch die Angabe des Wertes 14 eine Zahl von 15 Iterationen festgelegt. Diese Größe dient im weiteren Verlauf der Erzeugung einer entsprechenden Anzahl an Modulen vom Typ divv. Durch eine Änderung des zugewiesenen Wertes kann die Anzahl der Iterationen auf einfache Weise angepasst werden, wobei 15 die maximale Anzahl von Iterationen ist, die durchgeführt werden kann. Es könnte zwar eine größere Anzahl vorgegeben werden, effektiv werden aber nur 15 Iterationsschritte ausgeführt. Wie bereits beim Modul divv beschrieben, werden – unabhängig von der Anzahl von Iterationen – auf Berechnungsebene die Variablen als Ganzzahlen implementiert, die eine Fixkommazahl mit 14 Nachkommastellen repräsentieren. Über die Zahl 15 hinausgehende Iterationen würden dazu führen, dass Zahlen auftreten, die im definierten Format kleiner als 1 sind und somit als 0 betrachtet werden. Zugehörige Module „divv“ würden keine Änderung der Eingangsgrößen berechnen. Somit führt ein Wert von über 14 für „Drehung“ zu keinem Fehler, aber auch nicht zu einem genaueren Ergebnis,

```
42 entity divvend is
43 generic (Drehung : Integer:=14) ;
44    Port (clk    : in  std_logic ;
45          reset  : in  std_logic ;
46          enable : in  std_logic ;
47          x_in   : in  std_logic_vector(15 downto 0);
48          y_in   : in  std_logic_vector(15 downto 0) ;
49          x_out  : out std_logic_vector(15 downto 0) ;
50          y_out  : out std_logic_vector(29 downto 0) ;
51        result   : out std_logic_vector(15 downto 0) ) ;
52 end divvend;
```

Quellcode 3.6: Deklaration von Variablen und Ports für das Modul divvend

Die Signale in der Port-Definition von Zeile 44 bis 51 in Quellcode 3.6 entsprechen bezüglich ihres Formats und ihrer Bedeutung den Port-Signalen des Moduls divv und sind bereits in Kapitel 3.2.1 ausführlich beschrieben worden. Der wesentliche Unterschied besteht darin, dass es sich hier um die externen Ports des FPGA handelt, über welche die Eingangsgrößen für die Gesamtberechnung und ihre Ergebnisse übermittelt werden. Das Signal „result“ enthält das Ergebnis für z nach der vorgegebenen Anzahl an Iterationen und somit den gewünschten Näherungswert für den Quotienten „y_in“ durch „x_in“. Die Ausgänge „x_out“ und „y_out“ werden eigentlich nicht benötigt. Sie werden beim VHDl-Entwurf implementiert, damit die Richtigkeit der Berechnung überprüft und die Ursachen für gegebenenfalls auftretende Fehler besser lokalisiert werden können.

Des Weiteren sind die internen Signale, welche die einzelnen Module des Typs „divv“ verbinden, zu definieren. Da die Anzahl der Iterationen einfach anpassbar sein soll und eine generische Signalbezeichnung aufwendig ist, werden diese Signale in Arrays

zusammengefasst, deren Größe variiert werden kann. Bei der Generierung der Module kann das benötigte Signal unter Angabe des Index aus dem Array zugeordnet werden.

Die implementierten Arrays sind Listen mit Signalen gleicher Bit-Länge. Da mit 16 Bit und 30 Bit zwei unterschiedliche Bit-Längen verwendet werden, sind zwei Array-Typen anzulegen (siehe Quellcode 3.7

```
57 type A_v16 is array (0 to Drehung) of std_logic_vector(15 downto 0);
58 type A_v30 is array (0 to Drehung) of std_logic_vector(29 downto 0);
```

Quellcode 3.7: Festlegung der Typen für die internen Signale für die Divisionsberechnung

Die einzelnen Ausgangssignale werden ihrer Länge entsprechend einem der beiden Array-Typen zugeordnet (siehe Quellcode 3.8)

```
72   signal x_outl  : A_v16  ;
73   signal y_outl  : A_v30  ;
74   signal z_outI  : A_v16  ;
```

Quellcode 3.8: Definition der Formate für die internen Ausgangssignale

Des Weiteren ist das Eingangssignal „y_in“ auf das bereits beschriebene Format einer Fixkommazahl mit 14 Nachkommastellen zu transformieren. Im Format std_logic_vector wird dies erreicht, indem 14 Bits 0 angehängt werden. Hierzu wird das Hilfssignal „y_in_hilf“ eingeführt. Dessen ersten 16 Komponenten (29 bis 14) sind identisch zu „y_in“. Die weiteren 14 Elemente (13 bis 0) werden auf 0 gesetzt (siehe Quellcode 3.9).

```
78 signal y_in_hilf : std_logic_vector(29 downto 0):=(others =>'0');
79 begin
80  y_in_hilf(29 downto 14)  <=  y_in  ;
81  y_in_hilf(13 downto 0)   <=  (others => '0') ;
```

Quellcode 3.9: Transformation des Eingangssignals für y auf 30 Bit Länge

Das so generierte „y_in_hilf“ Signal dient als Eingangssignal für das erste divv-Modul. Für das Eingangssignal „x_in“ ist eine derartige Transformation nicht erforderlich, weil die für die Berechnung erforderliche Verschiebung um 14 Stellen innerhalb des Moduls vorgenommen wird.

Die Hauptaufgabe des Moduls „divvend“ ist die Generierung der Sub-Module vom Typ „divv“ sowie die Definition der Verbindungen zwischen den Modulen mithilfe der entsprechenden Port-Zuweisungen. Die Implementierung ist in Quellcode 3.10 dargestellt.

```
82  gen1 : for i in 0 to Drehung generate
83
84  Beginn : if i = 0  generate
85    Division : divv
86     generic map ( 1 )
87     port map (clk , reset,enable , x_in , y_in_hilf ,
88               (others => '0') , x_outl(0) , y_outl(0) , z_out(0)) ;
89   end generate ;
90
91  Berechnung : if i >= 1 generate
92    Division : divv
93     generic map ( 2**(i) )
94     port map (clk , reset , enable , x_outl(i-1) , y_outl(i-1) ,
95               z_out(i-1) , x_outl(i) ,y_outl(i) ,z_out(i)) ;
96   end generate ;
97
98  end generate ;
99
100 result <= z_outl(Drehung) ;
101 x_out  <= x_outl(Drehung) ;
102 y_out  <= y_outl(Drehung) ;
```

Quellcode 3.10: Erzeugung und Verbindung der einzelnen Module für die Divisionsberechnung

Im einzelnen werden die Module generiert und die einzelnen Einträge für Variablen und Ports der Deklaration in Quellcode 3.1 entsprechend zugewiesen. Hierbei ist zwischen dem ersten und allen weiteren Modulen zu unterscheiden, da auf das erste Modul die Eingangssignale von „divvend“ zu übertragen sind. Bei der Initialiserung des ersten Moduls wird zunächst durch die Zuweisung „generic map (1)“ in Zeile 89 der generischen Variablen des ersten Moduls (i=0) der Wert $2^0 = 1$ zugewiesen. Anschließend werden in den Zeilen 90 und 91 die Ports gemäß der Reihenfolge im Quellcode 3.1 (Zeilen 35 bis 43) belegt. Dabei werden die Eingänge „clk“, „reset“, „enable“ und „x_in“ von „divvend“ direkt aus dem Hauptmodul übernommen. Anstelle von „y_in“ wird, wie bereits erläutert, das transformierte Signal „y_in_hilf“ verwendet. Die Iteration startet mit $z = 0$, weshalb divvend kein Eingangssignal für diese Größe hat. Daher wird durch die Anweisung „others => 0“ dieser Eingang im ersten Modul auf 0 gesetzt. Die Ausgänge des ersten Moduls bilden jeweils das erste Element (i=0) der Ergebnisarrays.

Die auf das erste Modul folgenden Module werden analog für beliebige positive Werte von i definiert. Wie beim ersten Modul werden die Steuersignale „clk“, „reset“ und „enable“ vom Hauptmodul übernommen. Der einzige Unterschied besteht in den Eingängen für die Größen x, y und z. Diese erhalten die Ausgangssignale vom vorhergehenden Modul zugewiesen, welche aus dem Array mit den Ausgangssignalen ausgelesen werden können. Abschließend werden in Zeile 100 bis 102 die Ausgangssignale des letzten divv-Moduls an die Ausgänge von divvend übergeben.

3.2.3 Simulation

In diesem Abschnitt werden für den Entwurf mithilfe von Vivado die einzelnen Signale simuliert und so die Richtigkeit der Implementierung verifiziert. Zunächst wird der Aufbau der Schaltung überprüft, indem durch „Schematic“ ein Schaltplan in Abhängigkeit der vorgegebenen Anzahl an Iterationen erzeugt wird. Abbildung 3.3. zeigt als Beispiel hierfür den Schaltplan für vier Iterationen (i = 3).

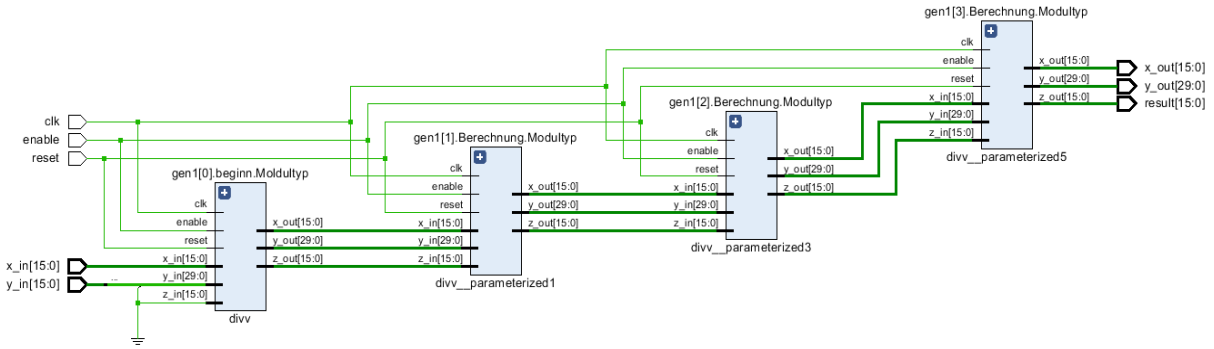

Abbildung 3.3.:Schaltplan der Divisionsberechnung für i=3

Es ist zu erkennen, dass die richtige Anzahl an Modulen erzeugt wird, wobei die Zuweisungen der Blöcke „Beginn“ für das erste Modul links und „Berechnung“ für alle weiteren Module (vgl. Quellcode 3.10) korrekt vorgenommen werden. Das erste Modul erhält die externen Signale „x_in“ und „y_in“ als Eingang. Der Anfangswert für z ist immer Null. Die dementsprechende Zuweisung für den Eingang z_in des ersten Moduls wird korrekt umgesetzt, indem der Eingang mit Masse verbunden wird. Bei den weiteren Modulen werden jeweils die Ausgänge des vorherigen Moduls für x, y, und z auf die zugehörigen Eingänge gelegt. Die Ausgangssignale des letzten Moduls, welche die Endergebnisse der Berechnung enthalten, sind mit den Ausgängen „x_out“, „y_out“ bzw. „result“ verbunden. Im oberen Bereich der Abbildung 3.3. sind die drei Steuersignale „clk“, „reset“ und „enable“ zu sehen. Diese werden, wie geplant unmittelbar vom Eingang des FPGA an die entsprechenden Eingänge der divv-Module weitergegeben.

Nachdem die Richtigkeit der Schaltung überprüft worden ist, werden im zweiten Schritt die einzelnen Signale simuliert. Als Beispiel wird die Berechnung mit 15 Iterationen (i = 14) in Abbildung 3.4. betrachtet. Im rechten Teil der Abbildung ist der zeitliche Verlauf der einzelnen Signale dargestellt. Durch die Platzierung eines Markers (gelbe Linie) werden die Werte ausgelesen und in der Spalte links dargestellt. Die numerischen Werte sind im Format einer Fixkommazahl mit 14 Stellen dargestellt.

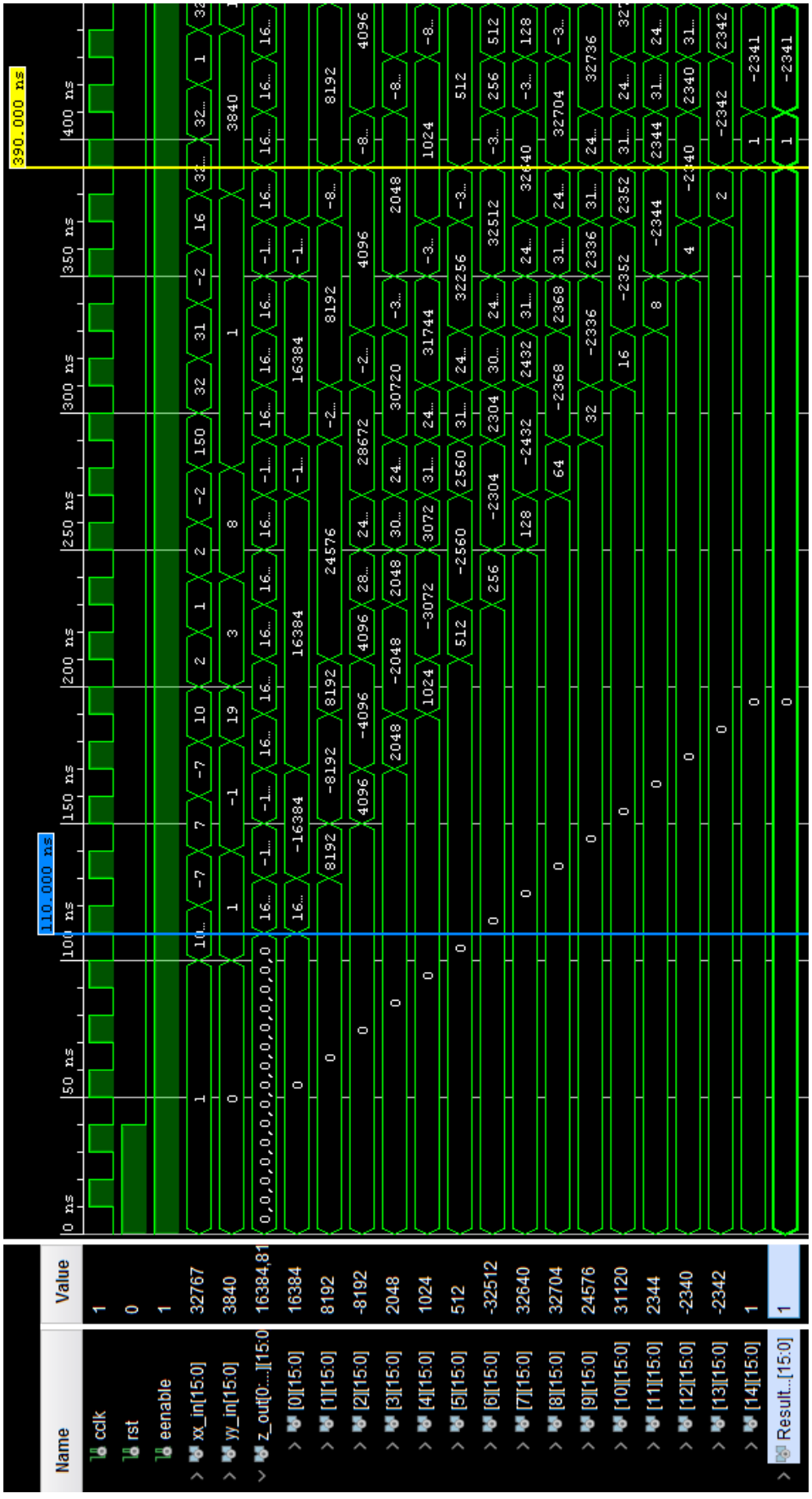

Abbildung 3.4.: Verhaltenssimulation der Signale der Divisionsberechnung für i=14

Die oberste drei Signale sind die Steuersignale. Während „reset“ und „enable“ die konstanten Werte 0 bzw. 1 annehmen, wechselt das Taktungssignal „clk“ periodisch zwischen 0 und 1. Hierfür wird, wie aus der obenliegenden Zeitskala abzulesen ist, eine Gesamtperiodendauer von 20 ns festgelegt. Allerdings dient diese Simulation primär der Überprüfung der Signale und gibt nicht die tatsächlichen Zeitabläufe für die einzelnen Berechnungen wieder. Durch eine spätere Zeitsimulation werden die tatsächlichen zeitlichen Abläufe quantitativ bestimmt und ggf. Anpassungen an der Periodendauer des Clocking-Signals vorgenommen.

Die Testsignale „x_in“ und „y_in“ werden anhand von Zahlenreihen mit beliebigen Zahlen erzeugt, wobei die Signaländerungen jeweils zum Zeitpunkt einer abfallenden Flanke (Wechsel von 1 auf 0) des Clocking-Signals durchgeführt werden. Durch Vergleich des Clocking-Signals mit den darunter liegenden Signalen ist zu erkennen, dass die Berechnungen in den Sub-Modulen und die Änderungen an deren Ausgangssignalen jeweils bei der steigenden Flanke des Taktsignals durchgeführt werden. Des weiteren ist die Pipeline-Arbeitsweise zu erkennen, d. h. es werden jeweils 14 Berechnungen parallel ausgeführt, die zeitversetzt gestartet werden.

Die Ergebnisse der Berechnung für den Quotienten 1/23 der sich durch Wahl der Eingangssignale „x_in“ = 23 und „y_in“ = 1 ergibt, sind in Tabelle 3.1 aufgelistet.

Tabelle 3.1: Dezimale Berechnungsergebnisse für „x_in“ = 23 und „y_in“ = 1

Modul Nr. i	z_out(i) = $z_i \cdot 2^{14}$	z_i	Fehler
0	16384	1,000000	2200,00%
1	8192	0,500000	1050,00%
2	4096	0,250000	475,00%
3	2048	0,125000	187,50%
4	1024	0,062500	43,75%
5	512	0,031250	-28,13%
6	768	0,046875	7,81%
7	640	0,039063	-10,16%
8	704	0,042969	-1,17%
9	736	0,044922	3,32%
10	720	0,043945	1,07%
11	712	0,043457	-0,05%
12	716	0,043701	0,51%
13	714	0,043579	0,23%
14	713	0,043518	0,09%
15	713	0,043518	0,09%
⋮	⋮	⋮	⋮

Für die Ergebnisse der einzelnen Module ist es ohne Bedeutung, ob die Berechnung nach dem Modul abgebrochen wird oder noch weitere Iterationen durchgeführt wer-

den. Numerisch stimmen diese Werte exakt mit jenen überein, die durch eine Tabellenkalkulation berechnet wurden und der Tabelle A.2 im Anhang entnommen werden können. Die letzte Zeile enthält das Ergebnis für eine Berechnung mit 16 Iterationen. Wie bereits erwähnt, führt dies nicht zu einem anderem Ergebnis, weil nur mit 14 binären Nachkommastellen gerechnet wird. Daher führen Rechnungen mit einer Iterationsanzahl $i \geq 15$ zu demselben Ergebnis wie bei einer Anzahl von Iterationen von i =14, in diesem Fall: $713 \cdot 2^{-14}$. Des Weiteren ist in Tabelle 3.1 die relative Abweichung zum exakten Ergebnis 1/23 ≈ 0,0434782.. dargestellt. Nach 15 Iterationen wird eine relative Abweichung von 0,09 % erreicht. Das bedeutet, dass die ersten beiden von Null verschiedenen Dezimalstellen exakt sind und die dritte Stelle um 1 vom korrekten Wert abweicht. Die Ergebnisse in Tabelle 3.1 belegen die grundsätzliche richtige Durchführung der einzelnen Iterationsschritte. Im Weiteren wird die Berechnung für unterschiedlich große Werte und Vorzeichen der Eingabesignale untersucht. Die Ergebnisse dieser Simulationen sind in Tabelle 3.2 zusammengefasst.

Tabelle 3.2: Dezimale Berechnungsergebnisse nach 15 Iterationen für verschiedene Quotienten

Nr.	x_in	y_in	z_{exakt}	z_out(i) = $z_{14} \cdot 2^{14}$	z_{14}	Fehler
1	7	1	0,142857143...	2341	0,1428833	0,0183%
2	-7	-1	0,142857143...	2341	0,1428833	0,0183%
3	7	-1	-0,142857143...	-2341	-0,1428833	-0,0183%
4	-7	1	-0,142857143...	-2341	-0,1428833	-0,0183%
5	2	3	1,5	24575	1,4999389	-0,0041%
6	10	19	1,9	31129	1,8999633	-0,0019%
7	1000	1	0,001	17	0,0010375	3,7597%
8	9000	1	0,000111111...	1	0,0000610	-45,0684%
9	10000	-1	0,0001	-1	-0,0000610	-38.4844%
10	8	1	0,125	2048	0,125	0,0000%
11	1	2	2	32767	1,9999389	–
12	-1	9	-9	-32767	-1,9999389	–

Die ersten vier Zeilen enthalten die simulierten Ergebnisse für die möglichen Eingaben zur Berechnung von 1/7 bzw. -1/7. Es ist zu erkennen, dass unabhängig vom Vorzeichen des Quotienten betragsmäßig dasselbe Ergebnis erreicht wird. Dies belegt, dass die insgesamt vier Varianten, die in Abhängigkeit der Vorzeichen zur Berechnung herangezogen werden (vgl. Quellcode 3.4), korrekt entworfen worden.

Die beiden folgenden Zeilen 5 und 6 zeigen, dass die Berechnungen auch für den Fall „y_in“ größer als „x_in“ d. h. für Quotienten größer als 1 funktionieren. Ferner ist zu erkennen, dass der relative Fehler mit steigendem Betrag des Quotienten tendenziell abnimmt. Dies ist darauf zurückzuführen, das der Algorithmus bei 15 Iterationsschritten mit einer begrenzten Genauigkeit von 14 binären Nachkommastellen arbeitet. So-

mit kann der Betrag des absoluten Fehlers mit $e_{max} = 2^{-14} \approx 6{,}1035 \cdot 10^{-5}$ nach oben abgeschätzt werden. Folglich sinkt der auf den Wert des Quotienten bezogene relative Fehler mit der Größe des Quotienten. Dementsprechend führt die Berechnung von sehr kleinen Quotienten zu großen Fehlern. Zeile 7 zeigt das am Beispiel der Berechnung von 1/1000. Werden, wie in Zeile 8 und 9, Quotienten berechnet, die kleiner sind als 2^{-13} erreicht die Berechnung die Grenze ihres Auflösungsvermögens und es ergibt sich immer das Ergebnis 2^{-14}, was meist einen sehr großen Fehler bedeutet. Quotienten die kleiner sind als 2^{-14} können aufgrund der Restriktion der Bit-Länge für die Eingabegrößen nicht berechnet werden.

Ein Sonderfall ist die Berechnung von Quotienten, die einer ganzzahligen Potenz von 1/2 entsprechen. Sofern diese Potenz kleiner oder gleich 14 ist, führt die Iteration zum exakten Ergebnis. Zeile 10 zeigt dies am Beispiel der Berechnung von 1/8.

Des Weiteren ist, wie bereits in Kapitel 2.4.4 erläutert, die Anwendung des CORDIC-Algorihtmus auf Quotientenbeträge von weniger als 2 beschränkt. Zeile 11 und 12 zeigen die Berechnungsergebnisse für unzulässige Eingaben. Je nach Vorzeichen wächst bzw. sinkt das Ergebnis kontinuierlich bis der maximale Betrag von

$$\left|z_{14}\right|_{max} = \sum_{i=0}^{14} 2^{-i} = \sum_{i=0}^{14} \left(2^{-1}\right)^{i} = \frac{1-2^{-15}}{1-2^{-1}} = \frac{2^{15}-1}{2^{14}} = \frac{32767}{16384} \approx 1{,}9999389 \qquad (3.1)$$

erreicht ist. Dieser Wert entspricht dem maximalen Betrag des Quotienten, der mit 15 Iterationsschritten berechnet werden kann. Für beliebige Eingaben, deren Betrag über diesem Grenzwert liegt, wird immer dieses Resultat berechnet.

Diese Simulationen belegen, dass der Entwurf den CORDIC-Algorithmus korrekt und fehlerfrei abbildet. Die aufgezeigten Probleme der geringen Genauigkeit bei kleinen Quotienten und des nach oben begrenzten Betrags für berechenbare Quotienten können durch eine Vorverarbeitung der Eingangssignale gelöst werden. Durch eine einfache Stellenverschiebung im Binärbereich können günstigere bzw. zulässige Werte für die Eingangssignale erreicht werden. Ein möglicher Ansatz hierzu ist, die Anzahl der binären Stellen der beiden Eingabewerte durch die Multiplikation des kleineren Wertes mit 2^k anzugleichen. Das Ergebnis der Iteration dann durch eine Multiplikation mit bzw. Division durch 2^k auf den tatsächlichen Wert zurückverschoben werden. So würde beispielsweise statt 9/1 in die Berechnung für 9/8 durchgeführt und das Ergebnis mit 8 multipliziert werden, d. h. um drei Binärstellen nach links verschoben. Statt des Quotienten 1/1000 würde nach diesem Ansatz 512/1000 berechnet und das Ergebnis um neun Stellen nach rechts verschoben werden. Allerdings können diese Operationen wegen der auf 16 Bit beschränkten Länge des Ausgangssignals nicht auf dem

FPGA realisiert werden. Diese begrenzt das binäre Format des Berechnungsergebnisses auf eine Vorkomma- und 14 Nachkommastellen. Somit können von Null verschiedene Zahlen nur im Wertebereich $2^{-14} \leq z \leq 2 - 2^{-14}$ dargestellt werden. Da der Fokus dieser Arbeit auf der Implementierung des CORDIC Algorithmus auf dem FPGA liegt, werden solche Konzepte zur Erweiterung des Gültigkeitsbereichs und Verbesserung der Genauigkeit mithilfe von Operationen, die außerhalb des FPGA durchzuführen sind, nicht näher betrachtet.

Des Weiteren wird eine Zeitsimulation durchgeführt, um die korrekte Funktion des Entwurfs auf der Zeitebene zu überprüfen. Ein Ausschnitt des Simulationsergebnisses ist in Abbildung 3.5. dargestellt.

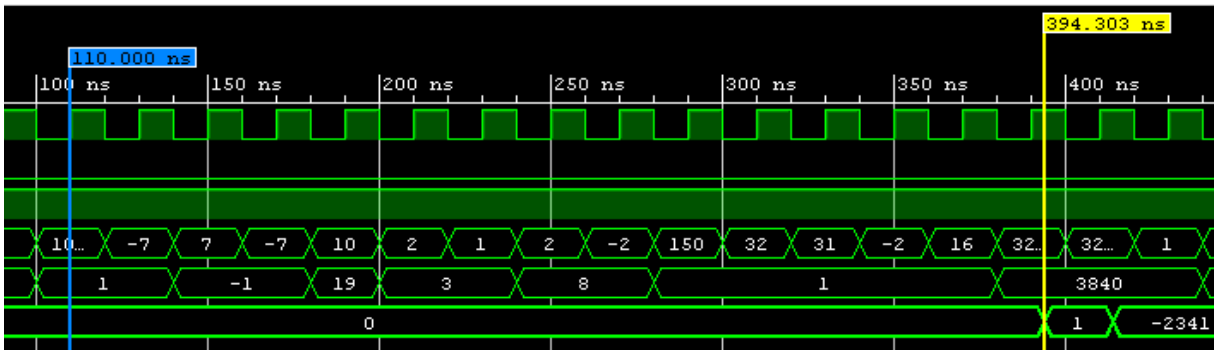

Abbildung 3.5.: Ergebnisse der Zeitsimulation der Division mit $i = 14$ für einen Takt von 50 MHz

Die Berechnung startet bei einer positiven Flanke mit der Zeitkoordinate 110 ns (blaue Linie). Der Abschluss der Berechnung ist daran zu erkennen, dass sich das unterste Signal, welches in Abbildung 3.5. dargestellt ist, von 0 auf 1 ändert. Für diesen Wechsel (gelbe Linie) ergibt sich eine Zeitkoordinate von 394,303. Bei idealer Operation der Bauelemente ohne Zeitverzug würde die Berechnung 14 Arbeitstakte mit jeweils 20 ns, d. h. 280 ns erfordern. Unter Berücksichtigung der Anfangszeit liegt der „ideale“ Abschlusszeitpunkt der Simulation bei 390 ns. Somit liegt eine Verzögerung von 4,303 ns vor. Diese Zeitspanne resultiert aus einer Schaltverzögerung, die in einzelnen Gattern bzw. Flipflops zwischen der Änderung des Eingangssignals und der entsprechenden Änderung des Ausgangssignals auftritt. Mit dem verwendeten FPGA wären zwar prinzipiell Taktfrequenzen von bis zu 100 MHz möglich, jedoch zeigt die Zeitsimulation, dass für Frequenzen von mehr als 50 MHz bzw. Taktzeiten von weniger als 20 ns, die Berechnung nicht korrekt funktioniert.

Als Ergebnis der durchgeführten Simulationen kann festgehalten werden, dass der Entwurf wie gewünscht funktioniert. Unter Berücksichtigung der erläuterten Einschränkungen, die sich aus dem CORDIC-Algorithmus und der maximalen Signallänge

ergeben, sind die Berechnungsergebnisse korrekt. Des Weiteren ist die korrekte Funktion auf eine Taktfrequenz von maximal 50 MHz beschränkt.

3.3 Arcussinus-Berechnung in VHDL

Das Konzept für die Umsetzung der Berechnung des Arkussinus mithilfe des CORDIC-Algorithmus ähnelt weitgehend dem Ansatz für die Divisionsberechnung (siehe Abbildung 3.2.). Es werden wieder zwei Modul-Typen für die Berechnung angelegt. Das Hauptmodul „Arsin_end“ steuert die Berechnung und erzeugt die Submodule vom Typ „arsin“, mit denen jeweils ein Iterationsschritt umgesetzt wird.

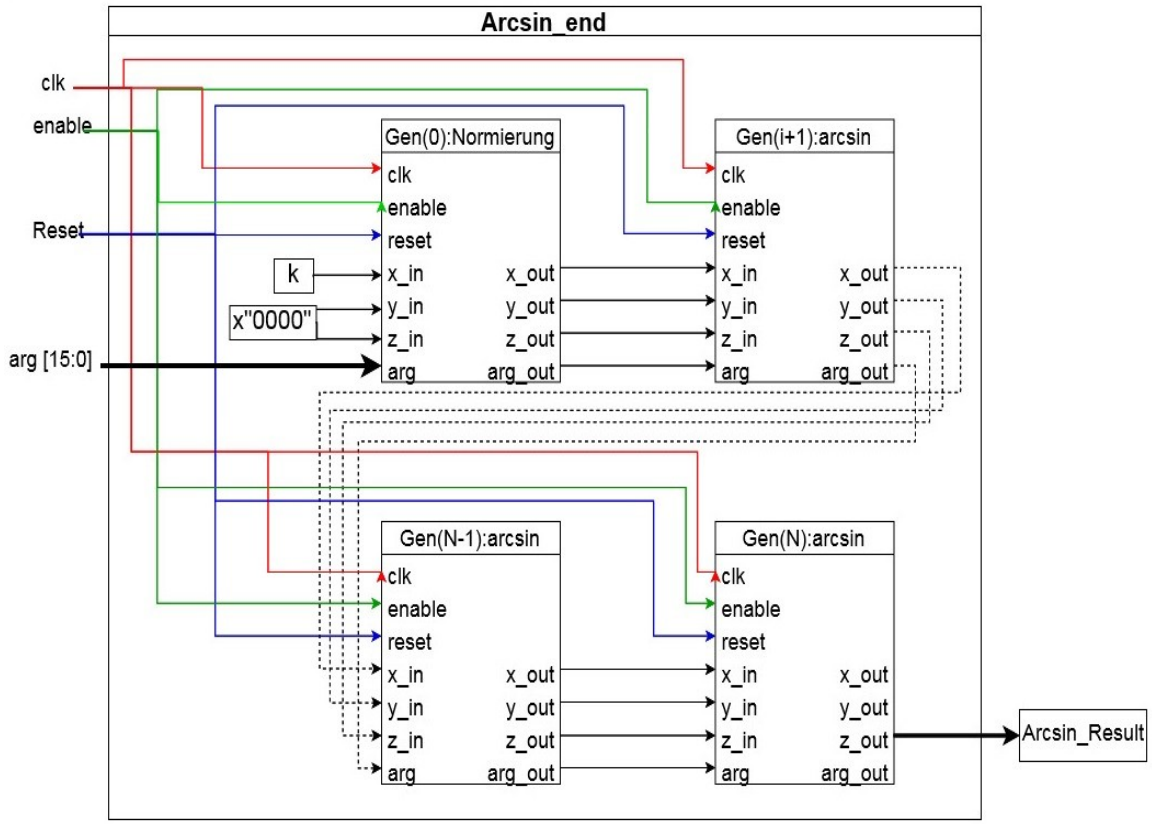

Abbildung 3.6.: Konzept für die Realisierung des CORDIC-Algorithmus zur Berechnung des Arkussinus mit FPGA (eigene Darstellung)

Der wesentliche Unterschied zur Division besteht in den Eingangssignalen. Für die Arkussinus-Berechnung wird zusätzlich das Signal „arg“ verwendet. Dieses stellt das Argument dar, für das die Berechnung ausgeführt werden soll. Dieses wird im Zuge der Iteration nur für die Ermittlung der Entscheidungsfaktoren benötigt und daher unverändert von Submodul zu Submodul weitergegeben. Im Gegensatz dazu wird bei der Division der zu berechnende Quotient durch Eingangswerte für x (Nenner) und y

(Nenner) festgelegt. Hier werden Eingangssignale für x, y und z nicht benötigt, da die Startwerte für diese Größen durch die Iterationsvorschrift (siehe. Kapitel 2.4.2) vorgegeben sind.

3.3.1 Submodul „Arcsin

Ausgangspunkt für den Entwurf dieses Moduls ist dessen Deklaration und die Beschreibung der eingehenden Signale sowie der Definition der Ein- und Ausgänge (siehe Quellcode 3.11). Das Modul verwendet zwei generische Variablen (Zeile 18 und 19) die in Abhängigkeit der Iterationsnummer i durch das Hauptmodul erzeugt werden. Bei „k1“ handelt es sich um einen Divisor der Größe 2^i, der für die Berechnung des Ausgangswertes „y_out“ benötigt wird. Die Variable ist identisch zur Variable k1, die bei der Divisionsberechnung eingeführt wurde. Die zweite generische Variable nimmt den Wert arctan(2^{-i}) an und stellt die Schrittweite für die Änderung von z im i-ten Iterationsschritt dar.

```
17 entity arcsin is
18 generic(k1   : integer ;
19         arct : integer) ;
20 Port ( clk : in  std_logic ;
21      reset : in  std_logic ;
22     enable : in  std_logic ;
23       x_in : in  std_logic_vector(15 downto 0) ;
24       y_in : in  std_logic_vector(15 downto 0) ;
25       z_in : in  std_logic_vector(15 downto 0) ;
26        arg : in  std_logic_vector(15 downto 0) ;
27    arg_out : out std_logic_vector(15 downto 0) ;
28      x_out : out std_logic_vector(15 downto 0) ;
29      y_out : out std_logic_vector(15 downto 0) ;
30      z_out : out std_logic_vector(15 downto 0) ) ;
31 end arcsin;
```

Quellcode 3.11: Deklaration von Variablen und Ports für das Modul „arcsin“

Die Ein- und Ausgangssignale in den Zeilen 20 bis 30 sind abgesehen von den zusätzlichen Signalen „arg“ und „arg_out“ in Bezug auf Format und Bedeutung ähnlich zu denen des Moduls „divv“ (siehe Quellcode 3.1) und wurden bereits dort erläutert.

Die Besonderheit bei diesem Modul ist, dass für alle numerischen Größen Signale mit der gleichen Bitbreite von 16 Bit verwendet werden. Dieses Format entspricht jeweils einer vorzeichenbehafteten Fixkommazahl mit 14 Nachkommastellen. Dadurch ist der darstellbare Zahlenbereich auf einen Betrag von maximal $2 - 2^{-14}$ beschränkt, was für diese Berechnung ausreichend ist. Die tatsächlichen auftretenden Grenzwerte sind:

- Da die Sinusfunktion auf den Wertebereich [-1 ; 1] abbildet, können zulässige Eingaben nur aus diesem Bereich stammen, d. h. $|\mathrm{arg}| \leq 1$.

- Bei der Berechnung von z wird in jedem Schritt eine Änderung von arctan(2^{-i}) addiert bzw. subtrahiert. Somit ist der maximal berechenbare Betrag beschränkt auf

$$|z| \leq \sum_{i=0}^{14} \arctan\left(2^{-i}\right) \approx 1{,}78 \tag{3.2}$$

  - Die Werte für x und y nähern sich ausgehend von den Startwerten 1/K≈0,6 und 0 an die Werte

$$\lim_{i\to\infty} x_i = \sqrt{1-arg^2} \quad \text{und} \quad \lim_{i\to\infty} y_i = arg \tag{3.3}$$

    an. Daher sind die Werte im Lauf der Iteration betragsmäßig immer kleiner als der Maximalwert des Arguments, d. h. 1.

Die modulinterne Berechnung erfolgt mit internen Signalen im Integerformat. Diese Signale repäsentieren die Anfangswerte („x0“, „y0“, „z0“), die Berechnungsergebnisse („x1“, „y1“, „z1“) sowie das auszuwertende Argument („arg0“). Wie bereits beim Modul „DIVV“ erläutert, ist die Länge dieser Signale zu beschränken, da sonst die Signale automatisch in 32 Bit Länge erzeugt werden, was deutlich länger als erforderlich ist. Hier wird für alle Größen dasselbe Zahlenformat wie für die externen Signale verwendet, nämlich signed Integer mit 16 Bit Länge (siehe Quellcode 3.12).

```
36  SUBTYPE S8 IS INTEGER RANGE  -2**15 TO (2**15)-1 ;
37
38  signal x1 , x0 , y0 , y1 , z1 , z0, arg0 : S8 :=0 ;
```

Quellcode 3.12: Beschränkung der Länge der Variablen im Modul „Arcsin“

Im Quellcode 3.12 wird zunächst für die internen Signale ein Subtyp des Formats signed Integer mit einer Läge von 16 Bit angelegt und anschließend die internen Signale mit diesem Typen angelegt und mit dem Wert 0 initialisiert.

Da die externen Signale, dasselbe Zahlenformat als Std_Logic_vector verwenden, können die externen Eingangssignale nach der Umwandlung in Integer-Signale direkt dem zugehörigen internen Signal zugewiesen werden (siehe Quellcode 3.13).

```
40     x0 <= to_integer(signed(x_in));
41     y0 <= to_integer(signed(y_in));
42     z0 <= to_integer(signed(z_in));
43   arg0 <= to_integer(signed(arg)) ;
```

Quellcode 3.13: Definition der inneren Signale Modul „Arcsin“

Der Entwurf des Prozesses für die Durchführung der Iterationsschritte des CORDIC-Algorithmus im Submodul ist in Quellcode 3.14 dargestellt. Wie bereits beim Modul „DIVV“ (siehe Quellcode 3.4) wird der Prozess mit einer Sensitivy-Liste aufgerufen, welche aus den Steuersignalen „clk“ und „reset“ besteht. Die logische Struktur der Ak-

tionen, die nach dem Triggern des Prozesses infolge der Änderung eines dieser Signale durchgeführt werden, ist im Prozess im Quellcode 3.4 zu erkennen: Falls das Signal „reset“ den Wert Logisch 1 hat, wird die Berechnung zurückgesetzt, wobei alle Ausgangsignale auf 0 gesetzt werden (siehe Zeile 47 bis 50). Anderenfalls wird unter der Bedingung, dass „Enable“ den Wert einer logischen 1 hat, bei einer steigenden Flanke des Takt-Signals „clk“ die Berechnung ausgelöst (siehe Zeile 51 und 52).

```
45 process (clk,reset)
46   begin
47    if  (reset = '1') then
48             x1  <= 0;
49             y1  <= 0;
50             z1  <= 0;
51    elsif rising_edge(clk) then
52     if enable = '1' then
53       if  (arg0 - y0 < 0) then
54             x1 <= x0 + y0/k1 ;
55             y1 <= y0 - x0/k1 ;
56             z1 <= z0 - arct  ;
57       elsif  (arg0 -y0 > 0)  then
58             x1 <= x0 - y0/k1 ;
59             y1 <= y0 + x0/k1 ;
60             z1 <= z0 + arct  ;
61       else
62             x1 <= x0  ;
63             y1 <= y0  ;
64             z1 <= z0  ;
65       end if ;
66      arg_out <= arg   ;
67     end if ;
68  end if ;
69  end process ;
```

Quellcode 3.14: Prozess für die Durchführung eines Iterationsschrittes im Modul „Arcsin“

Die eigentliche Berechnung, die in den Zeilen 53 bis 65 entworfen ist, entspricht der Iterationsvorschrift in Gleichung 2.52. Dabei wuerden die drei Berechnungsvarianten, die sich aus den drei möglichen Ausprägungen des Entscheidungsfaktors ergeben, über die Auswertung von if-Bedingungen aufgerufen. Für die Berechnung von „x1“, „y1“ sowie „z1“ werden die Schrittweiten 2^{-i} bzw. $\arctan(2^{-i})$ benötigt. Das Problem, dass auf der Ebene des Moduls die Iterationsnummer „i“ nicht verfügbar ist, wird wie bei der Divisionsberechnung durch generische Variablen gelöst. Die generischen Variablen k1 und arct, werden wie bereits erwähnt vom Hauptmodul „Arcsin-End erzeugt und dem Submodul übergeben. Im letzten Teil des Prozesses wird nach Abschluss der Berechnungen das eingegangene Argument (arg_in) unverändert dem Ausgang (arg_out) zugewiesen (siehe Zeile 66). Diese Vorgehensweise wurde bereits beim unveränderten Signal „x-in“ im Prozess des Moduls „Divv“ angewendet und dient dazu,

die Störung des Berechnungsverlaufes durch die Generierung ungetakteter Latches zu verhindern.

```
70    x_out  <= std_logic_vector(to_signed(x1, x_out'length)) ;
71    y_out  <= std_logic_vector(to_signed(y1, y_out'length)) ;
72    z_out  <= std_logic_vector(to_signed(z1, z_out'length)) ;
```

Quellcode 3.15:Erzeugung der Ausgangssignale im Modul „Arcsin“

Die anderen Signale, welche durch den getakteten Prozess erzeugt worden sind, werden nach der Umwandlung in einen std_logic_vector den zugehörigen Ausgangsgrößen zugewiesen (siehe Quellcode 3.16).

3.3.2 Haupt-Modul „Arcsin_end

Die Deklaration der Variablen sowie der Ein- und Ausgänge de Hauptmoduls für die Berechnung des Arkussinus erfolgt mithilfe des dargestellten Entwurfs im Quellcode 3.16.

```
37 entity arcsin_end is
38  generic (   N : Integer := 14) ;
39     Port ( clk : in  std_logic ;
40           reset : in  std_logic ;
41          enable : in  std_logic ;
42             arg : in  std_logic_vector(15 downto 0);
43  Arcsin_result : out std_logic_vector(15 downto 0) ) ;
44 end arcsin_end;
```

Quellcode 3.16: Deklaration von Variablen und Ports für das Modul Arcsin_end

Durch die generische Variable „N“ wird die Zahl der Iterationsschritte definiert. Die dargestellte Zuweisung des Wertes 14 impliziert die Durchführung von 15 Iterationen im CORDIC-Algorithmus. Wie bereits bei der Divisionsberechnung erläutert und durch Simulation nachgewiesen wurde, können zwar beliebig viele Iterationen vorgegeben werden, aber Werte von N > 14 führen aufgrund der Beschränkung des Zahlenformats auf 14 binäre Nachkommastellen zu keinem genaueren Ergebnis, sondern nur zu einer Verlängerung der Berechnungsdauer. Die Ports „clk“, „reset“ und „enable“ enthalten die bereits hinlänglich erläuterten Steuersignale. Die beiden weiteren Ports erhalten als Signale eine Repräsentation numerischer Werte im Format vorzeichenbehafteter binärer Fixkommazahlen mit 14 Nachkommastellen. Das Signal „arg“ entspricht dem Wert, für den der Arkussinus berechnet werden soll. „Arcsin_result“ ist das Berechnungsergebnis nach „N“+1 Iterationen. Für die Größen x, y und z werden keine externen Signale benötigt. Für die hier durchgeführte Berechnung des Arkussinus sind sie lediglich Hilfsgrößen, die als interne Signale von Submodul zu Submodul weitergegeben werden. Dafür werden, wie bereits beim Hauptmodul der Division erläu-

tert wurde, mehrere Arrays generiert, die alle Berechnungsergebnisse der jeweiligen Größe enthalten (siehe Quellcode 3.15). Die Komponente i ∈ {0,1,...,N} von „x_out“, „y_out“ bzw. „z_out“ enthält den entsprechenden Wert am Ausgang von Submodul i.

```
65 type A_v16 is array (0 to N) of std_logic_vector(15 downto 0);
66
67     signal x_out_l    : A_v16  ;
68     signal y_out_l    : A_v16  ;
69     signal arg_out_l  : A_v16  ;
70     signal z_out_l    : A_v16  ;
```

Quellcode 3.17: Festlegung der Typen für die internen Signale für die Arkussinusberechnung

Wie in Zeile 70 von Quellcode 3.17 zu sehen, wird zusätzlich das interne Signal arg_out_1 definiert. Dieses enthält kein Berechnungsergebnis. Es dient lediglich dazu, das auszuwertende Argument unverändert von Submodul zu Submodul weiterzugeben.

Die Kernaufgabe des Hauptmoduls ist auch bei der Akussinusberechnung die Erzeugung und Verbindung der Submodule welche die einzelnen Berechnungsschritte durchführen. Der zugehörige Entwurf ist in Quellcode 3.18 dargestellt. Durch das „generate“-Statement werden die einzelnen Submodule erzeugt. Hierbei ist zu unterscheiden zwischen dem ersten Modul mit i = 0 (Zeile 76 bis 83) und den weiteren Modulen (Zeile 85 bis 92).

```
74 gen : for i in 0 to n generate
75
76     beginn : if i = 0 generate
77       asin : arcsin
78       generic map (k1=>2**(i),
79                    arct=>integer(16384.0*arctan(2.0**real(-i))) )
80       port map (clk , reset , enable ,x"26e0" , (others => '0') ,
81                 (others => '0') , arg , arg_out_l(i) , x_out_l(i) ,
82                 y_out_l(i) , z_out_l(i));
83     end generate ;
84
85     D : if i >= 1 generate
86       asin : arcsin
87       generic map (k1=>2**(i),
88                    arct=>integer(16384.0*arctan(2.0**real(-i))) )
89       port map (clk , reset , enable ,x_out_l(i-1) , y_out_l(i-1) ,
90                 z_out_l(i-1),arg_out_l(i-1), arg_out_l(i) ,
91                 x_out_l(i) , y_out_l(i) , z_out_l(i));
92     end generate ;
93
94 end generate ;
95
96 Arcsin_result<=z_out_l(N) ;
```

Quellcode 3.18: Erzeugung und Verbindung der einzelnen Module für die Arkussinusberechnung

In beiden Fällen werden zunächst die generischen Variablen „k1“ und „arct“ erzeugt, welche die Verschiebungen für die Größen x und y bzw. z bei der Durchführung des CORDIC-Algorithmus enthalten. Durch die Einbindung als generische Variable werden Wertelisten mit Konstanten berechnet und in Registern abgespeichert. Somit entsteht keine zusätzliche Hardware für deren Berechnung. Durch die dargestellte Berechnung wird der Arkustangens von 2^{-i} zunächst als Realzahl berechnet und durch die anschließende Multiplikation mit $2^{14} = 16384$ in das Binärformat konvertiert, welches für die Berechnungen genutzt wird.

Der wesentliche Unterschied zwischen den beiden Submodulvarianten besteht in der Zuweisung der Eingänge und Ausgänge, welche durch das Statement „port map“ gemäß der Reihenfolge in der Moduldefinition (siehe Quellcode 3.11) erfolgt. Für das erste Modul (Zeilen 80 – 82) erhält der Eingang „x_in“ den Wert des Skalierungsfaktors K als Hexadezimalzahl, d. h. 26e0, während die Eingänge „y_in“ und „z_in“ durch die Anweisung „others => '0'“ auf 0 gesetzt werden. Außerdem wird dem Eingang „arg_in“ das Eingangssignal „arg“ des Hauptmoduls zugewiesen. Für die weiteren Module werden diese Eingänge mit den Ausgängen des vorherigen Moduls verbunden (Zeilen 89 – 90). Abschließend wird das Berechnungsergebnis des letzten Moduls auf den entsprechenden Ausgang „Arcsin_result“ des Hauptmoduls übertragen.

3.3.3 Simulation

Der VHDL-Entwurf für die Berechnung des Arkussinus wird wie der Entwurf für die Divisionsberechnung mithilfe von Vivado anhand mehrerer Simulationen überprüft. Die Vorgehensweise ist analog (siehe Abschnitt 3.2.3). Zunächst wird die Schaltung überprüft, danach die Berechnungsergebnisse analysiert und abschließend das Zeitverhalten bestimmt.Beispielhaft wird der Schaltplan für die Schaltung mit 2 Iterationsschritten ($i = 3$) betrachtet. Der durch die Funktion Schematic erzeugte Schaltplan ist in Abbildung 3.7. zu sehen

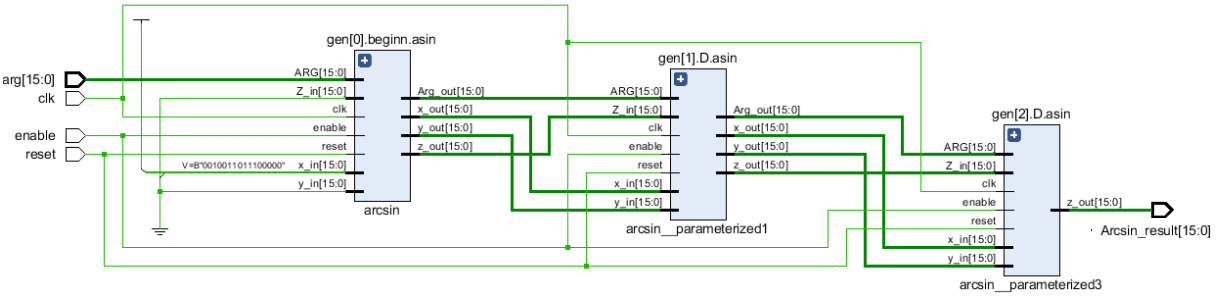

Abbildung 3.7.: Simulierter Schaltplan der Arkussinusberechnung für i=3

Der Schaltplan belegt die grundsätzliche Richtigkeit der Implementierung. Es ist klar zu erkennen, dass drei Submodule erzeugt worden sind. Die drei externen Steuersignale „clk“, „enable“ und „reset“ werden an die zugehörigen Eingänge aller Submodule geleitet. Das vierte Signal „arg“ mit dem auszuwertenden Argument wird hingegen nur zum ersten Modul geführt. Des Weiteren erhält das erste Modul ein Signal mit einem konstanten Input („x_in“), wobei der exakte Wert im Schaltplan als Binärzahl dargestellt ist. Die beiden verbliebenen Eingänge („y_in“ und „z_in“) sind mit der Masse verbunden, d. h. mit 0 belegt. Außerdem ist zu erkennen, wie die vier internen Signale jeweils von den Submodulausgängen zu den entsprechenden Eingängen des Folgemoduls geführt werden. Eine Ausnahme bildet nur das letzte Modul, bei dem nur ein Ausgang, nämlich der für das Berechnungsergebnis, benötigt wird.

Im zweiten Schritt werden die Signalverläufe und die Berechnungsergebnisse der einzelnen Module analysiert. Das Ergebnis für die Simulation mit 15 Iterationen (i=14) ist Abbildung 3.8. zu entnehmen. Der Verlauf ähnelt dem Signalverlauf für die Divisionsberechnung. Der Tabelle am linken Rand können die einzelnen Signalbezeichnungen und die Werte an der Position des gelben Markers abgelesen werden. Der Marker ist hier am Ende des ersten Iterationsdurchlaufs positioniert.

Die drei obersten Signale sind die Steuersignale, welche zwischen den Werten 0 und 1 wechseln können. Zunächst steht das „reset“-Signal auf „1“ besitzt hat, wodurch alle Ausgangssignale den Wert „0“ erhalten. Da „enable“ permanent den Wert „1“, kann nach dem Wechsel von „reset“ auf „0“ die Berechnung gestartet werden. Die zu berechnenden Werte werden mit dem Signal „arg“ zugewiesen. Zunächst hat dieses Signal den Wert „0“. Danach wird eine Folge von Werten zugewiesen, wobei der Wechsel zum nächsten Wert jeweils bei einer fallenden Flanke des „clk“-Signals erfolgt.

Es ist zu erkennen, dass nach Zugeweisung der Werte, die Signale zwischen den Modulen, welche alle anfänglich den Wert 0 hatten, sukzessive die Berechnungsergebnisse des ersten Argumentes erhalten. In Folge werden diese durch die Berechnungsergebnisse der folgenden Schritte überschrieben. Das bedeutet, dass die Pipeline wie gewünscht funktioniert.

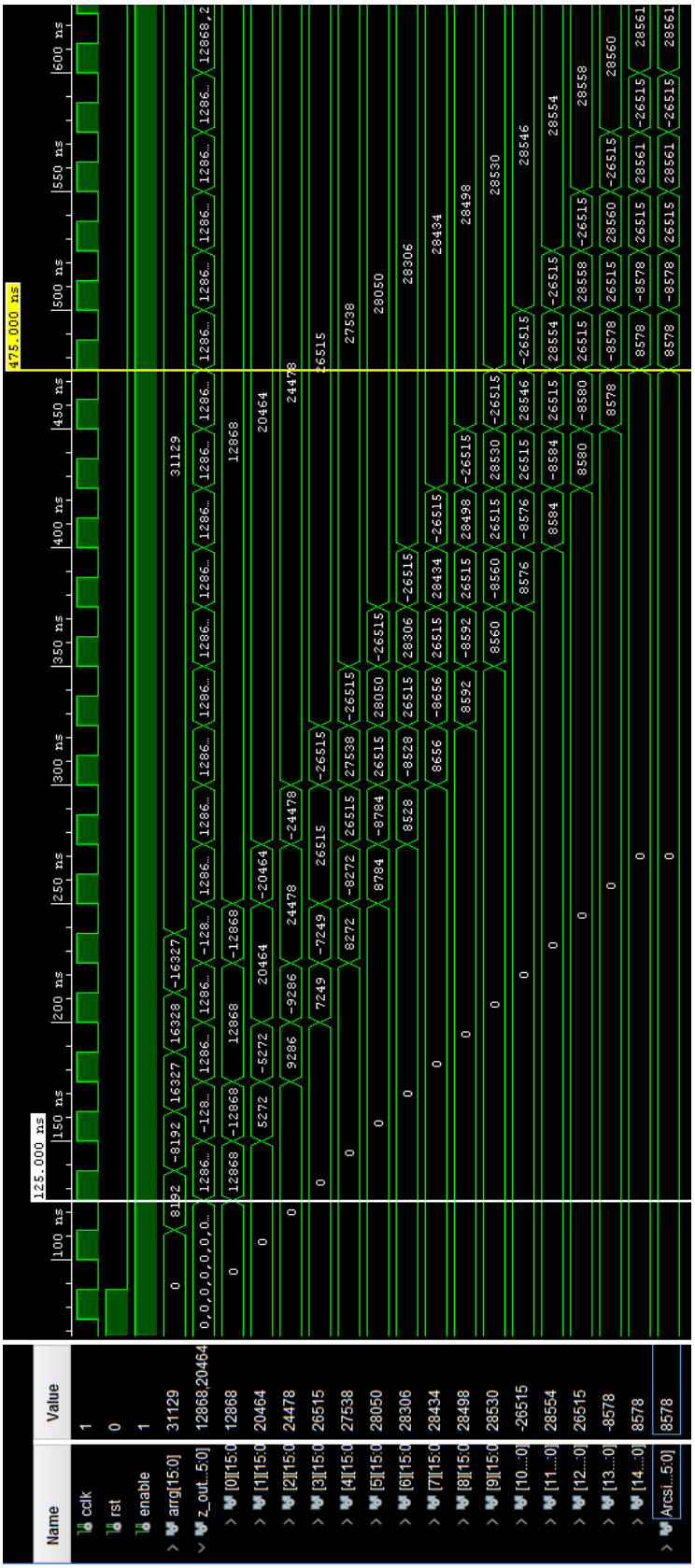

Abbildung 3.8.: Verhaltenssimulation der Signale der Arcus-Sinus-Berechnung für i=14

Im Folgenden wird der Berechnungsablauf für ein Argument von 0,5 = $8192 \cdot 2^{-14}$ im Detail betrachtet. Die Ergebnisse der einzelnen Submodule sind in Tabelle 3.3 zusammengestellt. Zusätzlich ist die relative Abweichung zum exakten Ergebnis von $\pi/6 = 30°$ angegeben.

Tabelle 3.3: Dezimale Berechnungsergebnisse der Arkussinusberechnung mit „arg“ = 0,5

Modul Nr. i	z_out(i) = $z_i \cdot 2^{14}$	z_i		Fehler
0	12868	0,785400	45,000°	50,00 %
1	5272	0,321777	18,436°	-38,55 %
2	9286	0,566772	32,474°	8,25 %
3	7249	0,442444	25,350°	-15,50 %
4	8272	0,504883	28,928°	-3,57 %
5	8784	0,536133	30,718°	2,39 %
6	8528	0,520508	29,823°	-0,59 %
7	8656	0,528320	30,271°	0,90 %
8	8592	0,524414	30,047°	0,16 %
9	8560	0,522461	29,935°	-0,22 %
10	8576	0,523438	29,991°	-0,03 %
11	8584	0,523926	30,019°	0,06 %
12	8580	0,523682	30,005°	0,02 %
13	8578	0,523560	29,998°	-0,01 %
14	8578	0,523560	29,998°	-0,01 %
15	8578	0,523560	29,998°	-0,01 %
⋮	⋮	⋮		⋮

Die dargestellten Berechnungsergebnisse entsprechen denen einer theoretischen Kalkulation (siehe Anhang B). Bei dieser theoretischen Berechnung wurden bei den einzelnen Termen in den Berechnungsformeln die Dezimalanteile, welche über 14 binäre Nachkommastellen hinaus gehen, ohne Rundung gestrichen. Unter Berücksichtigung dieser Einschränkung, welche aus dem gewählten Zahlenformat resultiert, stimmen die einzelnen Ergebnisse überein. Somit ist die Berechnung korrekt implementiert.

Die Berechnungsergebnisse der Submodule stellen nicht nur Zwischenergebnisse dar, sondern auch das Endergebnis, falls die Berechnung nach diesem Modul abgebrochen wird. Aus Tabelle 3.3 ist zu entnehmen, dass nach 14 Iterationen (i=13) der Endwert erreicht wird, was darauf zurückzuführen ist, dass der numerische Wert für „y_out(13)“ identisch zum Argument ist und deshalb der Entscheidungsfaktor in den folgenden Schritten den Wert 0 annimmt. Außerdem wurden mehr als 15 Iterationsschritte simuliert, was aber wie erwartet, zu keinem anderen Ergebnis führt als 15 Iterationen. Der Grund dafür ist, dass die Verschiebungen der Berechnungsgrößen, deren Beträge die mathematische Form $y_i \cdot 2^{-i}$, $x_i \cdot 2^{-i}$ und $\arctan(2^{-i})$ haben, im vorgegeben Zahlenformat 0 sind.

Um die Korrektheit des Entwurfs für beliebige Argumente und die Genauigkeit des Algorithmus zu eruieren, werden weitere Simulationen für verschiedene Argumente mit i=14 simuliert. Die Simulationsergebnisse sind in Tabelle 3.4 zusammengestellt. Zusätzlich sind die „exakten“ Ergebnisse, welche mit einem Taschenrechner auf 12 Stellen genau berechnet wurden, sowie die relative Abweichung der Iterationsergebnisse von diesen Werten dargestellt.

Tabelle 3.4: Dezimale Berechnungsergebnisse nach 15 Iterationen für verschiedene Argumente

Nr.	arg	arg· 2^{14}	z_out(i) = $z_{14} \cdot 2^{14}$	z_{14}	z_{exakt}	Fehler
1	0,25	4096	2341	14,471°	14,478°	-0,05 %
2	-0,25	-4096	2341	-14,471°	-14,478°	-0,05 %
3	0,75	12288	-2341	48,574°	48,590°	-0,03 %
4	-0,75	-12288	-2341	-48,574°	-48,590°	-0,03 %
5	0,01	164	165	0,577°	0,573°	0,71 %
6	0,001	16	17	0,059°	0,057°	3,76 %
7	2^{-13}	2	3	0,010°	0,007°	50,00 %
8	1	16384	28561	99,879°	90,000°	10,98 %
9	-1	-16384	-28561	-99,879°	-90,000°	10,98 %
10	~0,997	16327	26515	92,724°	85,219°	8,81 %
11	~0,996	16326	24469	85,569°	85,178°	0,46 %

Die Simulationen 1 bis 4 zeigen die Ergebnisse für verschiedene Werte mit unterschiedlichem Vorzeichen. Die Ergebnisse sind unabhängig vom Vorzeichen, d. h. für Argumente gleichen Betrages wird ein betragsmäßig identisches Ergebnis erreicht. Dies gilt nicht nur für die dargestellten Ergebnisse nach 15 Iterationen, sonder auch für jeden einzelnen Schritt der Iteration. Insgesamt wird bei Argumenten mittlerer Größe eine sehr gute Genauigkeit erreicht.

Bei sehr kleinen Argumenten, wie sie in den Simulationen 5,6 und 7 untersucht werden ist das Ergebnis meist im Betrag um $1 \cdot 2^{-14}$ größer als das Argument. Vor dem Hintergrund, dass für sehr kleine Winkel die Näherungsbeziehung

$$\sin(\varphi) = \varphi - \frac{1}{6}\varphi^3 + \frac{1}{120}\varphi^5 + \ldots + \frac{(-1)^{2n-1}}{(2n-1)!}\varphi^{2n-1} + \ldots \approx \varphi \qquad (3.4)$$

gilt, kann der absolute Fehler der Ergebnisse für kleine Argumente mit $1 \cdot 2^{-14} \approx 0{,}0035°$ abgeschätzt werden. Dies ist zwar gering , kann aber bei entsprechend kleinen Argumenten zu einer erheblichen relativen Abweichung führen.

In den Simulationen 8 und 9 werden die Arkussinuswerte für die maximal zulässigen Argumente bestimmt. Hier führt die Simulation zu einem Ergebnis, das deutlich größer als das korrekte Resultat von $\pi/2 = 90°$ ist. Ein Grund für dieses Resultat ist, dass

die Summe der Drehungen nicht auf den Bereich ± 90° beschränkt ist. Werden alle Teildrehungen in eine Richtung ausgeführt, ergibt sich für die Gesamtdrehung ein Betrag von

$$\varphi_{max} = \max\left|z_{14}\right| = \sum_{i=0}^{14} \arctan\left(2^{-i}\right) = 1{,}7422\ldots \triangleq 28561 \cdot 2^{14} \tag{3.5}$$

Dieser maximale Winkel wird nicht nur bei der Simulation des maximalen zulässigen Argumentes, sondern auch bei etwas kleineren Argumenten erreicht. Um dieses Verhalten des Algorithmus zu erläutern, werden in Tabelle 3.5 die Berechnungsergebnisse der ersten Iterationsschritte der Simulation dargestellt. Zum besseren Verständnis ist der dem Wert von z_i entsprechende Winkel angegeben.

Tabelle 3.5: Einzelschritte der Berechnung von arcsin(1)

Modul i	x_out(i) = $x_i \cdot 2^{14}$	y_out(i) = $y_i \cdot 2^{14}$	z_out(i) = $z_i \cdot 2^{14}$	z
0	9952	9952	12868	45,00°
1	4976	14928	20464	71,56°
2	1244	16172	24478	85,60°
3	-777	16327	26515	92,72°
4	-1797	16279	27538	96,30°
5	-2305	16223	28050	98,09°
⋮	⋮	⋮	⋮	⋮

Entscheidend für dieses Verhalten ist, dass für die Festlegung des Entscheidungsfaktors lediglich das Argument – hier 1 = 16384·2^{-14} -- mit dem Wert y_i verglichen wird, genauer gesagt: solange das Argument größer ist als y_i wird weiter gegen den Uhrzeigersinn gedreht. Aus der Tabelle wird ersichtlich, dass der „kritische“ Punkt für das Überdrehen über 90° hinaus in Modul 3 erreicht wird. Ist das Argument größer als der Ausgangswert für y aus Modul 2, d. h. 16172·2^{-14}, erfolgt eine weitere Teildrehung gegen den Uhrzeigersinn. Dies hat zur Folge, dass der Ausgangswert von Modul 3 für x negativ wird, was gleichbedeutend mit einer Gesamtdrehung von mehr als 90° ist. Ist das Argument kleiner als 16172·2^{-14}, würde in Gegenrichtung gedreht und wegen der rasch abfallen Größe der Teildrehungen zu keinem späteren Zeitpunkt eine Gesamtdrehung von 90° erreicht. Für die Stabilität des Algorithmus ist allerdings ein anderer Wert des Arguments ausschlaggebend. Das einmalige Überschreiten von 90° kann kompensiert werden, falls im nächsten Schritt im Uhrzeigersinn gedreht wird. Dafür ist es erforderlich, dass das Argument kleiner ist als der angegebene y-Ausgangswert von Modul 3, d. h. 16327·2^{-14}. Anderenfalls erfolgt eine weitere Drehung entgegen des Uhrzeigersinns, wodurch y immer weiter abnimmt und sich so dessen Differenz zum Argument mit jedem weiteren Schritt anwächst anstatt reduziert zu werden. Die Folge davon ist

dass kontinuierlich bis zum Ende weiter entgegen des Uhrzeigersinns gedreht und der oben genannte Endwert der Drehung erreicht wird.

Somit ist die Berechnung auf Argumente beschränkt, deren Betrag kleiner ist als der Wert $16326 \cdot 2^{-14} \approx 0{,}99646$. Im Gegensatz zur Division lässt sich diese Begrenzung der Genauigkeit des CORDIC-Prozessors nicht durch eine Verschiebeoperation des Eingangssignals vor der Zuweisung an den Prozessor beheben. Der Grund dafür liegt in der Nichtlinearität der inversen trigonometrischen Funktionen. Im Grunde existieren hier zwei Lösungsansätze. Die erste Möglichkeit ist eine Modifikation des Berechnungsalgorithmus, bspw. indem das Vorzeichen von x bei der Bestimmung des Entscheidungsfaktors berücksichtigt oder beschränkt wird. Dies würde aber einen erhöhten Verarbeitungsaufwand und damit höhere Berechnungszeiten erfordern. Der andere Ansatz für die Lösung des Problems ist die Betrachtung der Abweichung von 1 bzw. dem Winkel $\pi/2$. Aus der grundlegenden Identität $\sin^2(\varphi) + \cos^2(\varphi) = 1$ folgt für den Bereich $|\varphi| \leq \pi/2$

$$\cos(\varphi) = \sqrt{1 - \sin^2(\varphi)} = \sqrt{1 - arg^2} \tag{3.6}$$

Somit gilt für die Auswertung des Arguments

$$\varphi = \operatorname{sgn}(arg) \cdot \arccos\left(\sqrt{1 - arg^2}\right) = \operatorname{sgn}(arg) \cdot \left(\frac{\pi}{2} - \arcsin\left(\sqrt{1 - arg^2}\right)\right) \tag{3.7}$$

was allerdings eine relativ aufwendige Manipulation des Eingangssignals darstellt. Ferner greift Gleichung (3.7) den grundlegenden Zusammenhang zwischen Arkussinus und Arkuskosinus auf, der bereits in Gleichung (2.54) aufgezeigt wurde. Für die Berechnung des Arkuskosinus ist daher kein eigener CORDIC-Prozessor notwendig. Stattdessen kann für das Argument der Arkussinus berechnet werden und anschließend von 90° bzw. $\pi/2$ subtrahiert werden.

Des weiteren wurde – wie bei der Divisionsberechnung, eine Zeitsimulation durchgeführt . Dies führt zu dem Ergebnis, dass die Arkussinusberechnung nur mit einer Taktfrequenz von maximal 40 MHz fehlerfrei realisiert werden kann.

4 Implementierung und Test auf der Hardware

Abschließend wird der Entwurf auf der Hardware implementiert und getestet. Dazu wird ein Board des Typs Nexys 4 Master von der Firma Digilent verwendet (siehe Abbildung 4.), Auf diesem Board ist ein FPGA des Typs Artix-7 von der Firma Xilinx integriert. Die Untersuchungen fanden im Labor der Fachhochschule Dortmund statt. Anzumerken ist, dass aufgrund der Sperrung der Räumlichkeiten für Studierende wegen der Corona-Pandemie die Untersuchungen nicht von mir, sondern von Herrn Dipl.-Ing. Rolf Paulus durchgeführt wurden.

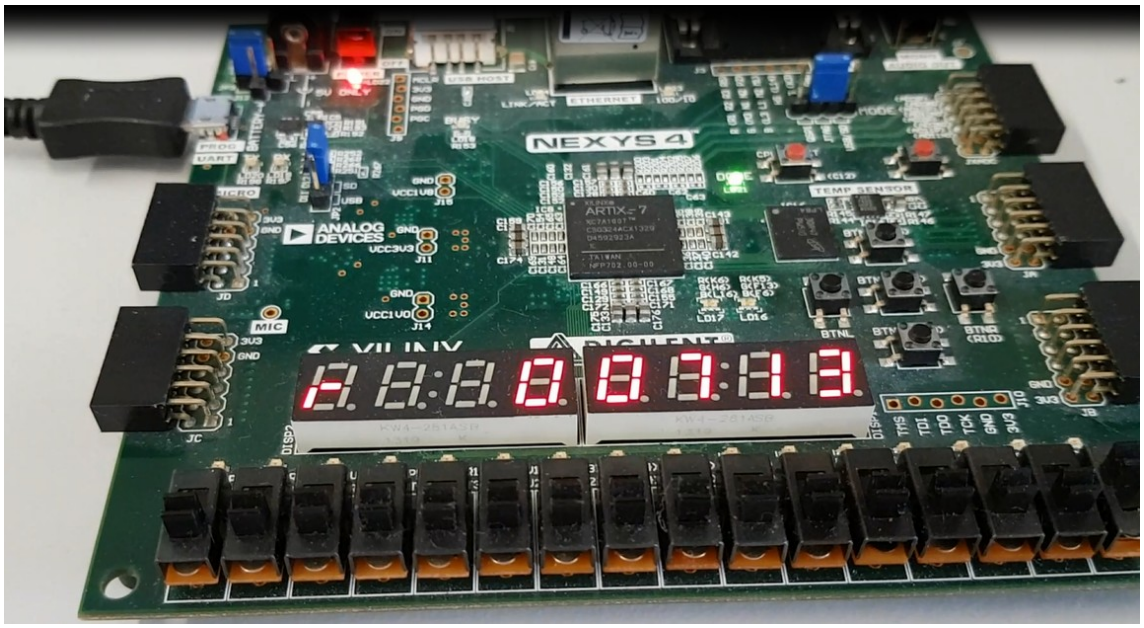

Abbildung 4.: Testhardware Board Nexys 4 Master (eigene Aufnahme)

Die Konfiguration des FPGAs wird über einen Rechner durchgeführt, der über den USB Anschluss oben links mit dem Board verbunden wird. Des weiteren wird ein Frontend für die Eingabe eingebunden. Dieses besteht aus 16 Schaltern, mit denen Eingabegrößen als Binärzahlen festgelegt werden. Des weiteren enthält das Board als Backend eine achtstellige Siebensegmentanzeige, in welcher die Eingaben und Berechnungsergebnisse als Dezimalzahl im Ganzzahlformat, d. h. $x \cdot 2^{-14}$, dargestellt werden. Dazu werden die letzten fünf Stellen verwendet, wobei ggf. eine weitere Stelle für die Angabe des negativen Vorzeichens verwendet wird. Die erste Stelle dient der Spezifizierung der Art des Wertes, bspw. zeigt das „r“ in Abbildung 4. an, dass es sich um das Resultat der Berechnung handelt. Gesteuert wird das Board über die fünf Push-but-

tons, die sich in einer kreuzförmigen Anordnung rechts oberhalb der Siebensegmentanzeige befinden.

Für die Durchführung der Divisionsberechnung wird das Board erst über den Reset-Button zurückgesetzt. Anschließend kann der Wert für x, d. h. der Divisor des zu berechnenden Quotienten eingegeben werden. Nachdem die Eingabe mit dem Enter-Buton bestätigt wurde, kann der y-Wert, d. h. der Divvend eingegeben werden. Nach dessen Bestätigung wird die Berechnung durchgeführt und das Ergebnis angezeigt, Durchgeführt wurden die Berechnungen;

- y / x = 1 / 23 = 712,3… $\cdot 2^{-14}$ Ergebnis; $713 \cdot 2^{-14}$
- y / x = 213 / -716 = -4874,0…$\cdot 2^{-14}$ Ergebnis: $4875 \cdot 2^{-14}$

Die Berechnungsergebnisse sind somit im Rahmen der zu erwartenden Genauigkeit. Diese Berechnung wurde mit einer Taktfrequenz von 50 MHz durchgeführt, die bereits durch die Simulation als größtmögliche geeignete Frequenz bestimmt wurde. Auch im Hardwaretest führt eine höhere Frequenz (hier 100 MHz) zu einer inkorrekten Berechnung.

Ähnlich gestalten sich die Durchführung und die Ergebnisse der Hardwaretests für die Akussinusberechnung. Im Unterschied zur Divisionsberechnung wird nur ein Eingabewert benötigt. Bei mittleren Werten für diese Argumente ergeben sich ebenfalls geringe Abweichungen, welche im Rahmen der zu erwartenden Genauigkeit liegen. Die Berechnung für den Arkussinus wurde bei einer Taktfrequenz von 25 MHz durchgeführt. Des Weiteren wurde ihre korrekte Funktionsweise auch bei einer Frequenz von 40 MHz verifiziert.

Somit bestätigen die Ergebnisse der Hardwaretests die Resultate der Simulationen und belegen die prinzipielle Richtigkeit der beiden entworfenen Schaltungen und ihre Eignung für die Durchführung der geplanten Berechnungen.

5 Zusammenfassung

Die Evaluation mathematischer Funktionen ist ein häufig auftretendes Problem in der Verarbeitung von Messsignalen. Hierzu bieten sich mehrere Möglichkeiten an, die sich wesentlich in Bezug auf den Hardwareaufwand, die Verarbeitungszeit und die Genauigkeit unterscheiden. In dieser Arbeit, die ein Teil des POLDI-Projektes zur Entwicklung eines kompakten optischen Winkelsensors darstellt, wird mithilfe eines FPGA die Berechnung von Divisionen und des Arkussinus realisiert. Aufgrund der Rahmenbedingungen des Projektes, die einen möglichst geringen Hardwareaufwand vorgeben, wird die Funktionsauswertung mithilfe des CORDIC-Algorithmus implementiert.

Der Voteil des CORDIC-Algorithmus in diesem Zusammenhang ist, dass die Funktionsevaluation auf einfache Iterationen zurückgeführt werden, die ausschließlich Additions- , Subtraktions- und Verschiebeoperationen sowie eine Lookuptable begrenzten Umfangs beinhalten. Die Vermeidung von Multiplikationen ermöglicht die Umsetzung der Aufgabe mit geringem Hardwareaufwand.

Ausgehend von den grundlegenden Iterationsvorschriften der CORDIC-Algortihmen zur Vectoring-Berechnung im Linearmodus und im Zirkularmodus werden in VHDL zwei Module zur Berechnung der Division und des Arkussinus entworfen. Eine wesentliche Eigenschaft des Entwurfs ist seine Anpassbarkeit, die dadurch erreicht wird, dass die einzelnen Iterationsschritte über generisch erzeugte Submodule realisiert werden. Die Anzahl der Iterationsschritte ist theoretisch beliebig erweiterbar, wobei aber nach 15 Schritten die maximale Genauigkeit erreicht wird, die innerhalb der vorgegebenen internen Signallänge von 16 bit darstellbar ist. Des weiteren wird bei dem Entwurf eine Pipeline-Struktur umgesetzt, in der mit jedem Arbeitstakt eine neue Funktionsevaluation gestartet werden kann. Auf diese Weise wird ein größtmöglicher Durchsatz bei der kontinuierlichen Verarbeitung von Messwerten erreicht.

Der VHDL-Entwurf ist mithilfe der Entwurfssoftware ViVado simuliert als auch auf einem Testboard implementiert und getestet worden. Dadurch wurde die fehlerfreie Funktion des Entwurfs verifiziert. Die Tests zeigen, dass sich die Divisionsberechnung mit einer Taktfrequenz von 50 MHz realisieren lässt, während für die Evaluation des Arkussinus 40 MHz die Obergrenze darstellt. Die Genauigkeit der Berechnungen liegen im Rahmen des Rundungsfehlers durch die Beschränkung des Zahlenformats. Zu beachten ist allerdings, dass die Berechnungen jeweils nur bis zu einer definierten Obergrenze durchführbar sind. Bei der Division können nur Quotienten, deren Betrag sich auf weniger als $32767 \cdot 2^{-14} \approx 1{,}999938..$ beläuft, berechnet werden. Die Berechnung

des Arkussinus ist limitiert auf Werte, deren Betrag kleiner als 16326·2-14≈0.9964… ist. Diese Begrenzungen lassen sich möglicherweise durch eine Vorverarbeitung der Messwerte umgehen.

Insgesamt belegen die Ergebnisse dieser Arbeit, dass der CORDIC-Algorithmus geeignet ist, die Berechnungen für den POLDI-Sensor mit geringem Aufwand und guter Genauigkeit umzusetzen. Er bietet sich daher dafür an, auch andere Funktionsevaluationen, die in diesem Kontext erforderlich sind, umzusetzen.

Literaturverzeichnis

[1] E. Hering und G. Schönfelder, „Geometrische Größen“, in *Sensoren in Wissenschaft und Technik*, E. Hering und G. Schönfelder, Hrsg. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 127–327.

[2] AdviCo, „POLDI – The Optical Micro-Angle-Sensor“, 2020. http://www.advico.de/index.php/26-home/slider/19-slide-poldi (zugegriffen Juli 22, 2020).

[3] C. Demske, „Entwurf eines VHDL-Designs und einer Applikationssoftware zur Konfiguration und Kalibrierung eines optischen Winkelgebers“, Bachelorthesis, Fachhochschule Dortmund, 2019.

[4] M. Beuler, „CORDIC-Algorithmus zur Auswertung elementarer Funktionen in Hardware“, Fachhochschule Gießen-Friedberg, FH-Report 11/08, 2008. Zugegriffen: Apr. 29, 2020. [Online]. Verfügbar unter: http://digdok.bib.thm.de/volltexte/2009/4148/pdf/CORDIC_Algorithmus.pdf.

[5] J. E. Volder, „The CORDIC Trigonometric Computing Technique“, *IRE Trans. Electron. Comput.*, Bd. EC-8, Nr. 3, S. 330–334, Sep. 1959, doi: 10.1109/TEC.1959.5222693.

[6] J. S. Walther, „A unified algorithm for elementary functions“, in *Proceedings of the May 18-20, 1971, spring joint computer conference*, Atlantic City, New Jersey, Mai 1971, S. 379–385, doi: 10.1145/1478786.1478840.

[7] R. Weber, „CORDIC_Implementierung_in_einem_Fahrspurfuehrungssystem_auf_einer_Plattform_mit_gekoppelten_SoCs.pdf“, Bachelorarbeit, Hochschule für Angewandte Wissenschaften Hamburg, 2011.

[8] J. M. Muller, *Elementary functions: algorithms and implementation*, 2nd ed. Boston, MA: Birkhäuser, 2006.

[9] P. Pirsch, *Architekturen der digitalen Signalverarbeitung*. Stuttgart: Teubner, 1996.

[10] B. Bhattarai, „FPGA Implementation of CORDIC Processor“, George Washington University, Washington D:C:, Technical Report, Sep. 2013. doi: 10.13140/RG.2.1.4432.1364.

[11] F. Slomka, „VHDL-Einführung“, Universität Ulm - Institut für Eingebettete Systeme/Echtzeitsysteme, 2015. Zugegriffen: Mai 27, 2020. [Online]. Verfügbar unter: https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.050/vorlesungen/sose15/err/VHDL_Einfu%CC%88hrung.pdf.

[12] F. Dressler und S. Podlipnig, „Leistung und Pipelining. (Einführung in die Technische Informatik)“, Universität Innsbruck, 2012. Zugegriffen: Mai 27, 2020. [Online]. Verfügbar unter: https://docplayer.org/3171213-Leistung-und-pipelining-einfuehrung-in-die-technische-informatik-falko-dressler-stefan-podlipnig-universitaet-innsbruck.html.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

A. Theoretische Berechnungen mit Gleitkommazahlen

Tabelle A.1: Berechnungsschritte für arcsin(0,9) mit dem CORDIC-Algorithmus

i	2^{-i}	$\arctan(2^{-i})$	k_i	x_i	y_i	d_i	z_i	
0	1	0,785398	0,707107	0,607253	0,000000	1	0,000000	0,0000°
1	0,5	0,463648	0,894427	0,607253	0,607253	1	0,785398	45,0000°
2	0,25	0,244979	0,970143	0,303627	0,910880	-1	1,249046	71,5651°
3	0,125	0,124355	0,992278	0,531346	0,834973	1	1,004067	57,5288°
4	0,0625	0,062419	0,998053	0,426975	0,901391	-1	1,128422	64,6538°
5	0,03125	0,031240	0,999512	0,483312	0,874705	1	1,066003	61,0775°
6	0,015625	0,015624	0,999878	0,455977	0,889809	1	1,097243	62,8674°
7	0,0078125	0,007812	0,999969	0,442074	0,896933	1	1,112867	63,7626°
8	0,00390625	0,003906	0,999992	0,435067	0,900387	-1	1,120679	64,2102°
9	0,001953125	0,001953	0,999998	0,438584	0,898688	1	1,116773	63,9864°
10	0,0009765625	0,000977	1,000000	0,436829	0,899544	1	1,118726	64,0983°

B. Theoretische Berechnungen mit 16-Bit-Fixkommazahlen

In den nachfolgenden Berechnungen bilden das verwendete Format einer vorzeichenbehafteten 16-Bit-Fixkommazahl. Abzüglich des Vorzeichenbits und der Vorkommastelle hat dieses Format 14 binäre Nachkommastellen. Daher werden alle Bruchteile, die kleiner als $1 \cdot 2^{-14}$ ersatzlos gestrichen, d. h. es wir abgerundet.

Tabelle A.2: Berechnung von 1/23 mit Tabellenkalkulation

i	x_i	y_i	z_i	d_i	2^{-i}	Abweichung	$z_i \cdot 2^{14}$
0	23	1	0,000000	-1	1	-100,00%	0
1	23	-22	1,000000	1	0,5	2200,00%	16384
2	23	-10,5	0,500000	1	0,25	1050,00%	8192
3	23	-4,75	0,250000	1	0,125	475,00%	4096
4	23	-1,875	0,125000	1	0,0625	187,50%	2048
5	23	-0,4375	0,062500	1	0,03125	43,75%	1024
6	23	0,28125	0,031250	-1	0,015625	-28,13%	512
7	23	-0,078125	0,046875	1	0,0078125	7,81%	768
8	23	0,1015625	0,039063	-1	0,00390625	-10,16%	640
9	23	0,01171875	0,042969	-1	0,001953125	-1,17%	704
10	23	-0,033203125	0,044922	1	0,0009765625	3,32%	736
11	23	-0,0107421875	0,043945	1	0,00048828125	1,07%	720
12	23	0,00048828125	0,043457	-1	0,000244140625	-0,05%	712
13	23	-0,005126953125	0,043701	1	0,0001220703125	0,51%	716
14	23	-0,0023193359375	0,043579	1	6,103515625E-05	0,23%	714
15	23	-0,00091552734375	0,043518	1	3,0517578125E-05	0,09%	713

Tabelle A.3 Berechnung von arcsin(0,5) mit Tabellenkalkulation

i	$x_i \cdot 2^{14}$	$y_i \cdot 2^{14}$	$z_i \cdot 2^{14}$	2^i	$\arctan(2^{-i})$	d_i
Startwerte	9952	0	0	-		1
0	9952	9952	12868	1	12868	-1
1	14928	4976	5272	2	7596	1
2	13684	8708	9286	4	4014	-1
3	14772	6998	7249	8	2037	1
4	14335	7921	8272	16	1023	1
5	14088	8368	8784	32	512	-1
6	14218	8148	8528	64	256	1
7	14155	8259	8656	128	128	-1
8	14187	8204	8592	256	64	-1
9	14203	8177	8560	512	32	1
10	14196	8190	8576	1024	16	1
11	14193	8196	8584	2048	8	-1
12	14195	8193	8580	4096	4	-1
13	14196	8192	8578	8192	2	1
14	14196	8192	8579	16384	1	1

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die von mir vorgelegte Prüfungsleistung selbst-ständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.

_____	_____
Ort, Datum	Unterschrift