

CHIPIMPLEMENTATION EINER  
ZWEIDIMENSIONALEN  
FOURIERTRANSFORMATION FÜR DIE  
AUSWERTUNG EINES SENSOR-ARRAYS

THOMAS LATTMANN

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung  
im Studiengang Informations- und Elektrotechnik  
am Department Informations- und Elektrotechnik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Karl-Ragmar Riemschneider  
Zweitgutachter: Prof. Dr.-Ing. Jürgen Vollmer

Abgegeben am 20.04.2018

---

[content/abstract.pdf](#)

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Stand der Technik . . . . .	1
1.3	Ziel dieser Arbeit . . . . .	1
<b>2</b>	<b>Grundlagen</b>	<b>2</b>
2.1	Binäre Zahlendarstellung von Festkommazahlen . . . . .	2
2.1.1	Integer-Zahl im 1er-Komplement . . . . .	2
2.1.2	Integer-Zahl im 2er-Komplement . . . . .	2
2.1.3	Darstellung dualer Zahlen im SQ-Format . . . . .	3
2.1.4	Numerisch bedingte Ungenauigkeiten . . . . .	4
2.2	Mathematische Grundlagen . . . . .	4
2.2.1	Komplexe Multiplikation . . . . .	4
2.2.2	Matrixmultiplikation . . . . .	4
2.3	Fourierreihenentwicklung . . . . .	5
2.4	Fouriertransformation . . . . .	6
2.4.1	Diskrete Fouriertransformation (DFT) . . . . .	7
2.4.2	Summen- und Matrizenschreibweise der DFT . . . . .	8
2.4.3	2D-DFT mit reellen Eingangswerten . . . . .	9
2.4.4	Berechnung der Diskreten Fouriertransformation mittels FFT . . . . .	11
2.4.5	Inverse DFT . . . . .	12
2.5	Diskrete Kosinus Transformation (DCT) . . . . .	12
2.5.1	Verwendung der DCT . . . . .	12
2.5.2	Berechnung der DCT . . . . .	12
<b>3</b>	<b>Analyse</b>	<b>13</b>
3.1	Bewertung verschiedener DFT- und DCT-Größen . . . . .	13
3.1.1	Bewertung verschiedener DCT-Größen . . . . .	14
3.1.2	Bewertung verschiedener DFT-Größen . . . . .	14
3.1.3	Bewertungsfazit . . . . .	14
3.2	Genauere Betrachtung der 8x8-DFT . . . . .	15
3.3	Einordnung des Rechenaufwands . . . . .	17
3.3.1	8x8-DFT mit komplexen Eingangswerten . . . . .	18
3.3.2	8x8-DFT mit reellen Eingangswerten . . . . .	18
3.3.3	Direkte Multiplikation zweier 8x8 Matrizen mit komplexen Werten . . . . .	19
3.3.4	Betrachtung des Butterfly-Algorithmus für 8 Eingangswerte . . . . .	19
3.3.5	Fazit der Berechnungs-Gegenüberstellungen . . . . .	20

<b>4</b>	<b>Entwurf</b>	<b>22</b>
4.1	Projekt- und Programmstruktur . . . . .	22
4.1.1	Vorüberlegungen aus Hardwaresicht . . . . .	22
4.1.2	Bibliotheken . . . . .	22
4.1.3	Struktureller Aufbau . . . . .	23
4.2	Entwicklungsstufen . . . . .	23
4.2.1	2D-DFT-Implementierung in VHDL . . . . .	23
4.2.2	Weiterentwicklung zur 2D-DFT . . . . .	24
4.2.3	Direkte Weiterverarbeitung der Zwischenergebnisse . . . . .	25
4.2.4	Optimierte 8x8 DFT als Matrixmultiplikation . . . . .	26
4.2.5	Berechnungsschema der geraden und ungeraden Zeilen . . . . .	27
4.2.6	Test der Matrixmultiplikation . . . . .	29
4.3	Synthesergebnisse von Teilkomponenten . . . . .	30
4.3.1	13 Bit Konstantenmultiplizierer . . . . .	30
4.3.2	Bildung des 2er-Komplements eines 13 Bit Vektors . . . . .	30
4.3.3	13 Bit Addierer . . . . .	31
4.3.4	Vergleich der Synthesergebnisse . . . . .	31
4.4	Schema der Zustandsfolge . . . . .	33
4.5	UML-Diagramm . . . . .	35
<b>5</b>	<b>Evaluation</b>	<b>38</b>
5.1	Simulation . . . . .	38
5.1.1	NC Sim - positive Zahlendarstellung . . . . .	38
5.2	Anzahl benötigter Takte . . . . .	38
5.3	Zeitabschätzung im Einsatz als ABS-Sensor . . . . .	38
5.4	Testumgebung . . . . .	41
5.4.1	Struktogramm des Testablaufs . . . . .	41
5.4.2	Reale Eingangswerte . . . . .	41
5.5	Chipdesign . . . . .	41
5.5.1	Anzahl Standardzellen . . . . .	41
5.5.2	Visualisierung der Netzliste . . . . .	41
5.5.3	Floorplan, Pading . . . . .	41
<b>6</b>	<b>Schlussfolgerungen</b>	<b>42</b>
6.1	Zusammenfassung . . . . .	42
6.2	Bewertung und Fazit . . . . .	42
6.3	Ausblick . . . . .	42
<b>7</b>	<b>Abkürzungsverzeichnis</b>	<b>43</b>
	<b>Abbildungsverzeichnis</b>	<b>44</b>
	<b>Tabellenverzeichnis</b>	<b>45</b>

---

<b>Literatur</b>	<b>46</b>
<b>8 Anhang</b>	<b>47</b>
8.1 Skript zur Bewertung von Twiddlefaktormatrizen . . . . .	47
8.2 Gate-Report des 12 Bit Konstantenmultiplizierers . . . . .	51
8.3 Twiddlefaktormatrix im S1Q10-Format . . . . .	51
8.4 Programmcode . . . . .	56
8.5 Testumgebung . . . . .	79

# 1 Einleitung

## 1.1 Motivation

Sensorarray beschreiben

## 1.2 Stand der Technik

Der verwendete Prozess ist mit  $350\text{ }\mu\text{m}$  im Vergleich zu modernen Prozessen mit beispielsweise  $20\text{ nm}$  Strukturbreite um die Größenordnung  $10^4$  größer. Entsprechend handelt es sich um einen relativ alten Prozess.

Kurze Beschreibung zu Standardzellen.

## 1.3 Ziel dieser Arbeit

Im Rahmen des Integrated Sensor Array (ISAR)-Projekts der HAW Hamburg soll zur Signalvorverarbeitung einer Matrix von Magnetsensoren eine zweidimensionale diskrete Fouriertransformation (2D-DFT) in VHDL implementiert werden. Mit der 2D-DFT sollen relevante Signalanteile identifiziert werden, um so den Informationsgehalt der Sensorsignale auf relevante Anteile zu reduzieren. Die Sensoren basieren auf dem anisotropen magnetoresistiven Effekt (AMR)- bzw. in einem späteren Schritt tunneltmagnetoresistiven Effekt (TMR).

In einem Text zitiert dann so [1, S. 10-20] und blabla.

Winkelberechnung als Ziel beschreiben.

## 2 Grundlagen

Um einen guten Ausgangspunkt für spätere Erläuterungen zu haben, sollen an dieser Stelle die wesentlichen Grundlagen zusammengefasst werden.

### 2.1 Binäre Zahlendarstellung von Festkommazahlen

Im Rahmen dieses Projekts wird von Ein- sowie Ausgangswerten mit einer Genauigkeit von 12 Bit ausgegangen. Basierend auf älteren Sensoren wird von Werten im Bereich von  $-2 < z < 2$  ausgegangen. Aus diesem Grund müssen sowohl ein Ganzzahlanteil, sowie Nachkommastellen repräsentiert werden können. Wie dies gelingt, wird in den nächsten Abschnitten gezeigt. Hierfür werden Festkommazahlen verwendet, aufgrund der Rechenoperationen haben diese dennoch unterschiedlich viele Vor- sowie Nachkommastellen.

#### 2.1.1 Integer-Zahl im 1er-Komplement

Bei der Interpretation des Bitvektors als Integerwert im Einerkomplement werden die Bits anhand ihrer Position im Bitvektor gewichtet, wobei das niederwertigste Bit (LSB, least significant bit) dem Wert für den Faktor  $2^0$  entspricht, das Bit links davon dem für  $2^1$  und so weiter. Die Summe aller Bits, ohne das höchstwertigste, multipliziert mit ihrer Wertigkeit (Potenz) ergibt den Betrag der Dezimalzahl. Das höchstwertigste Bit (MSB, most significant bit) gibt Auskunft darüber, ob es sich um eine negative oder positive Zahl handelt, wobei eine 0 für eine positive Zahl steht. Entsprechend besagt die 1, dass die Zahl negativ ist. Dies hat zur Folge, dass es eine positive und eine negative Null und somit eine Doppeldeutigkeit gibt. Des Weiteren wird ein LSB an Auflösung verschenkt. Der Wertebereich erstreckt sich von  $-2^{\text{MSB}-1} + 1 \text{ LSB}$  bis  $2^{\text{MSB}-1} - 1 \text{ LSB}$ .

Diese Darstellung hat den Vorteil, dass sich das Ergebnis einer Multiplikation der Zahlen  $a \cdot b$  und  $-a \cdot b$  nur im vordersten Bit unterscheidet. Darüber hinaus lässt sich das Vorzeichen des Ergebnisses durch eine einfache XOR-Verknüpfung der beiden MSB der Multiplikanden ermitteln. Die eigentliche Multiplikation beschränkt sich auf die Bits MSB-1 bis LSB.

Nachteile zeigen sich hingegen bei der Addition sowie Subtraktion negativer Zahlen. Auch hierfür gibt es schematische Rechenregeln, diese erfordern jedoch mehr Zwischenschritte als im Zweierkomplement.

#### 2.1.2 Integer-Zahl im 2er-Komplement

Bei der Interpretation als Zweierkomplement kann anhand des MSB ebenfalls erkannt werden, ob es sich um eine positive oder negative Zahl handelt. Hier bedeutet ein gesetztes MSB

$-2^{MSB-1}$ , was der negativsten darstellbaren Zahl entspricht. Hierbei sind alle anderen Bits auf 0. Für gesetzte Bits wird der Dezimalwert, wie beim Einerkomplement beschrieben, berechnet und auf den negativen Wert aufaddiert. Wenn das MSB nicht gesetzt ist, wird der errechnete Dezimalwert auf 0 addiert. Auf diese Weise lassen sich Zahlen im Wertebereich von  $-2^{MSB-1}$  bis  $2^{MSB-1} - 1$  LSB darstellen. Der positive Wertebereich ist also um ein LSB kleiner als der negative und es gibt keine doppelte Null.

Um das Vorzeichen umzukehren müssen alle Bits invertiert werden. Auf das Resultat muss abschließend 1 LSB addiert werden.

Vorteil bei dieser Darstellung ist, dass die mathematischen Operationen Addition, Subtraktion und Multiplikation direkt angewandt werden können. Unterstützt werden sie z.B. von den Datentypen `unsigned` sowie `signed`, welche in der Bibliothek u.a. `ieee.numeric_std.all` definiert sind.

### 2.1.3 Darstellung dualer Zahlen im SQ-Format

Im SQ-Format werden Zahlen als vorzeichenbehafteter Quotient (signed quotient) dargestellt. Wie beim 2er-Komplement entscheidet das höchstwertigste Bit, ob es sich um eine positive oder negative Zahl handelt. In Abbildung 2.1 ist exemplarisch die Interpretation von Dualzahlen im SQ3-Format, also für vier Bit, zu sehen.

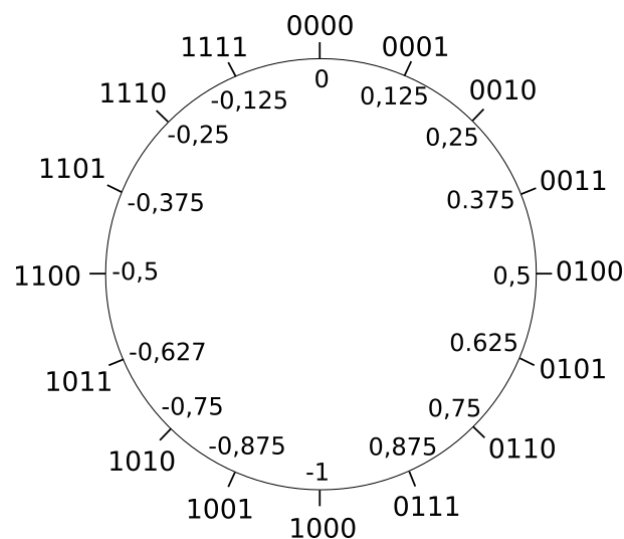


Abbildung 2.1: Interpretation von Dualzahlen im SQ3-Format

Der darstellbare Zahlenbereich liegt hier bei  $-1 \leq z < 1$ . Benötigt werden Zahlen im Bereich von etwa  $\pm 2$ , weshalb ein Vorkommabit benötigt wird. Da 12 Bit zur Verfügung stehen, von denen eins für das Vorzeichen und ein weiteres für eine Vorkommastelle verwendet werden, bleiben 10 Bits für die Nachkommazahlen übrig. Die Aufteilung der Bits wird über die Bezeichnung S1Q10 definiert. Da für den Quotient 10 Bit zur Verfügung stehen, beträgt die maximale Auflösung  $1 \text{ LSB} = 2^{-10} = 1024^{-1} = 9,765625 \cdot 10^{-4}$ . Der Wertebereich liegt in diesem Fall liegt bei  $-2$  bis  $1,999\,023\,438$ .



Für die Addition oder Multiplikation zweier Zahlen müssen beide einerseits dieselbe Bitbreite und andererseits das gleiche Darstellungsformat besitzen.

### 2.1.4 Numerisch bedingte Ungenauigkeiten

Numerische Ungenauigkeiten entstehen immer dann, wenn die zur Verfügung stehenden Bits es nicht ermöglichen eine Zahl exakt abzubilden. Bei einem Bitshift, welcher häufig für die Division durch Zwei oder Vielfachen von Zwei verwendet wird, kann immer Information verloren gehen. Dies ist immer dann der Fall, wenn die Bits die abgeschnitten werden eine 1 sind. Das hat zur Folge, dass beispielsweise bei einer Division durch Zwei der resultierende Wert um 1 LSB kleiner ist, als er eigentlich sein sollte. Dieses Problem kann bei jedem Bitshift auftreten. Die Wahrscheinlichkeit für eine 1 liegt im Mittel bei 50 %, weshalb davon ausgegangen werden muss, dass ein positives Ergebnis etwas kleiner und ein negatives vom Betrag her etwas größer ist, als bei verlustfreier Berechnung.

Da diese Arbeit den Schwerpunkt in der Aufwandsabschätzung einer Chipimplementation einer 2D-DFT auf einem Application Specific Integrated Circuit, *dt.: Anwendungsspezifischer Integrierter Schaltkreis* (ASIC) hat, ist diese Problematik kein Gegenstand dieser Arbeit und wird an dieser Stelle nur in Grundzügen erwähnt.

## 2.2 Mathematische Grundlagen

Zu den mathematischen Grundlagen werden die komplexe Multiplikation sowie die Matrixmultiplikation gezählt, welche nachfolgen kurz behandelt werden. Auf die Fourierreihenentwicklung sowie insbesondere die Fouriertransformation und ihre diskrete Variante wird im Anschluss detaillierter eingegangen, da sie elementarer Bestandteil dieser Arbeit sind. Da auch die diskrete Kosinustransformation als mögliche Transformationsart im Raum stand, um in den Bildbereich zu gelangen, wird diese ebenfalls kurz aufgegriffen.

### 2.2.1 Komplexe Multiplikation

Im allgemeinen Fall müssen gemäß Gl. (2.1) bei der komplexen Multiplikation vier einfache Multiplikation sowie zwei Additionen durchgeführt werden.

$$\begin{aligned} e + jf &= (a + jb) \cdot (c + jd) \\ &= a \cdot c + j(a \cdot d) + j(b \cdot c) + j^2(b \cdot d) \\ &= a \cdot c - b \cdot d + j(a \cdot d + b \cdot c) \end{aligned} \tag{2.1}$$

### 2.2.2 Matrixmultiplikation

Um nachfolgende Abschnitte besser erörtern zu können, soll zunächst die Matrixmultiplikation besprochen werden. Wie in Abbildung 2.2 verdeutlicht, wird  $\text{Element}(i, j)$  der Ergebnismatrix dadurch berechnet, dass die  $\text{Elemente}(i, k)$  einer Zeile der 1. Matrix mit den  $\text{Elementn}(k, j)$

aus der zweiten Matrix multipliziert und die Werte aufsummiert werden.  $i$  und  $j$  sind für die Berechnung eines Elements konstant, während  $k$  über alle Elemente einer Zeile bzw. Spalte läuft.

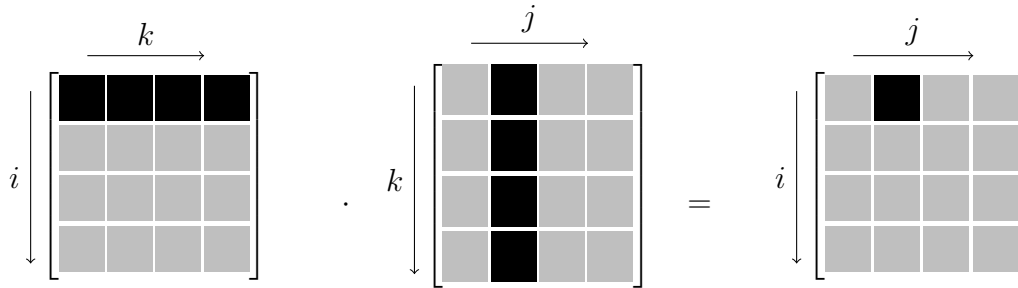


Abbildung 2.2: Veranschaulichung der Matrixmultiplikation

## 2.3 Fourierreihenentwicklung

Mit einer Fourierreihe kann ein periodisches Signal aus einer Summe von Sinus- und Kosinusfunktionen zusammengesetzt werden. Die Schreibweise als Summe von Sinus- und Kosinusfunktionen (Gl. 2.2) ist eine der häufigsten Darstellungsformen.

$$x(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} (a_k \cos(kt) + b_k \sin(kt)) \quad (2.2)$$

Die Fourierkoeffizienten lassen sich über die Gleichungen (2.3) und (2.4) berechnen:

$$a_k = \frac{1}{\pi} \int_{-\pi}^{\pi} x(t) \cdot \cos(kt) dt \quad \text{für } k \geq 0 \quad (2.3)$$

$$b_k = \frac{1}{\pi} \int_{-\pi}^{\pi} x(t) \cdot \sin(kt) dt \quad \text{für } k \geq 1 \quad (2.4)$$

Mit der Exponentialschreibweise lassen sich Sinus und Kosinus auch wie in (2.5) und (2.6) ausdrücken:

$$\cos(kt) = \frac{1}{2} (e^{jkt} + e^{-jkt}) \quad (2.5)$$

$$\sin(kt) = \frac{1}{2j} (e^{jkt} - e^{-jkt}) \quad (2.6)$$

und zusammengefasst ergibt sich in (Gl. 2.7) der komplexe Zeiger, der eine Rotation im Gegenuhrzeigersinn auf dem Einheitskreis beschreibt. In Abbildung 2.3 wird dies grafisch dargestellt.

$$\begin{aligned}
 \cos(kt) + j \cdot \sin(kt) &= \frac{1}{2} (e^{jkt} + e^{-jkt}) + j \cdot \frac{1}{2j} (e^{jkt} - e^{-jkt}) \\
 &= \frac{1}{2} (e^{jkt} + e^{jkt}) \\
 &= e^{jkt}
 \end{aligned} \tag{2.7}$$

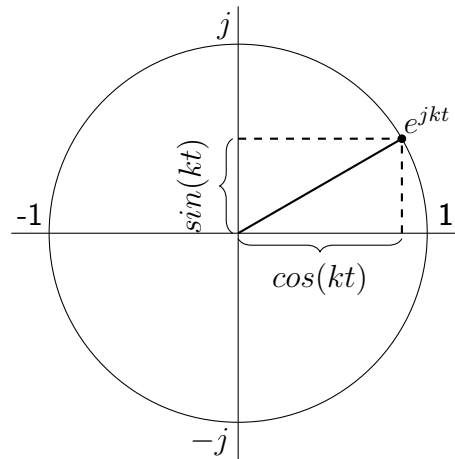


Abbildung 2.3: Einheitskreis, Zusammensetzung des komplexen Zeigers aus Sinus und Kosinus

Die Fourierkoeffizienten  $a_k$  und  $b_k$  lassen sich auch als komplexe Zahl  $c_k$  zusammengefasst berechnen:

$$c_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} x(t) e^{-j2\pi kt} dt \quad \forall k \in \mathbb{Z} \tag{2.8}$$

$$x(t) = \sum_{-\infty}^{\infty} c_k e^{jkt} \tag{2.9}$$

## 2.4 Fouriertransformation

Mit der Fouriertransformation kann umgekehrt ein periodisches Signal  $x(t)$  in eine Summe aus Sinus- und Kosinusfunktionen unterschiedlicher Frequenzen zerlegt werden. Da diese Funktionen jeweils mit nur einer Frequenz periodisch sind, entsprechen diese Frequenzen den Frequenzbestandteilen von  $x(t)$ .

Grundlage für die Fouriertransformation ist das Fourierintegral (Gl. 2.10)

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-j2\pi ft} dt \tag{2.10}$$

Wenn Sinus- und Kosinusfunktionen wie in Gl. (2.5) und (2.6) als Exponentialfunktion

geschrieben werden, können sie zu einer komplexen Exponentialfunktion zusammengefasst werden. Hieraus lässt sich ableiten, dass das Spektrum, also  $X(f)$  komplexwertig sein muss.

Signalformen wie etwa ein Rechteck haben entsprechend sehr viele dieser Frequenzbeiträge. Deren Höhe ist Information darüber, wie groß ihr Anteil, also die Amplitude des Zeitsignals, ist. Die Fouriertransformation kann als das Gegenteil der Fourierreihenentwicklung gesehen werden, mit ihr erhält man das Spektrum eines Zeitsignals. Eine Vertiefung dieses umfangreichen Gebiets der Fourier-Analyse findet sich u.a. in ...

In der vorliegenden Arbeit wird künftig  $X^*$  für die eindimensionale diskrete Fouriertransformation (1D-DFT) und  $X$  für die zweidimensionale diskrete Fouriertransformation (2D-DFT) stehen.

### 2.4.1 Diskrete Fouriertransformation (DFT)

Die diskrete Fouriertransformation (DFT) ist die zeit- und wertdiskrete Variante der Fouriertransformation, die statt von  $-\infty$  bis  $\infty$  über einen Vektor von  $N$  Werten, also von 0 bis  $N-1$  läuft. Dies hat zur Folge, dass sich ihr Frequenzspektrum periodisch nach  $N$  Werten wiederholt.

Da es sich um eine endliche Anzahl diskreter Werte handelt, geht das Integral aus Gleichung (2.10) in die Summe aus Gleichung (2.11) über.

Üblicherweise wird die (diskrete) Fouriertransformation genutzt, um vom Zeitbereich in den Frequenzbereich zu gelangen. In diesem Fall enthielte der Eingangsvektor Werte im Zeitbereich, der Ausgangsvektor Werte im Frequenzbereich. Um von Daten im Zeitbereich sprechen zu können, müssen diese zeitlich versetzt auf den gleichen Bezugspunkt erfasst worden sein. Bezogen auf das Sensorarray würde eine bestimmte Anzahl an zeitlich versetzten zeit- und wertdiskretisierten Daten eines einzelnen Sensors in einem Vektor zusammengefasst und darauf die DFT angewandt werden, um beim Ausgangsvektor von Daten im Frequenzbereich sprechen zu können.

Statt zeitlich versetzter Daten werden beim Sensorarray die Daten von mehreren Sensoren gleichzeitig erfasst. Da das Sensorarray zweidimensional ist, ergibt sich an Stelle eines Vektors eine Matrix. Weil die Werte gleichzeitig erfasst werden und diese verschiedene Koordinaten repräsentieren, muss hier von Orts- anstatt von Zeitwerten gesprochen werden. Von der Transformation ins Frequenzspektrum spricht man wiederum bei Zeitwerten, da das Spektrum die Frequenzen darstellt, aus denen das Zeitsignal zusammengesetzt ist. Da bei der eben beschriebenen Datenerfassung Ortsdaten transformiert werden, spricht man hier allgemeiner von einer Transformation in den Bildbereich.

In dieser Arbeit werden statt Zeit- bzw. Ortsbereich respektive Frequenzbereich und Bildverarbeitung häufig auch die Begriffe Ein- und Ausgangsvektor bzw. -matrix verwendet.

Mit der eindimensionalen diskreten Fouriertransformation (1D-DFT) wird die spaltenweise DFT einer Matrix bezeichnet, in der Regel ist sie der erste Schritt der Berechnung der 2D-DFT. Die Größe der Eingangsmatrix gibt die Größe der Twiddlefaktormatrix vor, beide müssen identisch und quadratisch sein. In dieser Arbeit wird die DFT einer Matrix der Größe  $N \times N$  auch  $N \times N$ -DFT genannt.

## 2.4.2 Summen- und Matrizenschreibweise der DFT

### 1D-DFT

Die DFT findet wie bereits erwähnt üblicherweise Anwendung, um vom Zeit- in den Frequenzbereich zu gelangen.

$$X^*[m] = \frac{1}{N} \cdot \sum_{n=0}^{N-1} x[n] \cdot e^{-\frac{j2\pi mn}{N}} \quad (2.11)$$

In Gleichung (2.11) ist die übliche Verwendung von Eingangsvektor  $x[n]$  und Ausgangsvektor  $X[n]$  zu sehen. Eine spaltenweise Multiplikationen einer Matrix ist auch denkbar und ist darüber hinaus Grundlage für die 2D-DFT. Gleichung (2.13) zeigt die Summenformel aus (2.11), umgeschrieben zu einer Matrixmultiplikation.

Mit Gleichung (2.12) werden zunächst alle Twiddlefaktoren in Matrixform berechnet, wobei  $n$  der Index des zu berechnenden Elements des Vektors im Zeitbereich und  $m$  das Äquivalent im Frequenzbereich ist.

$$W = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} e^{-\frac{j2\pi mn}{N}} \quad (2.12)$$

Somit gilt:

$$X^* = W \cdot x \quad (2.13)$$

In Matlab kann die Twiddlefaktormatrix mit

$$W = e^{-\frac{j2\pi}{N} \cdot [0:N-1]' \cdot [0:N-1]} \quad (2.14)$$

berechnet werden, wobei  $N$  die Anzahl der Elemente je Zeile bzw. Spalte ist.

Anhand der beiden Summen die jeweils von 0 bis  $N-1$  laufen, lassen sich die Anzahl der benötigten komplexen Multiplikationen  $m_{DFT}$  einer DFT errechnen. Siehe auch Gleichung (2.15).

$$m_{DFT} = N^2 \quad (2.15)$$

### 2D-DFT

Die 2D-DFT wird hingegen häufig in der Bildverarbeitung verwendet, um vom Orts- in den Fourierraum zu gelangen. Da es sich somit nicht mehr um eine Abhängigkeit der Zeit handelt, werden andere Indizes verwendet.

$$\begin{aligned} X[u, v] &= \frac{1}{N} \sum_{n=0}^{N-1} X^*[m] \cdot e^{-\frac{j2\pi mn}{N}} \\ &= \frac{1}{MN} \sum_{m=0}^{M-1} \left( \sum_{n=0}^{N-1} f(m, n) \cdot e^{-\frac{j2\pi mn}{N}} \right) \cdot e^{-\frac{j2\pi mn}{M}} \end{aligned} \quad (2.16)$$

Auch hier lässt sich die Berechnung in Matrizenschreibweise darstellen:

$$\begin{aligned} X &= W \cdot x \cdot W \\ &= X^* \cdot W \end{aligned} \quad (2.17)$$

Die Gleichungen (2.13) und (2.17) werden wesentlicher Bestandteil der Umsetzung der 2D-DFT sein.

Wie in Gleichung (2.17) beschrieben, kann die 2D-DFT als “doppelte” Matrizenmultiplikation geschrieben werden. Es wird also erst die 1D-DFT berechnet und die sich daraus ergebende Matrix  $X^*$  (Abb. 2.18) wird anschließend mit der Twiddlefaktor-Matrix  $W$  multipliziert. Man könnte es auch als zweite 1D-DFT betrachten, bei der Twiddlefaktor-Matrix und Eingangsmatrix vertauscht sind.

Veranschaulicht wird dies in den Abbildungen 2.18 und 2.19.

$$\begin{bmatrix} W \\ \cdot \\ x \end{bmatrix} = \begin{bmatrix} X^* \end{bmatrix} \quad (2.18)$$

$$\begin{bmatrix} X^* \\ \cdot \\ W \end{bmatrix} = \begin{bmatrix} X \end{bmatrix} \quad (2.19)$$

### 2.4.3 2D-DFT mit reellen Eingangswerten

Bei der oben beschriebenen Berechnung können die Eingangssignale auch komplex sein. Da das Ausgangssignal der 1D-DFT unabhängig von den Eingangssignalen in jedem Fall komplex ist, kann es dort direkt als Eingangssignal für die komplexe 2D-DFT genutzt werden.

Es wäre jedoch auch möglich, das komplexe Ausgangssignal der 1D-DFT als zwei von einander unabhängige rein reelle Eingangssignale der 2D-DFTs zu betrachten und später wieder zusammenzusetzen. Gleiches gilt dann natürlich auch für ein komplexes Eingangssignal, welches ebenfalls in zwei von einander unabhängigen DFTs transformiert werden kann. Da bei dieser Umsetzung kein Imaginärteil in die Berechnung der Ergebnisse einfließt, hat sie den Vorteil, dass aus Symmetriegründen die Hälfte der Multiplikationen eingespart werden können. Hierbei ist es erforderlich, dass der Imaginärteil der gespiegelten Ergebnisse negiert wird. Abbildung 2.5 zeigt die redundanten Werte der DFT. Es müssen bei der 8x8-DFT also statt 16 nur 8 Multiplikationen mit reellem Multiplikand und komplexen Multiplikator erfolgen.

Wie bereits beschrieben lässt sich dieses Verfahren auch für komplexe Eingangssignale, deren Real- und Imaginärteil separat voneinander mit der DFT transformiert werden, anwenden. Anschließend müssen die Ergebnisse zusammengesetzt werden. Wie dies geschieht ist der Abbildung 2.4 zu entnehmen. Die Abbildung stellt die schematische Berechnung der 2D-DFT eines reellen Eingangssignals dar. Um die 2D-DFT eines komplexen Eingangssignals zu berechnen, muss entweder eine identische Einheit für den Imaginärteil vorhanden sein oder Real- und Imaginärteil müssten zeitlich versetzt berechnet werden. Die Ergebnisse beider 2D-DFTs müssen identisch zusammengefasst werden, wie es zum Abschluss der einzelnen 2D-DFTs geschehen muss.

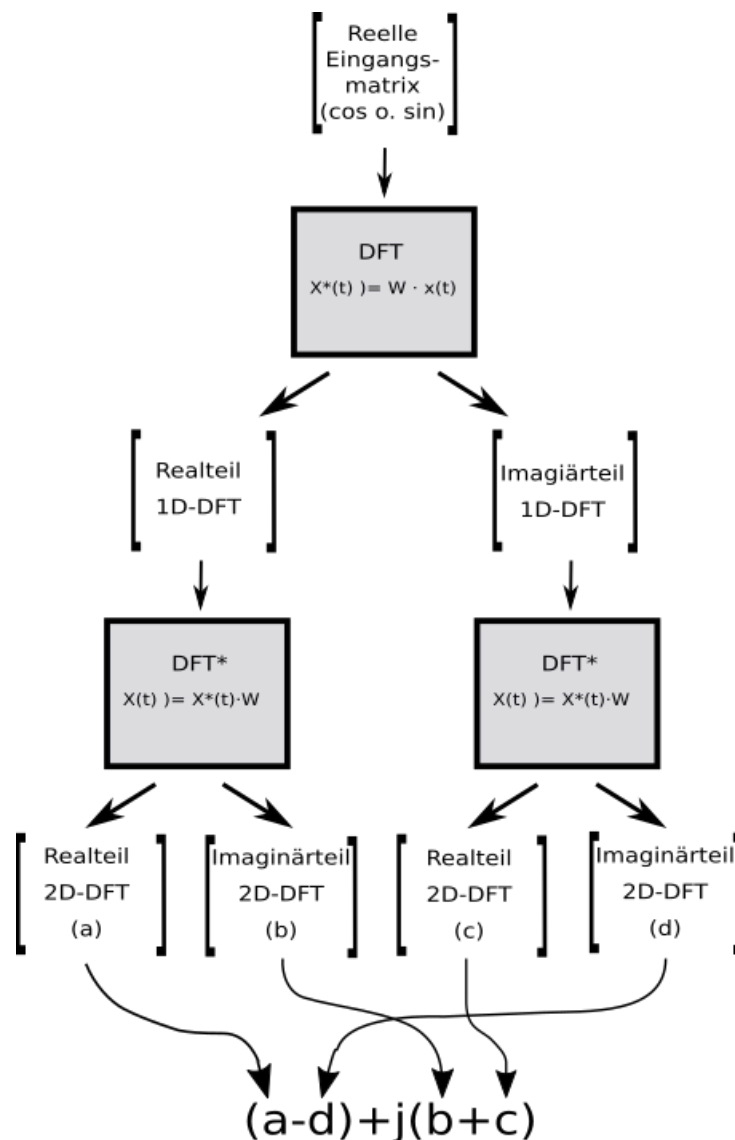


Abbildung 2.4: Veranschaulichung der Berechnung der DFT mit reellen Eingangswerten

Da die gegebenen Eingangssignale aus einer Sinus- und einer Kosinuskomponente bestehen und es sich auf diese Weise als ein komplexes Signal auffassen lässt, kann die komplexe Be-

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5								
6	A4	B4	C4	D4	E4	F4	G4	H4
7	A3	B3	C3	D3	E3	F3	G3	H3
8	A2	B2	C2	D2	E2	F2	G2	H2

(a) Realteil

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5								
6	-A4	-B4	-C4	-D4	-E4	-F4	-G4	-H4
7	-A3	-B3	-C3	-D3	-E3	-F3	-G3	-H3
8	-A2	-B2	-C2	-D2	-E2	-F2	-G2	-H2

(b) negierter Imaginärteil

Abbildung 2.5: Redundante Werte der spaltenweisen DFT einer 8x8-Matrix. Der Imaginärteil der redundanten Werte hat den selben Betrag mit negiertem Vorzeichen.

rechnung sowohl bei der 1D-DFT als auch bei der 2D-DFT genutzt werden. Da hierdurch in beiden Fällen eine vollständige Auslastung einer komplexen Berechnung gegeben ist und wie bereits erwähnt, bei der reellen Berechnung zusätzlicher Speicher erforderlich wäre, wird dieses Verfahren angewandt.

#### 2.4.4 Berechnung der Diskreten Fouriertransformation mittels FFT

Die Mathematiker Cooley und Tukey haben einen Algorithmus entwickelt und im Jahr 1965 veröffentlicht, mit dem sich die DFT mit vergleichsweise wenig Multiplikationen und somit deutlich schneller als bei der allgemeinen DFT berechnen lässt. Das Verfahren wird als Fast Fouriertransformation (FFT) bezeichnet. Grundlage ist, dass sich eine DFT in kleinere Teil-DFTs aufspalten lässt, welche durch Ausnutzen von Symmetrieeigenschaften in der Summe weniger Koeffizienten haben. Üblich ist die Radix-2 FFT, Ausgangspunkt ist also eine DFT mit 2 Eingangswerten. Da mit jeder weiteren Teil-DFT sich die Anzahl der Eingangswerte verdoppelt, eignet sich diese Methode nur für Eingangsvektoren der Größe  $2^n$ . Dieser vermeintliche Nachteil lässt sich durch Auffüllen des Eingangsvektors mit Nullen (Zeropadding) eliminieren. Dies hat zur Folge, dass die Größe des Ausgangsvektors immer eine Potenz von Zwei ist. Die Anzahl der benötigten komplexen Multiplikationen  $m_{FFT}$  kann mit der Gleichung (2.20) abgeschätzt werden.

$$m_{FFT} = \frac{N}{2} \log_2(N) \quad (2.20)$$



### 2.4.5 Inverse DFT

Die inverse diskrete Fouriertransformation (IDFT) ist die Umkehrfunktion der DFT. Wenn das Eingangssignal  $x[n]$  zeitabhängig und somit als  $\vec{x}(t)$  geschrieben werden kann, dann handelt es sich bei  $X^*[m]$  um dessen Darstellung im Frequenzbereich und kann als  $\vec{X}^*(f)$  geschrieben werden. Mit der IDFT ist es möglich, aus der Frequenzdarstellung das Zeitsignal zu errechnen.

$$x[n] = \frac{1}{N} \sum_{m=0}^{N-1} X^*[m] \cdot e^{\frac{j2\pi mn}{N}} \quad (2.21)$$

Gleichung (2.21) ist bis auf die Drehrichtung des komplexen Zeigers und die Vertauschten Ein- und Ausgangsvektoren identisch zu Gleichung (2.11).

## 2.5 Diskrete Kosinus Transformation (DCT)

### 2.5.1 Verwendung der DCT

Die DCT findet häufig in der Bildverarbeitung Anwendung,

### 2.5.2 Berechnung der DCT

Für die Berechnung der DCT gibt es verschiedene Varianten, welche sich in der Symmetrie der Ergebnismatrix unterscheiden. (Stimmt das wirklich? was sonst?)

Darüber hinaus wird in der Bildverarbeitung häufig die erste Zeile der Twiddlefaktormatrix mit dem Faktor  $\frac{1}{\sqrt{2}}$ , sowie die gesamte Matrix mit  $\sqrt{\frac{2}{N}}$ ,  $N$  = Anzahl Elemente in einer Zeile bzw. Spalte multipliziert.

Da es hier um eine Aufwandsabschätzung geht, wird sich auf die in der Bildverarbeitung gängigste Variante jedoch ohne die skalierenden Faktoren beschränkt. Diese berechnet sich zu

$$X^*[k] = \sum_{n=0}^{N-1} x[n] \cos \left[ \frac{\pi k}{N} \left( n + \frac{1}{2} \right) \right] \quad \text{für } k = 0, \dots, N-1 \quad (2.22)$$

Die Twiddlefaktormatrix kann in Matlab mit

$$W = \cos \left( \frac{\pi}{N} \cdot ([0 : N-1]') * ([0 : N-1] + \frac{1}{2}) \right) \quad (2.23)$$

berechnet werden. Da die diskrete Kosinus Transformation (DCT) anders als die DFT nur auf Kosinusfunktionen und nicht aus einer Kombination aus Sinus und Kosinus, liefert sie rein reellwertige Werte.

## 3 Analyse

Im diesem Kapitel werden zunächst die DFT und die DCT in verschiedenen Größen einander gegenüber gestellt und eine Entscheidung darüber getroffen, welche sich besser dafür eignet auf einem ASIC implementiert zu werden. Hierbei spielen in erster Linie die Anzahl unterschiedlicher Faktoren eine Rolle, da für identische Faktoren nur eine Multiplikationseinheit nötig ist. Gleiche Faktoren gehen also mit einer kleineren Chipfläche einher, was zusammen mit der schnellen Berechnung, also geringe Zahl benötigter Takte, die beiden Hauptziele bei der Chipimplementierung darstellen. Als interessante Kandidaten wurden primär die Matrizen mit den Größen 8x8, 9x9 und 15x15 ausgewählt. Die 8x8-Matrix hat dieselbe Anzahl der Sensoren wie das derzeitige Demo-Array, sodass die Eingangswerte direkt transformiert werden können. Die beiden anderen haben aufgrund ihrer ungeraden Zahl ihren Mittelpunkt zwischen den mittleren Sensorelementen, was für die weitere Verarbeitung des transformierten Signals von Bedeutung ist. Die Matrix der Dimension 15x15 lässt sich durch Interpolation der Daten errechnen, während es für die 9x9-Matrix bisher keine Überlegungen gibt, wie sie errechnet werden könnte. Die 12x12 sowie die 16x16 werden zum besseren Einordnen der Bewertungen ebenfalls betrachtet. Darüber hinaus ist aus Abschnitt 2.4.4 bekannt, dass die FFT auf  $2^n$  Elementen basiert und es sich hierbei um ein sehr schnelles und effizientes Verfahren handelt.

Im zweiten Schritt wird untersucht, wie die 8x8-DFT, welche als Favorit aus der ersten Betrachtung herausgegangen ist, optimiert werden kann.

### 3.1 Bewertung verschiedener DFT- und DCT-Größen

In diesem Abschnitt sollen Erkenntnisse gewonnen werden, auf denen basierend später die Wahl der Transformation und die Größe ihrer Matrix getroffen werden kann. Die Bewertung berücksichtigt wie bereits angedeutet die beiden Eigenschaften Anzahl verschiedener Faktoren und die gesamte Anzahl an Faktoren, wobei die erst genannte größeren Einfluss auf eine negative Bewertung hat, da sich gleiche Faktoren mit Hilfe des Distributivgesetzes ausklammern lassen. Bei den Faktoren wird zwischen solchen unterschieden, die als trivial erachtet werden, da sie nur einer Addition bedürfen ( $\pm 1$ ), zusätzlich zur Addition nur eine Division durch 2 erfolgt ( $\pm 0,5$ ) oder gar keine Berechnung nötig ist (0) und solchen, die als nicht trivial betrachtet werden müssen, da eine Multiplikation unumgänglich ist (beispielsweise  $\frac{\sqrt{2}}{2}$ ). Begründet werden kann dies mit dem dualen Zahlensystem, da eine Multiplikation mit als trivial eingestuft Werten kein komplexes Schaltnetz erfordert. Um z.B. einen Wert durch 2 zu teilen, erfolgt im Dualen einfach ein Bitshift um eins nach rechts.

Interessant ist die DFT trotz ihrer komplexen Ausgangsmatrix, da sich aus Real- und Imaginärteil direkt der Winkel berechnen lässt. Wie in der Einleitung beschrieben, handelt es sich

hierbei um eine der Größen, die im Rahmen dieses Projekts anhand des Sensorarrays ermittelt werden soll.

### 3.1.1 Bewertung verschiedener DCT-Größen

In Tabelle 3.1 ist die Gegenüberstellung der genannten Größen zu sehen. Für die Bewertung wurde das Matlab-Skript aus Anhang 8.1 geschrieben. Ersichtlich ist, dass die Anzahl verschiedener nicht trivialer Werte etwa der Wurzel aus der Anzahl aller Werte ist. Dies bedeutet im Umkehrschluss, dass im Schnitt jede Zeile einen neuen Faktor einführt. Die Summe nicht trivialer Werte weist bei allen Matrizen mehr als 50% auf.

Tabelle 3.1: Bewertung der DCT-Twiddlefaktor-Matrizen

N	8	9	12	15	16
$N \times N$	64	81	144	225	256
$\sum$ trivialer Werte	8	33	28	63	16
$\sum$ nicht trivialer Werte	56	48	116	162	240
Anzahl verschiedener nicht trivialer Werte	7	7	10	13	15
Verhältnis $\sum$ trivial / $\sum$ nicht trivial	0.143	0.6875	0.2414	0.389	0.067

### 3.1.2 Bewertung verschiedener DFT-Größen

In der Tabelle 3.2 werden die DFT-Matrizen einander gegenüber gestellt. Anders als die DCT haben die Twiddlefaktormatrix und deshalb auch das Ergebnis der DFT einen Real- und einen Imaginärteil. Die Beurteilung basiert auf dem Matlab-Skript aus Anhang 8.2. Wie zu sehen ist, schneiden vor allem die 8x8- und die 12x12-DFT gut ab. Da letztere nur zum Vergleich mit aufgenommen wurde, ist die 8x8-DFT der klare Favorit, welcher im folgenden Abschnitt genauer betrachtet werden soll.

### 3.1.3 Bewertungsfazit

Sowohl die DCT als auch die DFT finden häufig in der Bildverarbeitung Anwendung, so dass bereits diverse Algorithmen für die weiteren Berechnungen vorhanden sind. Beide haben symmetrische Twiddlefaktormatrizen. Der Vorteil der DCT gegenüber der DFT ist, dass sie rein reelle Ergebniswerte liefert. Dem steht als großer Nachteil gegenüber, dass beinahe alle ihrer Twiddlefaktoren zu den nicht trivialen gezählt werden müssen. Des Weiteren verteilen sich ihre Werte auf mehr verschiedene Zahlen, sodass eine effiziente Implementierung gegenüber der DFT, trotz der rein reellen Werte, weniger praktikabel erscheint.

Tabelle 3.2: Bewertung der DFT-Twiddlefaktor-Matrizen

N	8	9	12	15	16
$N \times N$	64	81	144	225	256
trivial $\Re$	48	45	128	81	128
nicht triv. $\Re$	16	36	16	144	128
triv. $\Im$	48	21	96	45	128
nicht triv. $\Im$	16	60	48	180	128
$\sum$ triv.	96	66	224	126	256
$\sum$ nicht triv.	32	96	64	324	256
Anzahl verschiedener nicht trivialer Werte	1	7	1	13	3
Verhältnis $\sum$ trivial / $\sum$ nicht trivial	3	0,6875	3,5	0,3889	1

Als Ausschlag gebendes Kriterium wird letztlich der Vorteil eines komplexen Ergebnisses herangezogen, aus dem sich ohne Umwege der Winkel berechnen lässt.

## 3.2 Genauere Betrachtung der 8x8-DFT

Da die 8x8-DFT als Favorit aus der Betrachtung der Transformationsmatrizen herausgegangen ist, wird diese im nächsten Abschnitt auf ihre Eigenschaften hin untersucht. Dies wird die Grundlage für eine effiziente Implementierung sein. Die Twiddlefaktormatrix der 8x8-DFT besteht, wie bereits aus Gleichung (2.12) bekannt, aus komplexen Zeigern. Die möglichen Werte sind in Abbildung 3.1 zu sehen, in Abbildung 3.2 sind zur besseren Veranschaulichung die Zeiger auf 8 Einheitskreise aufgeteilt, wobei jeder einen Laufindex ( $m$ ) des Zeitbereichs abdeckt. In den einzelnen Kreisen sind wiederum alle Laufindizes ( $n$ ) des Frequenzbereichs zu sehen.

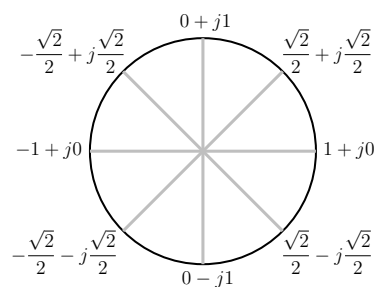


Abbildung 3.1: Einheitskreis mit relevanten Werten der 8x8-DFT

Wie anhand der Grafiken 3.1 zu sehen ist, setzen sich die Faktoren ausschließlich aus den Zahlen  $\pm 1$ ,  $\pm\sqrt{2}/2$  und 0 zusammen. Gemäß der Definition für nicht triviale Werte aus Abschnitt 3.1 zählt ausschließlich der letztgenannte zu diesen. Ebenfalls ist ersichtlich, dass der Betrag

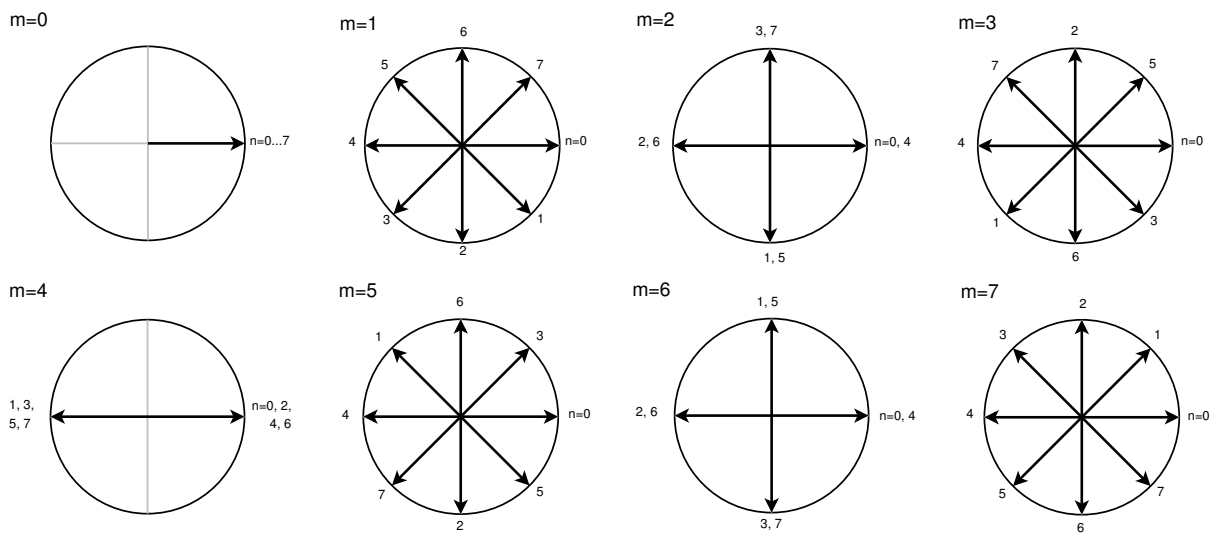


Abbildung 3.2: Twiddlefaktoren der  $8 \times 8$ -Matrix, aufgeteilt auf die Laufindizes  $m$  und  $n$ .  $m$  bezieht sich auf das Element im Ausgangsvektor  $\vec{X}$ ,  $n$  auf den Eingangsvektor  $\vec{x}$ . Siehe auch Gl. (2.11).

aller Zahlen immer 1 ist. In Abbildung 3.3 ist die Twiddlefaktormatrix auf zwei Matrizen aufgeteilt, wobei die Zahlen durch Farben repräsentiert werden. Die linke enthält alle realen Anteile, die rechte alle imaginären. Anhand der Grafik lässt sich gut die Symmetrie erkennen, mit der die Werte auftreten. Diese Grafik soll als Ausgangspunkt für die folgende Betrachtung dienen. Es lässt sich auch gut erkennen, dass die Kreise aus Abbildung 3.2 die Werte der korrespondierenden Zeilen widerspiegeln.

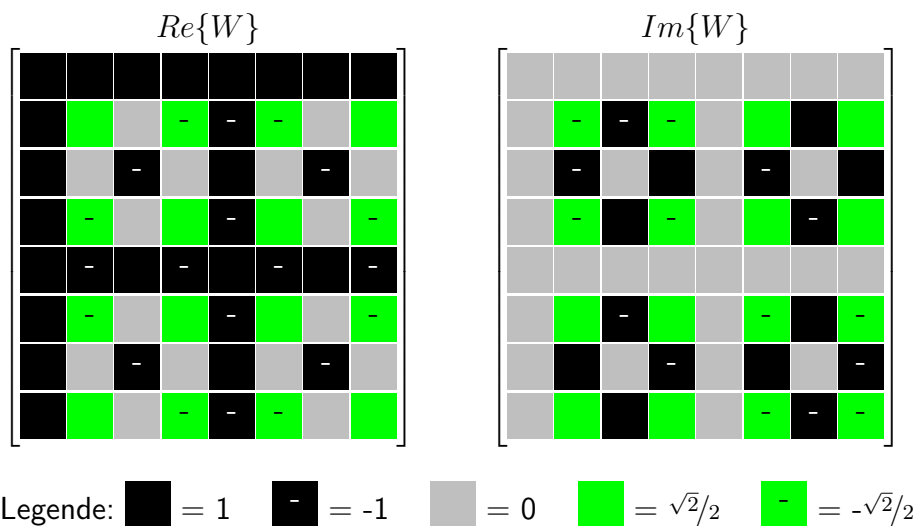


Abbildung 3.3: Matrix-Darstellung der  $8 \times 8$ -DFT-Twiddlefaktoren aufgeteilt nach Real- und Imaginärteil.

Auf den ersten Blick sticht die erste Zeile hervor, da sie im Realteil nur aus positiven Einsen und im Imaginärteil nur aus Nullen besteht. Mit der fünften Zeile verhält es sich ähnlich. Anders ist hier, dass sich positive und negative Einsen abwechseln. In die gleiche Gruppe können noch die dritte und die siebte Zeile zusammengefasst werden. Im Unterschied zu den vorigen können hier aber auch die Imaginärteile eine positive bzw. negative Eins haben. Entsprechend ist dann der Realteil Null. Hier müssen zur Berechnung des Ergebnisses also auch Imaginärteile der Eingangsmatrix mit einbezogen werden. Für die vier bisher betrachteten Zeilen gilt, dass zur Berechnung eines Elements der Ergebnismatrix ausschließlich Additionen oder Subtraktionen erforderlich sind.

Alle Werte, die bis jetzt vorkamen, haben entweder nur einem Real- oder einem Imaginärteil. Dies hat den Vorteil, dass weniger Berechnungen erfolgen müssen, da von einer vollständig komplexen Multiplikation nur eine Multiplikation einer komplexen Zahl mit einer rein reellen (bzw. imaginären) übrig bleiben. Auf diese Weise reduziert sich der in Gleichung (2.1) gezeigte Aufwand zu dem in Gleichung (3.1). Darüber hinaus sind bei den bisherigen Zahlen keine Multiplikationen nötig, weshalb sich der Rechenaufwand auf den Additionsteil der Gleichung beschränkt.

$$\begin{aligned} e + jf &= a \cdot (c + jd) \\ &= a \cdot c + j(a \cdot d) \end{aligned} \tag{3.1}$$

Für die übrigen vier Zeilen gelten die bisherigen Beobachtungen nicht oder nur teilweise, weshalb sie nicht zur ersten Gruppe gezählt werden können. Dafür haben sie aber alle gemein, dass die Hälfte der Faktoren sowohl einen Real- als auch einen Imaginärteil besitzen, welche symmetrisch angeordnet sind. Für diese vier Faktoren sind deshalb jeweils die gesamten vier Multiplikationen aus Gleichung 2.1 nötig.

Eine besondere Eigenschaft ist, dass der Faktor für nicht triviale Multiplikationen im Real- und Imaginärteil zumindest vom Betrag her identisch sind. Dies liegt daran, dass der Einheitskreis in acht Teile geteilt wird und für beispielsweise  $\frac{2 \cdot \pi}{8} = \frac{\pi}{4}$  der Sinus- und Kosinuswert identisch sind. Hieraus resultiert, dass die Hälfte der Berechnungen der nicht trivialen Werte, die für die reelle Matrix gemacht werden müssen, direkt für den imaginären Anteil übernommen werden könnten. Die andere Hälfte müsste lediglich negiert werden. Deshalb kann das berechnete Verhältnis von 3 in Tabelle 3.2 als deutlich höher angenommen werden.

### 3.3 Einordnung des Rechenaufwands

Nachdem nun die Symmetrien der 8x8-Twiddlefaktormatrix der DFT analysiert wurden, soll eine Abschätzung des Rechenaufwands erfolgen. Hierbei wird in vier Kategorien unterschieden. Zum einen werden die erforderlichen Berechnungen bezüglich der 8x8-Twiddlefaktormatrix einerseits für reelle und andererseits für komplexe Eingangswerte betrachtet. Als dritte Variante soll aufgezeigt werden, wie viele Multiplikationen nötig wären, wenn die Twiddlefaktormatrix als variabel angenommen wird. Als letztes soll der Butterfly-Algorithmus auf die Anzahl der benötigten Multiplikationen hin untersucht werden.

Abschließend wird die Bildung des Zweierkomplements der Konstantenmultiplikation unter dem Gesichtspunkt der benötigten Zeit und Fläche gegenüber gestellt. Dies geschieht vor dem Hintergrund, dass je nach Implementierung zwar weniger Multiplikationseinheiten, dafür aber zusätzliche Einheiten zur Negierung von Werten existieren müssen.

Da die Multiplikationen im Normalfall als bedeutend aufwändiger angenommen werden müssen, wird sich in der folgenden Betrachtung hierauf beschränkt. Tatsächlich ist es so, dass die Multiplikationen mit einer Konstanten über ein Schaltnetz innerhalb eines Taktes erfolgen können und somit gerade bei wenigen Multiplikationen die Anzahl der Additionen an Bedeutung gewinnt. Trotzdem erlaubt der Vergleich eine gute Abschätzung.

### 3.3.1 8x8-DFT mit komplexen Eingangswerten

Die Sensormatrix liefert für jedes Sensorelement einen Sinus- und einen Kosinuswert. Diese können für die Berechnung der DFT zu einer komplexen Zahl zusammengefasst werden ( $\cos(x) + j \cdot \sin(x)$ ). Auf diese Weise lässt sich die Berechnung mathematisch kompakter durchführen.

Die Twiddlefaktormatrix der 8x8-DFT weist, wie in Abb. 3.3 zu sehen, insgesamt nur 16 Faktoren auf, die einen Real- und einen Imaginärteil besitzen. Da diese Faktoren sowohl für den Real- als auch den Imaginärteil betragsmäßig alle denselben Wert haben, lässt er sich ausklammern. Bezogen auf die erforderlichen Additionen verschiebt sich so lediglich deren Durchführung auf einem früheren Zeitpunkt. Man könnte sich die Twiddlefaktormatrix also als Matrix mit nur noch einem einzigen komplexen Faktor in den Zeilen 2, 4, 6, 8 vorstellen. Trotz des selben Betrags beider Anteile müssen beide vorhanden sein, sonst würden die Sinusanteile keinen Einfluss in den Realteil des Ergebnisses haben. Da alle Zeilen der Twiddlefaktormatrix mit allen Spalten der Eingangsmatrix multipliziert werden müssen, ergeben sich  $4 \cdot 8 = 32$  Multiplikationen für Real bzw. Imaginärteil der Ergebnismatrix. Zusammen sind also 64 reelle Multiplikationen für die 1D- bzw. 128 2D-DFT nötig.

### 3.3.2 8x8-DFT mit reellen Eingangswerten

Anders als bei der Multiplikation komplexer Eingangswerte sind bei der getrennten Berechnung von Real- und Imaginärteil ungleich viele positive und negative Faktoren je Zeile vorhanden, sodass zu diesem Zeitpunkt davon ausgegangen werden muss, dass eine Negation mancher Werte erforderlich sein wird. Wie ein Vergleich der Gleichungen (2.1) und (3.1) zeigt, entfallen die Hälfte der Multiplikationen, wenn die Eingangswerte rein reell sind. Da der Imaginärteil der Eingangswerte aber getrennt berechnet wird, treten diese Multiplikationen an anderer Stelle wieder auf. Hier findet also keine Ersparnis statt. Allerdings kommen bei rein reellen Eingangswerten beispielsweise keine  $j^2$ -Komponenten zustande, welche ausmultipliziert und anschließend aufaddiert werden müssten. Dies führt zu der in Abschnitt 2.4.3 gezeigten Eigenschaft, dass die letzten drei Zeilen für den Realteil des Ergebnisses direkt bzw. für den Imaginärteil negiert aus den Zeilen 2-4 übernommen werden können. Spätestens an dieser Stelle müssen also Negationen erfolgen. Da zu den drei Zeilen aus Abb. 2.4 auch die beiden gehören, in denen Multiplikationen durchgeführt werden müssen, entfallen bei reellen Eingangswerten die Hälfte

der Multiplikationen im Vergleich zu komplexen Eingangswerten, weshalb für die 1D-DFT nur 32 bzw. für die 2D-DFT nur 64 Multiplikationen nötig sind.

Interessant ist dieser Ansatz dann, wenn entweder die Recheneinheit so klein wie irgend möglich gehalten werden soll und Real- und Imaginärteil der Eingangsmatrix nacheinander berechnet werden können oder die Berechnung äußerst schnell erfolgen muss. In beiden Fällen wird im Vergleich zur Berechnungen mit komplexen Eingangswerten deutlich mehr Speicher benötigt. Insgesamt übersteigt bei diese Art der Berechnung der Flächenbedarf der gesamten Einheit den der komplexen Variante. Auch die Leitungen um den Speicher anzubinden dürfen nicht vernachlässigt werden.

### 3.3.3 Direkte Multiplikation zweier 8x8 Matrizen mit komplexen Werten

Diese Art der Implementation hätte den Vorteil, dass sich zu einem späteren Zeitpunkt für ein anderes Transformationsverfahren entschieden und einfach deren Twiddlefaktoren geladen werden könnten. Da keinerlei Optimierungen möglich sind, ist hier auch eine flexible Größe denkbar. Um einen Vergleich zu ermöglichen, wird die Multiplikation zweier 8x8 Matrizen betrachtet. Die in Abschnitt 2.2.2 erläuterte Matrixmultiplikation bedarf bei einer 8x8 Matrix je Element der Ausgangsmatrix 8 komplexe Multiplikationen. Für die  $8 \cdot 8 = 64$  Elemente werden deshalb 512 komplexe Multiplikationen benötigt. Da es sich sowohl bei den Eingangswerten als auch bei der Twiddlefaktormatrix um komplexe Zahlen handelt, sind, wie in Abschnitt 2.2.1 beschrieben, insgesamt  $512 \cdot 4 = 2048$  Multiplikationen nötig. Für die 2D-DFT sind mit 4096 entsprechend doppelt so viele Multiplikationen nötig.

### 3.3.4 Betrachtung des Butterfly-Algorithmus für 8 Eingangswerte

Abbildung 3.4 illustriert die FFT anhand eines Eingangsvektors mit acht Werten. Um diesen Algorithmus anwenden zu können ist es erforderlich, dass die Werte im Eingangsvektor in umgekehrte Bitreihenfolge getauscht werden (bitreversed order). Dies geschieht nach dem Muster, dass die Indizes der Eingangswerte, wie üblich bei 0 beginnend, binär dargestellt werden. Nun wird die Reihenfolge der Bits getauscht. Auf diese Weise tauschen bei einem 8-Bit Vektor die Elemente 2 und 5 sowie 4 und 7 ihre Position. Andernfalls wären die Ergebnisse in vertauschter Reihenfolge. Anhand der Grafik lässt sich erkennen, dass die DFT in mehrere Stufen aufgeteilt wird.

Aus Gleichung (2.12) ist bekannt, dass die Variablen der Twiddlefaktorberechnung die Indizes der Eingangs- sowie Ausgangsvektoren sind. Hieraus lässt sich bereits erkennen, dass die gesamte Twiddlefaktormatrix  $N$  verschiedene komplexe Werte enthält. Dies wird auch aus Abbildung 3.1 am Beispiel für  $N=8$  ersichtlich. Darüber hinaus lässt sich erkennen, dass die komplexen Zeiger den Einheitskreis in  $N$  Bereiche mit einem Winkel von  $\frac{2\pi}{N}$  unterteilen. Bekannt ist ebenfalls, dass der erste Wert immer die 1 ist. Daraus ergeben sich bei einer DFT mit 2 Eingangswerten die Twiddlefaktoren 1 und  $-1$ , so dass eine Multiplikation entfällt. Dies bildet die erste Stufe.



Ähnlich verhält es sich mit der zweiten Stufe. Hier ergeben sich die Werte  $1, -j, -1, j$ , was ebenfalls bedeutet, dass keine Multiplikation erfolgen muss. Der nächste Schritt zur Reduzierung des Rechenaufwandes ergibt sich aus der Erkenntnis, dass die Werte  $\exp(-i2\pi mn/N)$  und  $\exp(-i2\pi \frac{mn}{2}/N) = -\exp(-i2\pi mn/N)$  lediglich ein negiertes Vorzeichen haben. Auch dies lässt sich der Abb. (3.1) entnehmen. Auf diese Weise fällt der Faktor  $-j$  weg. Dies bedeutet, dass sich so die Hälfte der Multiplikationen einsparen lässt.

Bei der dritten Stufe gibt es wegen der acht Eingangswerte theoretisch auch acht Faktoren. Aus den genannten Symmetriegründen halbiert sich die Anzahl. Wiederum die Hälfte davon sind komplexe Faktoren, die übrigen erfordern keine Multiplikation. Dies bedeutet, dass zwei komplexe Multiplikationen durchgeführt werden müssen, was insgesamt acht reellen Multiplikationen entspricht.

Wie gezeigt wurde, werden nur 2 komplexe Multiplikationen benötigt statt der nach Gleichung (2.20) geschätzten  $8/2 \cdot 3 = 12$ . So ergeben sich für alle 8 Spalten einer  $8 \times 8$ -Matrix tatsächlich nur  $2 \cdot 8 = 16$  komplexe Multiplikationen. Für die 2D-DFT sind somit lediglich 32 komplexe, beziehungsweise 128 reelle Multiplikationen erforderlich.

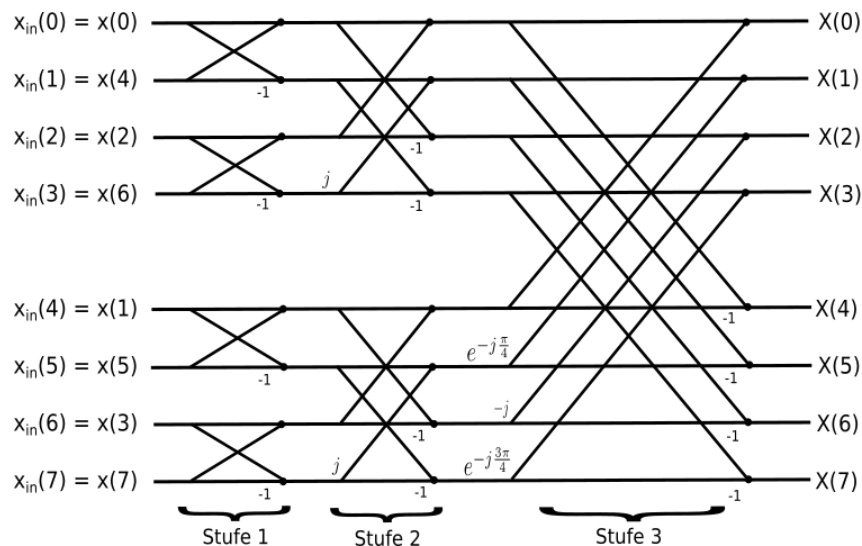


Abbildung 3.4: Berechnungsschema der DFT mit 8 Eingangswerten nach dem Butterfly-Verfahren

### 3.3.5 Fazit der Berechnungs-Gegenüberstellungen

Es konnte gezeigt werden, dass die optimierte Matrixmultiplikation mit komplexen Eingangswerten die gleiche Anzahl benötigter Multiplikationen wie die FFT hat. Das liegt daran, dass es vom Betrag her nur einen einzigen Faktor gibt. Dieser tritt nur in der Hälfte der Zeilen der Twiddlefaktormatrix auf und kann wegen des Distributivgesetzes ausgeklammert werden. Gleiches gilt für die Berechnung mit rein reellen Eingangswerten, hier kann wegen der Symmetrieeigenschaften zusätzlich noch die Hälfte der Multiplikationen eingespart werden. Bei Twiddlefaktormatrizen mit mehreren als nicht trivial einzustufenden Faktoren ist davon auszugehen, dass die FFT das effizientere Verfahren darstellt.

Als Vorteil kann bei der Multiplikation mit komplexen Eingangswerten gesehen werden, dass die Programmierung der 2D-DFT als einfacher angenommen werden kann. Begründet wird dies damit, dass es möglich ist, eine einzige Einheit für die Berechnung der 1D-DFT und der 2D-DFT verwenden zu können.

Die Matrixmultiplikation mit ladbaren Faktoren ist so weit abgeschlagen, dass sie nicht ansatzweise in Betracht gezogen werden kann. Sie verdeutlicht dafür sehr gut, wie sehr sich Berechnungsaufwand einsparen lässt, wenn sich für ein dediziertes System entschieden und dieses optimiert wird. Falls also zwei verschiedene Transformationsverfahren auf einem Chip vorhanden sein sollen, wäre es immer noch effizienter beide optimiert zu implementieren. Abschließend werden in Tabelle 3.3 die Anzahl benötigter reeller Multiplikationen der verglichenen Methoden aufgeführt.

Tabelle 3.3: Auflistung benötigter reeller Multiplikationen der verschiedenen Methoden für die 2D-DFT

Methode	Anzahl reeller Multiplikationen
komplexe Eingangswerte	128
reelle Eingangswerte	64
ladbare Matrixmultiplikation	4096
FFT	128

## 4 Entwurf

### 4.1 Projekt- und Programmstruktur

#### 4.1.1 Vorüberlegungen aus Hardwaresicht

Für die binäre Darstellung der Eingangswerte sind zwölf Bit vorgesehen, wovon zehn auf die Nachkommastellen entfallen. Bei den beiden Rechenoperationen Addition und Subtraktion entstehen keine weiteren Nachkommastellen. Je nach Vorzeichen und Zahlenwert kann es jedoch sein, dass der Vorkomma-Wertebereich von zwei Bit nicht mehr ausreicht. Aus diesem Grund muss der Zielvektor immer um ein Bit breiter sein - von 12 Bit ausgehend also 13 Bit. Da dies bei jeder Addition geschieht, würde die benötigte Bitbreite ohne Gegenmaßnahmen immer weiter anwachsen. Die einfachste Möglichkeit dies zu verhindern ist die Division eines Summationsergebnisses durch Zwei. Diese Operation ist in Hardware mittels eines Bitshifts um Eins nach rechts sehr einfach umzusetzen. So kann beliebig oft auf den selben Vektor eine Addition ausgeführt werden, ohne einen Überlauf zu provozieren. Die Kehrseite dieser Vorgehensweise ist, dass durch den Bitshift die Genauigkeit des Ergebnisses sinkt. Dies ist immer dann der Fall, wenn das hinausgeschobene Bit eine Eins war, was im Mittel mit einer Wahrscheinlichkeit von 50% angenommen werden kann. In der Hälfte aller Fälle wird also der Wert um ein halbes LSB zu viel verringert.

Für die verlustfreie Multiplikation zweier Zahlen wird sogar die doppelte Breite des Vektors benötigt. Hier kann sich der Bitbereich vor und nach dem Komma ändern. Aus diesem Grund müssen die positiven Ergebnisse der 2D-DFT als etwas zu klein und die negativen vom Betrag her etwas zu groß angenommen werden.

Wenn diese Maßnahme jedoch nicht ergriffen würde und alle entstehenden Bits beibehalten werden sollen, müsste mit etwa 70 Bit je Ausgangswert gerechnet werden. Sollte wenigstens auf die bei der Multiplikation entstehenden zusätzlichen Nachkommabits verzichtet werden, würde die benötigte Bitbreite immer noch bei über 40 liegen. Auch dies würde noch eine immense Vergrößerung der Schaltung alleine schon wegen der zusätzlichen Leitungen bedeuten. Desweiteren würden sich hierdurch die benötigten Zeiten aller Berechnungen erhöhen, was eine langsamere Taktfrequenz oder eine Unterteilung in Teilschaltnetzwerke und somit in der Summe mehr Takte mit sich ziehen würde.

#### 4.1.2 Bibliotheken

```
library ieee;  
  use ieee.std_logic_1164.all;  
  use ieee.numeric_std.all;  
  use ieee.std_logic_arith.all;
```

```

use ieee.std_logic_unsigned.all;
library STD;
use STD.TEXTIO.ALL;
use ieee.std_logic_textio.all;
VHDL 2008 kann auch Kommazahlen darstellen ( signed fixed : sfixed(2 downto -10) )

```

### 4.1.3 Struktureller Aufbau

Das Programm wurde, wie üblich, auf verschiedene Dateien aufgeteilt. Es wurden alle Konstanten in einem Paket deklariert, welches in allen anderen Dateien eingebunden werden muss. So ist es möglich insbesondere die Bitbreiten eines Vektors an einer zentralen Stelle zu setzen. Die Bitbreiten liegen für Eingangswerte bei 12 Bit, Summen 13 Bit und Produkte 26 Bit. Da alle anderen Dateien auf diese Konstanten zugreifen, muss dieses Paket zuerst kompiliert werden. In einem weiteren Paket findet die Deklaration der eigenen Datentypen statt. Hier sei insbesondere die 8x8 Matrix mit ihren 12 Bit Vektoren vom Typ signed erwähnt. Da die weiteren Dateien diese Datentypen verwenden, ist es erforderlich, dass dieses Paket als zweites kompiliert wird. Alle weiteren Pakete können in beliebiger Reihenfolge kompiliert werden, da sie erst später ineinander greifen. Zum Testen bzw. für die Simulation müssen Eingangswerte geladen werden. Hierfür wurde die Komponente `read_input_matrix` geschrieben. Die Werte müssen in einer Datei mit dem Namen `InputMatrix_komplex.txt` stehen und im dualen Zahlenformat vorliegen.

```

writefile (write_results)
resize-Funktion
Matlab fi
NC Sim, nur Integerdarstellung möglich, bei Vektoren sogar nur positiv

```

## 4.2 Entwicklungsstufen

### 4.2.1 2D-DFT-Implementierung in VHDL

In Tabelle 4.1 ist eine Auflistung der für die Berechnung veranschlagten Takte für die Multiplikation einer beliebigen Matrix mit der Twiddlefaktormatrix für die 8x8-DFT zu sehen. Grundlage ist, dass in einem Takt Summanden paarweise aufaddiert werden und in einer Variablen zwischengespeichert werden. Dieses Verfahren kann auch als Baumstruktur aufgefasst werden. Wie das Aufsummieren erfolgt, kann in Abschnitt 4.2.5 detaillierter nachgelesen werden.

Wie in Abschnitt 4.3.1 gezeigt wird, kann die Multiplikation mit einer Konstanten innerhalb eines Taktes mit einem Schaltnetz erfolgen.

In Abschnitt 4.3 wird gezeigt, dass der kritische Pfad der Negierung sogar etwas länger als beim Konstantenmultiplizierer ist. Um keine zu langen Signal- und Gatterlaufzeiten hervorzurufen, sollte hierfür unbedingt ein eigener Takt eingeplant werden. Dadurch relativiert sich der zeitliche Gewinn allerdings etwas.

Tabelle 4.1: Takte für die komplexe DFT

Zeile	Additionen pro Element ( $N$ )	Takte pro Element ( $\log_2(N)$ )	Takte für Multiplikation	Summe der Takte
1	8	3	0	3
2	12	3,6	1	5
3	8	3	0	3
4	12	3,6	1	5
5	8	3	0	3
6	12	3,6	1	5
7	8	3	0	3
8	12	3,6	1	5

Anhand der rechten Spalte ergeben sich so  $(3+5) \cdot 4 \cdot 8 = 256$  Takte sowohl für den Real- als auch den Imaginärteil der komplexen Ausgangsmatrix. Real- und Imaginärteil werden parallel berechnet und sind somit zeitgleich fertig.

### 4.2.2 Weiterentwicklung zur 2D-DFT

Ziel ist es die gleiche DFT-Einheit für beide DFTs zu verwenden

Zähler für 64 Werte kann als 6 Bit Vektor realisiert werden, der bei 63 einen Überlauf hat und wieder bei 0 anfängt.

Vorderen 3 Bit sind die der Zeile, die hinteren für die Spalte.

Das dritte Bit von vorne sagt einem, ob es eine gerade oder ungerade Zeile ist.

Zweite interne Matrix erforderlich, da sonst die Werte oberhalb der Nebendiagonalen überschrieben wurden

Die in Gleichung (2.17) beschriebene Berechnung der 2D-DFT lässt sich auch wie folgt schreiben:

$$\begin{aligned}
 X &= W \cdot x \cdot W \\
 &= (x^T \cdot W)^T \cdot W
 \end{aligned} \tag{4.1}$$

$$\begin{aligned}
 &= X^* \cdot W \\
 &= ((x \cdot W)^T \cdot W)^T \\
 &= (X^{*T} \cdot W)^T
 \end{aligned} \tag{4.2}$$

In Matlab muss hierfür entweder die Funktion `transpose()` oder `.'` verwendet werden. Letzteres muss elementweise angewandt werden, da das Apostroph alleine die komplex konjugiert Transponierte bildet.

Die alternativen Schreibweisen der 2D-DFT haben den Vorteil, dass in beiden Fällen die Eingangsmatrix auf der linken Seite steht. Möglich ist dies, da die Twiddlefaktormatrix identisch

mit ihrer Transponierten ist. Dass nun in den Gleichungen (4.1) und (4.2) sowohl die Eingangs- als auch die 1D-DFT-Matrix links steht, ist eine wichtige Voraussetzung dafür, dass mit der selben Recheneinheit mit der die 1D-DFT berechnet wird auch die 2D-DFT berechnet werden kann. Die zweite Voraussetzung ist das Transponieren einer Matrix. Diese lässt sich durch spaltenweises Abspeichern und zeilenweises Auslesen der Ergebnis-Matrix realisieren. Hierfür ist es lediglich notwendig die beiden Indizes, welche ein Matrixelement ansprechen, beim Speichern getauscht werden. Nun sind nun alle Voraussetzungen erfüllt, um beide Berechnungen mit der selben Einheit durch zu führen. In Grafik (4.1) ist das hier beschriebene veranschaulicht.

(Auf diese Weise wird die direkte Weiterverarbeitung von Werten denkbar.)

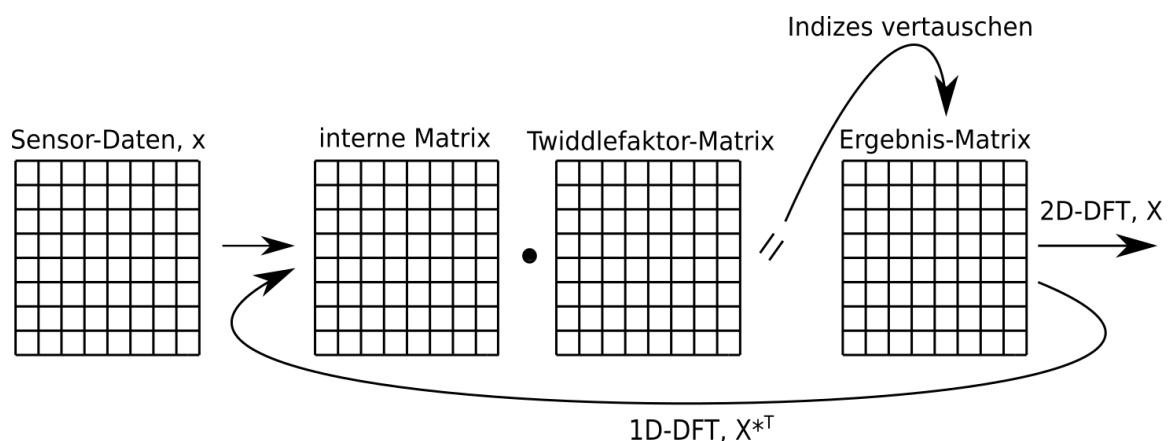


Abbildung 4.1: Darstellung der Berechnung der 2D-DFT aus Gleichung (4.2)

**$W = \text{transpose}(W)$**

### 4.2.3 Direkte Weiterverarbeitung der Zwischenergebnisse

Um die Anzahl an Gattern und somit den Flächenbedarf zu reduzieren ist es das Ziel, die Ergebnisse der 1D-DFT aus der 1. Berechnungsstufe im nächsten Schritt direkt als Eingangswerte für die 2D-DFT zu verwenden. Auf diese Weise würden  $64 \cdot 2 \cdot 12 \text{ Bit} = 1536 \text{ Bit} = 1,5 \text{ kBit} = 192 \text{ Byte}$  an Speicher eingespart werden. Wie sich im Laufe der Entwicklung gezeigt hat, lässt sich das nicht nutzen. Das liegt daran, dass dazu übergegangen wurde, immer nur ein Element zur Zeit berechnet wird und die bereits errechneten demnach zwischengespeichert werden müssen. Dieser Ansatz wurde verfolgt, da der Entwicklungsaufwand in VHDL für die spaltenweise Berechnung der Ausgangswerte einfacher umzusetzen war und es zunächst nur um die mathematische Umsetzung und nicht um die Platzeffizienz auf einem Chip ging.

Unklar war zu diesem Zeitpunkt noch, wie der Speicher realisiert werden soll. In der finalen Variante des Chips soll es einen Random Access Memory (RAM) geben, der als zentraler Speicher von allen Komponenten genutzt wird. Da die Entwicklung im Projekt noch nicht soweit fortgeschritten ist und dies nicht zu den Aufgaben der vorliegenden Arbeit gehört, wurde auf das Speichern in lokalen Speicherzellen ausgewichen, welche als Variable oder Signal im VHDL-Code definiert und von der Software als Flip-Flop synthetisiert werden.

#### 4.2.4 Optimierte 8x8 DFT als Matrixmultiplikation

Anfangs wurde angenommen, dass Multiplikationen mit den Twiddlefaktoren  $\pm 1$  und  $\pm \frac{\sqrt{2}}{2}$  durchgeführt werden müssen. Dass bei einer optimierten 8x8-DFT wegen des expliziten ausprogrammierens der Berechnungen die Multiplikation mit  $\pm 1$  wegfällt, wurde recht schnell klar. Erst bei genauer Betrachtung der Twiddlefaktor-Matrix viel auf, dass in jeder Zeile gleich viele Additionen wie Subtraktionen vorhanden sind. Durch Umsortieren ist es dadurch möglich auf das Invertieren der Eingangswerte sowie den hierfür benötigten Takt und die Inverter zu verzichten. Weiter wird auch nur die Multiplikation mit  $+\frac{\sqrt{2}}{2}$  benötigt.

Anfangs wurde in Betracht gezogen das 1er-Komplement zu verwenden, da hierbei zwei betragsmäßig identische Zahlen sich nur durch ihr höchstwertigstes Bit unterscheiden. Auf diese Weise könnte das selbe Resultat für den Imaginär- wie für den Realteil verwendet werden, das Vorzeichen würde sich über eine einfache XOR-Verknüpfung beider MSB der Multiplizierten ergeben. Diesem Vorteil steht jedoch eine aufwändigere Subtraktion (bzw. Addition negativer Zahlen) gegenüber. Der zusätzliche Aufwand entspricht etwa dem der Bildung des 2er-Komplements. Aus diesem Grund und da das 2er-Komplement deutlich verbreiteter ist sowie weitere Vorteile bringt wie beispielsweise keine Doppeldeutigkeit durch eine negative Null hat, wurde sich hierfür entschieden.

In späteren Analysen konnte festgestellt werden, dass sowohl für den Real- als auch den Imaginärteil gleichviele Multiplikationen mit positiven wie mit negativen Faktoren durchgeführt werden müssen. Dies lässt sich anhand des Einheitskreises in Abb. 3.1 und der Abbildung 3.2 nachvollziehen. Aus Abschnitt 2.2.2 ist bekannt, dass bei einer Matrixmultiplikation Elemente multipliziert und anschließend die Ergebnisse aufsummiert. Da es das Kommutativgesetz erlaubt die berechneten Ergebnisse auch in einer anderen Reihenfolge zu addieren,

In Abbildung 3.3 ist zu sehen, dass die 2., 4., 6., und 8. Zeile je vier nicht triviale und komplexe Faktoren enthält. Darüber hinaus ist ersichtlich, dass für komplexe Eingangswerte in den genannten Zeilen 12 und in den übrigen 8 Multiplikationen erfolgen müssen. Dies kann anhand der Gleichungen (2.1) und (3.1) nachvollzogen werden. Das lässt sich ausnutzen, um keine Negationen der Eingangs- und Zwischenwerte durchführen zu müssen.

Darüber hinaus minimiert sich bei geschickter Anordnung das Risiko eines Überlaufs. Wie in Abschnitt 2.1.4 erwähnt, ist dies nicht Gegenstand dieser Arbeit, weshalb der Einfachheit wegen zur Sicherheit dennoch nach jeder Addition oder Subtraktion das Ergebnis durch einen Bitshift halbiert wird. Es sei an dieser Stelle lediglich angemerkt, dass über die Eingangswerte die Annahme getroffen werden kann, dass aufeinanderfolgende Werte das selbe Vorzeichen haben. Dies ließe sich ausnutzen, um noch weiter die Wahrscheinlichkeit zu reduzieren, dass es zu einem Überlauf kommt.

Dies hat zur Folge, dass ein sehr großer Wert entstehen kann, welcher Maßgebend für die Anzahl der Vorkommabits sein kann. Wegen der Null im Imaginärteil der ersten Zeile der Twiddlefaktormatrix sind auch alle Imaginärteile der ersten Spalte der Ausgangsmatrix Null. Bezogen auf den Imaginärteil gilt das Gleiche auch für die fünfte Zeile der Twiddlefaktormatrix. Da sich beim Realteil der fünften Zeile der Twiddlefaktormatrix der fünften Zeile positive und negative Einsen abwechseln, sind hier auch keinerlei Multiplikationen nötig.

Aus der anfänglichen Implementation bei der alle nicht trivialen Werte einer Berechnung die

entweder mit  $+\frac{\sqrt{2}}{2}$  oder  $-\frac{\sqrt{2}}{2}$  multipliziert wurden, einzeln berechnet werden, wird mittels des Distributivgesetzes der gemeinsame Faktor ausgeklammert, sodass nur noch jeweils eine Multiplikation erforderlich ist.

Bei den weiteren Zeilen sind hingegen die Zahlen zur Hälfte positiv und zur anderen negativ. Außerdem enthalten die geraden Zeilen den Faktor  $\pm\frac{\sqrt{2}}{2}$ . Dies lässt sich ausnutzen, um die Anzahl der Multiplikationen zu reduzieren. Zunächst können die

Für jede gerade Zeile der DFT ist jeweils für den Real- und den Imaginärteil eine Multiplikation nötig, so dass sich insgesamt acht Multiplikationen ergeben

#### 4.2.5 Berechnungsschema der geraden und ungeraden Zeilen

In Abbildung 4.2 ist die Berechnung der ungeraden Zeilen am Beispiel der ersten zu sehen.

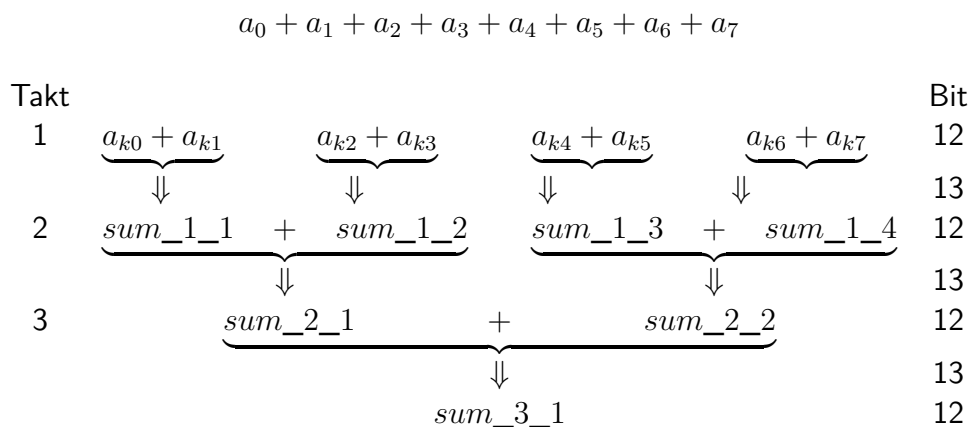


Abbildung 4.2: Vorgehensweise der Akkumulation der ungeraden Spalten der Eingangswerte.

Wie der linken Spalte zu entnehmen ist, werden 3 Takte für die Berechnungen der Werte aus den ungeraden Spalten der Eingangsmatrix bzw. ungeraden Zeilen der 1D-DFT-Matrix benötigt. 1. Takt für Additionen bzw. Subtraktionen und 2. sowie 3. Takt für das Aufsummieren. Der Bitvektor des Ergebnisses ist zwar 12 Bit breit, aber beim letzten Bitshift von 13 auf 12 werden nur 11 Bit übernommen. Es wird also ein doppelter Bitshift vollzogen. Dies erfolgt, damit sowohl in den geraden als auch in den ungeraden Zeilen gleich viele Bitshifts erfolgen und die Werte somit identisch skaliert sind.

Die Berechnung der geraden Zeilen wird in Abbildung 4.3 am Beispiel der zweiten Zeile gezeigt

Auch hier ist der linken Spalte die Anzahl der benötigten Takte zu entnehmen. In diesem Fall werden 5 Takte für die Berechnungen benötigt. Diese setzen sich zusammen aus 1 Takt für Additionen bzw. Subtraktionen, 2.-3. sowie 5. Takt für das Aufsummieren und der 4. Takt für die Multiplikationen.



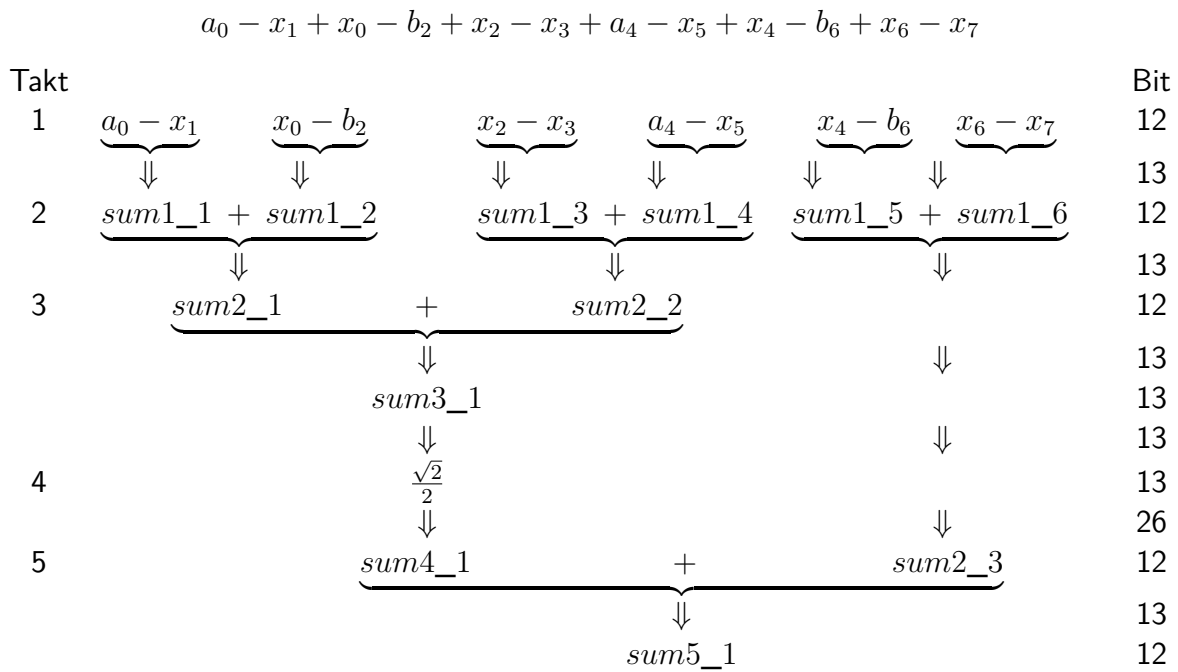


Abbildung 4.3: Vorgehensweise der Akkumulation der geraden Spalten der Eingangswerte.

Wie rechts am Rand zu sehen, ergibt sich durch die Addition eine Bitbreitenerweiterung um 1 bzw. bei der Multiplikation eine Verdoppelung. Bei einer früheren Implementierung, die nur die 1D-DFT beherrschte, wurde zumindest die Erweiterung bei der Addition umgesetzt. Da bei der 2D-DFT die selbe Recheneinheit genutzt werden soll, wurde in Absprache mit dem ISAR-Team entschieden, dass die Summanden vor jeder Summation durch einen Bitshift nach rechts halbiert werden. Auf diese Weise hat ein Additionsergebnis immer 13 Bit Breite. Durch den Bitshift kann das Resultat der 1D-DFT direkt als Eingang für die 2D-DFT verwendet werden.

Zu bedenken gilt es bei einem Bitshift, dass das Ergebnis mit jedem Mal eine Division durch 2 erfährt. Bei hintereinander erfolgenden Bitshifts wird demnach durch  $2^{N_B}$  geteilt, wobei  $N_B$  die Anzahl der Bitshifts ist. Den beiden obigen Darstellungen der Summationen kann entnommen werden, dass, um ein Überlaufen des Bitvektors zu vermeiden es nötig ist, drei respektive vier Bitshifts durch zu führen. Wie bereits erläutert erfolgt bei den ungeraden Zeilen abschließend ein doppelter Bitshift. Auf diese Weise ergibt sich für die 1D-DFT, dass das Ergebnis um den Faktor 16 kleiner ist, als beispielsweise bei der Berechnung mit Matlab. Da bei dem zweiten Durchlauf, um die 2D-DFT zu berechnen, ebenfalls durch 16 geteilt wird, ergibt sich insgesamt eine Division durch  $2^{2 \cdot 4} = 256$ .

### Erwartete Anzahl benötigter Takte

Aus den Abbildungen 4.2 und 4.3 können die Takte die zur Berechnung der 1D- bzw. 2D-DFT benötigt werden abgeleitet werden.

Für ungeraden Zeilen sind je Element 3 Takte nötig und mit 8 Elementen pro Zeile und

4 ungeraden Zeilen errechnen sich so  $3 \cdot 8 \cdot 4 = 96$  Takte. Analog errechnet sich für die ungeraden Zeilen mit je 5 Takten pro Element  $5 \cdot 8 \cdot 4 = 160$  Takte. In der Summe ergeben sich so  $96 + 160 = 256$  Takte für die 1D-DFT. Da die 2D-DFT ohne Takte fürs Umspeichern oder ähnliches sofort im Anschluss berechnet werden kann, verdoppelt sich die Anzahl der Takte auf 512 für die vollständige Berechnung.

### Zusammenhang von DFT und IDFT bei der Matrixmultiplikation

Durch die umgekehrte Drehrichtung des komplexen Zeigers in Gleichung (2.21) werden in der Matrizenschreibweise die Zeilen 2 und 8, 3 und 7 sowie 4 und 6 vertauscht. Nachvollziehen lässt sich das gut anhand der Grafik (3.1). Verdeutlicht wird das vorgehen in Abbildung 4.4.

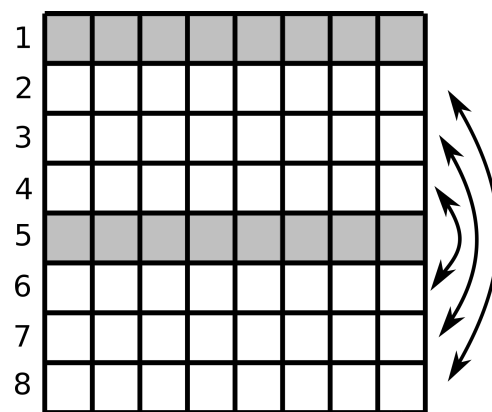


Abbildung 4.4: Um von der DFT zur IDFT zu kommen, müssen bei der Matrixmultiplikation die Zeilen 2 und 8, 3 und 7 sowie 4 und 6 der Twiddlefaktormatrix vertauscht werden.

## 4.2.6 Test der Matrixmultiplikation

### 1D-DFT mit Integer-Werten

### 2D-DFT mit Integer-Werten

### 2D-DFT mit Werten SQ-Format

Unter anderem weil NC Sim bzw. dessen Unterprogramm SimVision zur Anzeige von Signalverläufen (Waveform) nur Integer darstellen kann und bei als Vektor gebündelten Signalen diese nicht einmal als vorzeichenbehaftet (signed), wurde der Einfachheit halber zunächst die Berechnung als Ganzzahl-Multiplikation mit dem Faktor 3 betrachtet. Da es bei diesem Faktor und den gewählten Eingangswerten nicht zu einem Überlauf kommen kann, war es zu diesem Zeitpunkt noch nicht nötig, sich Gedanken über die Breite des Ergebnisvektors bzw. den Ausschnitt daraus für die weitere Berechnung zu machen. Deshalb konnte an dieser Stelle noch auf den Bitshift zur Halbierung der Werte verzichtet werden.

Erst als der Faktor  $\frac{\sqrt{2}}{2}$  übernommen wurde, wurden die Ergebnisse breiter als der Vektor für die weitere Berechnung an Bits zur Verfügung stellt.

$\frac{\sqrt{2}}{2}_{10} = 0001011010100_2$  in S2Q10, als Integer betrachtet jedoch  $724_{10}$ .

Daraus folgt, dass ein Teil der Bits abgeschnitten werden müssen. Da die Dualzahlen jetzt im S1Q10-Format betrachtet werden, es sich also um Kommazahlen handelt, müssen die hinteren Bits abgeschnitten werden. Zudem können vorne Bits ohne Informationsverlust gestrichen werden, da durch die Multiplikation ein weiteres Negations-Bit dazugekommen ist und auf Grund des gegebenen Faktors der Wertebereich vorne nie ganz ausgenutzt wird. (Verifizieren / Belegen!)

## 4.3 Syntheseergebnisse von Teilkomponenten

### 4.3.1 13 Bit Konstantenmultiplizierer

Der duale Wert lässt sich am einfachsten mit der Matlab-Funktion `fi()` ermitteln. Der Funktion werden hierfür Kommagetrennter Dezimalwert, 1 für vorzeichenbehaftet, die gesamte Anzahl an Stellen (13) und die Anzahl der Nachkommastellen (10) übergeben. Der vollständige Aufruf sieht dann wie folgt aus:

```
val=fi(sqrt(2)/2,1,13,10)
```

Der erzeugte Datentyp hat unter anderem die Eigenschaften `val.bin`, welche einem mit `0001011010100` den Wert als Binärzahl zurück gibt, `val.double` gibt den approximierten Dezimalwert mit `0,70703125` zurück und `val.dec` interpretiert den Dualwert als Integer, was `724` entspricht. Letzterer ist wichtig zu kennen, um die Werte der Simulation nachvollziehen zu können.

Der Berechnung aus Gleichung (4.3) kann entnommen werden, dass die Abweichung weit unter einem Prozent liegt.

$$\frac{100}{\frac{\sqrt{2}}{2}} \cdot 0,70703125 = 99,989\% \quad (4.3)$$

Zeigen, welche Bits heraus genommen werden müssen! und belegen warum.

Der vollständige Gate-Report befindet sich in Abschnitt 8.3 auf Seite 51

### 4.3.2 Bildung des 2er-Komplements eines 13 Bit Vektors

In Abb. 4.6 ist die nicht explizit implementierte, aber in Abschnitt 3.3.2 erwähnte Negierung von Zahlen zu sehen.

Für die Negierung eines 13 Bit Vektors hat das Synthesewerkzeug `encounter` 22 Standardzellen verwendet. Das sind knapp doppelt so viele Gatter, wie der Vektor Bits breit ist. Der Unterschied zum Konstantenmultiplizierer fällt somit sehr gering aus. Wie zu sehen, handelt es sich fast ausschließlich um Inverter und Addierer. In Abschnitt 2.1.2 wurde bereits beschrieben, dass für die Bildung des 2er-Komplements zunächst alle Bits invertiert werden müssen. Abschließend wird auf den Vektor 1 LSB addiert. Beide Pfade weisen die gleiche Länge auf

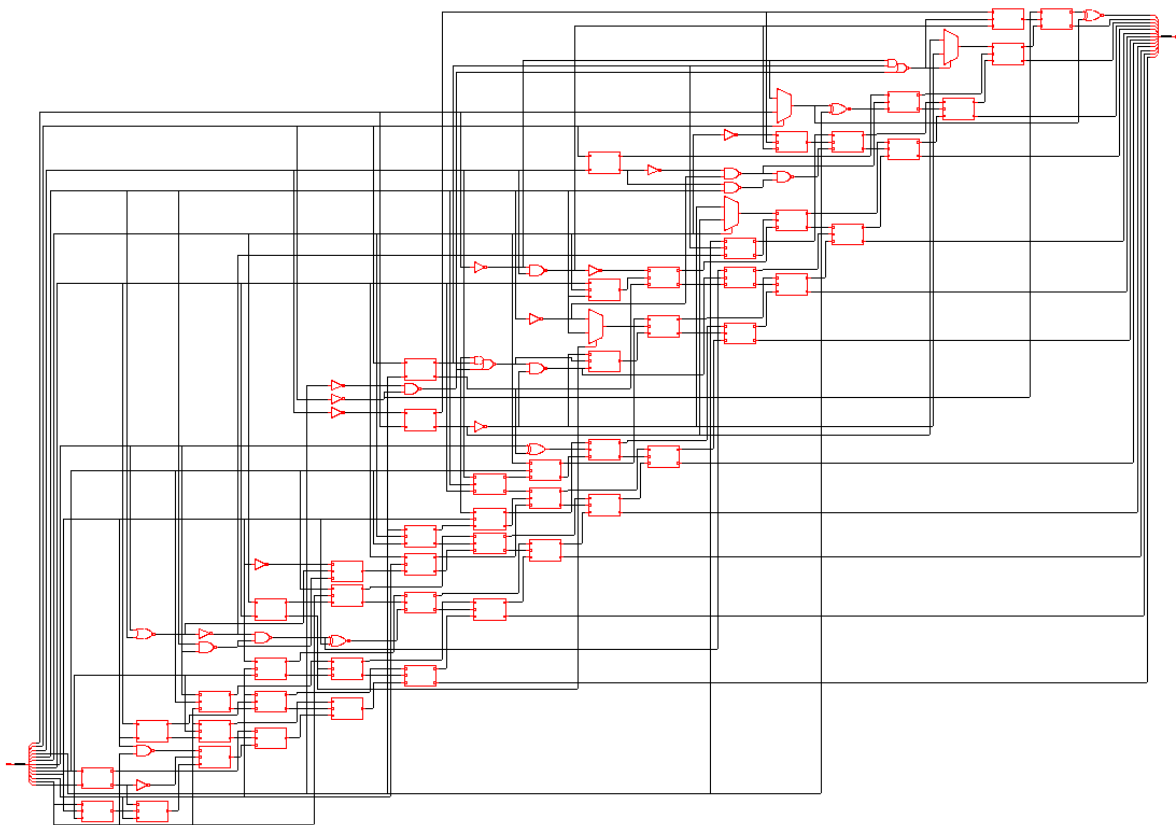


Abbildung 4.5: Schaltnetz des 13 Bit Konstantenmultiplizierers für  $\frac{\sqrt{2}}{2} = 0.70711 \simeq 0.70703125 = 0001011010100_2$  in Encounter; Eingang links, Ausgang rechts

und verwenden überwiegend die selben Gattertypen, weshalb darauf geschlossen werden kann, dass die maximale Gatterlaufzeit in der gleichen Größenordnung liegen muss.

### 4.3.3 13 Bit Addierer

Der 13 Bit Addierer hat zwei 13 Bit Eingänge, allerdings werden durch einen Bitshift beide Eingangswerte halbiert, damit kein Überlauf entsteht. Insofern fließen nur jeweils die höheren 12 Bit in die Berechnung ein.

### 4.3.4 Vergleich der Synthesergebnisse

In Tabelle 4.2 ist eine Gegenüberstellung der Synthesergebnisse unter den Aspekten Anzahl der Gatter und Fläche.

#### Gegenüberstellung der Konstantenmultiplikation und der Bildung des 2er-Komplements

Unter diesem Punkt sollen die Konstantenmultiplikation und die Bildung des 2er-Komplements unter Aspekten der benötigten Zeit und des benötigten Platzes auf einem Chip betrachtet

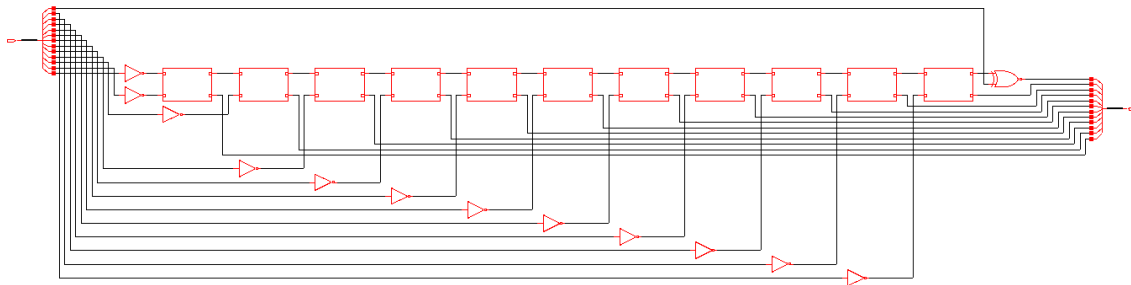


Abbildung 4.6: Schaltnetz einer Einheit zur Bildung des 2er-Komplements eines 13 Bit Vektors; Eingang links, Ausgang rechts

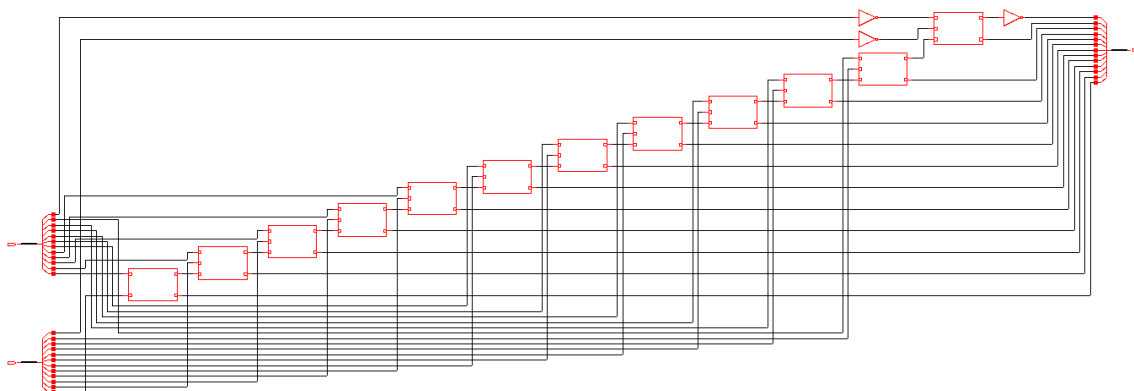


Abbildung 4.7: Schaltnetz eines 12 Bit Addierers, Eingänge links (12 Bit), Ausgang rechts (13 Bit)

Tabelle 4.2: Vergleich

	Gatter	Fläche (Prozess: 350nm)
13 Bit Konstantenmultiplizierer für $\sqrt{2}/2$	82	6612 $\mu\text{m}^2$
13 Bit regulärer Multiplizierer	175	23 261 $\mu\text{m}^2$
12 Bit Addierer	15	3257 $\mu\text{m}^2$
13 Bit 2er-Komplement-Bildung	24	2147 $\mu\text{m}^2$

werden. Um einen Eindruck hiervon zu erhalten, werden im Kapitel Entwurf in Abschnitt 4.3 jeweils die Schaltnetzze gezeigt. Wie dort erläutert, lässt sich anhand dieser sagen, dass es bei dieser Art der Implementierung keinen zeitlichen Gewinn gibt, da beide kritischen Pfade etwa gleich lang sind. Für die knapp 4 mal mehr Gatter bei der Multiplikation ist auch ein größerer Verdrahtungsaufwandt erforderlich, sodass die Konstantenmultiplizierer auf einem Chip eine etwas größere Fläche beanspruchen. Da es sich hier insgesamt aber um wenige Gatter handelt, wirkt sich dies erst bei sehr vielen Instanzen aus. Es kann an dieser Stelle deshalb festgehalten werden, dass dieser Unterschied nicht als entscheidend geltend gemacht werden kann.

## 4.4 Schema der Zustandsfolge

test test

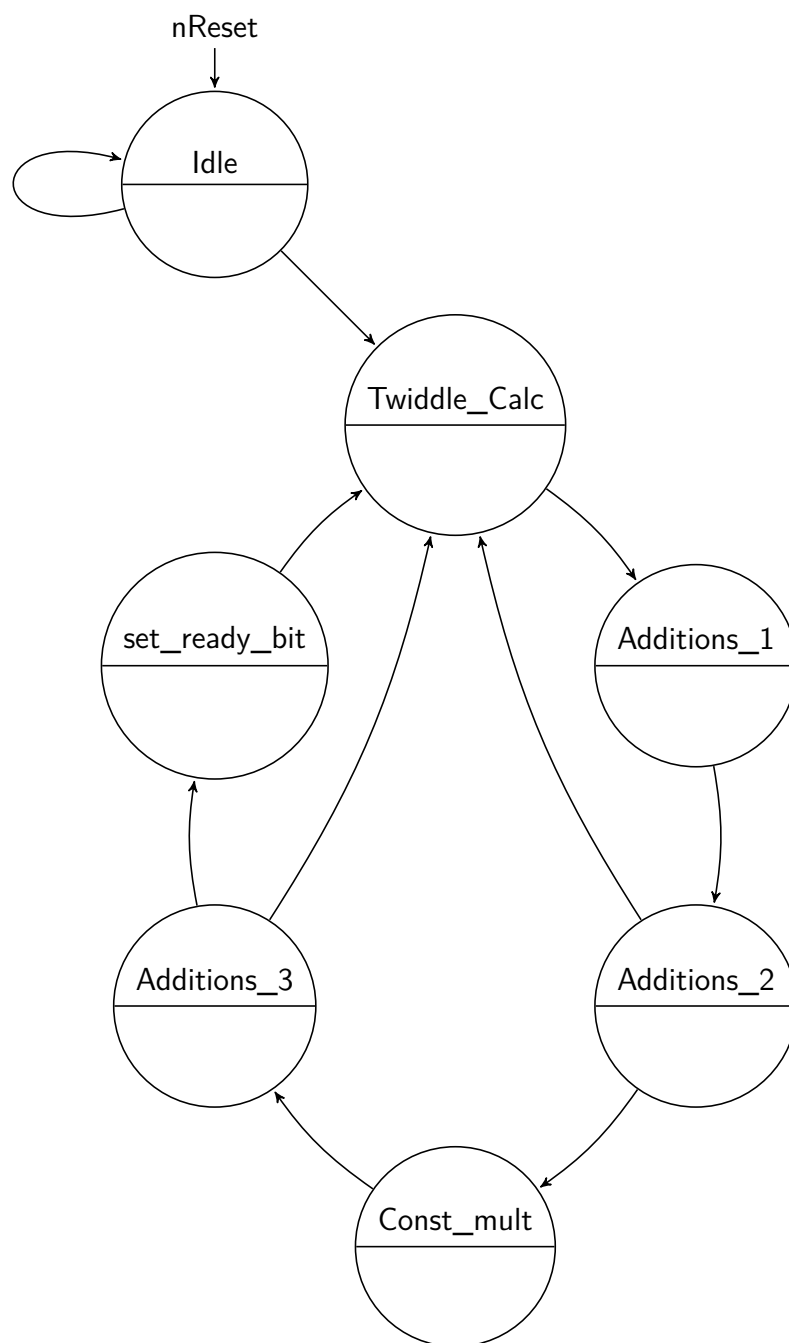
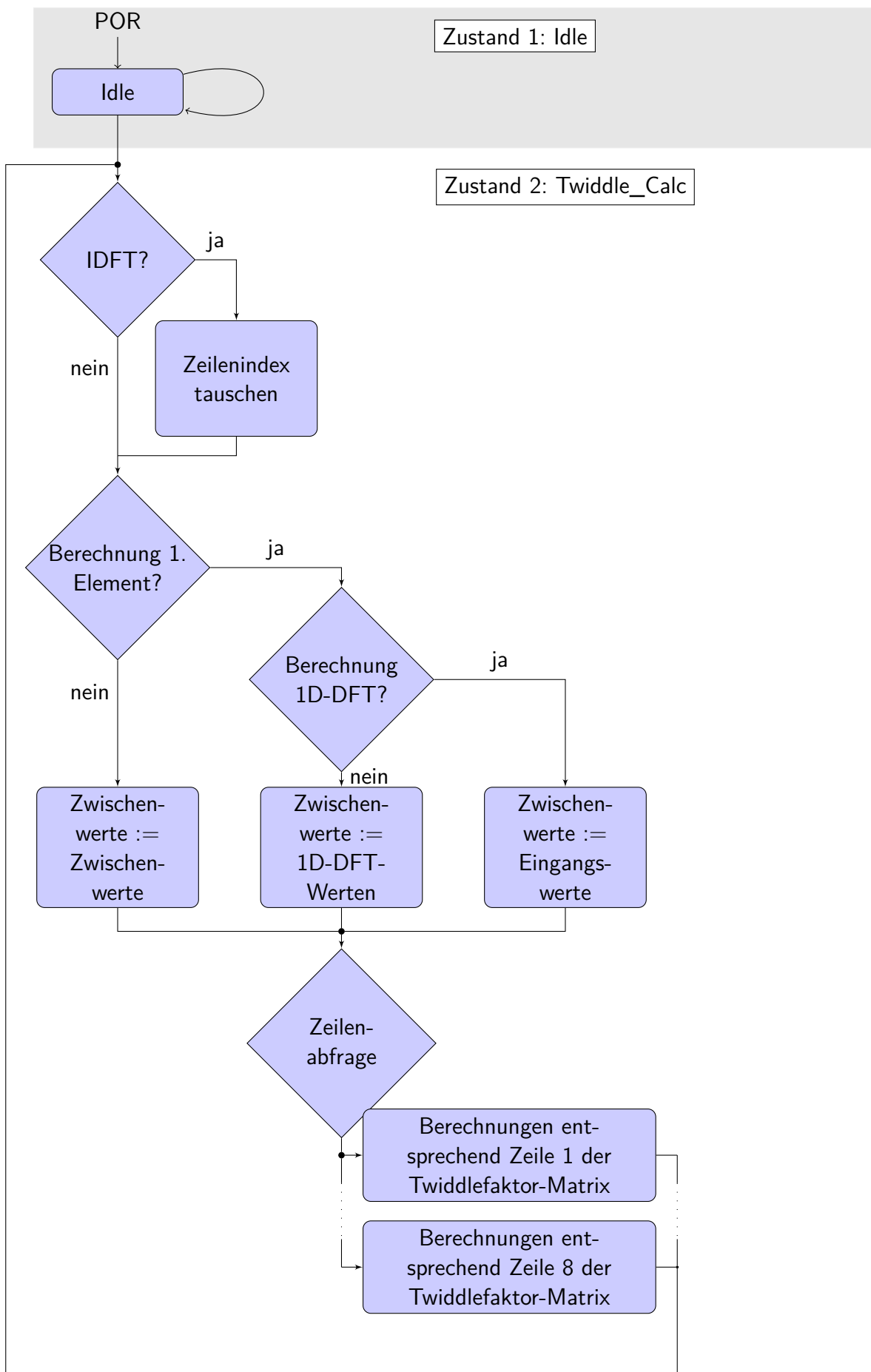
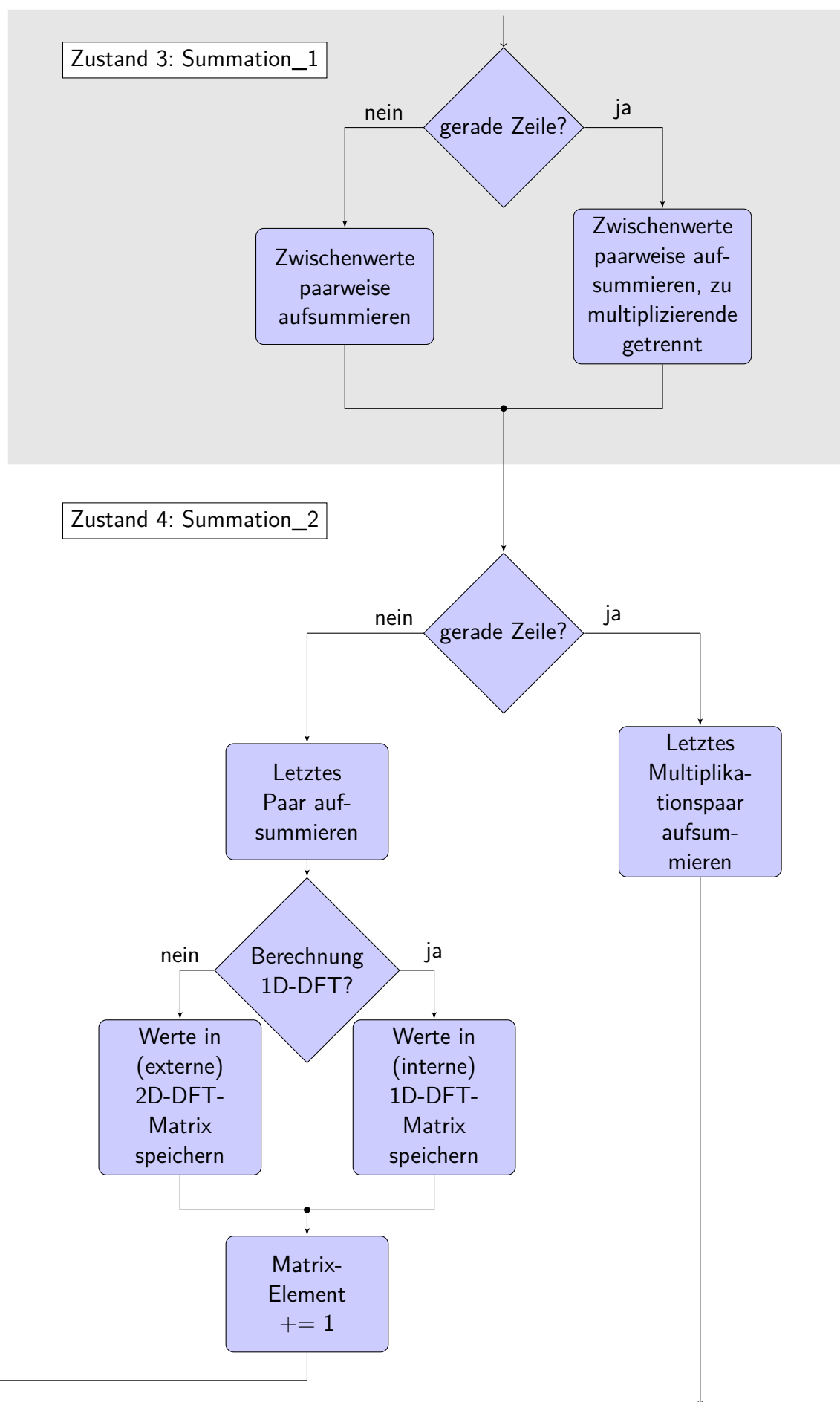


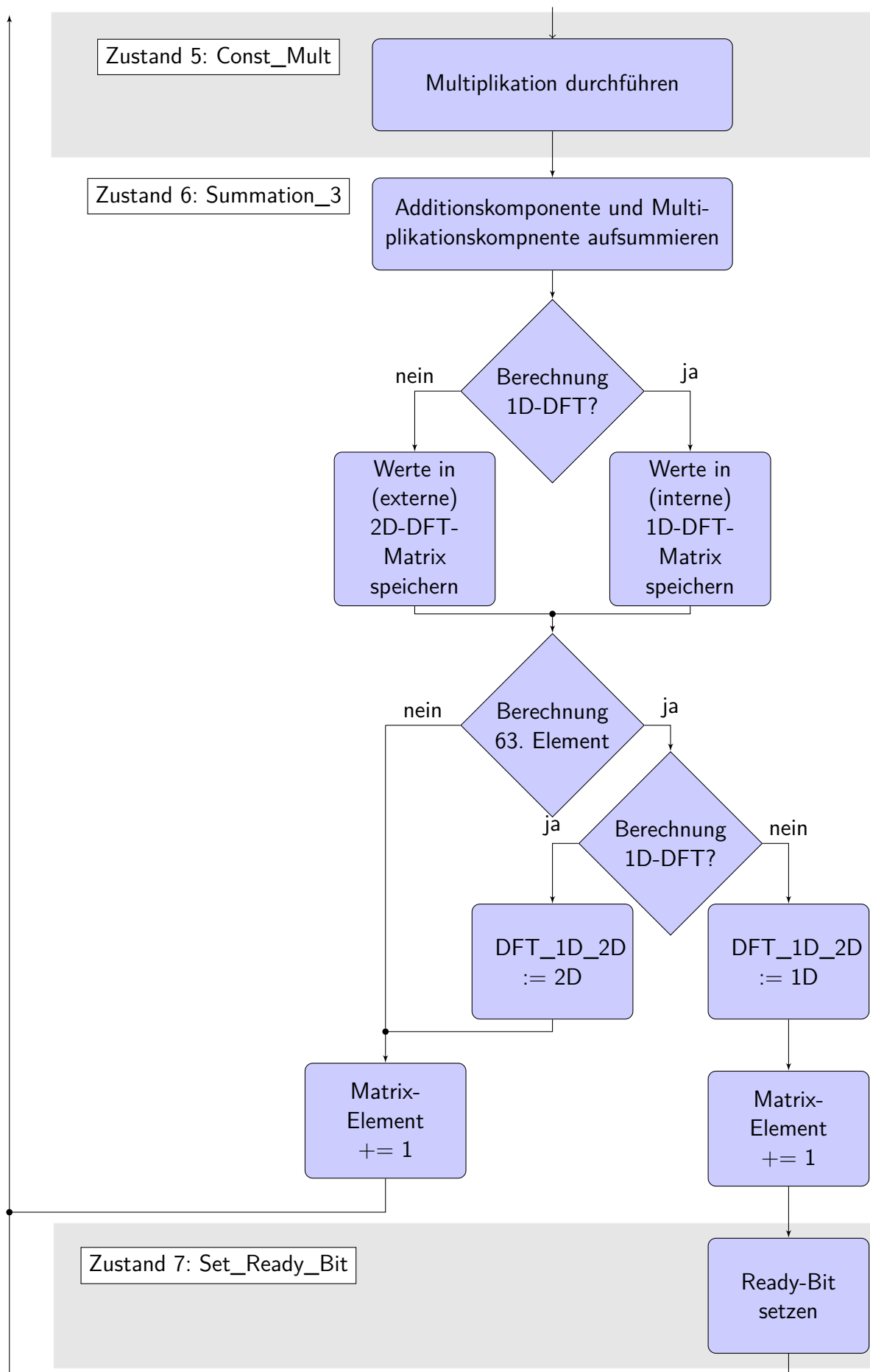
Abbildung 4.8: Automatengraf

## 4.5 UML-Diagramm









## 5 Evaluation

### 5.1 Simulation

#### 5.1.1 NC Sim - positive Zahlendarstellung

### 5.2 Anzahl benötigter Takte

Anhand der Simulation kann die Anzahl der vorausgesagten benötigten Takte verifiziert werden.

Nachdem `nReset` auf '1' gesetzt wird, werden die Eingangswerte eingelesen. Wenn dieser Vorgang abgeschlossen ist, geht `loaded` auf '1'. Mit der nächsten steigenden Taktflanke, in Bild 5.1 bei 340 ns, beginnt die Berechnung der 2D-DFT. Beendet ist sie, nachdem die Matrizenmultiplikation auf die Eingangswerte und anschließend auf die 1D-DFT-Werte angewandt wurde. Also nach  $2 \cdot 64$  einzelnen Berechnungen. Wenn dies erfolgt ist, wird `result_ready` auf '1' gesetzt. Dies geschieht bei 20 820 ns. Bei einer Taktfrequenz von  $(40 \text{ ns})^{-1}$  (siehe 8.17) ergeben sich so 512 Takte. Dies bestätigt auch der Edge Count, ebenfalls auf dem Bild zu sehen, welcher die Flanken des `clk`-Signals zählt. In der Simulation ist zu erkennen, dass die Berechnung der Elemente unterschiedlich viele Takte beansprucht. Hieran lässt sich ebenfalls sehen, dass die 1. (ungerade) Zeile weniger Takte gegenüber der 2. (geraden) Zeile benötigt.

### 5.3 Zeitabschätzung im Einsatz als ABS-Sensor

Anhand der nun bekannten Größe von 512 Takten kann ermittelt werden, ob diese Implementation vom zeitlichen Aspekt her akzeptabel ist. Da ein Einsatzszenario der ABS-Sensor ist, wird an dieser Stelle ein Blick hierauf geworfen. Da der ABS-Sensor an der Radnabe sitzt, wird hierfür die Raddrehzahl benötigt. Um diese zu ermitteln, wird von einer maximalen Geschwindigkeit von  $v_{max} = 250 \text{ KM/h}$  ausgegangen. Weiter wird ein relativ kleiner Reifenumfang von ca. 1 m angenommen. Als maximale Taktfrequenz des Sensors ist 1 MHz vorgegeben.

Der Reifen hat eine Breite von 175 cm, eine Flankenhöhe von 75 % der Breite und die Felge einen Durchmesser von 14 Zoll. Somit errechnet sich der Reifenumfang gemäß (5.1)

$$\begin{aligned} U &= (175 \text{ cm} \cdot 75\% \cdot 2 + 14 \cdot 2,54 \text{ cm}) \cdot \pi \\ &\simeq 0,94 \text{ m} \end{aligned} \tag{5.1}$$

In Gleichung 5.2 wird die Anzahl der Radumdrehungen bei maximaler Geschwindigkeit berechnet

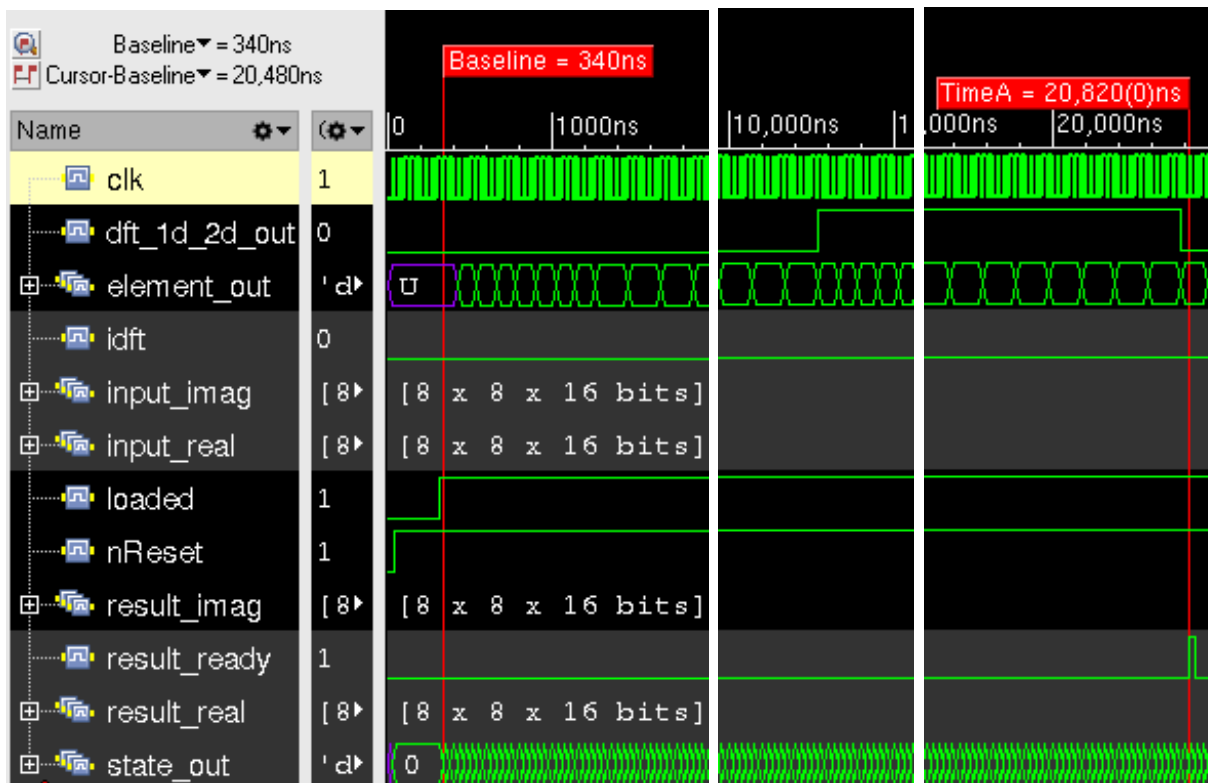


Abbildung 5.1: Ausschnitt des Simulationstools NCsim von der Berechnung und Verifikation der 2D-DFT.

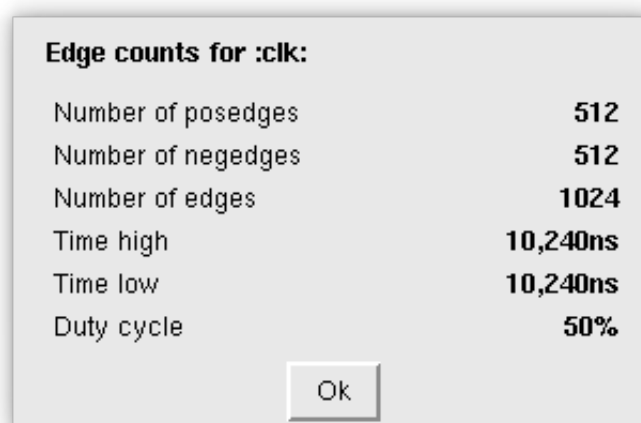


Abbildung 5.2: Edge Count des Taktsignals für die vollständige Simulation der 2D-DFT.

$$\begin{aligned}
 RPM &= \frac{\frac{250 \text{ Km/h}}{0,94 \text{ m}}}{60 \text{ sec}} \\
 &= 4386 \frac{\text{U}}{\text{min}} \\
 &= 73 \frac{\text{U}}{\text{sec}}
 \end{aligned} \tag{5.2}$$

Durch die Taktfrequenz und die benötigten Takte kann in (5.3) die maximale Anzahl der 2D-DFTs pro Sekunde errechnet werden.

$$\begin{aligned}
 N_{DFT,sec} &= \frac{100 \text{ MHz}}{512 \text{ Takte}} \\
 &= 195312
 \end{aligned} \tag{5.3}$$

Somit ist es nun möglich die unter diesen Voraussetzungen maximale Zahl der 2D-DFTs während einer Umdrehung zu bestimmen (5.4)

$$\begin{aligned}
 N_{DFT,U} &= \frac{195\,312 \frac{2D-DFT}{\text{sec}}}{73 \frac{\text{U}}{\text{sec}}} \\
 &= 2675 \frac{2D - DFT}{\text{U}}
 \end{aligned} \tag{5.4}$$

Nun kann in (5.5) gezeigt werden, dass bei einer Winkelauflösung von  $1^\circ$  knapp 7,5 2D-DFTs berechnet werden könnten. Die Dauer liegt somit gut im zeitlichen Rahmen, der vorganden ist. Darüber hinaus kann an dieser Stelle bereits gesagt werden, dass noch reichlich Zeit für andere Berechnungen vorhanden ist.

$$\begin{aligned}
 N_{DFT,1^\circ} &= \frac{2675 \frac{2D - DFT}{\text{U}}}{360^\circ} \\
 &= 7,43 \frac{2D - DFT}{1^\circ}
 \end{aligned} \tag{5.5}$$

Um eine Aussage über die restliche zur Verfügung stehenden Zeit bzw. Takte machen zu können, wird in Gleichung (5.6) gezeigt, dass pro Winkel etwa 3800 Takte für Berechnungen zu Verfügung stehen. Somit ist gezeigt, dass für andere Aufgaben ausreichen Zeit vorhanden ist und die Implemenatation erfolgreich ist.

$$\begin{aligned} N_{Takte,U} &= \frac{100 \text{ MHz}}{73 \frac{U}{sec}} \\ &= 1,37 \cdot 10^6 \frac{Takte}{Umdrehung} \\ N_{Takte,1^\circ} &= \frac{1,37 \cdot 10^6 \frac{Takte}{Umdrehung}}{360^\circ} \\ &\simeq 3800 \text{ Takte} \end{aligned} \tag{5.6}$$

Da 512 etwa 13,5% von 3800 sind, resultiert hieraus, dass noch etwa 86,5% bzw. knapp 3300 Takte nutzbar sind.

## 5.4 Testumgebung

### 5.4.1 Struktogramm des Testablaufs

### 5.4.2 Reale Eingangswerte

## 5.5 Chipdesign

### 5.5.1 Anzahl Standardzellen

Benötigte Standardzellen für 1D / 2D

Benötigte Standardzellen bei 3 Lagen / 4 Lagen

### 5.5.2 Visualisierung der Netzliste

### 5.5.3 Floorplan, Padring

## **6 Schlussfolgerungen**

### **6.1 Zusammenfassung**

### **6.2 Bewertung und Fazit**

Es konnte eine effiziente Berechnung implementiert werden, die der FFT in nichts nachsteht. Wenn nicht die Ausgangssituation gewesen wäre, dass eine möglichst flexibel gehaltene Matrixmultiplikation erstrebenswert ist, hätte auch eine FFT, dessen Berechnungsvorschrift bekannt ist, implementiert werden können. Für DFT anderer Größe als  $2^N$  gilt dies nicht.

### **6.3 Ausblick**

## 7 Abkürzungsverzeichnis

<b>1D-DFT</b>	eindimensionale diskrete Fouriertransformation
<b>2D-DFT</b>	zweidimensionale diskrete Fouriertransformation
<b>ADC</b>	Analog Digital Converter
<b>ADU</b>	Analog Digital Umsetzer
<b>AMR</b>	anisotroper magnetoresistiver Effekt
<b>ASIC</b>	Application Specific Integrated Circuit, <i>dt.: Anwendungsspezifischer Integrierter Schaltkreis</i>
<b>DCT</b>	diskrete Kosinus Transformation
<b>DFT</b>	diskrete Fouriertransformation
<b>FFT</b>	Fast Fouriertransformation
<b>FT</b>	Fouriertransformation
<b>IDFT</b>	inverse diskrete Fouriertransformation
<b>ISAR</b>	Integrated Sensor Array
<b>LSB</b>	Least Significant Bit
<b>MSB</b>	Most Significant Bit
<b>RAM</b>	Random Access Memory
<b>TMR</b>	tunnelmagnetoresistiver Effekt



# Abbildungsverzeichnis

2.1	Interpretation von Dualzahlen im SQ3-Format . . . . .	3
2.2	Veranschaulichung der Matrixmultiplikation . . . . .	5
2.3	Einheitskreis, Zusammensetzung des komplexen Zeigers aus Sinus und Kosinus	6
2.4	Veranschaulichung der Berechnung der DFT mit reellen Eingangswerten . . . .	10
2.5	Redundante Werte der spaltenweisen DFT einer 8x8-Matrix. Der Imaginärteil der redundanten Werte hat den selben Betrag mit negiertem Vorzeichen. . . .	11
3.1	Einheitskreis mit relevanten Werten der 8x8-DFT . . . . .	15
3.2	Twiddlefaktoren der 8x8-Matrix, aufgeteilt auf die Laufindizes $m$ und $n$ . $m$ bezieht sich auf das Element im Ausgangsvektor $\vec{X}$ , $n$ auf den Eingangsvektor $\vec{x}$ . Siehe auch Gl. (2.11). . . . .	16
3.3	Matrix-Darstellung der 8x8-DFT-Twiddlefaktoren aufgeteilt nach Real- und Imaginärteil. . . . .	16
3.4	Berechnungsschema der DFT mit 8 Eingangswerten nach dem Butterfly-Verfahren . . . . .	20
4.1	Darstellung der Berechnung der 2D-DFT aus Gleichung (4.2) . . . . .	25
4.2	Vorgehensweise der Akkumulation der ungeraden Spalten der Eingangswerte. .	27
4.3	Vorgehensweise der Akkumulation der geraden Spalten der Eingangswerte. . .	28
4.4	Um von der DFT zur IDFT zu kommen, müssen bei der Matrixmultiplikation die Zeilen 2 und 8, 3 und 7 sowie 4 und 6 der Twiddlefaktormatrix vertauscht werden. . . . .	29
4.5	Schaltnetz des 13 Bit Konstantenmultiplizierers für $\frac{\sqrt{2}}{2} = 0.70711 \simeq 0.70703125 = 0001011010100_2$ in Encounter; Eingang links, Ausgang rechts .	31
4.6	Schaltnetz einer Einheit zur Bildung des 2er-Komplements eines 13 Bit Vektors; Eingang links, Ausgang rechts . . . . .	32
4.7	Schaltnetz eines 12 Bit Addierers, Eingänge links (12 Bit), Ausgang rechts (13 Bit)	32
4.8	Automatengraf . . . . .	34
5.1	Ausschnitt des Simulationstools NCSim von der Berechnung und Verifikation der 2D-DFT. . . . .	39
5.2	Edge Count des Taktsignals für die vollständige Simulation der 2D-DFT. . . .	39

# Tabellenverzeichnis

3.1	Bewertung der DCT-Twiddlefaktor-Matrizen . . . . .	14
3.2	Bewertung der DFT-Twiddlefaktor-Matrizen . . . . .	15
3.3	Auflistung benötigter reeller Multiplikationen der verschiedenen Methoden für die 2D-DFT . . . . .	21
4.1	Takte für die komplexe DFT . . . . .	24
4.2	Vergleich . . . . .	33

# Literatur

- [1] M. Krey, „Systemarchitektur und Signalverarbeitung für die Diagnose von magnetischen ABS-Sensoren“, *test*, 2015.

## 8 Anhang

### 8.1 Skript zur Bewertung von Twiddlefaktormatrizen

```
1 %% Dateiname: dct_bewertung.m
2 %% Funktion: Bewertet die Koeffizienten der DCT-Twiddlefaktormatrix
3 %%           darauf basierend, wie trivial die Berechnungen mit
4 %%           den Twiddlefaktoren sind.
5 %%           Als trivial gelten Berechnungen mit den Werten -1, -0.5, 0,
6 %%           +0.5, +1
7 %%           Es wird ein Verhaeltnis aus trivialen und nicht trivialen
8 %%           Werten
9 %%           erstellt.
10 %% Argumente: N (Groesse der NxN DCT-Matrix)
11 %% Author:    Thomas Lattmann
12 %% Datum:     17.10.2017
13 %% Version:   1.0

14 function dct_bewertung(N)

15 % Twiddlefaktor-Matrix erzeugen
16 W = cos(pi/N*([0:N-1]')*([0:N-1]+.5));
17 W = round(W*1000000)/1000000;

18 % Werte kleiner 0,000001 auf 0 setzen (arithmetische Ungenauigkeiten)
19 W(abs(W) < 0.000001) = 0;

20
21
22 % Anzahl verschiedener Werte ermitteln
23 different_nums = unique(W);
24 different_non_trivial_nums = different_nums(find(different_nums ~= 1));
25 different_non_trivial_nums = different_non_trivial_nums(find(
26     different_non_trivial_nums ~= -1));
27 different_non_trivial_nums = different_non_trivial_nums(find(
28     different_non_trivial_nums ~= 0.5));
29 different_non_trivial_nums = different_non_trivial_nums(find(
30     different_non_trivial_nums ~= -0.5));
31 different_non_trivial_nums = different_non_trivial_nums(find(
32     different_non_trivial_nums ~= 0));
33
34 different_non_trivial_nums = unique(abs(different_non_trivial_nums));
35 different_non_trivial_nums
36 %non_trivial = length(abs(different_non_trivial_nums))
```

```

% Jeweils die Menge der verschiedenen Werte ermitteln
37 num_count = zeros(1, length(different_nums));
    for k = 1:length(different_nums)
39         for n = 1:N
                for m = 1:N
41                     if different_nums(k) == W(m,n)
                        num_count(k) = num_count(k) + 1;
43                     end
                end
45         end
    end
47

% nicht triviale Werte der Matrix z hlen
49 nontrivial_nums = 0;
    for k = 1:length(different_nums)
        if abs(different_nums(k)) != 1
51             if abs(different_nums(k)) != 0.5
53                 if different_nums(k) != 0
55                     nontrivial_nums = nontrivial_nums + num_count(k);
                    end
57             end
        end
59    end

61    nums_of_matrix = N*N;

63    trivial_nums = N*N - nontrivial_nums

65    nontrivial_nums

67    v = trivial_nums/nontrivial_nums

69 end

```

Listing 8.1: Octave-Skript zur Bewertung unterschiedlicher DCT-Twiddlefaktormatrizen

```

1 %% Dateiname: dft_bewertung.m
  %% Funktion: Bewertet die Koeffizienten der DFT-Twiddlefaktormatrix
3 %%           darauf basierend, wie trivial die Berechnungen mit
  %%           den Twiddlefaktoren sind.
5 %%           Als trivial gelten Berechnungen mit den Werten -1, -0.5, 0,
  %%           +0.5, +1
  %%           Es wird ein Verhaeltnis aus trivialen und nicht trivialen
  %%           Werten
7 %%           erstellt.
  %% Argumente: N (Groesse der NxN DFT-Matrix)
9 %% Author:    Thomas Lattmann
  %% Datum:     17.10.2017
11 %% Version:   1.0

13 function dft_bewertung(N)

```

```

15 % Twiddlefaktor-Matrix erzeugen
W = exp(-i*2*pi*[0:N-1]'/[0:N-1]/N);
17 W = round(W*1000000)/1000000;

19 % Matrix nach Im und Re trennen und Werte runden
W_r = real(W);
21 W_i = imag(W);

23 % Werte kleiner 0,000001 auf 0 setzen (arithmetische Ungenauigkeiten)
W_r(abs(W_r) < 0.000001) = 0;
25 W_i(abs(W_i) < 0.000001) = 0;

27

29 % Anzahl verschiedener Werte ermitteln
different_nums_real = unique(W_r);
31 different_nums_imag = unique(W_i);

33 different_nums = [different_nums_real; different_nums_imag];
different_nums = unique(different_nums);
35 different_non_trivial_nums = different_nums(find(different_nums ~= 1));
different_non_trivial_nums = different_non_trivial_nums(find(
    different_non_trivial_nums ~= -1));
37 different_non_trivial_nums = different_non_trivial_nums(find(
    different_non_trivial_nums ~= 0.5));
different_non_trivial_nums = different_non_trivial_nums(find(
    different_non_trivial_nums ~= -0.5));
39 different_non_trivial_nums = different_non_trivial_nums(find(
    different_non_trivial_nums ~= 0));

41 different_non_trivial_nums = unique(abs(different_non_trivial_nums));
non_trivial = length(abs(different_non_trivial_nums))

43

45 % Jeweils die Menge der verschiedenen Werte ermitteln (hier Re)
num_count_real = zeros(1, length(different_nums_real));
for k = 1:length(different_nums_real)
47     for n = 1:N
49         for m = 1:N
            if different_nums_real(k) == W_r(m,n)
                num_count_real(k) = num_count_real(k) + 1;
51         end
53     end
end

55

57 % Jeweils die Anzahl der verschiedenen Werte ermitteln (hier Im)
num_count_imag = zeros(1, length(different_nums_imag));
59 for k = 1:length(different_nums_imag)
    for n = 1:N

```

```

61     for m = 1:N
62         if different_nums_imag(k) == W_i(m,n)
63             num_count_imag(k) = num_count_imag(k) + 1;
64         end
65     end
66 end
67 end
68
69 % nicht triviale Werte der reellen Matrix z hlen
70 nontrivial_nums_real = 0;
71 for k = 1:length(different_nums_real)
72     if abs(different_nums_real(k)) != 1
73         if abs(different_nums_real(k)) != 0.5
74             if different_nums_real(k) != 0
75                 nontrivial_nums_real = nontrivial_nums_real + num_count_real(k);
76             end
77         end
78     end
79 end
80 end
81
82 % nicht triviale Werte der imaginären Matrix z hlen
83 nontrivial_nums_imag = 0;
84 for k = 1:length(different_nums_imag)
85     if abs(different_nums_imag(k)) != 1
86         if abs(different_nums_imag(k)) != 0.5
87             if different_nums_imag(k) != 0
88                 nontrivial_nums_imag = nontrivial_nums_imag + num_count_imag(k);
89             end
90         end
91     end
92 end
93
94 nums_of_each_matrix = N*N;
95
96 trivial_nums_real = N*N - nontrivial_nums_real
97 trivial_nums_imag = N*N - nontrivial_nums_imag
98
99 nontrivial_nums_real
100 nontrivial_nums_imag
101
102 trivial_nums_total = trivial_nums_real + trivial_nums_imag
103 nontrivial_nums_total = nontrivial_nums_real + nontrivial_nums_imag
104
105 v = trivial_nums_total/nontrivial_nums_total
106
107 end

```

Listing 8.2: Octave-Skript zur Bewertung unterschiedlicher DFT-Twiddlefaktormatrizen

## 8.2 Gate-Report des 12 Bit Konstantenmultiplizierers

```

1 rc:/> report gates

```

---

```

3   Generated by:      Encounter(R) RTL Compiler RC14.25 - v14.20-s046_1
4   Generated on:      May 30 2017  03:29:41 pm
5   Module:           multiplier
6   Technology library: c35_CORELIB_TYP 3.02
7   Operating conditions: _nominal_ (balanced_tree)
8   Wireload mode:     enclosed
9   Area mode:         timing library

```

---

```

11

```

Gate	Instances	Area	Library
ADD21	5	728.000	c35_CORELIB_TYP
AOI210	2	145.600	c35_CORELIB_TYP
AOI220	18	1638.000	c35_CORELIB_TYP
CLKIN0	6	218.400	c35_CORELIB_TYP
IMUX20	38	3458.000	c35_CORELIB_TYP
INV0	27	982.800	c35_CORELIB_TYP
NAND20	12	655.200	c35_CORELIB_TYP
NOR20	8	436.800	c35_CORELIB_TYP
OAI220	6	546.000	c35_CORELIB_TYP
XNR20	15	1638.000	c35_CORELIB_TYP
XNR30	6	1201.200	c35_CORELIB_TYP
XNR31	3	600.600	c35_CORELIB_TYP
XOR20	5	637.000	c35_CORELIB_TYP
<b>total</b>	<b>151</b>	<b>12885.600</b>	

```

29

```

Type	Instances	Area	Area %
inverter	33	1201.200	9.3
logic	118	11684.400	90.7
<b>total</b>	<b>151</b>	<b>12885.600</b>	<b>100.0</b>

```

37
39 rc:/>

```

Listing 8.3: RC Gate-Report

## 8.3 Twiddlefaktormatrix im S1Q10-Format

```

1 %% Dateiname:      twiddle2file.m
  %% Funktion:       Erzeugt eine Datei mit den binären komplexen

```



```

3  %%                               Twiddlefaktoren
  %% Argumente:                   N (Groesse der NxN DFT-Matrix)
5  %% Aufbau der Datei:           Wie die Matrix, enthaelt Realteil und Imaginaerteil.
  %%                               Alle Werte sind wie im Beispiel durch Leerzeichen
  getrennt:
7  %%                               Re{W(1,1)} Im{W(1,1)} Re{W(1,2)} Im{W(1,2)}
  %%                               Re{W(2,1)} Im{W(2,1)} Re{W(2,2)} Im{W(2,2)}
9  %% Abhaenigkeiten:            (1) twiddle_coefficients.m
  %%                               (2) dec_to_slq10.m
11 %%                               (3) bit_vector2integer.m
  %%                               (4) zweier_komplement.m
13 %% Author:                     Thomas Lattmann
  %% Datum:                      02.11.17
15 %% Version:                    1.0

17 function twiddle2file(N)

19 % Dezimale Twiddlefaktormatrix erstellen
  W_dec = twiddle_coefficients(N);
21  W_dec_real = real(W_dec);
  W_dec_imag = imag(W_dec);
23
25  W_bin_int_real = zeros(size(W_dec_real));
  W_bin_int_imag = zeros(size(W_dec_imag));

27  for m = 1:N
    for n = 1:N
29      bit_vector = dec_to_slq10(W_dec_real(m,n));
      W_bin_int_real(m,n) = bit_vector2integer(bit_vector);

31      bit_vector = dec_to_slq10(W_dec_imag(m,n));
      W_bin_int_imag(m,n) = bit_vector2integer(bit_vector);
33    end
35  end

37  fid=fopen('Twiddle_slq10_komplex.txt', 'w+');

39  for m=1:N
    for n=1:N
41      fprintf(fid, '%012d ', W_bin_int_real(m,n));
      fprintf(fid, '%012d', W_bin_int_imag(m,n));
43      if n < N
        fprintf(fid, ' ');
45      end
    end
47    if m < N
      fprintf(fid, '\n');
49    end
51  end

  fclose(fid);

```

```

53 end

```

Listing 8.4: Erstellen der Twiddlefaktormatrix-Datei

```

%% Dateiname: twiddle_coefficients.m
2 %% Funktion:  Erstellt eine Matrix (W) mit den Twiddlefaktoren fuer die DFT
    der
%%
    Groesse, die mit N an das Skript uebergeben wurde.
4 %% Argumente: N (Groesse der NxN DFT-Matrix)
%% Author:    Thomas Lattmann
6 %% Datum:    02.11.17
%% Version:   1.0

8
function W = twiddle_coefficients(N)
10
    % Twiddlefaktoren fuer die DFT
12    W = exp(-i*2*pi*[0:N-1]'*[0:N-1]/N)

14    % auf 6 Nachkommastellen reduzieren
    W = round(W*1000000)/1000000;
16

18    % negative Nullen auf 0 setzen
    W_real = real(W);
    W_imag = imag(W);
20    W_real(abs(W_real)<00000.1) = 0;
    W_imag(abs(W_imag)<00000.1) = 0;
22    W = W_real + i*W_imag;

24 end

```

Listing 8.5: Erzeugen der Twiddlefaktormatrix

```

%% Dateiname:    dec_to_slq10.m
2 %% Funktion:    Konvertiert eine Dezimalzahl in das binaere S1Q10-Format
%% Argumente:    Dezimalzahl im Bereich von -2...+2-1/2^10
4 %% Abhaengigkeiten: (1) zweier_komplement.m
%% Author:    Thomas Lattmann
6 %% Datum:    02.11.17
%% Version:   1.0

8
function bit_vector = dec_to_slq10(val)
10

12    bit_width=12;
    bit_vector=zeros(1,bit_width);
    dec_temp=0;
14    val_abs=abs(val);
    val_int=floor(val_abs);
16    val_frac=val_abs-val_int;

18    if val > 2-1/2^(bit_width-2) % 1.99902... bei 12 Bit und somit 10 Bit
        fuer Nachkomma

```

```

    disp('Diese Zahl kann nicht im s1q11-Format dargestellt werden.')
20 elseif val < -2
    disp('Diese Zahl kann nicht im s1q11-Format dargestellt werden.')
22 else

    % Vorkommastellen
    if abs(val) >= 1
24         bit_vector(2) = 1;
26         if val == -2
28             bit_vector(1) = 1;
            end
30     end

    % Nachkommastellen
    for k = 1:bit_width-2
32         % berechnen der Differenz des Twiddlefaktors und des derzeitigen
34         % Wertes der Binaerzahl
        d = val_frac - dec_temp;
36         if d >= 1/2^k
            bit_vector(k+2) = 1;
38             dec_temp = dec_temp+1/2^k;
            end
40     end

    % 2er-Komplement bilden, falls val negativ
    if val < 0
42         bit_vector=zweier_komplement(bit_vector);
44     end
46 end
end

```

Listing 8.6: Dezimalzahl nach S1Q10 konvertieren

```

1 %% Dateiname: zweier_komplement.m
2 %% Funktion:  Bilden des 2er-Komplements eines "Bit"-Vektors
3 %% Argumente: Vektor aus Nullen und Einsen
4 %% Author:    Thomas Lattmann
5 %% Datum:     02.11.17
6 %% Version:   1.0
7
8 function bit_vector = zweier_komplement(bit_vector)
9     bit_width=length(bit_vector);
10
11     for j = 1:bit_width
12         bit_vector(j) = not(bit_vector(j));
13     end
14     bit_vector(bit_width) = bit_vector(bit_width) + 1;
15     for j = 1:bit_width-1
16         if bit_vector(bit_width -j +1) == 2
17             bit_vector(bit_width -j +1) = 0;
18             bit_vector(bit_width -j) = bit_vector(bit_width -j) + 1;
19         end

```

```

    end
21 end

```

Listing 8.7: Bildung des 2er-Komplements

```

1 %% Dateiname: bit_vector2integer.m
  %% Funktion: Wandelt einen Vektor von Zahlen in eine einzelne Zahl (
    Integer)
3 %%           Beispiel: [0 1 1 0 0 1] => 11001
  %%           Um fuehrende Nullen zu erhalten muss z.B. printf('%06d',
    Integer)
5 %%           genutzt werden. Hierbei wird vorne mit Nullen aufgefuellt,
    wenn
  %%           'Integer' weniger als 6 stellen hat.
7 %% Argumente: Vektor (aus Nullen und Einsen)
  %% Author:    Thomas Lattmann
9 %% Datum:     02.11.17
  %% Version:   1.0
11
function bin_int = bit_vector2integer(bit_vector)
13
    bin_int=0;
15    bit_width=length(bit_vector);

17    % Konvertierung von Vektor nach Integer
    for l = 1:bit_width
19        bin_int = bin_int + bit_vector(bit_width - l + 1)*10^(l-1);
    end
21
end

```

Listing 8.8: Binär-Vektor in Binär-Integer umwandeln

```

  %% Dateiname: s1q10_to_dec.m
2 %% Funktion: Konvertiert eine binaere Zahl im S1Q10-Format als Dezimalzahl
  %% Argumente: Vektor aus Nullen und Einsen
4 %% Author:    Thomas Lattmann
  %% Datum:     02.11.17
6 %% Version:   1.0

8 function dec = s1q10_to_dec(bit_vector)

10    % Dezimalzahl aus s1q10 Binaerzahl berechnen

12    bit_width=length(bit_vector);
    dec = 0;

14
    if bit_vector(1) == 1
16        dec = -2;
        if bit_vector(2) == 1
18            dec = -1;
        end
    end

```

```

20  elseif bit_vector(2) == 1
      dec = 1;
22  end

24  for n = 3:bit_width
      if bit_vector(n) == 1
26      dec = dec + 1/2^(n-2);
      end
28  end
end

```

Listing 8.9: Kontroll-Skript für S1Q10 nach Dezimal

## 8.4 Programmcode

```

1  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;

3

5
   package constants is
7     constant mat_size : integer;
     constant bit_width_extern : integer;
9     constant bit_width_adder : integer;
     constant bit_width_multiplier : integer;
11  end constants;

13  package body constants is
     constant mat_size : integer := 8;
15     constant bit_width_extern : integer := 13;
     constant bit_width_adder : integer := bit_width_extern+1;
17     constant bit_width_multiplier : integer := bit_width_adder*2;

19  end constants;

```

Listing 8.10: Deklaration der Konstanten

```

— Package, welches ein 2D-Array bereitstellt.
2 — Das 2D-Array besteht aus 1D-Arrays, dies bringr gegenueber der direkten
   Erzeugung (m,n) statt (m)(n) den Vorteil, dass
— dass zeilen- sowie spaltenweise zugewiesen werden kann. Sonst waere nur
   die komplette Matrix oder einzelne Elemente moeglich.

4
   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
   use ieee.numeric_std.all;
8   library work;
   use work.all;
10  use constants.all;

```

```

12 package datatypes is
14     type t_1d_array is array(integer range 0 to mat_size-1) of signed(
        bit_width_extern-1 downto 0);
        type t_2d_array is array(integer range 0 to mat_size-1) of t_1d_array;

16
        type t_1d_array6_13bit is array(integer range 0 to 5) of signed(
            bit_width_adder-1 downto 0);

18
20     subtype t_twiddle_coeff_long is signed(16 downto 0);
        constant twiddle_coeff_long : t_twiddle_coeff_long := "
0010110101000010";
22     subtype t_twiddle_coeff is signed(bit_width_adder-1 downto 0);
        --constant twiddle_coeff : t_twiddle_coeff := twiddle_coeff_long(16
        downto 16-(bit_width_adder-1));

24
26
28     -- Zustandsautomat 1D-DFT
        subtype t_dft8_states is std_logic_vector(2 downto 0);
30     constant idle : t_dft8_states := "000";
        constant twiddle_calc : t_dft8_states := "001";
32     constant additions_stage1 : t_dft8_states := "010";
        constant additions_stage2 : t_dft8_states := "011";
34     constant const_mult : t_dft8_states := "100";
        constant additions_stage3 : t_dft8_states := "101";
36     constant set_ready_bit : t_dft8_states := "110";

38 end datatypes;

```

Listing 8.11: Deklaration eigener Datentypen

```

library IEEE;
2 use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;
4 use ieee.numeric_std.all;

6 library STD; -- for reading text file
use STD.TEXTIO.ALL;
8 use ieee.std_logic_textio.all;

10 library work;
use work.all;
12 use datatypes.all;
use constants.all;

14

16 entity read_input_matrix is
    port(

```

```

18         clk           : in  bit;
19         loaded        : out bit;
20         input_real    : out t_2d_array;
21         input_imag    : out t_2d_array
22     );
23 end entity read_input_matrix;
24
25
26 architecture bhv of read_input_matrix is
27 begin
28     reading : process
29
30         variable element_1_real : std_logic_vector(bit_width_extern-1
31         downto 0) := (others => '0');
32         variable element_1_imag : std_logic_vector(bit_width_extern-1
33         downto 0) := (others => '0');
34         variable element_2_real : std_logic_vector(bit_width_extern-1
35         downto 0) := (others => '0');
36         variable element_2_imag : std_logic_vector(bit_width_extern-1
37         downto 0) := (others => '0');
38         variable element_3_real : std_logic_vector(bit_width_extern-1
39         downto 0) := (others => '0');
40         variable element_3_imag : std_logic_vector(bit_width_extern-1
41         downto 0) := (others => '0');
42         variable element_4_real : std_logic_vector(bit_width_extern-1
43         downto 0) := (others => '0');
44         variable element_4_imag : std_logic_vector(bit_width_extern-1
45         downto 0) := (others => '0');
46         variable element_5_real : std_logic_vector(bit_width_extern-1
47         downto 0) := (others => '0');
48         variable element_5_imag : std_logic_vector(bit_width_extern-1
49         downto 0) := (others => '0');
50         variable element_6_real : std_logic_vector(bit_width_extern-1
51         downto 0) := (others => '0');
52         variable element_6_imag : std_logic_vector(bit_width_extern-1

```

```

variable textfilename : string(1 to 29);

54
begin
56

58
    if bit_width_extern = 12 then
60        textfilename := "InputMatrix_komplex_12Bit.txt";
    else
62        textfilename := "InputMatrix_komplex_16Bit.txt";
    end if;

64
    file_open(fstatus, infile, textfilename, read_mode);

66
    if fstatus = NAME_ERROR then
68        file_open(fstatus, infile, "HDL/InputMatrix_komplex.txt",
read_mode);
    --
    --report "Ausgabe-Datei befindet sich im Unterverzeichnis 'HDL
    '.'";
70
    end if;

72
    for i in 0 to mat_size-1 loop

74
        wait until clk = '1' and clk'event;
        readline(infile, inline);
76
        read(inline, element_1_real);
        read(inline, r_space);
78
        read(inline, element_1_imag);
        read(inline, r_space);
80
        read(inline, element_2_real);
        read(inline, r_space);
82
        read(inline, element_2_imag);
        read(inline, r_space);
84
        read(inline, element_3_real);
        read(inline, r_space);
86
        read(inline, element_3_imag);
        read(inline, r_space);
88
        read(inline, element_4_real);
        read(inline, r_space);
90
        read(inline, element_4_imag);
        read(inline, r_space);
92
        read(inline, element_5_real);
        read(inline, r_space);
94
        read(inline, element_5_imag);
        read(inline, r_space);
96
        read(inline, element_6_real);
        read(inline, r_space);
98
        read(inline, element_6_imag);
        read(inline, r_space);
100
        read(inline, element_7_real);
        read(inline, r_space);

```



```

102     read(inline , element_7_imag);
103     read(inline , r_space);
104     read(inline , element_8_real);
105     read(inline , r_space);
106     read(inline , element_8_imag);

108     input_real(i)(0) <= signed(element_1_real);
109     input_imag(i)(0) <= signed(element_1_imag);
110     input_real(i)(1) <= signed(element_2_real);
111     input_imag(i)(1) <= signed(element_2_imag);
112     input_real(i)(2) <= signed(element_3_real);
113     input_imag(i)(2) <= signed(element_3_imag);
114     input_real(i)(3) <= signed(element_4_real);
115     input_imag(i)(3) <= signed(element_4_imag);
116     input_real(i)(4) <= signed(element_5_real);
117     input_imag(i)(4) <= signed(element_5_imag);
118     input_real(i)(5) <= signed(element_6_real);
119     input_imag(i)(5) <= signed(element_6_imag);
120     input_real(i)(6) <= signed(element_7_real);
121     input_imag(i)(6) <= signed(element_7_imag);
122     input_real(i)(7) <= signed(element_8_real);
123     input_imag(i)(7) <= signed(element_8_imag);

124     if i = mat_size-1 then
125         loaded <= '1' after 10 ns;
126     end if;
127 end loop;
128 file_close(infile);
129 wait;

132 end process;
134 end bhv;

```

Listing 8.12: Eingangs-Matrix aus Textdatei einlesen

```

library ieee;
2 use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
4 library work;
  use work.all;
6 use datatypes.all;

8 entity read_input_matrix_tb is
  end entity read_input_matrix_tb;

10 architecture arch of read_input_matrix_tb is

12     signal clk          : bit := '0';
14     signal loaded       : bit := '0';
15     signal input_real   : t_2d_array;
16     signal input_imag   : t_2d_array;

```

```

18  component read_input_matrix is
      port(
20          clk          : in  bit;
          loaded         : out bit;
22          input_real   : out t_2d_array;
          input_imag    : out t_2d_array
24      );
  end component;

26  begin
28      dut : read_input_matrix
          port map(
30          clk          => clk ,
          loaded        => loaded ,
32          input_real  => input_real ,
          input_imag   => input_imag
34      );

36      clk <= not clk after 20 ns;
  end arch;

```

Listing 8.13: Testbench für das Einlesen aus einer Textdatei

```

library IEEE;
2  use ieee.std_logic_1164.all;
  —use ieee.std_logic_arith.all;
4  use ieee.numeric_std.all;

6  library STD; — for writing text file
  use STD.TEXTIO.ALL;
8  use ieee.std_logic_textio.all;

10 library work;
  use work.all;
12 use datatypes.all;
  use constants.all;
14

16  entity write_results is
18      port(
          result_ready : in  bit;
20          result_real  : in  t_2d_array;
          result_imag   : in  t_2d_array;
22          write_done   : out bit
      );
24  end entity write_results;

26

28  architecture bhv of write_results is
  begin

```

```

writing_to_file : process(result_ready)

30     variable fstatus : file_open_status; — status r,w
32     variable outline : line; — writeout line
34     file      outfile : text; — filehandle

36     —variable output1 : bit_vector(3 downto 0) := "0101";
37     —variable output2 : bit_vector(3 downto 0) := "0110";

38     variable element_1_real : std_logic_vector(bit_width_extern-1 downto 0)
39     ;
40     variable element_1_imag : std_logic_vector(bit_width_extern-1 downto 0)
41     ;
42     variable element_2_real : std_logic_vector(bit_width_extern-1 downto 0)
43     ;
44     variable element_2_imag : std_logic_vector(bit_width_extern-1 downto 0)
45     ;
46     variable element_3_real : std_logic_vector(bit_width_extern-1 downto 0)
47     ;
48     variable element_3_imag : std_logic_vector(bit_width_extern-1 downto 0)
49     ;
50     variable element_4_real : std_logic_vector(bit_width_extern-1 downto 0)
51     ;
52     variable element_4_imag : std_logic_vector(bit_width_extern-1 downto 0)
53     ;
54     variable element_5_real : std_logic_vector(bit_width_extern-1 downto 0)
55     ;
56     variable element_5_imag : std_logic_vector(bit_width_extern-1 downto 0)
57     ;
58     variable element_6_real : std_logic_vector(bit_width_extern-1 downto 0)
59     ;
60     variable element_6_imag : std_logic_vector(bit_width_extern-1 downto 0)
61     ;
62     variable element_7_real : std_logic_vector(bit_width_extern-1 downto 0)
63     ;
64     variable element_7_imag : std_logic_vector(bit_width_extern-1 downto 0)
65     ;
66     variable element_8_real : std_logic_vector(bit_width_extern-1 downto 0)
67     ;
68     variable element_8_imag : std_logic_vector(bit_width_extern-1 downto 0)
69     ;
70     variable space : character := ' ';

72     begin

74         file_open(fstatus, outfile, "/home/tlattmann/cadence/mat_mult/HDL/
Results.txt", write_mode);

76         —if result_ready = '1' then

78         for i in 0 to mat_size-1 loop

```

```
element_1_real := std_logic_vector(result_real(i)(0));
element_1_imag := std_logic_vector(result_imag(i)(0));
element_2_real := std_logic_vector(result_real(i)(1));
element_2_imag := std_logic_vector(result_imag(i)(1));
element_3_real := std_logic_vector(result_real(i)(2));
element_3_imag := std_logic_vector(result_imag(i)(2));
element_4_real := std_logic_vector(result_real(i)(3));
element_4_imag := std_logic_vector(result_imag(i)(3));
element_5_real := std_logic_vector(result_real(i)(4));
element_5_imag := std_logic_vector(result_imag(i)(4));
element_6_real := std_logic_vector(result_real(i)(5));
element_6_imag := std_logic_vector(result_imag(i)(5));
element_7_real := std_logic_vector(result_real(i)(6));
element_7_imag := std_logic_vector(result_imag(i)(6));
element_8_real := std_logic_vector(result_real(i)(7));
element_8_imag := std_logic_vector(result_imag(i)(7));
```

```
write(outline, element_1_real);
write(outline, space);
write(outline, element_1_imag);
write(outline, space);
write(outline, element_2_real);
write(outline, space);
write(outline, element_2_imag);
write(outline, space);
write(outline, element_3_real);
write(outline, space);
write(outline, element_3_imag);
write(outline, space);
write(outline, element_4_real);
write(outline, space);
write(outline, element_4_imag);
write(outline, space);
write(outline, element_5_real);
write(outline, space);
write(outline, element_5_imag);
write(outline, space);
write(outline, element_6_real);
write(outline, space);
write(outline, element_6_imag);
write(outline, space);
write(outline, element_7_real);
write(outline, space);
write(outline, element_7_imag);
write(outline, space);
write(outline, element_8_real);
write(outline, space);
write(outline, element_8_imag);
```

```
writeline(outfile, outline);
```

```
end loop;
```

```

114     write_done <= '1';
116     file_close(outfile);
118     --end if;
120 end process;
end bhv;

```

Listing 8.14: Ergebnis-Matrix in Textdatei schreiben

```

library IEEE;
2 use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
4
  library STD; -- for writing text file
6 use STD.TEXTIO.ALL;
  use ieee.std_logic_textio.all;
8
  library work;
10 use work.all;
  use datatypes.all;
12 use constants.all;
14
  entity write_test_tb is
16 end entity write_test_tb;
18
  architecture bhv of write_test_tb is
20
    signal clk          : bit;
22    signal loaded       : bit;
    signal result_ready : bit;
24    signal write_done   : bit;
    signal loop_running : bit;
26    signal loop_number  : signed(2 downto 0);
    signal input_real    : t_2d_array;
28    signal input_imag   : t_2d_array;
    signal output        : std_logic_vector(bit_width_extern-1 downto 0);
30
    component read_input_matrix
32      port(
        clk          : in  bit;
34        loaded       : out bit;
        input_real    : out t_2d_array;
36        input_imag   : out t_2d_array
      );
38 end component;

    component write_results
40      port(
42        result_ready : in  bit;

```

```

44     result_real   : in  t_2d_array;
45     result_imag   : in  t_2d_array;
46     write_done    : out bit;
47     loop_number   : out signed(2 downto 0);
48     loop_running  : out bit;
49     output        : out std_logic_vector(bit_width_extern-1 downto 0)
50 );
51 end component;
52
53 begin
54     mat : read_input_matrix
55     port map(
56         clk          => clk ,
57         loaded       => loaded ,
58         input_real   => input_real ,
59         input_imag   => input_imag
60     );
61
62     write : write_results
63     port map(
64         result_ready => result_ready ,
65         result_real  => input_real ,
66         result_imag  => input_imag ,
67         write_done   => write_done ,
68         loop_number  => loop_number ,
69         loop_running => loop_running ,
70         output       => output
71     );
72
73     result_ready <= loaded after 20 ns;
74     clk         <= not clk after 10 ns;
75
76 end bhv;

```

Listing 8.15: Testbench für das schreiben in eine Textdatei

```

library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
  use ieee.numeric_std.all;
4 library work;
  use work.all;
6 use datatypes.all;
  use constants.all;
8
10 library STD; — for reading text file
  use STD.TEXTIO.ALL;
12 use ieee.std_logic_textio.all;
14
15 entity dft8optimiert is
  port(

```

```

16     clk           : in  bit;
17     nReset        : in  bit;
18     loaded        : in  bit;
19     input_real     : in  t_2d_array;
20     input_imag     : in  t_2d_array;
21     result_real    : out t_2d_array;
22     result_imag    : out t_2d_array;
23     result_ready   : out bit;
24     idft          : in  bit;
25     state_out      : out t_dft8_states;
26     element_out    : out unsigned(5 downto 0);
27     dft_1d_2d_out  : out bit
28 );
29 end dft8optimiert;
30
31
32 architecture arch of dft8optimiert is
33
34     signal dft_state, next_dft_state : t_dft8_states;
35
36 begin
37
38     FSM_TAKT: process(clk)
39     begin
40         if clk='1' and clk'event then
41             dft_state <= dft_state;
42             state_out <= dft_state;
43             if nReset='0' then
44                 dft_state <= idle;
45                 state_out <= idle;
46             elsif loaded = '0' then
47                 dft_state <= idle;
48                 state_out <= idle;
49             elsif loaded='1' and dft_state = idle then
50                 dft_state <= twiddle_calc;
51                 state_out <= twiddle_calc;
52             else
53                 dft_state <= next_dft_state;
54                 state_out <= next_dft_state;
55             end if;
56         end if;
57     end process;
58
59
60     FSM_KOMB: process(dft_state)
61     --constant twiddle_coeff : signed(16 downto 0) := "00010110101000001";
62     variable twiddle_coeff : signed(bit_width_adder-1 downto 0);
63
64     variable mult_re, mult_im : signed(bit_width_multiplier-1 downto 0);
65
66

```

```

variable W_row, l_col : integer;
variable dft_1d_real, dft_1d_imag : t_2d_array;
variable matrix_real, matrix_imag : t_2d_array;
variable temp_re, temp_im : t_1d_array6_13bit;
variable temp14bit_re, temp14bit_im : signed(bit_width_adder downto 0);
variable dft_1d_2d : bit;
variable element : unsigned(5 downto 0) := "000000";

variable row_col_idx : integer := 0;

--variable LineBuffer : LINE;

begin
  twiddle_coeff := "0001011010100";
  -- Flip-Flops
  -- werden das 1. Mal sich selbst zu gewiesen, bevor sie einen Wert
  haben!
  result_ready <= '0';
  element := element;
  dft_1d_2d := dft_1d_2d;
  temp_re := temp_re;
  temp_im := temp_im;
  mult_re := mult_re;
  mult_im := mult_im;
  dft_1d_real := dft_1d_real;
  dft_1d_imag := dft_1d_imag;
  matrix_real := matrix_real;
  matrix_imag := matrix_imag;
  dft_1d_2d_out <= dft_1d_2d;

  -- Die Matrix hat 64 Elemente -> 2^6=64 -> 6-Bit Vektor passt genau.
  Ueberlauf = 1. Element vom naechsten Durchlauf.
  -- Der Elemente-Vektor kann darueber hinaus in vordere Haelfte = Zeile
  und hintere Haelfte = Spalte aufgeteilt werden.
  -- So laesst sich auch ein Matrix-Element mit zwei Indizes ansprechen:

  -- Bei der IDFT sind die Zeilen 1 und 7, 2 und 6, 3 und 5 vertauscht. 1
  und 4 bleiben wie sie sind.

  row_col_idx := to_integer(element(5 downto 3)); -- Wird bei der
  Twiddlefaktor-Matrix als Zeilen-, bei der Zwischen- und
  -- Ausgangsmatrix als
  Spaltenindex verwendet.

  if idft = '1' then
    if row_col_idx = 0 then
      W_row := 0;
    else

```



```

112     W_row := 8-row_col_idx;  — Twiddlefaktor-Matrix
    end if;
114 else
    W_row := row_col_idx;  — Twiddlefaktor-Matrix
116 end if;

118 l_col := to_integer(element(2 downto 0));  — Input-Matrix

120
122 if element = "000000" then
    if dft_1d_2d = '0' then
        matrix_real := input_real;
124        matrix_imag := input_imag;
    else
126        matrix_real := dft_1d_real;
        matrix_imag := dft_1d_imag;
128    end if;
end if;
130

132 case dft_state is
    when idle =>
134        next_dft_state <= twiddle_calc;

136        when twiddle_calc =>  — dft_state_out = 1
            — Mit resize werden die 12 Bit Eingangswerte vorzeichengerecht auf
            13 Bit erweitert, um um die richtige Groesse zu haben.
138            — Bei der Addition muessen die Summanden die gleiche Bit-Breite
            wie der Ergebnis-Vektor haben.
            case W_row is
                — Die Faktoren (Koeffizienten) der Twiddlefaktor-Matrix W lassen
                sich ueber  $\exp(-i \cdot 2 \cdot \pi \cdot [0:7]' \cdot [0:7]/8)$  berechnen.
                — 1. Zeile aus W -> nur Additionen
140                when 0 =>
                    — Die 1. Zeile aus W besteht nur aus den Faktoren (1+j0).
                    Daraus resultiert, dass die reellen
144                    — und die imaginaeren Werte der Eingangs-Matrix unabhaengig
                    von einander aufsummiert werden.
                    — Real
146                    temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
                    + resize(matrix_real(1)(l_col), bit_width_adder);
                    temp_re(1) := resize(matrix_real(2)(l_col), bit_width_adder)
                    + resize(matrix_real(3)(l_col), bit_width_adder);
148                    temp_re(2) := resize(matrix_real(4)(l_col), bit_width_adder)
                    + resize(matrix_real(5)(l_col), bit_width_adder);
                    temp_re(3) := resize(matrix_real(6)(l_col), bit_width_adder)
                    + resize(matrix_real(7)(l_col), bit_width_adder);
150                    — Imag
                    temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
                    + resize(matrix_imag(1)(l_col), bit_width_adder);

```

```

152         temp_im(1) := resize(matrix_imag(2)(l_col), bit_width_adder)
+ resize(matrix_imag(3)(l_col), bit_width_adder);
        temp_im(2) := resize(matrix_imag(4)(l_col), bit_width_adder)
+ resize(matrix_imag(5)(l_col), bit_width_adder);
154         temp_im(3) := resize(matrix_imag(6)(l_col), bit_width_adder)
+ resize(matrix_imag(7)(l_col), bit_width_adder);

156
158         -- 2. Zeile aus W besteht aus den Faktoren
        -- 0: ( 1.00000 + 0.00000i), 1: ( 0.70711 + 0.70711i), 2:
(0.00000 + 1.00000i), 3: (-0.70711 + 0.70711i),
        -- 4: (-1.00000 + 0.00000i), 5: (-0.70711 - 0.70711i), 6:
(0.00000 - 1.00000i), 7: ( 0.70711 - 0.70711i)

160
        -- Wegen der Faktoren (+/-0.70711 +/-0.70711i) haben die geraden
Zeilen (beginnend bei 1) 12 statt 8 Subtraktionen
162         -- Zunaechst werden die Werte aufsummiert, die mit dem Faktor 1 "
multipliziert" werden muessen.
        -- Dann werden die Werte aufsummiert, die mit 0,70711
multipliziert werden muessen. Um sowohl den Quelltext und
164         -- insbesondere auch den Platzbedarf auf dem Chip klein zuhalten,
wird die Multiplikation auf die Summe aller und
        -- nicht auf die einzelnen Werte angewandt.
166         -- Da immer genau die Haelfte der Faktoren positiv und die andere
negativ ist, werden die Eingangswerte so sortiert,
        -- dass keine Negationen noetig sind.
168         when 1 =>
            -- Real
170             temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
- resize(matrix_real(4)(l_col), bit_width_adder);
            temp_re(1) := resize(matrix_imag(2)(l_col), bit_width_adder)
- resize(matrix_imag(6)(l_col), bit_width_adder);
172             -- MultiPart
            temp_re(2) := resize(matrix_real(1)(l_col), bit_width_adder)
- resize(matrix_real(3)(l_col), bit_width_adder);
174             temp_re(3) := resize(matrix_imag(1)(l_col), bit_width_adder)
- resize(matrix_imag(7)(l_col), bit_width_adder);
            temp_re(4) := resize(matrix_imag(3)(l_col), bit_width_adder)
- resize(matrix_real(5)(l_col), bit_width_adder);
176             temp_re(5) := resize(matrix_real(7)(l_col), bit_width_adder)
- resize(matrix_imag(5)(l_col), bit_width_adder);
            -- Imag
178             temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
- resize(matrix_real(2)(l_col), bit_width_adder);
            temp_im(1) := resize(matrix_real(6)(l_col), bit_width_adder)
- resize(matrix_imag(4)(l_col), bit_width_adder);
180             -- MultiPart
            temp_im(2) := resize(matrix_imag(1)(l_col), bit_width_adder)
- resize(matrix_real(1)(l_col), bit_width_adder);
182             temp_im(3) := resize(matrix_real(5)(l_col), bit_width_adder)
- resize(matrix_real(3)(l_col), bit_width_adder);

```

```

temp_im(4) := resize(matrix_real(7)(l_col), bit_width_adder)
- resize(matrix_imag(3)(l_col), bit_width_adder);
184      temp_im(5) := resize(matrix_imag(7)(l_col), bit_width_adder)
- resize(matrix_imag(5)(l_col), bit_width_adder);

186      -- 3. Zeile aus W
      -- 0: (1.00000 + 0.00000i), 1: (0.00000 + 1.00000i), 2: (-1.00000
+ 0.00000i), 3: (-0.00000 - 1.00000i),
188      -- 4: (1.00000 - 0.00000i), 5: (0.00000 + 1.00000i), 6: (-1.00000
+ 0.00000i), 7: (-0.00000 - 1.00000i)
      when 2 =>
190          -- Real
          temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
- resize(matrix_real(2)(l_col), bit_width_adder);
192          temp_re(1) := resize(matrix_imag(1)(l_col), bit_width_adder)
- resize(matrix_imag(3)(l_col), bit_width_adder);
          temp_re(2) := resize(matrix_real(4)(l_col), bit_width_adder)
- resize(matrix_real(6)(l_col), bit_width_adder);
194          temp_re(3) := resize(matrix_imag(5)(l_col), bit_width_adder)
- resize(matrix_imag(7)(l_col), bit_width_adder);
          -- Imag
196          temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
- resize(matrix_real(1)(l_col), bit_width_adder);
          temp_im(1) := resize(matrix_real(3)(l_col), bit_width_adder)
- resize(matrix_imag(2)(l_col), bit_width_adder);
198          temp_im(2) := resize(matrix_imag(4)(l_col), bit_width_adder)
- resize(matrix_real(5)(l_col), bit_width_adder);
          temp_im(3) := resize(matrix_real(7)(l_col), bit_width_adder)
- resize(matrix_imag(6)(l_col), bit_width_adder);

200      -- 4. Zeile aus W
      -- 0: ( 1.00000 + 0.00000i), 1: (-0.70711 + 0.70711i), 2:
(-0.00000 - 1.00000i), 3: ( 0.70711 + 0.70711i)
      -- 4: (-1.00000 + 0.00000i), 5: ( 0.70711 - 0.70711i), 6: (
0.00000 + 1.00000i), 7: (-0.70711 - 0.70711i)
204      when 3 =>
          -- Real
206          temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
- resize(matrix_imag(2)(l_col), bit_width_adder);
          temp_re(1) := resize(matrix_imag(6)(l_col), bit_width_adder)
- resize(matrix_real(4)(l_col), bit_width_adder);
208          -- MultPart
          temp_re(2) := resize(matrix_imag(1)(l_col), bit_width_adder)
- resize(matrix_real(1)(l_col), bit_width_adder);
210          temp_re(3) := resize(matrix_real(3)(l_col), bit_width_adder)
- resize(matrix_imag(5)(l_col), bit_width_adder);
          temp_re(4) := resize(matrix_imag(3)(l_col), bit_width_adder)
- resize(matrix_imag(7)(l_col), bit_width_adder);
212          temp_re(5) := resize(matrix_real(5)(l_col), bit_width_adder)
- resize(matrix_real(7)(l_col), bit_width_adder);

```

```

214      — Imag
      temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
— resize(matrix_imag(4)(l_col), bit_width_adder);
216      temp_im(1) := resize(matrix_real(2)(l_col), bit_width_adder)
— resize(matrix_real(6)(l_col), bit_width_adder);
      — MultPart
218      temp_im(2) := resize(matrix_imag(3)(l_col), bit_width_adder)
— resize(matrix_real(1)(l_col), bit_width_adder);
      temp_im(3) := resize(matrix_real(5)(l_col), bit_width_adder)
— resize(matrix_imag(1)(l_col), bit_width_adder);
220      temp_im(4) := resize(matrix_imag(5)(l_col), bit_width_adder)
— resize(matrix_real(3)(l_col), bit_width_adder);
      temp_im(5) := resize(matrix_real(7)(l_col), bit_width_adder)
— resize(matrix_imag(7)(l_col), bit_width_adder);
222
224      — 5. Zeile
      — 0: (1.00000 + 0.00000i), 1: (-1.00000 + 0.00000i), 2: (1.00000
— 0.00000i), 3: (-1.00000 + 0.00000i),
      — 4: (1.00000 - 0.00000i), 5: (-1.00000 + 0.00000i), 6: (1.00000
— 0.00000i), 7: (-1.00000 + 0.00000i)
226      when 4 =>
      — Real
228      temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
— resize(matrix_real(1)(l_col), bit_width_adder);
      temp_re(1) := resize(matrix_real(2)(l_col), bit_width_adder)
— resize(matrix_real(3)(l_col), bit_width_adder);
230      temp_re(2) := resize(matrix_real(4)(l_col), bit_width_adder)
— resize(matrix_real(5)(l_col), bit_width_adder);
      temp_re(3) := resize(matrix_real(6)(l_col), bit_width_adder)
— resize(matrix_real(7)(l_col), bit_width_adder);
232      — Imag
      temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
— resize(matrix_imag(1)(l_col), bit_width_adder);
234      temp_im(1) := resize(matrix_imag(2)(l_col), bit_width_adder)
— resize(matrix_imag(3)(l_col), bit_width_adder);
      temp_im(2) := resize(matrix_imag(4)(l_col), bit_width_adder)
— resize(matrix_imag(5)(l_col), bit_width_adder);
236      temp_im(3) := resize(matrix_imag(6)(l_col), bit_width_adder)
— resize(matrix_imag(7)(l_col), bit_width_adder);
238
240      — 6. Zeile
      — 0: ( 1.00000 + 0.00000i), 1: (-0.70711 - 0.70711i), 2: (
0.00000 + 1.00000i), 3: ( 0.70711 - 0.70711i),
      — 4: (-1.00000 + 0.00000i) 5: ( 0.70711 + 0.70711i), 6:
(-0.00000 - 1.00000i), 7: (-0.70711 + 0.70711i)
242      when 5 =>
      — Real
      temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
— resize(matrix_real(4)(l_col), bit_width_adder);
244      temp_re(1) := resize(matrix_imag(2)(l_col), bit_width_adder)
— resize(matrix_imag(6)(l_col), bit_width_adder);

```

```

246      --MultPart
      temp_re(2) := resize(matrix_real(3)(l_col), bit_width_adder)
-- resize(matrix_real(1)(l_col), bit_width_adder);
      temp_re(3) := resize(matrix_real(5)(l_col), bit_width_adder)
248 -- resize(matrix_imag(1)(l_col), bit_width_adder);
      temp_re(4) := resize(matrix_imag(5)(l_col), bit_width_adder)
-- resize(matrix_imag(3)(l_col), bit_width_adder);
      temp_re(5) := resize(matrix_imag(7)(l_col), bit_width_adder)
-- resize(matrix_real(7)(l_col), bit_width_adder);
250      -- Imag
      temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
-- resize(matrix_real(2)(l_col), bit_width_adder);
252      temp_im(1) := resize(matrix_real(6)(l_col), bit_width_adder)
-- resize(matrix_imag(4)(l_col), bit_width_adder);
      --MultPart
254      temp_im(2) := resize(matrix_real(1)(l_col), bit_width_adder)
-- resize(matrix_imag(1)(l_col), bit_width_adder);
      temp_im(3) := resize(matrix_real(3)(l_col), bit_width_adder)
-- resize(matrix_real(5)(l_col), bit_width_adder);
256      temp_im(4) := resize(matrix_imag(3)(l_col), bit_width_adder)
-- resize(matrix_real(7)(l_col), bit_width_adder);
      temp_im(5) := resize(matrix_imag(5)(l_col), bit_width_adder)
-- resize(matrix_imag(7)(l_col), bit_width_adder);
258
      -- 7. Zeile
260      -- 0: (1.00000 + 0.00000i), 1: (-0.00000 - 1.00000i), 2:
      (-1.00000 + 0.00000i), 3: ( 0.00000 + 1.00000i),
      -- 4: (1.00000 - 0.00000i), 5: (-0.00000 - 1.00000i), 6:
      (-1.00000 + 0.00000i), 7: (-0.00000 + 1.00000i)
262      when 6 =>
      -- Real
264      temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
-- resize(matrix_imag(1)(l_col), bit_width_adder);
      temp_re(1) := resize(matrix_imag(3)(l_col), bit_width_adder)
-- resize(matrix_real(2)(l_col), bit_width_adder);
266      temp_re(2) := resize(matrix_real(4)(l_col), bit_width_adder)
-- resize(matrix_imag(5)(l_col), bit_width_adder);
      temp_re(3) := resize(matrix_imag(7)(l_col), bit_width_adder)
-- resize(matrix_real(6)(l_col), bit_width_adder);
268      -- Imag
      temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
-- resize(matrix_imag(2)(l_col), bit_width_adder);
270      temp_im(1) := resize(matrix_real(1)(l_col), bit_width_adder)
-- resize(matrix_real(3)(l_col), bit_width_adder);
      temp_im(2) := resize(matrix_imag(4)(l_col), bit_width_adder)
-- resize(matrix_imag(6)(l_col), bit_width_adder);
272      temp_im(3) := resize(matrix_real(5)(l_col), bit_width_adder)
-- resize(matrix_real(7)(l_col), bit_width_adder);
274      -- 8. Zeile

```

```

-- 0: ( 1.00000 + 0.00000i), 1: ( 0.70711 - 0.70711i), 2:
276 (-0.00000 - 1.00000i), 3: (-0.70711 - 0.70711i),
-- 4: (-1.00000 + 0.00000i), 5: (-0.70711 + 0.70711i), 6:
278 (-0.00000 + 1.00000i), 7: ( 0.70711 + 0.70711i)
when 7 =>
-- Real
temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder)
- resize(matrix_imag(2)(l_col), bit_width_adder);
280 temp_re(1) := resize(matrix_imag(6)(l_col), bit_width_adder)
- resize(matrix_real(4)(l_col), bit_width_adder);
--MultPart
282 temp_re(2) := resize(matrix_real(1)(l_col), bit_width_adder)
- resize(matrix_imag(1)(l_col), bit_width_adder);
temp_re(3) := resize(matrix_imag(5)(l_col), bit_width_adder)
- resize(matrix_real(3)(l_col), bit_width_adder);
284 temp_re(4) := resize(matrix_real(7)(l_col), bit_width_adder)
- resize(matrix_imag(3)(l_col), bit_width_adder);
temp_re(5) := resize(matrix_imag(7)(l_col), bit_width_adder)
- resize(matrix_real(5)(l_col), bit_width_adder);
286 -- Imag
temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder)
- resize(matrix_imag(4)(l_col), bit_width_adder);
288 temp_im(1) := resize(matrix_real(2)(l_col), bit_width_adder)
- resize(matrix_real(6)(l_col), bit_width_adder);
--MultPart
290 temp_im(2) := resize(matrix_real(1)(l_col), bit_width_adder)
- resize(matrix_imag(3)(l_col), bit_width_adder);
temp_im(3) := resize(matrix_imag(1)(l_col), bit_width_adder)
- resize(matrix_real(5)(l_col), bit_width_adder);
292 temp_im(4) := resize(matrix_real(3)(l_col), bit_width_adder)
- resize(matrix_imag(5)(l_col), bit_width_adder);
temp_im(5) := resize(matrix_imag(7)(l_col), bit_width_adder)
- resize(matrix_real(7)(l_col), bit_width_adder);
294
when others => element := element; -- "dummy arbeit", es sind
bereits alle Faelle abgedeckt!
296 end case;

next_dft_state <= additions_stage1;

298
300 when additions_stage1 => -- dft_state_out = 2
302
-- Es wird vor jeder Addition ein Bitshift auf die Summanden
angewandt, um den Wertebereich der Speichervariable beim
zurueckschreiben nicht zu ueberschreiten (1. Mal)
304
-- Zeilen 1, 3, 5, 7 (ungerade) aufsummieren (bzw. 0(000XXX), 2(010
XXX), 4(100XXX), 6(110XXX) beginnend bei 0)
306 if element(3) = '0' then

```

```

308
310      -- Real
      temp_re(0) := resize(temp_re(0)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_re(1)(bit_width_adder-1 downto 1),
bit_width_adder);
312      temp_re(1) := resize(temp_re(2)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_re(3)(bit_width_adder-1 downto 1),
bit_width_adder);
      -- Imag
314      temp_im(0) := resize(temp_im(0)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(1)(bit_width_adder-1 downto 1),
bit_width_adder);
      temp_im(1) := resize(temp_im(2)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(3)(bit_width_adder-1 downto 1),
bit_width_adder);
316    else
      -- gerade Zeilen aus W
318      -- Real
      --ConstPart
320      temp_re(0) := resize(temp_re(0)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_re(1)(bit_width_adder-1 downto 1),
bit_width_adder);
      --MultPart
322      temp_re(2) := resize(temp_re(2)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_re(3)(bit_width_adder-1 downto 1),
bit_width_adder);
      temp_re(4) := resize(temp_re(4)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_re(5)(bit_width_adder-1 downto 1),
bit_width_adder);
324      -- Imag
      --ConstPart
326      temp_im(0) := resize(temp_im(0)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(1)(bit_width_adder-1 downto 1),
bit_width_adder);
      --MultPart
328      temp_im(2) := resize(temp_im(2)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(3)(bit_width_adder-1 downto 1),
bit_width_adder);
      temp_im(4) := resize(temp_im(4)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(5)(bit_width_adder-1 downto 1),
bit_width_adder);
330    end if;

332    next_dft_state <= additions_stage2;

334
336    when additions_stage2 => -- dft_state_out = 3
      -- Es wird vor jeder Addition ein Bitshift auf die Summanden
angewandt, um den Wertebereich der Speichervariable nicht zu
ueberschreiten (2. Mal)

```

— Zusätzlich wird beim Zuweisen der ungeraden Zeilen an die 1D-DFT-Matrix zwei weitere Male geshiftet.

— 1 Mal, um den Wertebereich der 1D- bzw. 2D-DFT-Matrix klein genug zu halten, ein weiteres Mal, um gleich oft wie bei den geraden Zeilen zu shiften

— Zeilen 1, 3, 5, 7 (wie oben)

if element(3) = '0' then

— Real

temp\_re(0) := resize(temp\_re(0)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder) + resize(temp\_re(1)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder);

— Imag

temp\_im(0) := resize(temp\_im(0)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder) + resize(temp\_im(1)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder);

— Hier werden die Bits um 2 Stellen nach rechts geschoben,  
damit die Werte mit den Zeilen 2, 4, 6, 8 vergleichbar sind. Dort wird  
insgesamt gleich

— oft geshiftet, aber auch 1x mehr aufaddiert.

— Indizes vertauschen -> Transponiert abspeichern

if dft\_1d\_2d = '0' then

dft\_1d\_real(l\_col)(row\_col\_idx) := resize(temp\_re(0)(  
bit\_width\_adder-1 downto 2), bit\_width\_extern);

dft\_1d\_imag(l\_col)(row\_col\_idx) := resize(temp\_im(0)(  
bit\_width\_adder-1 downto 2), bit\_width\_extern);

else

result\_real(l\_col)(row\_col\_idx) <= resize(temp\_re(0)(  
bit\_width\_adder-1 downto 2), bit\_width\_extern);

result\_imag(l\_col)(row\_col\_idx) <= resize(temp\_im(0)(  
bit\_width\_adder-1 downto 2), bit\_width\_extern);

end if;

element := element+1;

element\_out <= element;

— naechster Zustand

next\_dft\_state <= twiddle\_calc;

else

— Real

temp\_re(2) := resize(temp\_re(2)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder) + resize(temp\_re(4)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder);

— Imag

temp\_im(2) := resize(temp\_im(2)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder) + resize(temp\_im(4)(bit\_width\_adder-1 downto 1),  
bit\_width\_adder);



```

372      -- naechster Zustand
373      next_dft_state <= const_mult;
374  end if;

376
377  when const_mult => -- dft_state_out = 4
378
379      -- Der Zielvektor der Multiplikation ist 26 Bit breit, die beiden
380      Multiplikanten sind mit je 13 Bit wie gefordert halb so breit.
381
382      -- Zeilen 2, 4, 6, 8 (vergleichbar mit oben)
383      mult_re := temp_re(2) * twiddle_coeff; --(16 downto 16-(
384      bit_width_adder-1));
385      mult_im := temp_im(2) * twiddle_coeff; --(16 downto 16-(
386      bit_width_adder-1));
387
388      next_dft_state <= additions_stage3;
389
390  when additions_stage3 => -- dft_state_out = 5
391
392      -- Die vordersten 12 Bit des Multiplikationsergebnisses werden
393      verwendet und um 1 Bit nach rechts geschiftet, damit der Wert halbiert
394      wird und der Zielvektor spaeter keinen Ueberlauf hat.
395      -- Um wieder die vollen 13 Bit zu erhalten, wird die resize-
396      Funktion verwendet.
397      -- Real
398
399      temp14bit_re := resize(mult_re(bit_width_multiplier-4 downto
400      bit_width_multiplier-4-bit_width_extern), bit_width_adder+1) + resize(
401      temp_re(0)(bit_width_adder-1 downto 1), bit_width_adder+1);
402      temp_re(0) := temp14bit_re(bit_width_adder downto 1);
403
404      -- Imag
405
406      temp14bit_im := resize(mult_im(bit_width_multiplier-4 downto
407      bit_width_multiplier-4-bit_width_extern), bit_width_adder+1) + resize(
408      temp_im(0)(bit_width_adder-1 downto 1), bit_width_adder+1);
409      temp_im(0) := temp14bit_im(bit_width_adder downto 1);
410
411      -- Indizes vertauschen -> Transponiert abspeichern
412      if dft_1d_2d = '0' then
413          dft_1d_real(l_col)(row_col_idx) := temp_re(0)(bit_width_adder-1
414      downto 1);
415          dft_1d_imag(l_col)(row_col_idx) := temp_im(0)(bit_width_adder-1
416      downto 1);
417      else
418          result_real(l_col)(row_col_idx) <= temp_re(0)(bit_width_adder-1
419      downto 1);
420          result_imag(l_col)(row_col_idx) <= temp_im(0)(bit_width_adder-1
421      downto 1);

```

```

408     end if;

410     next_dft_state <= twiddle_calc;
411     if element = 63 then
412         if dft_1d_2d = '1' then
413             next_dft_state <= set_ready_bit;
414         end if;
415         dft_1d_2d := not dft_1d_2d;
416         dft_1d_2d_out <= dft_1d_2d;
417     end if;

418

420     element := element+1;
421     element_out <= element;

422

424     when set_ready_bit =>
425         result_ready <= '1';
426         next_dft_state <= twiddle_calc;

428

429     when others => next_dft_state <= twiddle_calc;
430 end case;

432 end process;
end arch;

```

Listing 8.16: Berechnung der 2D-DFT

```

library ieee;
2 use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
4 library work;
  use work.all;
6 use constants.all;
  use datatypes.all;
8

entity dft8optimiert_top is
10 --     port(
11 --         result_real : out t_2d_array;
12 --         result_imag : out t_2d_array
13 --     );
14 end entity dft8optimiert_top;

16 architecture arch of dft8optimiert_top is

18     signal nReset      : bit;
19     signal clk          : bit;
20     signal input_real   : t_2d_array;
21     signal input_imag   : t_2d_array;
22     signal result_real  : t_2d_array;
23     signal result_imag  : t_2d_array;

```

```

24  signal loaded          : bit;
25  signal result_ready    : bit;
26  signal write_done      : bit;
27  signal idft            : bit := '0';
28
29  signal state_out       : t_dft8_states;
30  signal element_out     : unsigned(5 downto 0);
31  signal dft_1d_2d_out  : bit;
32
33  component dft8optimiert
34  port(
35      clk          : in  bit;
36      nReset       : in  bit;
37      loaded       : in  bit;
38      input_real   : in  t_2d_array;
39      input_imag   : in  t_2d_array;
40      result_real  : out t_2d_array;
41      result_imag  : out t_2d_array;
42      result_ready : out bit;
43      idft         : in  bit;
44      state_out    : out t_dft8_states;
45      element_out  : out unsigned(5 downto 0);
46      dft_1d_2d_out : out bit
47  );
48  end component;
49
50
51  component read_input_matrix
52  port(
53      clk          : in  bit;
54      loaded       : out bit;
55      input_real   : out t_2d_array;
56      input_imag   : out t_2d_array
57  );
58  end component;
59
60
61  component write_results
62  port(
63      result_ready : in  bit;
64      result_real  : in  t_2d_array;
65      result_imag  : in  t_2d_array;
66      write_done   : out bit
67  );
68  end component;
69
70
71  begin
72      dft : dft8optimiert
73      port map(

```

```

76         nReset      => nReset ,
77         clk         => clk ,
78         loaded      => loaded ,
79         input_real  => input_real ,
80         input_imag  => input_imag ,
81         result_real => result_real ,
82         result_imag => result_imag ,
83         result_ready => result_ready ,
84         idft        => idft ,
85         state_out   => state_out ,
86         element_out => element_out ,
87         dft_1d_2d_out => dft_1d_2d_out
88     );
89
90     mat : read_input_matrix
91     port map(
92         clk         => clk ,
93         loaded      => loaded ,
94         input_real  => input_real ,
95         input_imag  => input_imag
96     );
97
98     write : write_results
99     port map(
100         result_ready => result_ready ,
101         result_real  => result_real ,
102         result_imag  => result_imag ,
103         write_done   => write_done
104     );
105
106     clk    <= not clk after 20 ns;
107     nReset <= '1' after 40 ns;
108 end arch;

```

Listing 8.17: Top-Level-Entität der 2D-DFT

## 8.5 Testumgebung

```

#!/bin/bash
2
matlab_script="binMat2decMat.m"
4
./simulate.sh && matlab -nojvm -nodisplay -nosplash -r $matlab_script
6
stty echo

```

Listing 8.18: Aufruf der Testumgebung, Vergleich von VHDL- und Matlab-Ergebnissen

tlab

```
1 #!/bin/bash
3 # global settings
5 errormax=15
6 worklib=worklib
7 #testbench=top_level_tb
8 testbench=dft8optimiert_top
9 architecture=arch
10 simulation_time="1500ns"
11
13 # VHDL-files
15 constant_declarations="constants.vhdl"
16 datatype_declarations="datatypes.vhdl"
17
18 main_entity="dft8optimiert.vhdl"
19 top_level_entity="dft8_optimiert_top.vhdl"
20 #top_level_testbench=
21
22 embedded_entity_1="read_input_matrix.vhdl"
23 embedded_entity_2="write_results.vhdl"
24
25 constant_declarations=$directory$constant_declarations
26 datatype_declarations=$directory$datatype_declarations
27 function_declarations=$directory$function_declarations
28 main_entity=$directory$main_entity
29 top_level_entity=$directory$top_level_entity
30 #top_level_testbench=$directory$top_level_testbench
31
32 embedded_entity_1=$directory$embedded_entity_1
33 embedded_entity_2=$directory$embedded_entity_2
34
35
37 # libs und logfiles
39 cdslib="cds.lib"
40 elab_logfile="ncelab.log"
41 ncvhdl_logfile="nchvdl.log"
42 ncsim_logfile="ncsim.log"
43
44 cdslib=${base_dir}${work_dir}${cdslib}
45 elab_logfile=${directory}${elab_logfile}
46 ncvhdl_logfile=${directory}${ncvhdl_logfile}
47 ncsim_logfile=${directory}${ncsim_logfile}
48
49 ###
51
```

```

ncvhdl \
53 -work $worklib \
   -cdslib $cdslib \
55 -logfile $ncvhdl_logfile \
   -errormax $errormax \
57 -update \
   -v93 \
59 -linedebug \
   $constant_declarations \
61 $datatype_declarations \
   $embedded_entity_1 \
63 $embedded_entity_2 \
   $main_entity \
65 $top_level_entity \
   # $top_level_testbench
67 #-status \

69 ncelab \
   -work $worklib \
71 -cdslib $cdslib \
   -logfile $elab_logfile \
73 -errormax $errormax \
   -access +wc \
75 ${worklib}.${testbench}
   #-status \

77 ncsim \
79 -cdslib $cdslib \
   -logfile $ncsim_logfile \
81 -errormax $errormax \
   -exit \
83 ${worklib}.${testbench}:${architecture} \
   -input testRUN.tcl
85 #-status \

87

89 #ncvhdl -work worklib -cdslib /home/tlattmann/cadence/mat_mult/cds.lib -
   logfile /home/tlattmann/cadence/mat_mult/nchvdl.log -errormax 15 -update
   -v93 -linedebug /home/tlattmann/cadence/mat_mult/HDL/constants.vhdl /
   home/tlattmann/cadence/mat_mult/HDL/datatypes.vhdl /home/tlattmann/
   cadence/mat_mult/HDL/functions.vhdl /home/tlattmann/cadence/mat_mult/HDL
   /read_input_matrix.vhdl /home/tlattmann/cadence/mat_mult/HDL/
   write_results.vhdl /home/tlattmann/cadence/mat_mult/HDL/dft8optimiert.
   vhdl /home/tlattmann/cadence/mat_mult/HDL/dft8_optimiert_top.vhdl -
   status

91 #ncelab -work worklib -cdslib /home/tlattmann/cadence/mat_mult/cds.lib -
   logfile /home/tlattmann/cadence/mat_mult/ncelab.log -errormax 15 -access
   +wc worklib.dft8optimiert_top -status

```

```

93 #ncsim -cdslib /home/tlattmann/cadence/mat_mult/cds.lib -logfile /home/
    tlattmann/cadence/mat_mult/ncsim.log -errormax 15 worklib.
    dft8_optimiert_top:arch -input testRUN.tcl -status

95 #database -open waves -into waves.shm -default
#probe -create -shm :clk :input_imag :input_real :loaded :mult_im_out :
    mult_re_out :multState_out :nReset :result_imag :result_ready :
    result_real :sum1_stage1_3v6_re_out :sum1_stage2_2v3_re_out :
    sum1_stage2_3v3_re_out :sum1_stage3_1v1_re_out :sum3_stage1_im_out :
    sum3_stage1_re_out :sum3_stage2_im_out :sum3_stage2_re_out :
    sum3_stage3_im_out :sum3_stage3_re_out :sum3_stage4_im_out :
    sum3_stage4_re_out :write_done

```

Listing 8.19: Simulations des VHDL-Quelltextes

```
run 32us
```

Listing 8.20: Dauer der Simulation

```

1 filename_2 = 'InputMatrix_komplex.txt';
  filename_1 = 'Results.txt';

3

5 delimiterIn = ' ';

7 bit_width_extern = 13

9 Input_bin = importdata(filename_2, delimiterIn);
  Input_bin_real = Input_bin(:,1:2:end);
  Input_bin_imag = Input_bin(:,2:2:end);

11

13 Results_vhdl_bin = importdata(filename_1, delimiterIn);
  Results_vhdl_bin_real = Results_vhdl_bin(:,1:2:end);
  Results_vhdl_bin_imag = Results_vhdl_bin(:,2:2:end);

15

17 Input_dec_imag = nan(8);
  Results_vhdl_dec_real = nan(8);
19 Results_vhdl_dec_imag = nan(8);
  Result_octave_real_1d = nan(8);
21 Result_octave_imag_1d = nan(8);

23

25 a=fi(0,1,bit_width_extern,bit_width_extern-2);

N = 8;
27 for m = 1:N
    for n = 1:N
29         a_bin=mat2str(Results_vhdl_bin_real(m,n),bit_width_extern);
          Results_vhdl_dec_real(m,n) = a.double;
31         a_bin=mat2str(Results_vhdl_bin_imag(m,n),bit_width_extern);
          Results_vhdl_dec_imag(m,n) = a.double;
33

```

```

    a.bin=mat2str(Input_bin_real(m,n),bit_width_extern);
35    Input_dec_real(m,n) = a.double;
    a.bin=mat2str(Input_bin_imag(m,n),bit_width_extern);
37    Input_dec_imag(m,n) = a.double;
    end
39 end

41
Input_dec=Input_dec_real+1i*Input_dec_imag;
43

45 TW=exp(-i*2*pi*[0:7]'/8);
47
49
51 %Result_octave_1d=TW*Input_dec;
51 %Result_octave_real_1d=real(Result_octave_1d)/16
51 %Result_octave_imag_1d=imag(Result_octave_1d)
53
55 Result_octave=TW*Input_dec*TW.';
55 Result_octave=Result_octave./256;

57 Results_vhdl_dec_real
Result_octave_real=real(Result_octave)
59
Result_octave_imag=imag(Result_octave);
61 Results_vhdl_dec_imag;

63 diff_real=Result_octave_real-Results_vhdl_dec_real
diff_imag=Result_octave_imag-Results_vhdl_dec_imag;
65
quit

```

Listing 8.21: Berechnung der Differenzen der DFT in Matlab und VHDL