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

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

Abgegeben am 20.04.2018

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Stand der Technik 1
1.3 Ziel dieser Arbeit 1

2 Grundlagen **2**
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 Zahler im SQ-Format 3
2.1.4 Numerisch bedingte Ungenauigkeiten 3
2.2 Komplexe Multiplikation 4
2.3 Matrixmultiplikation 4
2.4 Fourierreihenentwicklung 4
2.5 Fouriertransformation 6
2.6 Diskrete Fouriertransformation (DFT) 7
2.6.1 Summen- und Matrizenschreibweise der DFT 7
2.6.2 2D-DFT mit rellen Eingangswerten 9
2.6.3 Berechnung der Diskreten Fouriertransformation mittels FFT 11
2.6.4 Inverse DFT 12
2.7 Diskrete Kosinus Transformation (DCT) 13
2.7.1 Verwendung der DCT 13
2.7.2 Berechnung der DCT 13

3 Analyse **14**
3.1 Bewertung verschiedener DCT-Größen 14
3.2 Bewertung verschiedener DFT-Größen 15
3.3 Entscheidung DCT vs. DFT 18
3.4 Abschätzung des Rechenaufwands 18
3.4.1 Gegenüberstellung von reellen und komplexen Eingangswerten 18
3.4.2 Direkte Multiplikation zweier 8x8 Matrizen 20
3.4.3 Optimierte 8x8 DFT als Matrixmultiplikation 21
3.4.4 Gegenüberstellung von Butterfly-Algorithmus und optimierter Matrixmultiplikation 21
3.5 Kompromiss aus benötigter Chipfläche und Genauigkeit des Ergebnisses 21

4 Entwurf **23**
4.1 Interpretation binärer Zahlen . . . 23
4.2 Entwicklungsstufen . . . 23
4.2.1 Multiplikation . . . 23
4.2.2 Addierer . . . 23
4.2.3 Konstantenmultiplikation . . . 23
4.2.4 1D-DFT mit Integer-Werten . . . 24
4.2.5 2D-DFT mit Integer-Werten . . . 24
4.2.6 2D-DFT mit Werten SQ-Format . . . 24
4.2.7 Zusammenhang von DFT und IDFT bei der Matrixmultiplikation . . . 24
4.3 Test der Matrixmultiplikation . . . 24
4.4 Implementierung des Konstantenmultiplizieres . . . 24
4.4.1 Syntheseergebnis eines 13 Bit Konstantenmultiplizierers . . . 24
4.4.2 Syntheseergebnis für die Bildung des Zweierkomplements eines 13 Bit Vektors . . . 25
4.5 Entwickeln der 2D-DFT in VHDL . . . 25
4.6 Direkte Weiterverarbeitung der Zwischenergebnisse . . . 27
4.7 Berechnungsschema der geraden und ungeraden Zeilen . . . 27
4.7.1 Erwartete Anzahl benötigter Takte . . . 30
4.8 Schema der Zustandsfolge . . . 30
4.9 UML-Diagramm . . . 32
4.10 Projekt- und Programmstruktur . . . 35
4.11 Bibliotheken und Hardwarebeschreibungssprache . . . 35

5 Evaluation **36**
5.1 Simulation . . . 36
5.1.1 NC Sim - positive Zahlendarstellung . . . 36
5.2 Anzahl benötigter Takte . . . 36
5.3 Zeitabschätzung im Einsatz als ABS-Sensor . . . 36
5.4 Testumgebung . . . 39
5.4.1 Struktogramm des Testablaufs . . . 39
5.4.2 Reale Eingangswerte . . . 39
5.5 Chipdesign . . . 39
5.5.1 Anzahl Standardzellen . . . 39
5.5.2 Visualisierung der Netzliste . . . 39
5.5.3 Floorplan, Padring . . . 39

6 Schlussfolgerungen **40**
6.1 Zusammenfassung . . . 40
6.2 Bewertung und Fazit . . . 40
6.3 Ausblick . . . 40

7 Abkürzungsverzeichnis **41**

Abbildungsverzeichnis **42**

Tabellenverzeichnis **43**

Literatur **44**

8 Anhang **45**

8.1 Skript zur Bewertung von Twiddlefaktormatrizen 45
8.2 Gate-Report des 12 Bit Konstatenmultiplizierers 49
8.3 Twiddlefaktormatrix im S1Q10-Format 49
8.4 Programmcode 54
8.5 Testumgebung 77

1 Einleitung

1.1 Motivation

Sensorarray beschreiben

1.2 Stand der Technik

Der verwendete Prozess ist mit 350 um im Vergleich zu modernen Prozessen mit beispielsweise 20 nm Strukturbreite um die Grösenordnung 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 tunnelmagnetoresistiven Effekt (TMR).

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

2 Grundlagen

2.1 Binäre Zahlendarstellung von Festkommazahlen

Im Rahmen dieses Projekts wird von Ein- sowie Ausgangswerten mit einer Genauigkeit von 12 Bit ausgegangen, es sollen Werte von $-2 < z < 2$ dargestellt werden können.

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 as 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. Dies hat zur Folge, dass es eine positive und eine negative Null und somit eine Doppeldeutigkeit gibt. Desweiteren wird ein LSB an Auflösung verschenkt. Der Wertebereich erstreckt sich von $-2^{MSB-1}+1\,LSB$ bis $2^{MSB-1} - 1\,LSB$

Diese Darstellung hat den Vorteil, dass sich das Ergebnis einer Multiplikation der Zahlen $a \cdot b$ und $-a \cdot b$ nur im vorderste 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. Da als einziger konstanter Multiplikand in der 8x8-DFT-Matrix der Faktor $\pm\frac{\sqrt{2}}{2}$ auftaucht, also das oben angeführte Beispiel zutrifft, erschien diese Darstellungsform zwischenzeitlich interessant.

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 es MSB ebenfalls erkannt werden, ob es sich um eine positive oder negative Zahl handelt. Dennoch wird es nicht als Vorzeichenbit gewertet. Viel mehr bedeutet ein gesetztes MSB -2^{-1}, welches 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 den neuen Wert muss abschließend 1 LSB addiert werden.

Vorteile bei dieser Darstellung ist, dass die mathematischen Operationen Addition, Subtraktion und Multiplikation direkt angewand 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 Zahler 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. Der darstellbare Zahlenbereich liegt hier bei $-1 \leq z < 1$. Um den geforderten Bereich von etwa ± 2 abbilden zu können wird deshalb ein Vorkommabit benötigt.

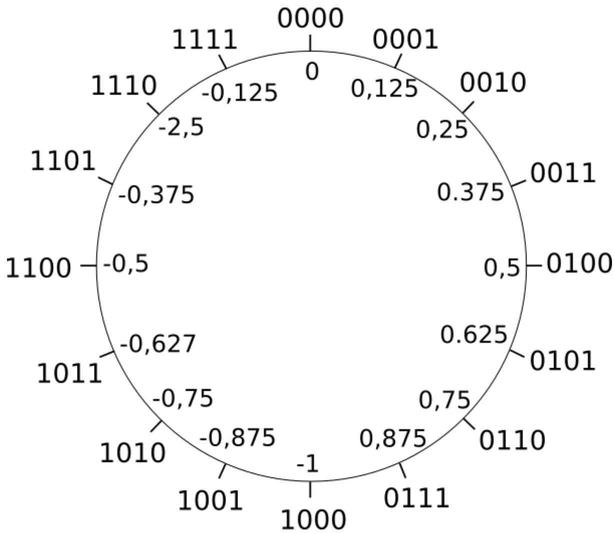


Abbildung 2.1: Interpretation von Dualzahlen im SQ3-Format

Da 12 Bit zur Verfügung stehen, von denen eins für das Vorzeichen und ein weiters 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 $1\,LSB = 2^{-10} = \frac{1}{1024} = 9,765625 \cdot 10^{-4}$. Der Wertebereich liegt in diesem Fall liegt bei -2 bis $1,999023438$.

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

2.1.4 Numerisch bedingte Ungenauigkeiten

Bei einem Bitshift 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. Da dieses Problem bei jedem Bitshift auftritt und die Wahrscheinlichkeit für eine 1 bei 50% liegt, muss davon ausgegangen werden, dass ein positives Ergebnis etwas kleiner und ein negatives vom Betrag her etwas größer ist, als bei verlustfreier Berechnung.

Von Prof. Vollmer:

$$S_N = \frac{P_Q}{(2^0)^2} + \frac{P_Q}{(2^1)^2} + \frac{P_Q}{(2^2)^2} + \cdots + \frac{P_Q}{(2^{L-1})^2} \tag{2.1}$$

L : Stufe (Additionstakte?)

Da diese Arbeit den Schwerpunkt in der Aufwandsabschätzung einer Chipimplementierung 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. Für eine

Vielleicht lass ich das auch weg!

2.2 Komplexe Multiplikation

Im allgemeinen Fall müssen gemäß Gl. (2.2) bei der komplexen Multiplikation vier einfache Multiplikation sowie zwei Additionen durchgeführt werden.

$$\begin{aligned} e + jf &= (a + jb) \cdot (c + jd) \\ &= a \cdot c + j(a \cdot d) + j(b \cdot c) + j^2(b \cdot d) \\ &= a \cdot c + b \cdot d + j(a \cdot d + b \cdot c) \end{aligned} \tag{2.2}$$

2.3 Matrixmultiplikation

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

2.4 Fourierreihenentwicklung

Mit einer Fourierreihe kann ein periodisches Signal aus einer Summe von Sinus- und Konsinusfunktionen zusammengesetzt werden. Die Schreibweise als Summe von Sinus- und Kosinusfunktionen (Gl. 2.3) ist eine der häufigsten Darstellungsformen.

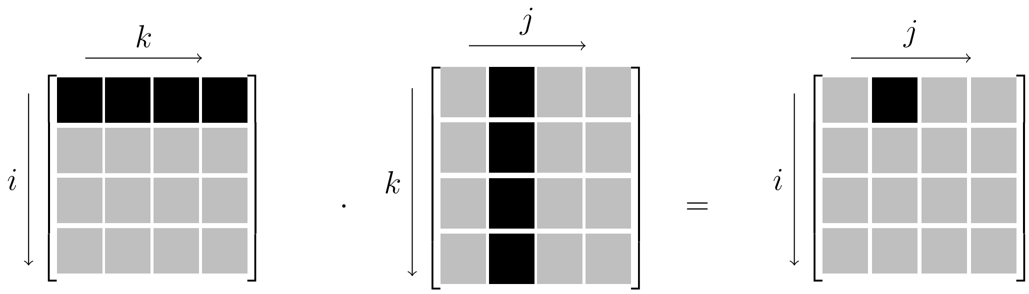


Abbildung 2.2: Veranschaulichung der Matrixmultiplikation

$$x(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} (a_k cos(kt) + b_k sin(kt)) \tag{2.3}$$

Die Fourierkoeffizienten lassen sich über die Gleichungen (2.4) und (2.5) berechnen:

$$a_k = \frac{1}{\pi} \int_{-\pi}^{\pi} x(t) \cdot cos(kt) dt \quad \text{für} \quad k \geq 0 \tag{2.4}$$

$$b_k = \frac{1}{\pi} \int_{-\pi}^{\pi} x(t) \cdot sin(kt) dt \quad \text{für} \quad k \geq 1 \tag{2.5}$$

Mit der Exponentialschreibweise lassen sich Sinus und Kosinus auch wie in (2.6) und (2.7) ausdrücken:

$$cos(kt) = \frac{1}{2} \left(e^{jkt} + e^{-jkt} \right) \tag{2.6}$$

$$sin(kt) = \frac{1}{2j} \left(e^{jkt} - e^{-jkt} \right) \tag{2.7}$$

und zusammengefasst ergibt sich in (Gl. 2.8) der komplexe Zeiger, der eine Rotation im Gegenuhrzeigersinn auf dem Einheitskreis beschreibt. In Abbildung 2.3 dies zusätzlich noch grafisch dargestellt.

$$\begin{aligned} cos(kt) + j \cdot sin(kt) &= \frac{1}{2} \left(e^{jkt} + e^{-jkt} \right) + j \cdot \frac{1}{2j} \left(e^{jkt} - e^{-jkt} \right) \\ &= \frac{1}{2} \left(e^{jkt} + e^{jkt} \right) \\ &= e^{jkt} \end{aligned} \tag{2.8}$$

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

$$c_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} x(t) e^{-j2\pi kt} dt \quad \forall k \in \mathbb{Z} \tag{2.9}$$

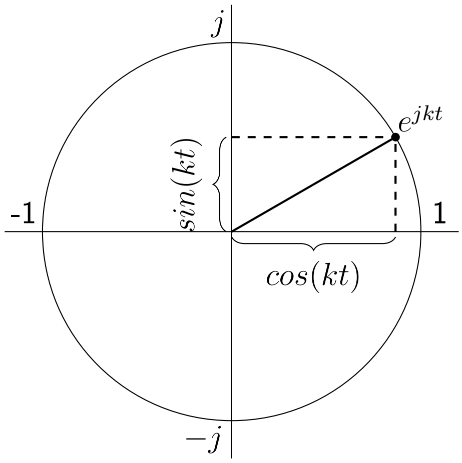


Abbildung 2.3: Einheitskreis, Zusammensetzung des komplexen Zeigers aus Sinus und Kosinus

$$x(t) = \sum_{-\infty}^{\infty} c_k e^{jkt} \tag{2.10}$$

2.5 Fouriertransformation

Mit der Fouriertransformation kann umgekert 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. 2.11)

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-j2\pi ft} \tag{2.11}$$

Wenn Sinus- und Kosinusfunktionen wie in Gl. (2.6) und (2.7) als Exponentialfunktion geschrieben werden, können sie zu einer komplexen Exponentialfunktion zusammengefasst werden. Hieraus lääst sich ableiten, dass das Spektrum, also $X(f)$ komplexwertig sein muss.

Für 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.

2.6 Diskrete Fouriertransformation (DFT)

Die Diskrete Fouriertransformation (DFT) ist die zeit- und wertdiskrete Variante der Fouriertransformation, die statt von $-\infty$ bis ∞ 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 (2.11) in die Summe aus Gleichung (2.12) über.

Üblicher Weise wird die (diskrete) Fouriertransformation genutzt, um vom Zeitbereich in den Frequenzbereich zu gelangen. In diesem Fall enthielte der Eingangsvektor Werten im Zeitbereich, der Ausgangsvektor Werten im Frequenzbereich. Um von Daten im Zeitbereich sprechen zu können, müssen diese zeitliche 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 so 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 Bilbereich.

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 NxN auch NxN-DFT genannt.

2.6.1 Summen- und Matrizenschreibweise der DFT

1D-DFT

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

$$X^*[m] = \frac{1}{N} \cdot \sum_{n=0}^{N-1} x[n] \cdot e^{-\frac{j2\pi mn}{N}} \tag{2.12}$$

In Gleichung (2.12) 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 (2.14) zeigt die Summenformel aus (2.12) umgeschrieben zu einer Matrixmultiplikation.

Mit Gleichung (2.13) 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.

$$\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} e^{-\frac{j2\pi mn}{N}} = W \tag{2.13}$$

Somit gilt:

$$X^* = W \cdot x \tag{2.14}$$

In Matlab kann die Twiddlefaktormatrix mit

$$W = e^{-\frac{i2\pi}{N}\cdot[0:N-1]'\cdot[0:N-1]} \tag{2.15}$$

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

2D-DFT

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

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

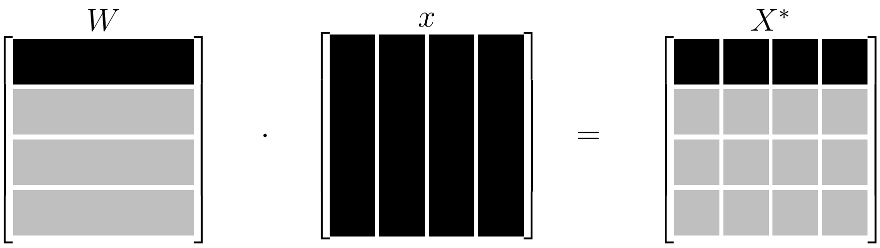
Auch hier lässt sich die Berechnung in Matrizenschreibweise darstellen:

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

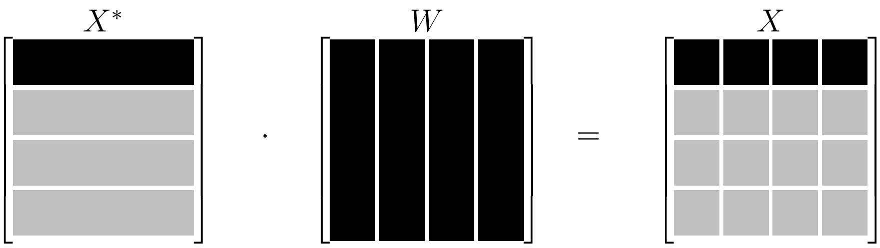
Die Gleichungen (2.14) und (2.17) werden wesentlicher Bestandteil der Umsetzung der 2D-DFT sein.

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.



(2.18)



(2.19)

2.6.2 2D-DFT mit rellen 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 relle Eingangssignale der 2D-DFTs zu betrachten und später wieder zusammen zu setzen. Gleiches gilt dann natürlich auch für ein komplexes Eingangssignal, welches ebenfalls in zwei von einander unabhängigen DFTs transformiert werden. 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. Die grau hinterlegten Felder sind die Multiplikationen der Twiddlefaktormatrix. Es müssen bei der 8x8-DFT also statt 16 nur 8 Multiplikationen mit rellem Multiplikand und komplexen Multiplkator erfolgen.

Wie bereits beschrieben lässt sich dieses Verfahren auch für komplexe Eingangssignale, deren Real- und Imaginärteil separat von einander mit der DFT transformiert werden, anwenden. Anschließend müssen die Ergebnisse zusammen gesetzt werden. Wie dies geschieht ist der Abbildung 2.4 zu entnehmen.

In Abbildung 2.4 ist die schematische Berechnung der 2D-DFT eines reellen Eingangssignals zu sehen. Um die 2D-DFT eines komplexen Eingangssignals zu berechnen, muss entweder eine identische Einheit für den Imaginärteil vorhanden sein oder noch mehr zeitlich versetzt berechnet werden. Die Ergebnisse beider 2D-DFTs müssen identisch zusammengefasst werden, wie es zum Abschluss der einzelnen 2D-DFTs geschehen muss.

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 Berechnung 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

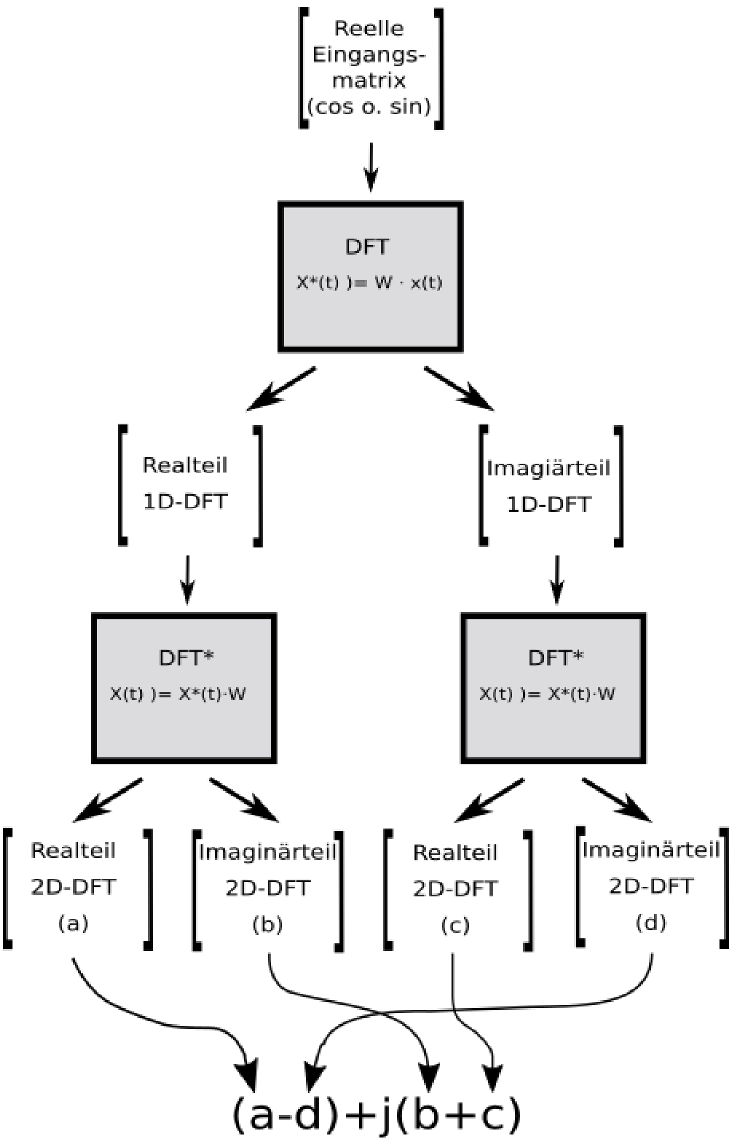
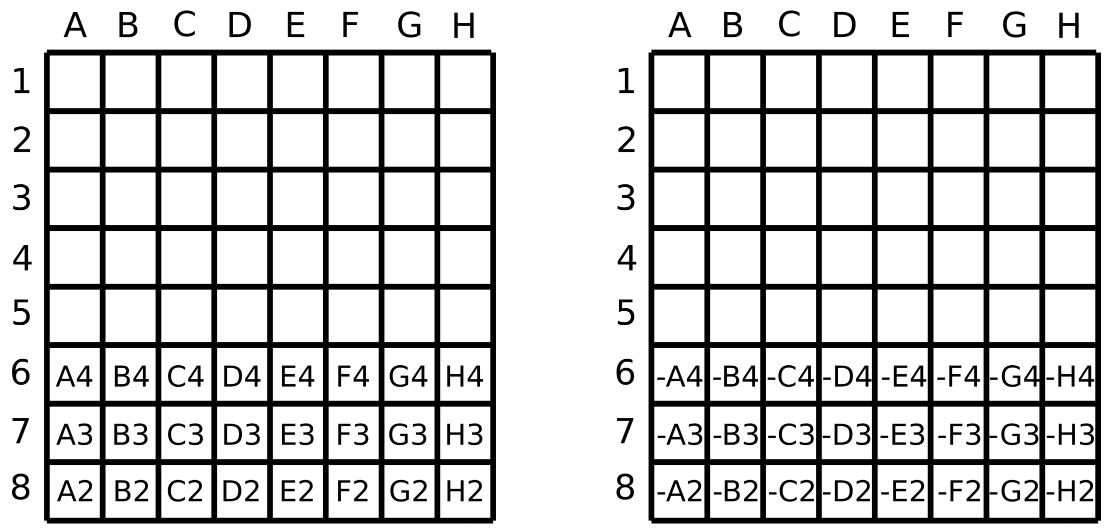


Abbildung 2.4: Veranschaulichung der Berechnung der DFT mit reellen Eingangswerten



(a) Realteil

(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.

bereits erwähnt bei der reellen Berechnung zusätzlicher Speicher erforderlich wäre, wird dieses Verfahren angewandt.

2.6.3 Berechnung der Diskreten Fouriertransformation mittels FFT

Die Mathematiker Cooley und Tukey haben einen Algorithmus entwickelt und im Jahr 1965 veröffentlich, 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 vermeindliche 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. Abbildung (2.6) illustriert dies 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.

Aus Gleichung (2.13) 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) aus Abschnitt (3.2) 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 ergibt sich bei einer DFT mit 2 Eingangswerten die Twiddlefaktoren 1 und -1, sodass eine Multiplikation entfällt.

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 Zweite 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. Bedeutend wichtiger ist jedoch, 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 Symmentriegründen 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 wiederum insgesamt acht reellen Multiplikationen entspricht.

Wie gezeigt wurde, werden nur zwei komplexe Multiplikationen benötigt. Eine Abschätzung der benötigten komplexen Multiplikationen erhält man mit der Gleichung (2.20):

$$\frac{N}{2}\log_2(N) = \frac{8}{2} \cdot 3 = 12 \tag{2.20}$$

Insbesondere bei größeren FFTs ist die relative Abweichung bedeutend geringer.

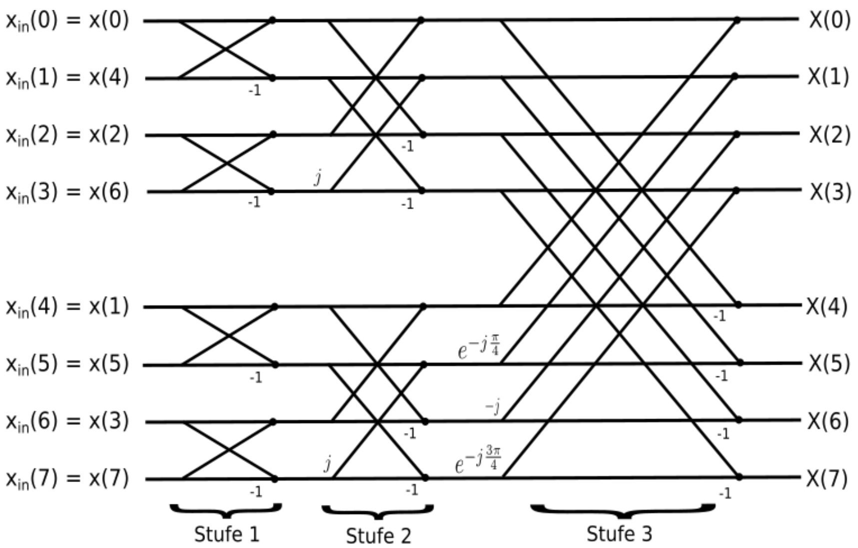


Abbildung 2.6: Berechnungsschema der DFT mit 8 Eingangswerten nach dem Butterfly-Verfahren

2.6.4 Inverse DFT

Die Inverse Diskrete Fouriertransformation (IDFT) ist die Umkehrfunktion der DFT. Wenn das Eingangssignal x zeitabhängig und somit als $\vec{x}(t)$ geschrieben werden kann, dann handelt es sich bei X um dessen Darstellung im Frequenzbereich und kann als $\vec{X}(f)$ geschrieben werden. Mit der IDFT ist es möglich aus der Frequendarstellung das Zeitsignal zu errechnen.

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

beschrieben. 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).

2.7 Diskrete Kosinus Transformation (DCT)

2.7.1 Verwendung der DCT

Die DCT findet häufig in der Bildverarbeitung Anwendung,

2.7.2 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 1. 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

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

Die Twiddlefaktormatrix kann in Matlab mit

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

berechnet werden. Da die Diskrete Cosinus 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 nur eine Multiplikationseinheit nötig ist. Gleiche Faktoren gehen 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 die selbe Anzahl der Sensoren wie das derzeitige Demo-Array, so dass 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 ist diejenige, die sich durch Interpolation der Daten einfach errechnen lässt. Die 12x12 sowie die 16x16 werden zum besseren Einordnen der Bewertungen ebenfalls betrachtet. Darüber hinaus ist aus Abschnitt 2.6.3 kekannt, dass die FFT auf 2^n Elementen basiert und es sich hierbei um ein sehr schnelles und effizientes Verfahren handelt.

Die Bewertung berücksichtigt wie bereits angedeutet die beiden Eigenschaften Anzahl verschiedener Faktoren und die gesamte Anzahl an Faktoren, wovon die erst genannte eine höhere Priorität hat. Bei den Faktoren wird zwischen solchen unterschieden, die als trivial erachtet werden, da sie nur eine Addition bedürfen (± 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, welches die als trivial eingestuften Werte mit wenigen Bits darstellt. Um z.B. einen Wert durch 2 zu teilen, erfolgt im dualen einfach ein Bitshift um 1 nach rechts.

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

3.1 Bewertung verschiedener DCT-Größen

In Tabelle 3.1 ist die Gegenüberstellung der genannten Größen zu sehen. Für die Bewertung wurd 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×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

3.2 Bewertung verschiedener DFT-Größen

In der Tabelle 3.2 werden die DFT-Matrizen einander gegenüber gestellt. Anders als die DCT hat die DFT

Die Beurteilung basiert auf dem Matlab-Skript aus Anhang 8.2. Die beiden Skripte unterscheiden sich neben den Koeffizienten darin, dass die Twiddlefaktormatrizen der DFT komplex sind.

Tabelle 3.2: Bewertung der DFT-Twiddlefaktor-Matrizen

N	8	9	12	15	16
N×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 triviale Werte werden 0, $\pm$0,5 sowie $\pm$1 aufgefasst. Andere Werte die sich gut binär darstellen lassen treten nicht auf. Alle übrigen Werte werden als nicht trivial betrachtet, da eine Multiplikation mit ihnen eine aufwändigere Berechnung bedeutet.

Bei der 8×8 Matrix gibt es, wie in Grafik 3.1 zu sehen, als nicht trivialen Wert mit $|\sqrt{2}/2|$ für Real- und Imaginärteil nur einen einzigen Wert, welche dazu noch gemeinsam auftreten. 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. Darüberhinaus ist dies auch der einzige Wert, der sowohl einen Real- aus auch einen Imaginärteil besitzt. Alle anderen Faktoren haben in einem von beiden Teilen $|1|$ und somit im anderen Teil 0.

In der bereits erwähnten Grafik 3.1 sind zur Veranschaulichung alle möglichen Zeiger der Twiddlefaktoren ($W_{m,n}$) für die 8×8 Matrix dargestellt. Berechnet werden diese mit der Gleichung (2.13), wobei es sich bei N um die Anzahl der Elemente im Vektor bzw. der Spalte einer Matrix von Werten im Zeitbereich handelt. n ist der Laufindex über die einzelnen Elemente, m das Äquivalent für den zu berechnenden Vektor (Matrixspalte) im Frequenzbereich. Beide fangen bei 0 an und laufen entsprechend bis $N-1$.

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 können. Die andere Hälfte muss über die Bildung des 2er-Komplements lediglich negiert werden, was ein bedeutend geringerer Aufwand ist, als eine Multiplikation. Deshalb ist das berechnete Verhältnis von 3 in Tabelle 3.2 in Wirklichkeit deutlich höher und übertrifft mit 7 die 12×12 Matrix um den Faktor 2. Dies gilt unter der Annahme, dass die Bildung des 2er-Komplements nicht berücksichtigt wird, was zumindest einer besseren Näherung entspricht, als es als eine volle Multiplikation zu werten.

Hierzu Abschnitt Abschätzung des Rechenaufwandes?

Anfangs wurde angenommen, dass das 1er-Komplement eine gute Wahl sein könnte, da hierbei die Darstellung negativer Zahlen einzig durch Setzen des höchstwertigsten Bit erfolgt. Auf diese Weise könnte immer das selbe Resultat für den Imaginär- wie für den Realteil verwendet werden, das Vorzeichen würde sich über eine einfache XOR-Verknüpfung beider MSB ergeben. Diesem Vorteil steht jedoch eine komlexere Subtraktion (bzw. Addition negativer Zahlen) gegenüber. Der zusätzliche Aufwand entspricht etwa dem der Bildung des 2er-Komplements. Aus diesem Grund wurde sich für dieses entschieden, da es deutlich verbreiteter ist und weitere Vorteile bringt wie beispielsweise keine Doppeldeutigkeit durch eine negative Null hat.

In Abbildung 3.2 sind zur weiteren Veranschaulichung die komplexen Zeiger der Twiddlefaktoren dargestellt. Sie sind aufgeteilt auf 8 Einheitskreise, wobei jeder einen Laufindex (m) des Zeitbereichs abdeckt. In den einzelnen Kreisen sind wiederum alle Laufindizes (n) des Frequenzbereichs zu sehen.

Anhand der Gleichung (2.13) für die Twiddlefaktoren und des Einheitskreises in Abb. 3.1 lässt sich erkennen, dass die Zeiger im Gegenuhrzeigersinn rotieren und sich sowohl für den Realteil als auch den Imaginärteil gleichmäßig auf positive und negative Werte aufteilen. Das lässt sich ausnutzen, um keine Negationen der Eingangs- und Zwischenwerte erfolgen muss. Darüber hinaus minimiert sich bei geschickter Anordnung das Risiko eines Überlaufs. Da zur Sicherheit dennoch nach jeder Addition oder Subtraktion das Ergebnis durch einen Bitshift halbiert wird. Da über die Eingangswerte die Annahme getroffen werden kann, dass aufeinanderfolgende Werte das selbe Vorzeichen haben, kann hier noch weiter die Genauigkeit optimiert werden.

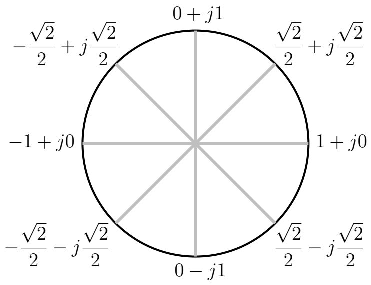


Abbildung 3.1: Einheitskreis mit relevanten Werten der 8x8-DFT

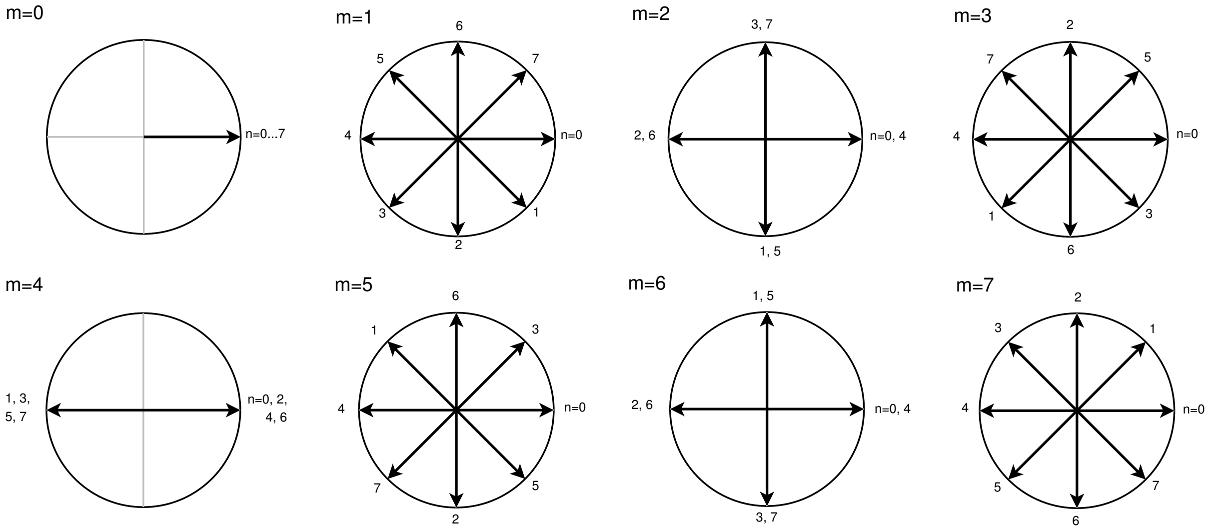


Abbildung 3.2: Twiddlefaktoren der 8×8-Matrix, aufgeteilt auf die Laufindizes m und n. m bezieht sich auf das Element im Ausgangsvektor $\vec{X}$, n auf den Eingangsvektor $\vec{x}$. Siehe auch Gl. (2.12)

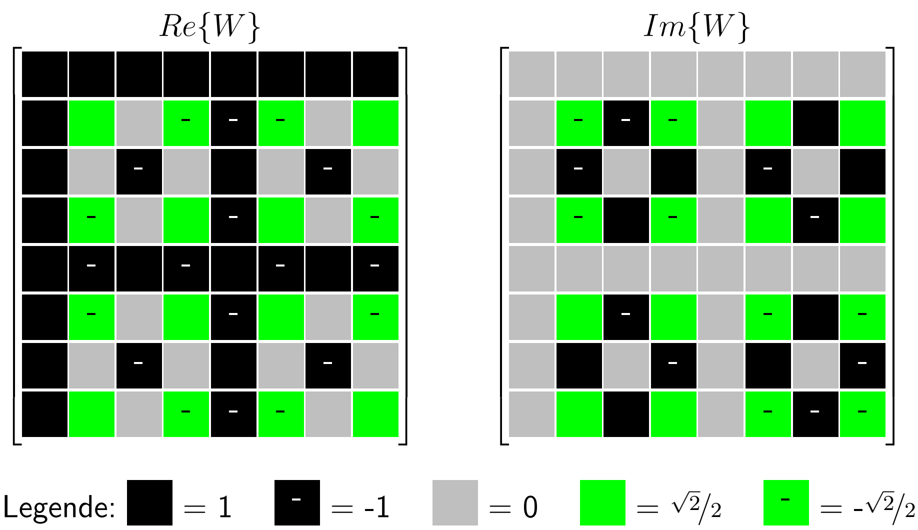


Abbildung 3.3: Matrix-Darstellung der 8x8-DFT-Twiddlefaktoren aufgeteilt nach Real- und Imaginärteil

Sowohl der Abbildung 3.2 als auch insbesondere der Darstellung 3.3 lassen sich sehr gut die Symmetrien erkennen, die diese Twiddlefaktormatrix so vorteilhaft machen.

3.3 Entscheidung DCT vs. DFT

Noch nicht fertig!

Sowohl die DCT als auch die DFT finden häufig in der Bildverarbeitung Anwendung. Der Vorteil der DCT gegenüber der DFT ist, dass sie rein reelle Ergebniswerte liefert. Ihr großer Nachteil zeigt sich u.a. insbesondere deutlich bei den 8x8-Matrizen, da sich hier

nicht trivial darstellbare Zahlen der DCT einem einzigen bei der 8x8-DFT gegenüber stehen.

Auch wenn bei der DFT mit der Berechnung des imaginären Teils zusätzlicher Implementierungsaufwamd hinzukommt, wird davon ausgegangen, dass dieser geringer ist, als alle x Multiplikationen umzusetzen. Ebenso ist die Annahme, dass der Platzbedarf auf einem Chip in einer ähnlichen Größenordnung liegt, da auf der einen Seite der zusätzliche Speicherbedarf für eine weitere Matrix den x Konstantenmultiplizierer-Schaltnetzen gegenüber stehen.

Es ist nicht geklärt, welche Berechnung für eine Weiterverarbeitung sinnvoller ist. Dies heraus zu finden ist jedoch nicht Bestandteil der Aufgabenstellung dieser Arbeit. An dieser Stelle sollen lediglich Vor- und Nachteile zusammengetragen werden, die eine Entscheidung rechtfertigen.

Ein Einsatzszenario der Transformationen ist die Filterung von Rauschen und anderen Störgrößen. Hierfür ist die DFT gut geeignet.

Da es bei dieser Arbeit vor allem um die Aufwandsabschätzung einer optimierten Matrizenmultiplikation zur Vorverarbeitung der Sensordaten geht, welche als Ausgangspunkt für eine finale Implementation dient, und es sich hier um keine endgültige Entscheidung handelt, ist die DFT gut geeignet.

Tabelle 3.3: Gegenüberstellung der Vor- und Nachteile von DCT und DFT

Eigenschaft	Vorteil	Nachteil
Imaginärteil Vorhanden	DCT	DFT
Anzahl Multiplikationen	DFT	DCT
Platzbedarf	-	-

3.4 Abschätzung des Rechenaufwands

3.4.1 Gegenüberstellung von reellen und 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. Auf diese Weise lässt sich die Berechnung mathematisch kompakter schreiben.

In Tabelle (3.4) 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 Ausummieren erfolgt, kann in Abschnitt (4.7) detaillierter nachgelesen werden.

Wie in Abschnitt (4.4) gezeigt wird, kann die Multiplikation mit einer Konstanten innerhalb eines Taktes mit einem Schaltnetz erfolgen. Anders als bei der komplexen Multiplikation mit der Twiddlefaktormatrix sind bei der getrennten Berechnung 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. Um keine zu langen Signal- und Gatterlaufzeiten hervor zu rufen, sollte hierfür ebenfalls ein Takt eingeplant werden, wordurch der zeitliche Gewinn wiederum etwas relativiert wird.

Tabelle 3.4: 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)·4·8 = 256 Takte sowohl für den Real- als auch den Imaginärteil der komplexen Ausgangsmatrix. Real- und Imaginärteil werden parallel berechnet und sind somit zeitgleich fertig.

Wie ein Vergleich der Gleichungen (2.2) und (3.1) zeigt, entfallen die Hälfte der Multiplikationen, wenn die Eingangswerte in Real- und Imaginärteil getrennt werden. Wenn die Eingangswerte rein reell sind, kommen beispielsweise keine j^2-Komponenten zustande, welche auf die reellen Elemente aufaddiert werden müssten. Aus diesem Grund müssen weniger Werte aufsummiert werden, wie sich in Tabelle (3.5) zeigt.

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

Aus Abschnitt (2.6.2) ist bekannt, dass die letzten drei Zeilen direkt oder negiert aus den Zeilen 2-4 übernommen werden können. Die Takte der 6.-8. Zeilen sind deshalb in der Tabelle (3.5) grau hinterlegt. Gegenüber der komplexen Matrix ergeben sich hier statt 256 Takten (3+4+2+4+3)·8 = 128 Takte. Der Imaginärteil errechnet sich noch schneller, da die 1. und

Tabelle 3.5: Takte für die reelle DFT am Beispiel der reellen Ausgangsmatrix

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

5. Zeile keinen Beitrag leisten und auch hier die Zeilen 2-4 in diesem Fall nach einer Negation die Werte der letzten 3 Zeilen ergeben. So ergeben sich dort (3+2+3)·8=64 Takte. Vermultich müssen an dieser Stelle wieder Takte für das Negieren eingeplant werden. Da beide parallel berechnet werden, sind die hierfür benötigten Takte sozusagen frei verfügbar.

Interessant ist dieser Ansatz dann, wenn einerseits die Recheneinheit so klein wie irgend möglich gehalten werden soll und andererseits die Berechnung noch schneller erfolgen muss. Abbildung (2.4) zeigt, dass im Vergleich zur komplexen Berechnung der 2D-DFT voraussichtlich 3x so viel Speicher für Zwischenwerte vorhanden sein muss. Ingesamt übersteigt so der Flächenbedarf der gesamten Einheit der der komplexen Variante. Auch die Leitungen um den Speicher anzubinden dürfen nicht vernachlässigt werden.

3.4.2 Direkte Multiplikation zweier 8x8 Matrizen

Die in Abschnitt (2.3) erläuterte Matrixmultiplikation bedarf bei einer 8x8 Matrix je Ergebnis der Ausgangsmatrix 8 Multiplikationen. Für die 8·8=64 Elemente werden deshalb 512 Multiplikationen benötigt. Da es sich sowohl bei den Eingangswerten als auch bei der Twiddlefaktormatrix um komplexe Zahlen handelt, sind, wie in Abschnitt (2.2) beschrieben, insgesamt 512·4=2048 Multiplikationen nötig.

Sollte sich dazu entschieden werden die Sinus- und Kosinusanteile separat zu berechnen, um ein rein reelles Eingangssignal weiter zu verarbeiten, sind, wie in Abschnitt (2.6.2) hergeleitet, knapp die Hälfte der Multiplikationen unnötig. In Abbildung (2.5) ist zu sehen, dass von den 64 Ergebniswerten nur 40 berechnet werden müssen. Da die Eingangswerte zwar rein reell, die Twiddlefaktormatrix aber komplex ist, verdoppelt sich die Anzahl der Multiplikationen. Somit müssen für die gesamten 64 Werte 40·8·2=640 Multiplikationen durchgeführt werden.

Im komplexen Fall verdoppelt sich für die 2D-DFT schlicht die Anzahl der reellen Multiplikationen und liegt somit bei 4096. Im reellen Fall müssen, wie in Abbildung (2.4) gezeigt, der Real- sowie der Imaginärteil separat mit der Twiddlefaktormatrix multipliziert werden. So ergeben sich alles in allem 640·3·2=3840 reelle Multiplikationen. Diese Zahl liegt nur nur geringfügig unterhalb der komplexen Berechnung.

Hierbei wird von einer Twiddlefaktormatrix mit 64 komplexen Werten ausgegangen. In Wirklichkeit sind es nur 16, die übrigen erfordern überhaupt keine Multiplikation, da entweder der Real- oder der Imaginärteil 0 ist. Da dies aber Bestandteil der optimierten Matrixmultiplikation ist, wird an dieser Stelle nicht weiter darauf eingegangen. Später werden nur die komplexen Varianten verglichen. Dies wird als ausreichend erachtet, da aufgrund der hier und in Abschnitt (2.6.2) angedeutete deutlich erhöhte Bedarf an Takten die reelle Matrixmultiplikation nicht von Interesse ist.

3.4.3 Optimierte 8x8 DFT als Matrixmultiplikation

Aus der anfänglichen Implementation bei der alle Werte einer Berechnung die entweder mit $+\frac{\sqrt{2}}{2}$ oder $-\frac{\sqrt{2}}{2}$ multipliziert werden müssen einzelnd berechnet werden, wird sinngemäß der gemeinsame Faktor ausgeklammert, sodass nur noch jeweils eine Multiplikation erforderlich ist.

Da die erste Zeile der Twiddlefaktormatrix nur aus Einsen im Real- und Nullen im Imaginärteil besteht, kann und muss hier nichts optimiert werden. 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 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

3.4.4 Gegenüberstellung von Butterfly-Algorithmus und optimierter Matrixmultiplikation

Die DFT wurde als Matrixmultiplikation implementiert, um die gewonnenen Erkenntnisse auch auf andere Dimensionen als 2^n, insbesondere ungerade, übertragen zu können. Zu einem frühen Zeitpunkt der Überlegungen für diese Arbeit gab es noch die Idee die DFT so flexibel wie möglich zu halten, um unkompliziert auf andere Größen wechseln zu können. Hierfür sollten alle Koeffizienten der Twiddlefaktormatrix ladbar sowie die Größe der Matrix über eine globale Deklaration definierbar sein. Diese Herangehensweise bedingt die Implementation als Matrixmultiplikation. Die Hoffnung der Projektgruppe bestand darin, dass das Synthesewerkzeug den VHDL-Code soweit optimiert, dass dies nicht händisch erfolgen müsste. Als klar war, dass die Optimierung nicht so tief greift, wurden die entsprechenden Schritte manuell umgesetzt.

Die Implementierung des Butterfly-Algorithmus nach Cooley und Tukey wurde bereits in Grafik (2.6) gezeigt. Sie stellt eine effiziente Berechnung der DFT dar, in Abschnitt (3.4.3) konnte gezeigt werden, dass sich beide nur unwesentlich im Rechenaufwand unterscheiden.

3.5 Kompromiss aus benötigter Chipfläche und Genauigkeit des Ergebnisses

Durch die Begrenzung der Bitbreite ist es nötig nach jeder Addition den Wert zu halbieren. Hierbei steigt die Abweichung gegenüber einer verlustfreien Berechnung immer dann, wenn das letzte eine 1 ist. Im Mittel ist dies bei der Hälfte der Additionen der Fall. In 50% aller Fälle wird also der Wert um ein halbes LSB zu viel verringert. Bei der Multiplikation verdoppelt sich sogar die resultierende Bitbreite. Da mit dem vollständigen 13 Bit Vektor nach der Addition weitergerechnet wird, muss die Konstante ebenfalls in 13 Bit hinterlegt sein. Deshalb hat das Ergebnis 26 Bit, von denen für die weitere Berechnung wieder nur 12 übernommen werden. In den Abbildungen (4.4) und (4.5) wird das hier beschriebene Vorgehen veranschaulicht. Bei diesem Verfahren kommt es unweigerlich zur Akkumulation von Fehlern.

Da für die Berechnung einer Zahl der 1D-DFT je nach Zeile entweder 8 oder 12 Werte akkumuliert sowie 0 bis 4 Werte multipliziert werden und für die 2D-DFT entsprechend doppelt so viele, akkumulieren sich zwangsläufig Fehler. Bei 12 Bit Eingangswerten wäre ein 47? Bit Ausgangsvektor nötig, um dies vollständig zu vermeiden. Dies ist jedoch aus u.a. Platzgründen nicht umsetzbar.

Mit jeder Addition kommt 1 bit dazu. So werden aus 12 Bit bis zur Multiplikation 15 $(12 + \log_2(8))$, 8 = Anzahl der Zahlen die mit $\frac{\sqrt{2}}{2}$ multipliziert werden müssen. Bei der Multiplikation verdopplet sich der Wert, also 30 und eine letzte Addition macht 31. Beim zweiten Durchlauf werden es so $(31+3)\cdot 2+1=69$ Bit.

$\Rightarrow$ Anhand eines Simulationsbeispiels zeigen, dass die mit VHDL berechneten Werte immer kleiner als die in Matlab berechneten sind.

4 Entwurf

4.1 Interpretation binärer Zahlen

Matlab fi

immer 10 Nachkommastellen, außer bei Multiplikation

NC Sim, nur Integerdarstellung möglich, bei Vektoren sogar nur positiv

4.2 Entwicklungsstufen

4.2.1 Multiplikation

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

4.2.2 Addierer

CLA, RC, in einem Takt

4.2.3 Konstantenmultiplikation

Dieser Punkt muss irgendwie mit der Implementierung des Konstantenmultiplizierers zusammengeführt werden.

Der duale Wert lässt sich am einfachsten mit der Matlab-Funktion `fi()` ermitteln. Der Funktion werden hierfür Kommagetrennt der Deziamlwert, 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 (4.1) kann entnommen werden, dass die Abweichung weit unter einem Prozent liegt.

$$\frac{100}{\frac{\sqrt{2}}{2}} \cdot 0,70703125 = 99,989\% \tag{4.1}$$

4.2.4 1D-DFT mit Integer-Werten

4.2.5 2D-DFT mit Integer-Werten

4.2.6 2D-DFT mit Werten SQ-Format

4.2.7 Zusammenhang von DFT und IDFT bei der Matrixmultiplikation

4.3 Test der Matrixmultiplikation

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.

${\frac{\sqrt{2}}{2}}_{10} = 001011010100_2$ 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!)

4.4 Implementierung des Konstantenmultiplizieres

Anfangs wurde angenommen, dass Multiplikationen mit den Twiddlefaktoren ± 1 und $\pm\frac{\sqrt{2}}{2}$ durchgeführt werden müssen. Dass bei einer optimierten 8x8-DFT wegen des explizieten ausprogrammierens der Berechnungen die Multiplikation mit ± 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.

4.4.1 Syntheseergebnis eines 13 Bit Konstantenmultiplizierers

Der vollständige Gate-Report befindet sich in Abschnitt 8.3 auf Seite 49

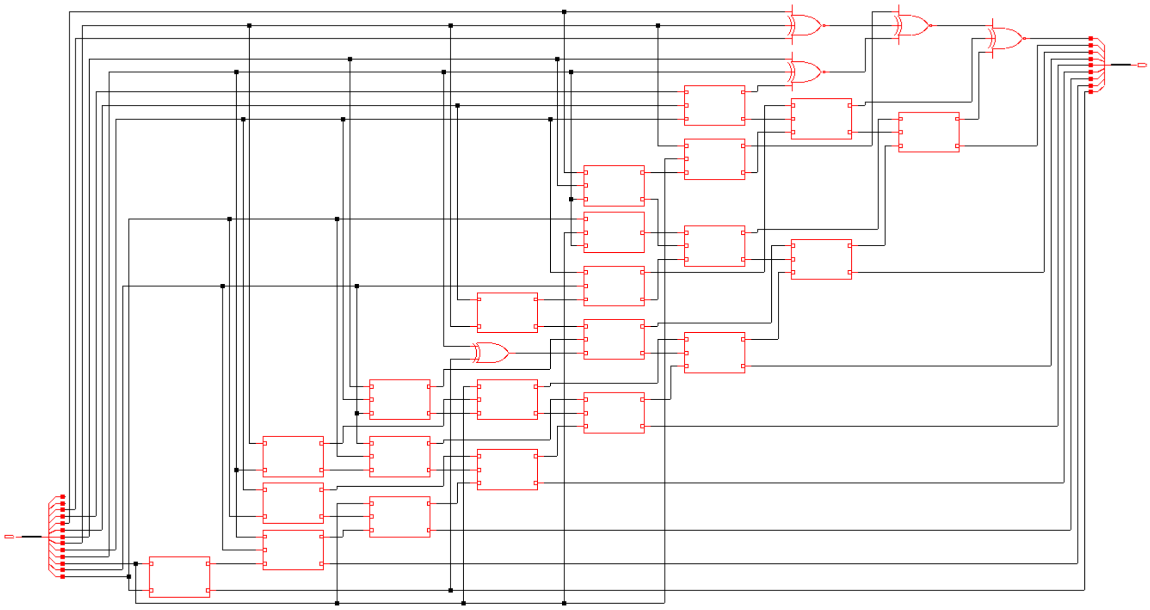


Abbildung 4.1: 13 Bit Konstantenmultiplizierer für $\frac{\sqrt{2}}{2} = 0.70711 \simeq 0.70703125 = 0001011010100_2$ in Encounter; Eingang links, Ausgang rechts

Tabelle 4.1: Vergleich Konstanten- mit regulärem Multiplizierer

	Konstantenmultiplizierer	regulärer Multiplizierer
Gatter	27	175
Fläche (Prozess: 350nm)	6612 um^2	23 261 um^2

4.4.2 Syntheseergebnis für die Bildung des Zweierkomplements eines 13 Bit Vektors

Zum Vergleich soll hier die nicht implementierte aber in Abschnitt 3.4.1 erwähnte Negierung von Zahlen gezeigt werden.

Für die Negierung eines 13 Bit Vektors mit 22 Standardzellen sind knapp doppelt so viele Gatter nötig wie der Vektor Bits breit ist. Wie zu in Abb. 4.2 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.

4.5 Entwickeln der 2D-DFT in VHDL

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.

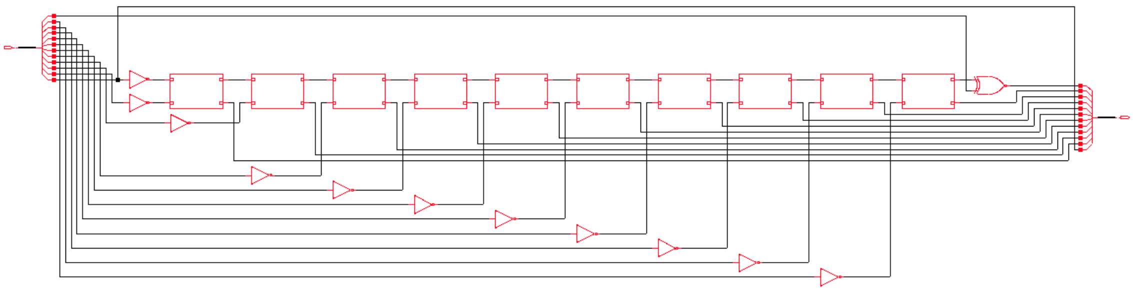


Abbildung 4.2: Netzliste einer Einheit zur Bildung des 2er-Komplements eines 13 Bit Vektors; Eingang links, Ausgang rechts

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

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

$$\begin{aligned} X &= W \cdot x \cdot W \\ &= \left(x^T \cdot W\right)^T \cdot W \qquad (4.2) \\ &= X^* \cdot W \\ &= \left((x \cdot W)^T \cdot W\right)^T \qquad (4.3) \\ &= \left(X^{*T} \cdot W\right)^T \end{aligned}$$

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 (4.2) und (4.3) 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 (4.3) ist das hier beschriebene veranschaulicht.

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

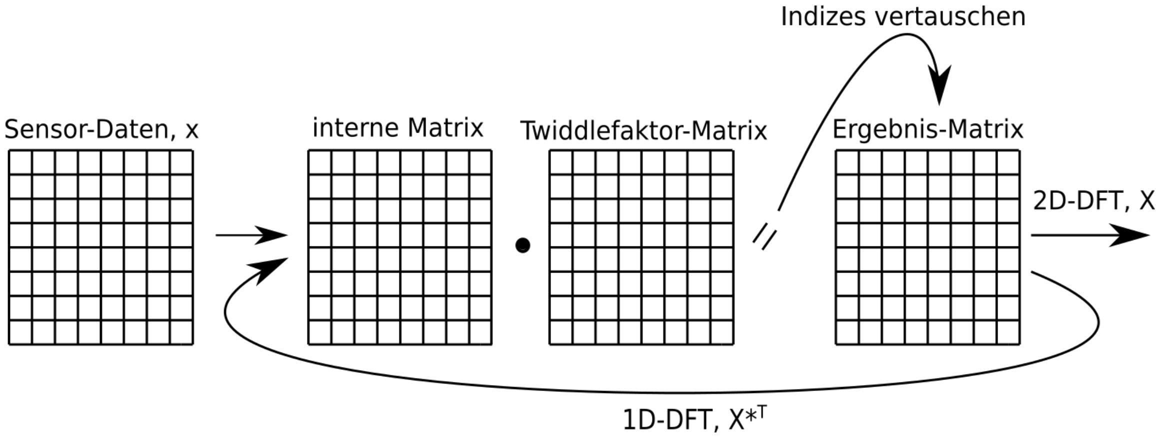


Abbildung 4.3: Darstellung der Berechnung der 2D-DFT aus Gleichung (4.3)

4.6 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 $64 \cdot 2 \cdot 12$ Bit = 1536 Bit = 1,5kBit = 192 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.

4.7 Berechnungsschema der geraden und ungeraden Zeilen

In Abbildung (4.4) ist die Berechnung der ungeraden Zeilen am Beispiel der ersten zu sehen.

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 alo ein doppelter Bitshift vollzogen. Dies erfolgt,

$$a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$$

Takt		Bit
1	$\underbrace{a_{k0} + a_{k1}} \quad \underbrace{a_{k2} + a_{k3}} \quad \underbrace{a_{k4} + a_{k5}} \quad \underbrace{a_{k6} + a_{k7}}$	12
	$\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow$	13
2	$\underbrace{sum\_1\_1 \; + \; sum\_1\_2} \quad \underbrace{sum\_1\_3 \; + \; sum\_1\_4}$	12
	$\Downarrow \quad \Downarrow$	13
3	$\underbrace{sum\_2\_1 \quad + \quad sum\_2\_2}$	12
	$\Downarrow$	13
	sum_3_1	12

Abbildung 4.4: Vorgehensweise der Akkumulation der ungeraden Spalten der Eingangswerte

damit sowohl in den geraden als auch den ungeraden Zeilen gleich viele Bitshifts erfolgen und die Werte somit identisch skaliert sind.

Die Berechnung der geraden Zeilen wird in Abbildung (4.5) am Beispiel der zweiten Zeile gezeigt

$$a_0 - x_1 + x_0 - b_2 + x_2 - x_3 + a_4 - x_5 + x_4 - b_6 + x_6 - x_7$$

Takt		Bit
1	$\underbrace{a_0 - x_1} \quad \underbrace{x_0 - b_2} \quad \underbrace{x_2 - x_3} \quad \underbrace{a_4 - x_5} \quad \underbrace{x_4 - b_6} \quad \underbrace{x_6 - x_7}$	12
	$\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow$	13
2	$\underbrace{sum1\_1 + sum1\_2} \quad \underbrace{sum1\_3 + sum1\_4} \quad \underbrace{sum1\_5 + sum1\_6}$	12
	$\Downarrow \quad \Downarrow \quad \Downarrow$	13
3	$\underbrace{sum2\_1 \quad + \quad sum2\_2}$	12
	$\Downarrow \qquad \Downarrow$	13
	$sum3\_1$	13
	$\Downarrow \qquad \Downarrow$	13
4	$\frac{\sqrt{2}}{2}$	13
	$\Downarrow \qquad \Downarrow$	26
5	$\underbrace{sum4\_1 \quad + \quad sum2\_3}$	12
	$\Downarrow$	13
	$sum5\_1$	12

Abbildung 4.5: Vorgehensweise der Akkumulation der geraden Spalten der Eingangswerte

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.

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 bei dem zweiten Durchlauf, um die 2D-DFT zu berechnen, ebenfalls durch 16 geteilt wird, ergibt sich insgesamt eine Division durch $2^{2 \cdot 4} = 256$.

4.7.1 Erwartete Anzahl benötigter Takte

Aus den Abbildungen 4.4 und 4.5 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.

4.8 Schema der Zustandsfolge

test test

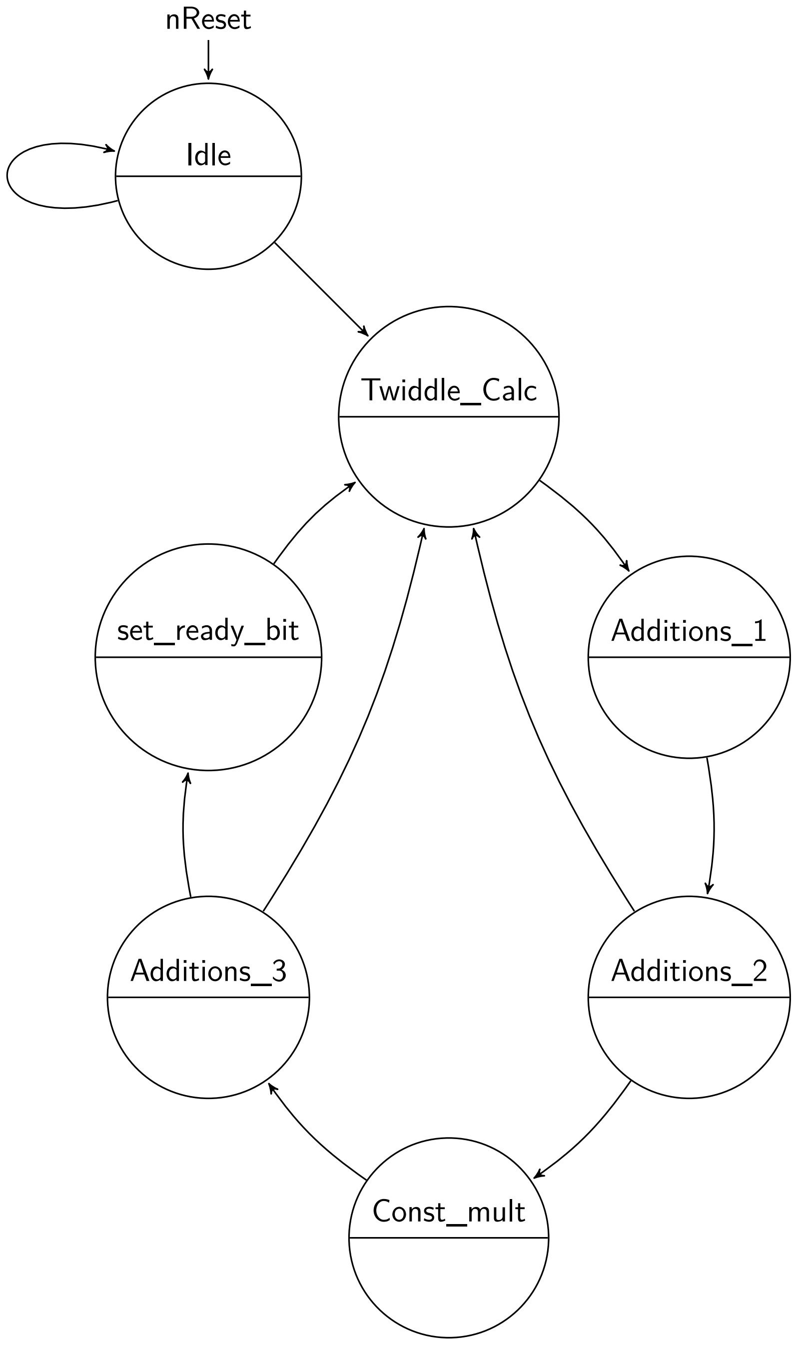
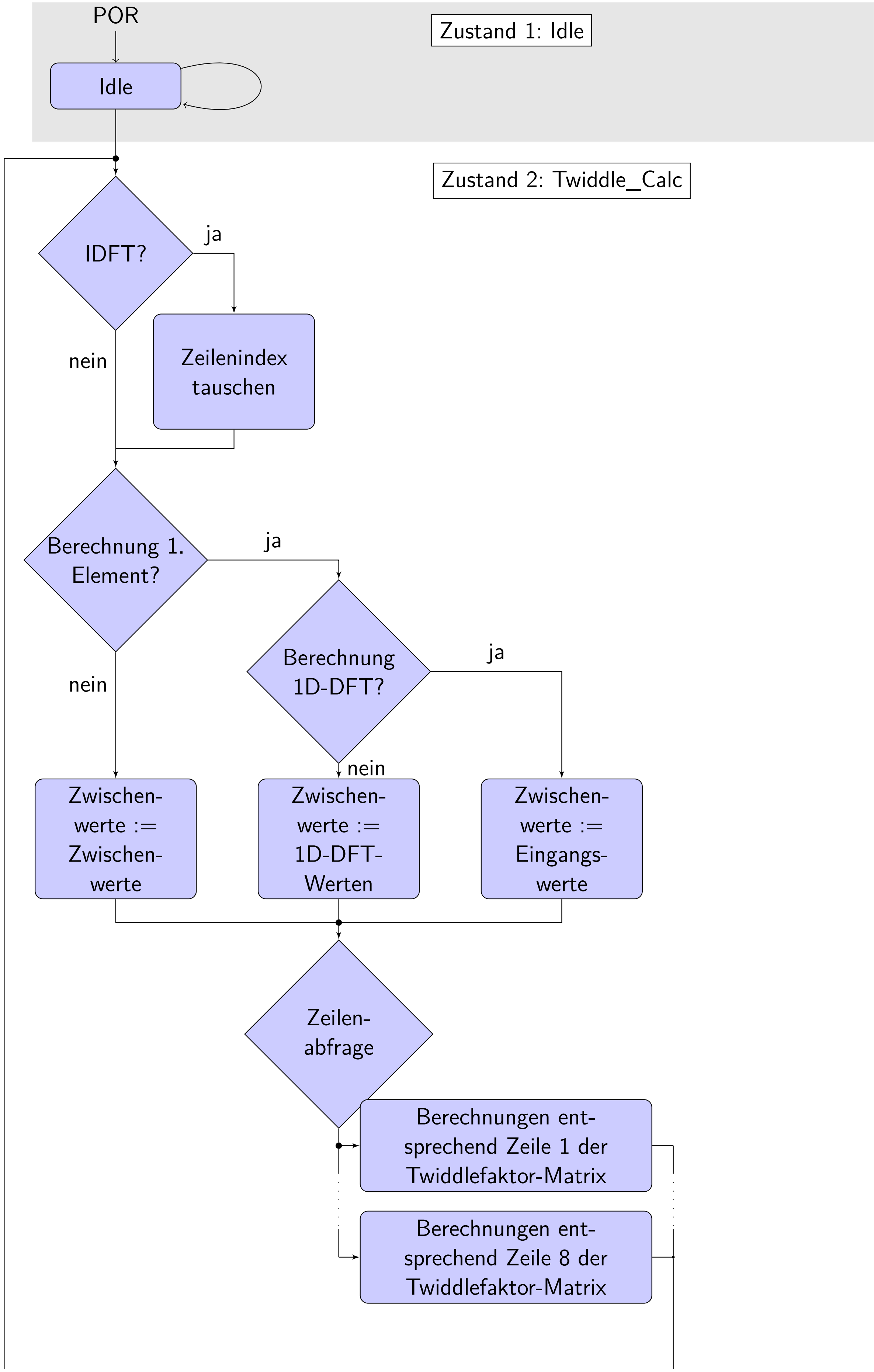
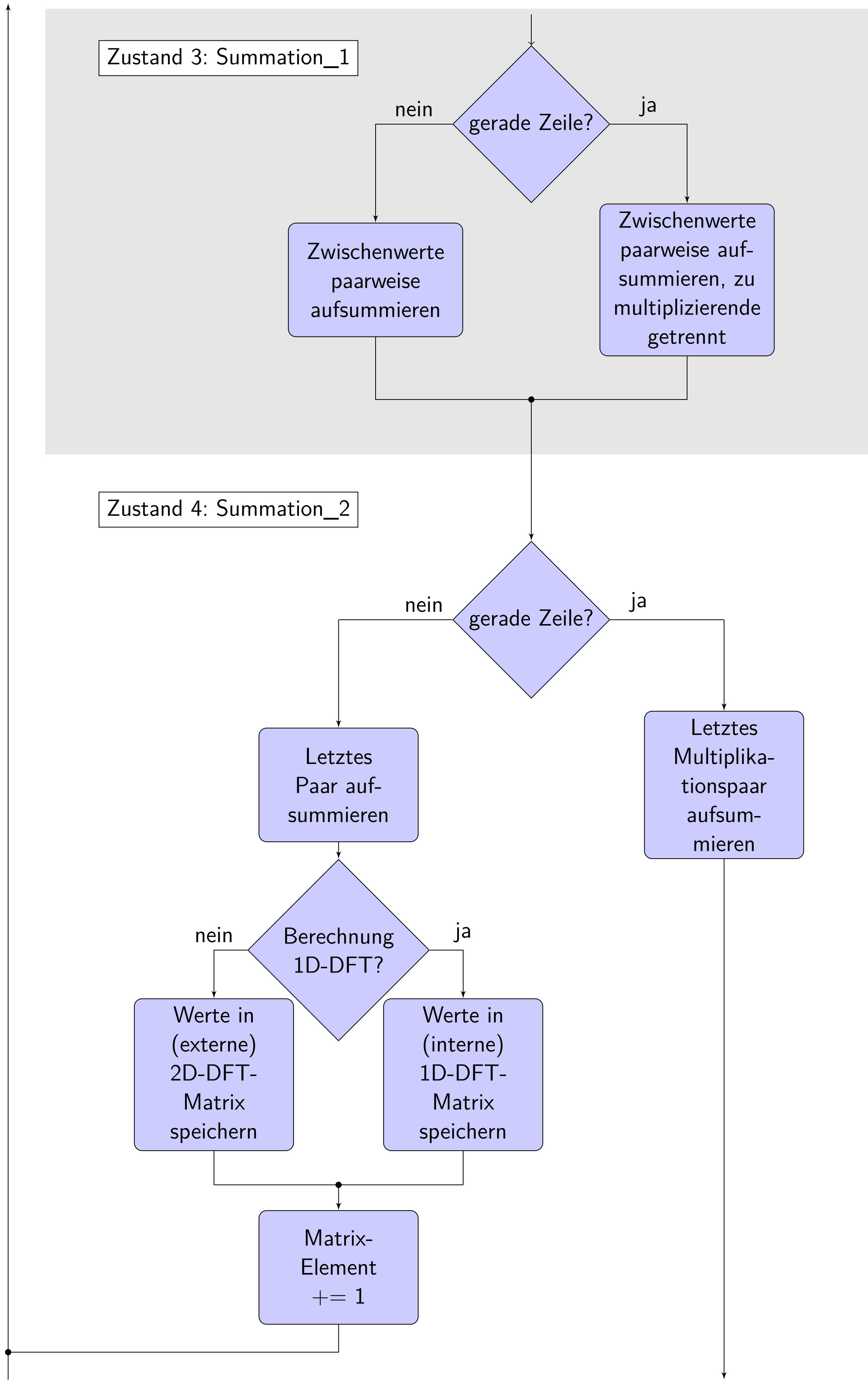
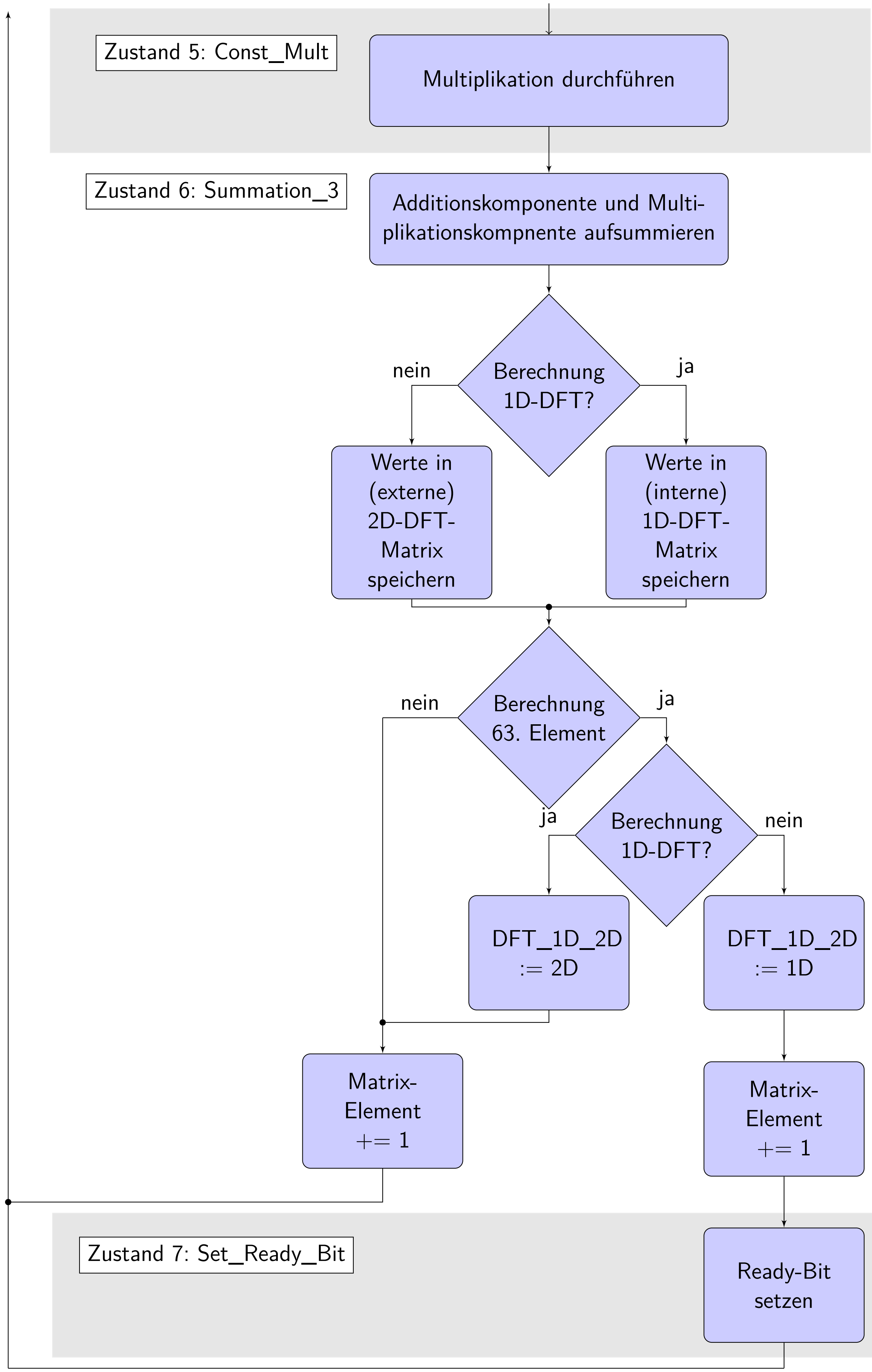


Abbildung 4.6: Automatengraf

4.9 UML-Diagramm




Zustand 3: Summation_1
gerade Zeile?
nein
ja
Zwischenwerte paarweise aufsummieren
Zwischenwerte paarweise auf-summieren, zu multiplizierende getrennt
Zustand 4: Summation_2
gerade Zeile?
nein
ja
Letztes Paar auf-summieren
Letztes Multiplika-tionspaar aufsum-mieren
Berechnung 1D-DFT?
nein
ja
Werte in (externe) 2D-DFT-Matrix speichern
Werte in (interne) 1D-DFT-Matrix speichern
Matrix-Element += 1


Zustand 5: Const_Mult
Multiplikation durchführen
Zustand 6: Summation_3
Additionskomponente und Multiplikationskompnente aufsummieren
Berechnung 1D-DFT?
nein
ja
Werte in (externe) 2D-DFT-Matrix speichern
Werte in (interne) 1D-DFT-Matrix speichern
Berechnung 63. Element
nein
ja
Berechnung 1D-DFT?
ja
nein
DFT_1D_2D := 2D
DFT_1D_2D := 1D
Matrix-Element += 1
Matrix-Element += 1
Zustand 7: Set_Ready_Bit
Ready-Bit setzen

4.10 Projekt- und Programmstruktur

Konstanten
 Datentypen
 readfile (read_input_matrix)
 writefile (write_results)
 resize-Funktion

4.11 Bibliotheken und Hardwarebeschreibungssprache

```
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))

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 Implemenatation 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 Geschwindikeit von $v_{max} = 250\,\text{KM/h}$ ausgegangen. Weiter wird ein realtiv 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

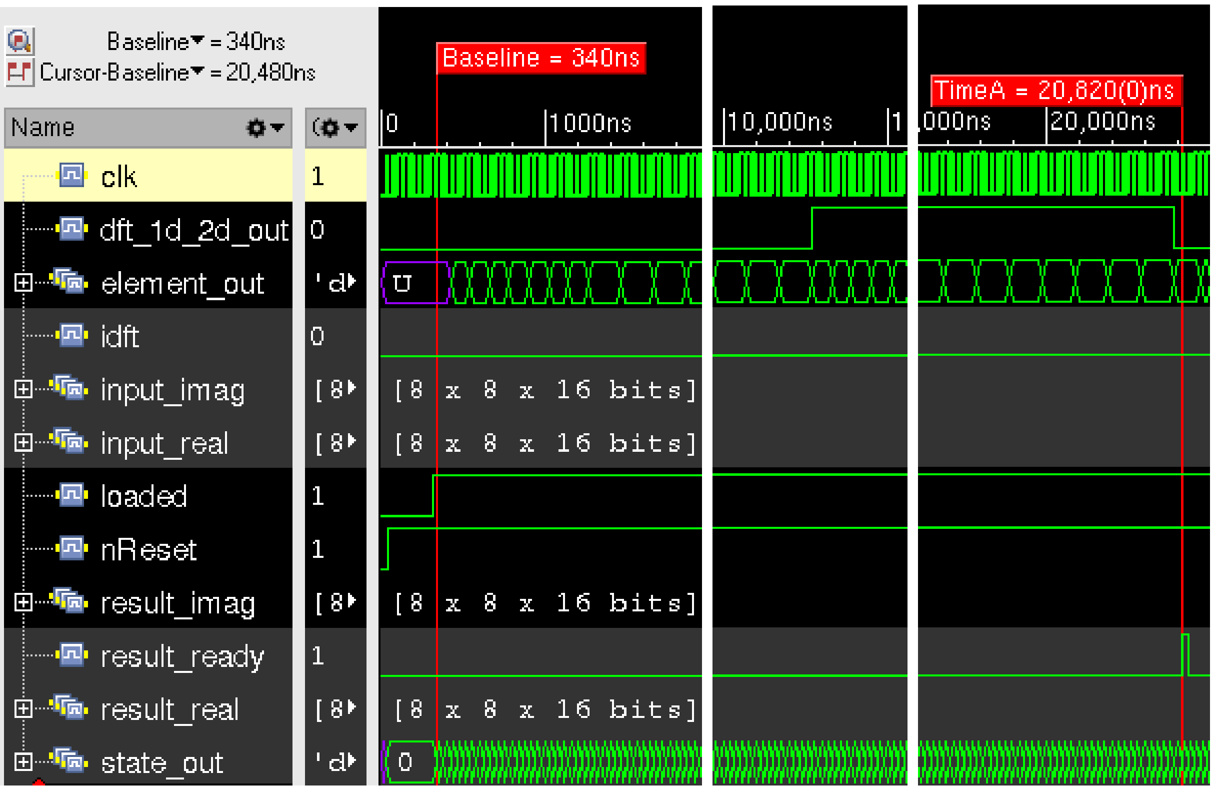


Abbildung 5.1: Ausschnitt des Simulationstools `NCSim` von der Berechnung und Verifikation der 2D-DFT

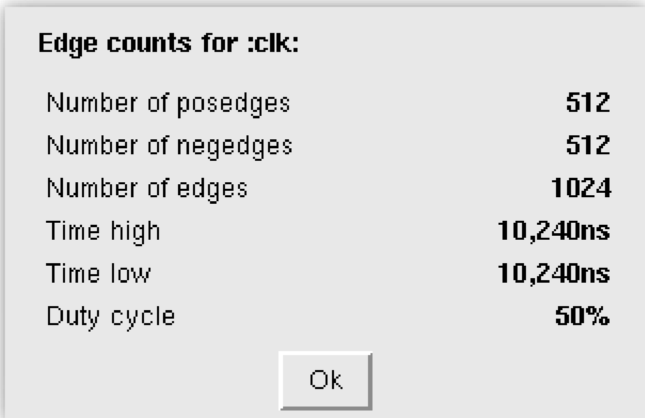


Abbildung 5.2: Edge Count des Taktsignals für die vollständige Simulation der 2D-DFT

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

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

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

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

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

Nun kann in (5.5) gezeigt werden, dass bei einer Winkelauflösung von 1° 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.

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

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.

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

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

1D-DFT eindimensionale diskrete Fouriertransformation
2D-DFT zweidimensionale diskrete Fouriertransformation

ADC Analog Digital Converter
ADU Analog Digital Umsetzer
AMR anisotroper magnetoresistiver Effekt
ASIC Application Specific Integrated Circuit, *dt.: Anwendungsspezifischer Integrierter Schaltkreis*

DCT Diskrete Cosinus Transformation
DFT Diskrete Fouriertransformation

FFT Fast Fouriertransformation
FT Fouriertransformation

IDFT Inverse Diskrete Fouriertransformation
ISAR Integrated Sensor Array

LSB Least Significant Bit

MSB Most Significant Bit

RAM Random Access Memory

TMR 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
2.6 Berechnungsschema der DFT mit 8 Eingangswerten nach dem Butterfly-Verfahren . . . 12

3.1 Einheitskreis mit relevanten Werten der 8x8-DFT . . . 17
3.2 Twiddlefaktoren der 8×8-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. (2.12) . . . 17
3.3 Matrix-Darstellung der 8x8-DFT-Twiddlefaktoren aufgeteilt nach Real- und Imaginärteil . . . 17

4.1 13 Bit Konstantenmultiplizierer für $\frac{\sqrt{2}}{2} = 0.70711 \simeq 0.70703125 = 0001011010100_2$ in Encounter; Eingang links, Ausgang rechts . . . 25
4.2 Netzliste einer Einheit zur Bildung des 2er-Komplements eines 13 Bit Vektors; Eingang links, Ausgang rechts . . . 26
4.3 Darstellung der Berechnung der 2D-DFT aus Gleichung (4.3) . . . 27
4.4 Vorgehensweise der Akkumulation der ungeraden Spalten der Eingangswerte . 28
4.5 Vorgehensweise der Akkumulation der geraden Spalten der Eingangswerte . . . 28
4.6 Automatengraf . . . 31

5.1 Ausschnitt des Simulationstools `NC Sim` von der Berechnung und Verifikation der 2D-DFT . . . 37
5.2 Edge Count für eine 2D-DFT . . . 37

Tabellenverzeichnis

3.1 Bewertung der DCT-Twiddlefaktor-Matrizen 15
3.2 Bewertung der DFT-Twiddlefaktor-Matrizen 15
3.3 Gegenüberstellung der Vor- und Nachteile von DCT und DFT 18
3.4 Takte für die komplexe DFT 19
3.5 Takte für die reelle DFT am Beispiel der reellen Ausgangsmatrix 20

4.1 Vergleich Konstanten- mit regulärem Multiplizierer 25

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

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

function dct_bewertung(N)

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

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


  % Anzahl verschiedener Werte ermitteln
   different_nums = unique(W);
   different_non_trivial_nums = different_nums(find(different_nums ~= 1));
   different_non_trivial_nums = different_non_trivial_nums(find(different_non_trivial_nums ~= -1));
   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));
   different_non_trivial_nums = different_non_trivial_nums(find(different_non_trivial_nums ~= 0));

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

```

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


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

  nums_of_matrix = N*N;

  trivial_nums = N*N - nontrivial_nums

  nontrivial_nums

  v = trivial_nums/nontrivial_nums

end
```

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

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

function dft_bewertung(N)
```

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

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

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



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

different_nums = [different_nums_real; different_nums_imag];
different_nums = unique(different_nums);
different_non_trivial_nums = different_nums(find(different_nums ~= 1));
different_non_trivial_nums = different_non_trivial_nums(find(
  different_non_trivial_nums ~= -1));
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));
different_non_trivial_nums = different_non_trivial_nums(find(
  different_non_trivial_nums ~= 0));

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

% 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)
  for n = 1:N
    for m = 1:N
      if different_nums_real(k) == W_r(m,n)
        num_count_real(k) = num_count_real(k) +1;
      end
    end
  end
end


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

```
      for m = 1:N
        if different_nums_imag(k) == W_i(m,n)
          num_count_imag(k) = num_count_imag(k) +1;
        end
      end
    end
  end


  % nicht triviale Werte der reellen Matrix z hlen
  nontrivial_nums_real = 0;
  for k = 1:length(different_nums_real)
    if abs(different_nums_real(k)) != 1
      if abs(different_nums_real(k)) != 0.5
        if different_nums_real(k) != 0
          nontrivial_nums_real = nontrivial_nums_real + num_count_real(k);
        end
      end
    end
  end

  % nicht triviale Werte der imagin ren Matrix z hlen
  nontrivial_nums_imag = 0;
  for k = 1:length(different_nums_imag)
    if abs(different_nums_imag(k)) != 1
      if abs(different_nums_imag(k)) != 0.5
        if different_nums_imag(k) != 0
          nontrivial_nums_imag = nontrivial_nums_imag + num_count_imag(k);
        end
      end
    end
  end

  nums_of_each_matrix = N*N;

  trivial_nums_real = N*N - nontrivial_nums_real
  trivial_nums_imag = N*N - nontrivial_nums_imag

  nontrivial_nums_real
  nontrivial_nums_imag

  trivial_nums_total = trivial_nums_real + trivial_nums_imag
  nontrivial_nums_total = nontrivial_nums_real + nontrivial_nums_imag

  v = trivial_nums_total/nontrivial_nums_total

end
```

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

8.2 Gate-Report des 12 Bit Konstatenmultiplizierers

```
rc:/> report gates
==========================================================================
  Generated by:            Encounter(R) RTL Compiler RC14.25 - v14.20-s046_1
  Generated on:            May 30 2017  03:29:41 pm
  Module:                  multiplier
  Technology library:      c35_CORELIB_TYP 3.02
  Operating conditions:    _nominal_ (balanced_tree)
  Wireload mode:           enclosed
  Area mode:               timing library
==========================================================================


 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
-----------------------------------------------
total           151   12885.600


  Type     Instances      Area    Area %
-----------------------------------------
inverter         33    1201.200     9.3
logic           118   11684.400    90.7
-----------------------------------------
total           151   12885.600   100.0

rc:/>
```

Listing 8.3: RC Gate-Report

8.3 Twiddlefaktormatrix im S1Q10-Format

```
%% Dateiname:          twiddle2file.m
%% Funktion:           Erzeugt eine Datei mit den binaeren komplexen
```

```
%%                       Twiddlefaktoren
%% Argumente:            N (Groesse der NxN DFT-Matrix)
%% Aufbau der Datei:     Wie die Matrix, enthaelt Realteil und Imaginaerteil.
%%                       Alle Werte sind wie im Beispiel durch Leerzeichen getrennt:
%%                       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)}
%% Abhaenigkeiten:       (1) twiddle_coefficients.m
%%                       (2) dec_to_s1q10.m
%%                       (3) bit_vector2integer.m
%%                       (4) zweier_komplement.m
%% Author:               Thomas Lattmann
%% Datum:                02.11.17
%% Version:              1.0

function twiddle2file(N)

  % Dezimale Twiddlefaktormatrix erstellen
  W_dec = twiddle_coefficients(N);
  W_dec_real = real(W_dec);
  W_dec_imag = imag(W_dec);

  W_bin_int_real = zeros(size(W_dec_real));
  W_bin_int_imag = zeros(size(W_dec_imag));

  for m = 1:N
    for n = 1:N
      bit_vector = dec_to_s1q10(W_dec_real(m,n));
      W_bin_int_real(m,n) = bit_vector2integer(bit_vector);

      bit_vector = dec_to_s1q10(W_dec_imag(m,n));
      W_bin_int_imag(m,n) = bit_vector2integer(bit_vector);
    end
  end

  fid=fopen('Twiddle_s1q10_komplex.txt', 'w+');

  for m=1:N
    for n=1:N
      fprintf(fid, '%012d ', W_bin_int_real(m,n));
      fprintf(fid, '%012d', W_bin_int_imag(m,n));
      if n < N
        fprintf(fid, ' ');
      end
    end
    if m < N
      fprintf(fid, '\n');
    end
  end

  fclose(fid);
```

```
end
```

Listing 8.4: Erstellen der Twiddlefaktormatrix-Datei

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

function W = twiddle_coefficients(N)

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

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

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

end
```

Listing 8.5: Erzeugen der Twiddlefaktormatrix

```
%% Dateiname:       dec_to_s1q10.m
%% Funktion:        Konvertiert eine Dezimalzahl in das binaere S1Q10-Format
%% Argumente:       Dezimalzahl im Bereich von -2...+2-1/2^10
%% Abhaenigkeiten: (1) zweier_komplement.m
%% Author:          Thomas Lattmann
%% Datum:           02.11.17
%% Version:         1.0

function bit_vector = dec_to_s1q10(val)

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

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

    % Vorkommastellen
    if abs(val) >= 1
      bit_vector(2) = 1;
      if val == -2
        bit_vector(1) = 1;
      end
    end

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

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

Listing 8.6: Dezimalzahl nach S1Q10 konvertieren

```
%% Dateiname: zweier_komplement.m
%% Funktion:  Bilden des 2er-Komplements eines "Bit"-Vektors
%% Argumente: Vektor aus Nullen und Einsen
%% Author:    Thomas Lattmann
%% Datum:     02.11.17
%% Version:   1.0

function bit_vector = zweier_komplement(bit_vector)
  bit_width=length(bit_vector);

  for j = 1:bit_width
    bit_vector(j) = not(bit_vector(j));
  end
  bit_vector(bit_width) = bit_vector(bit_width) + 1;
  for j = 1:bit_width-1
    if bit_vector(bit_width -j +1) == 2
      bit_vector(bit_width -j +1) = 0;
      bit_vector(bit_width -j) = bit_vector(bit_width -j) + 1;
    end
```

```
  end
end
```

Listing 8.7: Bildung des 2er-Komplements

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

function bin_int = bit_vector2integer(bit_vector)

  bin_int=0;
  bit_width=length(bit_vector);

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

end
```

Listing 8.8: Binär-Vektor in Binär-Integer umwandeln

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

function dec = s1q10_to_dec(bit_vector)

  % Dezimalzahl aus s1q10 Binaerzahl berechnen

  bit_width=length(bit_vector);
  dec = 0;

  if bit_vector(1) == 1
    dec = -2;
    if bit_vector(2) == 1
      dec = -1;
    end
```

```
  elseif bit_vector(2) == 1
    dec = 1;
  end

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

Listing 8.9: Kontroll-Skript für S1Q10 nach Dezimal

8.4 Programmcode

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;



package constants is
  constant mat_size : integer;
  constant bit_width_extern : integer;
  constant bit_width_adder : integer;
  constant bit_width_multiplier : integer;
end constants;

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

end constants;
```

Listing 8.10: Deklaration der Konstanten

```
-- Package, welches ein 2D-Array bereitstellt.
-- 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.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
library work;
use work.all;
use constants.all;
```

```
package datatypes is
    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;

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


    subtype t_twiddle_coeff_long is signed(16 downto 0);
    constant twiddle_coeff_long : t_twiddle_coeff_long := "00101101010000010";
    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));




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

end datatypes;
```

Listing 8.11: Deklaration eigener Datentypen

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

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

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


entity read_input_matrix is
  port(
```

```
        clk         : in   bit;
        loaded      : out  bit;
        input_real  : out  t_2d_array;
        input_imag  : out  t_2d_array
      );
end entity read_input_matrix;

architecture bhv of read_input_matrix is
begin
  reading :  process

    variable   element_1_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_1_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_2_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_2_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_3_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_3_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_4_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_4_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_5_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_5_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_6_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_6_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_7_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_7_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_8_real  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');
    variable   element_8_imag  : std_logic_vector(bit_width_extern-1
   downto 0) := (others => '0');

    variable   r_space   : character;

    variable  fstatus        : file_open_status;   -- status r,w
    variable  inline       : line;      -- readout line
    file   infile    : text;      -- filehandle for reading ascii text

```

```
    variable textfilename : string(1 to 29);

    begin



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

      file_open(fstatus,infile, textfilename,  read_mode);

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

      for i in 0 to mat_size-1 loop

        wait until clk = '1' and clk'event;
          readline(infile, inline);
          read(inline, element_1_real);
          read(inline, r_space);
          read(inline, element_1_imag);
          read(inline, r_space);
          read(inline, element_2_real);
          read(inline, r_space);
          read(inline, element_2_imag);
          read(inline, r_space);
          read(inline, element_3_real);
          read(inline, r_space);
          read(inline, element_3_imag);
          read(inline, r_space);
          read(inline, element_4_real);
          read(inline, r_space);
          read(inline, element_4_imag);
          read(inline, r_space);
          read(inline, element_5_real);
          read(inline, r_space);
          read(inline, element_5_imag);
          read(inline, r_space);
          read(inline, element_6_real);
          read(inline, r_space);
          read(inline, element_6_imag);
          read(inline, r_space);
          read(inline, element_7_real);
          read(inline, r_space);
```

```
            read(inline, element_7_imag);
            read(inline, r_space);
            read(inline, element_8_real);
            read(inline, r_space);
            read(inline, element_8_imag);

            input_real(i)(0) <= signed(element_1_real);
            input_imag(i)(0) <= signed(element_1_imag);
            input_real(i)(1) <= signed(element_2_real);
            input_imag(i)(1) <= signed(element_2_imag);
            input_real(i)(2) <= signed(element_3_real);
            input_imag(i)(2) <= signed(element_3_imag);
            input_real(i)(3) <= signed(element_4_real);
            input_imag(i)(3) <= signed(element_4_imag);
            input_real(i)(4) <= signed(element_5_real);
            input_imag(i)(4) <= signed(element_5_imag);
            input_real(i)(5) <= signed(element_6_real);
            input_imag(i)(5) <= signed(element_6_imag);
            input_real(i)(6) <= signed(element_7_real);
            input_imag(i)(6) <= signed(element_7_imag);
            input_real(i)(7) <= signed(element_8_real);
            input_imag(i)(7) <= signed(element_8_imag);

            if i = mat_size-1 then
              loaded <= '1' after 10 ns;
            end if;
        end loop;
        file_close(infile);
        wait;


  end process;
end bhv;
```

Listing 8.12: Eingangs-Matrix aus Textdatei einlesen

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library work;
use work.all;
use datatypes.all;

entity read_input_matrix_tb is
end entity read_input_matrix_tb;

architecture arch of read_input_matrix_tb is

  signal clk          : bit := '0';
  signal loaded       : bit := '0';
  signal input_real   : t_2d_array;
  signal input_imag   : t_2d_array;
```

```
  component read_input_matrix is
    port(
          clk          : in   bit;
          loaded       : out  bit;
          input_real   : out  t_2d_array;
          input_imag   : out  t_2d_array
        );
  end component;

  begin
    dut : read_input_matrix
      port map(
               clk          => clk,
               loaded       => loaded,
               input_real   => input_real,
               input_imag   => input_imag
             );

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

Listing 8.13: Testbench für das Einlesen aus einer Textdatei

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

library STD;  -- for writing text file
use STD.TEXTIO.ALL;
use ieee.std_logic_textio.all;

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



entity write_results is
  port(
        result_ready  : in   bit;
        result_real   : in   t_2d_array;
        result_imag   : in   t_2d_array;
        write_done    : out  bit
      );
end entity write_results;


architecture bhv of write_results is
begin
```

```
writing_to_file : process(result_ready)

  variable fstatus : file_open_status; -- status r,w
  variable outline : line; -- writeout line
  file     outfile : text; -- filehandle

  --variable output1 : bit_vector(3 downto 0) := "0101";
  --variable output2 : bit_vector(3 downto 0) := "0110";

  variable element_1_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_1_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_2_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_2_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_3_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_3_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_4_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_4_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_5_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_5_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_6_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_6_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_7_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_7_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_8_real : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable element_8_imag : std_logic_vector(bit_width_extern-1 downto 0)
 ;
  variable space : character := ' ';

  begin

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

    --if result_ready = '1' then

  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;
```

```

    write_done <= '1';
    file_close(outfile);
   --end if;

  end process;
end bhv;
```

Listing 8.14: Ergebnis-Matrix in Textdatei schreiben

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

library STD; -- for writing text file
use STD.TEXTIO.ALL;
use ieee.std_logic_textio.all;

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


entity write_test_tb is
end entity write_test_tb;


architecture bhv of write_test_tb is

  signal clk          : bit;
  signal loaded       : bit;
  signal result_ready : bit;
  signal write_done   : bit;
  signal loop_running : bit;
  signal loop_number  : signed(2 downto 0);
  signal input_real   : t_2d_array;
  signal input_imag   : t_2d_array;
  signal output       : std_logic_vector(bit_width_extern-1 downto 0);

  component read_input_matrix
    port(
          clk        : in  bit;
          loaded     : out bit;
          input_real : out t_2d_array;
          input_imag : out t_2d_array
        );
  end component;

  component write_results
    port(
          result_ready : in  bit;
```

```
        result_real   : in   t_2d_array;
        result_imag   : in   t_2d_array;
        write_done    : out  bit;
        loop_number   : out  signed(2 downto 0);
        loop_running  : out  bit;
        output        : out  std_logic_vector(bit_width_extern-1 downto 0)
      );
  end component;

begin

  mat : read_input_matrix
    port map(
             clk         => clk,
             loaded      => loaded,
             input_real  => input_real,
             input_imag  => input_imag
            );

  write : write_results
    port map(
             result_ready  => result_ready,
             result_real   => input_real,
             result_imag   => input_imag,
             write_done    => write_done,
             loop_number   => loop_number,
             loop_running  => loop_running,
             output        => output
            );

  result_ready <= loaded  after 20 ns;
  clk          <= not clk after 10 ns;

end bhv;
```

Listing 8.15: Testbensch für das schreiben in eine Textdatei

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
library work;
use work.all;
use datatypes.all;
use constants.all;


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

entity dft8optimiert is
port(
```

```
      clk            : in   bit;
      nReset         : in   bit;
      loaded         : in   bit;
      input_real     : in   t_2d_array;
      input_imag     : in   t_2d_array;
      result_real    : out  t_2d_array;
      result_imag    : out  t_2d_array;
      result_ready   : out  bit;
      idft           : in   bit;
      state_out      : out  t_dft8_states;
      element_out    : out  unsigned(5 downto 0);
      dft_1d_2d_out  : out  bit
    );
end dft8optimiert;


architecture arch of dft8optimiert is

  signal dft_state, next_dft_state : t_dft8_states;


begin

  FSM_TAKT: process(clk)
  begin
    if clk='1' and clk'event then
      dft_state <= dft_state;
      state_out <= dft_state;
      if nReset='0' then
        dft_state <= idle;
        state_out <= idle;
      elsif loaded = '0' then
        dft_state <= idle;
        state_out <= idle;
      elsif loaded='1' and dft_state = idle then
        dft_state <= twiddle_calc;
        state_out <= twiddle_calc;
      else
        dft_state <= next_dft_state;
        state_out <= next_dft_state;
      end if;
    end if;
  end process;


  FSM_KOMB: process(dft_state)
    --constant twiddle_coeff : signed(16 downto 0) := "00010110101000001";
    variable twiddle_coeff : signed(bit_width_adder-1 downto 0);

    variable mult_re, mult_im : signed(bit_width_multiplier-1 downto 0);

```

```
    variable W_row, I_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 n chsten Durchlauf.
    -- Der Elemente-Vektor kann darueber hinaus in vordere Haelfte = Zeile
    und hintere Haelfte = Spalte augeteilt 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
```

```
        W_row := 8-row_col_idx;  -- Twiddlefaktor-Matrix
      end if;
    else
      W_row := row_col_idx;  -- Twiddlefaktor-Matrix
    end if;

    I_col := to_integer(element(2 downto 0));  -- Input-Matrix


    if element = "000000" then
      if dft_1d_2d = '0' then
        matrix_real := input_real;
        matrix_imag := input_imag;
      else
        matrix_real := dft_1d_real;
        matrix_imag := dft_1d_imag;
      end if;
    end if;


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

      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.
        -- 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*2*pi*[0:7]'*[0:7]/8) berechnen.
          -- 1. Zeile aus W -> nur Additionen
          when 0 =>
            -- Die 1. Zeile aus W besteht nur aus den Faktoren (1+j0).
Daraus resultiert, dass die rellen
            -- und die imaginaeren Werte der Eingangs-Matrix unabhaengig
von einander aufsummiert werden.
            -- Real
            temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder)
+ resize(matrix_real(1)(I_col), bit_width_adder);
            temp_re(1) := resize(matrix_real(2)(I_col), bit_width_adder)
+ resize(matrix_real(3)(I_col), bit_width_adder);
            temp_re(2) := resize(matrix_real(4)(I_col), bit_width_adder)
+ resize(matrix_real(5)(I_col), bit_width_adder);
            temp_re(3) := resize(matrix_real(6)(I_col), bit_width_adder)
+ resize(matrix_real(7)(I_col), bit_width_adder);
            -- Imag
            temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder)
+ resize(matrix_imag(1)(I_col), bit_width_adder);
```

```
                temp_im(1) := resize(matrix_imag(2)(I_col), bit_width_adder)
    + resize(matrix_imag(3)(I_col), bit_width_adder);
                temp_im(2) := resize(matrix_imag(4)(I_col), bit_width_adder)
    + resize(matrix_imag(5)(I_col), bit_width_adder);
                temp_im(3) := resize(matrix_imag(6)(I_col), bit_width_adder)
    + resize(matrix_imag(7)(I_col), bit_width_adder);


            -- 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)

            -- Wegen der Faktoren (+/-0.70711 +/-0.70711i) haben die geraden
    Zeilen (beginnend bei 1) 12 statt 8 Subtraktionen
            -- 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
            -- insbesondere auch den Platzbedarf auf dem Chip klein zuhalten,
     wird die Multiplikation auf die Summe aller und
            -- nicht auf die einzelnen Werte angewandt.
            -- Da immer genau die Haelfte der Faktoren positiv und die andere
     negativ ist, werden die Eingangswerte so sortiert,
            -- dass keine Negationen noetig sind.
            when 1 =>
                -- Real
                temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder)
    - resize(matrix_real(4)(I_col), bit_width_adder);
                temp_re(1) := resize(matrix_imag(2)(I_col), bit_width_adder)
    - resize(matrix_imag(6)(I_col), bit_width_adder);
                -- MultPart
                temp_re(2) := resize(matrix_real(1)(I_col), bit_width_adder)
    - resize(matrix_real(3)(I_col), bit_width_adder);
                temp_re(3) := resize(matrix_imag(1)(I_col), bit_width_adder)
    - resize(matrix_imag(7)(I_col), bit_width_adder);
                temp_re(4) := resize(matrix_imag(3)(I_col), bit_width_adder)
    - resize(matrix_real(5)(I_col), bit_width_adder);
                temp_re(5) := resize(matrix_real(7)(I_col), bit_width_adder)
    - resize(matrix_imag(5)(I_col), bit_width_adder);
                -- Imag
                temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder)
    - resize(matrix_real(2)(I_col), bit_width_adder);
                temp_im(1) := resize(matrix_real(6)(I_col), bit_width_adder)
    - resize(matrix_imag(4)(I_col), bit_width_adder);
                -- MultPart
                temp_im(2) := resize(matrix_imag(1)(I_col), bit_width_adder)
    - resize(matrix_real(1)(I_col), bit_width_adder);
                temp_im(3) := resize(matrix_real(5)(I_col), bit_width_adder)
    - resize(matrix_real(3)(I_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);
                temp_im(5) := resize(matrix_imag(7)(l_col), bit_width_adder) - resize(matrix_imag(5)(l_col), bit_width_adder);

            -- 3. Zeile aus W
            -- 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)
            when 2 =>
                -- Real
                temp_re(0) := resize(matrix_real(0)(l_col), bit_width_adder) - resize(matrix_real(2)(l_col), bit_width_adder);
                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);
                temp_re(3) := resize(matrix_imag(5)(l_col), bit_width_adder) - resize(matrix_imag(7)(l_col), bit_width_adder);
                --Imag
                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);
                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);

            -- 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)
            when 3 =>
                -- Real
                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);
                --MultPart
                temp_re(2) := resize(matrix_imag(1)(l_col), bit_width_adder) - resize(matrix_real(1)(l_col), bit_width_adder);
                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);
                temp_re(5) := resize(matrix_real(5)(l_col), bit_width_adder) - resize(matrix_real(7)(l_col), bit_width_adder);
```

```
                -- Imag
                temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder) - resize(matrix_imag(4)(l_col), bit_width_adder);
                temp_im(1) := resize(matrix_real(2)(l_col), bit_width_adder) - resize(matrix_real(6)(l_col), bit_width_adder);
                --MultPart
                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);
                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);

            -- 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)
            when 4 =>
                -- Real
                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);
                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);
                -- Imag
                temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder) - resize(matrix_imag(1)(l_col), bit_width_adder);
                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);
                temp_im(3) := resize(matrix_imag(6)(l_col), bit_width_adder) - resize(matrix_imag(7)(l_col), bit_width_adder);

            -- 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)
            when 5 =>
                -- Real
                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);
```

```
                --MultPart
                temp_re(2) := resize(matrix_real(3)(I_col), bit_width_adder) - resize(matrix_real(1)(I_col), bit_width_adder);
                temp_re(3) := resize(matrix_real(5)(I_col), bit_width_adder) - resize(matrix_imag(1)(I_col), bit_width_adder);
                temp_re(4) := resize(matrix_imag(5)(I_col), bit_width_adder) - resize(matrix_imag(3)(I_col), bit_width_adder);
                temp_re(5) := resize(matrix_imag(7)(I_col), bit_width_adder) - resize(matrix_real(7)(I_col), bit_width_adder);
                -- Imag
                temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(matrix_real(2)(I_col), bit_width_adder);
                temp_im(1) := resize(matrix_real(6)(I_col), bit_width_adder) - resize(matrix_imag(4)(I_col), bit_width_adder);
                --MultPart
                temp_im(2) := resize(matrix_real(1)(I_col), bit_width_adder) - resize(matrix_imag(1)(I_col), bit_width_adder);
                temp_im(3) := resize(matrix_real(3)(I_col), bit_width_adder) - resize(matrix_real(5)(I_col), bit_width_adder);
                temp_im(4) := resize(matrix_imag(3)(I_col), bit_width_adder) - resize(matrix_real(7)(I_col), bit_width_adder);
                temp_im(5) := resize(matrix_imag(5)(I_col), bit_width_adder) - resize(matrix_imag(7)(I_col), bit_width_adder);

            -- 7. Zeile
            -- 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)
            when 6 =>
                -- Real
                temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder) - resize(matrix_imag(1)(I_col), bit_width_adder);
                temp_re(1) := resize(matrix_imag(3)(I_col), bit_width_adder) - resize(matrix_real(2)(I_col), bit_width_adder);
                temp_re(2) := resize(matrix_real(4)(I_col), bit_width_adder) - resize(matrix_imag(5)(I_col), bit_width_adder);
                temp_re(3) := resize(matrix_imag(7)(I_col), bit_width_adder) - resize(matrix_real(6)(I_col), bit_width_adder);
                -- Imag
                temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(matrix_imag(2)(I_col), bit_width_adder);
                temp_im(1) := resize(matrix_real(1)(I_col), bit_width_adder) - resize(matrix_real(3)(I_col), bit_width_adder);
                temp_im(2) := resize(matrix_imag(4)(I_col), bit_width_adder) - resize(matrix_imag(6)(I_col), bit_width_adder);
                temp_im(3) := resize(matrix_real(5)(I_col), bit_width_adder) - resize(matrix_real(7)(I_col), bit_width_adder);

            -- 8. 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)
        when 7 =>
            -- Real
            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);
            --MultPart
            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);
            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);
            -- Imag
            temp_im(0) := resize(matrix_imag(0)(l_col), bit_width_adder) - resize(matrix_imag(4)(l_col), bit_width_adder);
            temp_im(1) := resize(matrix_real(2)(l_col), bit_width_adder) - resize(matrix_real(6)(l_col), bit_width_adder);
            --MultPart
            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);
            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);

        when others => element := element; -- "dummy arbeit", es sind bereits alle Faelle abgedeckt!
      end case;

      next_dft_state <= additions_stage1;


    when additions_stage1 =>  -- dft_state_out = 2

      -- Es wird vor jeder Addition ein Bitshift auf die Summanden angewandt, um den Wertebereich der Speichervariable beim zurueckschreiben nicht zu ueberschreiten (1. Mal)

      -- Zeilen 1, 3, 5, 7 (ungerade) aufsummieren (bzw. 0(000XXX), 2(010XXX), 4(100XXX), 6(110XXX) beginnend bei 0)
      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);
            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
            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);
          else
            -- gerade Zeilen aus W
            -- Real
            --ConstPart
            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
            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);
            -- Imag
            --ConstPart
            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
            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);
          end if;

          next_dft_state <= additions_stage2;


      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)
```

```
    -- Zusaetzlich wird wird beim Zuweisen der ungeraden Zeilen an die 1D-DFT-Matrix zwei wweitere 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);
```

```
          -- naechster Zustand
          next_dft_state <= const_mult;
      end if;


    when const_mult =>  -- dft_state_out = 4

      -- Der Zielvektor der Multiplikation ist 26 Bit breit, die beiden Multiplikanten sind mit je 13 Bit wie gefordert halb so breit.

      -- Zeilen 2, 4, 6, 8 (vergleichbar mit oben)
      mult_re := temp_re(2) * twiddle_coeff; --(16 downto 16-(bit_width_adder-1));
      mult_im := temp_im(2) * twiddle_coeff; --(16 downto 16-(bit_width_adder-1));

      next_dft_state <= additions_stage3;


    when additions_stage3 =>  -- dft_state_out = 5

      -- Die vordersten 12 Bit des Multiplikationsergebnisses werden verwendet und um 1 Bit nach rechts geshiftet, damit der Wert halbiert wird und der Zielvektor spaeter keinen Ueberlauf hat.
      -- Um wieder die vollen 13 Bit zu erhalten, wird die resize-Funktion verwendet.
      -- Real

      temp14bit_re := resize(mult_re(bit_width_multiplier-4 downto bit_width_multiplier-4-bit_width_extern), bit_width_adder+1) + resize(temp_re(0)(bit_width_adder-1 downto 1), bit_width_adder+1);
      temp_re(0) := temp14bit_re(bit_width_adder downto 1);

      -- Imag
      temp14bit_im := resize(mult_im(bit_width_multiplier-4 downto bit_width_multiplier-4-bit_width_extern), bit_width_adder+1) + resize(temp_im(0)(bit_width_adder-1 downto 1), bit_width_adder+1);
      temp_im(0) := temp14bit_im(bit_width_adder downto 1);

      -- Indizes vertauschen -> Transponiert abspeichern
      if dft_1d_2d = '0' then
        dft_1d_real(l_col)(row_col_idx) := temp_re(0)(bit_width_adder-1 downto 1);
        dft_1d_imag(l_col)(row_col_idx) := temp_im(0)(bit_width_adder-1 downto 1);
      else
        result_real(l_col)(row_col_idx) <= temp_re(0)(bit_width_adder-1 downto 1);
        result_imag(l_col)(row_col_idx) <= temp_im(0)(bit_width_adder-1 downto 1);
```

```
        end if;

        next_dft_state <= twiddle_calc;
        if element = 63 then
          if dft_1d_2d = '1' then
            next_dft_state <= set_ready_bit;
          end if;
          dft_1d_2d := not dft_1d_2d;
          dft_1d_2d_out <= dft_1d_2d;
        end if;


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


      when set_ready_bit =>
        result_ready <= '1';
        next_dft_state <= twiddle_calc;


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

  end process;
end arch;
```

Listing 8.16: Berechnung der 2D-DFT

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.all;
use constants.all;
use datatypes.all;

entity dft8optimiert_top is
--     port(
--          result_real : out t_2d_array;
--          result_imag : out t_2d_array
--        );
end entity dft8optimiert_top;

architecture arch of dft8optimiert_top is

  signal nReset        : bit;
  signal clk           : bit;
  signal input_real    : t_2d_array;
  signal input_imag    : t_2d_array;
  signal result_real   : t_2d_array;
  signal result_imag   : t_2d_array;
```

```vhdl
signal loaded        : bit;
signal result_ready  : bit;
signal write_done    : bit;
signal idft          : bit := '0';

signal state_out     : t_dft8_states;
signal element_out   : unsigned(5 downto 0);
signal dft_1d_2d_out : bit;


component dft8optimiert
  port(
        clk           : in  bit;
        nReset        : in  bit;
        loaded        : in  bit;
        input_real    : in  t_2d_array;
        input_imag    : in  t_2d_array;
        result_real   : out t_2d_array;
        result_imag   : out t_2d_array;
        result_ready  : out bit;
        idft          : in  bit;
        state_out     : out t_dft8_states;
        element_out   : out unsigned(5 downto 0);
        dft_1d_2d_out : out bit
      );
end component;


component read_input_matrix
  port(
        clk        : in  bit;
        loaded     : out bit;
        input_real : out t_2d_array;
        input_imag : out t_2d_array
      );
end component;


component write_results
  port(
        result_ready : in  bit;
        result_real  : in  t_2d_array;
        result_imag  : in  t_2d_array;
        write_done   : out bit
      );
end component;


begin
  dft : dft8optimiert
    port map(
```

```
                    nReset          => nReset,
                    clk             => clk,
                    loaded          => loaded,
                    input_real      => input_real,
                    input_imag      => input_imag,
                    result_real     => result_real,
                    result_imag     => result_imag,
                    result_ready    => result_ready,
                    idft            => idft,
                    state_out       => state_out,
                    element_out     => element_out,
                    dft_1d_2d_out   => dft_1d_2d_out
                  );

        mat : read_input_matrix
          port map(
                    clk          => clk,
                    loaded       => loaded,
                    input_real   => input_real,
                    input_imag   => input_imag
                   );

      write : write_results
         port map(
                   result_ready => result_ready,
                   result_real  => result_real,
                   result_imag  => result_imag,
                   write_done   => write_done
                  );

      clk    <= not clk after 20 ns;
      nReset <= '1' after 40 ns;

end arch;
```

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

8.5 Testumgebung

```
#!/bin/bash

matlab_script="binMat2decMat.m"

./simulate.sh && matlab -nojvm -nodisplay -nosplash -r $matlab_script

stty echo
```

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

tlab

```bash
#!/bin/bash

# global settings

errormax=15
worklib=worklib
#testbench=top_level_tb
testbench=dft8optimiert_top
architecure=arch
simulation_time="1500ns"


# VHDL-files

constant_declarations="constants.vhdl"
datatype_declarations="datatypes.vhdl"

main_entity="dft8optimiert.vhdl"
top_level_entity="dft8_optimiert_top.vhdl"
#top_level_testbench=

embedded_entity_1="read_input_matrix.vhdl"
embedded_entity_2="write_results.vhdl"


constant_declarations=$directory$constant_declarations
datatype_declarations=$directory$datatype_declarations
function_declerations=$directory$function_declerations
main_entity=$directory$main_entity
top_level_entity=$directory$top_level_entity
#top_level_testbench=$directory$top_level_testbench

embedded_entity_1=$directory$embedded_entity_1
embedded_entity_2=$directory$embedded_entity_2


# libs und logfiles

cdslib="cds.lib"
elab_logfile="ncelab.log"
ncvhdl_logfile="nchvdl.log"
ncsim_logfile="ncsim.log"

cdslib=${base_dir}${work_dir}${cdslib}
elab_logfile=${dirctory}${elab_logfile}
ncvhdl_logfile=${directory}${ncvhdl_logfile}
ncsim_logfile=${directory}${ncsim_logfile}


##

```

```
ncvhdl \
-work $worklib \
-cdslib $cdslib \
-logfile $ncvhdl_logfile \
-errormax $errormax \
-update \
-v93 \
-linedebug \
$constant_declarations \
$datatype_declarations \
$embedded_entity_1 \
$embedded_entity_2 \
$main_entity \
$top_level_entity \
#$top_level_testbench
#-status \

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

ncsim \
-cdslib $cdslib \
-logfile $ncsim_logfile \
-errormax $errormax \
-exit \
${worklib}.${testbench}:${architecure} \
-input testRUN.tcl
#-status \


#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

#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
```

```
#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

#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

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

delimiterIn = ' ';

bit_width_extern = 13

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

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);


Input_dec_imag = nan(8);
Results_vhdl_dec_real = nan(8);
Results_vhdl_dec_imag = nan(8);
Result_octave_real_1d = nan(8);
Result_octave_imag_1d = nan(8);


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

N = 8;
for m = 1:N
  for n = 1:N
    a.bin=mat2str(Results_vhdl_bin_real(m,n),bit_width_extern);
    Results_vhdl_dec_real(m,n) = a.double;
    a.bin=mat2str(Results_vhdl_bin_imag(m,n),bit_width_extern);
    Results_vhdl_dec_imag(m,n) = a.double;

```

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


Input_dec=Input_dec_real+1i*Input_dec_imag;


TW=exp(-i*2*pi*[0:7]'*[0:7]/8);




%Result_octave_1d=TW*Input_dec;
%Result_octave_real_1d=real(Result_octave_1d.')/16
%Result_octave_imag_1d=imag(Result_octave_1d)

Result_octave=TW*Input_dec*TW.';
Result_octave=Result_octave./256;

Results_vhdl_dec_real
Result_octave_real=real(Result_octave)

Result_octave_imag=imag(Result_octave);
Results_vhdl_dec_imag;

diff_real=Result_octave_real-Results_vhdl_dec_real
diff_imag=Result_octave_imag-Results_vhdl_dec_imag;

quit
```

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