

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 im 1er-Komplement	2
2.1.2	Integer im 2er-Komplement	2
2.1.3	SQ-Format im 2er-Komplement	3
2.2	Auswirkungen der Bitbegrenzung	3
2.2.1	Maximale Auflösung	3
2.2.2	Rauschen	3
2.3	Komplexe Multiplikation	4
2.4	Matrixmultiplikation	4
2.5	Fourierreihenentwicklung	4
2.6	Fouriertransformation	6
2.7	Diskrete Fouriertransformation (DFT)	6
2.7.1	Summen- und Matrizenschreibweise der DFT	7
2.7.2	Rein reelle 2D-DFT	8
2.7.3	Berechnung der Diskreten Fouriertransformation mittels FFT	9
2.7.4	Inverse DFT	11
2.8	Diskrete Kosinus Transformation (DCT)	12
2.8.1	Verwendung der DCT	12
2.8.2	Berechnung der DCT	12
3	Analyse	13
3.1	Bewertung verschiedener DCT-Größen	13
3.2	Bewertung verschiedener DFT-Größen	13
3.3	Entscheidung DCT vs. DFT	16
3.4	Abschätzung des Rechenaufwands	17
3.4.1	Gegenüberstellung von reellen und komplexen Eingangswerten	17
3.4.2	Direkte Multiplikation zweier 8x8 Matrizen	18
3.4.3	Optimierte Multiplikation zweier 8x8 Matrizen	19
3.4.4	Gegenüberstellung von Butterfly und optimierter Matrixmultiplikation	19
3.5	Kompromiss aus benötigter Chipfläche und Genauigkeit des Ergebnisses	20
4	Entwurf	21
4.1	Interpretation binärer Zahlen	21
4.2	Entwicklungsstufen	21
4.2.1	Multiplikation	21

4.2.2	Addierer	21
4.2.3	Konstantenmultiplikation	21
4.2.4	1D-DFT mit Integer-Werten	22
4.2.5	2D-DFT mit Integer-Werten	22
4.2.6	2D-DFT mit Werten SQ-Format	22
4.2.7	Vertauschen der Twiddlefaktor-Matrix-Zeilen ergibt IDFT	22
4.3	Test der Matrizenmultiplikation	22
4.4	Implementierung des Konstantenmultiplizierers	22
4.4.1	Syntheseergebnis eines 13 Bit Konstantenmultiplizierers	22
4.4.2	Syntheseergebnis für die Bildung des Zweierkomplements eines 13 Bit Vektors	23
4.5	Entwickeln der 2D-DFT in VHDL	24
4.6	Direkte Weiterverarbeitung der Zwischenergebnisse	24
4.7	Berechnungsschema der geraden und ungeraden Zeilen	25
4.7.1	Erwartete Anzahl benötigter Takte	28
4.8	Automatengraf	28
4.9	UML-Diagramm	30
4.10	Projekt- und Programmstruktur	33
4.11	Bibliotheken und Hardwarebeschreibungssprache	33
5	Evaluation	34
5.1	Simulation	34
5.1.1	NC Sim - positive Zahlendarstellung	34
5.2	Anzahl benötigter Takte	34
5.3	Zeitabschätzung im Einsatz als ABS-Sensor	34
5.4	Testumgebung	37
5.4.1	Struktogramm des Testablaufs	37
5.4.2	Reale Eingangswerte	37
5.5	Chipdesign	37
5.5.1	Anzahl Standardzellen	37
5.5.2	Visualisierung der Netzliste	37
5.5.3	Floorplan, Padring	37
6	Schlussfolgerungen	38
6.1	Zusammenfassung	38
6.2	Bewertung und Fazit	38
6.3	Ausblick	38
7	Abkürzungsverzeichnis	39
	Abbildungsverzeichnis	40
	Tabellenverzeichnis	41
	Literatur	42
8	Anhang	43
8.1	Skript zur Bewertung von Twiddlefaktormatrizen	43
8.2	Gate-Report des 12 Bit Konstantenmultiplizierers	46

8.3 Twiddlefaktormatrix im S1Q10-Format	47
8.4 Programmcode	51
8.5 Testumgebung	71

1 Einleitung

1.1 Motivation

1.2 Stand der Technik

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

Kurze Beschreibung zu Standardzellen.

1.3 Ziel dieser Arbeit

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

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

2 Grundlagen

2.1 Binäre Zahlendarstellung von Festkommazahlen

2.1.1 Integer im 1er-Komplement

Bei der Interpretation des Bitvektors als Integer im Einerkomplement werden die Bits anhand ihrer Position im Bitvektor gewichtet, wobei das niederwertigste Bit (LSB, least significant bit) dem Wert für den Faktor 2^0 entspricht, das Bit links davon dem für 2^1 und so weiter. Die Summe aller Bits, ohne das höchstwertigste, multipliziert mit ihrer Wertigkeit (Potenz) ergibt den Betrag der Dezimalzahl. Das höchstwertigste Bit (MSB, most significant bit) gibt Auskunft darüber, ob es sich um eine negative oder positive Zahl handelt. 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 $-2^{MSB-1} + 1 \text{ LSB}$ bis $2^{MSB-1} - 1 \text{ LSB}$.

Diese Darstellung hat den Vorteil, dass sich das Ergebnis einer Multiplikation der Zahlen $a \cdot b$ und $-a \cdot b$ nur im vordersten Bit unterscheidet. Darüber hinaus lässt sich das Vorzeichen des Ergebnisses durch eine einfache XOR-Verknüpfung der beiden MSB der Multiplikanden ermitteln. Die eigentliche Multiplikation beschränkt sich auf die Bits MSB-1 bis LSB. 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. Darüberhinaus ist dieses Verfahren aufgrund der geringen Bedeutung in keiner VHDL-Bibliothek implementiert. (Verifizieren!)

2.1.2 Integer im 2er-Komplement

Bei der Interpretation als Zweierkomplement kann anhand des MSB ebenfalls erkannt werden, ob es sich um eine positive oder negative Zahl handelt. Dennoch wird es nicht als Vorzeichenbit gewertet. Viel mehr bedeutet ein gesetztes MSB -2^{MSB-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 $2^{MSB-1} - 1 \text{ LSB}$ 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 angewandt werden können. Unterstützt werden

sie z.B. von den Datentypen `unsigned` sowie `signed`, welche in der Bibliothek u.a. `ieee.numeric_std.all` definiert sind.

2.1.3 SQ-Format im 2er-Komplement

Im SQ-Format werden Zahlen als vorzeichenbehafteter Quotient (signed quotient) dargestellt. Die konkretere Schreibweise von beispielsweise S1Q10 bedeutet, dass zusätzlich zu einem Vorzeichenbit noch ein weiteres Bit vor dem Komma steht. Für den Quotient stehen 10 Bit zur Verfügung, was einer maximalen Auflösung von $1\text{ LSB} = 2^{-10} = \frac{1}{1024} = 9,765625 \cdot 10^{-4}$ entspricht. Der Wertebereich liegt in diesem Fall bei -2 bis $1,999023438$. Er wurde in der vorliegenden Arbeit so gewählt, da sich hiermit die Werte $\pm 3,3\text{ V}/2 = \pm 1,65\text{ V}$ darstellen lassen, was nach Abzug des Offsets den Eingangsspannungen des Analog Digital Converter (ADC) von 0 V bis $3,3\text{ V}$ entspricht und zum derzeitigen Stand des Projekts davon ausgegangen wird, dass der verwendete ADC Werte mit zwölf Bit Breite ausgibt. Es wird von einer Vorverarbeitung ausgegangen, die dies erledigt.

2.2 Auswirkungen der Bitbegrenzung

2.2.1 Maximale Auflösung

Um einen guten Kompromiss aus ausreichender Genauigkeit, Geschwindigkeit und Platzbedarf zu erzielen, wird von Eingangs- / Ausgangssignalen mit 12 Bit Breite zwischen den einzelnen Komponenten auf dem Chip ausgegangen.

Sicherlich ist eine hohe Genauigkeit erstrebenswert. Es gilt jedoch zu bedenken, dass mit höheren Bitbreiten auch der Platzbedarf jedes einzelnen Datensignals aufgrund der zusätzlich benötigten Leitungen sowie der Flip-Flops für die (Zwischen-) Speicherung, linear steigt. Bei Additionen und insbesondere Multiplikationen geht mit jedem zusätzlichen Bit ebenfalls ein linear steigender Zeitbedarf einher. Eine Bitbreite von größer 24 Bit (bei Eingangsspannungen kleiner 5 V) ist darüber hinaus bei ADC nicht sinnvoll, da durch thermisches Rauschen die ermittelten Werte beeinflusst werden und die Pegel des Rauschen in dieser Größenordnung liegen. Derzeit wird davon ausgegangen, dass der Chip in einer Strukturgröße von 350 nm gefertigt wird, sodass sich jeder zusätzliche Platzbedarf merklich auswirkt.

2.2.2 Rauschen

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 das Endergebnis

Von Prof. Vollmer:

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

L : Stufe (Additionstakte?)

Da dies kein Schwerpunkt der Arbeit ist, wird diese Problematik an dieser Stelle nur kurz angesprochen, um darauf aufmerksam zu machen.

Vielleicht lass ich das auch weg!

2.3 Komplexe Multiplikation

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

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

2.4 Matrixmultiplikation

Um nachfolgende Abschnitte besser erörtern zu können, soll zunächst die Matrixmultiplikation besprochen werden. Wie in Abbildung 2.1 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.

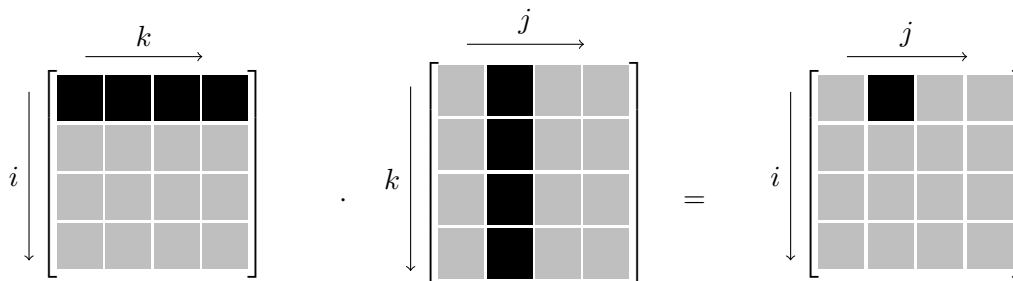


Abbildung 2.1: Veranschaulichung der Matrixmultiplikation

2.5 Fourierreihenentwicklung

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

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

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

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

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

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

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

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

und zusammengefasst ergibt sich in (Gl. 2.8) der komplexe Zeiger, der eine Rotation im Gegenuhrzeigersinn auf dem Einheitskreis beschreibt. In Abbildung 2.2 dies zusätzlich noch grafisch dargestellt.

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

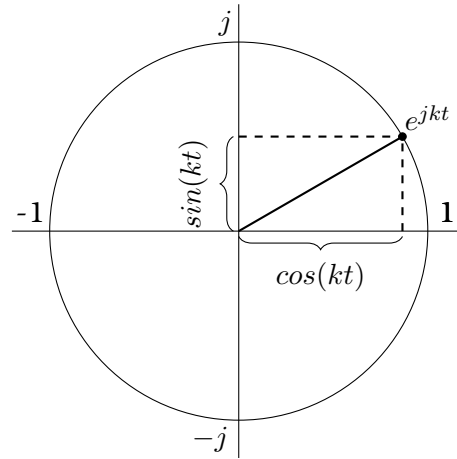


Abbildung 2.2: Einheitskreis, Zusammensetzung des komplexen Zeigers aus Sinus und Kosinus

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

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

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

2.6 Fouriertransformation

Mit der Fouriertransformation kann ein periodisches, abschnittsweise stetiges Signal $f(x)$ 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 $f(x)$.

Grundlage für die Fouriertransformation ist das Fourierintegral (Gl. 2.11)

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

Wenn Sinus und Kosinus wie in Gl. (2.6) und (2.7) als Exponentialfunktion geschrieben werden, können sie zu einer komplexen Exponentialfunktion zusammengefasst werden.

Für komplexere Signale, etwa ein Rechteck, ergeben sich 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.

- unendliche Dauer -> Leistungssignal?

- endliche Dauer -> Energiesignal?

Energiesignal:

Leistungssignal: Signal unendlicher Energie, aber mit endlicher mittlerer Leistung

Ein Zeitsignal hat ein eindeutig zuordbares Frequenzsignal (bijektiv), abgesehen von Amplitude? und Phase

Spektrum: Frequenzbestandteile eines Signals

Berechnung des Spektrums: Spektralanalyse, Frequenzanalyse

In der Praxis, also basierend auf echten Messdaten, wird die Bestimmung des Spektrums Spektrumschätzung genannt.

In der vorliegenden Arbeit wird künftig X^* für die 1D-DFT und X für die 2D-DFT stehen.

2.7 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 zeitlich versetzt auf den gleichen Bezugspunkt erfasst worden sein. Bezogen auf das Sensorarray würde eine bestimmte Anzahl an zeitlich versetzten zeit- und wertdiskretisierten Daten eines einzelnen Sensors in einem Vektor zusammengefasst und darauf die DFT angewandt werden, um beim Ausgangsvektor von Daten im Frequenzbereich sprechen zu können.

Statt zeitlich versetzter Daten werden beim Sensorarray die Daten von mehreren Sensoren gleichzeitig erfasst. Da das Sensorarray zweidimensional ist, ergibt sich an Stelle eines Vektors 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 Bildbereich.

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

2.7.1 Summen- und Matrizenschreibweise der DFT

1D-DFT

Die Eindimensionale Diskrete Fouriertransformation (1D-DFT) findet wie bereits erwähnt üblicherweise Anwendung, um vom Zeit- in den Frequenzbereich zu gelangen.

$$X^*[m] = \frac{1}{N} \cdot \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi mn}{N}} \quad (2.12)$$

Gleichung 2.14 zeigt die obige Summenformel 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.

$$\sum_{m=0}^{N-1} \sum_{n=0}^{N-1} e^{-j\frac{2\pi mn}{N}} = W \quad (2.13)$$

Somit gilt:

$$X^* = W \cdot x \quad (2.14)$$

In Matlab kann die Twiddlefaktormatrix mit

$$W = e^{-j\frac{2\pi}{N} \cdot [0:N-1]' \cdot [0:N-1]} \quad (2.15)$$

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 gelangen. Da es sich somit nicht mehr um eine Abhängigkeit der

Zeit handelt, werden andere Indizes verwendet.

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

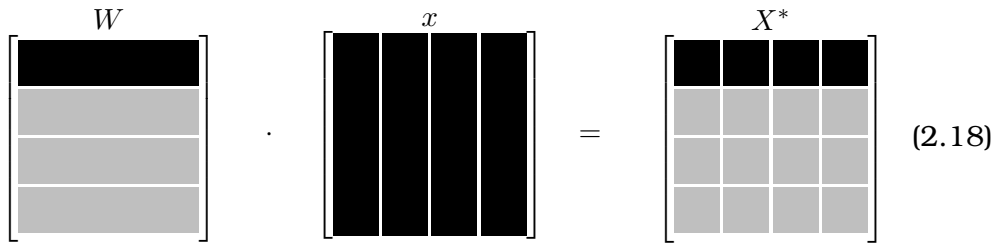
Auch hier lässt sich die Berechnung in Matrizenschreibweise darstellen:

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

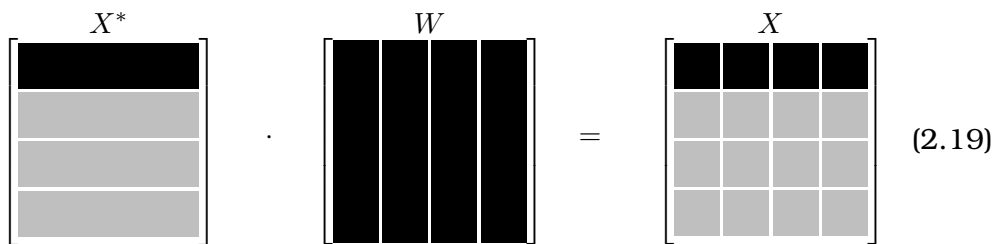
Die Gleichungen (2.14) und (2.17) werden wesentlicher Bestandteil der Umsetzung der 2D-DFT sein.

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.



$$W \cdot x = X^* \tag{2.18}$$



$$X^* \cdot W = X \tag{2.19}$$

2.7.2 Rein reelle 2D-DFT

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

Es wäre jedoch auch möglich, das komplexe Ausgangssignal der 1D-DFT als zwei von einander unabhängige rein reelle Eingangssignale der 2D-DFTs zu betrachten und

später wieder 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 (2.4) 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 reellem Multiplikand und komplexen Multiplikator 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 2.3 zu entnehmen.

In Abbildung 2.3 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 bereits erwähnt bei der reellen Berechnung zusätzlicher Speicher erforderlich wäre, wird dieses Verfahren angewandt.

2.7.3 Berechnung der Diskreten Fouriertransformation mittels FFT

Die Mathematiker Cooley und Tukey haben einen Algorithmus entwickelt, mit dem sich die DFT mit vergleichsweise wenig Multiplikationen und somit deutlich schneller als bei der allgemeinen DFT berechnen lässt. 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 (2.5) 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 ent-

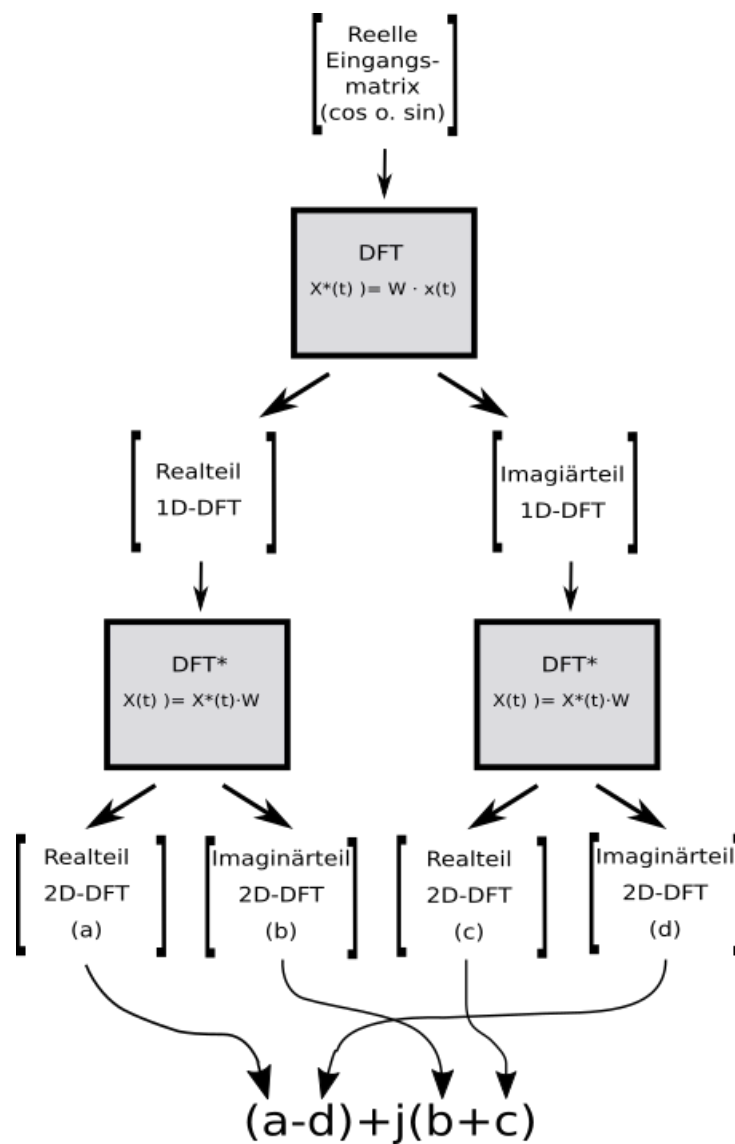


Abbildung 2.3: Veranschaulichung der reellen DFT

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

Abbildung 2.4: Redundante Werte der 8x8 DFT; Imaginärteil muss negiert werden, grau hinterlegt sind Multiplikationen der Twiddlefaktormatrix

hält. 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 Symmetriegründen halbiert sich die Anzahl. Wiederum die Hälfte davon sind komplexe Faktoren, die übrigen erfordern keine Multiplikation. Dies bedeutet, dass zwei komplexe Multiplikationen durchgeführt werden müssen, was 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):

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

Insbesondere bei größeren FFTs ist die relative Abweichung bedeutend geringer.

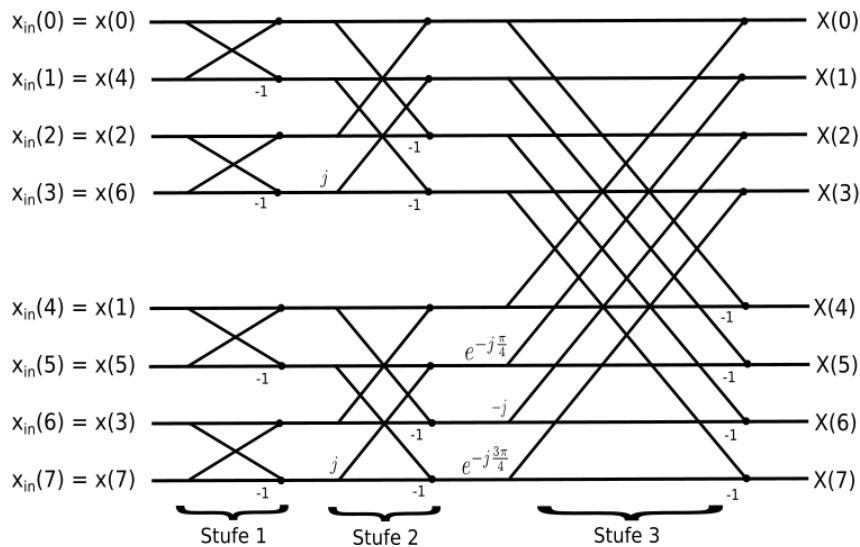


Abbildung 2.5: 8x8 Butterfly

2.7.4 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 Frequenzdarstellung das Zeitsignal zu errechnen.

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

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).

2.8 Diskrete Kosinus Transformation (DCT)

2.8.1 Verwendung der DCT

2.8.2 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 $\sqrt{\frac{2}{N}}$, $N = \text{Anzahl Elemente}$ in einer Zeile bzw. Spalte, multipliziert.

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

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

Die Twiddlefaktormatrix kann in Matlab mit

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

berechnet werden.

3 Analyse

3.1 Bewertung verschiedener DCT-Größen

Tabelle 3.1: Bewertung der DCT-Twiddlefaktor-Matrizen

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

3.2 Bewertung verschiedener DFT-Größen

In diesem Abschnitt werden verschiedene Größen von Twiddlefaktor-Matrizen auf ihre Werte untersucht und bewertet. Ziel ist es aus den in Frage kommenden jene zu ermitteln, die die trivialsten Berechnungen bei einer Multiplikation erfordert. Von Interesse sind aufgrund des dualen Zahlensystems Matrizen mit Werten, die sich einerseits mit wenigen Bits darstellen lassen und andererseits nur Bit-shifting zur Folge haben. Beide Anforderungen bedingen sich in der Regel gegenseitig.

In der folgenden Tabelle 3.2 werden die 8×8, 9×9, 12×12, 15×15 sowie 16×16-Matrix einander gegenüber gestellt. Da die Sensormatrix aus 8×8 Sensoren aufgebaut ist, besteht ein Interesse an einer ungeraden Matrix. Dies hätte den Vorteil, dass sich über dem Mittelpunkt der Sensormatrix kein Element der Twiddlefaktormatrix befindet. Auf diese Weise ließe sich die ... einfacher ermitteln. Bekannt ist jedoch auch, dass die Fast Fouriertransformation (FFT) auf Matrizen mit den Abmessungen 2^n basiert und es sich hierbei um ein sehr schnelles und effizientes Verfahren handelt. Deshalb werden auch Matrizen mit gerader Anzahl an Elementen untersucht. Die Beurteilung basiert auf dem Octave-Skript 8.2

Als triviale Werte werden 0, $\pm 0,5$ sowie ± 1 aufgefasst. Andere Werte die sich gut binär darstellen lassen tauchen nicht auf. Alle übrigen Werte werden als nicht trivial betrachtet, da eine Multiplikation mit ihnen eine komplexere 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 geachtelt wird und für

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

beispielsweise $\frac{2\pi}{8} = \frac{\pi}{4}$ Sinus und Cosinus identisch sind. Darüberhinaus ist dies auch der einzige Wert, der sowohl einen Real- als 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 vordersten Bit (Most Significant Bit (MSB)) 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 komplexere Subtraktion (bzw. Addition negativer Zahlen) gegenüber. Der zusätzliche Aufwand entspricht relativ genau dem der Bildung des 2er-Komplements. Aus diesem Grund wurde sich für dieses entschieden, da es deutlich gängiger 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

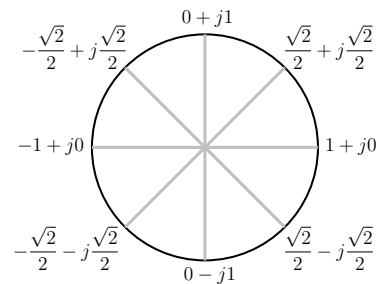


Abbildung 3.1: Einheitskreis mit relevanten Werten der 8x8-DFT

Twiddlefaktoren dargestellt. Sie sind aufgeteilt auf 8 Einheitskreise, wobei jeder einen Laufindex (m) des Zeitbereichs abdeckt. In den einzelnen Kreisen sind wiederum alle Laufinduxe (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- / Zwischenwerte erfolgen muss. Darüber hinaus minimiert sich bei geschickter Anordnung das Risiko eines Überlaufs. Da zur Sicherheit dennoch nach jeder Addition / 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.

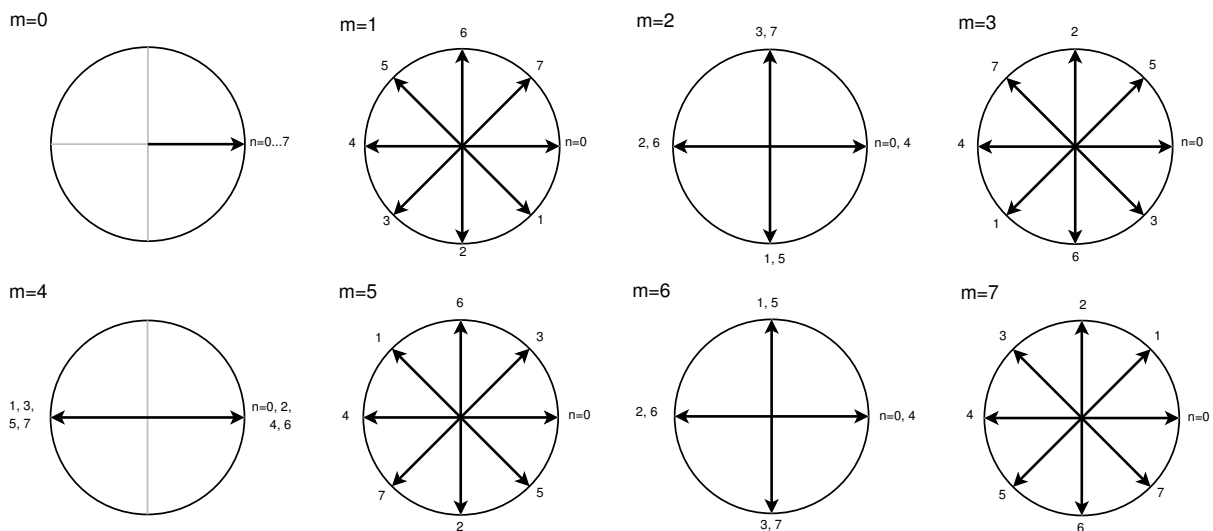


Abbildung 3.2: Twiddlefaktoren der 8x8-Matrix, aufgeteilt auf die Laufinduxe

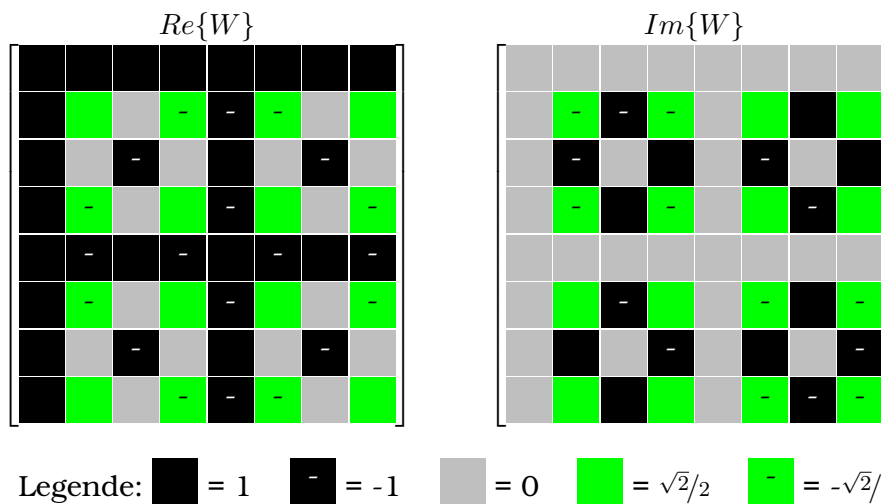


Abbildung 3.3: Matrix-Darstellung der 8x8-DFT-Twiddlefaktoren aufgeteilt nach Real- und Imaginärteil

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

Sowohl die Diskrete Cosinus Transformation (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 x nicht trivial darstellbare Zahlen der DCT einem einzigen bei der DFT gegenüber stehen.

Auch wenn bei der DFT mit der Berechnung des imaginären Teils zusätzlicher Implementierungsaufwand 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 eine erste Aufwandsabschätzung einer optimierten Matrizenmultiplikation geht, welche als Ausgangspunkt für eine finale Implementation dient, und es sich hier um keine endgültige Entscheidung handelt, kann mit Wahl der DFT kein grundlegender Fehler gemacht werden.

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 Aussummieren 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, wodurch 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 $(3+5) \cdot 4 \cdot 8 = 256$ Takte sowohl für den Real- als auch den Imaginärteil der komplexen Ausgangsmatrix. Real- und Imaginärteil werden parallel berechnet und sind somit zeitgleich fertig.

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.

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

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

Aus Abschnitt (2.7.2) 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 $(3+4+2+4+3) \cdot 8 = 128$ Takte. Der Imaginärteil errechnet sich noch schneller, da die 1. und 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 $(3+2+3) \cdot 8 = 64$ Takte. Vermutlich 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 (2.3) 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 (2.4) erläuterte Matrixmultiplikation bedarf bei einer 8x8 Matrix je Ergebnis der Ausgangsmatrix 8 Multiplikationen. Für die $8 \cdot 8 = 64$ 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 (2.3) beschrieben, insgesamt $512 \cdot 4 = 2048$ 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 (2.7.2) hergeleitet, knapp die Hälfte der Multiplikationen unnötig. In Abbil-

dung (2.4) 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 $40 \cdot 8 \cdot 2 = 640$ 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 (2.3) gezeigt, der Real- sowie der Imaginärteil separat mit der Twiddlefaktormatrix multipliziert werden. So ergeben sich alles in allem $640 \cdot 3 \cdot 2 = 3840$ reelle Multiplikationen. Diese Zahl liegt 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 (2.7.2) angedeutete deutlich erhöhte Bedarf an Takten die reelle Matrixmultiplikation nicht von Interesse ist.

3.4.3 Optimierte Multiplikation zweier 8x8 Matrizen

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 einzeln 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 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 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 (2.5) 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-DFFT 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 verdoppelt sich der Wert, also 30 und eine letzte Addition macht 31. Beim zweiten Durchlauf werden es so $(31+3) \cdot 2 + 1 = 69$ Bit.

⇒ 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 approxmierten 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.

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

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 Vertauschen der Twiddlefaktor-Matrix-Zeilen ergibt IDFT

4.3 Test der Matrizenmultiplikation

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

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

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

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

4.4 Implementierung des Konstantenmultiplizierers

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 explizierten 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 Syntheseresultat eines 13 Bit Konstantenmultiplizierers

Tabelle 4.1: Vergleich Konstanten- mit regulärem Multiplizierer

	Konstantenmultiplizierer	regulärer Multiplizierer
Gatter	27	175
Fläche (Prozess: 350nm)	$6612 \mu\text{m}^2$	23 261

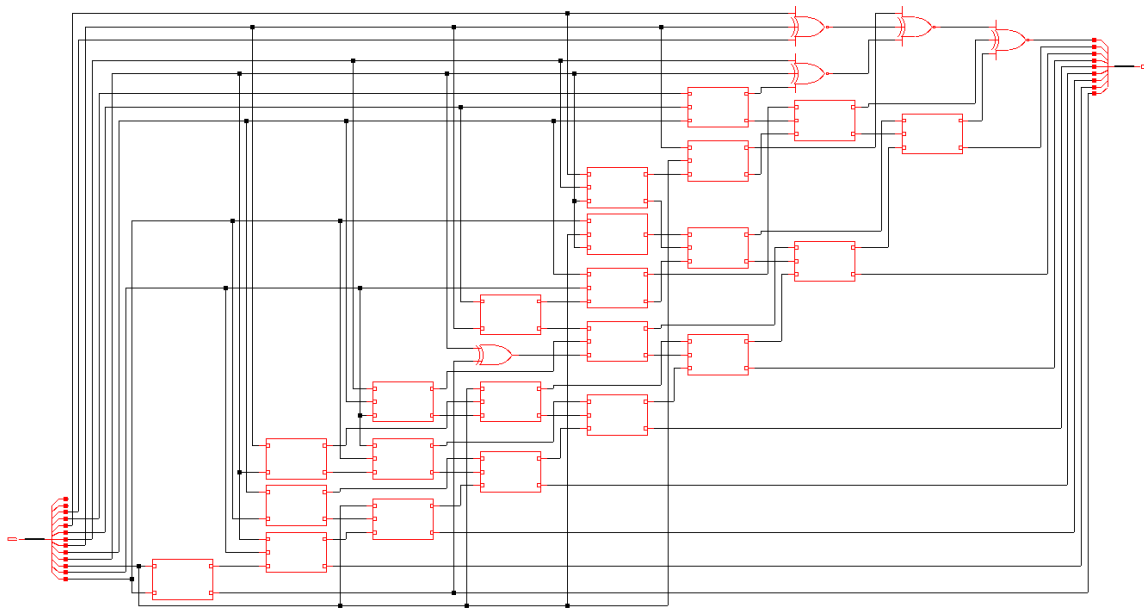


Abbildung 4.1: 13 Bit Konstantenmultiplizierer für $\frac{\sqrt{2}}{2} = 0.70711 \simeq 0.70703125 = 0001011010100_2$ in Encounter; Eingang links, Ausgang rechts

Der vollständige Gate-Report befindet sich in Abschnitt 8.3 auf Seite 46

4.4.2 Syntheseresultat 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.

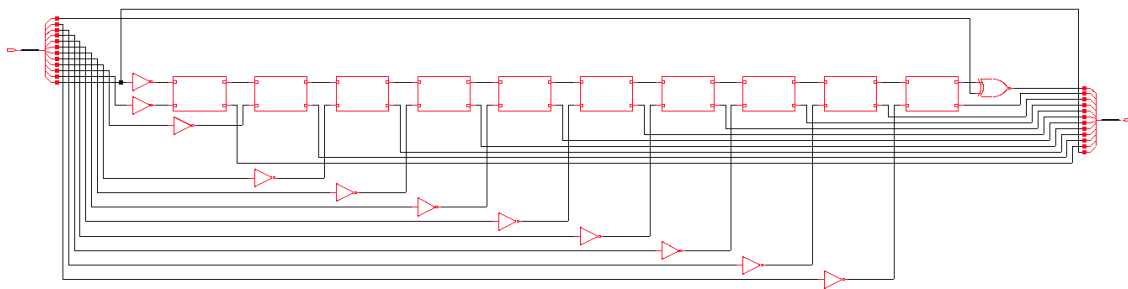


Abbildung 4.2: Netzliste einer Einheit zur Bildung des 2er-Komplements eines 13 Bit Vektors; Eingang links, Ausgang rechts

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.

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:

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

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

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 Matricelement 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.)

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 $64 \cdot 2 \cdot 12 \text{ Bit} = 1536 \text{ Bit} = 1,5 \text{ kBit} = 192 \text{ Byte}$ an Speicher eingespart werden. Wie sich im Laufe der Entwicklung gezeigt hat, lässt sich das nicht nutzen. Das liegt daran, dass dazu übergegangen wurde, immer nur ein Element zur Zeit berechnet wird und die bereits errechneten demnach zwischengespeichert werden müssen. Dieser Ansatz wurde verfolgt, da der Entwicklungsaufwand in VHDL für die spaltenweise Berechnung

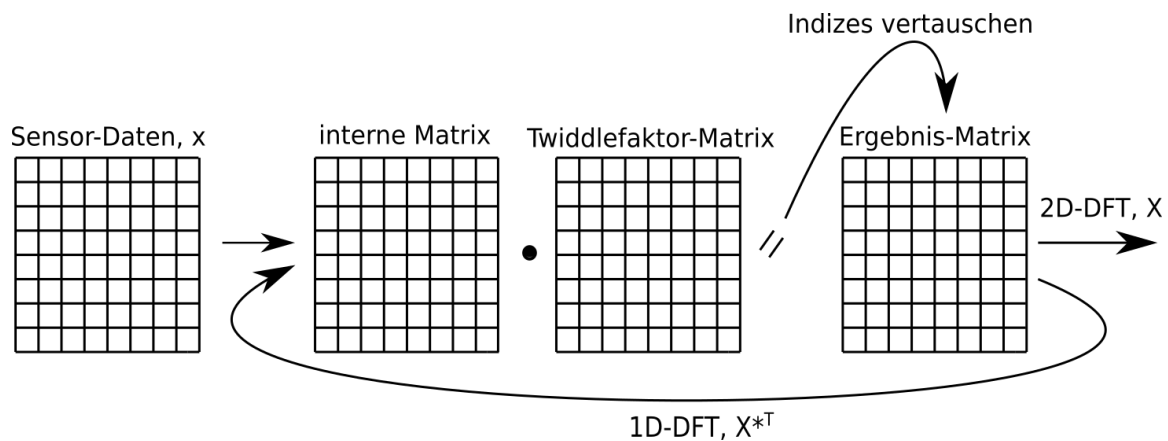


Abbildung 4.3: Darstellung der Berechnung der 2D-DFT aus Gleichung (4.3)

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.

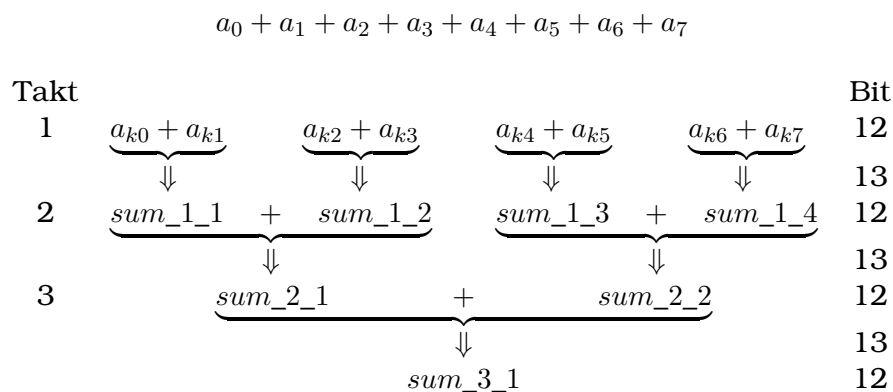


Abbildung 4.4: Vorgehensweise der Akkumulation der ungeraden Spalten der Eingangswerte

Wie der linken Spalte zu entnehmen ist, werden 3 Takte für die Berechnungen

der Werte aus den ungeraden Spalten der Eingangsmatrix bzw. ungeraden Zeilen der 1D-DFT-Matrix benötigt. 1. Takt für Additionen bzw. Subtraktionen und 2. sowie 3. Takt für das Aufsummieren. Der Bitvektor des Ergebnisses ist zwar 12 Bit breit, aber beim letzten Bitshift von 13 auf 12 werden nur 11 Bit übernommen. Es wird also ein doppelter Bitshift vollzogen. Dies erfolgt, damit sowohl in den geraden als auch 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

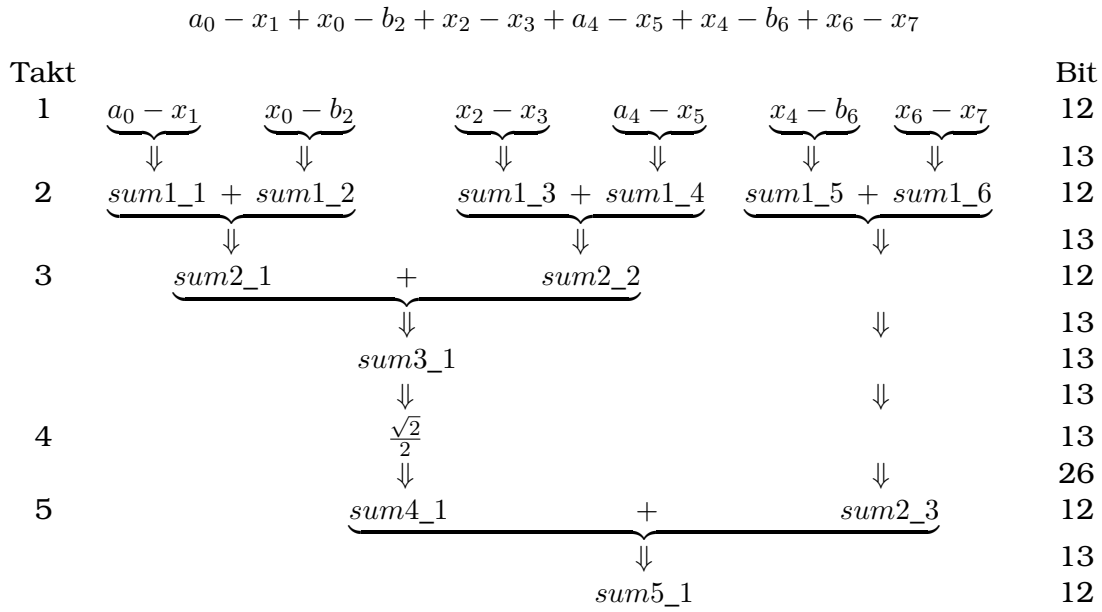


Abbildung 4.5: Vorgehensweise der Akkumulation der geraden Spalten der Eingangswerte

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 Automatengraf

test test

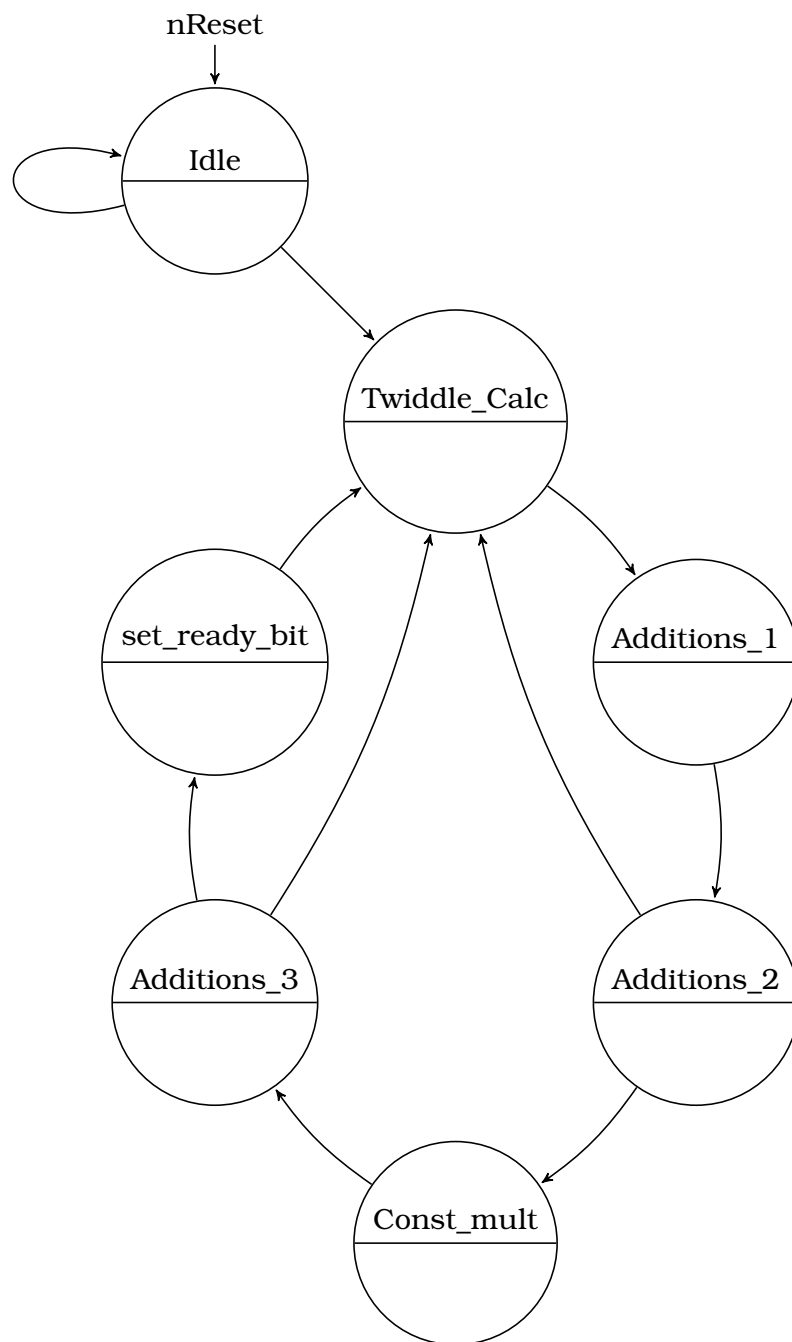
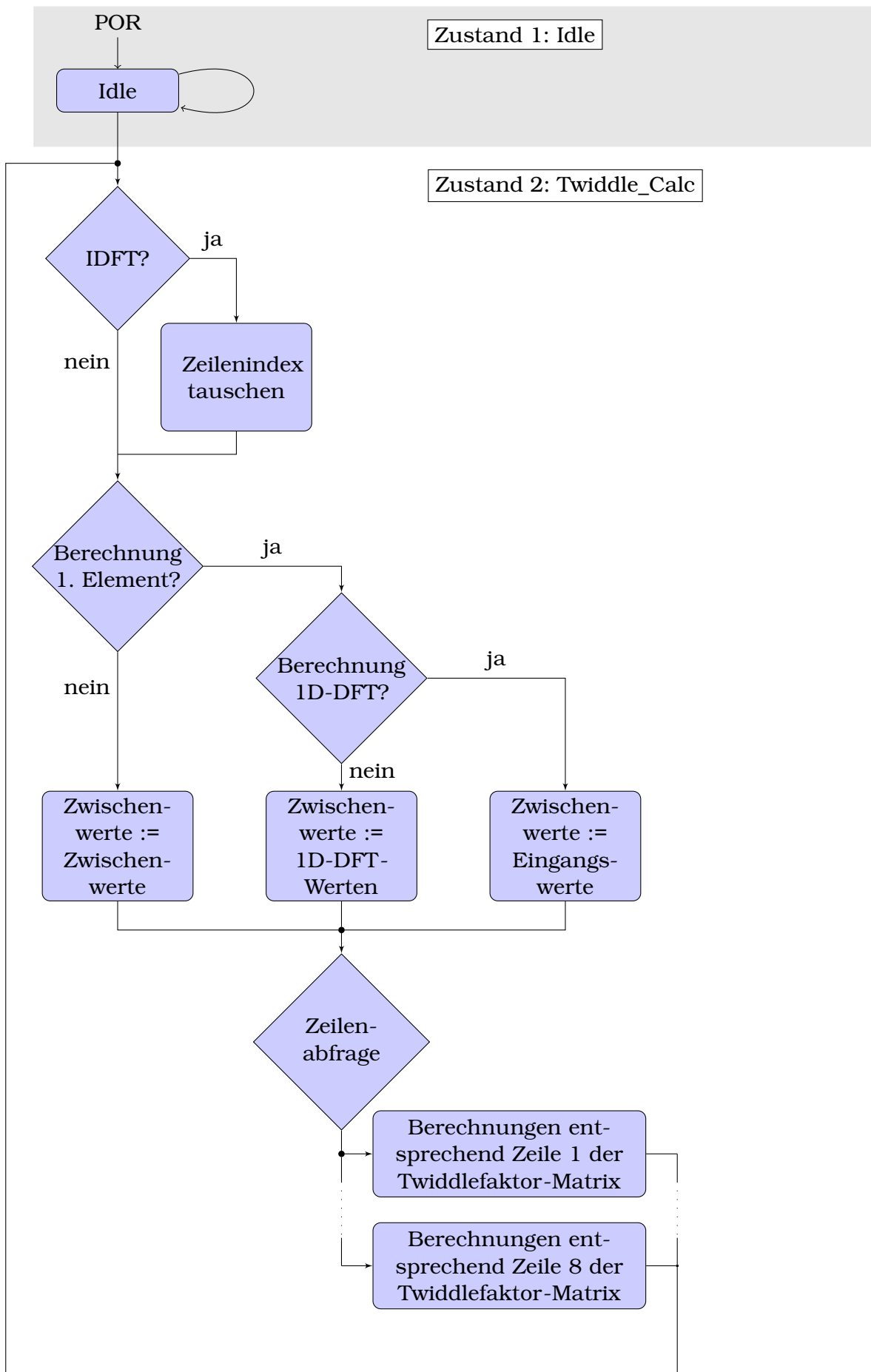
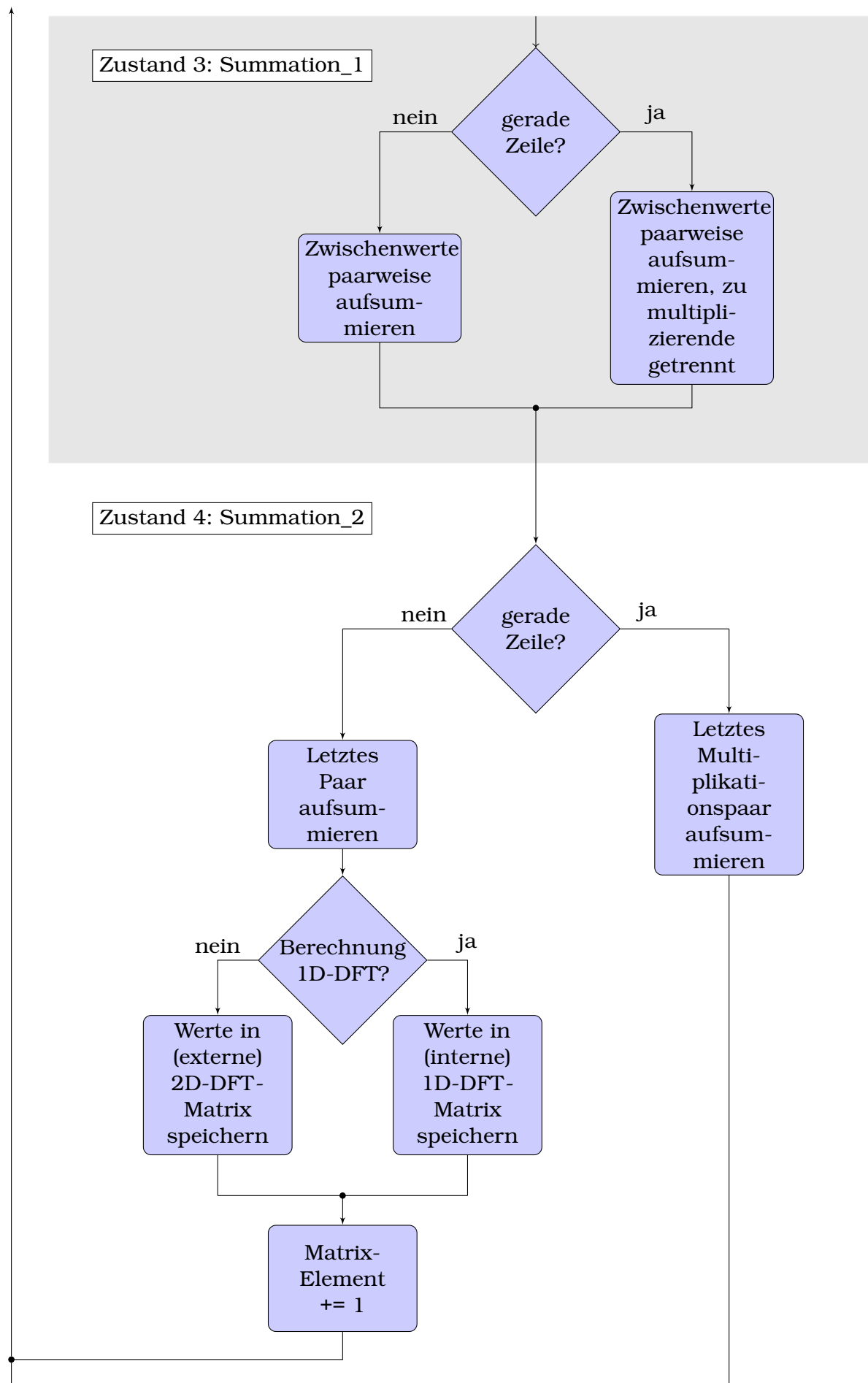
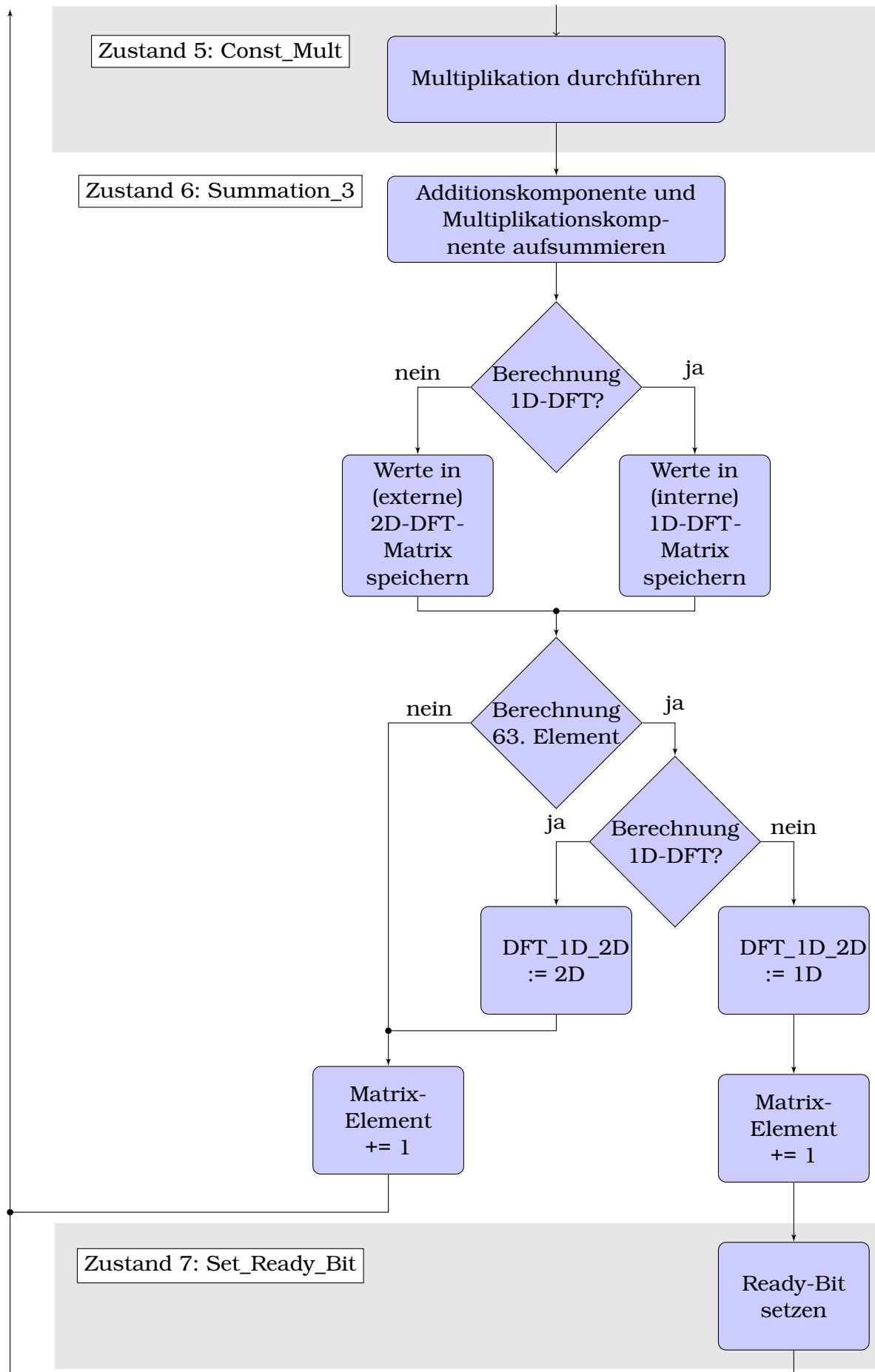


Abbildung 4.6: Automatengraf

4.9 UML-Diagramm







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

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

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

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

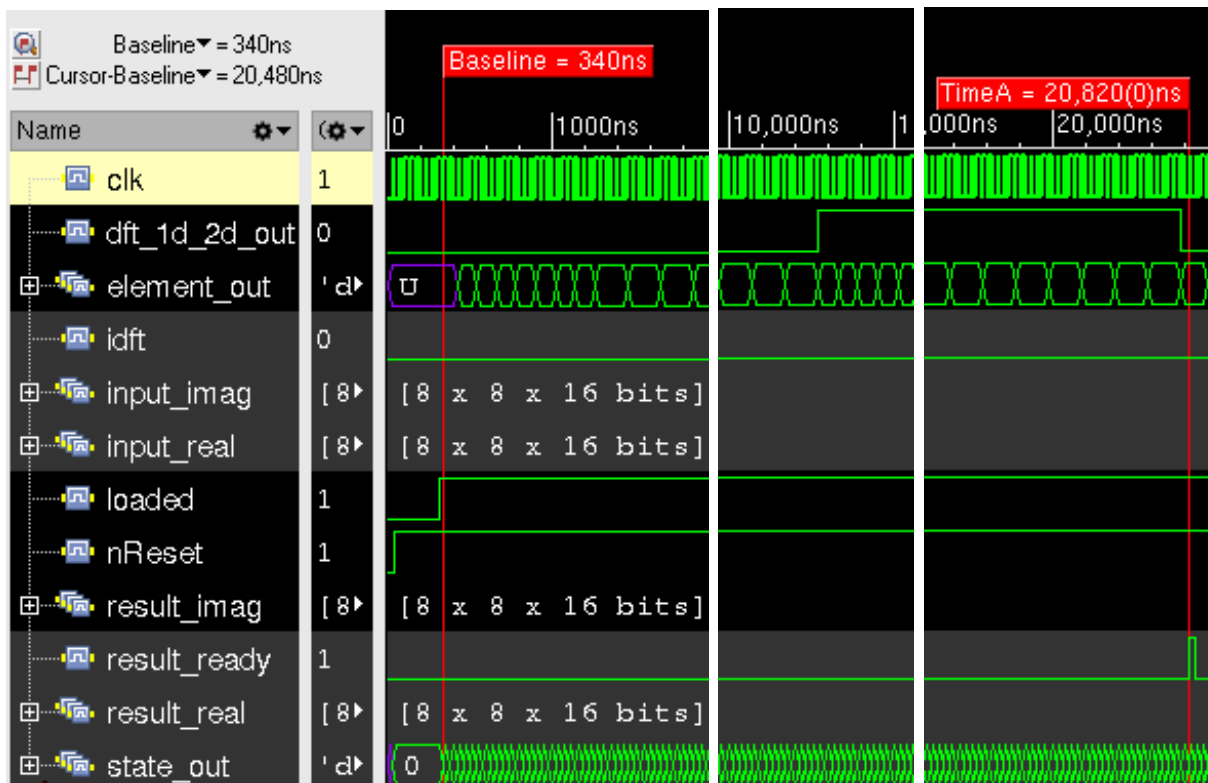


Abbildung 5.1: Simulations der 2D-DFT mit NC Launch

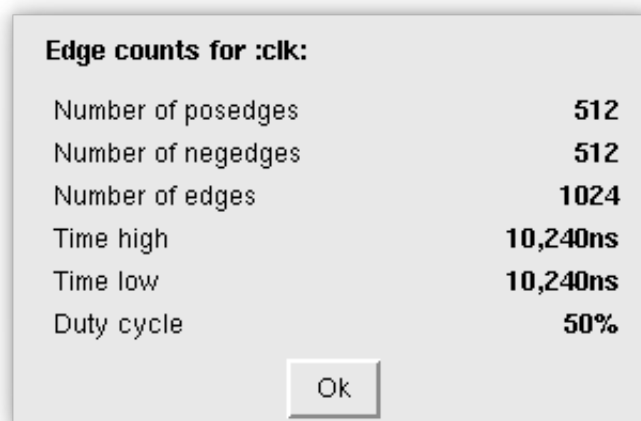


Abbildung 5.2: Edge Count für eine 2D-DFT

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

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

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

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

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

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.

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

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

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

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, Pading

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, <i>dt.: Anwendungsspezifischer Integrierter Schaltkreis</i>
DFT	Diskrete Fouriertransformation
FFT	Fast Fouriertransformation
FT	Fouriertransformation
IDFT	Inverse Diskrete Fouriertransformation
ISAR	Integrated Sensor Array
LSB	Least Significant Bit
MSB	Most Significant Bit
TMR	tunnelmagnetoresistiver Effekt

Abbildungsverzeichnis

2.1	Veranschaulichung der Matrixmultiplikation	4
2.2	Einheitskreis, Zusammensetzung des komplexen Zeigers aus Sinus und Kosinus	5
2.3	Veranschaulichung der reellen DFT	10
2.4	Redundante Werte der 8x8 DFT; Imaginärteil muss negiert werden, grau hinterlegt sind Multiplikationen der Twiddlefaktormatrix	10
2.5	8x8 Butterfly	11
3.1	Einheitskreis mit relevanten Werten der 8x8-DFT	15
3.2	Twiddlefaktoren der 8x8-Matrix, aufgeteilt auf die Laufindexe	15
3.3	Matrix-Darstellung der 8x8-DFT-Twiddlefaktoren aufgeteilt nach Real- und Imaginärteil	16
4.1	13 Bit Konstantenmultiplizierer für $\frac{\sqrt{2}}{2} = 0.70711 \simeq 0.70703125 = 0001011010100_2$ in Encounter; Eingang links, Ausgang rechts	23
4.2	Netzliste einer Einheit zur Bildung des 2er-Komplements eines 13 Bit Vektors; Eingang links, Ausgang rechts	23
4.3	Darstellung der Berechnung der 2D-DFT aus Gleichung (4.3)	25
4.4	Vorgehensweise der Akkumulation der ungeraden Spalten der Eingangswerte	25
4.5	Vorgehensweise der Akkumulation der geraden Spalten der Eingangswerte	26
4.6	Automatengraf	29
5.1	Simulations der 2D-DFT mit NC Launch	35
5.2	Edge Count für eine 2D-DFT	35

Tabellenverzeichnis

3.1	Bewertung der DCT-Twiddlefaktor-Matrizen	13
3.2	Bewertung der DFT-Twiddlefaktor-Matrizen	14
3.3	Gegenüberstellung der Vor- und Nachteile von DCT und DFT	17
3.4	Takte für die komplexe DFT	17
3.5	Takte für die reelle DFT am Beispiel der reellen Ausgangsmatrix	18
4.1	Vergleich Konstanten- mit regulärem Multiplizierer	22

Literatur

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

8 Anhang

8.1 Skript zur Bewertung von Twiddlefaktormatrizen

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

13 function dct_bewertung(N)

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

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

21
23 % Anzahl verschiedener Werte ermitteln
24 different_nums = unique(W);
25 different_non_trivial_nums = different_nums(find(different_nums ~= 1));
26 different_non_trivial_nums = different_non_trivial_nums(find(
27     different_non_trivial_nums ~= -1));
28 different_non_trivial_nums = different_non_trivial_nums(find(
29     different_non_trivial_nums ~= 0.5));
30 different_non_trivial_nums = different_non_trivial_nums(find(
31     different_non_trivial_nums ~= -0.5));
32 different_non_trivial_nums = different_non_trivial_nums(find(
33     different_non_trivial_nums ~= 0));

34 different_non_trivial_nums = unique(abs(different_non_trivial_nums));
35 different_non_trivial_nums
36 %non_trivial = length(abs(different_non_trivial_nums))

37
38 % Jeweils die Menge der verschiedenen Werte ermitteln
39 num_count = zeros(1, length(different_nums));
40 for k = 1:length(different_nums)
41     for n = 1:N
42         for m = 1:N
43             if different_nums(k) == W(m,n)
44                 num_count(k) = num_count(k) + 1;
45             end
46         end
47     end
48 end
```

```

45     end
46 end
47
48 % nicht triviale Werte der Matrix z hlen
49 nontrivial_nums = 0;
50 for k = 1:length(different_nums)
51     if abs(different_nums(k)) != 1
52         if abs(different_nums(k)) != 0.5
53             if different_nums(k) != 0
54                 nontrivial_nums = nontrivial_nums + num_count(k);
55             end
56         end
57     end
58 end
59 end
60
61 nums_of_matrix = N*N;
62
63 trivial_nums = N*N - nontrivial_nums
64
65 nontrivial_nums
66
67 v = trivial_nums/nontrivial_nums
68
69 end

```

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

```

1 %% Dateiname: dft_bewertung.m
2 %% Funktion: Bewertet die Koeffizienten der DFT-Twiddlefaktormatrix
3 %%           darauf basierend, wie trivial die Berechnungen mit
4 %%           den Twiddlefaktoren sind.
5 %%           Als trivial gelten Berechnungen mit den Werten -1, -0.5, 0, +0.5, +1
6 %%           Es wird ein Verhaeltnis aus trivialen und nicht trivialen Werten
7 %%           erstellt.
8 %% Argumente: N (Groesse der NxN DFT-Matrix)
9 %% Author:    Thomas Lattmann
10 %% Datum:     17.10.2017
11 %% Version:   1.0
12
13 function dft_bewertung(N)
14
15     % Twiddlefaktor-Matrix erzeugen
16     W = exp(-i*2*pi*[0:N-1]'*[0:N-1]/N);
17     W = round(W*1000000)/1000000;
18
19     % Matrix nach Im und Re trennen und Werte runden
20     W_r = real(W);
21     W_i = imag(W);
22
23     % Werte kleiner 0,000001 auf 0 setzen (arithmetische Ungenauigkeiten)
24     W_r(abs(W_r) < 0.000001) = 0;
25     W_i(abs(W_i) < 0.000001) = 0;
26
27
28
29     % 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 imaginären Matrix z hlen
nontrivial_nums_imag = 0;

```

```

85 for k = 1:length(different_nums_imag)
    if abs(different_nums_imag(k)) != 1
        if abs(different_nums_imag(k)) != 0.5
87             if different_nums_imag(k) != 0
                nontrivial_nums_imag = nontrivial_nums_imag + num_count_imag(k);
89             end
            end
91         end
    end
93
    nums_of_each_matrix = N*N;
95
    trivial_nums_real = N*N - nontrivial_nums_real
97    trivial_nums_imag = N*N - nontrivial_nums_imag
99
    nontrivial_nums_real
    nontrivial_nums_imag
101
    trivial_nums_total = trivial_nums_real + trivial_nums_imag
103    nontrivial_nums_total = nontrivial_nums_real + nontrivial_nums_imag
105
    v = trivial_nums_total/nontrivial_nums_total
107 end

```

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

8.2 Gate-Report des 12 Bit Konstantenmultiplizierers

```

1 rc:/> report gates
=====
3  Generated by:      Encounter(R) RTL Compiler RC14.25 - v14.20-s046_1
   Generated on:      May 30 2017  03:29:41 pm
5  Module:           multiplier
   Technology library: c35_CORELIB_TYP 3.02
7  Operating conditions: _nominal_ (balanced_tree)
   Wireload mode:      enclosed
9  Area mode:         timing library
=====
11
13  Gate      Instances    Area      Library
-----
15  ADD21      5         728.000    c35_CORELIB_TYP
   AOI210      2         145.600    c35_CORELIB_TYP
17  AOI220     18        1638.000    c35_CORELIB_TYP
   CLKIN0      6          218.400    c35_CORELIB_TYP
19  IMUX20     38        3458.000    c35_CORELIB_TYP
   INV0        27          982.800    c35_CORELIB_TYP
21  NAND20     12          655.200    c35_CORELIB_TYP
   NOR20       8          436.800    c35_CORELIB_TYP
23  OAI220      6          546.000    c35_CORELIB_TYP
   XNR20      15        1638.000    c35_CORELIB_TYP
25  XNR30       6          1201.200    c35_CORELIB_TYP
   XNR31       3           600.600    c35_CORELIB_TYP

```

```

27 XOR20          5    637.000    c35_CORELIB_TYP
29 total          151  12885.600
31
33  Type    Instances    Area    Area %
35 inverter      33   1201.200    9.3
36 logic         118  11684.400   90.7
37 total          151  12885.600  100.0
39 rc:/>

```

Listing 8.3: RC Gate-Report

8.3 Twiddlefaktormatrix im S1Q10-Format

```

1 %% Dateiname:      twiddle2file.m
2 %% Funktion:       Erzeugt eine Datei mit den binären komplexen
3 %%                 Twiddlefaktoren
4 %% Argumente:      N (Größe der NxN DFT-Matrix)
5 %% Aufbau der Datei: Wie die Matrix, enthält Realteil und Imaginärteil.
6 %%                 Alle Werte sind wie im Beispiel durch Leerzeichen getrennt:
7 %%                 Re{W(1,1)} Im{W(1,1)} Re{W(1,2)} Im{W(1,2)}
8 %%                 Re{W(2,1)} Im{W(2,1)} Re{W(2,2)} Im{W(2,2)}
9 %% Abhängigkeiten: (1) twiddle_coefficients.m
10 %%                (2) dec_to_slq10.m
11 %%                (3) bit_vector2integer.m
12 %%                (4) zweier_komplement.m
13 %% Author:         Thomas Lattmann
14 %% Datum:          02.11.17
15 %% Version:        1.0

17 function twiddle2file(N)

19 % Dezimale Twiddlefaktormatrix erstellen
20 W_dec = twiddle_coefficients(N);
21 W_dec_real = real(W_dec);
22 W_dec_imag = imag(W_dec);
23
24 W_bin_int_real = zeros(size(W_dec_real));
25 W_bin_int_imag = zeros(size(W_dec_imag));
26
27 for m = 1:N
28     for n = 1:N
29         bit_vector = dec_to_slq10(W_dec_real(m,n));
30         W_bin_int_real(m,n) = bit_vector2integer(bit_vector);
31
32         bit_vector = dec_to_slq10(W_dec_imag(m,n));
33         W_bin_int_imag(m,n) = bit_vector2integer(bit_vector);
34     end
35 end
36
37 fid=fopen('Twiddle_slq10_komplex.txt', 'w+');

```

```

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

```

Listing 8.4: Erstellen der Twiddlefaktormatrix-Datei

```

%% Dateiname: twiddle_coefficients.m
2 %% Funktion:  Erstellt eine Matrix (W) mit den Twiddlefaktoren fuer die DFT der
%%             Groesse, die mit N an das Skript uebergeben wurde.
4 %% Argumente: N (Groesse der NxN DFT-Matrix)
%% Author:     Thomas Lattmann
6 %% Datum:     02.11.17
%% Version:    1.0
8
function W = twiddle_coefficients(N)
10
    % Twiddlefaktoren fuer die DFT
12    W = exp(-i*2*pi*[0:N-1]'*[0:N-1]/N)
14
    % auf 6 Nachkommastellen reduzieren
    W = round(W*1000000)/1000000;
16
    % negative Nullen auf 0 setzen
18    W_real = real(W);
    W_imag = imag(W);
20    W_real(abs(W_real)<00000.1) = 0;
    W_imag(abs(W_imag)<00000.1) = 0;
22    W = W_real + i*W_imag;
24 end

```

Listing 8.5: Erzeugen der Twiddlefaktormatrix

```

%% Dateiname: dec_to_slq10.m
2 %% Funktion:  Konvertiert eine Dezimalzahl in das binaere SlQ10-Format
%% Argumente:  Dezimalzahl im Bereich von -2...+2-1/2^10
4 %% Abhaenigkeiten: (1) zweier_komplement.m
%% Author:     Thomas Lattmann
6 %% Datum:     02.11.17
%% Version:    1.0
8
function bit_vector = dec_to_slq10(val)
10
    bit_width=12;

```

```

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

18 if val > 2-1/2^(bit_width-2) % 1.99902... bei 12 Bit und somit 10 Bit fuer
   Nachkomma
   disp('Diese Zahl kann nicht im slq11-Format dargestellt werden.')
20 elseif val < -2
   disp('Diese Zahl kann nicht im slq11-Format dargestellt werden.')
22 else

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

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

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

```

Listing 8.6: Dezimalzahl nach S1Q10 konvertieren

```

1 %% Dateiname: zweier_komplement.m
   %% Funktion: Bilden des 2er-Komplements eines "Bit"-Vektors
3 %% Argumente: Vektor aus Nullen und Einsen
   %% Author: Thomas Lattmann
5 %% Datum: 02.11.17
   %% Version: 1.0

7
   function bit_vector = zweier_komplement(bit_vector)
9     bit_width=length(bit_vector);

11    for j = 1:bit_width
       bit_vector(j) = not(bit_vector(j));
13    end
       bit_vector(bit_width) = bit_vector(bit_width) + 1;
15    for j = 1:bit_width-1
       if bit_vector(bit_width-j+1) == 2
17       bit_vector(bit_width-j+1) = 0;
       end
       end

```

```

19         bit_vector(bit_width - j) = bit_vector(bit_width - j) + 1;
20     end
21 end

```

Listing 8.7: Bildung des 2er-Komplements

```

1 %% Dateiname: bit_vector2integer.m
2 %% Funktion: Wandelt einen Vektor von Zahlen in eine einzelne Zahl (Integer)
3 %%           Beispiel: [0 1 1 0 0 1] => 11001
4 %%           Um fuehrende Nullen zu erhalten muss z.B. printf('%06d', Integer)
5 %%           genutzt werden. Hierbei wird vorne mit Nullen aufgefuellt, wenn
6 %%           'Integer' weniger als 6 stellen hat.
7 %% Argumente: Vektor (aus Nullen und Einsen)
8 %% Author:    Thomas Lattmann
9 %% Datum:     02.11.17
10 %% Version:   1.0
11
12 function bin_int = bit_vector2integer(bit_vector)
13
14     bin_int=0;
15     bit_width=length(bit_vector);
16
17     % Konvertierung von Vektor nach Integer
18     for l = 1:bit_width
19         bin_int = bin_int + bit_vector(bit_width - l + 1)*10^(l-1);
20     end
21 end

```

Listing 8.8: Binär-Vektor in Binär-Integer umwandeln

```

1 %% Dateiname: slq10_to_dec.m
2 %% Funktion: Konvertiert eine binaere Zahl im SlQ10-Format als Dezimalzahl
3 %% Argumente: Vektor aus Nullen und Einsen
4 %% Author:    Thomas Lattmann
5 %% Datum:     02.11.17
6 %% Version:   1.0
7
8 function dec = slq10_to_dec(bit_vector)
9
10     % Dezimalzahl aus slq10 Binaerzahl berechnen
11
12     bit_width=length(bit_vector);
13     dec = 0;
14
15     if bit_vector(1) == 1
16         dec = -2;
17         if bit_vector(2) == 1
18             dec = -1;
19         end
20     elseif bit_vector(2) == 1
21         dec = 1;
22     end
23
24     for n = 3:bit_width
25         if bit_vector(n) == 1
26             dec = dec + 1/2^(n-2);
27         end
28     end
29 end

```

```

    end
28  end
end

```

Listing 8.9: Kontroll-Skript für S1Q10 nach Dezimal

8.4 Programmcode

```

1  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
3
5
   package constants is
7     constant mat_size : integer;
     constant bit_width_extern : integer;
9     constant bit_width_adder : integer;
     constant bit_width_multiplier : integer;
11  end constants;

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

```

Listing 8.10: Deklaration der Konstanten

```

— Package, welches ein 2D-Array bereitstellt.
2 — Das 2D-Array besteht aus 1D-Arrays, dies bringr gegenueber der direkten Erzeugung
   (m,n) statt (m)(n) den Vorteil, dass
— dass zeilen- sowie spaltenweise zugewiesen werden kann. Sonst waere nur die
   komplette Matrix oder einzelne Elemente moeglich.
4
   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
   use ieee.numeric_std.all;
8   library work;
   use work.all;
10  use constants.all;

12
   package datatypes is
14     type t_1d_array is array(integer range 0 to mat_size-1) of signed(
       bit_width_extern-1 downto 0);
     type t_2d_array is array(integer range 0 to mat_size-1) of t_1d_array;
16
     type t_1d_array6_13bit is array(integer range 0 to 5) of signed(bit_width_adder
       -1 downto 0);
18

20     subtype t_twiddle_coeff_long is signed(16 downto 0);
     constant twiddle_coeff_long : t_twiddle_coeff_long := "00101101010000010";
22     subtype t_twiddle_coeff is signed(bit_width_adder-1 downto 0);

```

```

24  --constant twiddle_coeff : t_twiddle_coeff := twiddle_coeff_long(16 downto 16-(
25  bit_width_adder-1));
26
27
28  -- Zustandsautomat 1D-DFT
29  subtype t_dft8_states is std_logic_vector(2 downto 0);
30  constant idle          : t_dft8_states := "000";
31  constant twiddle_calc  : t_dft8_states := "001";
32  constant additions_stage1 : t_dft8_states := "010";
33  constant additions_stage2 : t_dft8_states := "011";
34  constant const_mult     : t_dft8_states := "100";
35  constant additions_stage3 : t_dft8_states := "101";
36  constant set_ready_bit  : t_dft8_states := "110";
37
38  end datatypes;

```

Listing 8.11: Deklaration eigener Datentypen

```

1  library IEEE;
2  use ieee.std_logic_1164.all;
3  --use ieee.std_logic_arith.all;
4  use ieee.numeric_std.all;
5
6  library STD; -- for reading text file
7  use STD.TEXTIO.ALL;
8  use ieee.std_logic_textio.all;
9
10 library work;
11 use work.all;
12 use datatypes.all;
13 use constants.all;
14
15
16 entity read_input_matrix is
17   port(
18     clk          : in  bit;
19     loaded       : out bit;
20     input_real   : out t_2d_array;
21     input_imag   : out t_2d_array;
22   );
23 end entity read_input_matrix;
24
25
26 architecture bhv of read_input_matrix is
27   begin
28     reading : process
29
30       variable element_1_real : std_logic_vector(bit_width_extrn-1 downto 0) := (
31         others => '0');
32       variable element_1_imag : std_logic_vector(bit_width_extrn-1 downto 0) := (
33         others => '0');
34       variable element_2_real : std_logic_vector(bit_width_extrn-1 downto 0) := (
35         others => '0');
36       variable element_2_imag : std_logic_vector(bit_width_extrn-1 downto 0) := (
37         others => '0');
38       variable element_3_real : std_logic_vector(bit_width_extrn-1 downto 0) := (

```



```

others => '0');
variable element_3_imag : std_logic_vector(bit_width_extern-1 downto 0) := (
others => '0');
36 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');
38 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');
40 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');
42 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');
44 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');
46
variable r_space : character;
48
variable fstatus : file_open_status; — status r,w
50 variable inline : line; — readout line
file infile : text; — filehandle for reading ascii text
52
variable textfilename : string(1 to 29);
54
begin
56
58
if bit_width_extern = 12 then
60 textfilename := "InputMatrix_komplex_12Bit.txt";
else
62 textfilename := "InputMatrix_komplex_16Bit.txt";
end if;
64
file_open(fstatus, infile, textfilename, read_mode);
66
— if fstatus = NAME_ERROR then
— file_open(fstatus, infile, "HDL/InputMatrix_komplex.txt", read_mode);
— report "Ausgabe-Datei befindet sich im Unterverzeichnis 'HDL'.";
70 — end if;
72
for i in 0 to mat_size-1 loop
74
wait until clk = '1' and clk'event;
readline(infile, inline);
76 read(infile, element_1_real);
read(infile, r_space);
78 read(infile, element_1_imag);
read(infile, r_space);
80 read(infile, element_2_real);

```

```

82     read (inline , r_space);
      read (inline , element_2_imag);
      read (inline , r_space);
84     read (inline , element_3_real);
      read (inline , r_space);
86     read (inline , element_3_imag);
      read (inline , r_space);
88     read (inline , element_4_real);
      read (inline , r_space);
90     read (inline , element_4_imag);
      read (inline , r_space);
92     read (inline , element_5_real);
      read (inline , r_space);
94     read (inline , element_5_imag);
      read (inline , r_space);
96     read (inline , element_6_real);
      read (inline , r_space);
98     read (inline , element_6_imag);
      read (inline , r_space);
100    read (inline , element_7_real);
      read (inline , r_space);
102    read (inline , element_7_imag);
      read (inline , r_space);
104    read (inline , element_8_real);
      read (inline , r_space);
106    read (inline , element_8_imag);

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

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

132
      end process;
134 end bhv;

```

Listing 8.12: Eingangs-Matrix aus Textdatei einlesen

```
library ieee;
```

```

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

8 entity read_input_matrix_tb is
end entity read_input_matrix_tb;

10 architecture arch of read_input_matrix_tb is
12     signal clk          : bit := '0';
14     signal loaded       : bit := '0';
    signal input_real    : t_2d_array;
16     signal input_imag   : t_2d_array;

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

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

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

```

Listing 8.13: Testbench für das Einlesen aus einer Textdatei

```

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

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

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

16 entity write_results is
18     port(
        result_ready : in  bit;

```

```

20     result_real  : in  t_2d_array;
21     result_imag  : in  t_2d_array;
22     write_done   : out bit
23 );
24 end entity write_results;

26
27
28 architecture bhv of write_results is
29 begin
30     writing_to_file : process(result_ready)
31
32         variable fstatus : file_open_status; — status r,w
33         variable outline : line; — writeout line
34         file        outfile : text; — filehandle
35
36         —variable output1 : bit_vector(3 downto 0) := "0101";
37         —variable output2 : bit_vector(3 downto 0) := "0110";
38
39         variable element_1_real : std_logic_vector(bit_width_extern-1 downto 0);
40         variable element_1_imag : std_logic_vector(bit_width_extern-1 downto 0);
41         variable element_2_real : std_logic_vector(bit_width_extern-1 downto 0);
42         variable element_2_imag : std_logic_vector(bit_width_extern-1 downto 0);
43         variable element_3_real : std_logic_vector(bit_width_extern-1 downto 0);
44         variable element_3_imag : std_logic_vector(bit_width_extern-1 downto 0);
45         variable element_4_real : std_logic_vector(bit_width_extern-1 downto 0);
46         variable element_4_imag : std_logic_vector(bit_width_extern-1 downto 0);
47         variable element_5_real : std_logic_vector(bit_width_extern-1 downto 0);
48         variable element_5_imag : std_logic_vector(bit_width_extern-1 downto 0);
49         variable element_6_real : std_logic_vector(bit_width_extern-1 downto 0);
50         variable element_6_imag : std_logic_vector(bit_width_extern-1 downto 0);
51         variable element_7_real : std_logic_vector(bit_width_extern-1 downto 0);
52         variable element_7_imag : std_logic_vector(bit_width_extern-1 downto 0);
53         variable element_8_real : std_logic_vector(bit_width_extern-1 downto 0);
54         variable element_8_imag : std_logic_vector(bit_width_extern-1 downto 0);
55         variable space : character := ' ';
56
57     begin
58         file_open(fstatus, outfile, "/home/tlattmann/cadence/mat_mult/HDL/Results.txt"
59 , write_mode);
60
61         —if result_ready = '1' then
62
63         for i in 0 to mat_size-1 loop
64             element_1_real := std_logic_vector(result_real(i)(0));
65             element_1_imag := std_logic_vector(result_imag(i)(0));
66             element_2_real := std_logic_vector(result_real(i)(1));
67             element_2_imag := std_logic_vector(result_imag(i)(1));
68             element_3_real := std_logic_vector(result_real(i)(2));
69             element_3_imag := std_logic_vector(result_imag(i)(2));
70             element_4_real := std_logic_vector(result_real(i)(3));
71             element_4_imag := std_logic_vector(result_imag(i)(3));
72             element_5_real := std_logic_vector(result_real(i)(4));
73             element_5_imag := std_logic_vector(result_imag(i)(4));
74             element_6_real := std_logic_vector(result_real(i)(5));
75             element_6_imag := std_logic_vector(result_imag(i)(5));
76             element_7_real := std_logic_vector(result_real(i)(6));
77             element_7_imag := std_logic_vector(result_imag(i)(6));

```

```

78     element_8_real := std_logic_vector(result_real(i)(7));
    element_8_imag := std_logic_vector(result_imag(i)(7));

80     write(outline , element_1_real);
    write(outline , space);
82     write(outline , element_1_imag);
    write(outline , space);
84     write(outline , element_2_real);
    write(outline , space);
86     write(outline , element_2_imag);
    write(outline , space);
88     write(outline , element_3_real);
    write(outline , space);
90     write(outline , element_3_imag);
    write(outline , space);
92     write(outline , element_4_real);
    write(outline , space);
94     write(outline , element_4_imag);
    write(outline , space);
96     write(outline , element_5_real);
    write(outline , space);
98     write(outline , element_5_imag);
    write(outline , space);
100    write(outline , element_6_real);
    write(outline , space);
102    write(outline , element_6_imag);
    write(outline , space);
104    write(outline , element_7_real);
    write(outline , space);
106    write(outline , element_7_imag);
    write(outline , space);
108    write(outline , element_8_real);
    write(outline , space);
110    write(outline , element_8_imag);

112    writeline(outfile , outline);
end loop;
114

    write_done <= '1';
    file_close(outfile);
    —end if;
118

end process;
120 end bhv;

```

Listing 8.14: Ergebnis-Matrix in Textdatei schreiben

```

library IEEE;
2 use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
4
  library STD; — for writing text file
6 use STD.TEXTIO.ALL;
  use ieee.std_logic_textio.all;
8
  library work;
10 use work.all;
  use datatypes.all;

```

```

12 use constants.all;
14
16 entity write_test_tb is
17 end entity write_test_tb;
18
19 architecture bhv of write_test_tb is
20
21     signal clk          : bit;
22     signal loaded       : bit;
23     signal result_ready : bit;
24     signal write_done   : bit;
25     signal loop_running : bit;
26     signal loop_number  : signed(2 downto 0);
27     signal input_real   : t_2d_array;
28     signal input_imag   : t_2d_array;
29     signal output       : std_logic_vector(bit_width_extern-1 downto 0);
30
31     component read_input_matrix
32     port(
33         clk          : in  bit;
34         loaded       : out bit;
35         input_real   : out t_2d_array;
36         input_imag   : out t_2d_array
37     );
38 end component;
39
40 component write_results
41 port(
42     result_ready : in bit;
43     result_real  : in t_2d_array;
44     result_imag  : in t_2d_array;
45     write_done   : out bit;
46     loop_number  : out signed(2 downto 0);
47     loop_running : out bit;
48     output       : out std_logic_vector(bit_width_extern-1 downto 0)
49 );
50 end component;
51
52 begin
53
54     mat : read_input_matrix
55     port map(
56         clk          => clk,
57         loaded       => loaded,
58         input_real   => input_real,
59         input_imag   => input_imag
60     );
61
62     write : write_results
63     port map(
64         result_ready => result_ready,
65         result_real  => input_real,
66         result_imag  => input_imag,
67         write_done   => write_done,
68         loop_number  => loop_number,
69         loop_running => loop_running,

```

```

70         output      => output
71     );
72
73     result_ready <= loaded after 20 ns;
74     clk         <= not clk after 10 ns;
75
76 end bhv;

```

Listing 8.15: Testbench für das schreiben in eine Textdatei

```

library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
  use ieee.numeric_std.all;
4 library work;
  use work.all;
6 use datatypes.all;
  use constants.all;
8
10 library STD; — for reading text file
  use STD.TEXTIO.ALL;
12 use ieee.std_logic_textio.all;
14 entity dft8optimiert is
  port(
16     clk           : in  bit;
17     nReset        : in  bit;
18     loaded        : in  bit;
19     input_real     : in  t_2d_array;
20     input_imag     : in  t_2d_array;
21     result_real    : out t_2d_array;
22     result_imag    : out t_2d_array;
23     result_ready   : out bit;
24     idft           : in  bit;
25     state_out      : out t_dft8_states;
26     element_out    : out unsigned(5 downto 0);
27     dft_1d_2d_out  : out bit
28 );
  end dft8optimiert;
30
32 architecture arch of dft8optimiert is
33
34     signal dft_state, next_dft_state : t_dft8_states;
35
36 begin
37
38     FSM_TAKT: process(clk)
39     begin
40         if clk='1' and clk'event then
41             dft_state <= dft_state;
42             state_out <= dft_state;
43             if nReset='0' then
44                 dft_state <= idle;
45                 state_out <= idle;
46             elsif loaded = '0' then
47                 dft_state <= idle;

```

```

state_out <= idle;
50  elsif loaded='1' and dft_state = idle then
    dft_state <= twiddle_calc;
52  state_out <= twiddle_calc;
    else
54  dft_state <= next_dft_state;
    state_out <= next_dft_state;
56  end if;
    end if;
58 end process;

60
FSM_KOMB: process(dft_state)
62  —constant twiddle_coeff : signed(16 downto 0) := "00010110101000001";
    variable twiddle_coeff : signed(bit_width_adder-1 downto 0);
64
    variable mult_re, mult_im : signed(bit_width_multiplier-1 downto 0);
66
    variable W_row, I_col : integer;
    variable dft_1d_real, dft_1d_imag : t_2d_array;
68  variable matrix_real, matrix_imag : t_2d_array;
    variable temp_re, temp_im : t_1d_array6_13bit;
70  variable temp14bit_re, temp14bit_im : signed(bit_width_adder downto 0);
    variable dft_1d_2d : bit;
72  variable element : unsigned(5 downto 0) := "000000";
74
76  variable row_col_idx : integer := 0;
78  —variable LineBuffer : LINE;
80
begin
82  twiddle_coeff := "0001011010100";
    — Flip-Flops
84  — werden das 1. Mal sich selbst zu gewiesen, bevor sie einen Wert haben!
    result_ready <= '0';
86  element := element;
    dft_1d_2d := dft_1d_2d;
88  temp_re := temp_re;
    temp_im := temp_im;
90  mult_re := mult_re;
    mult_im := mult_im;
92  dft_1d_real := dft_1d_real;
    dft_1d_imag := dft_1d_imag;
94  matrix_real := matrix_real;
    matrix_imag := matrix_imag;
96  dft_1d_2d_out <= dft_1d_2d;
98
    — Die Matrix hat 64 Elemente -> 2^6=64 -> 6-Bit Vektor passt genau. Ueberlauf =
    1. Element vom n chsten Durchlauf.
100  — Der Elemente-Vektor kann darueber hinaus in vordere Haelfte = Zeile und
    hintere Haelfte = Spalte aufgeteilt werden.
    — So laesst sich auch ein Matrix-Element mit zwei Indizes ansprechen:
102
    — Bei der IDFT sind die Zeilen 1 und 7, 2 und 6, 3 und 5 vertauscht. 1 und 4
    bleiben wie sie sind.

```



```

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

108   if idft = '1' then
109     if row_col_idx = 0 then
110       W_row := 0;
111     else
112       W_row := 8-row_col_idx; — Twiddlefaktor-Matrix
113     end if;
114   else
115     W_row := row_col_idx; — Twiddlefaktor-Matrix
116   end if;

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

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

130
131   case dft_state is
132     when idle =>
133       next_dft_state <= twiddle_calc;

134
135     when twiddle_calc => — dft_state_out = 1
136       — Mit resize werden die 12 Bit Eingangswerte vorzeichengerecht auf 13 Bit
137       — erweitert, um die richtige Groesse zu haben.
138       — Bei der Addition muessen die Summanden die gleiche Bit-Breite wie der
139       — Ergebnis-Vektor haben.
140       case W_row is
141         — Die Faktoren (Koeffizienten) der Twiddlefaktor-Matrix W lassen sich
142         ueber  $\exp(-i*2*\pi*[0:7]*[0:7]/8)$  berechnen.
143         — 1. Zeile aus W -> nur Additionen
144         when 0 =>
145           — Die 1. Zeile aus W besteht nur aus den Faktoren (1+j0). Daraus
146           resultiert, dass die reellen
147           — und die imaginaeren Werte der Eingangs-Matrix unabhaengig von
148           einander aufsummiert werden.
149           — Real
150           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)(I_col), bit_width_adder) - resize(
matrix_imag(3)(I_col), bit_width_adder);
temp_im(5) := resize(matrix_imag(7)(I_col), bit_width_adder) - resize(
matrix_imag(5)(I_col), bit_width_adder);

```

```

186      — 3. Zeile aus W
187      — 0: (1.00000 + 0.00000i), 1: (0.00000 + 1.00000i), 2: (-1.00000 +
188      0.00000i), 3: (-0.00000 - 1.00000i),
189      — 4: (1.00000 - 0.00000i), 5: (0.00000 + 1.00000i), 6: (-1.00000 +
190      0.00000i), 7: (-0.00000 - 1.00000i)
191      when 2 =>
192      — Real
193      temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder) - resize(
194      matrix_real(2)(I_col), bit_width_adder);
195      temp_re(1) := resize(matrix_imag(1)(I_col), bit_width_adder) - resize(
196      matrix_imag(3)(I_col), bit_width_adder);
197      temp_re(2) := resize(matrix_real(4)(I_col), bit_width_adder) - resize(
198      matrix_real(6)(I_col), bit_width_adder);
199      temp_re(3) := resize(matrix_imag(5)(I_col), bit_width_adder) - resize(
200      matrix_imag(7)(I_col), bit_width_adder);
201      — Imag
202      temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(
203      matrix_real(1)(I_col), bit_width_adder);
204      temp_im(1) := resize(matrix_real(3)(I_col), bit_width_adder) - resize(
205      matrix_imag(2)(I_col), bit_width_adder);
206      temp_im(2) := resize(matrix_imag(4)(I_col), bit_width_adder) - resize(
207      matrix_real(5)(I_col), bit_width_adder);
208      temp_im(3) := resize(matrix_real(7)(I_col), bit_width_adder) - resize(
209      matrix_imag(6)(I_col), bit_width_adder);
210
211      — 4. Zeile aus W
212      — 0: ( 1.00000 + 0.00000i), 1: (-0.70711 + 0.70711i), 2: (-0.00000 -
213      1.00000i), 3: ( 0.70711 + 0.70711i)
214      — 4: (-1.00000 + 0.00000i), 5: ( 0.70711 - 0.70711i), 6: ( 0.00000 +
215      1.00000i), 7: (-0.70711 - 0.70711i)
216      when 3 =>
217      — Real
218      temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder) - resize(
219      matrix_imag(2)(I_col), bit_width_adder);
220      temp_re(1) := resize(matrix_imag(6)(I_col), bit_width_adder) - resize(
221      matrix_real(4)(I_col), bit_width_adder);
222      —MultPart
223      temp_re(2) := resize(matrix_imag(1)(I_col), bit_width_adder) - resize(
224      matrix_real(1)(I_col), bit_width_adder);
225      temp_re(3) := resize(matrix_real(3)(I_col), bit_width_adder) - resize(
226      matrix_imag(5)(I_col), bit_width_adder);
227      temp_re(4) := resize(matrix_imag(3)(I_col), bit_width_adder) - resize(
228      matrix_imag(7)(I_col), bit_width_adder);
229      temp_re(5) := resize(matrix_real(5)(I_col), bit_width_adder) - resize(
230      matrix_real(7)(I_col), bit_width_adder);
231
232      — Imag
233      temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(
234      matrix_imag(4)(I_col), bit_width_adder);
235      temp_im(1) := resize(matrix_real(2)(I_col), bit_width_adder) - resize(
236      matrix_real(6)(I_col), bit_width_adder);
237      —MultPart
238      temp_im(2) := resize(matrix_imag(3)(I_col), bit_width_adder) - resize(
239      matrix_real(1)(I_col), bit_width_adder);
240      temp_im(3) := resize(matrix_real(5)(I_col), bit_width_adder) - resize(
241      matrix_imag(1)(I_col), bit_width_adder);
242      temp_im(4) := resize(matrix_imag(5)(I_col), bit_width_adder) - resize(
243      matrix_real(3)(I_col), bit_width_adder);

```

```

temp_im(5) := resize(matrix_real(7)(I_col), bit_width_adder) - resize(
matrix_imag(7)(I_col), bit_width_adder);
222
    — 5. Zeile
224    — 0: (1.00000 + 0.00000i), 1: (-1.00000 + 0.00000i), 2: (1.00000 -
0.00000i), 3: (-1.00000 + 0.00000i),
    — 4: (1.00000 - 0.00000i), 5: (-1.00000 + 0.00000i), 6: (1.00000 -
0.00000i), 7: (-1.00000 + 0.00000i)
226    when 4 =>
    — Real
228    temp_re(0) := resize(matrix_real(0)(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);
230    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);
232    — Imag
    temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(
matrix_imag(1)(I_col), bit_width_adder);
234    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);
236    temp_im(3) := resize(matrix_imag(6)(I_col), bit_width_adder) - resize(
matrix_imag(7)(I_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),
240    — 4: (-1.00000 + 0.00000i) 5: ( 0.70711 + 0.70711i), 6: (-0.00000 -
1.00000i), 7: (-0.70711 + 0.70711i)
    when 5 =>
242    — Real
    temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder) - resize(
matrix_real(4)(I_col), bit_width_adder);
244    temp_re(1) := resize(matrix_imag(2)(I_col), bit_width_adder) - resize(
matrix_imag(6)(I_col), bit_width_adder);
    —MultPart
246    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);
248    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);
250    — Imag
    temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(
matrix_real(2)(I_col), bit_width_adder);
252    temp_im(1) := resize(matrix_real(6)(I_col), bit_width_adder) - resize(
matrix_imag(4)(I_col), bit_width_adder);
    —MultPart
254    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);

```

```

256         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);
258
        — 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)
262         when 6 =>
            — Real
264             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);
266             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);
268             — Imag
            temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(
matrix_imag(2)(I_col), bit_width_adder);
270             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);
272             temp_im(3) := resize(matrix_real(5)(I_col), bit_width_adder) - resize(
matrix_real(7)(I_col), bit_width_adder);

274             — 8. Zeile
            — 0: ( 1.00000 + 0.00000i), 1: ( 0.70711 - 0.70711i), 2: (-0.00000 -
1.00000i), 3: (-0.70711 - 0.70711i),
276             — 4: (-1.00000 + 0.00000i), 5: (-0.70711 + 0.70711i), 6: (-0.00000 +
1.00000i), 7: ( 0.70711 + 0.70711i)
            when 7 =>
                — Real
278                 temp_re(0) := resize(matrix_real(0)(I_col), bit_width_adder) - resize(
matrix_imag(2)(I_col), bit_width_adder);
280                 temp_re(1) := resize(matrix_imag(6)(I_col), bit_width_adder) - resize(
matrix_real(4)(I_col), bit_width_adder);
                —MultPart
282                 temp_re(2) := resize(matrix_real(1)(I_col), bit_width_adder) - resize(
matrix_imag(1)(I_col), bit_width_adder);
                temp_re(3) := resize(matrix_imag(5)(I_col), bit_width_adder) - resize(
matrix_real(3)(I_col), bit_width_adder);
284                 temp_re(4) := resize(matrix_real(7)(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(5)(I_col), bit_width_adder);
286                 — Imag
                temp_im(0) := resize(matrix_imag(0)(I_col), bit_width_adder) - resize(
matrix_imag(4)(I_col), bit_width_adder);
288                 temp_im(1) := resize(matrix_real(2)(I_col), bit_width_adder) - resize(
matrix_real(6)(I_col), bit_width_adder);
                —MultPart
290                 temp_im(2) := resize(matrix_real(1)(I_col), bit_width_adder) - resize(
matrix_imag(3)(I_col), bit_width_adder);

```

```

temp_im(3) := resize(matrix_imag(1)(I_col), bit_width_adder) - resize(
matrix_real(5)(I_col), bit_width_adder);
temp_im(4) := resize(matrix_real(3)(I_col), bit_width_adder) - resize(
matrix_imag(5)(I_col), bit_width_adder);
temp_im(5) := resize(matrix_imag(7)(I_col), bit_width_adder) - resize(
matrix_real(7)(I_col), bit_width_adder);

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

next_dft_state <= additions_stagel;

when additions_stagel => — 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
328   temp_im(2) := resize(temp_im(2)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(3)(bit_width_adder-1 downto 1),
bit_width_adder);
   temp_im(4) := resize(temp_im(4)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(5)(bit_width_adder-1 downto 1),
bit_width_adder);
330   end if;

332   next_dft_state <= additions_stage2;

334
336   when additions_stage2 => — dft_state_out = 3
   — Es wird vor jeder Addition ein Bitshift auf die Summanden angewandt, um
   den Wertebereich der Speichervariable nicht zu ueberschreiten (2. Mal)
   — Zusaetzlich wird wird beim Zuweisen der ungeraden Zeilen an die 1D-DFT-
   Matrix zwei wweitere Male geshiftet.
338   — 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

340   — Zeilen 1, 3, 5, 7 (wie oben)
   if element(3) = '0' then
342
   — Real
344   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
346   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);

348   — 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.
350   — Indizes vertauschen -> Transponiert abspeichern
   if dft_1d_2d = '0' then
352     dft_1d_real(I_col)(row_col_idx) := resize(temp_re(0)(bit_width_adder
-1 downto 2), bit_width_extern);
     dft_1d_imag(I_col)(row_col_idx) := resize(temp_im(0)(bit_width_adder
-1 downto 2), bit_width_extern);
354   else
     result_real(I_col)(row_col_idx) <= resize(temp_re(0)(bit_width_adder
-1 downto 2), bit_width_extern);
356     result_imag(I_col)(row_col_idx) <= resize(temp_im(0)(bit_width_adder
-1 downto 2), bit_width_extern);
     end if;

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

362   — naechster Zustand
   next_dft_state <= twiddle_calc;

364
366   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);
368
    -- Imag
370    temp_im(2) := resize(temp_im(2)(bit_width_adder-1 downto 1),
bit_width_adder) + resize(temp_im(4)(bit_width_adder-1 downto 1),
bit_width_adder);

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

376
when const_mult => -- dft_state_out = 4
378
    -- Der Zielvektor der Multiplikation ist 26 Bit breit, die beiden
    Multiplikanten sind mit je 13 Bit wie gefordert halb so breit.
380
    -- 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));
384
    next_dft_state <= additions_stage3;
386

388 when additions_stage3 => -- dft_state_out = 5

    -- Die vordersten 12 Bit des Multiplikationsergebnisses werden verwendet und
    um 1 Bit nach rechts geschiftet, 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.
392    -- Real

    templ4bit_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) := templ4bit_re(bit_width_adder downto 1);
396

    -- Imag
398    templ4bit_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) := templ4bit_im(bit_width_adder downto 1);
400

    -- Indizes vertauschen -> Transponiert abspeichern
402 if dft_ld_2d = '0' then
    dft_ld_real(I_col)(row_col_idx) := temp_re(0)(bit_width_adder-1 downto 1);
    dft_ld_imag(I_col)(row_col_idx) := temp_im(0)(bit_width_adder-1 downto 1);
404 else
    result_real(I_col)(row_col_idx) <= temp_re(0)(bit_width_adder-1 downto 1);
    result_imag(I_col)(row_col_idx) <= temp_im(0)(bit_width_adder-1 downto 1);
406 end if;
408

410 next_dft_state <= twiddle_calc;
    if element = 63 then
412        if dft_ld_2d = '1' then
            next_dft_state <= set_ready_bit;

```



```

414         end if;
415         dft_1d_2d := not dft_1d_2d;
416         dft_1d_2d_out <= dft_1d_2d;
417     end if;
418
419
420     element := element+1;
421     element_out <= element;
422
423
424     when set_ready_bit =>
425         result_ready <= '1';
426         next_dft_state <= twiddle_calc;
427
428
429     when others => next_dft_state <= twiddle_calc;
430 end case;
431
432 end process;
end arch;

```

Listing 8.16: Berechnung der 2D-DFT

```

library ieee;
2 use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
4 library work;
use work.all;
6 use constants.all;
use datatypes.all;
8
entity dft8optimiert_top is
10   port(
11       result_real : out t_2d_array;
12       result_imag : out t_2d_array
13   );
14 end entity dft8optimiert_top;
15
16 architecture arch of dft8optimiert_top is
17
18     signal nReset      : bit;
19     signal clk         : bit;
20     signal input_real  : t_2d_array;
21     signal input_imag  : t_2d_array;
22     signal result_real : t_2d_array;
23     signal result_imag : t_2d_array;
24     signal loaded      : bit;
25     signal result_ready : bit;
26     signal write_done  : bit;
27     signal idft        : bit := '0';
28
29     signal state_out    : t_dft8_states;
30     signal element_out  : unsigned(5 downto 0);
31     signal dft_1d_2d_out : bit;
32
33
34     component dft8optimiert
35         port(

```

```

36         clk           : in  bit;
37         nReset        : in  bit;
38         loaded        : in  bit;
39         input_real     : in  t_2d_array;
40         input_imag     : in  t_2d_array;
41         result_real    : out t_2d_array;
42         result_imag    : out t_2d_array;
43         result_ready   : out bit;
44         idft           : in  bit;
45         state_out      : out t_dft8_states;
46         element_out    : out unsigned(5 downto 0);
47         dft_1d_2d_out : out bit
48     );
49 end component;
50
51 component read_input_matrix
52 port(
53     clk           : in  bit;
54     loaded        : out bit;
55     input_real    : out t_2d_array;
56     input_imag    : out t_2d_array
57 );
58 end component;
59
60
61 component write_results
62 port(
63     result_ready : in  bit;
64     result_real  : in  t_2d_array;
65     result_imag  : in  t_2d_array;
66     write_done   : out bit
67 );
68 end component;
69
70
71 begin
72     dft : dft8optimiert
73     port map(
74         nReset      => nReset,
75         clk         => clk,
76         loaded      => loaded,
77         input_real  => input_real,
78         input_imag  => input_imag,
79         result_real  => result_real,
80         result_imag  => result_imag,
81         result_ready => result_ready,
82         idft        => idft,
83         state_out   => state_out,
84         element_out  => element_out,
85         dft_1d_2d_out => dft_1d_2d_out
86     );
87
88     mat : read_input_matrix
89     port map(
90         clk         => clk,
91         loaded      => loaded,
92         input_real  => input_real,

```

```

94         input_imag => input_imag
95     );
96
97     write : write_results
98     port map(
99         result_ready => result_ready,
100         result_real  => result_real,
101         result_imag  => result_imag,
102         write_done   => write_done
103     );
104
105     clk    <= not clk after 20 ns;
106     nReset <= '1' after 40 ns;
107
108 end arch;

```

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

8.5 Testumgebung

```

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

```

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

tlab

```

1  #!/bin/bash
3  # global settings
5  errormax=15
   worklib=worklib
7  #testbench=top_level_tb
   testbench=dft8optimiert_top
9  architecture=arch
   simulation_time="1500ns"
11
13 # VHDL-files
15 constant_declarations="constants.vhdl"
   datatype_declarations="datatypes.vhdl"
17
   main_entity="dft8optimiert.vhdl"
19 top_level_entity="dft8_optimiert_top.vhdl"
   #top_level_testbench=
21
   embedded_entity_1="read_input_matrix.vhdl"
23 embedded_entity_2="write_results.vhdl"

```

```

25 constant_declarations=$directory$constant_declarations
27 datatype_declarations=$directory$datatype_declarations
29 function_declarations=$directory$function_declarations
31 main_entity=$directory$main_entity
   top_level_entity=$directory$top_level_entity
   #top_level_testbench=$directory$top_level_testbench

33 embedded_entity_1=$directory$embedded_entity_1
   embedded_entity_2=$directory$embedded_entity_2
35

37 # libs und logfiles

39 cdslib="cds.lib"
   elab_logfile="ncelab.log"
41 ncvhdl_logfile="nchvdl.log"
   ncsim_logfile="ncsim.log"
43

45 cdslib=${base_dir}${work_dir}${cdslib}
   elab_logfile=${directory}${elab_logfile}
   ncvhdl_logfile=${directory}${ncvhdl_logfile}
47 ncsim_logfile=${directory}${ncsim_logfile}

49 ##

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

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

77 ncsim \
79 -cdslib $cdslib \
   -logfile $ncsim_logfile \
81 -errormax $errormax \

```

```

83 -exit \
   ${worklib}.${testbench}:${architecture} \
   -input testRUN.tcl
85 #-status \
87
89 #ncvhd1 -work worklib -cdslib /home/tlattmann/cadence/mat_mult/cds.lib -logfile /
   home/tlattmann/cadence/mat_mult/nchvdl.log -errormax 15 -update -v93 -linedebug
   /home/tlattmann/cadence/mat_mult/HDL/constants.vhdl /home/tlattmann/cadence/
   mat_mult/HDL/datatypes.vhdl /home/tlattmann/cadence/mat_mult/HDL/functions.vhdl
   /home/tlattmann/cadence/mat_mult/HDL/read_input_matrix.vhdl /home/tlattmann/
   cadence/mat_mult/HDL/write_results.vhdl /home/tlattmann/cadence/mat_mult/HDL/
   dft8optimiert.vhdl /home/tlattmann/cadence/mat_mult/HDL/dft8_optimiert_top.vhdl
   -status
91 #ncelab -work worklib -cdslib /home/tlattmann/cadence/mat_mult/cds.lib -logfile /
   home/tlattmann/cadence/mat_mult/ncelab.log -errormax 15 -access +wc worklib.
   dft8optimiert_top -status
93 #ncsim -cdslib /home/tlattmann/cadence/mat_mult/cds.lib -logfile /home/tlattmann/
   cadence/mat_mult/ncsim.log -errormax 15 worklib.dft8_optimiert_top:arch -input
   testRUN.tcl -status
95 #database -open waves -into waves.shm -default
   #probe -create -shm :clk :input_imag :input_real :loaded :mult_im_out :mult_re_out :
   multState_out :nReset :result_imag :result_ready :result_real :
   sum1_stage1_3v6_re_out :sum1_stage2_2v3_re_out :sum1_stage2_3v3_re_out :
   sum1_stage3_lv1_re_out :sum3_stage1_im_out :sum3_stage1_re_out :
   sum3_stage2_im_out :sum3_stage2_re_out :sum3_stage3_im_out :sum3_stage3_re_out :
   sum3_stage4_im_out :sum3_stage4_re_out :write_done

```

Listing 8.19: Simulations des VHDL-Quelltextes

```
run 32us
```

Listing 8.20: Dauer der Simulation

```

1 filename_2 = 'InputMatrix_komplex.txt';
  filename_1 = 'Results.txt';
3
4 delimiterIn = ' ';
5
6 bit_width_extern = 13
7
8 Input_bin = importdata(filename_2, delimiterIn);
9 Input_bin_real = Input_bin(:,1:2:end);
  Input_bin_imag = Input_bin(:,2:2:end);
11
12 Results_vhdl_bin = importdata(filename_1, delimiterIn);
13 Results_vhdl_bin_real = Results_vhdl_bin(:,1:2:end);
  Results_vhdl_bin_imag = Results_vhdl_bin(:,2:2:end);
15
16
17 Input_dec_imag = nan(8);
  Results_vhdl_dec_real = nan(8);
19 Results_vhdl_dec_imag = nan(8);
  Result_octave_real_ld = nan(8);

```

```

21 Result_octave_imag_1d = nan(8);
23
25 a=fi(0,1,bit_width_extern,bit_width_extern-2);
27
28 N = 8;
29 for m = 1:N
30     for n = 1:N
31         a.bin=mat2str(Results_vhdl_bin_real(m,n),bit_width_extern);
32         Results_vhdl_dec_real(m,n) = a.double;
33         a.bin=mat2str(Results_vhdl_bin_imag(m,n),bit_width_extern);
34         Results_vhdl_dec_imag(m,n) = a.double;
35
36         a.bin=mat2str(Input_bin_real(m,n),bit_width_extern);
37         Input_dec_real(m,n) = a.double;
38         a.bin=mat2str(Input_bin_imag(m,n),bit_width_extern);
39         Input_dec_imag(m,n) = a.double;
40     end
41 end
42
43 Input_dec=Input_dec_real+1i*Input_dec_imag;
44
45 TW=exp(-i*2*pi*[0:7]'*[0:7]/8);
46
47
48
49 %Result_octave_1d=TW*Input_dec;
50 %Result_octave_real_1d=real(Result_octave_1d)/16
51 %Result_octave_imag_1d=imag(Result_octave_1d)
52
53 Result_octave=TW*Input_dec*TW.';
54 Result_octave=Result_octave./256;
55
56 Results_vhdl_dec_real
57 Result_octave_real=real(Result_octave)
58
59 Result_octave_imag=imag(Result_octave);
60 Results_vhdl_dec_imag;
61
62 diff_real=Result_octave_real-Results_vhdl_dec_real
63 diff_imag=Result_octave_imag-Results_vhdl_dec_imag;
64
65 quit

```

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