Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

HAW HAMBURG

Masterarbeit

Andrey Luzhanskij

VHDL-Implementierung der zweidimensionalen Fourier-Transformation und Ortsfrequenz-Filterung für magnetische Sensor-Arrays

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Hamburg University of Applied Sciences
Faculty of Computer Science and Engineering
Department Information and Electrical Engineering

Andrey Luzhanskij

VHDL-Implementierung der zweidimensionalen Fourier-Transformation und Ortsfrequenz-Filterung für magnetische Sensor-Arrays

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Karl-Ragmar Riemschneider, HAW Hamburg
Zweitgutachter: Prof. Dr.-Ing. habil. Michael Berger, FH Westküste

Eingereicht am: 18. Dezember 2019

Andrey Luzhanskij

Thema der Arbeit

VHDL-Implementierung der zweidimensionalen Fourier-Transformation und Ortsfrequenz-Filterung für magnetische Sensor-Arrays

Stichworte

Fouiertransformation, 2D-DFT, VHDL, 2D-Filterung, FPGA, ASIC

Kurzzusammenfassung

In der vorliegenden Masterarbeit werden zwei Signalverarbeitungsmodule für magnetische Sensorarrays in VHDL für den zukünftigen Einsatz auf einem ASIC entworfen. Zuerst wird die 2D-DFT implementiert. Darauf folgt die Implementierung der 2D-Filterung sowohl im Ortsfrequenzbereich als auch über die 2D-Faltung. Alle Module werden gründlich auf einem FPGA mittels eines Testsystems verifiziert, das in dieser Arbeit auch verbessert wird. Schließlich werden die Module zwecks der Abschätzung vom Logikaufwand und vom Flächeverbrauch auf einem ASIC in Cadence Toolchain implementiert.

Andrey Luzhanskij

Title of Thesis

VHDL Implementation of the Two-Dimensional Fourier Transform and Spatial Frequency Filtering for Magnetic Sensor Arrays

Keywords

Fouier transform, 2D-DFT, VHDL, 2D-filtering, FPGA, ASIC

Abstract

In this master thesis two signal processing modules are designed in VHDL for magnetic sensor arrays for the future application on an ASIC. At first the 2D-DFT is implementied. This is followed by the implemantation of the 2D-filtering both in the spatial frequency domain and over 2D-convolution. All modules are thoroughly verified on an FPGA by means of a test system, that is improved in this thesis. Finally the modules are implementied in the Cadence toolchain for the purpose of estimation of logic size and area requirement.

Inhaltsverzeichnis

1 Einleitung

1.1 Das Forschungsprojekt “ISAR”

Diese Arbeit wurde im Rahmen des Forschungsprojektes ISAR[1] an der Hochschule für Angewandte Wissenschaften Hamburg angefertigt. Das Ziel des Projektes ist die Entwicklung eines Sensorarrays auf Basis von Tunnel-Magnetoresisitiven (TMR) Sensoren für den Einsatz in der Automobilelektronik, das primär zur Bestimmung von Winkeln dienen soll. Mit dem Array verspricht man sich (gegenüber von einzelnen Sensoren) dank der Ortsauflösung die Robustheit gegen äußere Störfelder sowie die Erkennung und die Korrektur jeglicher Positionsabweichungen aus der idealen Lage. Um dies zu erreichen, sollen die Messwerte der Sensoren vor der Winkelberechnung verarbeitet werden. Im Detail werden die Werte mittels der zweidimensionalen diskreten Fouriertransformation in die Ortsfrequenzbereich übertragen, dann gefiltert und zurücktransformiert. Optional werden die Messwerte vor der Transformation interpoliert, was jedoch zum Zeitpunkt der Arbeit noch nicht festgelegt wurde. Letztendlich sollen sowohl das Sensorarray als auch die Verarbeitung gemeinsam auf einem ASIC[2] realisiert werden.

Der aktuelle Prototyp des Sensorarrays aus den diskreten Bauelementen wurde von T. Mehm [11] entwickelt. Er stellt eine quadratische 8×8 Matrix aus TMR-Magnetsensoren dar. Jeder Sensor misst die magnetische Feldstärke in x- und y-Richtung (jeweils als “Kosinus”- und “Sinus”-Signale bezeichnet), die als komplexe Werte aufgefasst werden können. Die Sensoren liefern die Messungen für die jeweilige Richtung als analoge Differenzsignale, von denen jede Komponente einzeln mittels zweier 12-Bit ADU digitalisiert wird. Die Magnetfeldstärke in beiden Richtungen berechnet man aus den jeweiligen positiven

[1] **I**ntegrated **S**ensor **AR**ray
[2] Application specific integrated circuit

und negativen Signalen wie folgt:

$$H_x = \frac{cos_+ - cos_-}{2}$$
$$H_y = \frac{sin_+ - sin_-}{2} \tag{1.1}$$

1.2 Aufgabenstellung

Das Ziel der vorliegenden Arbeit ist die Entwicklung der zwei Glieder der Verarbeitungskette, der 2D-DFT und der anschließenden Filterung im Ortsfrequenzbereich, als VHDL-Module in Fixkommazahlen für das komplexwertige 8×8 Sensorarray. Im Unterkapitel 4.2 wird jedoch eine zusätzliche Alternative dazu vorgeschlagen. In der Entwicklung soll der Schwerpunkt auf der Einfachheit der Implementierung und der Verifizierbarkeit liegen, weil die Module gründlich auf dem FPGA getestet werden sollen. Zum Zwecke der Verifikation soll ein Testsystem [1] in Betrieb genommen werden, und, falls sich dafür eine Gelegenheit bietet, im Sinne des einfacheren Testablaufs verbessert werden.

Da die Fixkommazahlen überlaufen können, soll insbesondere bei der 2D-DFT die numerischen Eigenschaften in dieser Hinsicht betrachtet werden. Ferner sind die benötigen Bitbreiten für die erforderliche Genauigkeit zu bestimmen.

Falls die Zeit ausreicht, sollen die Sensordaten in Matlab interpoliert werden und die 2D-DFT soll auch für das 15×15 Sensorarray implementiert und verifiziert werden. Zum Schluss soll die Implementierung für den 350 nm Prozess der Firma AMS zwecks der realistischen Einschätzung des Logikaufwandes und des Flächenbedarfs auf einem ASIC erfolgen.

2 Das ISAR-Testsystem

2.1 Überblick

Für die Entwicklung und Verifikation der Datenverarbeitungsmodule wurde ein Testsystem entwickelt [1]. Das System besteht aus einer Zedboard-Entwicklingsplatine mit dem Xilinx Zynq-7000 FPGA, einer EK-TM4C1294XL [6] Evaluationsplatine auf Basis vom Texas Instruments TM4C1294NCPDT Mikrocontroller [4] und einem PC unter Linux Ubuntu Betriebssystem mit (abweichend vom Originalsystem) Matlab.

Der FPGA dient als Prototyp für das spätere Design einer anwendungsspezifischen integrierten Schaltung (ASIC). Auf ihm werden die Signalverarbeitungsmodule in VHDL implementiert. Der RAM als gemeinsame Kommunikationsschnittstelle sowohl zwischen den Signalverarbeitungsmodule als auch nach außen zu dem Mikrocontroller befindet sich ebenfalls auf dem FPGA [1]. Die Entwicklung und Simulation erfolgt im Xilinx Vivado IDE.

Der PC dient der Konzeptuntersuchung für die Verarbeitungsmodule und ihrer Verifikation. Dafür muss das PC in der Lage sein, mit dem FPGA Daten auszutauschen und diesen zu steuern. Abweichend vom Originalsystem wurde dafür Matlab (Version 2016b) eingesetzt, weil es wesentlich mehr Funktionalität als Octave im Bereich der digitalen Signalverbreitung und Fixkommarechnungen anbietet.

Die Rolle des Vermittlers zwischen dem PC und dem FPGA übernimmt der Mikrocontroller: die vom PC über seriellen UART (physikalisch USB) empfangenen Befehle und Daten werden in das speziell entwickelte parallele GPIO[1] Interface (und zurück) übersetzt, mit dem der Mikrocontroller mit dem FPGA kommuniziert und das physikalisch mit zwei Flachbandverbindungen zwischen den Pinleisten X8 und X6 auf der Evaluationsplatine und PMods A,

[1] General Purpose Input/Output

B, C, D an dem Zedboard realisiert ist. Dies schließt den Takt für den FPGA ein, der auch vom Mikrocontroller generiert wird.

Dieser Ansatz wurde insgesamt übernommen, jedoch überarbeitet und vereinfacht. Das Blockdiagramm des überarbeiteten Systems ist auf der Abbildung 2.1 dargestellt. Die Änderungen sind in den darauffolgenden Unterkapiteln vorgestellt.

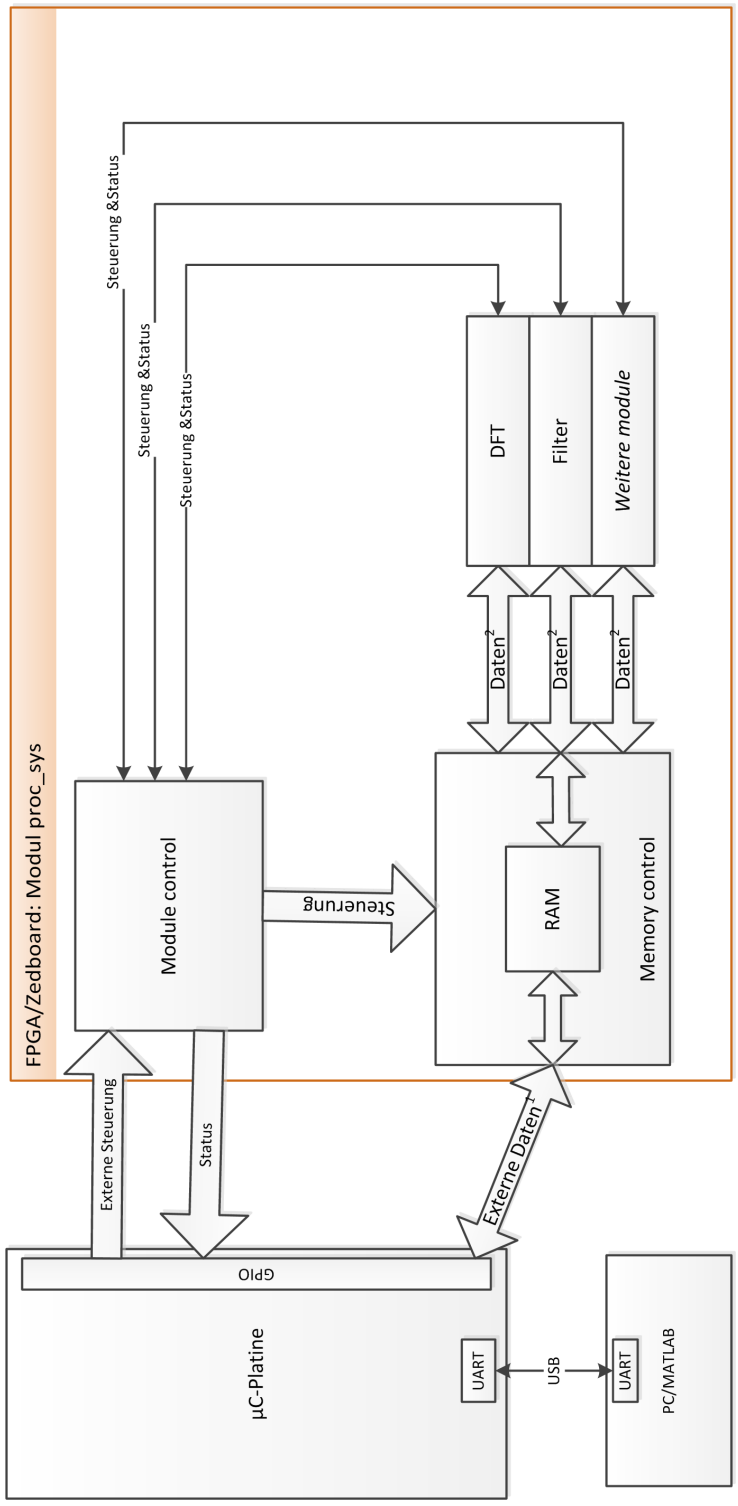

Abbildung 2.1: Das Blockdiagramm des überarbeiteten Testsystems.
1: Wird auch für die RAM-Adresse benutzt.
2: Schließt auch die Adresse und Kontrollsignale ein.

2.2 Zedboard (VHDL)

Für bessere Lesbarkeit der Code wurde ein einheitlicher Codierungsstil eingesetzt. Alle Signale werden großgeschrieben, die Namen von Entitys, Typen und Prozessen werden kleingeschrieben.

Das System ist im Modul `proc_sys` wie in Abbildung 2.1 zusammen verbunden. Seine Eingangs- und Ausgangssignale bilden das externe Interface, das in der Constraints-Datei an die PMod A-D angeschlossen sind. Die externen Anschlüsse sind in Tabelle 2.1 und die Belegung in Tabellen C.1 und C.2 dargestellt. Fast alle externen Eingänge sind in jeweiligen Modulen mittels Eingangsregister mit dem Takt synchronisiert.

Tabelle 2.1: Die externen Signale von Zedboard (Modul `proc_sys`).

Signal	**Richtung**	**Breite in Bit**	**Zweck**
`CLK`	Eingang	1	Das externe Taktsignal für das ganze System.
`EXT_RESET`	Eingang	1	Das externe Reset, der nur auf die Verarbeitungsmodule wirkt, da die Modul- und RAM-Steuerung sowieso direkt extern gesteuert werden können. Für den Power-On-Reset wurden anderseits die Anfangswerte für alle Register definiert.
`EXT_CTRL_EN`	Eingang	1	Aktiviert die externe Steurung vom Testsystem. Es ist der einzige Eingang, der nicht synchronisiert ist. S. Abschnitt 2.2.2.
`EXT_INSTR`	Eingang	5	Der externe Befehlsbus. S. Tabelle 2.3 und Tabelle 2.2

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle 2.1

Signal	Richtung	Breite in Bit	Zweck
EXT_DIN	Eingang	10	Der externe Dateneingang mit variabler Funktion. S. Tabelle 2.2
EXT_DOUT	Ausgang	8	Der externe Datenausgang. S. Tabelle 2.2
MOD_OUT	Ausgang	3	Nummer des aktuell aktiven Moduls. Nur gültig, falls ein Modul tatsächlich aktiv ist.
READY_OUT	Ausgang	8	Nach außen geführtes Ready-Signal des aktiven Moduls.

Zum Zweck des Tests vom System selbe wurden zwei einfache Platzhaltermodule entwickelt und anstelle von DFT und Filter angebunden. Das erste Modul befüllt das RAM im Bereich 0-9 mit Werten. Beim Realteil wird von 0 hochgezählt und beim Imaginärteil wird von 1023 (hexadezimal 0x3FF) heruntergezählt.

Das zweite Modul kopiert die Werte aus dem Bereich 0-9 in den Bereich 991-1000 in der umgekehrten Reihenfolge. Dies sorgt dafür, dass die Daten sich beim Auslesen in einer definierten Weise unterscheiden. Auf dieser Weise kann man sich überzeugen, dass aus dem richtigen Bereich gelesen wurde. Da diese Module keine weiterführende Verwendung haben, werden sie nicht weiter detailliert dokumentiert.

Der Code wird als die fertign voreingestellte Vivado-Projekte geliefert. Er ist ausführlich kommentiert und mit Hilfe von Doxygen[1] dokumentiert. Folgende Projekte wurden erstellt:

Memory Diente der Entwicklung und Simulation von RAM und Speicherverwaltung. Das Projekt ist synthetisierbar, aber nicht lauffähig.

[1] Ein freies Software-Dokumentationswerkzeug

System_Test Die entwickelten Modulsteuerung und Testmodule wurden hinzugefügt. Es diente der Simulation und der Verifikation des Testsystems selbst.

Processing_System Diente der Entwicklung und Verifikation der 2D-DFT und -Filtermodule.

2.2.1 RAM und Zugriffssteuerung

Das RAM-Modul stellt den Speicher auf Basis vom Zynq-BRAM für das Verarbeitungssystem zur Verfügung. Dabei soll es das Verhalten des SRAM der Firma AMS AG für die spätere Hardware-Realisierung reproduzieren. Dies bedingt die Besonderheiten der VHDL-Implementierung. Es werden also alle Signale und deren Funktionen gemäß dem Datenblatt vom AMS-SRAM übernommen. So hat das Modul z. B. die getrennten Eingänge für Schreib- und Leseaktivierung, obwohl dies für das Zynq BRAM unnötig wäre. Das Enable-Signal `EN` hat eigentlich die Funktion der Ausgangabschaltung: bei “high” wird der Datenausgang hochohmig. Dagegen `NRESET` deaktiviert die Steuerung vom RAM. Initialisierung oder Zurücksetzen werden von AMS-SRAM nicht unterstützt und sind daher nicht implementiert.[2]

Die Größe (1024 Adressen) und die Breite (32 Bit) sind aus der Vorgängerarbeit [1] übernommen. Abweichend davon werden für den Realteil die Bits 27-16 und für den Imaginärteil die Bits 11-0 verwendet (Abbildung 2.2). Die restlichen Bits werden mit dem Vorzeichen gefüllt und die Werte werden somit auf 16 Bit erweitert. Dies hat den Vorteil, dass beim Auslesen sofort die äquivalenten 16-Bit-Werte zur Verfügung stehen. Später hat es sich erwiesen, dass es die ganze 16-Bit Breite unter Umständen benötigt werden könnte.

Die Lese- (Abbildung 2.3) und Schreibvorgänge (Abbildung 2.4) benötigen zwei Takte um garantiert die Setup- und Haltezeiten einzuhalten: im ersten werden die Adresse und die Kontrollsignale gesetzt, beim Übergang zum zweiten werden die Daten eingespeichert bzw. stehen zur Verfügung. Alle Operationen, wie unter anderem in [8] vorgeschlagen, werden ausschließlich auf der steigenden Flanke durchgeführt. Ferner soll es vermieden werden, dass `WE` und `RE` bei gleicher steigender Flanke gesetzt sind, weil das Verhalten nicht definiert ist.

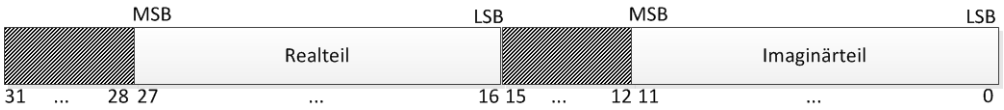

Abbildung 2.2: Organisation einer RAM-Zelle.

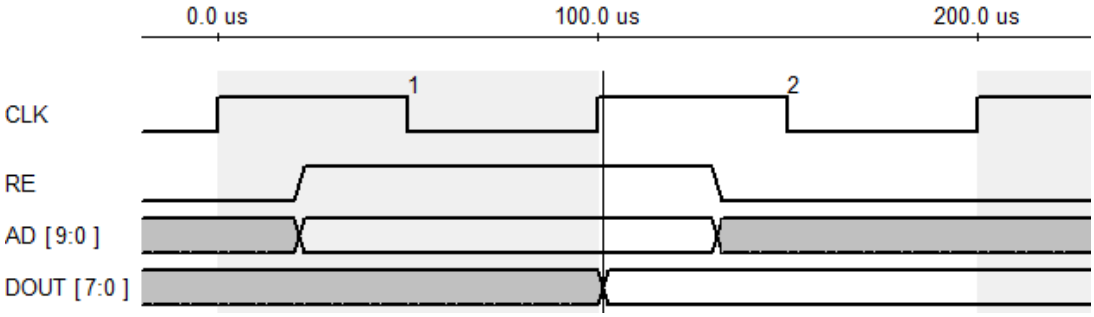

Abbildung 2.3: Zeitdiagramm des Lesevorgangs vom RAM. Die Verzögerungen relativ zum Takt sind absichtlich übertrieben.

Wie schon in 2.1 erwähnt, dient das RAM als Schnittstelle sowohl zwischen den Signalverarbeitungsmodulen als auch extern, die alle den Zugang zum RAM benötigen. Da das RAM nur einen Anschluss hat, wird der Zugriff im Modul `memory_access_ctrl` multiplexiert, statt die Module (wie in der Vorversion) mittels Tristate-Busse vom RAM Seine Funktion ist vergleichbar mit einem Weichensteller, der nach der Anweisung vom Fahrdienstleiter (Modul `module_controller`, 2.2.2), die “Weichen” (Signale) in Richtung RAM um-

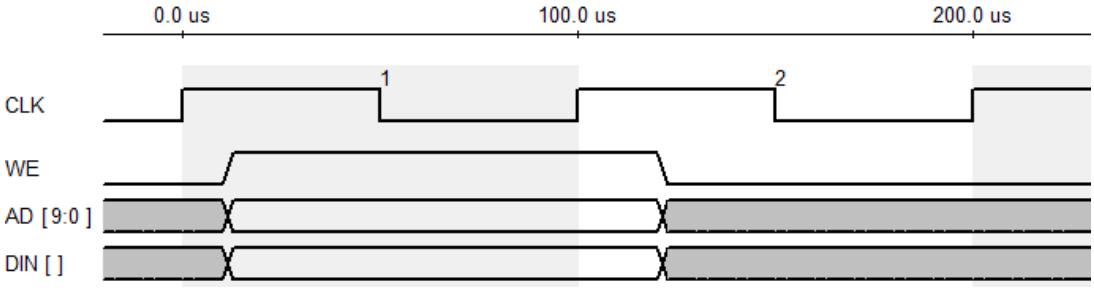

Abbildung 2.4: Zeitdiagramm des Schreibevorgangs von RAM. Die Verzögerungen relativ zum Takt sind absichtlich übertrieben.

schaltet. Diese Änderung wurde auch in der Arbeit von M. Willimczik vorgeschlagen, der dafür eine ausführliche Begründung angibt.[8]

In Richtung Verarbeitungsmodule braucht man nur eine Verzweigung für Datenausgang des RAM. Es ist unbedenklich, wenn auch inaktive daran angeschlossen sind. Anderseits stellt `memory_access_ctrl` ein modulspezifisches Ready-Signal zur Verfügung, das hochgesetzt wird, wenn das RAM auf das Verarbeitsmodul umgeschaltet ist. So kann dieses Modul sich überzeugen, das es über den RAM-Zugang verfügt.

Die RAM-Signale zu und von Verarbeitungsmodulen wurden in jeweils `record`-Typen `from_mem` und `to_mem` in `interface_pkg` zusammengefasst. Dies hat mehrere Vorteile:

- Gewährleistet einheitliches Interface
- Überschaubarere `entity`-Deklarationen
- Können erweitert werden, ohne die Schnittstellen aufzublähen. Diese Besonderheit wurde eingesetzt, um die 12-Bit Real- und Imaginärteile separat zu leiten.
- Die Code des RAM-Controllers kann einfacher gestaltet werden da die `record`-Signale statt einzelnen Signalen als ganzes umgeschaltet werden können.

Die ausführliche Beschreibung befindet sich in der Doxygen-Dokumentation.

Das Modul wurde als ein Mealy-Automat implementiert (Abbildung A.1). Dies reduziert die Anzahl der benötigten Zustände gegenüber einem Moore-Automaten. Die zum Teil direkte Steuerung der Ausgangssignale ist dabei unproblematisch, weil alle Eingangssignale, auch externe, mit dem Takt synchronisiert sind. Die Implementierung wurde auf drei Prozesse aufgeteilt, weil dies erlaubt, die Zustands- und Ausgangslogik unabhängig voneinander zu gestalten, primär um priorisierte Zustandsübergänge zu realisieren. Die externen RAM-Adresse und Daten werden über denselben 10-Bit-Bus übertragen. Eine RAM-Zelle an einer Adresse kann jedoch nur vollständig beschrieben werden. Daher müssen die Adresse und die empfangenen Bytes in einem Zwischenregister gespeichert werden, bevor sie endgültig eingespeichert werden können. Dies wird im dedizierten Zustand `EXT_WRITE` getan. Falls weitere Module dazukommen, muss `memory_access_ctrl` innerhalb des Zustandes `INT_ACCESS` angepasst werden.

Das Modul wird exklusiv durch `module_controller` über ein 5-Bit-Befehlsbus gesteuert. Diese Breite ist durch die Anzahl der benötigten Befehle veranlasst. Die gültigen Befehle sind in Tabelle 2.2 aufgelistet.

Tabelle 2.2: Die Befehle für das `memory_access_ctrl`

Befehlscode	Funktion
`00XXX`	Schaltet die RAM-Steuerung und den Eingang auf eines der Verarbeitungsmodule, drei LSBs sind hierfür reserviert
`010XX`	Das Byte `XX` wird auf den Ausgangsbus eingestellt. Dabei entspricht `00` dem MSB und `11` dem LSB.
`100XX`	Der Wert auf dem Eingangsbus wird als Byte `XX` interpretiert und im Zwischenregister gespeichert. Dabei entspricht `00` dem MSB und `11` dem LSB.
`10100`	Der Wert auf dem Eingangsbus wird als Adresse interpretiert und im Zwischenregister gespeichert.
`11XXX`	Die Daten im Zwischenregister werden unter der im Zwischenregister gespeicherten Adresse eingespeichert.

2.2.2 Modul-Steuerung

Der `module_controller` steuert den Verarbeitungsablauf über ein einheitliches Interface und besteht aus:

`EN` (Enable) schaltet die Register ab. Damit wird der Zustand des Moduls effektiv eingefroren.

`START` weist das Verarbeitungsmodul an, mit seiner Aufgabe zu beginnen.

`READY` wird von dem Verarbeitungsmodul auf high gesetzt, wenn es seine Aufgabe erledigt hat.

Sobald ein Verarbeitungsmodul mit seinem Schritt fertig ist, setzt es sein READY-Signal für einen Takt hoch. Der Modulkontroller aktiviert dann das nächste Modul mit einem Startsignal und stellt den `memory_access_ctrl` ein.

Das Modul und somit auch das ganze Verarbeitungssytem kann über den 5-Bit Befehlsbus extern gesteuert werden, wenn `EXT_CTRL_EN` auf high gesetzt

wird. Die externen Befehle (Tabelle 2.3) haben absichtlich ähnliche Funktion wie in Tabelle 2.2. Die RAM-Befehle (beginnende mit allen anderen als `00`) werden mit Priorität ausgeführt. Das heißt, `modul_controller` wechselt gleich in den Zustand `EXT_CTRL` und die Verarbeitungsmodule werden dabei suspendiert. Dies erlaubt die Daten extern auszulesen, ohne die Module zu beeinflussen und dann ihre Arbeit fortsetzen. Dagegen wird bei den mit `00` beginnenden Befehlen abgewartet, bis das aktive Modul seine Aufgabe erledigt hat, bevor es in den Zustand `EXT_CTRL` (zurück) zu wechselt.

Tabelle 2.3: Die Befehle für das `modul_controller`

Befehlscode	Funktion
`00`XXX	Aktiviert das Modul XXX und schaltet das RAM auf es um. Wenn `EXT_CTRL_EN` dabei auf low gesetzt wird, wird das vom externen Steuerung zur internen übergehen. Der Ablauf wird intern gesteuert startend mit dem Modul XXX.
Alle andere	Erlaubt externe Steuerung von RAM gemäß Tabelle 2.2. Die Verarbeitungsmodule werden mittels Enable-Signalen suspendiert.

Das Modul ist einfach genug um als ein *Moore*-Automat (s. Abbildung A.8) in drei Prozessen implementiert zu werden. Dies erlaubt ausnahmsweise auf Synchronisierung des Eingangs `EXT_CTRL_EN` zu verzichten, da seine mögliche zwischenzeitliche Änderungen keinen externen Effekt haben. Dabei werden jedem Verarbeitungsmodul einen Zustand zugewiesen. Bei Ergänzung mit zusätzlichen Modulen, müssen dann die Zustände und Übergangslogik entsprechend erweitert werden.

2.3 Mikrocontroller und Matlab

2.3.1 Mikrocontroller

Der Mikrocontroller auf der EK-TM4C1294XL Platine spielt die Rolle eines Vermittlers zwischen dem PC und dem Zedboard einschließlich eines SerDes. Er empfängt so die Befehle und Daten vom PC über UART und leitet sie weiter an Zedboard über die GPIO Pins. Auf Anweisung holt er die Daten vom Zedboard

ab, und schickt an PC zurück. Die Software wurde gegenüber Vorgängerversion [1] erheblich vereinfacht und verbessert.

Die GPIO Pins des Mikrocontrollers sind in mehrere Pinblöcke je 8 Pins aufgeteilt [4], die aber mehr oder weniger chaotisch auf die für die Steuerung des Zedboards benutzten Pinleisten X6 und X8 der Platine verteilt sind [6, p. 9-13]. Die Hardware erlaubt mehrere Pins in einem Block simultan zu beschreiben oder auszulesen [4], was auch vom Tivaware Treiberbibliothek unterstutzt wird [5]. Um diesen Vorteil zu nutzen, wurde die Pinbelegung optimiert[1], mit dem Ziel, dass es den Pins in einem Pinblock möglichst zusammenhängende Funktion zugewiesen wird, z. B. Bits eines Busses. Dies wurde größtenteils im Zedboard durchgeführt, weil die Signale dort sich frei den externen Anschlüssen (PMods) zuweisen lassen. Allerdings wurden auch 5 Drähte des Flachbandkabels an der Buchsenleiste, die an die Pinleiste X8 angeschlossen ist, auf der entsprechenden Pinbuchse verschoben.

Die Programmstruktur wurde vereinfacht. Die Unterbrechung wird nur für die Generierung des Taktes (10kHz) für Zedboard mittels Bit-Banging benutzt. Die ISR ist für einen stabileren Takt klein und einfach gehalten Die Kernfunktionalität ist in die Funktion `pcCommInterface()` übertragen. Diese Funktion implementiert das im Abschnitt 2.3.2 beschriebene Protokoll auf dem Mikrocontroller. In einer Schleife pollt sie das UART und führt die Befehle vom PC in einem `switch`-Block aus, wobei die Ausführung mit dem FPGA-Takt abgestimmt ist.

Unter anderem speichert diese Funktion die vom PC oder Zedboard geholten Daten in einem lokalen Puffer. Der Puffer ist als ein Feld von `int16_t`, zusammengesetzt abwechselnd aus Realteilen und dazugehörigen Imaginärteilen, so dass einer Zelle im Zedboard-RA zwei konsekutive Werte im Puffer entsprechen. Diese Anordnung hat den Vorteil, dass die Werte unmittelbar zur Verfügung stehen und direkt Byte für Byte empfangen oder gesendet werden können. Es muss jedoch beachtet werden, dass die Parameter einiger Funktionen die ein Bereich des RAMs darstellen sollen, sich stets auf FPGA-RAM und nicht das lokale Puffer beziehen. Die Fehlerprüfung ist durch die im Programm verteilten Debug-Assertionen[2] durchgeführt, die nur in Debug-Konfiguration wirksam sind. Dies ist so angebracht, denn sie fangen die Programmierfehler auf, die im Normalablauf nicht passieren dürfen.

[1] Die geänderte Pinbelegung ist in Tabellen C.1 und C.2 aufgelistet.

[2] Macro `assert` in `assert.h`

Die Code wurde in der Code Composer Studio (CCS) mittels C99 von Texas Instruments entwickelt. Der C99 Standard hat den z. B. Vorteil, dass das Typ `bool` unterstützt wird und auch dass alle Variablen nicht unbedingt am Anfang einer Funktion deklariert werden müssen. Das Programm wird mitsamt der verwendeten Bibliotheken als ein ganzes Projekt zur Verfügung gestellt, das sofort und unabhängig von dem Ausführungspfad lauffähig ist. Bei Bedarf sollten möglichst die bestehenden Projekte angepasst werden. Wird doch ein neues Projekt erwünscht, die das die Code benutzt, enthält Anhang C.2 einige aus Erfahrung gewonnenen Tipps zur Projekteinstellungen

Die Code ist ausführlich kommentiert. Darüber hinaus wird die mit Doxygen erstellte Dokumentation mitgeliefert.

2.3.2 UART-Protokoll

Zum Zweck der Kommunikation zwischen dem Mikrocontroller und dem PC wurde ein einfacher numerischer Protokoll implementiert. Dieser hat gegenüber den sonst verbreiteten Zeichen-Protokollen den Vorteil, dass die aufwändige Formatierung der Zahlen als Zeichenketten und zurück entfällt. Zum zweiten lässt er sich als die effizienten `switch` Strukturen statt Zeichenkettenvergleiche realisieren. Auch der zu übertragenden Datenvolumen verringert sich, da die Zahlen unmittelbar und nicht als die Zeichenketten übertragen werden. Dies hat zudem für das Testsystem den Vorteil, dass Matlab die Daten bitgenau in der Form erhält, wie sie vom FPGA produziert wurden, was einen direkten Vergleich ermöglicht.

Das Protokoll besteht aus einem 8-Bit Befehl und zwei 16-Bit Parameter, deren Bedeutung variabel und abhängig vom Befehl ist. 16-Bit ist bedingt durch den 10-bit Adressierungsbereich des FPGA-RAMs. Darüber hinaus dient der binäre Bestätigungswert `10100010` der Rückmeldung vom Mikrocontroller. Der Wert wurde für seinen asymmetrischen Bitmuster ausgewählt. Er bestätigt die Ausführung des Befehls und optional die Bereitschaft die Sendung oder den Empfang der Daten. Das erlaubt auf die Vollendung des Befehls im Matlab zu warten und so sicherzustellen, dass ein Befehl zu Ende ausgeführt wird, bevor Matlab den nächsten Befehl erteilt.

Der grobe Ablauf ist wie folgt:

1. Matlab auf dem PC sendet einen Befehl und die dazugehörige Parameter über UART an den Mikrocontroller.
2. Der Mikrocontroller bestätigt die Ausführung des Befehls durch die Rücksendung des Bestätigungswertes. Gegebenenfalls signalisiert dies auch die Bereitschaft Daten zu senden oder zu empfangen.
3. Je nach Befehl werden die Daten empfangen oder gesendet.

Die Tabelle 2.4 zählt alle gültige Befehle mit entsprechenden Parametern. Darüber hinaus kann die Funktionalität problemlos erweitert werden.

Tabelle 2.4: Befehle und Parameter des UART-Protokolls

Befehl	**erster Parameter**	**zweiter Parameter**	**Funktion des Bestätigungswertes**
0: Steuerung der Verarbeitungsmodule	0: Schaltet die externe Steuerung ab und aktiviert die interne Steuerung beginnend mit dem Module angegeben im zweiten Parameter.	Nummer des Moduls, worauf der Befehl angewandt werden soll	Der Bestätigungswert wird zurück gesendet, sobald der Befehl ausgeführt wurde.
	1: Aktiviert ein Modul extern.		
	2: Wie 1, aber Mikrocontroller wartet dann, bis der Modul seine Aufgabe erledigt hat, bevor er den Bestätigungswert sendet.		
	3: Mikrocontroller sendet den Bestätigungswert, sobald der aktive Modul seine Aufgabe erledigt. Dies ist primär dafür gedacht, wenn der FPGA auf die interne Steuerung eingestellt ist	Der zweite Parameter wird ignoriert.	
	4: Die Verarbeitungsmodule werden in den Reset-Zustand versetzt. Er wird durch die Parameterwerte 0-2 deaktiviert.		

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle 2.4

Befehl	**erster Parameter**	**zweiter Parameter**	**Funktion des Bestätigungswertes**
1: Lesen aus einem Bereich vom FPGA-RAM	Startadresse des RAM-Bereichs, aus dem gelesen werden soll.	Wie viele RAM-Zellen gelesen werden sollen.	Signalisiert, dass Mikrocontroller bereit ist, die angefragten Daten zu übermitteln.
2: Beschreiben eines Bereichs vom FPGA-RAM	Startadresse des RAM-Bereichs, die beschrieben werden soll.	Wie viele RAM-Zellen beschrieben werden sollen.	Der Mikrocontroller sendet den Bestätigungswert, wenn er bereit ist, die Daten zu empfangen, und nochmal, wenn das RAM beschrieben wurde.
3: Statusabfragen	0: der Mikrocontroller sendet den Bestätigungswert unmittelbar zurück. Dies kann der Identifikation des Mikrocontrollers dienen, oder auch für die Prüfung der Verbindung verwendet werden	Der zweite Parameter wird ignoriert.	Identifiziert den Mikrocontroller
	Zusätzlich zu 0 wird die Nummer des aktuell laufenden Moduls ausgegeben. Dies gilt aber nur wenn ein Modul tatsächlich aktiviert ist.		Signalisiert, dass Mikrocontroller bereit ist, die angefragten Daten zu übermitteln.
	Zusätzlich zu 0 wird der Status des Ready-Signals vom FPGA übermittelt.		

2.3.3 Matlab

Die Matlab-Klasse `ProcSysProtocol` implementiert das im Abschnitt 2.3.2 beschriebene Kommunikationsprotokoll. Unter anderem macht die Implementierung als eine Klasse die Realisierung der gesamten Funktionalität in einer einzigen Datei möglich, statt den sonst in mehreren Dateien verteilten Funktionen. Für eine tiefe Verständnis ist es sehr empfehlenswert die Matlab-Dokumentation zu den Klassen zu studieren. Weiter wird lediglich das Nötigste daraus erwähnt.

Die Implementierung als eine Klasse erlaubt die serielle Verbindung intern einzukapseln und automatisch verwalten. Die standardmäßige Methode zur Funktionsübergabe im Matlab sind aber die *Werte*parameter. Das würde bedeuten, dass die Funktionen die an einem Objekt dieser Klasse operieren, jedes Mal eine Kopie erstellen würden, inklusive des `serial`-Objekts. Dies ist allerdings unerwünscht, da das letzte wiederverwendet werden soll. Für die Klassen besteht es im Matlab jedoch die Möglichkeit die Referenzparameter-Semantik zu realisieren, in dem man die Klasse von der Klasse `handle` ableitet. Dies wurde ausgenutzt, so dass `ProcSysProtocol` von `handle` abgeleitet ist (über die Klasse `hidden_handle`, s. u.).

Ferner sorgt der Destruktor (im Matlab die Funktion `delete`) für die automatische ordnungsgemäße Abwicklung der seriellen Verbindung, die dann selbst in einem Fehlerfall stattfindet. Der Konstruktor führt die Initialisierung des Objektes durch, was für den automatischen Verbindungsaufbau zum Mikrocontroller ausgenutzt wird. Der Nullargumenten-Konstruktor sondiert nämlich die verfügbaren seriellen Ports nacheinander und versucht den Mikrocontroller zu detektieren. Falls dies erfolgt, wird die Verbindung eben aufgebaut und eine Meldung gezeigt. Dieses Verfahren wurde systemabhängig sowohl für Windows als auch für Linux implementiert, so dass es unter den beiden Betriebsystemen funktioniert. Da das Betreiben von den seriellen Ports (und zwar ihre Namen) der einzige essentielle Unterschied zwischen ihnen im Matlab ist, macht das auch die Klasse insgesamt unter den beiden Betriebssystemen funktionsfähig.

Unter Linux trat allerdings eine Komplikation auf, dass Matlab nur die seriellen Ports, die mit `/dev/ttyS` beginnen, akzeptiert. Der Mikrocontroller erscheint typischerweise unter als ein virtueller mit `/dev/ttyACM` beginnender Port und ist somit eigentlich unzulässig. Diese Hindernis wurde dadurch überwunden, dass der Konstruktor unter Linux die symbolischen Links zu `ttyACM`

im `ttyS` Namensbereich erstellt, die dann gegebenenfalls zum Verbindungsaufbau benutzt werden. Da der Ordner `/dev` nur im Arbeitsspeicher existiert und die Linkserstellung dort die Superuser-Rechte erfordert, wird einmal pro Rechnerneustart nach dem Admin-Passwort gefragt. Dem Konstruktor kann auch auf Wunsch die Name eines seriellen Ports übergeben werden, über die Verbindung zum Mikrocontroller aufgebaut wird. Dabei akzeptiert diese Konstruktorversion auch die `ttyACM` Namen und erstellt bei Bedarf das entsprechende symbolische Link.

Die Matlab Funktion `fread`, die die binären Daten aus dem seriellen Port liest, gibt die Werte stets als `double` zurück, gleichgültig in welchem Format sie vom Mikrocontroller übertragen wurden. Dies ist jedoch für die Verifikation der Fixkomma-Algorithmen ungünstig. Die Daten sollten zu diesem Zweck möglichst bitgenau so empfangen, wie sie auf dem FPGA erzeugt wurden. Deswegen enthält die Klasse die modifizierte Kopie von `fread`, die Werte in demselben Datentyp zurückgibt, in dem sie übertragen wurden. Darüber hinaus stellt die Klasse das Paar von Funktionen, die komplexe quadratische Matrizen in ein benutzerdefiniertes Bereich des FPGA-RAMs schreiben beziehungsweise aus einem benutzerdefinierten Bereich des FPGA-RAMs lesen können.

Die Klasse beinhaltet eine ausführliche Dokumentation. Sie kann entweder direkt in der Code angesehen werden oder alternativ mit "`help ProcSysProtocol`" im Matlab "Help-Browser" geöffnet werden. Dieser Browser würde auch die von `handle` vererbten Funktionen auflisten, was die Übersichtlichkeit verschlechtern würde. Deswegen wurde `ProcSysProtocol` nicht direkt von `handle` abgeleitet, sondern über über die Klasse `hidden_handle`[1]. Die letzte trägt nicht zur Funktionalität bei, sondern dient ausschließlich dem Zweck die `handle`-Funktionen zu verdecken. Ausgenommen davon ist die Funktion `isvalid`, weil sie nicht überschrieben werden darf.

Es sollte vermieden werden, die Skripte, die diese Klasse verwenden, mit Strg+C abzubrechen, da dies die Kommunikion zum Mikrocontroller durcheinander bringt. Um so einen Skript zu stoppen, muss man ein Breakpoint im Skript setzen, und dann in der Oberfläche per "Quit debugging" abbrechen. Dies stellt sicher, dass die Funktionen komplett ausgeführt werden, und sorgt somit für einen geordneten Abbruch. Sollte es doch einmal zu einem ungeordneten Abbruch kommen, muss die Software auf den Mikrocontroller neu geladen werden, damit er wieder ansprechbar wird.

[1] Entnommen aus https://stackoverflow.com/a/13050111

3 Implementierung und Verifikation vom zweidimensionalen DFT in VHDL

3.1 Grundlagen

Die zweidimensionale diskrete Fouriertransformation (2D-DFT) stellt den ersten Schritt in der Verarbeitungskette der Sensorarraydaten dar. Damit werden die Ortsdaten der magnetischen Feldstärke in den Ortsfrequenzbereich übertragen.

Die Transformation einer quadratischen N$\times$N Matrix ist definiert [9, 12] als

$$F[k,l] = \frac{1}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f[m,n] \cdot e^{-j2\pi \frac{k}{N} m} \cdot e^{-j2\pi \frac{l}{N} n} \tag{3.1}$$

Der ausgeblendete Vorfaktor $\frac{1}{N^2}$ wird uneinheitlich angewandt, zum einen bei der DFT oder erst später bei der inversen DFT. Alternativ kann der Faktor $\frac{1}{N}$ bei beiden Transformationen angewandt werden [12].

Die Formel kann als 1D-DFT der Reihen, gefolgt von 1D-DFT der Spalten gedeutet werden [10, 12]. Dabei heißt die Größe $e^{-j\frac{2\pi}{N}} = W_N$ *Twiddle-Faktor* [10, 14].

Äquivalent kann die 2D-DFT über die Multiplikation der Matrizen ausgedrückt werden [10, 14, 15]. Dafür wird zunächst die symmetrische N$\times$N

Twiddlefaktor-Matrix definiert, die alle entsprechenden Twiddle-Faktoren beinhaltet [14, 15]. Sie kann in Matlab mit der Funktion `dftmtx` erzeugt werden.

$$\underline{W} = \begin{pmatrix} W_N^0 = 1 & W_N^0 = 1 & W_N^0 = 1 & \dots & W_N^0 = 1 \\ W_N^0 = 1 & W_N^1 & W_N^2 & \dots & W_N^{N-1} \\ W_N^0 = 1 & W_N^2 & W_N^4 & \dots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ W_N^0 = 1 & W_N^{N-1} & W_N^{2(N-1)} & \dots & W_N^{(N-1)^2} \end{pmatrix} \tag{3.2}$$

Sei $\underline{S}$ die komplexe Matrix mit den Messwerten vom Sensorarray, dann ist die 2D-DFT X_{2D} [10]:

$$\underline{X_{2D}} = (\underline{W} \cdot \underline{S}) \cdot \underline{W} \tag{3.3}$$

Es ist erwähnenswert, dass der Ausdruck in Klammern gleich der 1D-DFT der Reihen ist [10, 14].

Die Ergebnis der Transformation X_{2D} ist eine Matrix, worin jedes Element eine Ortsfrequenz vertritt, wobei die hohe Frequenzen in der Mitte liegen. In der digitalen Bildverarbeitung ist es jedoch üblich, die 2D-DFT so zu shiften und darzustellen, dass der Gleichanteil in der Mitte und die hohen Frequenzen sich außen befinden [13]. Dies erfolgt mit der Matlab-Funktion `fftshift`.

3.2 Erarbeitung der Implementierung

3.2.1 Theoretische Erwägungen

Die Methode der Matrizenmultiplikation wurde für diese Arbeit aufgrund der relativ trivialen Implementierung und Verifizierung ausgewählt. Es gibt auch andere Algorithmen, die DFT zu berechnen, wie FFT oder mit Konstantenmultiplizierern, die die günstiger hinsichtlich der in gemessene Taktzyklen Ausführungszeit sind. Ihre Implementierungen sind aber kompliziert und haben darüber hinaus einen großen Flächenbedarf (vgl. [8, 10]). Es könnte sonst passieren, dass die Realisierung selbst zu viel Zeit und Aufwand in Anspruch nimmt, so dass am Ende keine Zeit für eine gründliche Verifikation bleibt. Deswegen werden nur Optimierungen vorgenommen, die zur einfacheren Logik und somit geringeren Flächenbedarf führen.

Die 2D-DFT über die Matrizenmultiplikation wird nun genau betrachtet. Sie besteht aus 2 konsekutiven komplexen Multiplikationen mit der Twiddlefaktor-Matrix, zuerst von links, was die 1D-DFT der Reihen ergibt und dann von rechts. Das Ergebnis der 1D-DFT muss also zuerst gespeichert werden, bevor sie dann nochmal mit der Twiddlefaktor-Matrix ausmultipliziert wird. Es bietet sich als eine Optimierung zur Vereinfachung an, die komplexe Matrixmultiplikation in der allgemeinen Form einmal zu realisieren und für beide Schritte zu verwenden.

Der Wert des Produktes der Matrizen an der Stelle $(k; l)$ ist gleich dem Skalarprodukt der k-ten Reihe der linken Matrix und l-ten Spalte der rechten Matrix (s. Abbildung 3.2). Diese sollen elementweise ausmultipliziert und aufsummiert werden.

Die Matrizen werden in FPGA RAM *Reihe nach Reihe* platziert (Abbildung 3.1)), so dass auch die Ausgangswerte am besten in der selben Reihenfolge berechnet werden sollten (dunkelgelbe Pfeile auf Abbildung 3.2). Es muss auch berücksichtigt werden, dass nur ein Wert gleichzeitig aus der RAM gelesen bzw. in die RAM geschrieben werden kann. Um ein Element des Produktes auszurechnen muss man also die jeweiligen Reihe und Spalte durchgehen (grüner Pfeil auf Abbildung 3.2). Dafür muss der linke Speicheriterator um eins (blaue Pfeile auf Abbildung 3.1) und der rechte Speicheriterator um N inkrementiert werden. Dabei werden die Werte paarweise ausmultipliziert und akkumuliert. Ist ein Wert fertig berechnet, muss zur nächsten rechten Spalte übergangen werden (blauer Pfeil auf Abbildung 3.2). Erst wenn die letzte Spalte erreicht ist, soll die nächste linke Reihe genommen werden (rote Pfeile auf Abbildung 3.2). Zu diesem Zweck braucht man auch noch die Iteratoren, die die Startadressen der aktuellen Reihe (links) bzw. Spalte (rechts) verfolgen würden. Der Spalteniterator soll um eins und der Reiheniterator um N inkrementiert werden (vgl. Abbildung 3.1). Die Berechnung ist abgeschlossen, wenn die letzte Reihe links und die letzte Spalte rechts erreicht wurden.

Dieser Algorithmus ist in Abbildung 3.3 dargestellt. Er soll zweimal ausgeführt werden. Beim ersten Durchlauf ist die Twiddlefaktor-Matrix links und rechts sind die Eingangsdaten (Sensormatrix), wobei das Ergebnis, wie schon erwähnt, die 1D-DFT der Reihen ist. Sie soll zu Verifikationszwecken in einem eigenen RAM-Bereich gespeichert werden. Beim zweiten Durchlauf ist die Twiddlefaktor-Matrix *rechts* und links ist die zuvor berechnete 1D-DFT der Reihen, was schließlich die 2D-DFT ergibt.

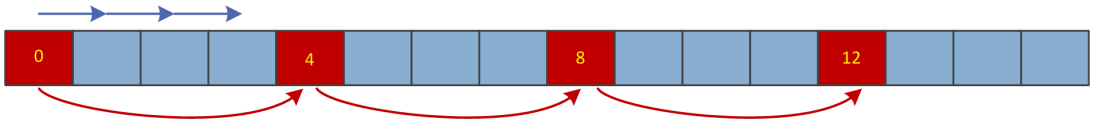

Abbildung 3.1: Veranschaulichte Platzierung einer Matrix in RAM, beispielhaft für 4×4. Rote Zellen markieren die Reihenanfänge, gelbe Zahlen den Abstand zum ersten Element der Matrix.

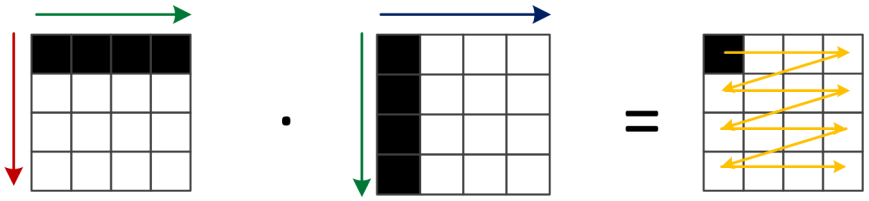

Abbildung 3.2: Veranschaulichung der Multiplikation von Matrizen. Beispiel für 4×4.

Bevor die 2D-DFT mit Fixkomma in VHDL realisiert wird, müssen das Fixkommaformat jeglicher am Berechnung beteiligten Zahlen (einschließlich RAM für Ein- und Ausgangsdaten) bestimmt werden. Deswegen wurde das hier präsentierte Konzept zuerst in Matlab so nah an die spätere VHDL-Realisierung implementiert, um ihre numerischen Eigenschaften zu untersuchen. Außerdem dient sie sozusagen als ein Prototyp, das gestattet, das Konzept selbst z. B. auf korrekte Adressierung zu überprüfen. Sie erlaubt dazu noch einen direkten Vergleich mit der Funktion `fft` bzw. `fft2`.

Konkret sollen die folgenden Fragen beantwortet werden:

- Wie groß können die Werte bei Berechnungen zahlenmäßig werden? Dies schließt jegliche Zwischenergebnisse ein und definiert die Anforderung an die Breite des Ganzteils der Fixkommazahlen.
- Wie breit braucht der Bruchteil jeglicher Fixkommazahlen zu sein, um den Genauigkeitsanforderungen zu genügen?

Alle Versuche sollen bezüglich der 8×8 DFT und Daten durchgeführt werden, weil es der Realisierung des Sensorarrays entspricht. Als das Maß für die Genauigkeit wurde als erstes die höchste zahlenmäßige Abweichung des Real- oder Imaginärteils von `fft` bzw. `fft2` angenommen. Schon nach den ersten Versuchen kam die Anforderung an die Abweichung der Argumente der komplexen Zahlen (ebenfalls mit Hilfe von `fft` bzw. `fft2` berechnete Werte) um

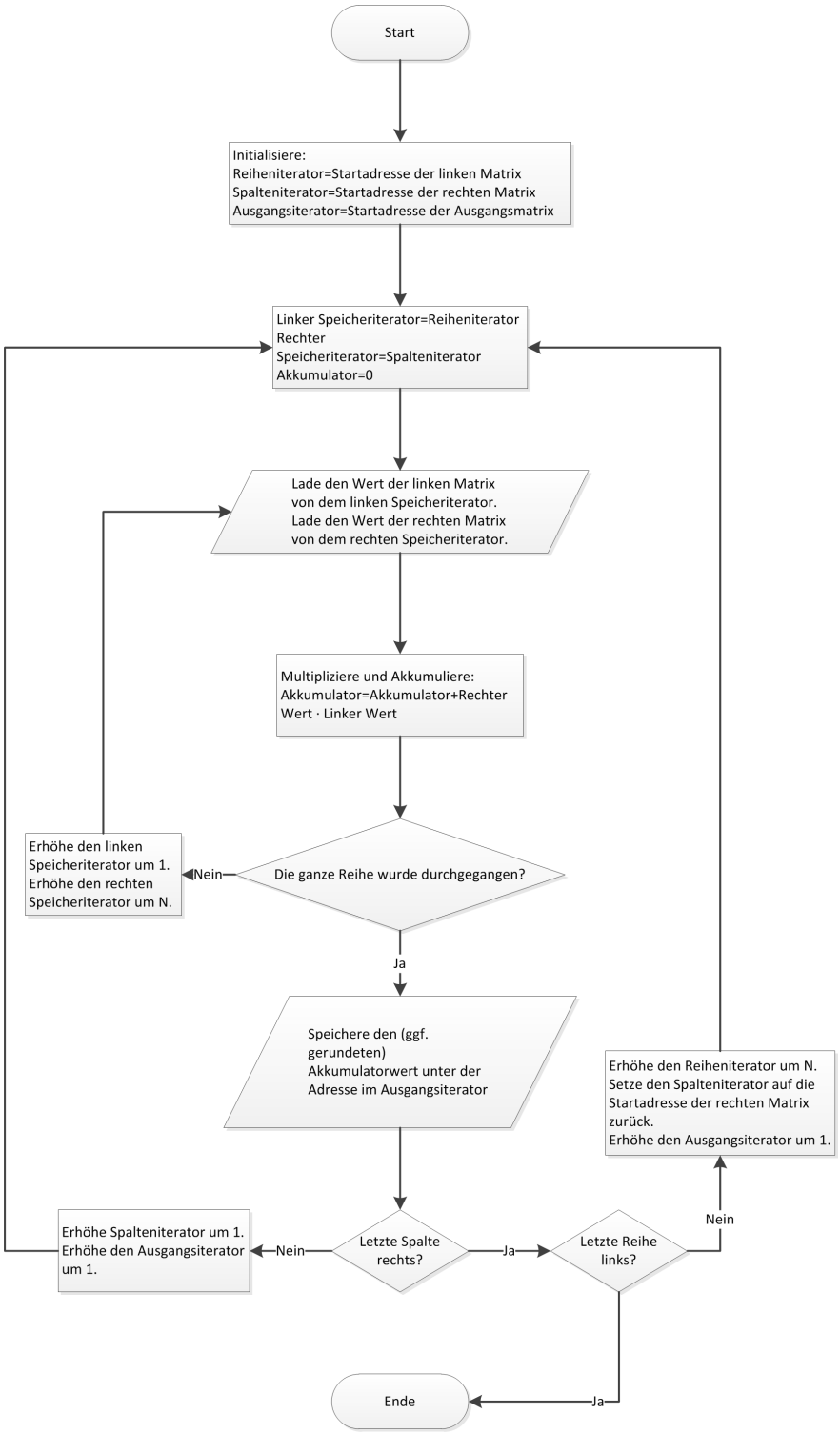

Abbildung 3.3: Algorithmus für die Multiplikation der Matrizen, die sich eindimensional im RAM befinden.

höchstens 0,5° hinzu, weil dies der gewünschten Genauigkeit eines Winkelsensors entspricht.

3.2.2 Ermittlung der maximalen Zahlengröße der Berechnungen

Um die größte auftretende Zahl in Berechnungen herauszufinden, wurde der Algorithmus zuerst in im Skript `determine_DFT_limits_floating.m` mit *Fließkomma* implementiert, da dies praktisch beliebige Zahlgrößen zulässt. Dafür muss der Algorithmus zumindest in Grundzügen entwickelt und getestet werden. Dies geschieht in den Funktionen innerhalb des Skriptes, da diese keine weitere Verwendung haben. Dort werden die Matrizen reihenweise in die Vektoren umgeformt und jeder Schritt der Matrizenmultiplikation, inklusive die Multiplikation zweier komplexen Zahlen, wird explizit ausgeschrieben ohne auf die fortgeschrittene Funktionalität von Matlab zurückgreifen. Bei jedem Berechnungsschritt wird mittels der Funktion `maxCompVal` geprüft, ob die ergebene Zahl (ohne Berücksichtigung des Vorzeichens) bisher die größte ist, die dann gespeichert wird.

Für die Testberechnungen der DFT werden Eingangsdaten benötigt. Zu diesem Zweck wurden die folgenden künstlich erzeugten Datensätze ausgewählt:

1. komplexe 2D Sinus-Welle
2. Alternierend -1 und 1 im Schachbrettmuster.
3. Quadratische Matrix gefüllt mit komplexen Zufallszahlen im Bereich [-2;2], weil erwartet wird, dass die Sensorwerte in diesem Bereich liegen werden. Um den Einfluss des Zufalls zu reduzieren, wurde jeder Versuch mit 50 verschiedenen Matrizen wiederholt.

Für die 2D-DFT wird die 1D-DFT als Zwischenschritt begerechnet. Dies ermöglicht sie auch getrennt zu untersuchen. Die höchsten aufgetretenen Zahlen für die 8×8 DFT sind in Tabelle 3.1 dargestellt.

Wie aus der Tabelle 3.1 zu entnehmen ist, können die 2D-DFT-Werte sehr groß werden. Der Ganzteil benötigt also 7 Bit inklusive des Vorzeichenbits. Dies würde bei der zuvor geplanten 12-Bit Breite nur 5 Nachkommabits erlauben, was die Genauigkeit sehr einschränken würde. Darüber hinaus ist zu

Tabelle 3.1: Höchste aufgetretene Zahl in 1D- und 2D-DFT.

Datensätze	**1D-DFT**	**2D-DFT**
2D-Sinus	4,9809	21,2903
Altern. -1, 1	8,0	64,0
Zufallsmatrizen[1]	11,08	38,3

berücksichtigen, dass das Fixkommaformat für die Eingangsdaten und das Ergebnis der DFT übereinstimmen sollten, weil der Ausgang der 1D-DFT der Eingang für die 2D-DFT ist.

Ein möglicher Ausweg ist die Verwendung der vollen 16-Bit Breite im RAM. Da zu erwarten ist, dass mit größeren DFT die Ausgangswerte noch größer werden, ist diese Lösung nicht skalierbar. Anderseits kann die Unbestimmtheit über den Vorfaktor in der Gleichung 3.1 ausgenutzt werden, indem man die 1D-DFT durch N teilt, bevor daraus die 2D-DFT berechnet wird. Diese Möglichkeit wurde auch untersucht und die Ergebnisse sind in Tabelle 3.2 präsentiert.

Tabelle 3.2: Höchste aufgetretene Zahl in 2D-DFT mit um N skalierten 1D-DFT.

Datensätze	**2D-DFT**
2D-Sinus	2,66
Altern. -1, 1	8,0
Zufallsmatrizen[2]	4,79

Wie man in Tabelle 3.2 erkennen kann, wurden die Ausgangswerte verringert, so dass der Ganzteil mit 4 Bits auskommen würde.

Eine weitere Alternative ist die ganze DFT mit N^2 normieren, was die Vergrößerung der Ausgangswerte gar verhindert. Dies kann auf einfache Weise dadurch erreicht werden, dass man die Twiddlefaktor-Matrix um Faktor N im Voraus reduziert. In diesem Fall würde der Ganzteil gleich breit sein, wie der von den Ausgangsdaten, also 2 Bit.

3.2.3 Untersuchung der Fixkommaimplementierung in Matlab

3.2.3.1 Fixkommarechnung in Matlab und die Klasse `FixedDFT`

Zwecks Bestimmung der Fixkommaformate von Daten und Berechnungen wird nun die 2D-DFT mit Fixkommazahlen in Matlab verwirklicht. Die Fixkommazahlen in Matlab sind die Zahlen, deren Skalierung im Allgemeinfall durch eine beliebige lineare Funktion der binären Bitfolge definiert werden kann.

$$D = F2^E \times I + B, \text{ wo:} \tag{3.4}$$

D ist der Dezimalwert der Fixkommazahl,

F ist der Steigungsanpassungsfaktor,

E ist ein ganzzahliger Exponent,

I ist die Bitfolge der Fixkommazahl, die als eine herkömmliche Ganzzahl gedeutet wird,

B ist ein Offset von Null.

In dieser Arbeit kommt jedoch lediglich die reine *Binärkomma-Skalierung* zur Anwendung, bei der die Steigung allein durch die Position des gedachten Kommas in der Bitfolge bestimmt wird. Das heißt $F = 1$, $B = 0$ und $E = -m$, m ist die Anzahl der Nachkommabits (äquivalente Bezeichnung: Breite des Bruchteils). Dabei stellt die Größe 2^{-m} den Wert eines LSBs dar. Das Komma kann auch außerhalb der Zahl liegen. Interessant ist der Fall, wenn das Komma links der Bitfolge liegt, da damit die Zahlen, deren Wertebereich unterhalb von Null liegt, sich darstellen lassen, ohne die Fixkommazahl verbreitern zu müssen.

In dieser Arbeit werden ausschließlich die vorzeichenbehafteten Fixkomazahlen verwendet. Das Vorzeichenbit wird stets der Einfachheit halber dem Ganzteil zugerechnet. Ferner wird eine verkürzte Angabe der Fixkommaformate angewandt:

Gesamtbreite=Breite des Ganzteils + Breite des Bruchteils

In Matlab werden die Fixkommazahlen als die Objekte der Klasse `fi` aus dem *Fixed-Point Designer Toolbox* repräsentiert. Ihr Format wird in `numerictype`-Objekten aufbewahrt. Angenommen die Binärkomma-Skalierung wird durch die drei Parameter eingestellt: die Anwesenheit des Vorzeichens, die Gesamtbreite der Zahl, die Breite des Bruchteils. Die Breite des Bruchteils darf dabei etwas verwirrend gleich oder größer als die Gesamtbreite sein, was das Komma links außerhalb der Zahl setzt. Des Weiteren kann ein `fimath`-Objekt mit einem `fi`-Objekt assoziiert werden, mit dem man beeinflussen kann, wie die Fixkommaberechnungen durchgeführt werden. So kann man so auswählen, ob die Fixkommazahlen bei arithmetischen Operationen überlaufen oder gesättigt werden.

Matlab ist in der Lage, die Fixkommazahlen in verschiedenen numerischen Formaten zurückzugeben. Am wichtigsten davon sind der dezimale Fließkommawert, die interne Darstellung als ein eingebauter Integertyp und als eine binäre Zeichenkette. Es ist auch andersherum möglich, die Werte den `fi`-Objekten in genannten Formaten zuzuweisen. Matlab aktualisiert dann automatisch die anderen Formate. Dies ist sehr nützlich, da es erlaubt, die Daten ohne zusätzliche Umwandlung mit dem FPGA auszutauschen. Für weitere Informationen über Fixkommarechnung empfiehlt sich, die Matlab-Dokumentation zu lesen.

Die Untersuchung verschiedener Implementierungsoptionen der 2D-DFT für das FPGA inklusive jeglicher Bitbreiten benötigt ein sehr flexibles Prototyp für Untersuchungen in Matlab. Dies wurde in der Klasse `FixedDFT` realisiert, die sämtliche Optionen für die Kalkulation speichern kann. Es ermöglicht, diese Optionen einmal bei der Konstruktion des Objektes einzustellen und dann bei Berechnungen automatisch anzuwenden. Folgende Einstellungen können mit dem Konstruktor vorgenommen werden:

- Dimension der DFT
- Fixkommaformat von Twiddle-Faktoren
- Format von Akkumulatoren (indirekt, wird aus Parametern errechnet (s. u.))
- Format der Ausgangsdaten
- Die Weise, wie der komplexe Produkt ausgerechnet wird
- Skalierung der DFT (optionale Parameter). Es werden folgende Parameterwerte unterstützt (vgl. Abschnitt 3.2.2):

0. Keine Skalierung (default)
1. Nur die Ausgabe der 1D-DFT wird herunterskaliert
2. Die Twiddefaktor-Matrix und damit die DFT insgesamt wird herunterskaliert.

Im Konstruktor wird zunächst die Twiddlefaktor-Matrix erzeugt, gegebenenfalls durch die Dimension geteilt und dann in das angegebene Fixkommaformat konvertiert und gespeichert. Daraufhin wird das Format des Akkumulators berechnet. Das Fixkommaformat des Produktes, also die Breiten des Ganz- und Bruchteils, ergibt sich aus der Summe der jeweiligen Breiten der Faktoren. Allerdings könnte sein, dass die daraus ergebene Anzahl der Nachkommabits, die in Hardware den Flip-Flops entsprechen, überschüssig ist. Der Konstruktor bietet die Möglichkeit an, die Breite des Akkumulatorbruchteils per Parameter zu verringern, um deren Einfluss auf das Ergebnis zu untersuchen. Danach wird das Format der Zwischenergebnissen der arithmetischen Operationen mittels `fimath` fest auf das Akkumulatorformat gesetzt, weil es so der Harsware-Implementierung entspricht. Schließlich wird das letztendlich verwendete Format des Akkumulators in der Konsole ausgegeben.

Die eigentlichen 1D- und 2D-DFT werden jeweils mit Funktionen `dft1D` und `dft2D` berechnet. Neben der Ergebnissen geben die Funktionen die höchsten zahlenmäßigen Abweichungen sowie die Abweichungen des Argumentes von jeweils `fft` und `fft2` zurück. Dabei wird die mögliche Skalierung berücksichtigt.

Die Klasse ermöglicht den Zeitaufwand auf Hardware in Taktzyklen abzuschätzen. Dafür hat die Klasse einen Attribut, der als Taktzyklenzähler verwendet wird. Für jeden Berechnungsschritt wird der Aufwand in Taktzyklen eingeschätzt und der Zähler entsprechend hochgesetzt. Um den Zählerwert zwischen Berechnungsschritten zu erhalten, wurde die Klasse von `handle` abgeleitet (vgl. Abschnitt 2.3.3). Auf dieser Weise wurde der Aufwand von ca. 2300 Taktzyklen prognostiziert.

Die Klasse wurde kontinuierlich nach Bedarf modifiziert und erweitert, so dass die oben beschriebene Funktionalität nicht auf einmal realisiert wurde. Es wurden z. B. zahlreiche statische Hilfsfunktionen für die Auswertung der Berechnungsergebnisse hinzugefügt.

3.2.3.2 Durchführung und Auswertung der Versuche in Matlab

Matlab stellt eine sehr nützliche Option zur Verfügung mittels

```
fipref('LoggingMode','On');
```

die Warnungen über Über- und Unterläufe in den Fixkommaberechnungen anzuschalten. Allerdings wurde dadurch die Konsole mit den Unterlaufwarnungen überflutet. Da die Unterläufe so gut wie unvermeidbar und nicht so wichtig sind, wurden sie mit

```
warning off fixed:fi:underflow
```

unterdrückt. Dagegen sind die Überlaufwarnungen von großer Bedeutung, da die Überläufe schwer zu ermittelnde Fehler verursachen können. Sie wurden in allen Testskripten aktiviert.

Als Testdaten wurden die im Abschnitt 3.2.2 aufgezählten Datensätze eingesetzt. Später wurde auch das mittelwertfreie Viertel eines 2D-Sinus hinzuaddiert, weil es mehr der erwarteten Form des Magnetfeldes entspricht. Die Datensätze werden in Fließkomma erzeugt und zunächst ins 12-Bit Festkommaformat übertragen, bevor sie in das endgültige Datenformat gebracht werden. Dies soll den voraussichtlichen Vorgang in Hardware folgen, wenn der vom Sensorarray kommende 12-Bit Datenstrom in das Format des RAM umgewandelt wird.

Als erstes wurde der Einfluss des Formats von Twiddlefaktoren untersucht mit dem Ziel das passende Fixkommaformat auszuwählen. Die Zahlenwerte in der unskalierten Twiddlefaktor-Matrix liegen im Bereich [-1;1], wofür die Breite von 1 Bit für den Ganzteil *fast* ausreichen würde. Allerdings könnte so die +1 nicht genau dargestellt werden. Es ist nun die Frage, wie man damit umgeht. Da sind die folgenden Optionen denkbar:

1. 2 Bits für den Ganzteil spendieren. Dafür müsste der Bruchteil verkleinert werden und somit womöglich die Genauigkeit reduziert werden. Darüber hinaus wird das zusätzliche Bit größtenteils verschwendet.
2. Die Eins einfach sättigen, also durch den größten darstellbaren Wert ersetzen. Dies würde jedoch zu Genauigkeitsverlust führen. Besonders in de ersten Reihe und Spalte, die komplett aus 1 bestehen, würde der Fehler akkumuliert.

3. Wie 2, aber der Fehler wird damit umgangen, dass der größte darstellbare Wert wie 1 behandelt. Das heißt, bei der Multiplikation wird einfach der andere Faktor dem Produkt zugewiesen. Dies würde jedoch die Berechnung und auch die Logik auf dem FPGA verkomplizieren. Diese Funktionalität wurde extra in die Klasse `FixedDFT` eingebaut.

Diese Optionen werden im Skript `twiddle_prod_investigation.m` mit 12-Bit Twiddlefaktoren geprüft. Bei den letzten 2 Optionen gibt Matlab zahlreiche Überlaufwarnungen aus, die hier jedoch erwartungsgemäß sind. Dazu wurden noch 2 Fälle angeschaut:

- 2 Bit Ganzzteil mit 11 Bit, Bruchteil um die Genauigkeit mit Optionen 2 und 3 zu vergleichen.
- 2 Bit Ganzzteil mit 8 Bit Bruchteil um die Auswirkung zu erproben.

Es wird die höchste numerische Differenz zu der von der `fft2` berechneten DFT angegeben. Als Testeingangsdaten wurde der 2D-Sinus mit 12-Bit Breite benutzt. Die Ergebnisse sind in der Tabelle 3.3 aufgeführt. Wie man in Tabelle 3.3 erkennen kann, führt die Option 2 zu einer bemerkbaren Verschlechterung, während der Workaround aus der Option 3 gewirkt hat. Noch auffälliger ist, dass die Verkleinerung des Bruchteils um 2 Bits nicht zum schlechteren Ergebnis geführt hat. Weitere Untersuchungen haben bestätigt, dass die Rechengenauigkeit nur geringfügig von Twiddle-Faktoren abhängt. Sie ist vielmehr durch die Anzahl der Nachkommabits der Ausgangsdaten begrenzt. Beispielhaft sind die Ergebnisse des in der Tabelle 3.4 dargestellten vorläufigen Tests.Da kein Genauigkeitsverlust mehr befürchtet werden muss, wurde die Option 1 für alle weiteren Versuche und Implementierung festgelegt.

Tabelle 3.3: Testergebnisse des Fixkommaformats der Twiddlefaktor-Matrix

Format der Twiddlefaktor-Matrix, Gesamtbreite=Ganzteil+Bruchteil	**höchste Differenz**
12=2+10	0,0078
12=1+11 (gesättigt)	0,0105
12=1+11 (gesättigt, mit Workaround)	0,0078
13=2+11	0,0078
10=2+8	0,0078

Tabelle 3.4: Ergebnisse vom vorläufigen Test

	Ohne Skalierung		**Mit skalierter 1D-DFT**		
Format	7+5=12 Bit	7+9=16 Bit	4+8=12 Bit	4+10=14 Bit	4+12=16 Bit
2D-Sinus	0,065102	0,000692	0,008789	0,00293	0,000695
Altern. 1/-1	0,03125	0,00000	0,003906	0,000977	0,000244
Zufallsmatrix	0,088704	0,001545	0,011262	0,002832	0,000898

Es wurde auch untersucht, ob man den Hardwareaufwand dadurch einsparen könnte, wenn man die Breite (des Bruchteils) der Akkumulatoren verringert. Dies wurde anhand des 16 (7+9) Bit Datenformats und 12-Bit Twiddle-Faktoren getestet (Tabelle 3.5). Das Format des nicht reduzierten Akkumulator hat in diesem Fall Format 26=7+19. Aus der Tabelle 3.5 kann man also fest-

Tabelle 3.5: Ergebisse mit reduzierter Breite des Akkumulators.

Akkumulator reduziert um… Bits	**2D-Sinus**	**Zufallsmatrizen**
0	0,003906	0,008077
2	0,003906	0,008077
4	0,003906	0,008077
6	0,003906	0,006785
8	0,003906	0,009028
10	0,009147	0,011995
12	0,071832	0,121094

stellen, dass es durchaus denkbar wäre, die Akkumulatorenbreite um 8 Bit ohne größerer Verlust an Genauigkeit zu verkleinern.

Es wurden noch einige Tests über kleine Implementationseinzelheiten durchgeführt. Sie sind von nachrangiger Bedeutung und werden hier nicht beschrieben um die Dokumentation nicht unverhältnismäßig aufzublähen. Unter diesen befinden sich die Tests von 1D-DFT, die vornehmlich der Überprüfung der Matlab-Implementierung selbst dienen, aber wenig interessant sind für die im aktuellen Abschnitt zu behandelnden Fragen.

Zum Schluss wurden 3 große Versuche der jeweils 3 im Abschnitt 3.2.2 erwähnten Implementierungsvarianten durchgeführt:

1. Unskalierte 2D-DFT mit dem 7 Bit Ganzteil der Ein- und Ausgangsdaten

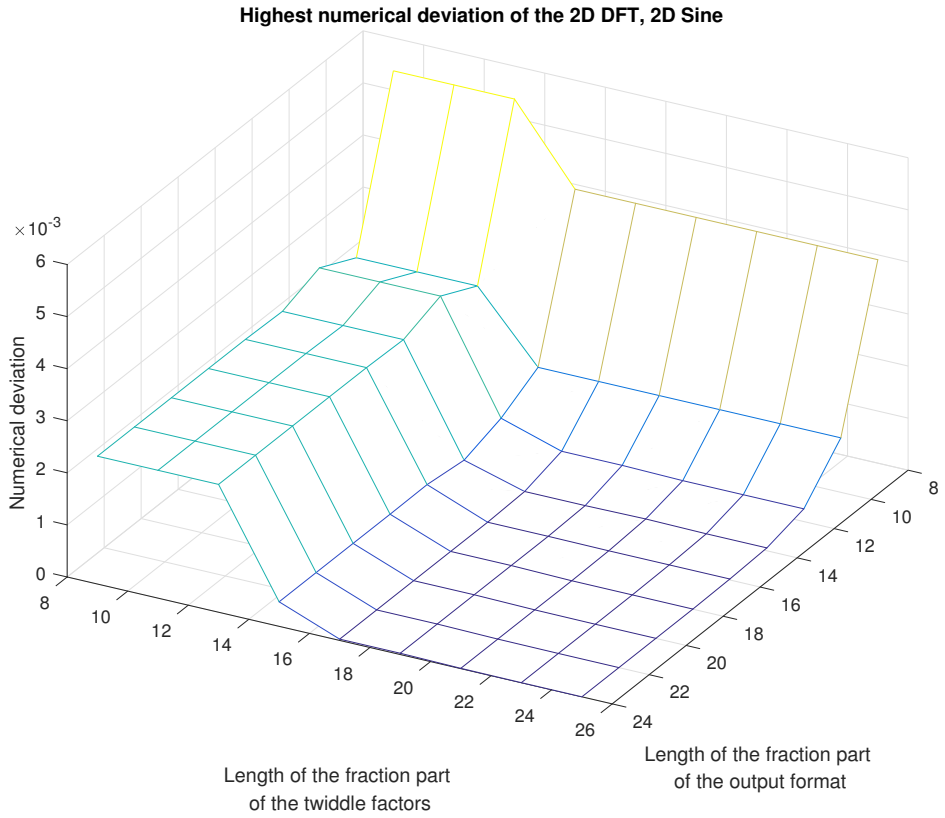

Abbildung 3.4: Höchste zahlenmäßige Abweichungen bei unskalierter 2D-DFT

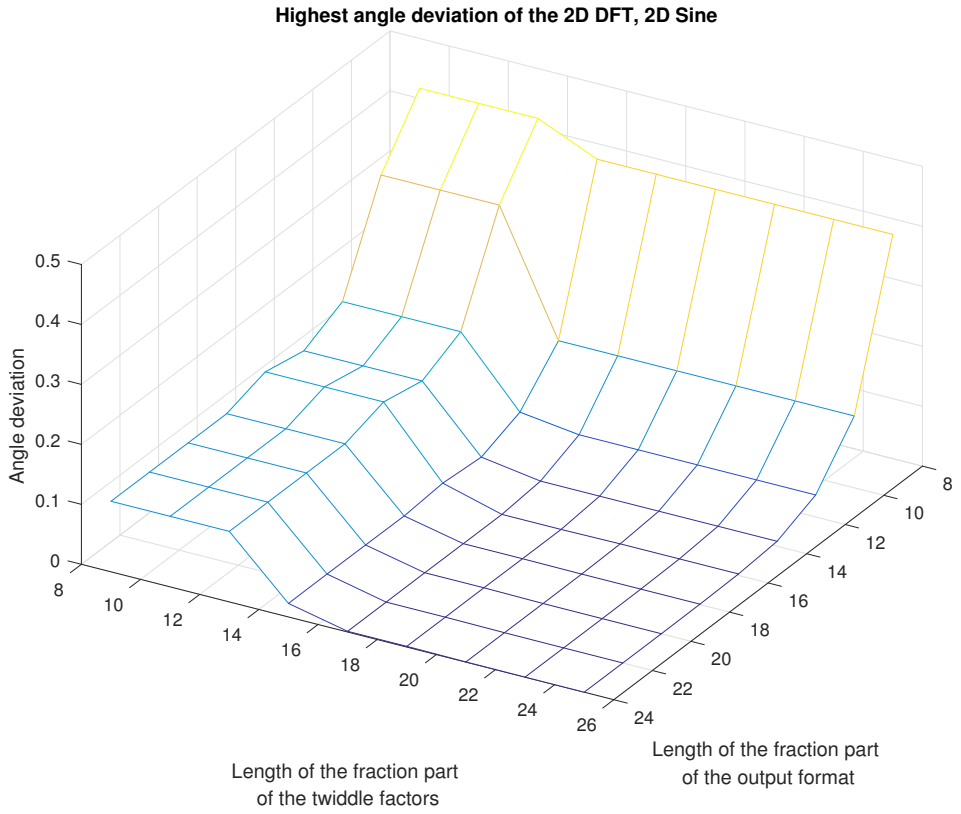

Abbildung 3.5: Höchste Abweichungen des Argumentes bei unskalierter 2D-DFT

2. 2D-DFT mit herunterskalierter 1D-DFT mit dem 4 Bit Ganzteil der Ein- und Ausgangsdaten
3. 2D-DFT mit herunterskalierten Twidddlefaktor-Matrix mit dem 2 Bit Ganzteil der Ein- und Ausgangsdaten

Es wurden mit 4 Datensätzen bei diversen Kombinationen der Bitbreieten der Daten und Twiddle-Faktoren getestet. Zusätzlich zu den im Abschnitt 3.2.2 aufgezählten Testdatensätzen wurde ein weiterer eingefügt. Aufgenommen bei jeder Kombination und jedem Datensatz wurden jeweils die höchsten numerischen Abweichungen und die größten Abweichungen des Argumenten der komplexen Zahlen. Um Berechnungen zu beschleunigen wurden sie mittels `parfor` aus der Parallel Computing Toolbox in mehreren Threads ausgeführt.

Die Abweichungen in Abhängigkeit von Breiten der Bruchteile der Twiddle-Faktoren und der Ausgangsdaten sind beispielhaft in den Abbildungen 3.4 und 3.5 dargestellt. Vollständige Versuchsdaten kann man in Tabellen B.11-B.16 finden. Für Gesamtbreiten muss man die entsprechenden Ganzteile dazurechnen. Die Testdaten überschreiten die von RAM zulässige Gesamtbreite von 16 Bit, um herauszufinden, wie breit sie sein müssten, die Anforderung der Winkelgenauigkeit von 0,5° zu erfüllen. Außerdem wird nachgewiesen, dass die größeren Breiten eine Verbesserung bewirken. Es sind sogar sehr große zahlenmäßige Abweichungen des Argumentes von nahezu 360° zu beobachten, die aber eigentlich auf dem Kreis betrachtet gering sind. Sie werden in Abschnitt 3.3.2.2 untersucht und erläutert. Basierend auf den Untersuchungen, wurde die *unskalierte 2D-DFT* mit 14-Bit breiten Twiddle-Faktoren für die vorzügliche Implementierung ausgewählt, weil sie von den in 16-Bit RAM passende Varianten am engsten an den Anforderungen liegt. Dagegen wurde die Implementierung mit der herunterskalierten 1D-DFT komplett verworfen, weil sie einen zusätzlichen Rechenschritt erfordert, ohne einen Vorteil zu bringen.

3.3 Implementierung und Verifikation auf FPGA

3.3.1 Implementierung in VHDL

3.3.1.1 `fixed_pkg` und VHDL-2008

Die Fixkomma-Implementierung von 2D-DFT (und auch 2D-Filter) erfolgte mit Hilfe der `fixed_pkg` Bibliothek, die in der Lage ist, die arithmetischen Operationen an die Fixkommazahlen *mit Berücksichtigung ihrer Fixkommaformate* anzuwenden. Das Ergebnis wird passend skaliert und ggf. vergrößert. Auch ist sie fähig, die Fixkommazahlen korrekt zu runden, was sich positiv auf die Genauigkeit auswirkt. Die zeichenbehafteten Fixkommazahlen, die durchweg verwendet wurden, werden mit dem Typ `sfixed` repräsentiert. Das Fixkommaformat wird durch Indexe festgelegt. Es wird angenommen, dass das Komma zwischen 0 und -1 liegt, so dass die Nachkommastellen durch *negative* Indexe angegeben werden. Zum Beispiel muss eine Fixkommazahl mit dem 5-Bit Ganzteil und dem 10-Bit Bruchteil als `sfixed(4 downto -10)` instantiiert werden. `fixed_pkg` hat die Funktionen `sfixed_high` und `sfixed_low`, mit denen man das Format des Berechnungsergebnis aus den Formaten der Operanden bekommen kann [16].

Die negative Indexe haben in dieser Arbeit kein Problem verursacht. Sollte es in Zukunft doch zu einem Problem damit kommen, sollte man sich trotzdem bemühen, möglichst die Funktionen aus dem `fixed_pkg` zu verwenden, anstatt sie zu reimplementieren, da die korrekte Skalierung nicht trivial und fehleranfällig ist. Dies könnte damit erreicht werden, dass die Funktionsparameter in `sfixed` konvertiert werden, und die Rückgabelwerte zurück in den genutzten Signaltyp .

`fixed_pkg` ist ein Teil vom VHDL-2008 Standard und erfordert diesen. Dies hat veranlasst, das Design auf VHDL-2008 umzustellen. Es behält zwar die Rückwärtskompatibilität, stellt aber einige bequeme Features zur Verfügung. Davon wurden die Folgenden benutzt:

- Es müssen nicht mehr alle Signale in der Sensivitätsliste ausgeschrieben werden, sondern durch `process(all)` werden alle gelesenen Signale automatisch eingeschlossen.

- Die Ausgangssignale können nun unmittelbar in der Architektur wieder gelesen werden.
- Es werden Blockkommentare `/* */` wie in C unterstützt.
- Es wurden zunächst die unbeschränkten `record`-Typen benutzt, die sehr bequem für die einheitliche Darstellung der komplexen Fixkommazahlen verschieder Formate sind. Später musste jedoch auf sie verzichtet werden, weil sie nicht durch die Cadence-Synthese unterstützt werden.

Die Standardversion von `fixed_pkg` macht von den neuartigen Package Generics Gebrauch, was von der Synthese in Vivado 2018.3 nicht unterstützt wird. Xilinx stellt jedoch eine kompatible Version zur Verfügung, die unmittelbar im Projekt eingebunden werden soll [17].

Es wurden auch einige weiteren Änderungen vorgenommen. Zuerst wurden jegliche Parameter, die noch als “ magical numbers” vorlagen, durch Definitionen von Sondertypen und symbolischen Konstanten vereinheitlicht. Somit ist dann z.B. möglich, den Typ der Datenbusse in RAM-Interface (`record`-Typen `to_mem` und `from_mem`) in einer einzigen Zeile anzupassen. Da die Daten nun die Fixkommazahlen sind, wurden die Busse entsprechend als ein `sfixed` definiert.

3.3.1.2 Konzeptdarstellung und Twiddlefaktor-Matrix in VHDL

Die 2D-DFT besteht aus zwei konsekutiven Matrizenmultiplikationen. Es bietet sich als eine Optimierung sich an, die gleiche Modulinstanz für die beiden zu verwenden. Dabei muss man allerdings berücksichtigen, dass die Twiddlefaktor-Matrix bei der ersten Multiplikation links und die Daten rechts stehen, bei der zweiten jedoch andersherum. Sie müssen also mittels eines Kreuzmultiplexors die entsprechende Seite umgeschaltet werden.

Des Weiteren ist es vorteilhaft, die Twiddlefaktor-Matrix lokal in einem ROM abzuspeichern, weil die Twiddle-Faktoren dann simultan mit den Daten geladen werden können. Darüber hinaus kann so ihr Format optimal unabhängig von dem Datenformat festgelegt werden. Das vereinfachte Blockdiagramm, das das vorgestellte Konzept des 2D-DFT-Moduls ist auf Abbildung 3.6 dargestellt. Als weitere Optimierung sollten die Berechnungen soweit wie möglich parallel ausgeführt werden. Dies ist aber nur in einem eingeschränkten Maß

möglich, weil man auf einmal nur einen Wert aus dem Speicher laden kann, und das RAM nicht gleichzeitig beschreiben und ausgelesen werden kann.

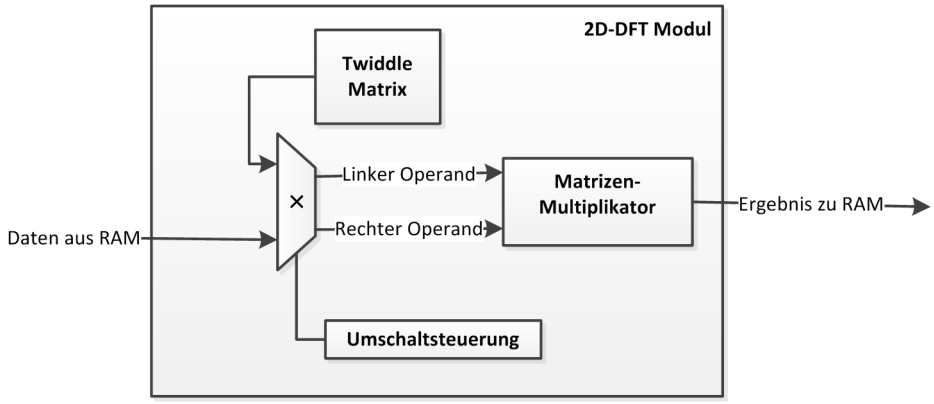

Abbildung 3.6: Vereinfachtes funktionales Diagramm des 2D-DFT Moduls.

Um die Realisierung von Twiddle-Faktoren zu vereinfachen, wurde in Matlab die Funktion `writeTwiddle` geschrieben. Die Funktion erzeugt den VHDL-Code eines Moduls `twiddle_factors_NxN`[1] als ROMs mit der Twiddlefaktor-Matrix mit der angegebenen Dimension und der Anzahl der Nachkommastellen. Optional kann die Matrix herunterskaliert werden. Da sich die inverse DFT von DFT nur durch die konjugierten Twiddle-Faktoren (und ggf. den Vorfaktor) unterscheidet, ist auch diese Option realisiert, so dass das Modul ansonsten unverändert für die inverse DFT benutzt werden kann. Die Auswahl des Ausgangsordners erfolgt mittels eines Dialogfensters. Der Datentyp eines Twiddle-Faktors ist eine komplexe Fixkommazahl, die als `record` mit einem Real- und Imaginärfeld realisiert ist. Die Funktion schreibt den Package `twiddle_pkg` zusätzlich zum ROM-Modul, wo der Indexbereich dieses Datentyps definiert wird. Das wird von anderen Modulen benutzt.

Das ROM-Modul hat zwei Architekturen:

`REC_ARRAY` speichert die Twiddle-Faktoren als Feld des Twiddle-Faktor-Datentyps.

`SLV`: Imaginär- und Realteil werden zusammen verpackt in einem `std_logic_vector` und als ein Feld gespeichert.

[1] wobei N ist variable Dimension der Twiddlefaktor-Matrix.

Dies ist dadurch motiviert, dass die eine unter unterschiedlichen Bedingungen effizienter als die andere sein kann. Es gab nämlich die Befürchtung, dass die Unterbringung von Twiddle-Faktoren im ROM viele Ressourcen verbrauchen würde, die sich jedoch nicht bestätigt hat. Im Gegenteil hat es Vivado geschafft, die 64 komplexe Twiddle-Faktoren der 8×8 Matrix in nur 9 LUTs[1] (mit REC_ARRAY) zu implementieren. Die Auswahl der Architektur erfolgt am einfachsten mit der expliziten Architekturbindung vor der Instantiierung. Beispiel:

```
architecture ARCH of some_entity is
--signals
--declaration of twiddle factors
component twiddle_factors_8x8 is
Port(
CLK: in std_logic;
EN: in std_logic;
ADDR: in address_type;
REN: in std_logic;
FACTOR_OUT: out twiddle_type
);
end component twiddle_factors_8x8;
--explicit binding
--architecture can be selected within parenthesis
for twiddle_factors :  twiddle_factors_8x8 use
entity work.twiddle_factors_8x8(REC_ARRAY);--
begin
--instantiation
twiddle_factors: twiddle_factors_8x8
--port map...
```

3.3.1.3 Matrizenmultiplikation in VHDL

Bei der Matrizenmultiplikation unterscheiden sich die Faktoren dadurch, dass der linke Operand reihenweise und der rechte Operand spaltenweise iteriert wird, was durch die entsprechende Adressierung erreicht wird. Da die Multiplikation kommutativ ist, spielt es keine Rolle, ob die Dateneingänge als links

[1] Look-Up Tabellen

oder rechts betrachtet werden, sondern es müssen *nur die jeweiligen Adressbusse* umgeschaltet werden. Diese Idee wurde im Modul `dft_mat_product` verwirklicht, das den Algorithmus auf Abbildung 3.3 implementiert

Es ist auch bequemer die Twiddle-Faktoren direkt in `dft_mat_product` zu instantiieren, damit sie nicht von außen geleitet werden müssen. Sie werden dann per ein externes Steuersignal auf rechts oder links (und die Daten aus dem RAM andersherum) schalten. Da die Faktoren und Ausgangsdaten unterschiedliche Speicherbereiche belegen können, müssen ihre Startadressen von außen eingestellt werden.

Die zentrale Stelle in der Implementierung nimmt die komplexe Multiplikation und Akkumulierung. Der Produkt zweier komplexen Zahlen in kartesischer Form ist:

$$(a + bi)(c + di) = ac - bd + (ad + bc)i \tag{3.5}$$

Sei q der Realteil und p der Imaginärteil des Akkumulator ist die Gesamtberechnung:

$$\begin{aligned} q_{neu} &= ac - bd + q_{alt} \\ p_{neu} &= ad + bc + p_{alt} \end{aligned} \tag{3.6}$$

Diese Berechnung ist vermutlich zu komplex, um in einem Takt erledigt zu werden, deswegen muss sie auf zwei Zustände verteilt werden. Die offensichtliche Methode ist, zuerst (im Zustand `MULTIPLY`) die vier Produkte (parallel) zu bilden und in Zwischenregistern zu speichern. Dann im zweiten Takt im Zustand `ACCUMULATE` diese gemäß Gleichungen 3.6 miteinander zu addieren und subtrahieren. Dabei ist zu berücksichtigen, dass die Summe bzw. die Differenz der Fixkommazahlen um 1 Bit größer ist, als die der Operanden, so dass der Akkumulator ständig wachsen müsste. Da dies in Hardware nicht möglich und nicht unbedingt sinnvoll ist, wird die Breite der Ergebnis mit der Funktion `resize` passend verringert und sicherheitshalber gesättigt.

Die Multiplikation wird von Vivado auf dem Zynq FPGA mittels der DSP48 Blöcke [18] realisiert, die in der Lage sind, die Multiplikation mit Addition oder Subtraktion (in der Form $r \cdot s \pm t$) zu vereinen und die eigene eingebaute Register besitzen [19]. Es ist attraktiv die Implementierung so zu optimieren, um diese Register auszunutzen. Zunächst soll die Berechnung wie folgt umgeordnet, wobei f und g zwei Zwischenregister sind:

$$\begin{aligned} f &= ac + p_{alt} & g &= ad + q_{alt} \\ p_{neu} &= f - bd & q_{neu} &= g + bc \end{aligned} \tag{3.7}$$

Auf diese Weise werden außerdem zwei Register eingespart, was auch außerhalb des FPGA einen Vorteil bringen könnte.

Damit der Akkumulator und die Zwischenregister an die eingebauten Register adaptiert werden können, darf sich keine zusätzliche Logik vor dem Akkumulator und um die Zwischenregister befinden. Deswegen wurde auf die Sättigung verzichtet. Das kann man sich leisten, wenn der Ganzteil von vornherein ausreichend breit ausgelegt wird. Die Breite des Ganzteils wird durch den generischen Parameter `ACC_HIGH` eingestellt, dessen Default-Wert so ausgerechnet wird, dass Überläufe des Akkumulators auf jeden Fall ausgeschlossen sind.

Nachdem ein Ausgangswert abgearbeitet wurde, muss man den Akkumulator wieder auf 0 setzten. Eine übliche Zuweisung würde jedoch seine Absorption durch den DSP48 Block verhindern. Statt dessen wird der Register-Reset Eingang von DSP48 benutzt. Dafür wurde ein internes Reset-Signal für den Akkumulator programmiert. Damit wurde schließlich das Ziel, die Register auf den DSP48+Block zu verschieben, erreicht. Da aber nicht sicher ist, ob diese Methode auch für ein ASIC geeignet ist, ist es immer noch möglich zwischen beiden Methoden mit dem generischen booleschen Parameter `PIPELINE_ACC` zu schalten. Mit `true` wird die für das FPGA optimierte Methode ausgewählt, andernfalls die konventionelle.

Das Modul ist als ein Moore-Automat (Abbildung A.12) implementiert mit dem Ziel, die Berechnung möglichst schnell, also in wenigen Zuständen, zu erledigen. Da kommt dem effizienten Laden der Operanden aus den Speichern (RAM und ROM) eine große Bedeutung zu, weil es für jeden Akkumulationsschritt nötig ist. Es sollte parallel mit und im Voraus zu der Berechnung passieren. Zunächst wird der Lesevorgang im Zustand `READ` initiiert, wo die Anfangsadressen und Read-Enable für die anschließende Lesevorgänge zugeschaltet werden. Was aus den Speichern geladen wird, wird dann durch Speicheriteratoren-Register kontrolliert, die durch die aktuellen Reihe und Spalte fortschreiten. Zwischen dem Inkrementieren von der Iteratoren und der Verfügbarkeit der nachfolgenden Werte liegen zwei Taktflanken: bei der ersten wird das Inkrement wirksam, bei der zweiten wird der Wert aus der aktualisierten Adresse geholt. Deshalb werden die Iteratoren im Zustand `MULTIPLY` inkrementiert, damit der nächste Wert bei der nächsten Runde verfügbar ist. Somit wird gewissermaßen eine Pipeline gebildet.

Der linke und der rechte Speicheriterator laufen zwar in die jeweils horizontale und die senkrechte Richtungen, müssen aber die gleiche Anzahl an Schritten machen, weil die Matrizen die gleiche Dimensionen haben. Das heißt es reicht,

nur bei einem zu überprüfen, ob er das Reihen- bzw. Spaltenende erreicht hat, was bei dem linken Speicheriterator sich einfacher realisieren lässt. Wenn dies also geschieht muss die Akkumulationschleife im Zustand ACCUMULATE abgebrochen werden. Das Inkrementieren, wie schon erwähnt, geschieht davor im Zustand MULTIPLY, so dass sie in ACCUMULATE zu weit laufen könnten. Um so einen Überlauf zu verhindern, wird die Abbruchbedingung schon in MULTIPLY geprüft und in einem Flip-Flop gespeichert (SAVE_VAL) und auf diese Weise um einen Takt verzögert, so dass der Abbruch rechtzeitig in ACCUMULATE stattfinden kann.

Mit dem generischen Parameter REDUCE_PREC lässt sich die Bitbreite der Bruchteile von Akkumulator und den Zwischenregistern verkleinern, wodurch Flip-Flops eingespart werden könnten (vgl. Tabelle 3.5). Anderseits wird dadurch die Rundung während des Akkumulierens notwendig, was wiederum Logik kostet. Auf FPGA bringt es jedoch keinen Vorteil, weil die DSP48-Register benutzt werden, die sowieso vorhanden sind. Im Gegenteil würde die Rundungslogik verhindern, dass der Akkumulator und die Zwischenregister in diesen Registern unterbracht werden können.

Es muss noch auf die Power-On Resets eingegangen werden, die einen fundamentalen Unterschied zwischen FPGA und ASIC darstellen. Es ist auf dem FPGA möglich, für die logische Bauteile (vor allem Flip-Flops) die Defaultwerte zu definieren. Sogar wenn keine Defaultwerte angegeben sind, werden sie trotzdem mit definierten Anfangswerten initialisiert, so dass ein Power-On Reset zwecklos ist. In der Tat empfiehlt Xilinx ausdrücklich gegen Power-On Resets und auch ansonsten die Resets spärlich einzusetzen [18]. Da das Design demnächst noch eine unbestimmte Zeit auf dem FPGA laufen soll, wurde dieser Hinweis befolgt. In diesem und auch allen anderen Modulen in dieser Arbeit wurden für alle Register Defaultwert angegeben und die Resets nur für Zustandsregister eingesetzt. Dagegen sind die Defaultwerte auf einem ASIC wirkungslos. Wenn es zum engültigen ASIC-Design kommen wird, müssen sie durch Power-On Resets ersetzt werden.

3.3.1.4 Modul dft_2d

Das Modul dft_mat_product ist seinerseits im Modul dft_2d instantiiert. Seine Aufgabe ist, die Matrixmultiplikation zweimal zu starten und dabei passend die Adressbusse schalten und die Adressbereiche einzustellen. Es ist als ein einfacher Moor-Automat realisiert (Abbildung A.20).

Die Implementierung insgesamt ist mit einem geringen Aufwand auf beliebige Größe skalierbar. Die Größe und die zu verwendende Speicherbereichen sind mit den generischen Parametern skalierbar. Dabei wird das Ergebnis der ersten Multiplikation, also der 1D-DFT der Reihen, zwecks Verifikation ein eigenständiger Speicherbereich zugewiesen.

Für eine andere DFT-Größe sind folgende Anpassungen vorzunehmen:

1. In Instanziierung von `dft_2d` in `proc_sys` die DFT-Größe und Speicherbereiche ändern.
2. Mit `writeTwiddle` die neue Twiddlefaktor-Matrix erzeugen und im Quellcodeordner speichern. Dann den Modulnamen in `dft_mat_product` anpassen.
3. Gegebenenfalls das Format der Daten in `interface_pkg` modifizieren.

3.3.2 Verifikation der FPGA-Implementierung

3.3.2.1 Beschreibung der Versuche

Als erstes wurde die FPGA-Implementierung simuliert (Anhang B.1.4) und Programmierfehler beseitigt. Die Simulation hat 2438 Taktzyklen für die gesamte Ausführung gezeigt. Zu diesem Zeitpunkt war das Filter-Modul noch nicht in Angriff genommen. Seine Ausgangssignale wurden auf ihre Default-Werte gesetzt.

Die 2D-DFT soll nun gründlich verifiziert werden. Das ist eben der Hauptzweck des im Kapitel 2 beschriebenen Systems. Mit seiner Hilfe werden die Testdaten auf das FPGA geladen, wo die 2D-DFT berechnet wird. Daraufhin werden die Ergebnisse ausgelesen und ausgewertet. Die Bewertungsparameter zu der mit Matlab-internen `fft` bzw. `fft2` ausgerechneten Werten waren zuerst die höchste zahlenmäßige Abweichung und die Abweichung des Argumentes. Anschließend werden diese Parameter als Tabellen und Graphen dargestellt.

Zuerst wurde mit der schon im Abschnitt 3.2.3.2 erwähntenm, künstlich erzeugten Daten getestet, um zu überprüfen, ob die 2D-DFT im Grunde funktioniert. Dafür wurden mit `FixedDFT.dft1D` bzw. `FixedDFT.df2D` zusätztlich zu den Matlab-internen `fft` bzw. `fft2` verglichen. Die Tabelle B.17 weist nach, dass die DFT auf FPGA fudurchaus funktioniert. In den weiteren Tests

wurde die Daten zur 1D-DFT zwar sicherheitshalber aufgenommen, aber nicht ausgewertet, weil die 2D-DFT schon funktioniert.

Danach wurde begonnen mit den echten Sensordaten zu testen, die T. Mehm im Laufe seiner Arbeit aufgenommen hat [11]. Diese Daten bestehen aus 9 Datenreihen mit jeweils 182 Datensätzen. Die letzten enthalten die separaten Vektoren mit positiven und negativen Kosinus- und Sinussignalen. Die eigentliche Sensorarray-Matrix bekommt man darauf gemäß der Formel 1.1, wessen ganzzahlige Elemente direkt als Fixkommazahlen interpretiert werden können. Ein Problem damit ist, dass viele Bruchteile durch die Division durch 2 entstehen, die nicht direkt einer Fixkommazahl zugeordnet werden können, sondern gerundet werden müssen, was sich negativ auf die Genauigkeit auswirken könnte. Eine Lösung könnte der Verzicht auf die Division sein, wodurch die Sensordaten effektiv skaliert werden. Das wäre zulässig, weil alle Werte in 10 Bit passten. Zum Vergleich wurden die Tests sowohl mit den auf diese Art skalierten als auch unskalierten, aber gerundeten Daten durchgeführt (Abbildungen B.42-B.45). In allen Versuchen mit Sensordaten wurden alle Datensätze in einer Schleife abgearbeitet.

Vorausschauend kann man sagen, dass die skalierten Daten genauer waren. Viel mehr ist generell vorteilhaft, die Sensordaten in den für das aktuellen Datenformat optimalen Bitbereich zu bringen. Das heißt sie so nach links zu shiften, dass die zwei MSBs im Bitbereich vom Ganzteil liegen. So wird der Wertebereich des Fixkommaformats am besten ausgenutzt, was die Genauigkeit verbessert.

Die Parameter werden im Anhang B.3.3 sind jeweils auf zwei Graphen präsentiert. Der erste stellt den Parameter bei jeder Sensormatrix dar, der zweite zusammengefasst nach den 9 Datenreihen. Zur besseren Vergleichbarkeit verschiedener Datenformate, wurden die höchsten zahlenmäßigen Abweichungen auf das positive Aussteuerung normiert.

Insgesamt lassen die Abbildungen B.42 und B.43 behaupten, dass die Berechnung von 2D-DFT auf dem FPGA ganz gut funktioniert. Die zahlenmäßigen Abweichungen sind mit maximal 0,11‰ vom Fullscale (auch entspricht 3-4 LSBs) sind sehr gering. Es funktioniert also erheblich besser mit den eigentlichen Sensordaten als mit den künstlich generierten Testdaten. Allerdings tauchten sehr hohe rechnerische Abweichungen des Argumentes mit max. 360° auf (Abbildungen 3.7 und 3.8), weit von den angestrebten 0,5°.

Als eine erste Abhilfe wurde versucht, die Genauigkeit durch breitere Twiddlefaktoren zu erhöhen, was fehlschlug. Dann wurde die 2D-DFT mit der herunterskalierten Twiddlefaktor-Matrix probiert, da dann das Fixkommaformat der Daten mehr Nachkommabits haben darf. Es ist übrigens erwähnenswert, dass es einfach ist, zwischen unterschiedlichen Implementierungsvarianten der 2D-DFT zu wecheln. Dafür muss die neue Twiddlefaktor-Matrix mit `writeTwiddle` mit gewünschten Eigenschaften generiert werden und das Datenformat (`data_type` in `interface_pkg`) angepasst werden. Leider hat sich gezeigt (s. Anhang B.3.3.3), dass diese Variante noch etwas schlechter ist als die unskalierte 2D-DFT. Auf der anderen Seite hat sie 50 LUTs weniger verbraucht, deshalb wurde sie im Sinne der Ressourceneinsparing weiterhin mituntersucht.

An dieser Stelle gab es kurz die Überlegung, das Datenformat zu erweitern, was gleich abgelehnt wurde, weil ein ASIC nur eingeschränkte Ressourcen hat. Im Gegenteil sollte man möglichst mit 12 Bit auskommen, was nun auch zu untersuchen ist. Darüber hinaus würde dies eine große Modifizierung vom Testsystem erfordern. Auch kam in Frage, ob die Abweichung des Argumentes überhaupt als ein Bewertungsmaß geeignet ist. Als ein weiterer Parameter wurde die euklidische Distanz auf der komplexen Zahlenebene aufgenommen, die sich wie folgt berechnen lässt:

$$d = |z_1 - z_2| \tag{3.8}$$

Abbildung 3.7: Höchste und durchschnittliche Abweichungen des Argumentes bei der unskalierten 16-Bit 2D-DFT, alle Datenreihen
Blau: durch 2 geteilte und gerundete Daten
Rot: nicht dividierte (skalierten) Daten

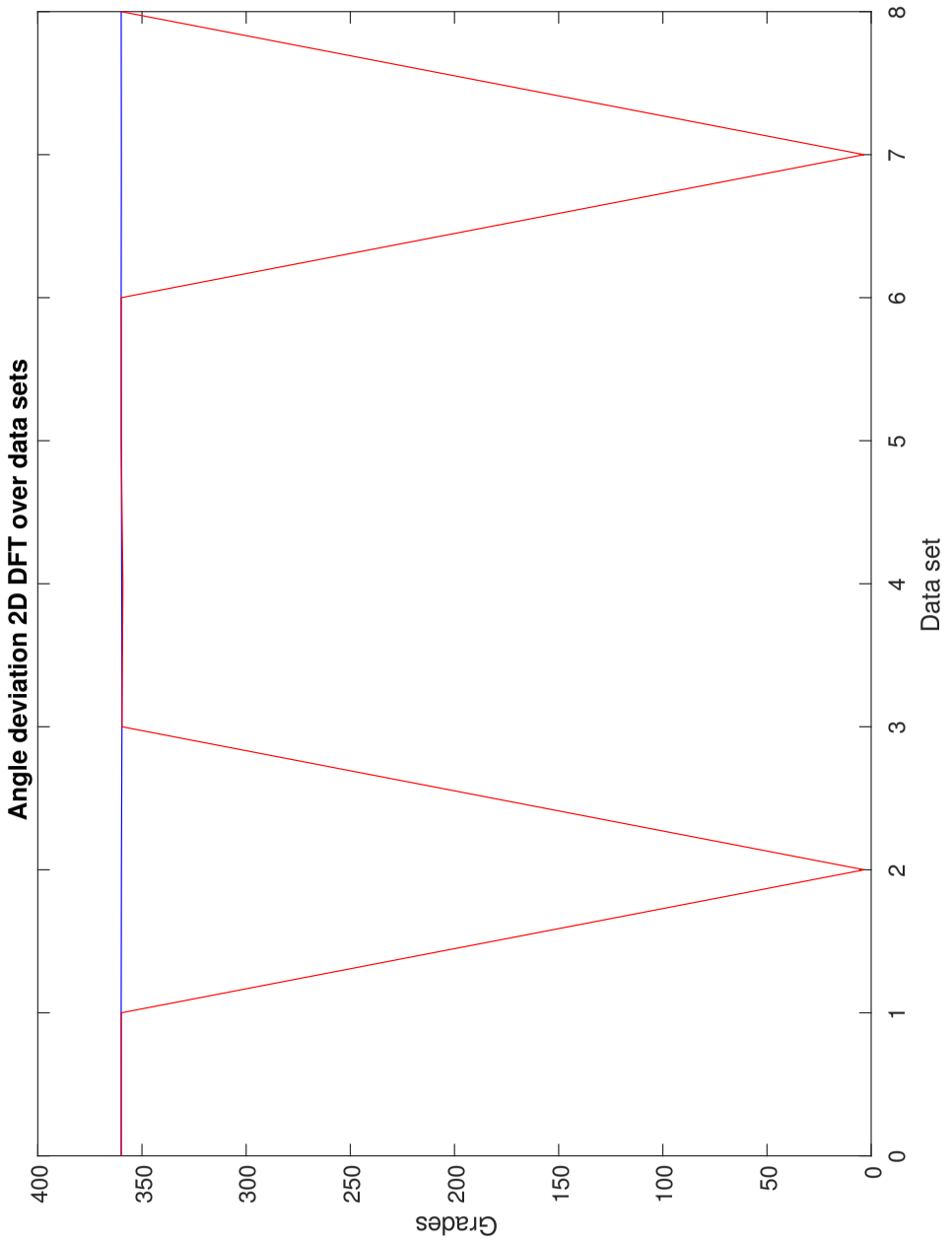

Abbildung 3.8: Höchste und durchschnittliche Abweichungen des Argumentes bei der unskalierten 16-Bit 2D-DFT, Übersicht über Datenreihen
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

3.3.2.2 Untersuchung der großen Abweichungen des Argumentes

Es soll nun untersucht werden wie die großen rechnerischen Abweichungen des Argumentes zustande kommen. Zu diesem Zweck wurde alle Werte aufgenommenen, bei denen eine Abweichung größer als ein Schwellwert vorkommt. Auch ihre Positionen in der Sensor-Matrix wurden erfasst. Um sie zu veraunschaulichen, wurde die Verteilung der Abweichungen innerhalb der Sensor-Matrix ermittelt, wobei die niedrigen Ortsfrequenzen in die Matrixmitte mit `fftshift` verschoben wurden. Der Einfachheit halber werden in diesem Abschnitt nur die in den passenden Bitbereich geshiftete Daten betrachtet.

Aus der Abbildung 3.9 wird offensichtlich, dass die nahezu 360° zahlenmäßigen Abweichungen des Argumentes dann auftreten, wenn der Realteil negativ ist und der Imaginärteil kleiner als ein LSB ist, sodass er unterläuft und zu 0 wird. Oder auch wenn sich das Vorzeichen sich aufgrund von Rechenungenauigkeiten ändert. Der Grund dafür ist, dass die Funktion atan2, die zugrunde der Argumentenfunktion einer komplexen Zahl liegt, an der Stelle von $\pm 180°$ singulär ist. Die Funktion weist deswegen einen Sprung im Bereich der negativen Realachse auf, was auf der Abbildung 3.10 veranschaulicht ist. Solche Sprünge lassen sich jedoch mit der Funktion `unwrap` glätten.

	Matlab FFT value	Matlab angle	FPGA value	FPGA angle
1	-0.28856-0.00029	-179.94288	-0.29102+0.00000	180.00000
2	-0.76677-0.00032	-179.97579	-0.76953+0.00000	180.00000
3	-2.53214-0.00095	-179.97853	-2.53320+0.00195	179.95582
4	-0.19215-0.00129	-179.61684	-0.19336+0.00000	180.00000
5	-0.24663-0.00364	-179.15341	-0.24805+0.00000	180.00000
6	-2.93361-0.00073	-179.98576	-2.93359+0.00195	179.96185
7	-3.25236-0.00169	-179.97019	-3.25195+0.00000	180.00000
8	-0.92350+0.00068	179.95773	-0.92383-0.00195	-179.87887
9	-0.63357-0.00085	-179.92279	-0.63281+0.00000	180.00000
10	-1.49728-0.00097	-179.96279	-1.49609+0.00000	180.00000
11	-1.98159-0.00029	-179.99152	-1.98047+0.00000	180.00000
12	-1.32304-0.00098	-179.95777	-1.32422+0.00000	180.00000
13	-2.69314-0.00083	-179.98231	-2.69336+0.00000	180.00000
14	-0.39167+0.00199	179.70869	-0.39258-0.00195	-179.71495

Abbildung 3.9: Werte mit besonders großen zahlenmäßigen Differenzen des Argumentes (>200°). Diese Differenzen ergeben sich aus den mit der Funktion `angle` kalkulierten Argumenten (die zweite und die vierte Spalten) aus den jeweils von Matlab (erste Spalte) und auf dem FPGA (dritte Spalte) berechneten 2D-DFT-Werten. Die tatsächliche auf dem Kreis betrachtete Differenz der Argumenten ist hier selbstverständlich klein.

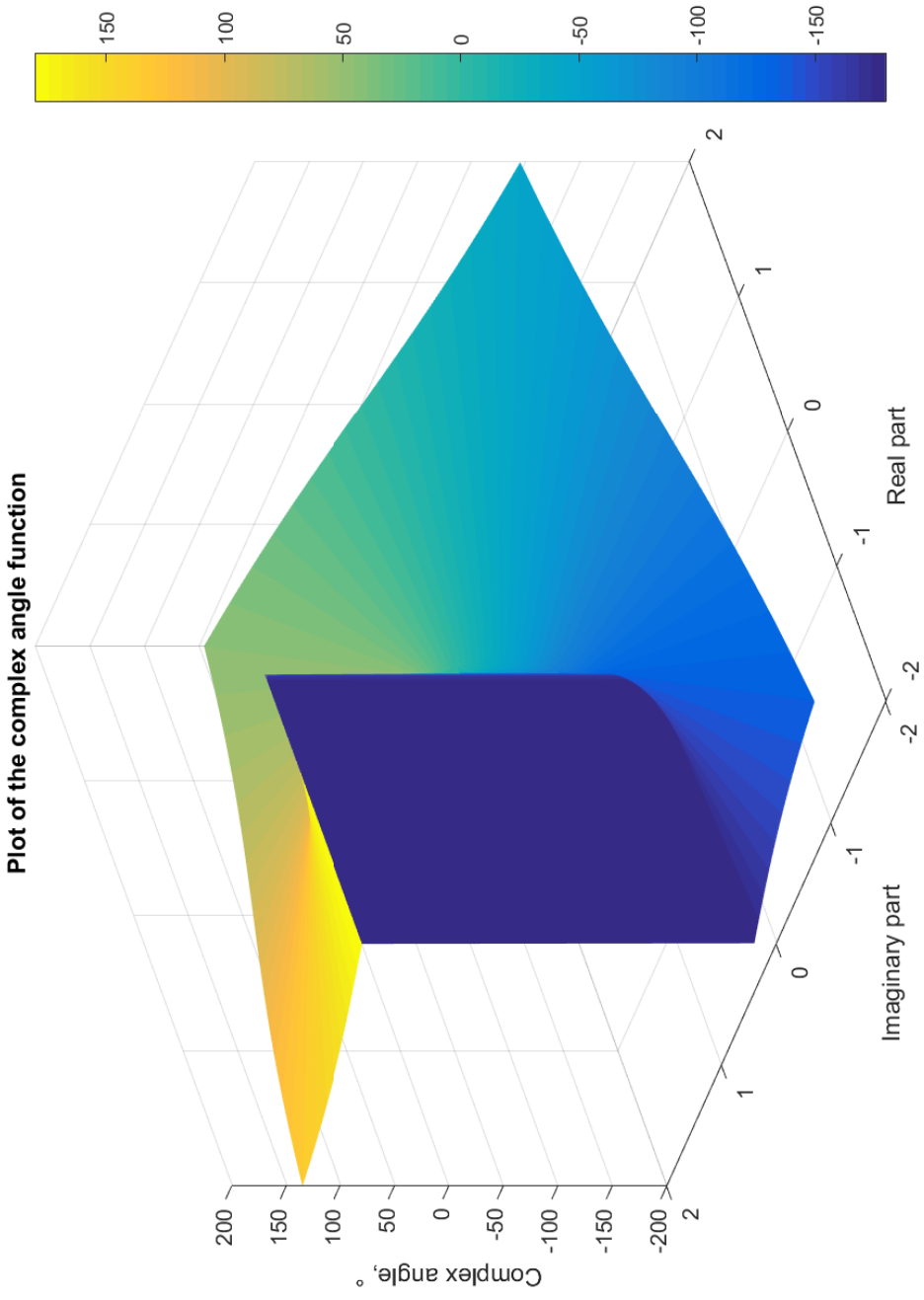

Abbildung 3.10: Darstellung der Funktion des Argumentes einer komplexen Zahl.

Mithilfe von `unwrap` lassen sich zwar übergroße rechnerische Abweichungen des Argumentes beseitigen (z. B. vgl. Abbildung 3.8 mit der Abbildung B.47), sie bleiben jedoch erheblich. Wie auf Abbildung B.48 sichtbar ist, wird dies durch kleine Real- oder Imaginärteile verursacht. Da der Argument einer komplexen Zahl aus dem Verhältnis des Real- und Imaginärteilen abhängt, führen in diesem Fall kleinste Ungenauigkeiten zu großen Abweichungen des Argumentes. Besonders deutlich ist es bei der 12-Bit 2D-DFT in den Abbildungen B.67 und B.69 erkennbar, wo die Abweichungen des Argumentes sehr groß werden, obwohl die euklidische Distanz (Abbildung B.66) gering bleibt.

Um zu bestimmen, wie schwerwiegend diese Komplikation ist, ist es wissenswert, an welchen Positionen, also bei welchen Ortsfrequenzen diese Abweichungen des Argumentes vorkommen. Dies ist auf den Abbildungen B.49, B.70 und B.70 präsentiert. Darauf ist zu sehen, dass die größeren Abweichungen des Argumentes ausschließlich bei den hohen Frequenzen auftreten, während die niedrigen Frequenzen in der Mitte unbetroffen bleiben. Dies macht das Problem weniger kritisch, weil die Magnetfelder inhärent niederfrequent sind und hohe Frequenzen eher dem Rausch zuzuordnen sind.

Die Annahme, dass die Abweichungen des Argumentes von der Größe der Werte abhängt, wurde zusätzlich durch die Aufnahme der Verteilung des durchschnittlichen Betrags der komplexen Zahlen in Ergebnissen der 2D-DFT bestätigt (Abbildungen B.50, B.61 und B.71). Die Werte sind eben im Bereich der niedrigen Frequenzen am größten. Darüber hinaus ist der Gleichanteil relativ hoch. Um seinen Einfluss auf die Abweichungen des Argumentes zu finden, wurde ihre Verteilung bei den vom Mittelwert bereinigten Sensordaten erfasst (Abbildung B.62). Sie ist bis auf die Mitte fast identisch mit Abbildung B.60, folglich ist der Zusammenhang gering.

Die Tabelle 3.6 fasst alle Bewertungsparameter zusammen.

3.3.2.3 15×15 2D-DFT

Es besteht die Option, dass die Sensordaten vor der 2D-DFT interpoliert werden. Die Motivation dafür wäre unter anderem, dass das Mittelelement bei einer Sensormatrix von ungerader Dimension direkt den Offset darstellt. Deshalb wurde auch die 15×15 2D-DFT getestet, zumal die Implementierung sich

Tabelle 3.6: Zusammenfassung der Bewertungsparameter über alle Werte

Version der 2D-DFT	**Max. num. Abweichung**	**Eukl. Abweichung**		**Abweichung des Argumentes**		
		maximal	durchsch.	maximal	durchsch.	Anzahl >1°
unskalierte 16-Bit	$1,1\times10^{-4}$	$1,23\times10^{-4}$	$1,9\times10^{-5}$	13,99°	0,0387°	167
skalierte 16-Bit	$2,56\times10^{-5}$	$3,13\times10^{-5}$	$1,2\times10^{-5}$	8,13°	0,053°	214
skalierte 12-Bit	$2,7\times10^{-5}$	$3,12\times10^{-5}$	$1,23\times10^{-5}$	180°	0,88°	24681

leicht skalieren lässt (s. Abschnitt 3.3.1.4). Zu diesem Zweck wurden die Sensordaten mit `interp2` linear interpoliert. Die Versuche wurden mit der 16-Bit 2D-DFT (s. Anhang B.3.3.5) und der 12-Bit 2D-DFT (s. Anhang B.3.3.6) durchgeführt. Die Ergebnisse sehen auf dem Niveau der 8×8 2D-DFT ganz ordentlich aus. Die Berechnung der 15×15 2D-DFT nimmt 14856 Taktzyklen in Anspruch.

3.4 Synthese und Layout in der Cadence Toolchain

Um die Vorstellung über Ressourcenverbrauch auf einem ASIC zu bekommen, wurde das Design in Cadence Toolchain für den 350 µm (3 Metalllagen) Prozess der Firma Austria Microsystems (AMS) implementiert. Genus ist das Synthesewerkzeug, während Innovus für Layout zuständig ist.

Als erstes wurde das Twiddlefaktor-ROM separat synthetisiert, um zu schauen, wie effizient es sich implementieren lässt. Die sich daraus ergebenden 59 Standardzellen für die 64 komplexen 14-Bit Twiddlefaktoren können als sehr gering erachtet werden.

Daraufhin wurde die 2D-DFT (mit `dft_2d` als Topmodul) eigenständig synthetisiert, um ihren eigenen Ressourcenbedarf herauszufinden. Um dies zu ermöglichen wurden die Default-Werte generischen Parametern zugewiesen.

Genus unterstützt eigentlich das `fixed_pkg` offiziell nicht und stellt auch keines zur Verfügung. Es hat aber die Version von Xilinx akzeptiert. Damit das klappt, wurde im Syntheseskript [10] die VHDL-2008 aktiviert und `fixed_pkg` in die Bibliothek `ieee` kompiliert. Man muss dabei auf die korrekte Kompilierordnung, dass heißt auf die Reihenfolge der Quellcode-Dateien im Skript achten. Ein Tipp dafür kann “Compile order” von Vivado in Project Manager geben, vorausgesetzt, es ist auf “automatic” eingestellt.

Es wurden viele verschiedene Implementierungsvarianten ausprobiert, die in Tabelle 3.7 aufgelistet sind. Fett markiert sind die Varianten, die für eine Implementation in Innovus ausgewählt wurden.

Tabelle 3.7: Synthese der 2D-DFT in Cadence Genus.

Größe	Eigenschaften	Anzahl von Standardzellen
8×8	**unskaliert, 16-Bit Daten, 14-Bit Twiddle-Faktoren**	1779
8×8	**herunterskaliert, 16-Bit Daten, 16-Bit Twiddle-Faktoren, `PIPELINE_ACC=true`**	1889
8×8	herunterskaliert, 16-Bit Daten, 16-Bit Twiddle-Faktoren, `PIPELINE_ACC=false`	2042
8×8	herunterskaliert, 16-Bit Daten, 16-Twiddle-Faktoren, um 8 Bit reduzierte Akkumulatoren und Zwischenregsiter	1869
8×8	herunterskaliert, 16-Bit Daten, 14-Bit Twiddle-Faktoren	1717
8×8	**herunterskaliert, 12-Bit Daten, 12-Bit Twiddle-Faktoren**	1420

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle 3.7

Größe	Eigenschaften	Anzahl von Standardzellen
15×15	**herunterskaliert, 16-Bit Daten, 16-Bit Twiddle-Faktoren**	3356
15×15	**herunterskaliert, 12-Bit Daten, 12-Bit Twiddle-Faktoren**	2404

Die Anzahlen von Standardzellen kann man angesichts des anvisierten Budgets von 10000 Zellen und im Vergleich zu Vorgängerarbeiten [8, 10] für niedrig halten. Bemerkenswert ist auch die geringe Differenz zwischen 12- und 16-Bit Varianten.

Für eine belastbare Abschätzung vom Flächenbedarf muss ein Layout in Innovus zu erstellt werden. In Tabelle 3.8 sind alle fett markierten Varianten aus der Tabelle 3.7 zu finden.

Tabelle 3.8: Flächenverbrauch von 2D-DFT beim Layout in Cadence Innovus.

Größe	Variante	Fläche, mm^2
8×8	unskalierte 2D-DFT, 16 Bit	0,3834
8×8	herunterskalierte 2D-DFT, 16 Bit	0,4381
8×8	herunterskalierte 2D-DFT, 12 Bit	0,3109
15×15	herunterskalierte 2D-DFT, 16 Bit	0,6773
15×15	herunterskalierte 2D-DFT, 12 Bit	0.4623

Der aufgewiesene Flächenbedarf von etwa die 0,5 mm^2 ist als durchaus moderat zu bewerten.

Es soll nun der realistische Bedarf des gesamten Testsystems (inklusive 2D-DFT) ermittelt werden, daher soll es mit dem originalen `sram1024x32`-Block von AMS implementiert werden. Er lag als eine Black-Box-Zelle vor und wird aus Lizenzgründen nicht mit der Arbeit geliefert. Der `memory_access_ctrl` wurde so modifiziert, um diesen Block zu instantiieren. Für die Implementierung wurde die unskalierte Variante genommen. Die Synthese ergab 1984 Standardzellen, inklusive AMS-SRAM. Die Komplexität der Logik hat sich also nur geringfügig erhöht.

Im Default-Layout hat Innovus die RAM-Zelle ineffizient mittig platziert, wie auf der Abbildung 3.11 gezeigt ist. Die Fläche wurde manuell geschrumpft und die RAM-Zelle verschoben, was in dem Layout auf der Abbildung 3.12 resultierte. Die belegte Fläche darauf beträgt 2,39 mm^2, davon 1,88 mm^2 durch AMS-RAM beleht.

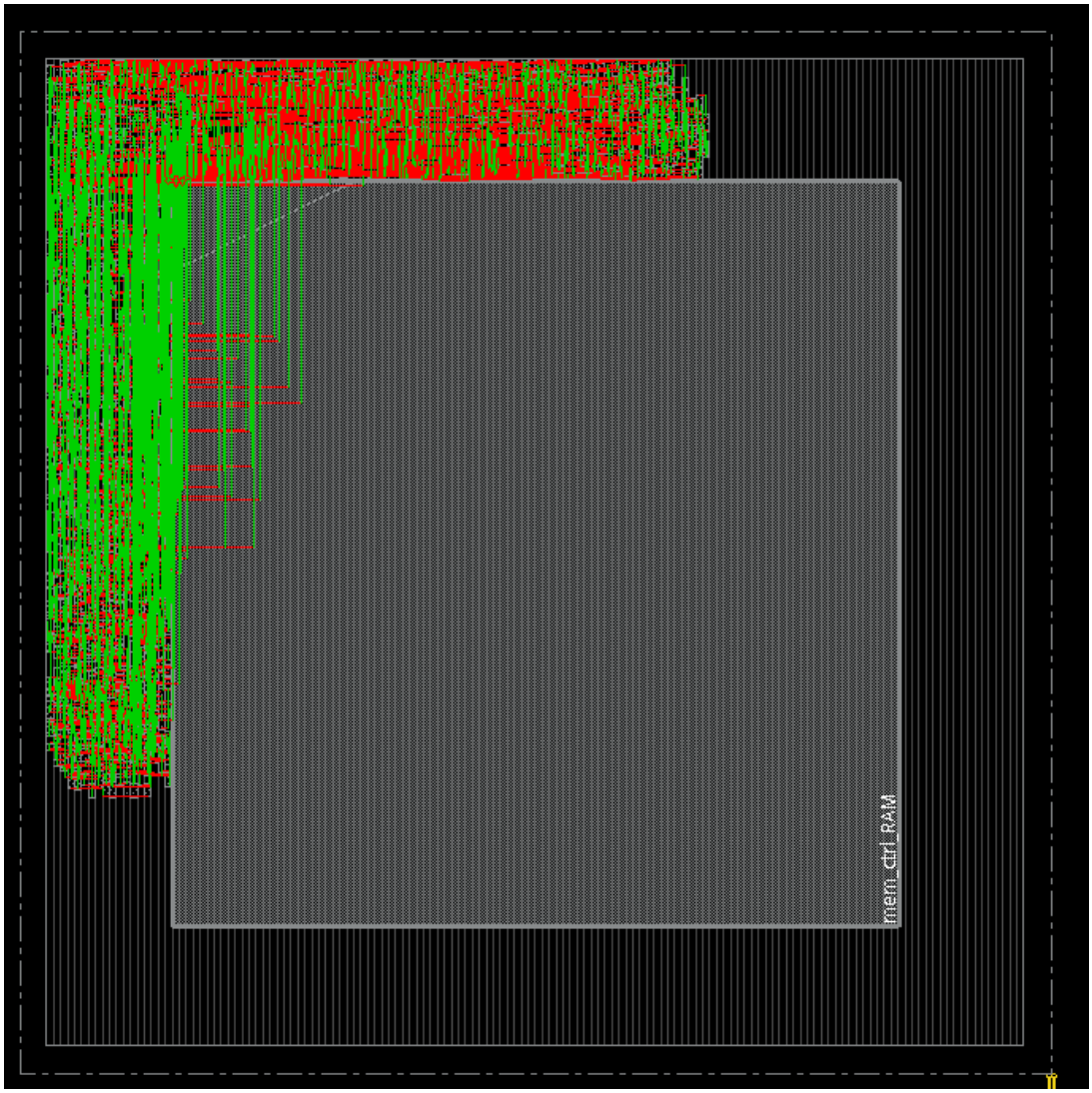

Abbildung 3.11: Default-Platzierung des Testsystems

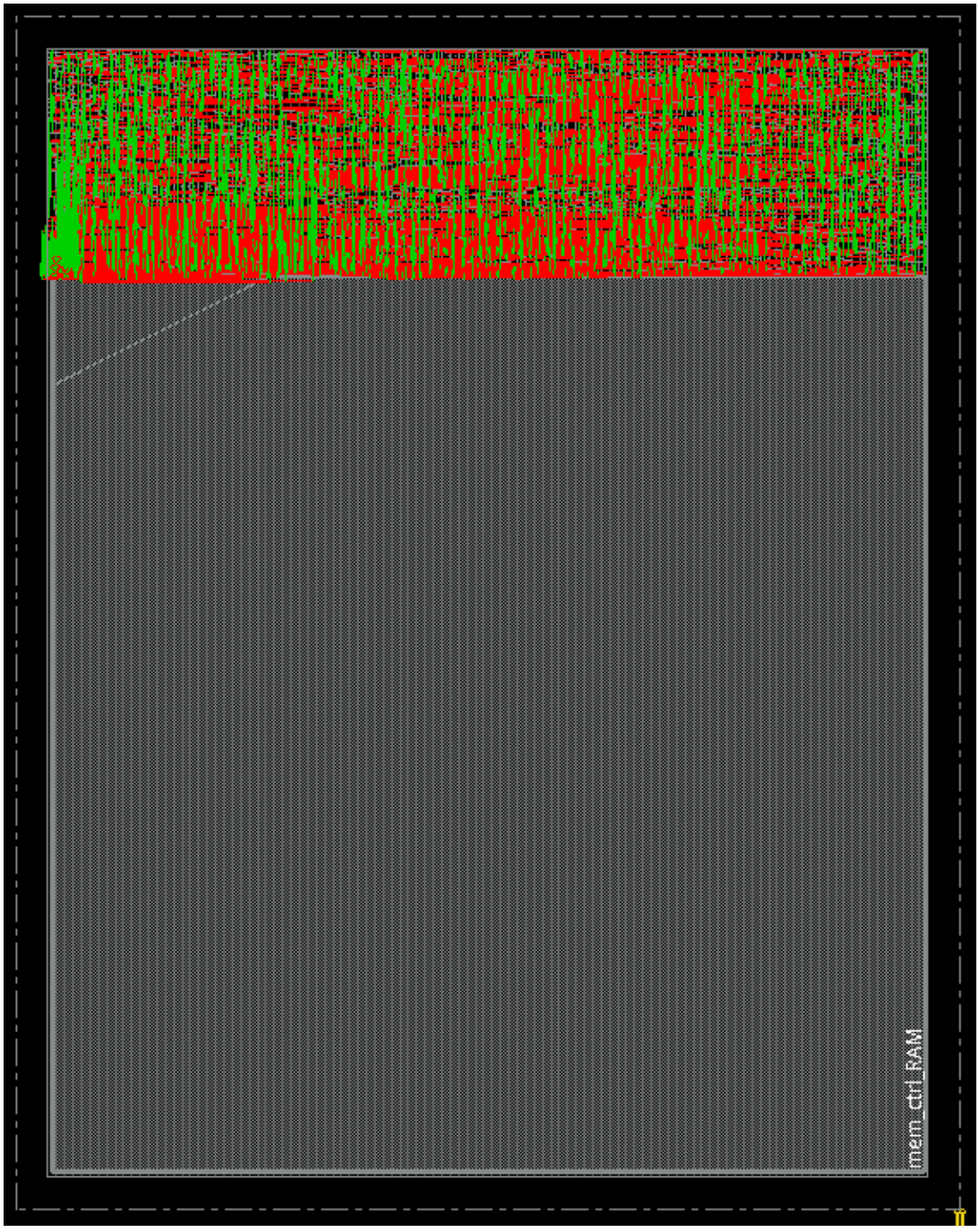

Abbildung 3.12: Platzierung des Testsystems nach dem Schrumpfen der Fläche und Verschieben der RAM-Zelle

4 Implementierung der zweidimensionalen Digitalfilter in VHDL

4.1 Implementierung im Ortsfrequenzbereich

4.1.1 Theorie

Im nächsten Verarbeitungsschritt sollen die Daten vom Sensor gefiltert werden. Dadurch sollen der Gleichanteil, das Rauschen und Störfelder eliminiert werden.

Der beabsichtigte Ablauf ist wie folgt [9]. Zuerst soll die Impulsantwort des Filters im Ortsbereich in Matlab ermittelt werden, z. B. mit der Funktion `fspecial`. Dann müssen daraus die Frequenzbereichskoeffizienten errechnet werden. Man muss erwähnen, dass es ein Fehler wäre, dafür die Matlab-Funkton `freqz2` zu verwenden, weil sie den *geshifteten* 2D-Frequenzgang zurückgibt, die 2D-DFT auf dem FPGA liegt aber nicht geshiftet vor. Statt dessen soll die Impulsantwort analog zu der bekannten Bezirhung

$$h[n] \circ\!\!-\!\!\bullet H(e^{j\omega})$$

zum Frequenzgang einfach mit `fft2` transformiert werden, die dann als die komplexe Koeffizienten-Matrix H benutzt wird.

Die Filterung erfolgt dann als das *Hadamard-Produkt* der 2D-DFT von Sensordaten und der Koeffizienten-Matrix (der 2D-Frequenzgang), bei dem die Elemente an der gleichen Position miteinander paarweise ausmultipliziert werden.

$$\underline{F} = \underline{X_{2D}} \circ \underline{H} \tag{4.1}$$

4.1.2 Implementierung und Verifikation in VHDL

Die Implementation der Formel 4.1 in VHDL ist ziemlich geradlinig. Man muss einfach konsekutiv die Ergebnisse von DFT und die Fulterkoeffizientenn aus dem Speicher lesen, miteinander multiplizieren und des Resultat zurückschreiben. Für eine kürzere Bearbeitungsdauer wäre es einerseits vorteilhaft, die Filterkoeffizienten lokal im ROM zu speichern, damit man sie simultan mit 2D-DFT-Daten laden könnte. Daüber hinaus könnten die Koeffizienten im für sie optimalen Fixkommaformat gespeichert werden. Anderseits wäre es für Testzwecken besser die Koeffizienten in dem von Matlab zugänglichen System-RAM zu halten, damit man verschiedene Koeffizienten-Matrizen ohne eine erneute Synthese erproben könnte. Dieses Dilemma ist im Modul so gelöst, dass für beide Fälle eine Architektur (jeweils `COEF_ROM` und `COEF_RAM`) implementiert ist. So lässt es sich, je nachdem was aktuell gewünscht wird, über die explizite Architekturbindung (ähnlich wie in Abschnitt 3.3.1.2) auswählen.

Die beiden Architekturen (Abbildungen A.30 und A.22) sind sehr ähnliche Moore-Automaten bis auf den zusätzlichen Zustand in `COEF_RAM`, der die Koeffizienten aus der RAM in die lokale Register lädt. Sie iterieren durch den Speicher, multiplizieren die Koeffizienten mit den 2D-DFT-Daten aus, und schreiben die Ergebnis zurück in die RAM. Da Speicherbereiche parallel durchgelaufen werden, wird nur einen einzigen Adresse-Iterator verwendet, was Register einspart. Der Zugriff auf die andere Bereich wird aus ihren Startadressen errechnet, die durch die generischen Parameter einstellbar sind. Darüber hinaus werden für die beiden Phasen der Multiplikation dieselben Register wiederverwendet. Die Architekturen sind somit nach dem Ressourcenverbrauch optimiert.

Die Architektur `COEF_ROM` instantiiert das ROM-Modul `filter_coef`, das mit der Matlab-Funktion `writeFilterCoef` erzeugt werden kann. Ergänzend zum Modul schreibt sie auch das Package `filter_pkg` mit Typdefinitionen in die selbe Datei `filter_coef.vhd`, die dann von `filter_2d` eingesetzt werden können. Die Funktion akzeptiert die Koeffizienten-Matrizen sowohl bereits im Fixkommaformat (als `fi`), als auch roh im Fließkomma. Im letzten Fall wird sie die Koeffizienten in die Fixkommazahlen der anzugebende Breite umwandeln.

Die beiden Architekturen wurden simuliert und getestet. Die Simulation zeigte 322 Taktzyklen für `COEF_ROM` und 386 Taktzyklen für `COEF_RAM`. Die praktische Versuche auf dem FPGA wurden mit den Sensordaten durchgeführt.

Zuerst wurden die 2D-DFT und -Filterung berechnet. Daraufhin wurden die Kalkulationsergebnisse aus dem FPGA geholt. Dann wurde die gefilterten Daten von den in Matlab elementweise ausmultiplizierten Koeffizienten und DFT-Daten subtrahiert und davon der Maximalwert ermittelt.

Für die Tests wurden die Koeffizienten entsprechend in das RAM geladen oder mitsynthetisiert. Zu der Zeit von dieser Arbeit wurden weder Koeffizienten noch der Filtertyp festgelegt. Als Testkoeffizienten wurde zum einen der ideale Bandpass verwendet.

$$H_{ideal} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Da diese Koeffizienten so trivial sind, ergab sich die perfekte Differenz von Null, was für einen realistischen Test nachteilig ist. Daher wurde es auch mit den zufällig generierten komplexen Koeffizienten getestet. Dieser Versuch zeigte die Höchstabweichung von $3{,}05 \cdot 10^{-5}$ (halbes LSB) über alle Sensordaten, was für die Fixkommarechnung den Bestfall darstellt. Das Modul wurde nicht in Cadence implementiert, weil seinen Beitrag zur Logik als relativ gering zu erwarten ist.

4.2 Implementierung im Ortsbereich über die zweidimensionale Faltung

4.2.1 Die 2D-Faltung

Die Filterung wird häufig in Filterung wird häufig im Ortsfrequenzbereich durchgeführt, weil es unter dem Einsatz der Algorithmen für DFT wie FFT schneller als im Ortsbereich ausgeführt werden kann. Diese Bedingung wird aber hier nicht erfüllt. Daraus entsteht die Frage, ob die Filterung im Ortsbereich über die

2D-Faltung doch schneller und ressourcenschonender als die Verarbeitungskette DFT→Filter→iDFT sein könnte. Um sich ein Bild davon zu machen, soll nun die zweidimensionale Faltung implementiert werden.

Dabei werden die Sensordaten s mit der Impulsantwort des Filters [21] in der Form einer *Faltungsmatrix* h [24], auch *Filterkern* genannt [25], gefaltet. Der mathematische Ausdruck für die gefilterten Daten

$$f[n_1, n_2] = s[n_1, n_2] * h[n_1, n_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} s[k_1, k_2] h[n_1 - k_1, n_2 - k_2] \quad (4.2)$$

Das kann so verstanden werden, dass die um 180° rotierte Faltungsmatrix durch die Daten durchgeschoben wird [22]. In der Praxis andersherum realisiert. Die Datenmatrix wird durch die Mitte der ebenfalls rotierten Faltungsmatrix geshiftet, wobei die überlappende Elemente paarweise ausmultipliziert und die entstandene Produkte aufsummiert werden [23, 25]. Dies ist erlaubt, weil die Faltung kommutativ ist:

$$\sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} s[k_1, k_2] h[n_1 - k_1, n_2 - k_2] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} h[k_1, k_2] s[n_1 - k_1, n_2 - k_2] \quad (4.3)$$

Dafür wäre es günstig, dass die Faltungsmatrix eine definierte Mitte hätte. Demzufolge werden meistens ungerade dimensionierte Faltungsmatrizen verwendet [25].

Noch eine Komplikation mit der Filterung im Ortsfrequenzbereich ist, dass die Multiplikation im Ortsfrequenzbereich der *zyklischen* Faltung im Ortsbereich, wodurch die Ränder der Sensormatrix sich nach der inversen 2D-DFT zusammen verwischen könnten. Eine Maßnahme dagegen ist Zero-Padding der Daten. Die vorgestellte Faltungsmethode impliziert Zero-Padding von allein durch die Tatsache, dass nur die überlappende Werte bei der Rechnung berücksichtigt werden [22]. Dies ist ein weiterer wesentlicher Vorteil dieser Faltungsmethode über die Filterung im Ortsfrequenzbereich.

4.2.2 Ausarbeitung des Ansatzes in Matlab

Die Implementierung der 2D-Faltung ist ziemlich komplex, da sowohl der Überlappungsbereich als auch die Anzahl der überlappenden Elemente variabel

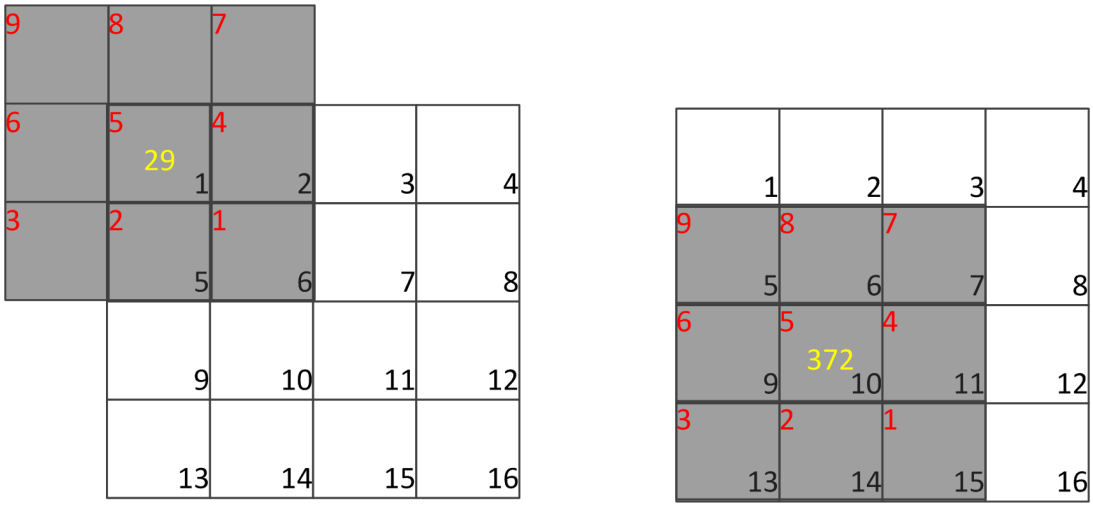

Abbildung 4.1: Veranschaulichung der zweidimensionalen Faltung anhand der 4×4 Daten und der 3×3 Faltungsmatrix.
Links ist die Faltungsmatrix um Element 1 zentriert. Der Faltungswert an der Stelle (gelb) ergibt sich aus den überlappenden roten (Faltungsmatrix) und schwarzen Zahlen (Daten):
$29 = 5 \cdot 1 + 4 \cdot 2 + 2 \cdot 5 + 1 \cdot 6$
Rechts ist die Faltungsmatrix um das Element 10 zentriert. Der Faltungswert an dieser Stelle entsprechend:
$372 = 9 \cdot 5 + 8 \cdot 6 + 7 \cdot 7 + 6 \cdot 9 + 5 \cdot 10 + 4 \cdot 11 + 3 \cdot 13 + 2 \cdot 14 + 1 \cdot 15$

sind. Es ist obendrein nicht trivial ,die richtige Adressierung für die im eindimensionalen Speicher liegenden Matrizen zu erreichen. Um darüber eine Vorstellung zu erlangen, wurde die 2D-Faltung mit der eindimensionalen Adressierung in Matlab realisiert, was das Debuggen erleichtert und den direkten Vergleich mit der Matlab-Funktion `conv2` erlaubt. Dies erfolgte als Klasse `Kernel` in Fließkomma, weil dabei nur die Adressierung von Bedeutung war.

Die Klasse bewahrt intern die Faltungsmatrix als Vektor auf. Ebenso formt die Funktion `convFPGA` als erstes die Eingangsdaten in den Vektor um. Als nächster Schritt muss der Bereich der Überlappung bestimmt werden. Vor allem die Startadressen dieses Bereichs und die Anzahl der überlappenden Spalten müssen ermittelt werden, um durch den Bereich iterieren zu können. Weiter wird die Adressierung anhand der Abbildung 4.2 erläutert.

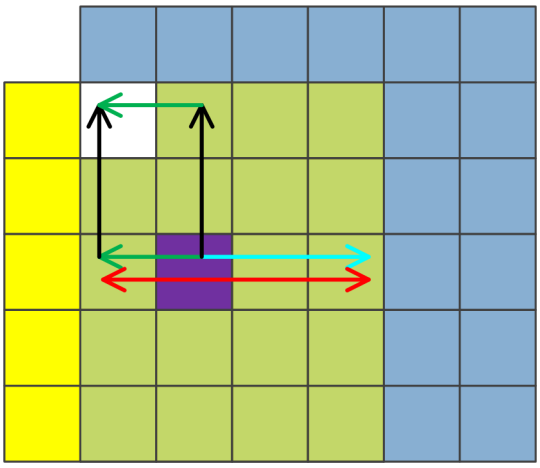

Abbildung 4.2: Illustration zur Adressierung bei der 2D-Faltung.
Gelb: Faltungsmatrix
Blau: Datenmatrix
Weiß: Faltungsbeginn
Violet: Position der Zentrierung
Grün: Abstand von der Position der Zentrierung zum linken Rand des Überlappungsbereichs
Schwarz: Abstand von der Position der Zentrierung zum oberen Rand des Überlappungsbereichs
Türkis: Abstand von der Position der Zentrierung zum rechten Rand des Überlappungsbereichs
Rot: Anzahl der überlappenden Spalten

Die Position, um die die Faltungsmatrix aktuell zentriert ist (violetter Kasten), ist bei der Faltungsmatrix (gelb) stets die Mitte, und bei der Datenmatrix (blau) muss sie mittels sowohl Reihen- als auch Spalteniteratoren verfolgt werden. Daraus muss die Speicheradresse der Position des Faltungsbeginns (weißer Kasten) ermittelt werden, die aus den Matrizenindexen dieser Position berechnet werden kann. Hierfür müssen diese Indexe ihrerseits gefunden werden. Da die Indexe nicht über *beide* Matrizen hinauslaufen dürfen, bekommt man die Abstände von der Position der Zentrierung zu den Rändern des Überlappungsbereichs (grüner und schwarzer Pfeil) als kleinste Werte der jeweiligen Indexe der Position der Zentrierung unter den Matrizen (violetter Kasten). Nach der Subtraktion dieser Abstände von der Position der Zentrierung, erhält man die gesuchten Indexe der Startposition. Ähnlich wird auch der Abstand zum rechten Rand (türkiser Pfeil) gefunden. Die Summe aus den Abständen zum linken und rechten Rand ergibt die Anzahl der überlappenden Spalten.

Die Startadressen des Überlappungsbereichs für die Faltungsmatrix und die Datenmatrix im Speicher werden wie folgt berechnet:

$$\text{Reihenindex} \times \text{Größe der Matrix} + \text{Spaltenindex}$$

Mit diesen Werten werden die Speicheriteratoren initialisiert, die die Faltungsmatrix und die Datenmatrix im Überlappungsbereich durchlaufen. Die Iteratoren werden inkrementiert bis sie den rechten Rand des Überlappungsbereichs erreichen, wonach mithilfe von den nunmehr bekannten Staradressen und Anzahl der überlappenden Spalten geprüft werden kann. Ist dies der Fall, wird zu der nächsten Reihe übergangen, indem die Startadressen um die Größe der jeweiligen Matrizen inkrementiert werden und diese Werte werden den Speicheriteratoren wieder zugewiesen. Dabei wird auch geprüft ob die letzte Reihe des Überlappungsbereichs erreicht wurde, worauf das Iterieren abgebrochen wird, und der akkumulierte Wert gerundet in das RAM geschrieben wird. Darauf folgend werden die Spaltungs- und Reiheniteratoren inkrementiert und das ganze Verfahren wird wiederholt, bis die ganze Datenmatrix schließlich abgearbeitet ist.

Die beschriebene Vorgehensweise wurde in der Funktion `convFPGA` der Klasse `Kernel` implementiert. Sie wurde mit den zufälligen Faltungs- und komplexen Datenmatrizen getestet. Es wurde die Differenz zu `conv2` gebildet. Die Abweichung war in der vernachlässigbaren Größenordnung von 10^{-14}. Beispielausgabe (man achte auf den Vorfaktor 10^{-14}):

```
1 dif =
```

```
1.0e-14 *
Columns 1 through 4
-0.0222 - 0.0444i   0.0333 + 0.0111i  -0.0222 + 0.0888i   0.0111 + 0.0000i
-0.0222 - 0.0222i   0.0333 - 0.0888i  -0.0222 + 0.0056i   0.0444 - 0.0444i
0.0000 + 0.0444i   0.0444 + 0.0333i   0.0444 - 0.0194i  -0.0444 - 0.0111i
-0.0888 + 0.0222i   0.0888 + 0.0888i  -0.0305 - 0.0222i  -0.0222 - 0.0999i
-0.0222 + 0.0333i   0.0666 - 0.0111i   0.0000 - 0.0444i   0.0000 + 0.0888i
-0.0111 + 0.0000i  -0.0222 + 0.0222i  -0.0222 + 0.0000i   0.0000 + 0.0555i
0.0000 + 0.0056i  -0.0333 - 0.0222i   0.0222 - 0.1110i  -0.1776 - 0.0444i
0.0111 + 0.0444i   0.0000 - 0.0028i   0.0888 + 0.0000i   0.0000 - 0.0222i
Columns 5 through 8
-0.0222 + 0.0444i   0.0000 + 0.0278i  -0.0444 + 0.0444i   0.0444 + 0.0000i
0.0028 - 0.1776i   0.0000 - 0.0444i   0.0444 + 0.0000i  -0.0056 - 0.0222i
0.0000 - 0.0222i   0.0444 - 0.0333i   0.0666 + 0.0389i  -0.0056 + 0.0333i
-0.0333 - 0.1776i   0.0444 + 0.0888i   0.0167 + 0.0222i  -0.0222 + 0.0056i
-0.0444 + 0.0666i   0.0000 - 0.0555i   0.0444 - 0.0333i  -0.0111 + 0.0111i
-0.0888 - 0.0333i  -0.0222 + 0.0000i  -0.0222 + 0.0000i  -0.0222 - 0.0333i
0.0000 - 0.0444i  -0.0222 - 0.0444i   0.0444 + 0.0000i   0.0222 + 0.0444i
0.0000 - 0.0222i  -0.0444 + 0.0000i   0.0000 - 0.0222i   0.0222 + 0.0222i
```

4.2.3 Implementierung und Verifikation in VHDL

Aus dieselben Erwägungen wie im Abschnitt 4.1.2 wurde das Modul `filter_conv` mit zwei Architekturen realisiert, eine, die die Faltungsmatrix in einem ROM aufbewahrt (`CONV_ROM`, Abbildung A.48), und die zweite, die die Faltungsmatrix aus der RAM lädt (`CONV_RAM`, Abbildung A.39). Die Rotation der Faltungsmatrix bringt keinen Nachteil, weil dies vorsorglich schon in Matlab gemacht werden kann.

Das ROM-Modul `kernel_coef` mit der Faltungsmatrix kann mit der Matlab-Funktion `writeKernel` erzeugt werden. Da bei einigen Filtertypen (z. B. Gaußfilter) die Werte in den äußeren Reihen und Spalten sehr klein sind und dann bei Konversion zu den Fixkommazahlen zu 0 werden, hat stellt die Funktion die Option zur Verfügung, die 0-wertigen Reihen und Spalten zu entfernen. Falls die Faltungsmatrix noch nicht in einem Fixkommaformat, wird sie die Koeffizienten in die Fixkommazahlen der anzugebende Breite umwandeln. Ergänzend zum Modul schreibt sie auch das Package `kernel_pkg` in die selbe Datei `kernel_pkg.vhd`. Das Package definiert das Koefiziententyp und die vorgerechnete Konstanten, die bei der Adressierung benutzt werden.

Die beiden Architekturen sind sehr ähnliche Moore-Automaten, die aus Matlab mit Anpassungen portiert wurden. Da die Faltungsmatrix und die Datenmatrix unterschidliche Größe haben und zueinander verschoben sind, braucht

man für sie zwei eigene Zusammenstellungen von Iteratoren. Wie im Abschnitt 4.2.2 erläutert wird, ist die Ermittlung der Position des Faltungsbeginns ziemlich kompliziert, deswegen wurde dafür einen dedizierten Zustand CALC_START vorgesehen. Dabei tritt zwar die Multiplikation auf, sie ist jedoch mit der Konstante, so dass dafür kein zusätzlicher Multiplizierer (DSP48 Block) synthetisiert wird. Da man die allererste Position des Faltungsbeginns aus den Matrizengrößen berechnen kann, werden die Iteratoren im Voraus mit den Passenden Zahlen im Zustand IDLE initialisiert, und CALC_START wird beim ersten Faltungswert übersprungen. Dann werden die überlappende Untermatrizen durchgegangen, und die paarweise Produkte akkumuliert. In den beiden Architekturen wurde der Akkumulator in die DSP48 Blöcke absorbiert.

Die beiden Architekturen wurden simuliert (s. Anhang B.1.5). Es wurden sowohl 7×7 als auch 5×5 Faltungsmatrizen probiert, weil es, wie schon erläutert passieren könnte, dass die äußeren Reihen und Spalten zu 0 werden. Die Dauer der 2D-Faltungsberechnung ist in Tabelle 4.1 präsentiert. Wie man aus

Tabelle 4.1: Dauer der 2D-Faltungsberechnung in Taktzyklen bei verschiedenen Architekturen und Größen der Faltungsmatrix

Größe der Faltungsmatrix	CONV_ROM	CONV_RAM
7×7	2193	4129
5×5	1413	2569

der Tabelle 4.1 sehen kann, nimmt die Faltung teilweise weniger Zeit als die 2D-DFT alleine (2438 Taktzyklen). Es ist also auf jeden Fall wert, diese Option weiter zu betrachten. Solches Resultat entstand vor allem dadurch, dass alle Koeffizienten reell sind, und somit nur einen Takt für die Multiplikation gebraucht wird. Desto mehr fällt das in der Architektur CONV_RAM bei jeder Multiplikation nötige Auslesen von Koeffizienten ins Gewicht. Darüber hinaus spielt auch eine Rolle, das nur die überlappenden Elemente bearbeitet werden.

Das Modul wurde mithilfe des Testsystems von Matlab mit Sensordaten geprüft. Da keine Koeffizienten vorliegen, wurden sie zufällig generiert. Dabei könnte es aber passieren, dass die Werte sich erhöhen (Verstärkung) und Überläufe verursachen. Üblicherweise wird die Faltungsmatrix durch die Summe der Elemente geteilt, um dies zu vermeiden. Dadurch wird allerdings lediglich der Mittelwert normalisiert [26], was die Überläufe nicht ganz ausschließt. Um sie vollständig zu eliminieren, wurde die zufällige Faltungsmatrix erstmal

mit der mit Einsen befüllten Matrix mittels `conv2` gefaltet und durch das höchste resultierende Element geteilt. Nach dieser Methode wurden die beiden Architekturen verifiziert und die höchste Abweichung über alle Sensordaten betrug $3{,}05 \cdot 10^{-5}$ (0,5 LSB).

Zum Vergleich des Logikverbrauchs auf einem ASIC wurde das Modul einzeln in Cadence Genus synthetisiert. Die ergebene Anzahl an Standardzellen:

- Architektur `CONV_ROM`: 1759 Standardzellen
- Architektur `CONV_RAM`: 1720 Standardzellen

Der Logikaufwand ist also vergleichbar mit der 2D-DFT (Tabelle 3.7). Er ist größer bei `CONV_ROM`, weil die Faltungsmatrix mit drin ist.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

In dieser Masterarbeit wurden Verarbeitungsodule mit der zweidimensionalen komplexen DFT und Filterung in VHDL entwickelt und verifiziert und das Testsystem, das ihrer Verifikation dient, verbessert.

Als erstes wurde das Testsystem, bestehend aus dem Zedboard mit Zynq FPGA, einem Mikrocontroller-Board und dem PC mit Matlab in Betrieb genommen. Auf dem FPGA dient das RAM als die Schnittstelle sowohl zwischen Modulen als auch nach außen. Das Zedboard kommuniziert mittels eines parallelen Interfaces über zwei Flachbandkabeln mit dem Mikrocontroller-Board. Der Mikrocontroller, der an den PC über die serielle UART angebunden ist, verbindet den PC und mit dem Zedboard.

Dabei wurden aber Möglichkeiten zur Vereinfachung erkannt und ausgenutzt. Statt die Verarbeitungsmodule vom RAM mittels Tri-State-Puffer zu trennen, wurde eine zentrale Arbitrierung realisiert. Ferner wurde die Zwei-Flanken-Operation zugunsten der exklusiven Operation auf der steigende Flanke eliminiert. Dafür wurden die einheitliche Interfaces für den Speicherzugriff durch die Verarbeitungsmodule und ihrer Steuerung definiert.

Auch die Operation auf dem Mikrocontroller wurde durch Umstellung der Hauptfunktonalität von Interrupt auf Polling-Betrieb wurde beachtlich vereinfacht. Darüber hinaus wurde anstatt der Kommunikation über den UART mit Zeichenketten, ein effizientes binäres Protokoll entwickelt, das in der Lage ist, die auf dem FPGA erzeugten Daten bitgenau an das PC zu übetragen. Ferner wurde die Pinbelegung für die Flachbandkabel optimiert.

Auf dem PC wurde die Matlab-Klasse `ProcSysProtocol` erstellt, die die einfachen und bequeme Steuerung vom FPGA und dem Datenaustausch mit ihm ermöglicht. Insgesamt haben alle diese Maßnahmen die spätere Entwicklung und Verifikation bedeutend erleichtert.

Dann erfolgte die Realisierung der 8×8 2D-DFT über die Multiplikation der Matrizen, weil diese am einfachsten zu implementieren ist. Zunächst erfolgte jedoch die Implementierung in Matlab zwecks der genaueren Ausarbeitung des Algorithmus als auch der Untersuchung der numerischen Eigenschaften in Hinsicht sowohl der Größenordnung der Werte als auch der Genauigkeitsanforderungen. Es wurden dabei verschiedene Bitbreiten und Varianten von Implementierung sowohl von Twiddle-Faktoren als auch der gesamten 2D-DFT untersucht. Die dadurch gewonnene Erkenntnisse wurden zugrunde der VHDL-Implementierung gelegt.

Danach erfolgten die Versuche auf dem FPGA hauptsächlich anhand der realen Daten vom Sensorarray. Es wurden verschiedene Implementierungsvarianten mit Fixkommaformaten getestet. Bewertungskriterien waren die höchste numerische Abweichungen, die Abweichungen des Argumentes der komplexen Zahlen und die euklidische Distanz von den mit der Matlab Funktion `fft2` berechneten Werten. Während die numerischen Abweichungen und die euklidische Distanz sehr klein in der Größenordnung von 10^{-5}-10^{-4} ausfielen, was für eine sehr gute Rechengenauigkeit spricht, wurden die rechnerischen Abweichungen des Argumentes bis zu 360° unnormal groß. Diese übergroße Abweichungen wurden separat untersucht, erläutert und korrigiert.

Da das 2D-DFT-Modul mühelos skalierbar ist, wurden die Sensordaten in Matlab auf 15×15 linear interpoliert und die Versuche für diese Größe wiederholt. Die Resultate waren ähnlich gut wie bei der 8×8 2D-DFT. Anschließend erfolgte die Synthese in Cadence Genus und das Layout in Cadence Innovus. Mit ca 1800 Standardzellen und ca 0.5 mm^2 Fläche kann das Design als sehr kompakt bewertet werden, besonders im Vergleich zu den Vorgängerarbeiten und zu dem vorgegebenen Budget von 10000 Standardzellen (s. Abbildung 5.1). Auf der anderen Seite übersteigt die Berechnungsdauer von 2436 Taktzyklen deutlich die angestrebten 1000 Taktzyklen. Diese Konstellation ist aber völlig in Ordnung, da die Einfachheit der Logik und der geringe Flächenbedarf in dieser Arbeit absichtlich vor der Ausführungsdauer priorisiert wurden.

Als nächstes wurde die 2D-Filterung im Ortsfrequenzbereich implementiert, und zwar mit zwei Architekturen. Die erste lädt die Filterkoeffizienten aus dem

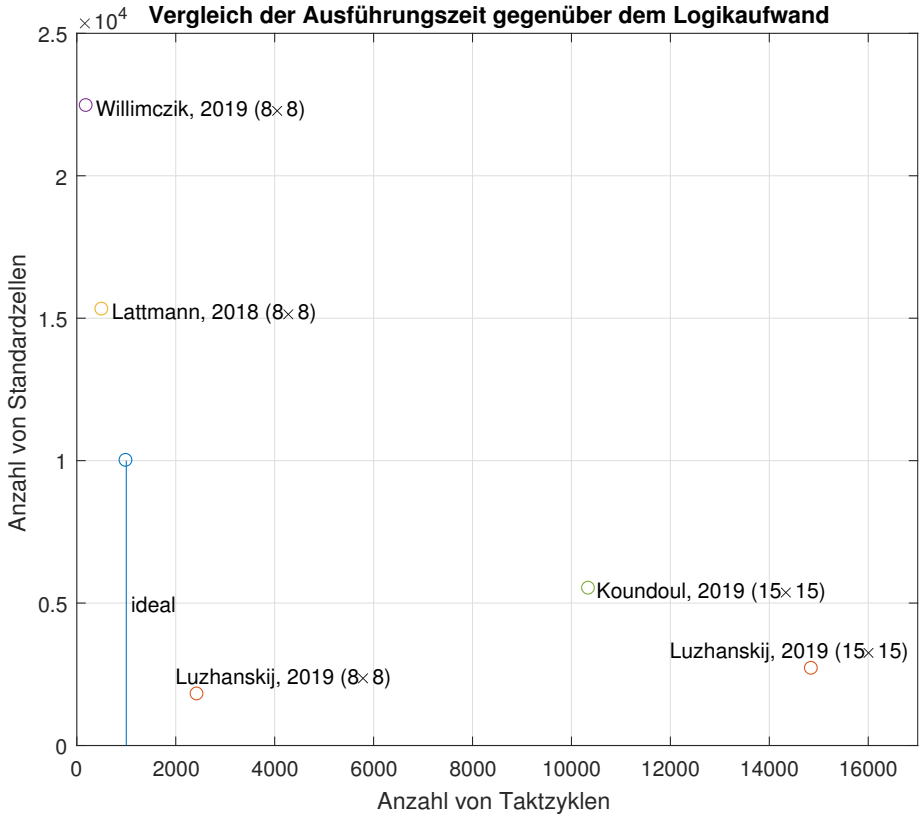

Abbildung 5.1: Vergleich der Ausführungszeit in Taktzyklen gegenüber dem Logikaufwand in Standardzellen im Idealfall und bei verschiedenen Arbeiten. Der Idealfall ist als eine senkrechte Linie dargestellt, mit einer maximalen Anzahl von 10000 Standardzellen und 1000 Taktzyklen.

RAM, was für die Untersuchung bequemer ist. Die zweite speichert die Filterkoeffizienten in einem ROM, wodurch sie etwas schneller ist, vor allem für die spätere Implementierung auf einem ASIC. Die beiden Architekturen erreichten die bestmögliche Abweichung von der Berechnung in Matlab von 0,5 LSB.

Es könnte allerdings sein, dass die Kette 2D-DFT→Filterung→inverse 2D-DFT von der Rechengeschwindigkeit her nicht die beste Variante ist. Überdies kann die 2D-Filterung im Ortsfrequenzbereich nach der Rücktransformation Randeffekte verursachen, weil sie im Ortsbereich der *zyklischen* Faltung entspricht. Aus diesen Gründen wurde zusätzlich die 2D-Filterung über 2D-Faltung ausprobiert, die die ganze Kette ersetzen könnte. Sie wurde ähnlich wie die 2D-Filterung im Ortsfrequenzbereich mit zwei Architekturen implementiert, wobei die ROM-Version sich diesmal viel schneller als die RAM-Variante gezeigt hat. Diese Version kam mit 2193 Taktzyklen sogar deutlich unter die Ausführungszeit der 2D-DFT. Sie wurde auch in Cadence Genus synthetisiert. Die daraus resultierende 1730 Standardzellen, die auch mit der 2D-DFT vergleichbar. Die Genauigkeit war mit einer Abweichung 0.5 LSB zur Matlab-Funktion `conv2` ebenfalls hervorragend.

Es wurden also alle Punkte der Aufgabenstellung erfolgreich gelöst:

- Das Testsystem wurde vereinfacht und vor allem die Handhabung verbessert.
- Es wurde eine sehr flächeneffeziente 2D-DFT entwickelt, die dann gründlich sowohl in 8×8 als auch 15×15 Größe getestet und in der Cadence Toolchain implementiert wurde.
- Es wurde die 2D-Filterung sowohl im Ortsfrequenzbereich als auch über die 2D-Faltung implementiert. Die beiden Versionen haben unterschiedliche Architekturen, die zum einen für den Einsatz auf einem ASIC und zum anderen für den Testbetrieb auf dem FPGA optimiert sind.

5.2 Ausblick

5.2.1 2D-Filterung

Die größte Baustelle ist wohl die 2D-Filterung. Hier müssen noch die theoretischen und empirischen Studien zur Auswahl des Filtertyps und der Koeffizien-

ten durchgeführt werden. Außerdem müssen noch die Randeffekte aufgrund der Filterung im Ortsfrequenzbereich untersucht werden, um zu bestimmen, inwieweit sie ein Problem darstellen. Eventuell muss dann an den Gegenmaßnahmen gearbeitet werden, wie das Zero-Padding bei der 2D-DFT oder die Filterung über Faltung.

Da die letzte sich in ersten Versuchen bewährt hat, lohnt es sich auf jeden Fall, sie weiterhin zum Vergleich mit der Filterung im Ortsfrequenzbereich zu untersuchen. Da die Filterkoeffizienten nicht feststanden, wurde die Faltung in dieser Arbeit für eine allgemeine Faltungsmatrix implementiert. Besonders interessant könnte diese Methode werden, falls die Faltungsmatrix *separierbar* würde. Das würde bedeuten, dass die Faltungsmatrix sich in einen Spalten- und einen Reihenvektor faktorisieren lässt.

$$h = \begin{pmatrix} A \cdot a & A \cdot b & A \cdot c \\ B \cdot a & B \cdot b & B \cdot c \\ C \cdot a & C \cdot b & C \cdot c \end{pmatrix} = \begin{pmatrix} A \\ B \\ C \end{pmatrix} \cdot \begin{pmatrix} a & b & c \end{pmatrix}$$

Dann reduziert sich die 2D-Faltung auf die konsekutive Faltung mit den beiden Vektoren [21]:

$$f = s * h = \left[s * \begin{pmatrix} A \\ B \\ C \end{pmatrix} \right] * \begin{pmatrix} a & b & c \end{pmatrix}$$

Dies hat das Potenzial, den Zeitaufwand noch weiter zu senken, vielleicht sogar unter die anvisierten 1000 Taktzyklen.

5.2.2 2D-DFT

Es wurde zwar eine nicht so schnelle Version von der 2D-DFT implementiert, dafür aber eine sehr kompakte. Daher kann man sie als Grundlage für eine weitere Optimierungen nach Geschwindigkeit mit zusätzlichen Logikaufwand einsetzen. Eine einfache Verbesserung dürfte die Zusammenlegung mit der Filterung in dem Ortsfrequenzbereich sein. Dabei würden die fertig berechneten Elemente der 2D-DFT sofort mit dem entsprechenden Filterkoeffizient ausmultipliziert werden, was die Lese- und Schreibzyklen einsparen würde.

Eine weitere Möglichkeit bietet das Einfügen von zusätzlichen RAM-Blöcken zwischen verschiedenen Verarbeitungsphasen, so dass für die Eingangs- und Ausgangsdaten verschiedene Blöcke benutzt werden könnten. Damit ließe

sich eine Pipeline organisieren, bei der mehrere Verarbeitungsmodule gleichzeitig arbeiten. Auf der anderen Seite könnte dafür die Flächengröße eines RAM-Blocks hinderlich sein.

Eine Methode gegen die schon erwähnten Randeffekte ist das Zero-Padding bei der 2D-DFT. Dies kann mit einem relativ geringen Aufwand so implementiert werden, dass die Twiddle-Faktoren aus der größeren 2D-DFT entnommen werden. Für die 8×8 DFT könnte so der zentrale Ausschnitt aus der 16×16 Twiddlefaktor-Matrix genommen werden. Die Multiplikation mit ihr wäre äquivalent dem Zero-Padding.

Aus den aufgetretenen großen rechnerischen Abweichungen des Argumentes sollte man eine Lektion für die zukünftige Winkelberechnung lernen. Dabei sollen die Messwerte der Sensoren mittels eines noch zu bestimmenden Algorithmus auf einen Winkel reduziert werden. Da man dafür die Funktion “atan2” auf einer oder anderer Weise verwenden muss, kann es vorkommen, dass die verschiedene Sensoren des Arrays bei der Sensorlage nahe von 180° die Winkel mit einer großen rechnerische Differenz ausgeben (z.B. einer zerleg 179° un der andere -179°). Diese sollte man im Algorithmus auffangen und korrigieren. Vielleicht könnte man erreichen, diese Komplikation ganz zu vermeiden.

Literaturverzeichnis

[1] Helck, Jannes: *Digitale Signalverarbeitungs-Module für einen Chipentwurf für ein Sensor-Array.* HAW, 2018

[2] *AMS 0.35um Static RAM DATASHEET.* AMS

[3] : *ZedBoard (Zynq™Evaluation and Development) Hardware User's Guide.* Avnet, 2014 – URL http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf. – Zugriffsdatum: 04.12.2019

[4] *Tiva™ TM4C1294NCPDT Microcontroller: DATA SHEET.* Texas Instruments, 2014

[5] *TivaWare™ Peripheral Driver Library: USER'S GUIDE.* Texas Instruments, 2016

[6] *Tiva™ C Series TM4C1294 Connected LaunchPad Evaluation Kit: User's Guide.* Texas Instruments, 2016

[7] *Vivado Design Suite User Guide: Logic Simulation* Xilinx, 2018

[8] Willimczik, Martin: *Hardware Implementierung zur Ortsfrequenzberechnung für Sensorarrays* HAW, 2019

[9] Koundoul, Ada: *Signalverarbeitung für ein magnetisches Sensor-Array als digitaler Chipentwurf* HAW, 2018

[10] Lattmann, Thomas: *Chipimplementation einer zweidimensionalen Fouriertransformation für die Auswertung eines Sensor-Arrays* HAW, 2018

[11] Mehm, Thorbjörn: *Schaltungsentwurf und Mikrocontrollersteuerung für ein Tunnel-Magnetoresistives Sensor-Array* HAW, 2019

[12] *2D Discrete Fourier Transform (DFT).* – URL http://www.di.univr.it/documenti/OccorrenzaIns/matdid/matdid027832.pdf. – Zugriffsdatum: 12.11.2019

[13] Robert Fisher, Simon Perkins, Ashley Walker, Erik Wolfart: *Fourier Transform.* 2003 – URL https://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm. – Zugriffsdatum: 10.12.2019

[14] *Discrete Fourier Transform.* – URL https://en.wikibooks.org/wiki/Digital_Signal_Processing/Discrete_Fourier_Transform. – Zugriffsdatum: 13.11.2019

[15] Barry Van Veen *Matrix Interpretation of the DFT.* , 2013 – URL https://youtu.be/W5iXRA4ro1g. – Zugriffsdatum: 14.11.2019

[16] Bishop, David: *Fixed point package user's guide.*

[17] *Vivado Design Suite User Guide: Synthesis. UG901 (v2018.3)* Xilinx, 2018

[18] *UltraFast Design Methodology Guide for the Vivado Design Suite UG949 (v2018.2)* Xilinx, 2018

[19] *7 Series DSP48E1 Slice. User Guide UG479 (v1.10)* Xilinx, 2018 – URL https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf. – Zugriffsdatum: 06.11.2019

[20] *Vivado Design Suite User Guide. Logic Simulation. UG900 (v2018.2)* Xilinx, 2018

[21] Song Ho Ahn: *Convolution.* 2018 – URL http://www.songho.ca/dsp/convolution/convolution.html#convolution_2d. – Zugriffsdatum: 12.12.2019

[22] Sneha H.L.: *2D Convolution in Image Processing.* All About Circuits, 2018 – URL https://www.allaboutcircuits.com/technical-articles/two-dimensional-convolution-in-image-processing/. – Zugriffsdatum: 12.12.2019

[23] Song Ho Ahn: *Example of 2D Convolution.* 2018 – URL http://www.songho.ca/dsp/convolution/convolution2d_example.html. – Zugriffsdatum: 12.12.2019

[24] *Faltungsmatrix* Wikipedia – URL https://de.wikipedia.org/wiki/Faltungsmatrix. – Zugriffsdatum: 12.12.2019

[25] Thormählen, Thorsten: *Multimediale Signalverarbeitung Bildverarbeitung: Filter.* Universität Marburg, 2018 – URL https://www.mathematik.uni-marburg.de/~thormae/lectures/mmk/mmk_6_1_ger_web.html#1. – Zugriffsdatum: 12.12.2019

[26] *Kernel (image processing).* Wikipedia – URL https://en.wikipedia.org/wiki/Kernel_(image_processing)#Normalization. – Zugriffsdatum: 15.12.2019

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Zustands- und ASMD-Diagramme

A.1 Testsystem

A.1.1 RAM-Zugriffsverwaltung

Abbildung A.1: Zustandsdiagramm des Moduls `memory_access_ctrl`. Das Diagramm ist insoweit vereinfacht, in dem die Signale funktionsgemäß und nicht unter tatsächlichen Bezeichnungen dargestellt sind.

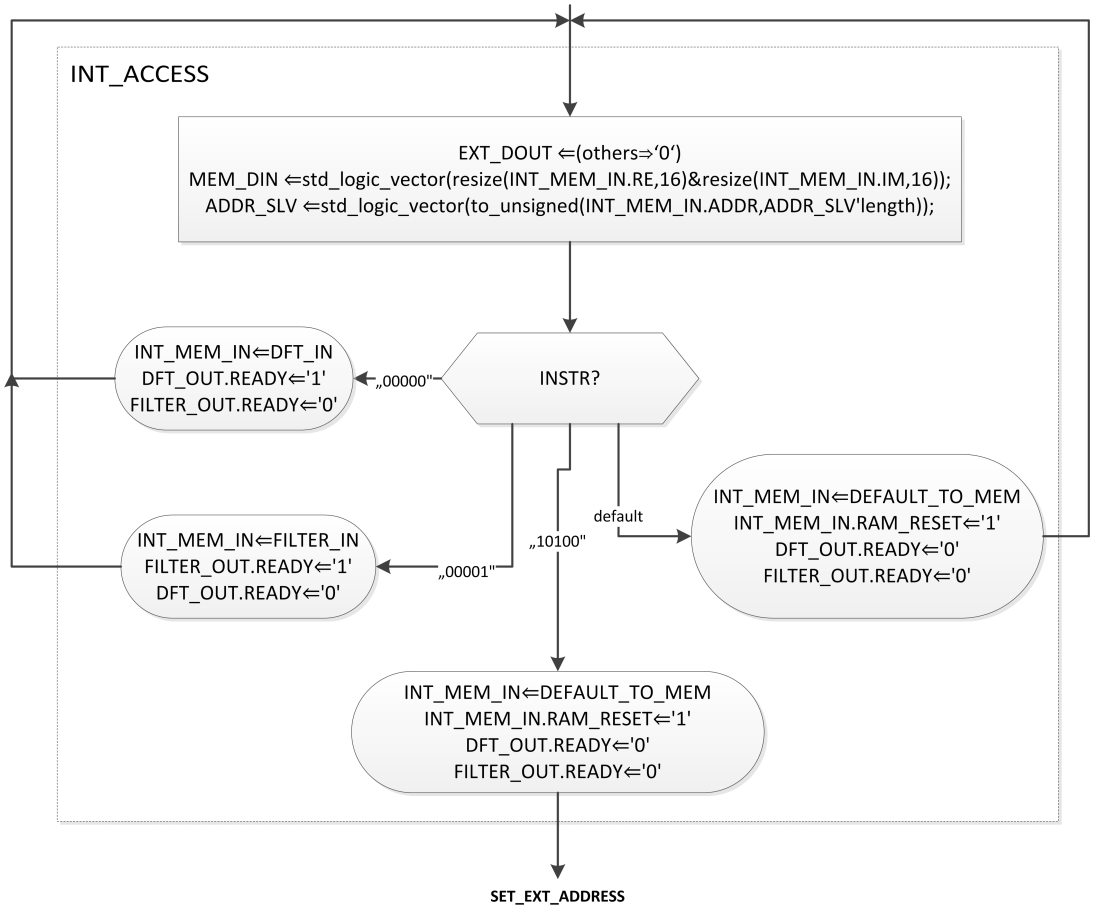

Abbildung A.2: ASMD-Diagramm des Zustands `INT_ACCESS` vom Modul `memory_access_ctrl`.

Abbildung A.3: ASMD-Diagramm des Zustands SET_EXT_ADDRESS vom Modul memory_access_ctrl.

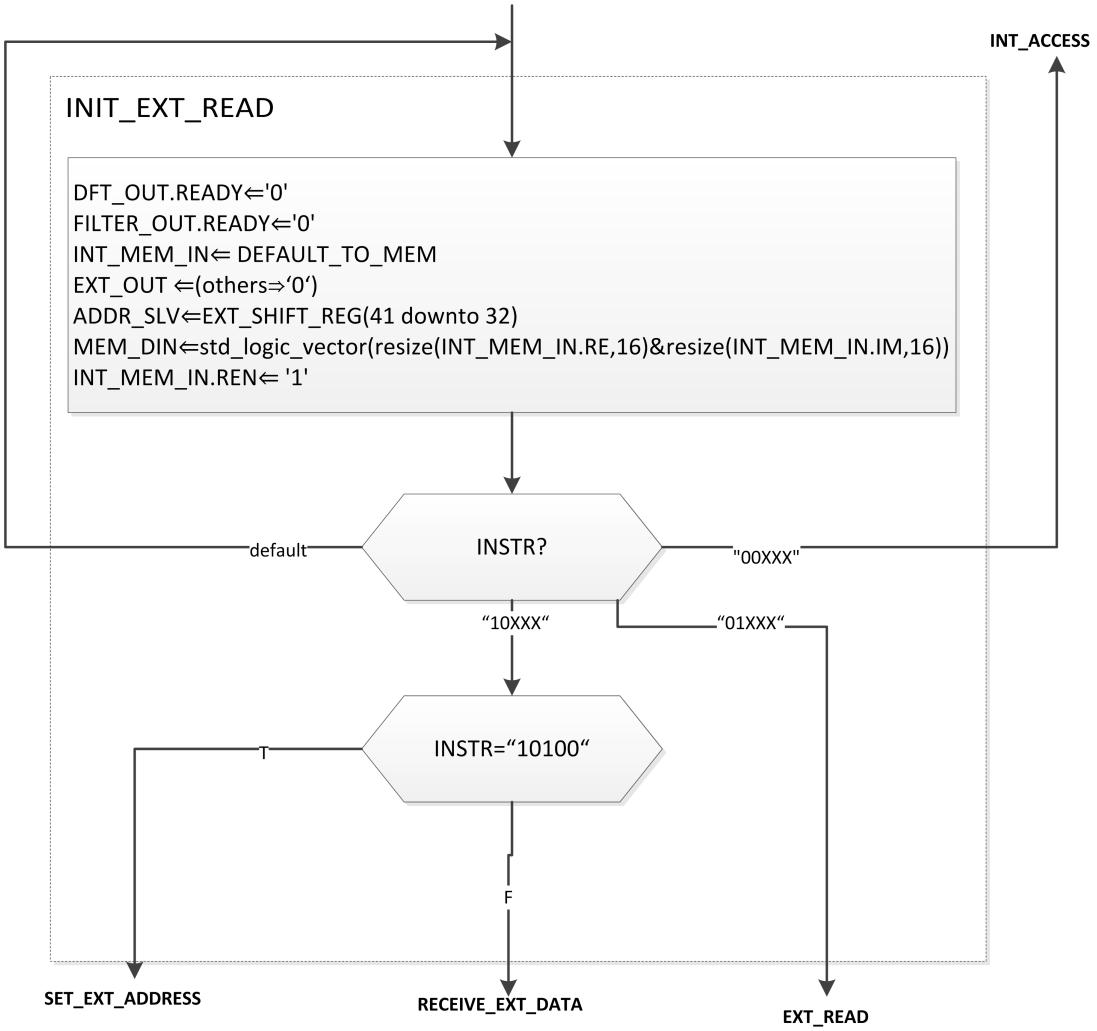

Abbildung A.4: ASMD-Diagramm des Zustands `INIT_EXT_READ` vom Modul `memory_access_ctrl`.

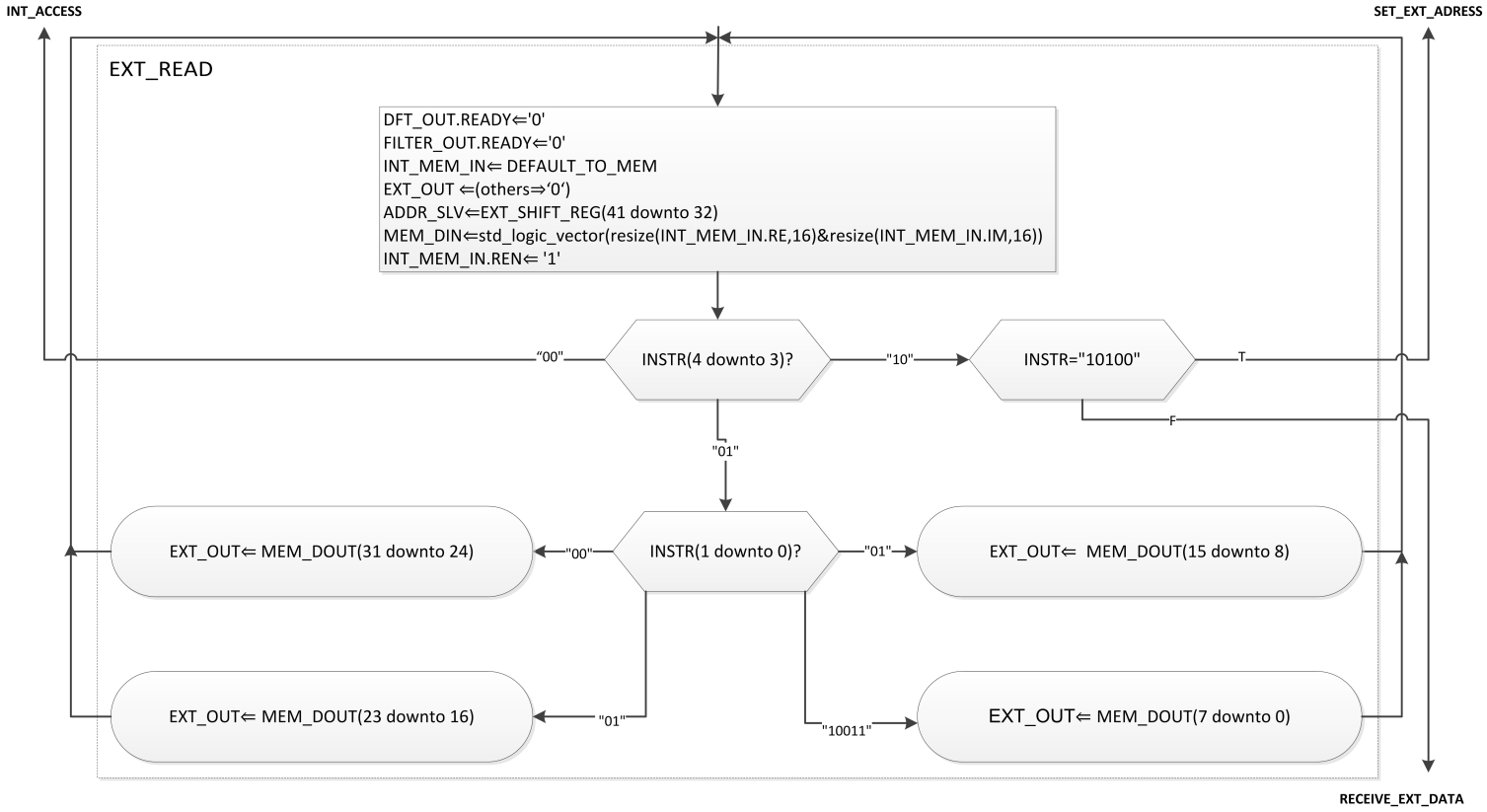

Abbildung A.5: ASMD-Diagramm des Zustands `EXT_READ` vom Modul `memory_access_ctrl`.

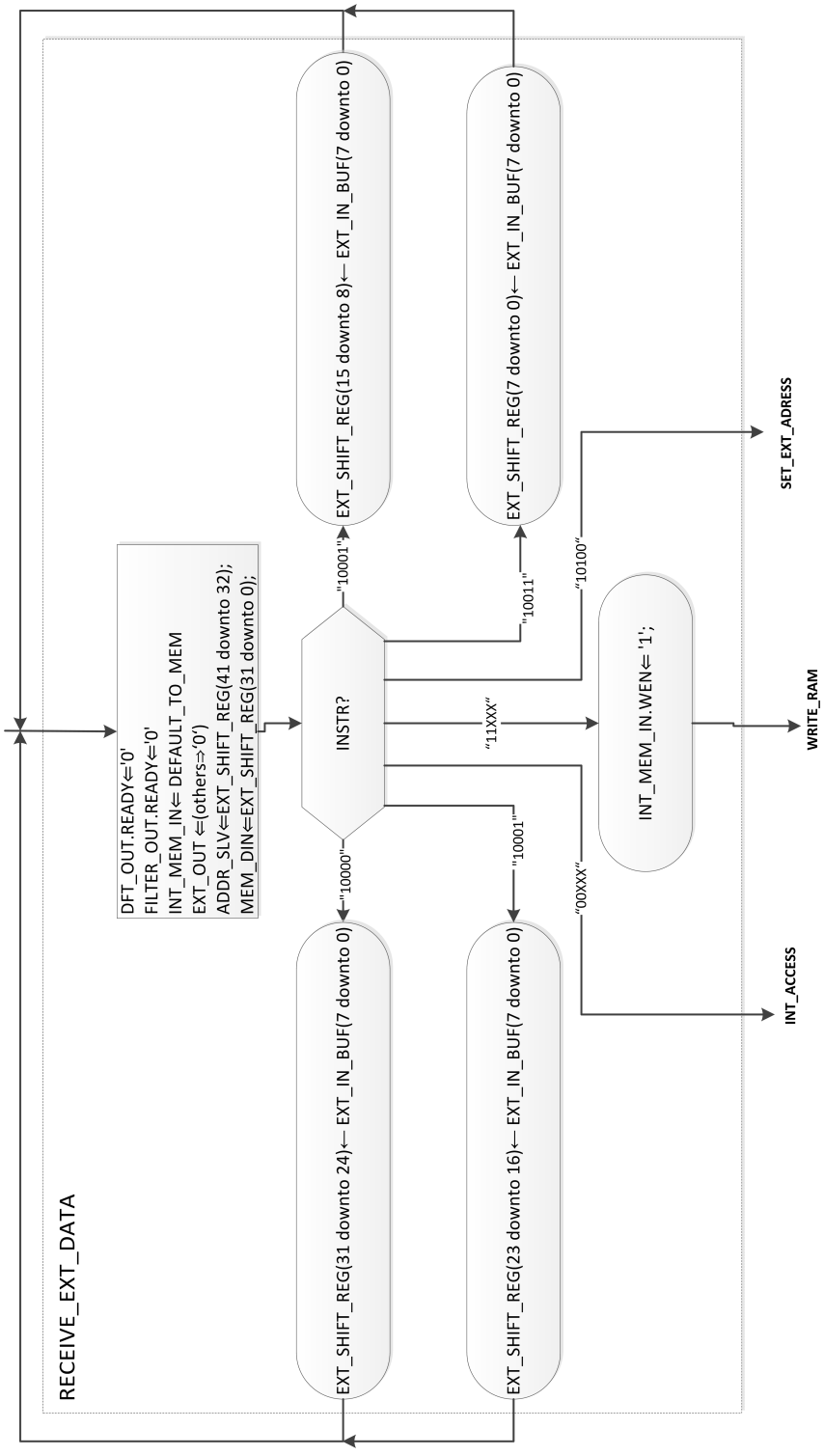

Abbildung A.6: ASMD-Diagramm des Zustands RECEIVE_EXT_DATA vom Modul memory_access_ctrl.

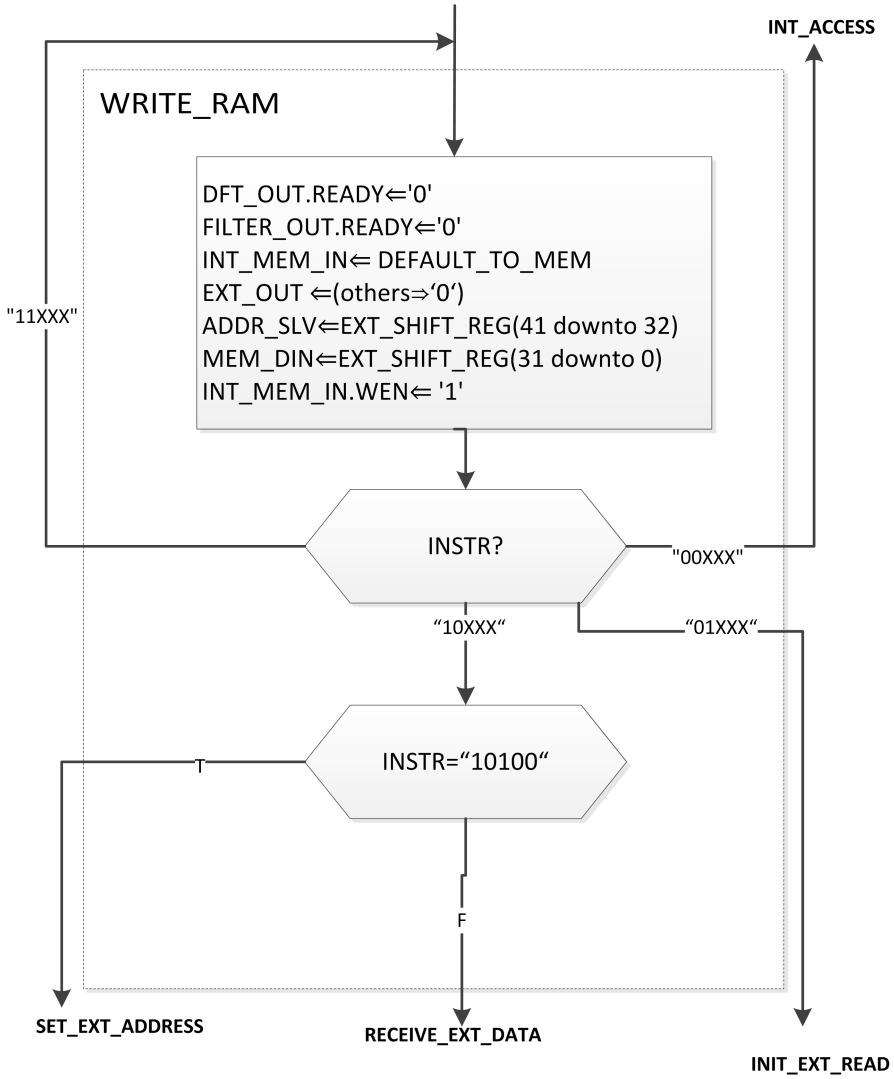

Abbildung A.7: ASMD-Diagramm des Zustands `WRITE_RAM` vom Modul `memory_access_ctrl`.

A.1.2 Modulsteuerung

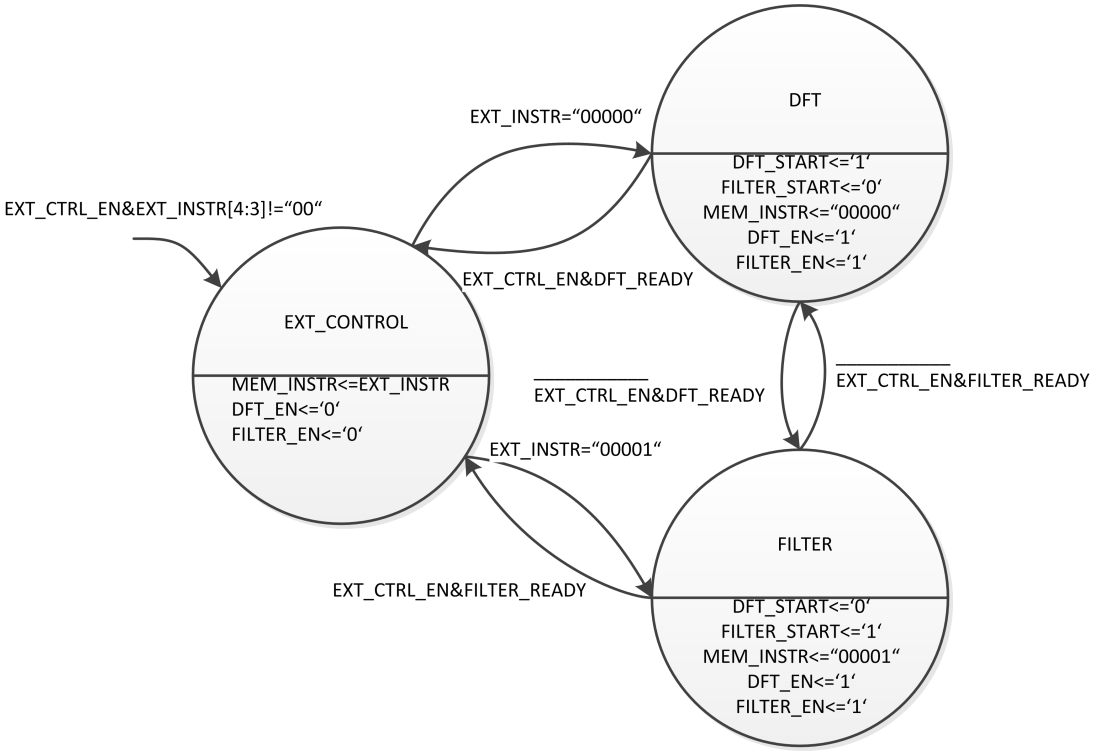

Abbildung A.8: Zustandsdiagramm des Moduls `module_controller`. Das Diagramm ist insoweit vereinfacht, in dem die Signale funktionsgemäß und nicht unter tatsächlichen Bezeichnungen dargestellt sind.

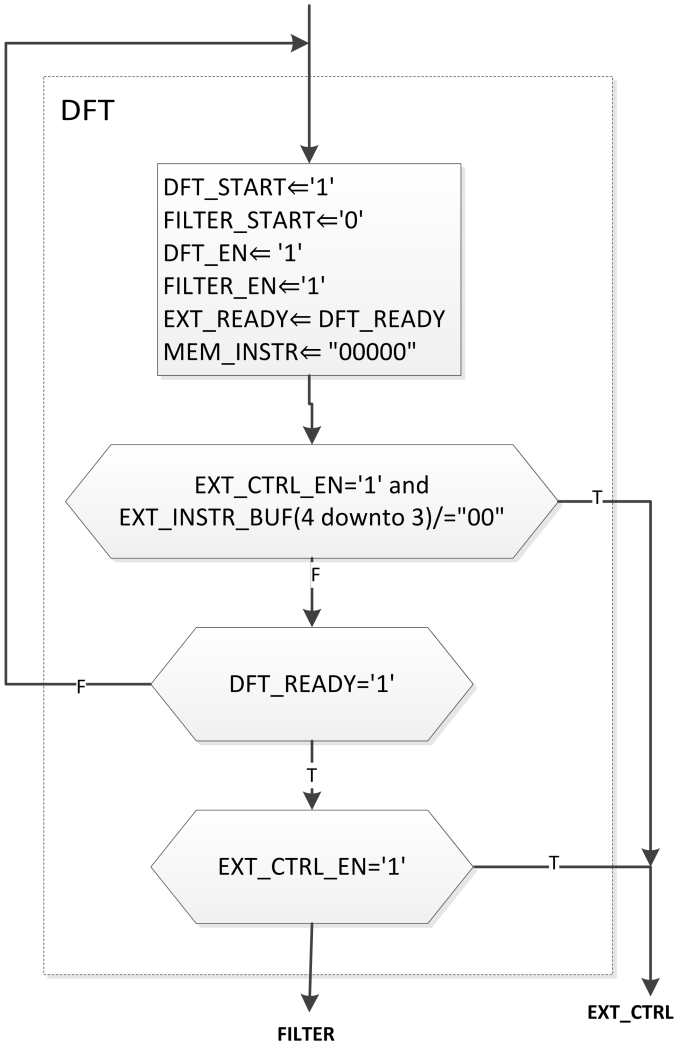

Abbildung A.9: ASMD-Diagramm des Zustands `DFT` vom Modul `module_controller`.

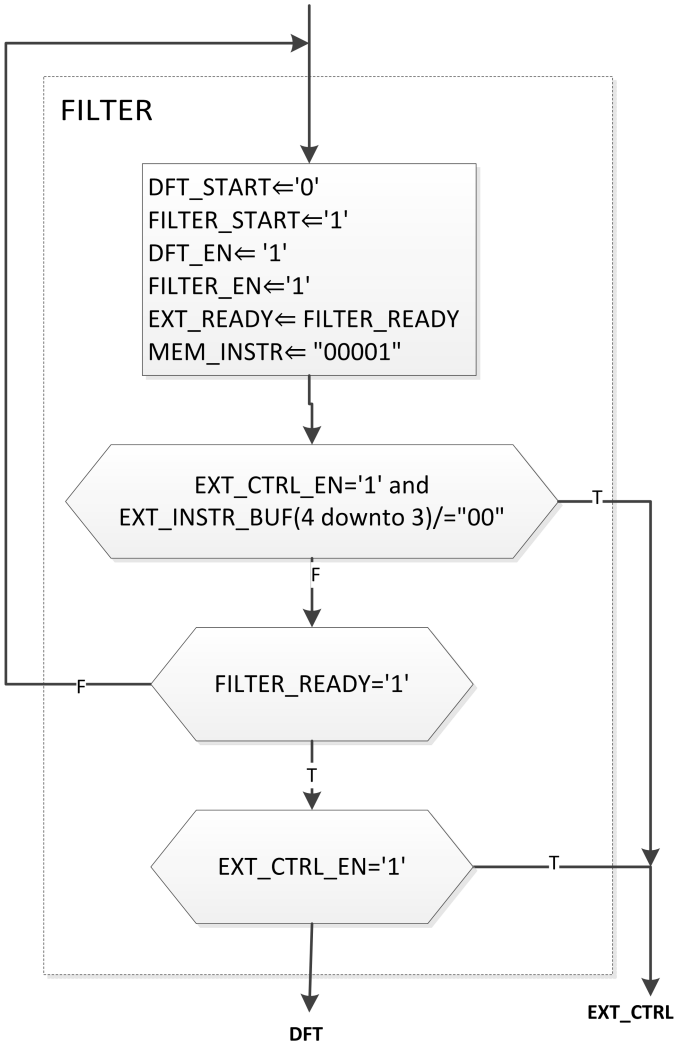

Abbildung A.10: ASMD-Diagramm des Zustands `FILTER` vom Modul `module_controller`.

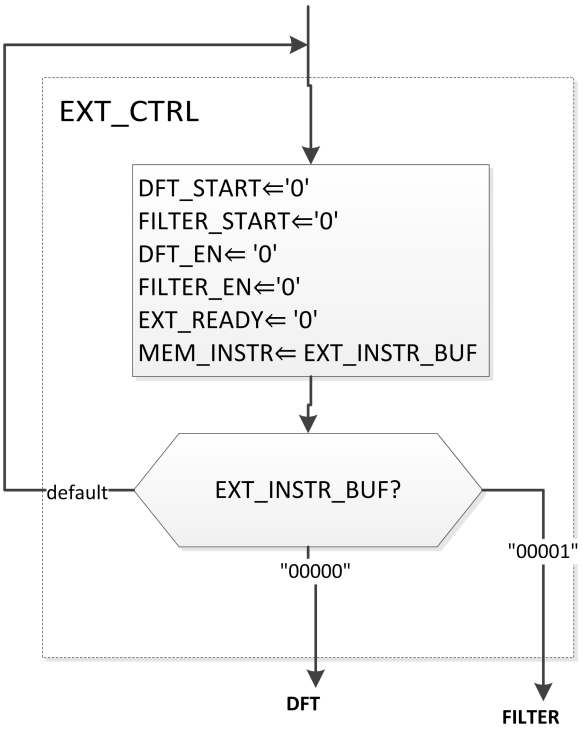

Abbildung A.11: ASMD-Diagramm des Zustands `EXT_CTRL` vom Modul `module_controller`.

A.2 2D-DFT

A.2.1 dft_mat_product

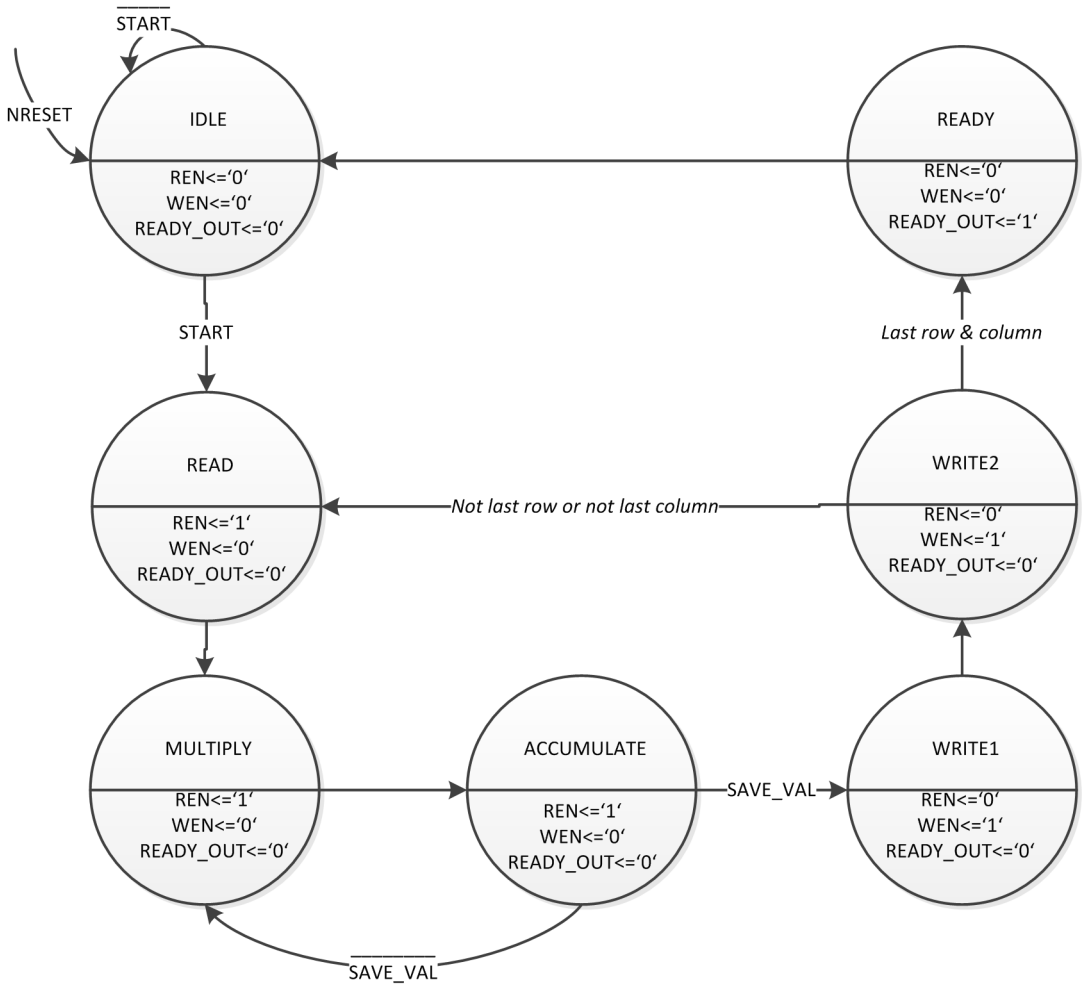

Abbildung A.12: Zustandsdiagramm des Moduls dft_mat_product.

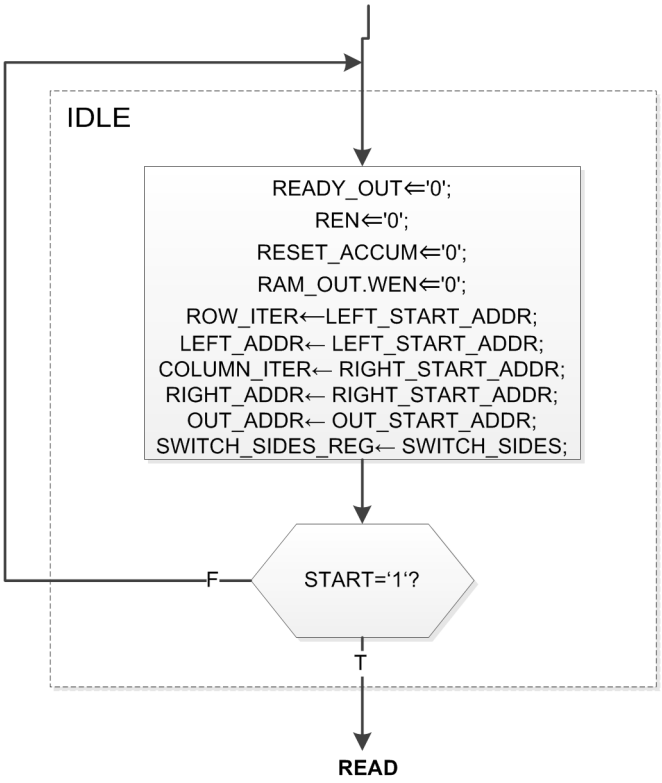

Abbildung A.13: ASMD-Diagramm des Zustands `IDLE` vom Modul `dft_mat_product`.

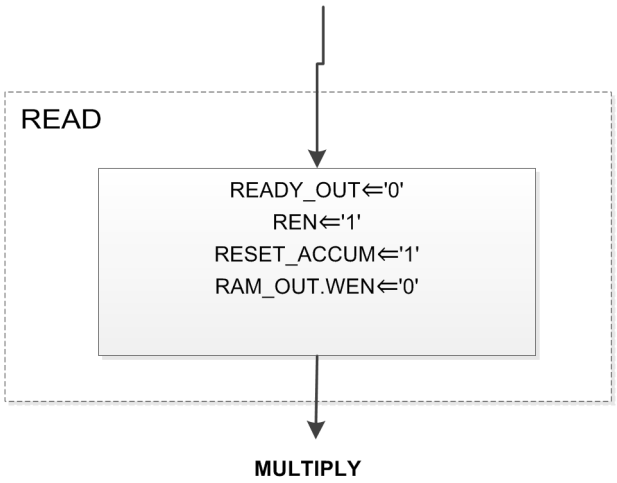

Abbildung A.14: ASMD-Diagramm des Zustands `READ` vom Modul `dft_mat_product`.

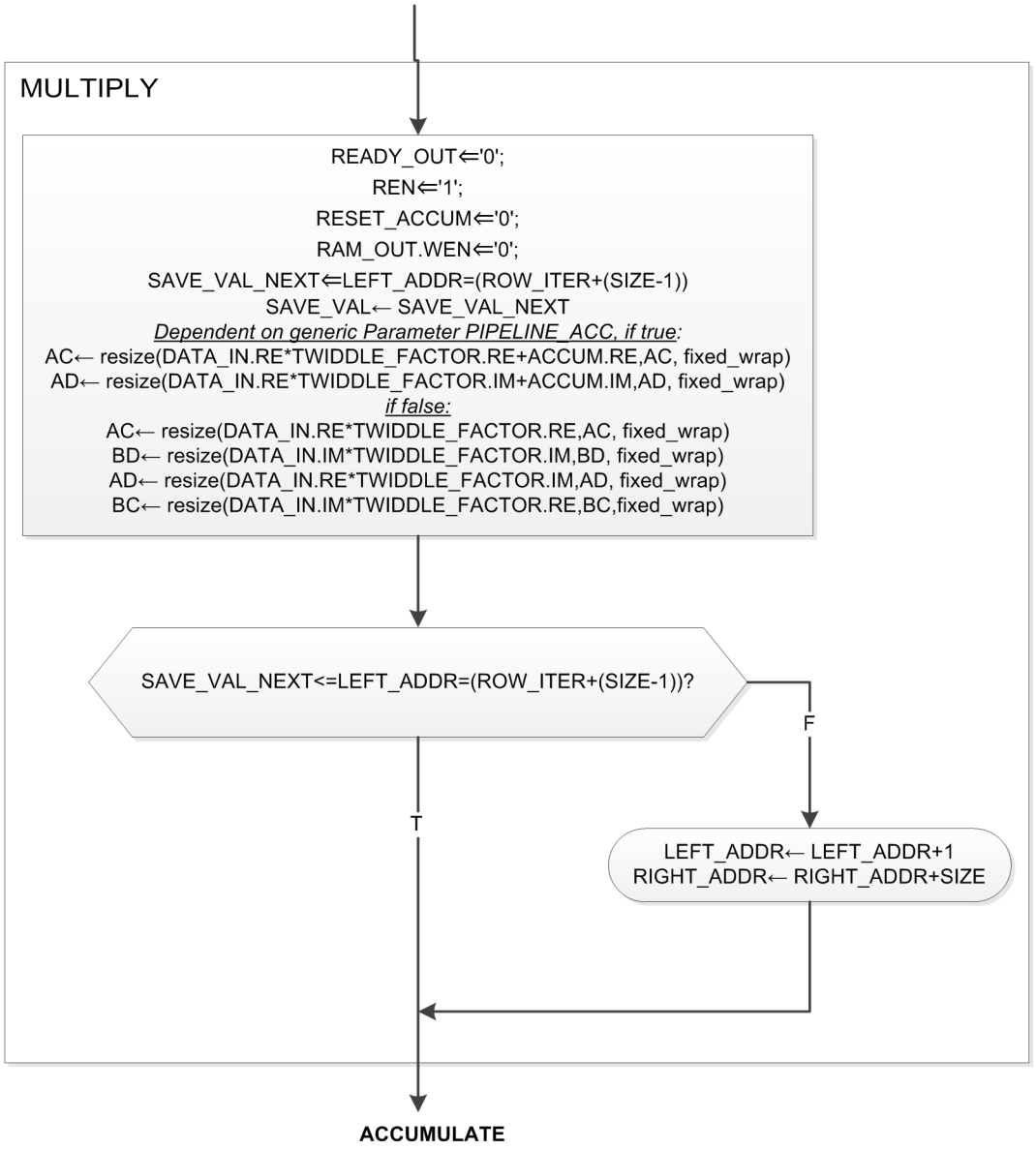

Abbildung A.15: ASMD-Diagramm des Zustands `MULTIPLY` vom Modul `dft_mat_product`.

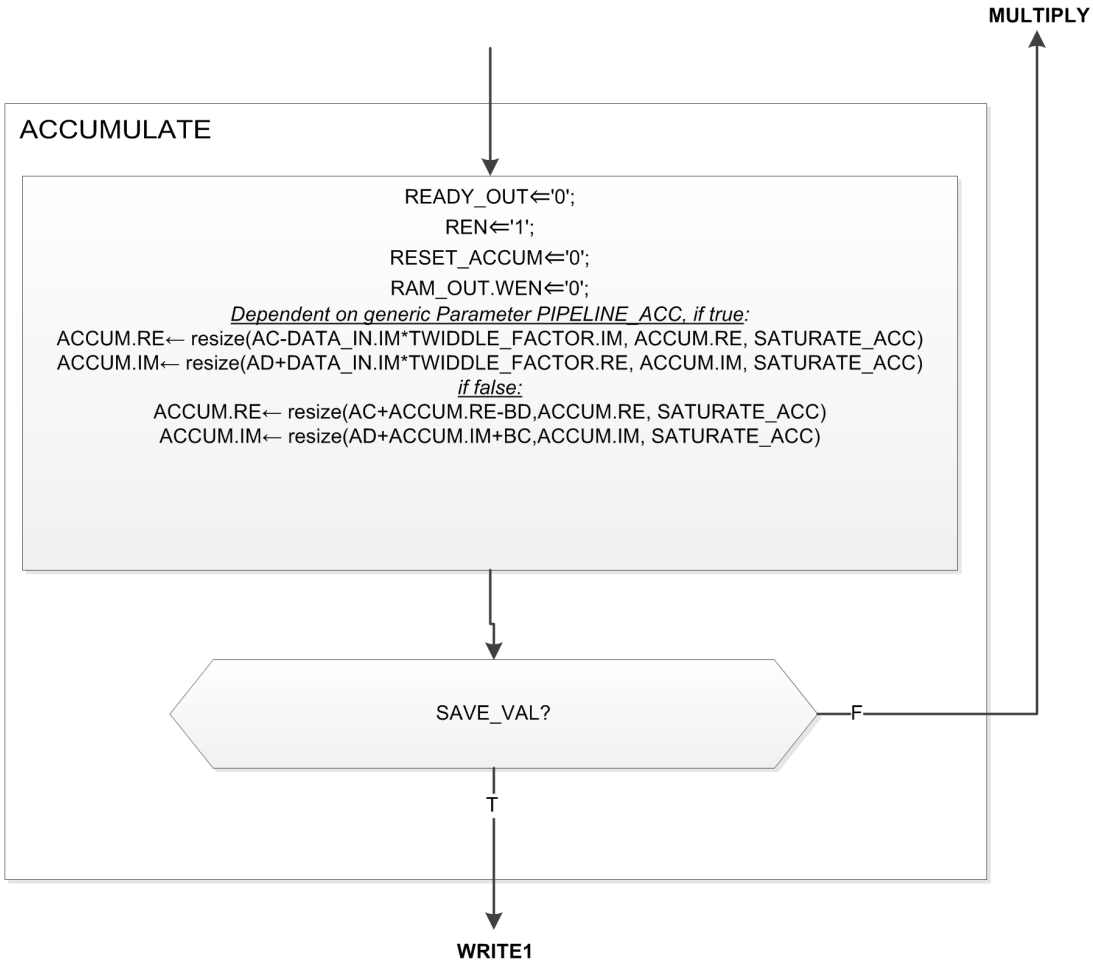

Abbildung A.16: ASMD-Diagramm des Zustands `ACCUMULATE` vom Modul `dft_mat_product`.

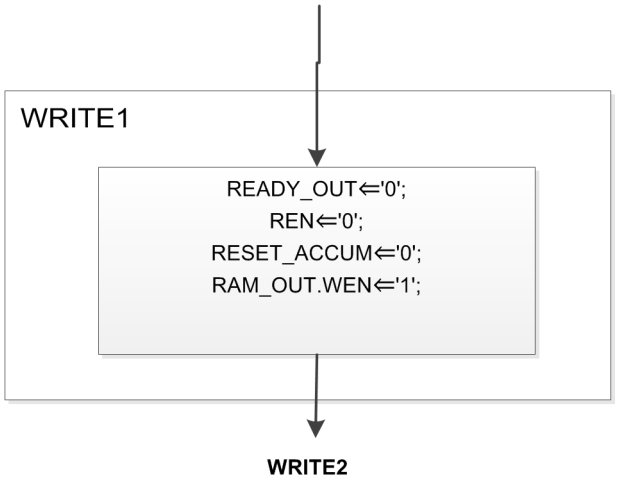

Abbildung A.17: ASMD-Diagramm des Zustands `WRITE1` vom Modul `dft_mat_product`.

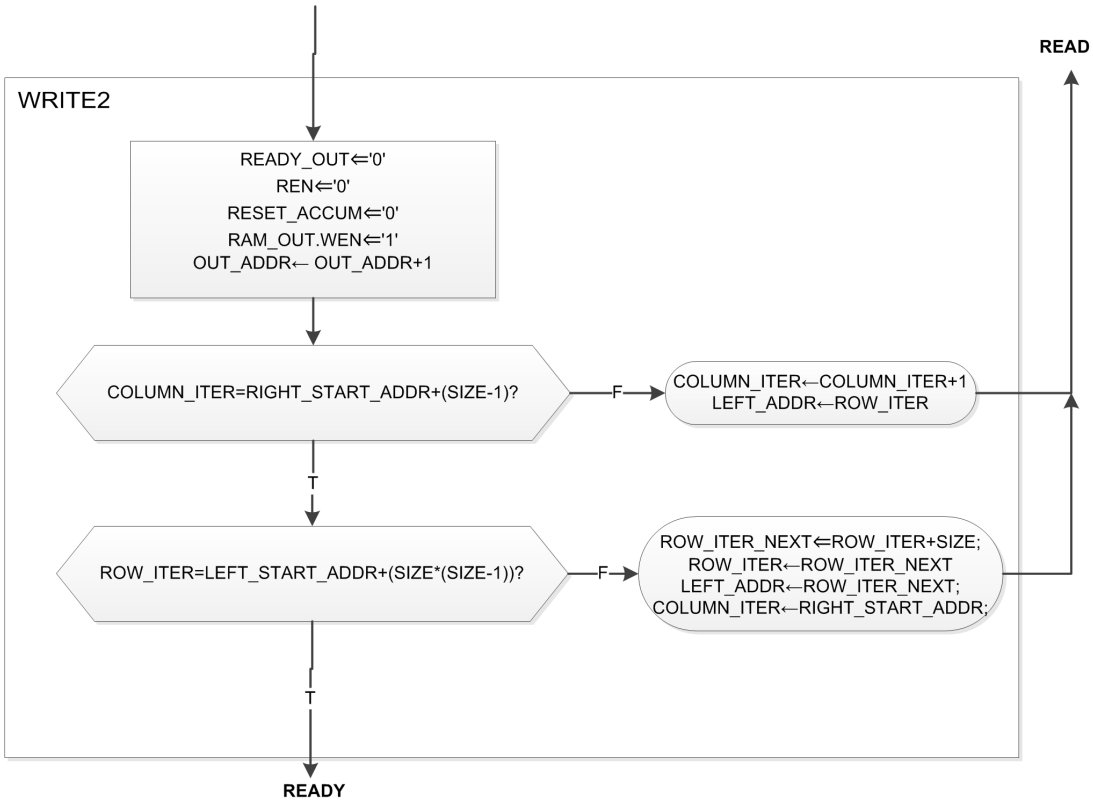

Abbildung A.18: ASMD-Diagramm des Zustands `WRITE2` vom Modul `dft_mat_product`.

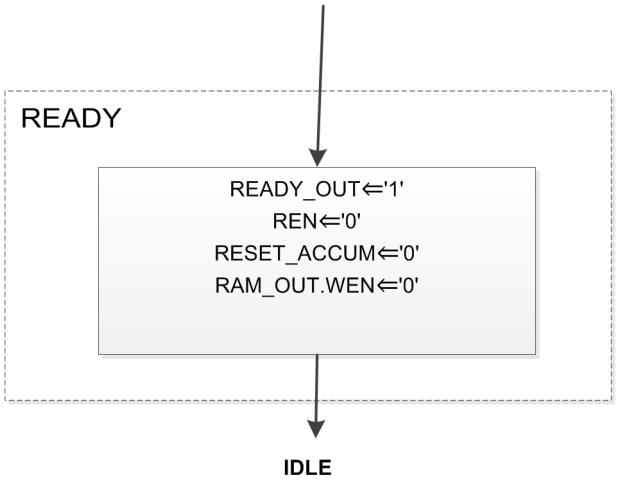

Abbildung A.19: ASMD-Diagramm des Zustands READY vom Modul dft_mat_product.

A.2.2 Modul dft_2d

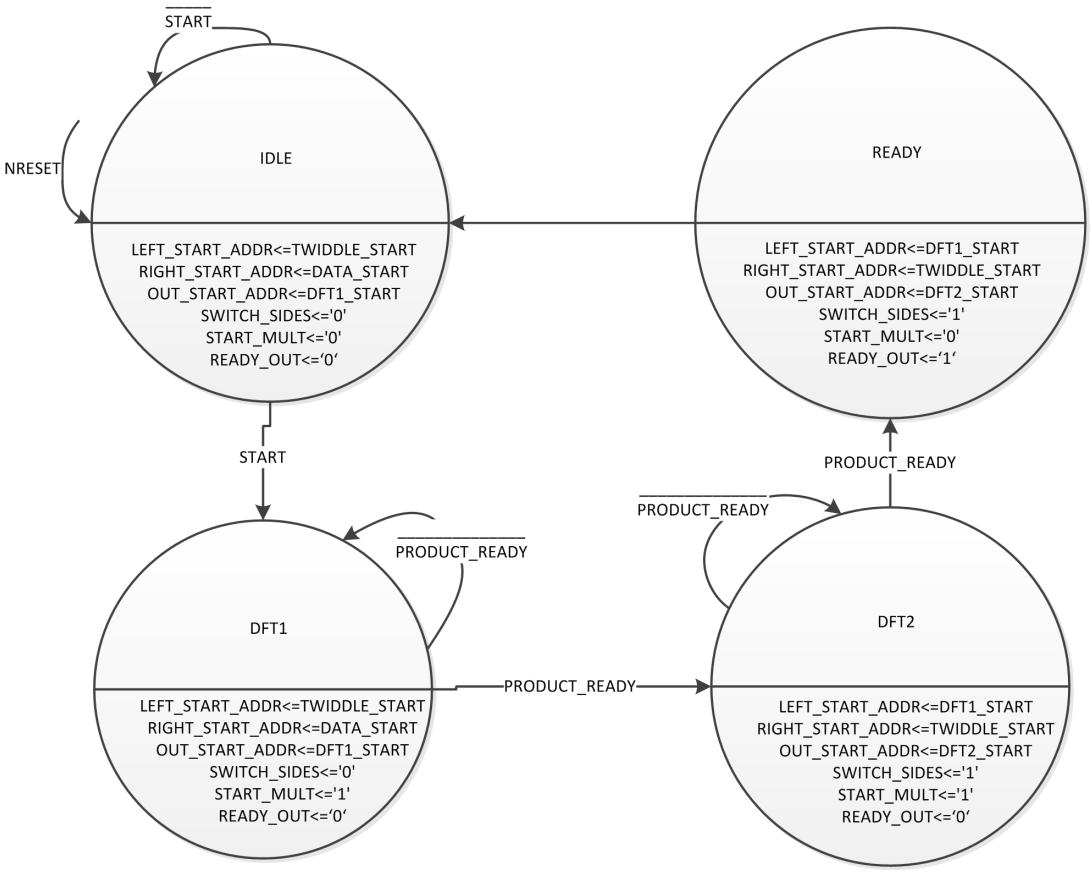

Abbildung A.20: Zustandsdiagramm des Moduls dft_2d.

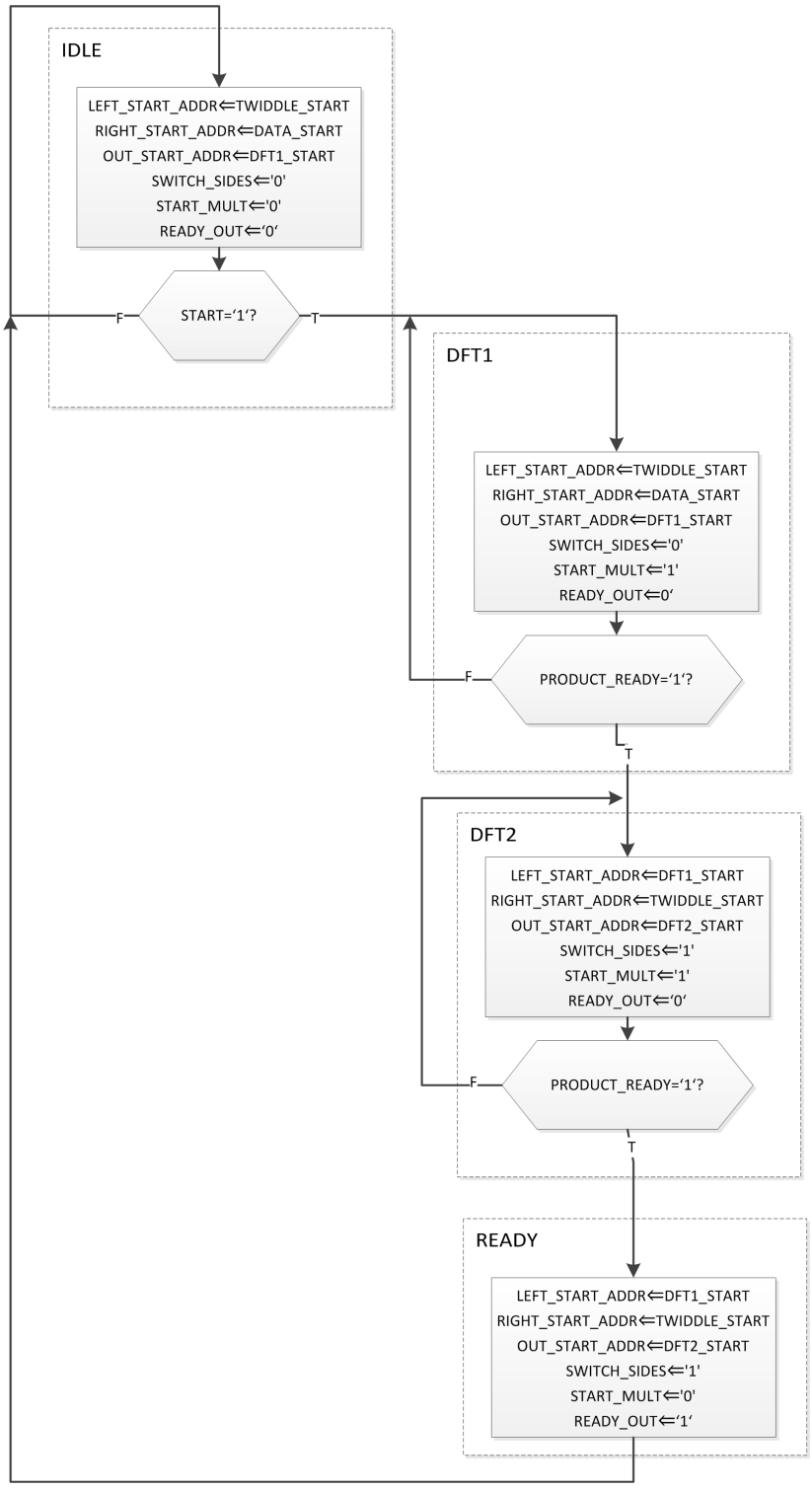

Abbildung A.21: ASMD-Diagramm des Moduls `dft_2d`.

A.3 Filterung im Frequenzbereich

A.3.1 Architektur COEF_ROM

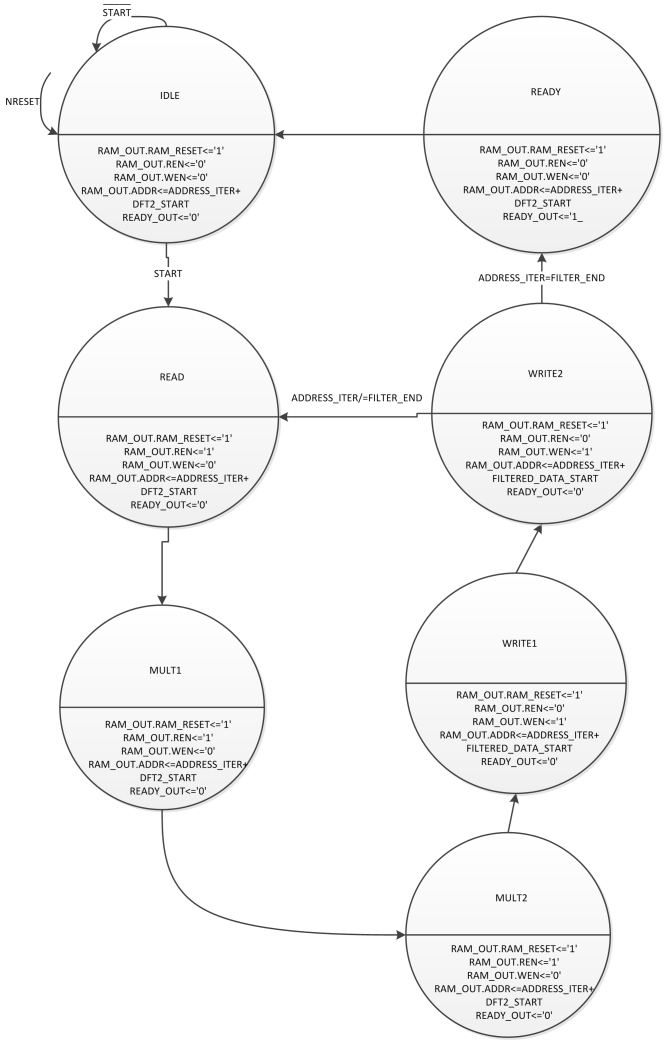

Abbildung A.22: Zustandsdiagramm des Moduls `filter_2d` in der Architektur `COEF_ROM`.

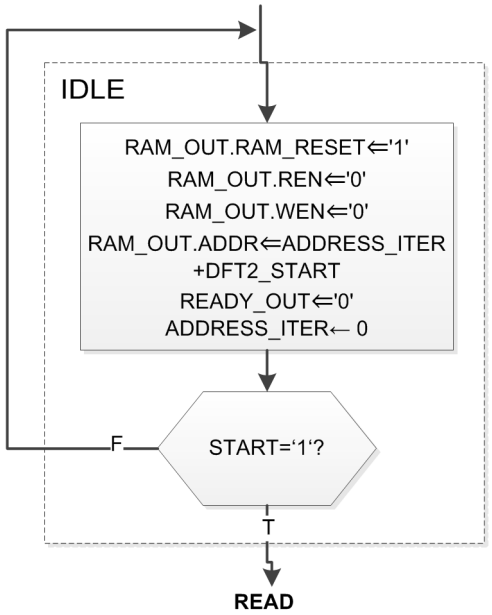

Abbildung A.23: ASMD-Diagramm des Zustands `IDLE` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

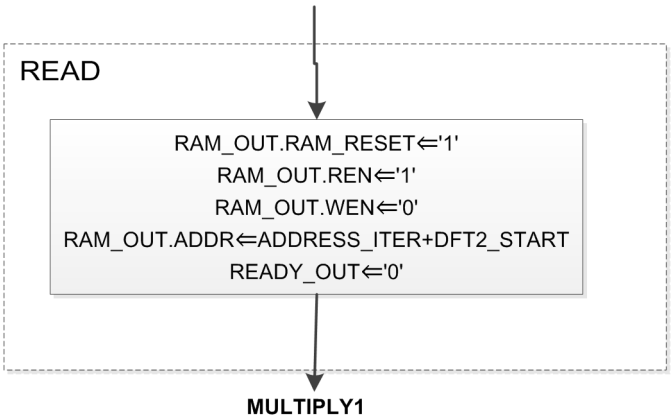

Abbildung A.24: ASMD-Diagramm des Zustands `READ` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

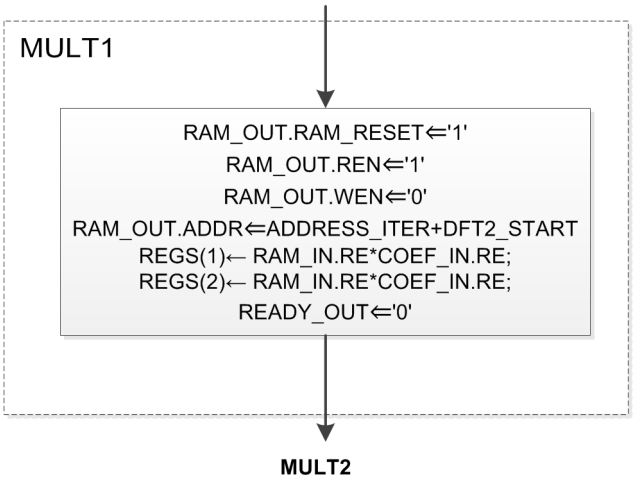

Abbildung A.25: ASMD-Diagramm des Zustands `MULT1` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

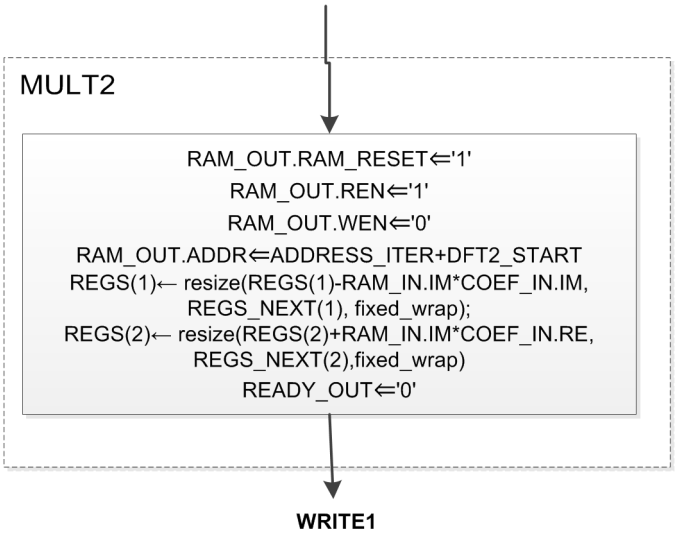

Abbildung A.26: ASMD-Diagramm des Zustands `MULT2` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

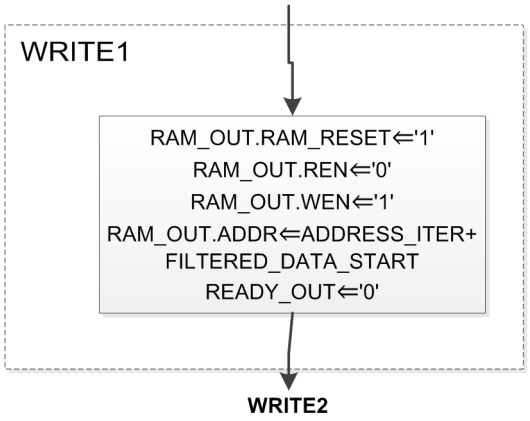

Abbildung A.27: ASMD-Diagramm des Zustands `WRITE1` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

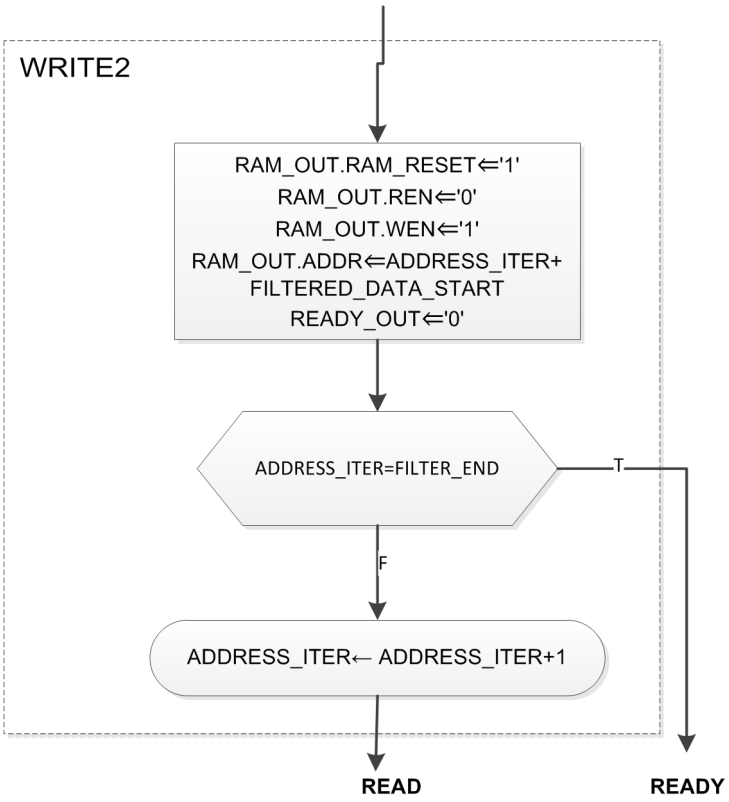

Abbildung A.28: ASMD-Diagramm des Zustands `WRITE2` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

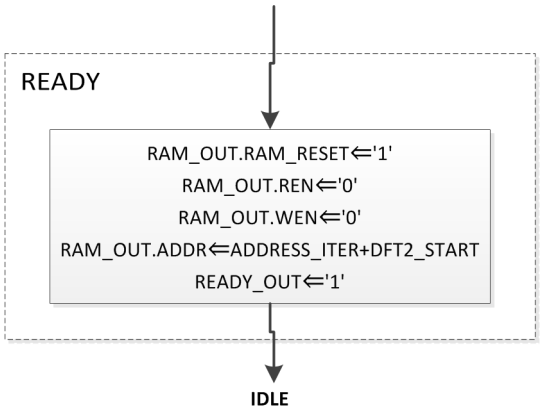

Abbildung A.29: ASMD-Diagramm des Zustands `READY` vom Modul `filter_2d` in der Architektur `COEF_ROM`.

A.3.2 Architektur COEF_RAM

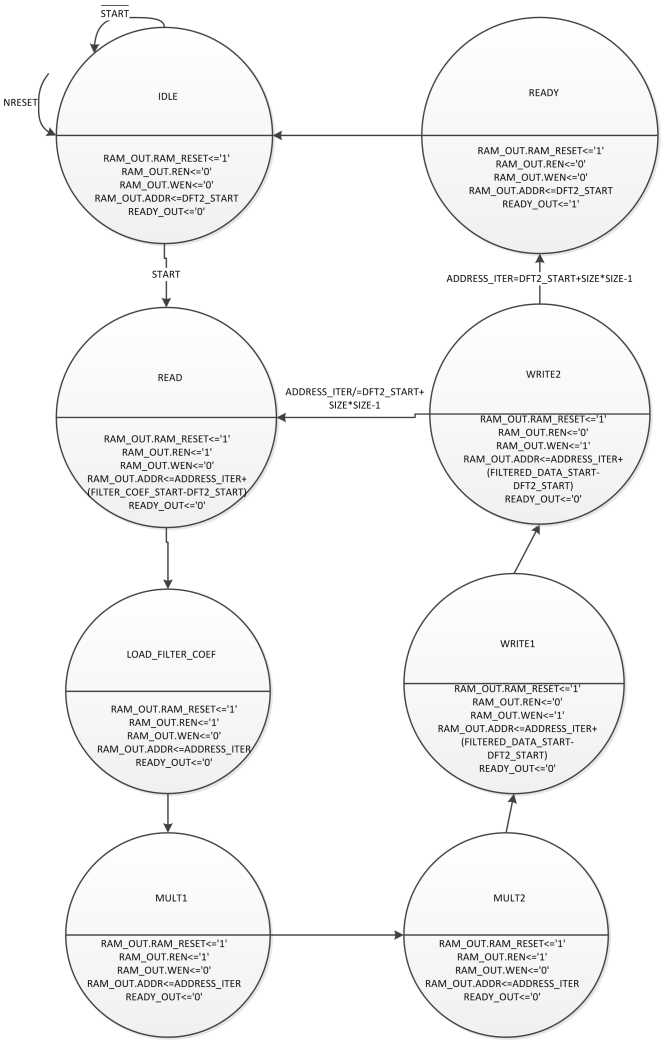

Abbildung A.30: Zustandsdiagramm des Moduls `filter_2d` in der Architektur `COEF_RAM`

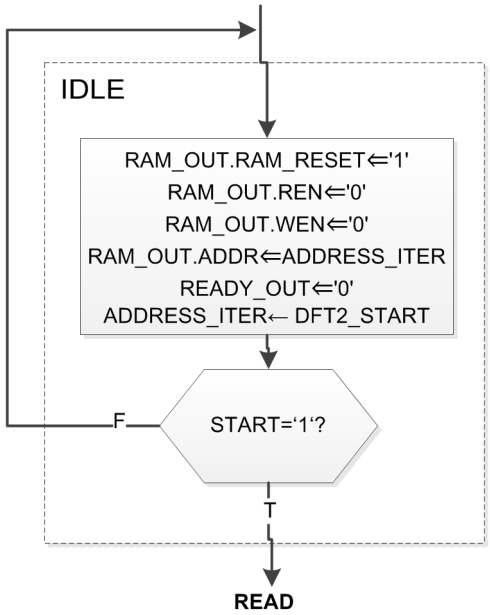

Abbildung A.31: ASMD-Diagramm des Zustands `IDLE` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

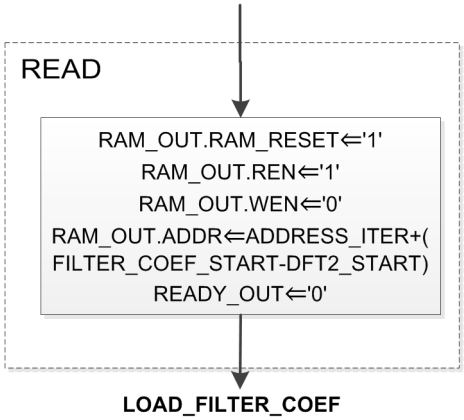

Abbildung A.32: ASMD-Diagramm des Zustands `READ` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

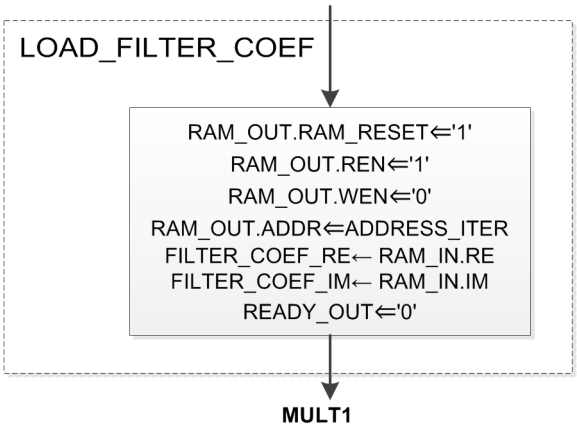

Abbildung A.33: ASMD-Diagramm des Zustands `LOAD_FILTER_COEF` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

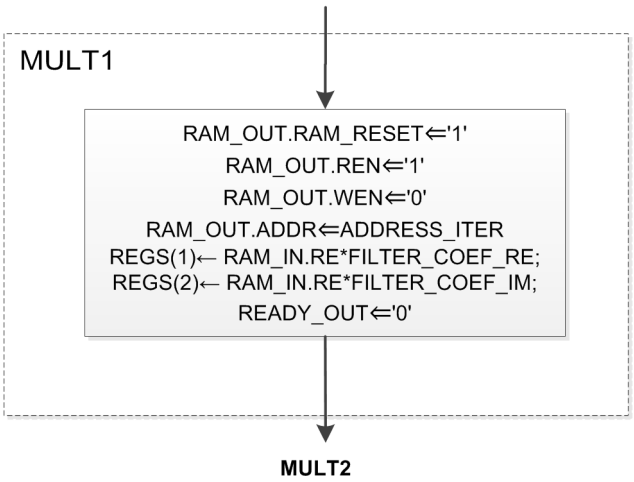

Abbildung A.34: ASMD-Diagramm des Zustands `MULT1` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

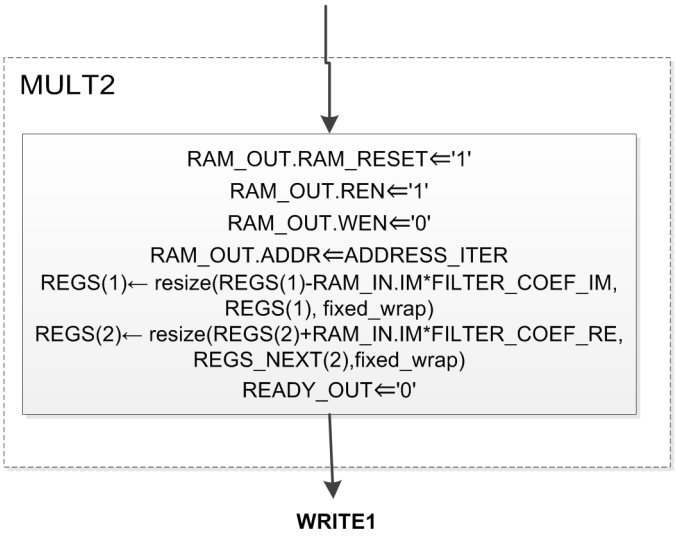

Abbildung A.35: ASMD-Diagramm des Zustands `MULT2` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

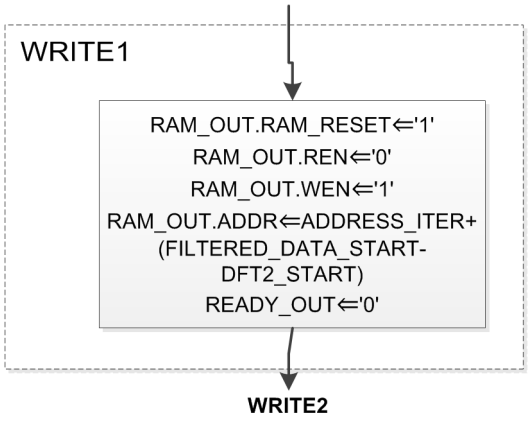

Abbildung A.36: ASMD-Diagramm des Zustands `WRITE1` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

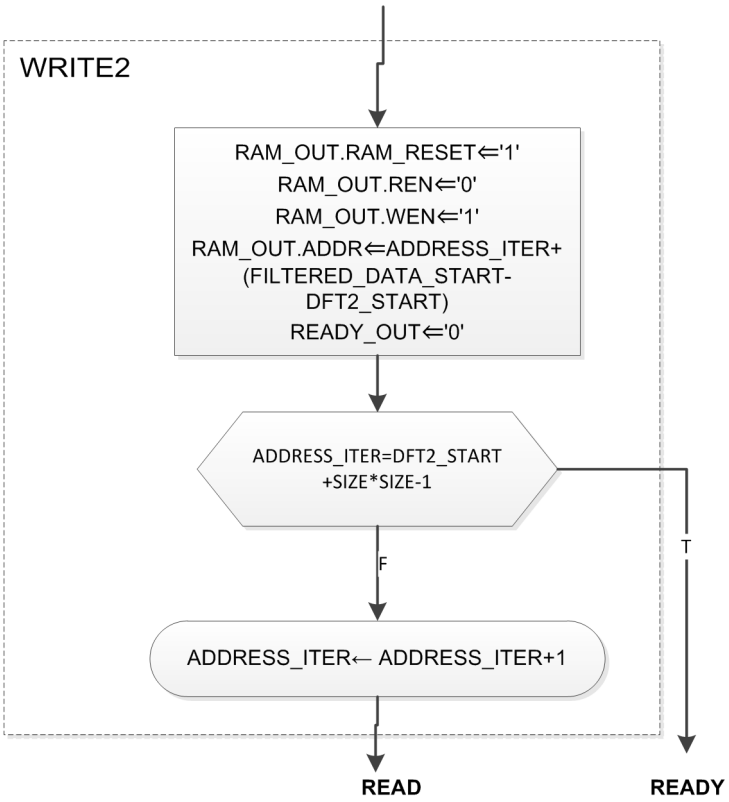

Abbildung A.37: ASMD-Diagramm des Zustands `WRITE2` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

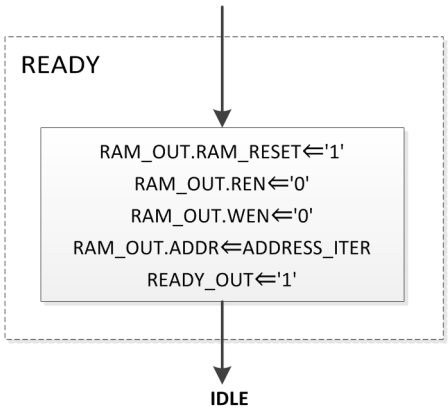

Abbildung A.38: ASMD-Diagramm des Zustands `READY` vom Modul `filter_2d` in der Architektur `COEF_RAM`.

A.4 Filterung über Faltung

A.4.1 Architektur CONV_RAM

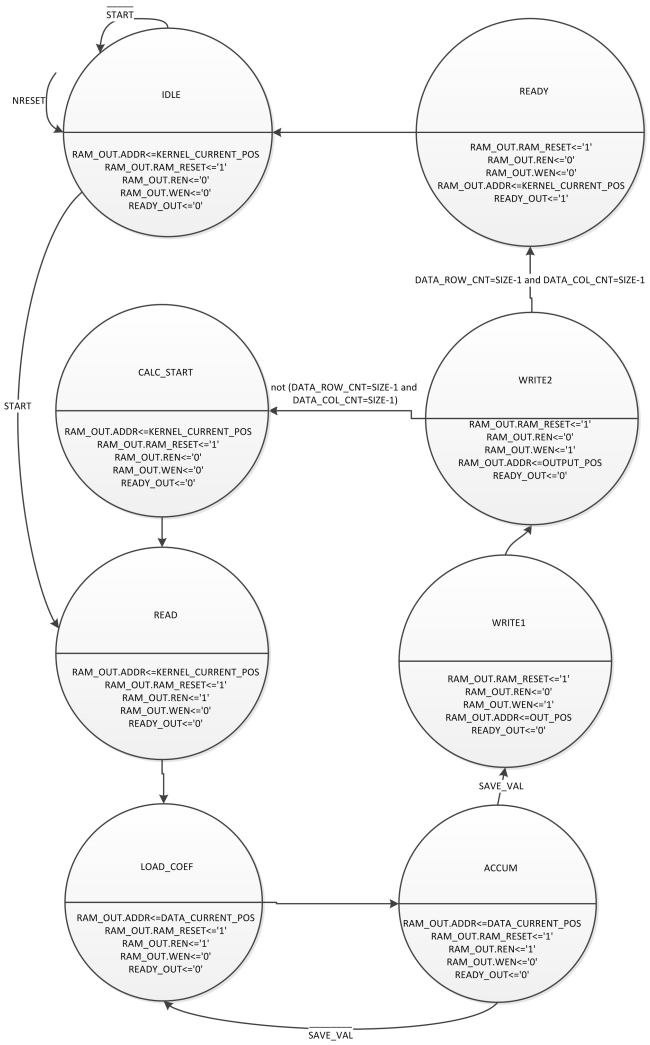

Abbildung A.39: Zustandsdiagramm des Moduls `filter_conv` in der Architektur `CONV_RAM`

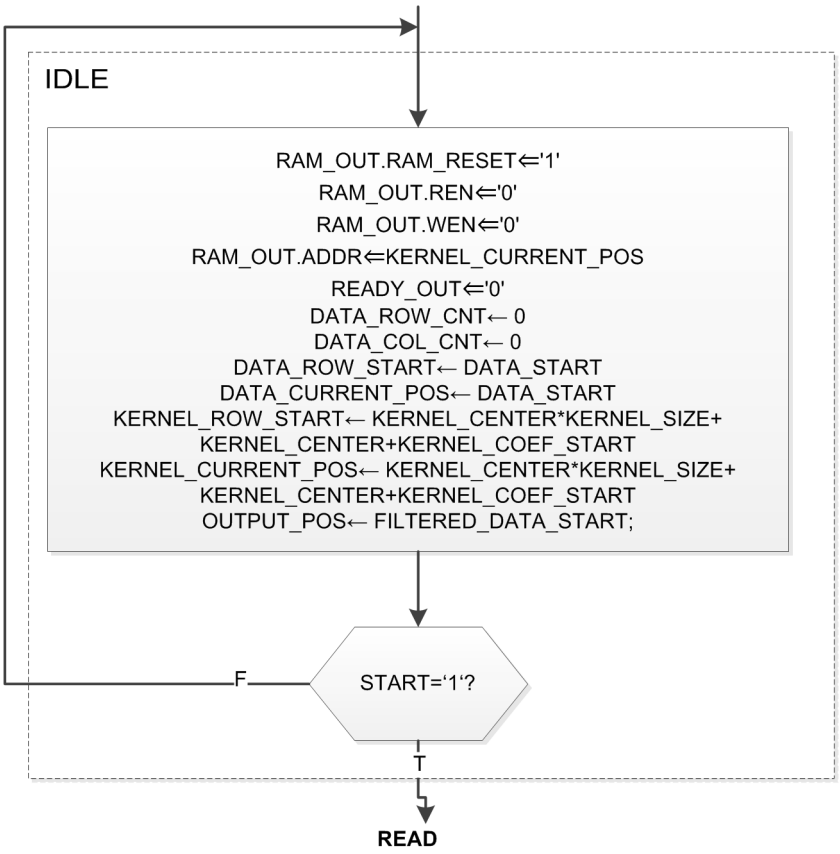

Abbildung A.40: ASMD-Diagramm des Zustands `IDLE` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

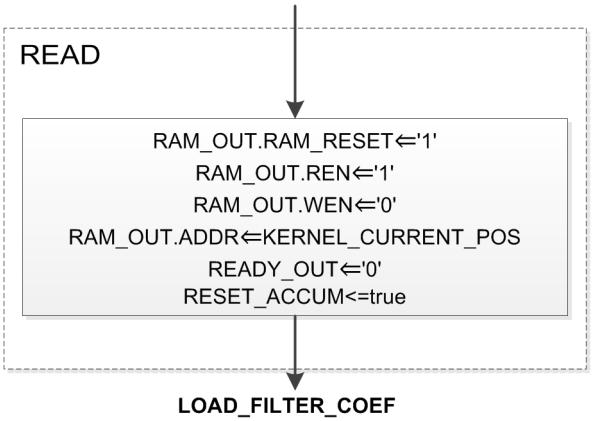

Abbildung A.41: ASMD-Diagramm des Zustands `READ` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

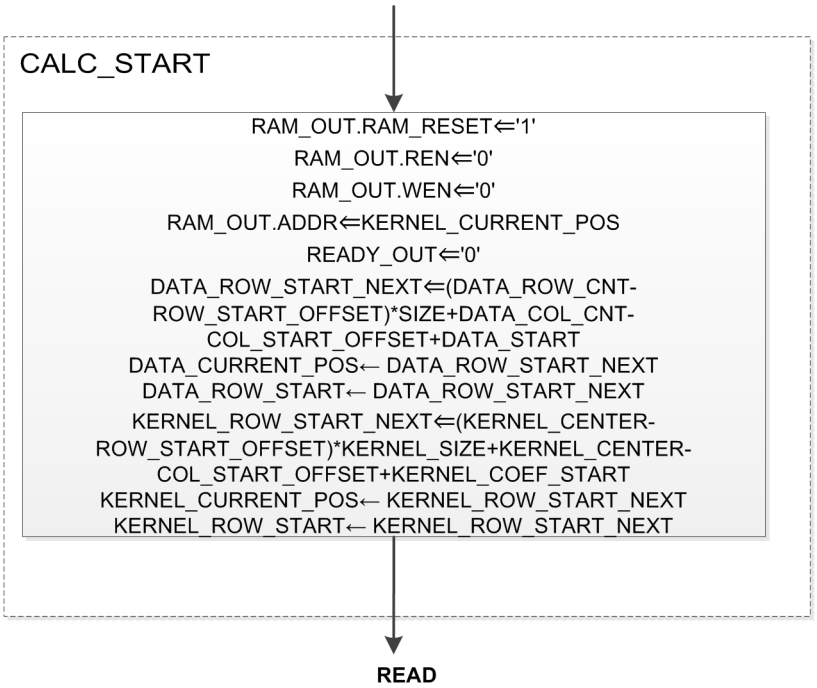

Abbildung A.42: ASMD-Diagramm des Zustands `CALC_START` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

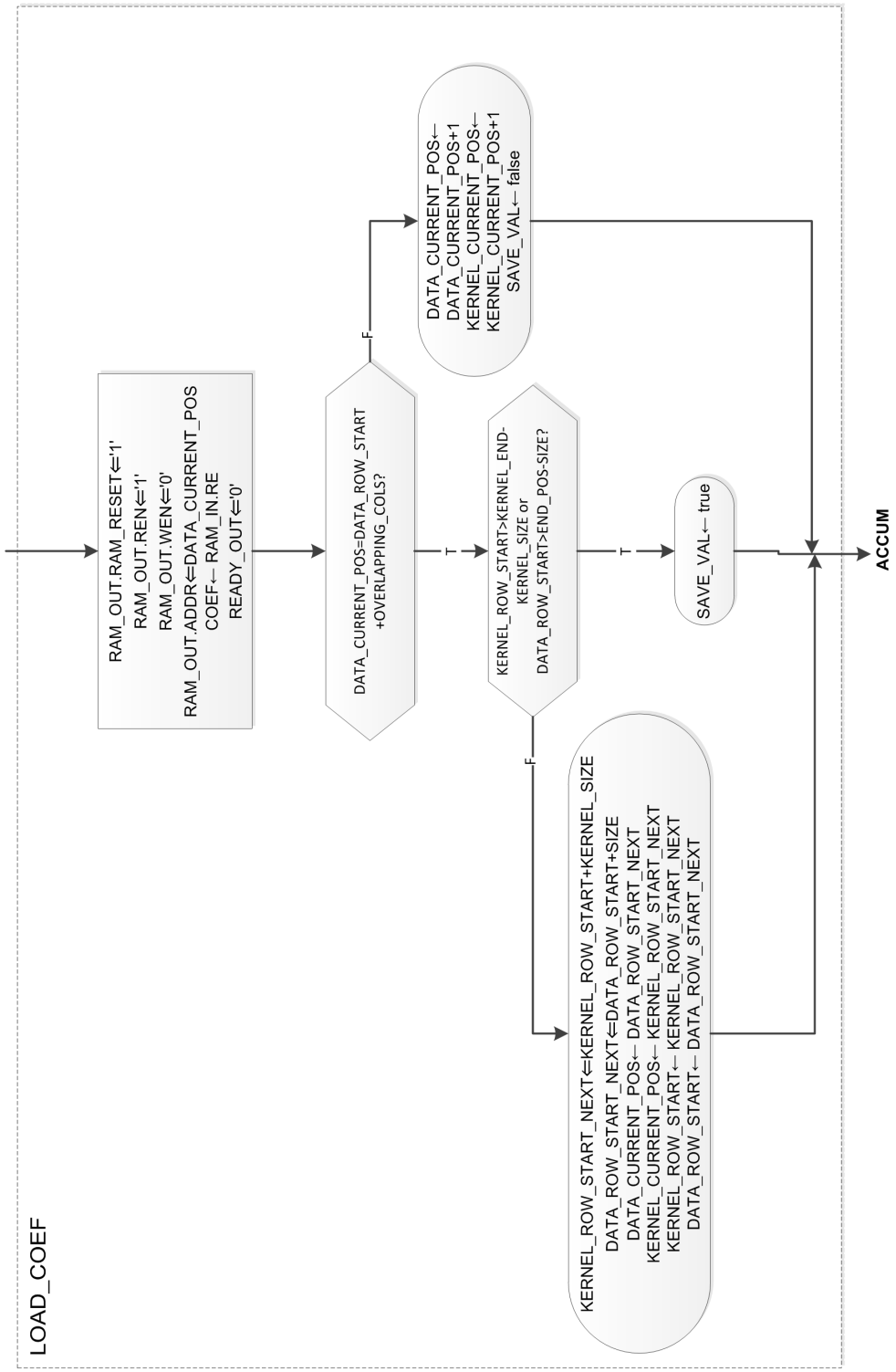

Abbildung A.43: ASMD-Diagramm des Zustands LOAD_COEF vom Modul `filter_conv` in der Architektur CONV_RAM.

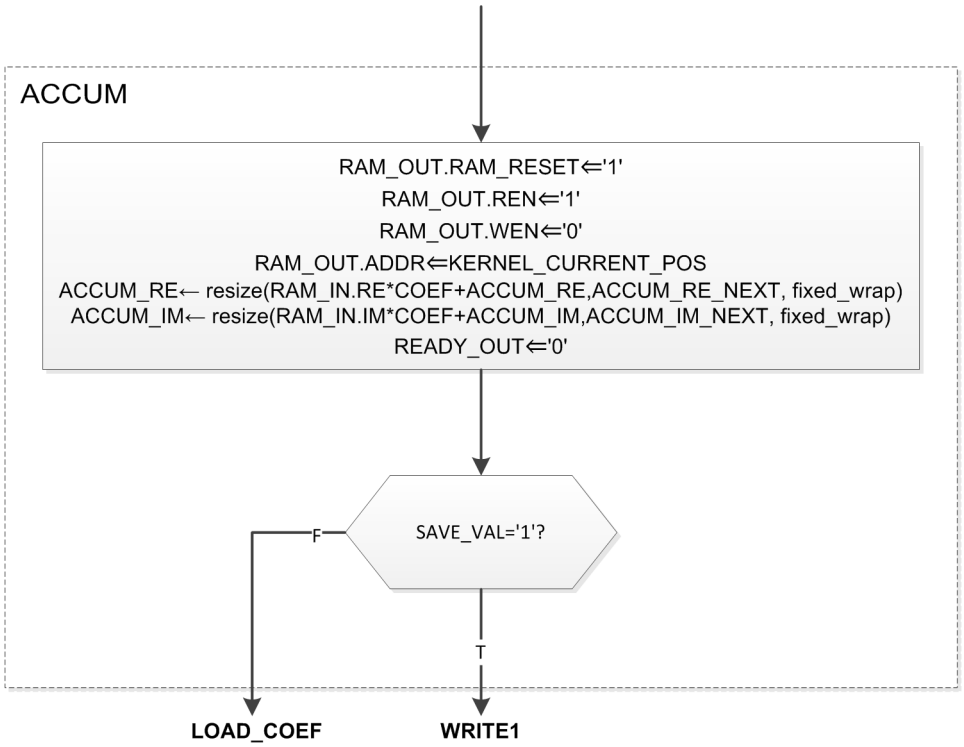

Abbildung A.44: ASMD-Diagramm des Zustands `ACCUM` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

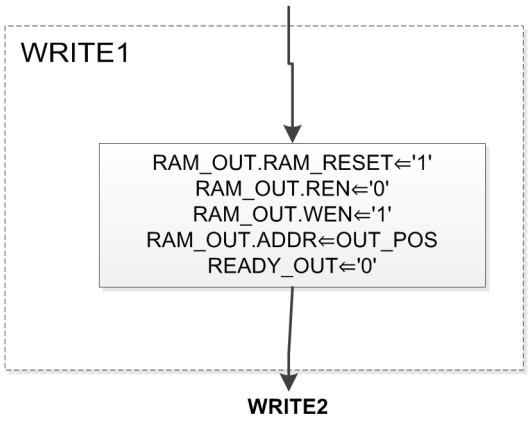

Abbildung A.45: ASMD-Diagramm des Zustands `WRITE1` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

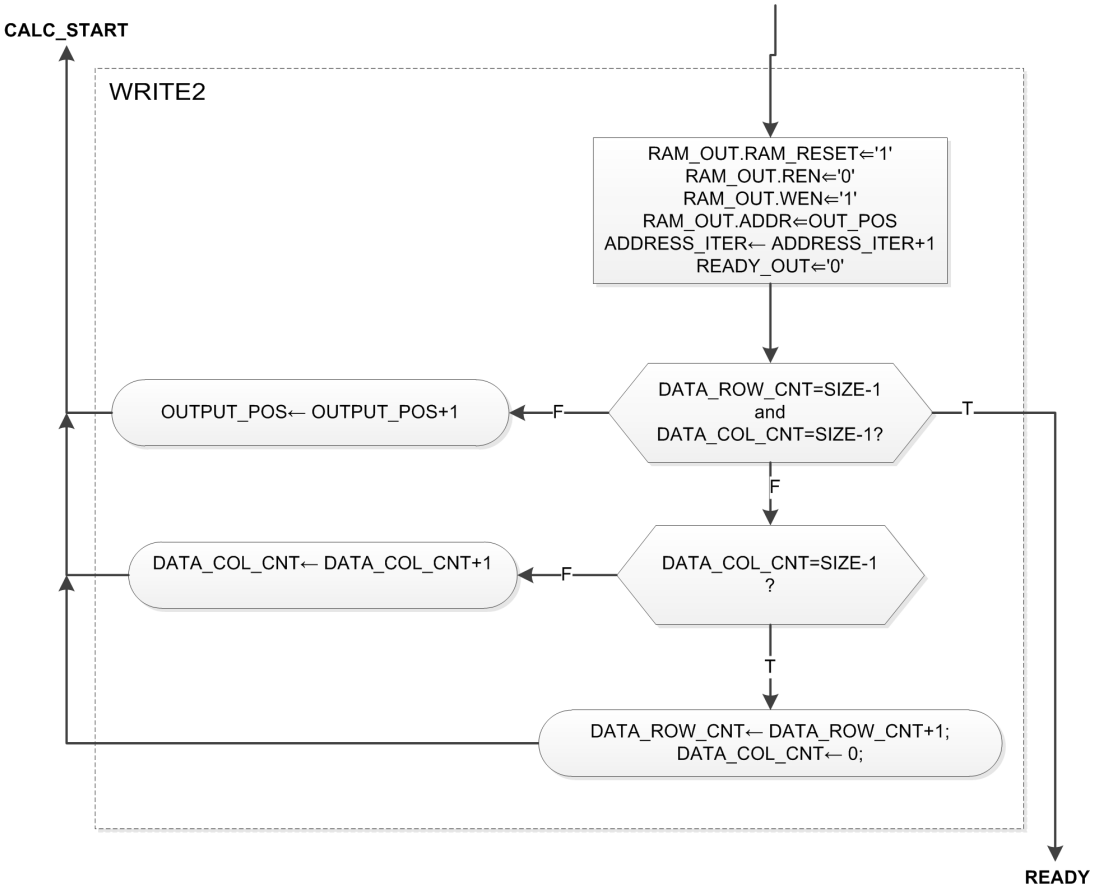

Abbildung A.46: ASMD-Diagramm des Zustands `WRITE2` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

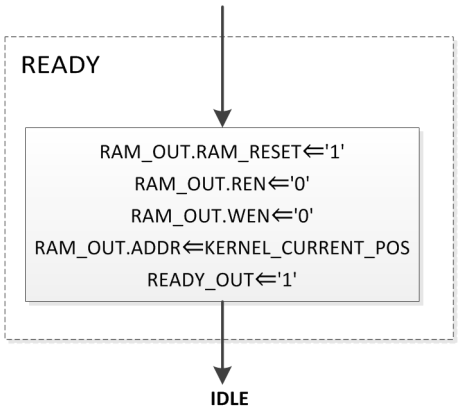

Abbildung A.47: ASMD-Diagramm des Zustands READY vom Modul filter_conv in der Architektur CONV_RAM.

A.4.2 Architektur CONV_ROM

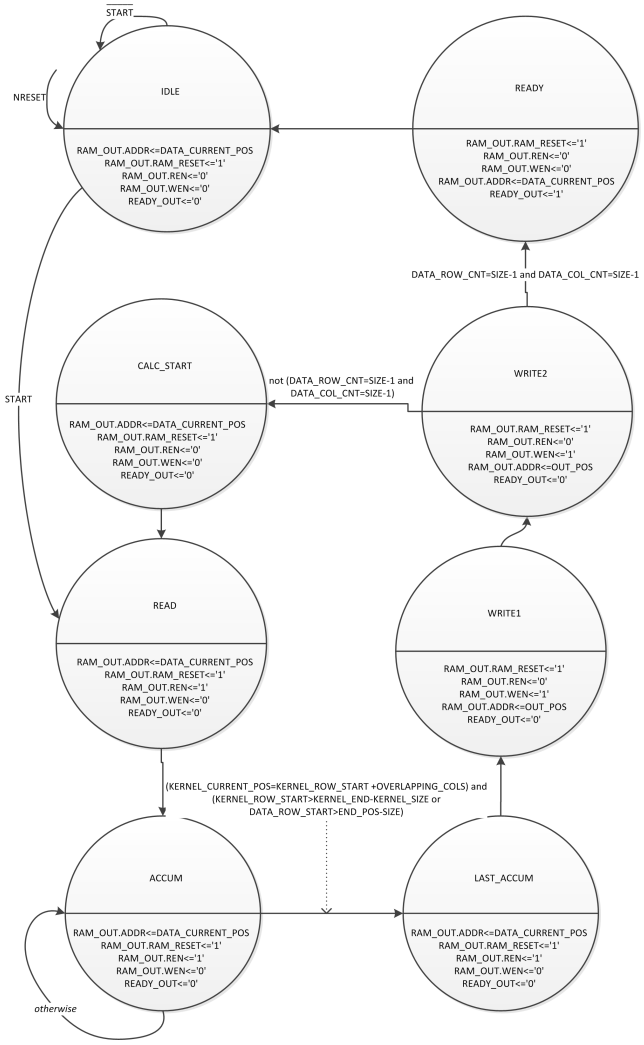

Abbildung A.48: Zustandsdiagramm des Moduls `filter_conv` in der Architektur `CONV_ROM`

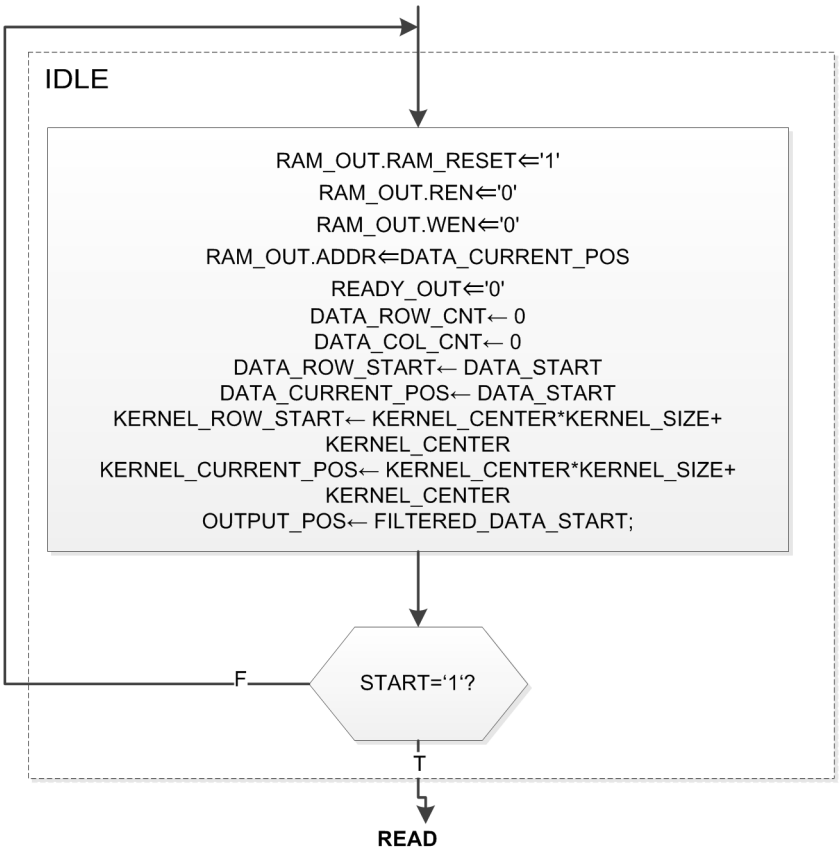

Abbildung A.49: ASMD-Diagramm des Zustands `IDLE` vom Modul `filter_conv` in der Architektur `CONV_ROM`.

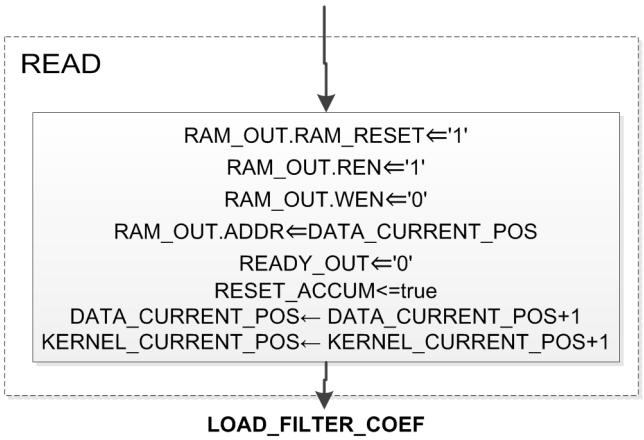

Abbildung A.50: ASMD-Diagramm des Zustands `READ` vom Modul `filter_conv` in der Architektur `CONV_ROM`.

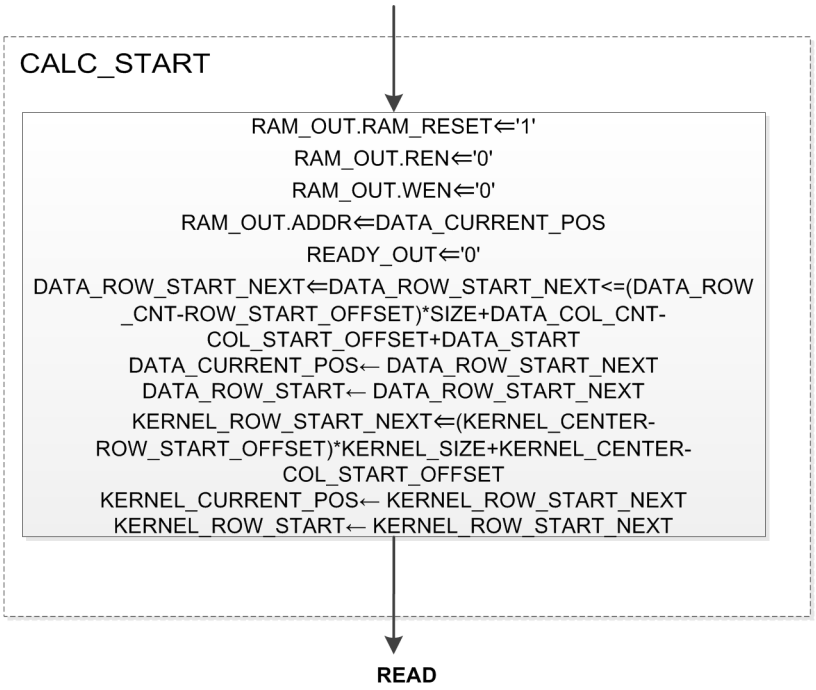

Abbildung A.51: ASMD-Diagramm des Zustands `CALC_START` vom Modul `filter_conv` in der Architektur `CONV_RAM`.

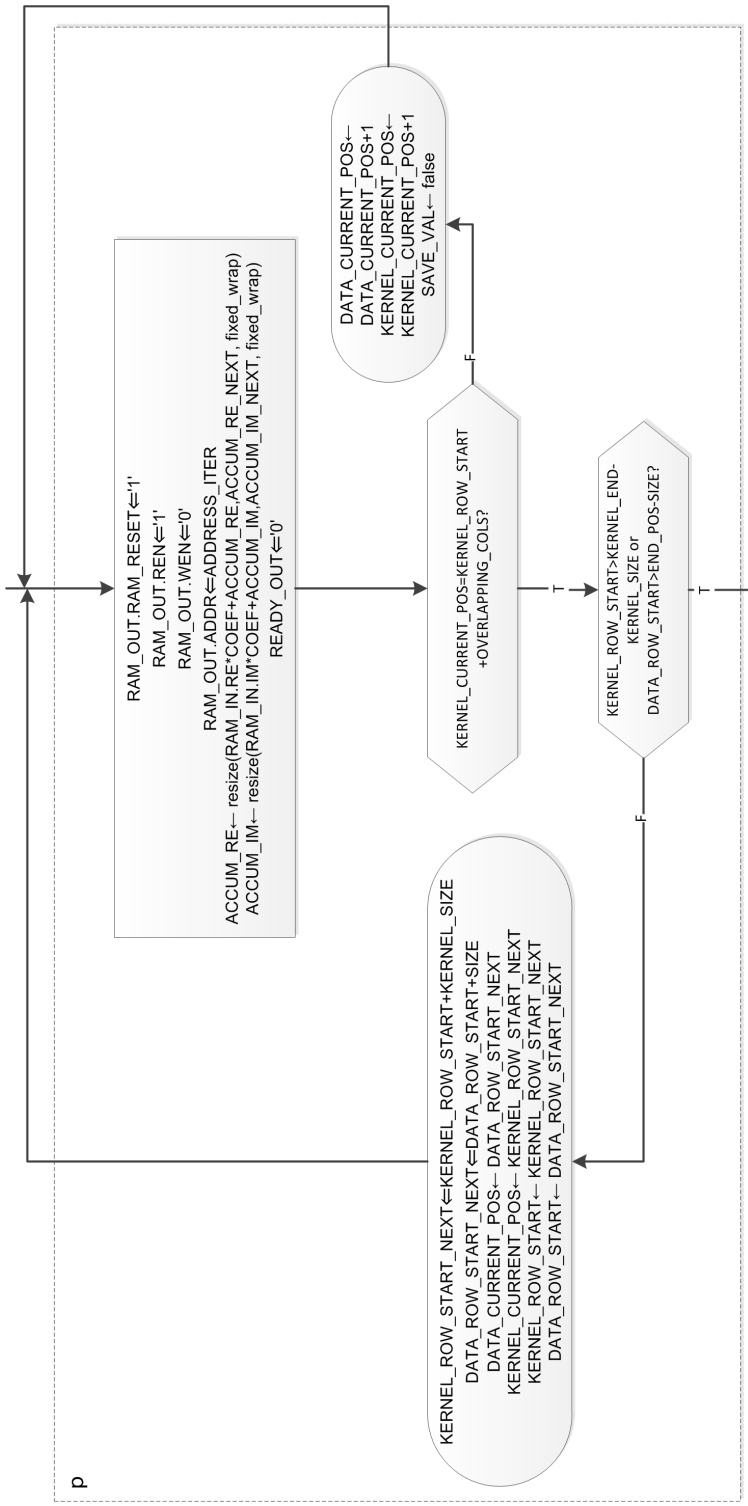

Abbildung A.52: ASMD-Diagramm des Zustands ACCUM vom Modul `filter_conv` in der Architektur CONV_ROM.

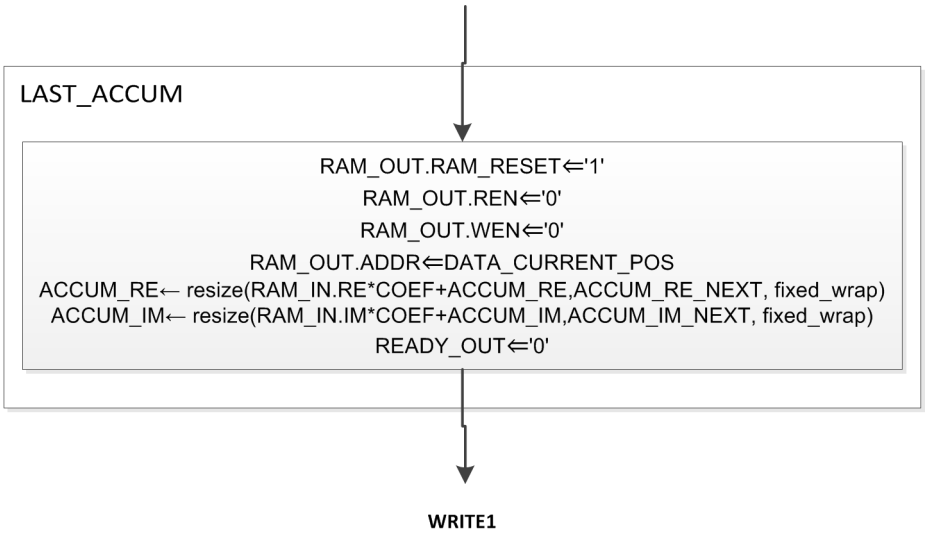

Abbildung A.53: ASMD-Diagramm des Zustands `LAST_ACCUM` vom Modul `filter_conv` in der Architektur `CONV_ROM`.

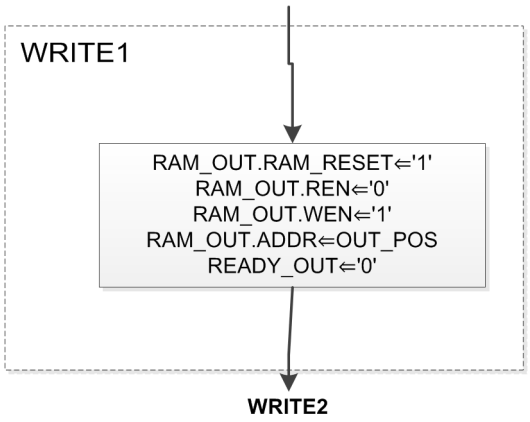

Abbildung A.54: ASMD-Diagramm des Zustands `WRITE1` vom Modul `filter_conv` in der Architektur `CONV_ROM`.

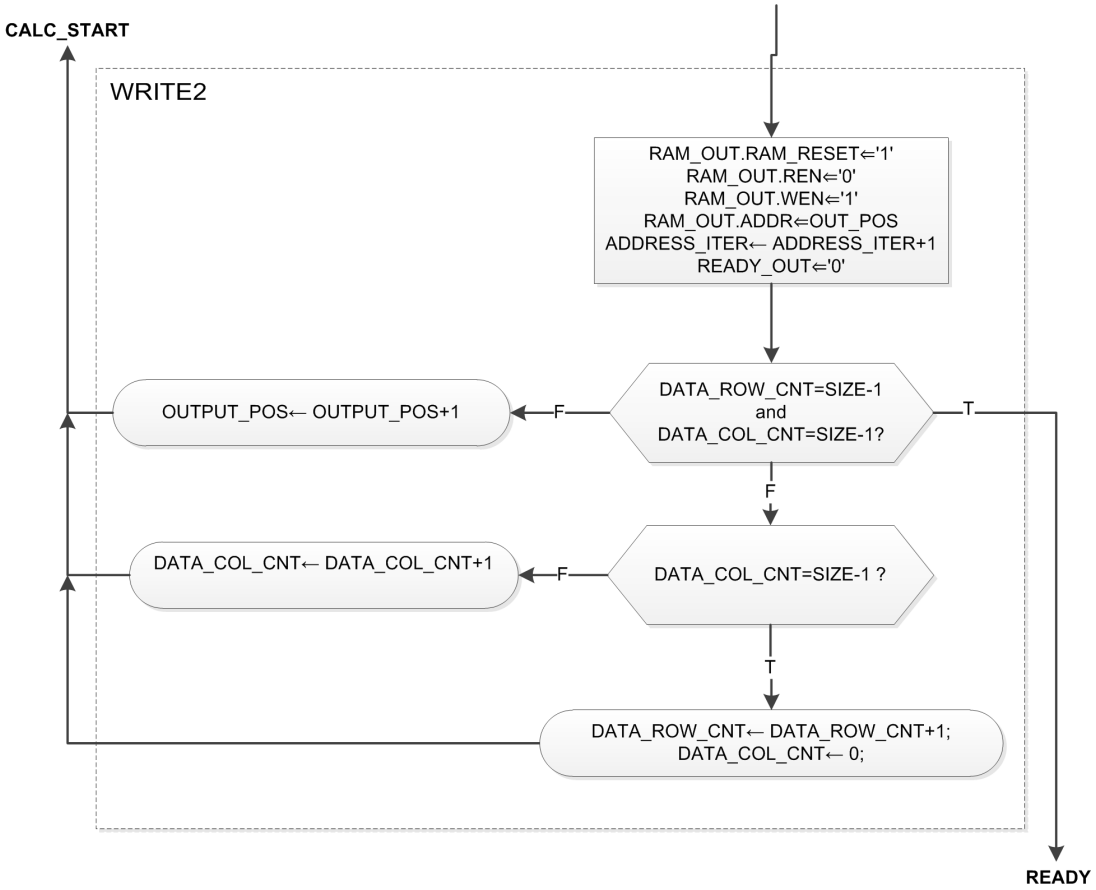

Abbildung A.55: ASMD-Diagramm des Zustands `WRITE2` vom Modul `filter_conv` in der Architektur `CONV_ROM`.

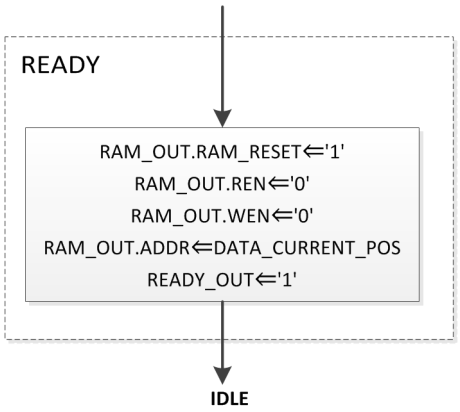

Abbildung A.56: ASMD-Diagramm des Zustands `READY` vom Modul `filter_conv` in der Architektur `CONV_ROM`.

Anhang B

Testverfahren und -ergebnisse

B.1 VHDL

In diesem Anhang werden die Testverfahren für FPGA vorgestellt. Der erste Schritt nach der Fertigstellung der Code ist die Eröffnung von “Elaborated Design” in Vivado IDE. Dies erlaubt auf dem Schema und in der Netzliste nach den unerwünschten Latches zu suchen. Nach ihrer Behebung wird ein Testbench für die Einzelsimulation erstellt und die Verhaltenssimulation durchgeführt. Nachdem das Modul synthetisiert und in physikalischen Elementen implementiert wird, werden die *Postimplementation*-Simulationen durchgeführt. Anders als die idealisierte Verahltenssimulation, verwendet die funktionelle Postimplementation-Simulation die (internen) Modelle für die eigentlich eingesetzten physikalischen Elemente mit realistischen Verzögerungen. Die Postimplementation-Zeitsimulation benutzt zusätzlich die berechneten Signallaufzeiten. [7]

Für Verarbeitungsmodule würde eine VHDL-Simulation zu kompliziert. Sie werden mit Hilfe von Matlab in Testsystem verifiziert. Dafür ist das Testsystem eigentlich vorgesehen.

B.1.1 Simulation von RAM

Für die Simulation wurde ein Testbench (Datei `ram_tb.vhd`) im Simulationset “sim_1” implementiert. Die Tabelle B.1 stellt den Testablauf dar. Dabei beziehen sich die Zeitintervallangaben auf das Signaldarstellung auf Abbildung B.1

Tabelle B.1: Testfälle für RAM

Testfall	**Zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, μs**
Beschreiben der RAM-Zelle 5	Nach dem Setzen WE und einstellen von AD auf 5 wird der Wert `0x00100001` in Zelle 5 gespeichert	ja	155-355
Beschreiben einer weiteren RAM-Zelle (13)	Nach dem Umstellung des Adresse-Eingangs auf 13 wird der Wert `0x10000100` in Zelle 13 gespeichert	ja	355-455
Lesen aus der Zelle 5	Nach dem Setzen RE und einstellen von AD auf 5 erscheint der Wert `0x00100001` am Datenausgang vom RAM	ja	555-655
Test von der Ausgangs-aktivierung	Nach dem Setzen des Ausgangsaktivierung `EN` auf “high” wird der Datenausgang abgeschaltet (hochohmig)	ja	755-955
Test vom simultanen Lesen und Schreiben	Nach dem Setzen von entsprechenden Signalen wird der neue Wert `0x01001000` in Zelle 13 gespeichert	ja	950-1050
Test der RAM Abschaltung	Nach dem Setzen von `NRESET` auf “low” reagiert RAM weder auf Schreib- noch auf Lesebefehle.	ja	1155-1250

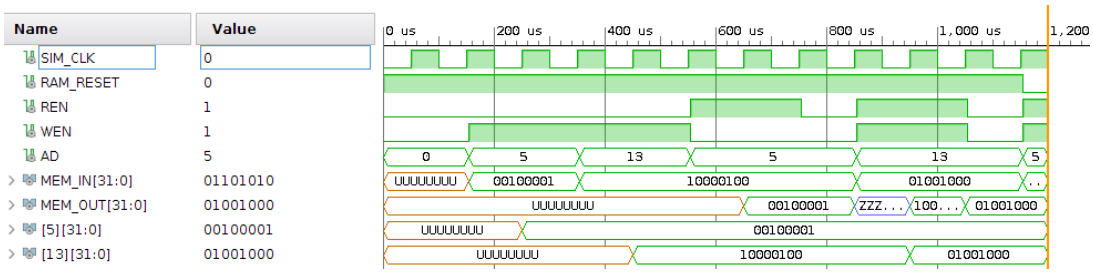

Abbildung B.1: Signalablauf-Diagramm der RAM-Simulation

B.1.2 Simulation der RAM-Verwaltung

Das Testbench `mac_tb.vhd` für die Verhaltenssimulation des `memory_access_ctrl` befindet sich im Simulationsset “new_1”. Die Zeitintervale in Tabelle B.2 beziehen sich auf die Abbildungen B.2 -B.3.

Tabelle B.2: Testfälle für RAM-Verwaltung

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
1: Schreiboperation durch Modul 0 (“DFT”)	Mit dem Befehl `00000` wird die RAM-Kontrolle an Modul 0 übergeben. Das Modul schreibt 0x34e als Realteil und 0x451 als Imaginärteil. Die werte werden unter Adresse 5 gespeichert	ja	150-450
2: Fehlbeschreibung von RAM	Der Imaginärteil wird zwar auf 0x156 gestzt, jedoch wird WEN auf low gesetzt. Der Inhalt der RAM darf sich nicht ändern	ja	450-550
3: Zurücklesen	Nach dem Setzen von REN erscheint der unter Testafall gespeicherte Werte am Datenausgang vom RAM	ja	450-650

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.2

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
4: Externes Beschreiben von RAM			
4a: Einstellung der Adresse	Der Wert 11 wird auf dem externen Eingangsbus eingestellt. Nachdem auf dem Befehlsbus der binäre Wert `10100` eingestellt wird, wechselt das Modul in den Zustand `SET_EXT_ADDRESS` und der Wert 11 wird aus dem Eingangsregister in die MSBs von Shiftregister übetragen.	ja	750-950
4b: Byte 4 der externen Daten	Der Wert 0x05 wird auf dem externen Eingangsbus eingestellt. Nachdem auf dem Befelsbus der binäre Wert `10000` eingestellt wird, wechselt das Modul in den Zustand `RECEIVE_EXT_DATA` und der Wert 0x05 wird aus dem Eingangsregister in die Bits 31-24 von Shiftregister übetragen.	ja	900-1050
4c: Byte 3 der externen Daten	Der Wert 0xd4 wird auf dem externen Eingangsbus eingestellt. Nachdem auf dem Befelsbus der binäre Wert `10001` eingestellt wird, wird der Wert 0xd4 aus dem Eingangsregister in die Bits 23-16 von Shiftregister übetragen.	ja	950-1150

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.2

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
4d: Byte 2 der externen Daten	Der Wert 0xfe wird auf dem externen Eingangsbus eingestellt. Nachdem auf dem Befelsbus der binäre Wert `10010` eingestellt wird, wird der Wert 0xfe aus dem Eingangsregister in die Bits 15-8 von Shiftregister übetragen.	ja	1050-1250
4e: Byte 1 der externen Daten	Der Wert 0x3c wird auf dem externen Eingangsbus eingestellt. Nachdem auf dem Befelsbus der binäre Wert `10011` eingestellt wird, wird der Wert 0x3c aus dem Eingangsregister in die Bits 7-0 von Shiftregister übetragen.	ja	1150-1350
4f: Übertragen der empfangenen Daten in das RAM	Nachdem auf dem Befelsbus der binäre Wert `11000` eingestellt, geht das Moduk in den Zustand `EXT_WRITE`. Dabei werden der empfangene Datenwert 0x05d4fe3c unter Andresse 11 gespeichert.	ja	1250-1350

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.2

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
5: Lesen der Daten durch das Modul 1 (“FILTER”)	Nachdem auf dem Befelsbus der binäre Wert `00001` eingestellt wird, wechselt das Modul zurück zum Zustand `INT_ACCESS` und die RAM-Kontrolle an Modul 1 übergeben. Das Modul lieste die empfangenen Daten aus der Zelle 11, so dass der die Werte 0x5d4 und 0xe3c auf dem RAM-Ausgang erscheinen.	ja	1350-1650
6: Externes Lesen von Daten			
6a: Einstellung der Adresse	Der Wert 5 wird auf dem externen Eingangsbus eingestellt. Nachdem auf dem Befehlsbus der binäre Wert `10100` eingestellt wird, wechselt das Modul in den Zustand `SET_EXT_ADDRESS` und der Wert 5 wird aus dem Eingangsregister in die MSBs von Shiftregister übetragen.	ja	1750-1950
6b: Byte 4	Nachdem der binäre Wert `01000` auf dem Befehlsbus eingestellt wird, wechselt das Modul zunächst in den Zustand und dann in den Zustand `EXT_READ`. Daraufhin erscheint der Wert vom Byte 4 (0x03) auf dem externen Ausgangsbus.	ja	1850-2150

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.2

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
6c: Byte 3	Nachdem der binäre Wert `01001` auf dem Befehlsbus eingestellt wird, erscheint der Wert vom Byte 3 (0x4e) auf dem externen Ausgangsbus.	ja	2150-2250
6d: Byte 2	Nachdem der binäre Wert `01010` auf dem Befehlsbus eingestellt wird, erscheint der Wert vom Byte 2 (0x04) auf dem externen Ausgangsbus.	ja	2250-2350
6f: Byte 3	Nachdem der binäre Wert `01011` auf dem Befehlsbus eingestellt wird, erscheint der Wert vom Byte 1 (0x21) auf dem externen Ausgangsbus.	ja	2350-2450
7: Zustandswechsel `EXT_READ`→`RECEIVE_EXT_DATA`	Im Zustand `EXT_READ` verursacht der Befehl `10000` den Übergang zum Zustand `RECEIVE_EXT_DATA`	ja	2450-2650
8: Zustandswechsel `RECEIVE_EXT_DATA`→`INIT_EXT_READ`	Im Zustand `RECEIVE_EXT_DATA` verursacht der Befehl `01000` den Übergang zum Zustand `INIT_EXT_READ`	ja	2550-2750
9: Zustandswechsel `INIT_EXT_READ`→`INT_ACCESS`	Im Zustand `INIT_EXT_READ` verursacht der Befehl `00XXX` den Übergang zum Zustand `INT_ACCESS`	ja	2650-2850

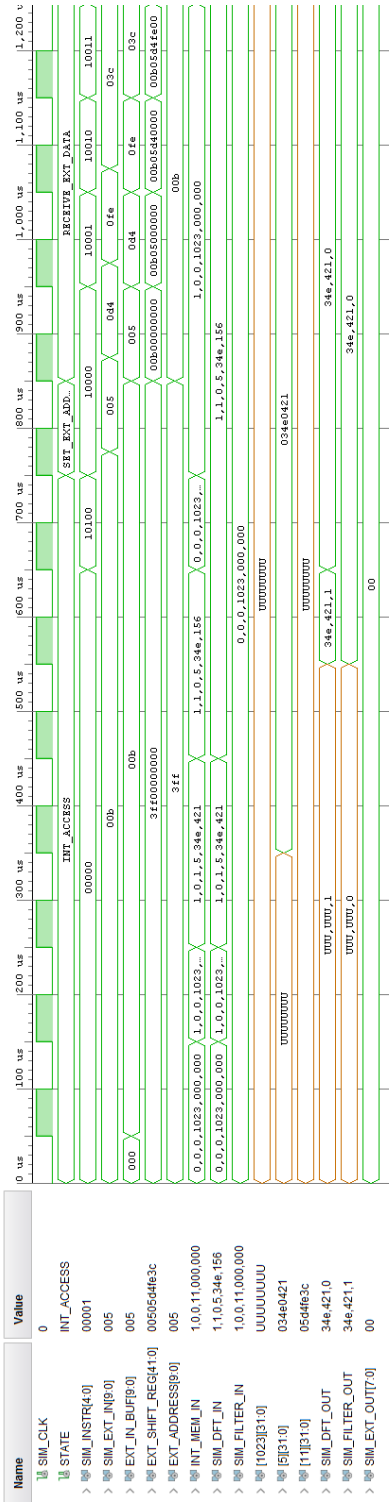

Abbildung B.2: Signalablauf-Diagramm der Simulation der RAM-Verwaltung,0-1200 µs

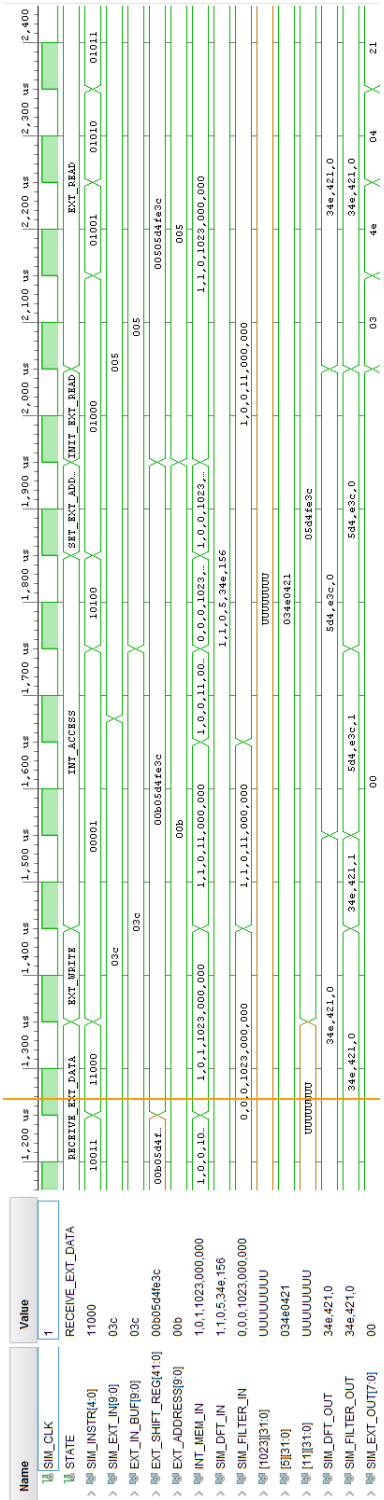

Abbildung B.3: Signalablauf-Diagramm der Simulation der RAM-Verwaltung, 1200-2400 µs

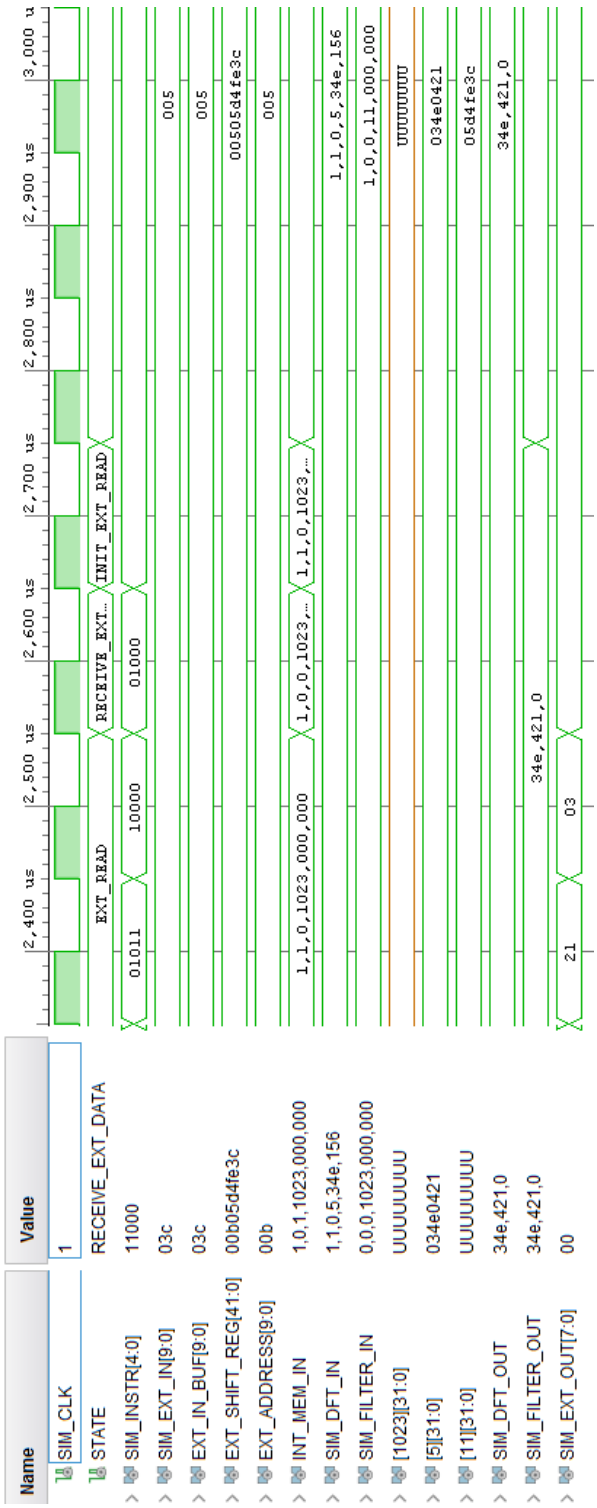

Abbildung B.4: Signalablauf-Diagramm der Simulation der RAM-Verwaltung, 2400-3000 µs

B.1.2.1 Postimplementation-Simulation

Daraufhin wurde die funktionelle Postimplementation-Simulation (Simulationsset mem_ctrl_func_impl_sim) durchgeführt. Es hat sich herausgestellt, dass Vivado die `record`-Typen bei Synthese aufrollt, also in ihre Komponentensignale aufteilt. Dies erforderte ein eigenständiges Testbench zwar mit denselben Testfällen, aber für die von Vivado modifizierte `entity`-Deklaration. Die Simulation wird mit dem Tcl-Skript `start_func_sim.tcl` gestartet.

Die Simulation hat ein Glitch gezeigt (Abbildung B.5), der nicht auf der Verhaltenssimulation aufgetreten hatte und dann behoben wurden. Dies illustriert die Wichtigkeit der Postimplementation-Simulation. Auf der anderen Seite sind die implementierten Designs nicht mehr durchsichtig. So ist es auch nicht möglich den Inhalt vom RAM direkt einzusehen.

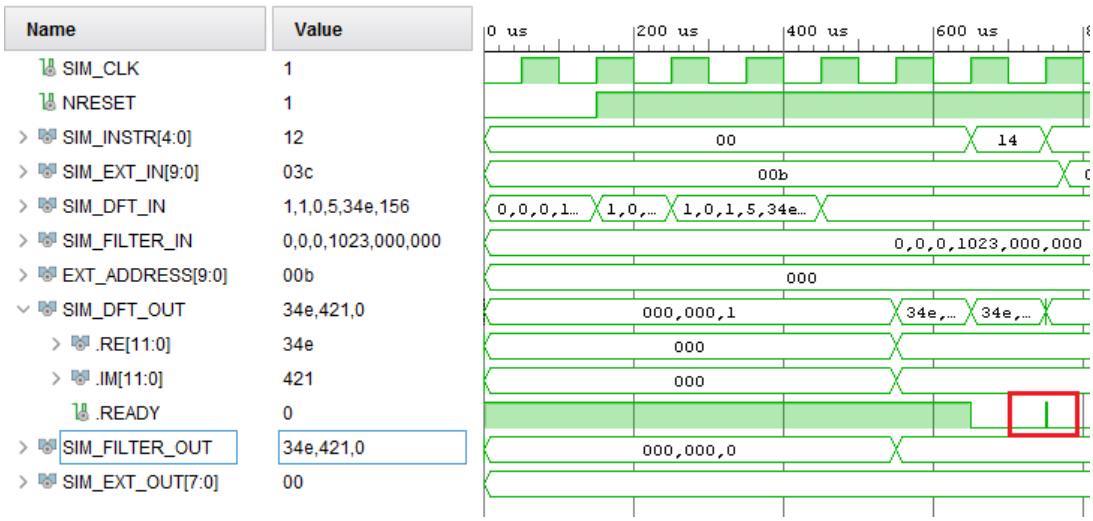

Abbildung B.5: Glitch auf der funktionellen Postimplementation-Simulation

B.1.3 Simulation des FPGA-Teil des Testsystems

Es hat sich als nicht sinnvoll erwiesen, die Modulsteuerung einzeln zu simulieren, da dafür das Verhalten aller anderen Module in der Simulation nachgemacht werden müsste. Statt dessen nimmt man lieber die Module selbst,

also führt die Simulation des gesamten FPGA-Teiltestsystems durch. Das entsprechende Testbench `proc_sys_tb.vhd` befindet sich im Simulationsset sys_behav_sim. Die Zeitangaben der Testfälle in Tabelle B.3 beziehensich auf Abbildungen B.6-B.19. Es wurde auch die funktionelle Postimplemenations-Simulation durchgeführt, die allerdings keine Auffälligkeiten zeigte.

Tabelle B.3: Testfälle für `proc_sys`

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
1: Befüllen vom RAM durch das ersten Testmodul	Das Modul befindet sich nach dem Aufspielen im Zustand `DFT`. Nachdem der Reset der Verarbeitungsmodule aufgehoben wird, füllt der erste Testmodul die RAM mit Werten (s. 11). Anschließend setzt das Testmodul sein Read-Signal für einen Takt hoch.	ja	0-1250
2: Externes Auslesen der RAM	Nach dem Setzen `EXT_CTRL_EN` hoch und des Befehlbuses auf `10100` geht die Modulsteuerung in den Zustand `EXT_CTRL`. Die im 1 produzierte Werte werden aus dem Bereich [0;9] ausgelesen. Sie erscheinen Byte nach Byte auf dem externen Ausgangsbus	ja	1250-8250
3: Externes Schreiben	Das RAM im Bereich [0;9] wird extern mit den von `0xff50` und `0x0060` (jeweils Real- und Imaginärteil) aufsteigenden Werte befüllt	ja	8250-15350

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.3

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
4: Externe Aktivierung eines Moduls	Mit dem Befehl `00001` geht die Modul Steuerung in den Zustand `FILTER` über. Es wird damit der zweite Testmodul aktiviert, der die Werte wie auf S. 11 beschrieben umspeichert.	ja	15200-19550
5: Lesen der umspeicherten Daten	Die im Testfall 4 umspeicherten Werten werden ausgelesen. Sie sollen Byte nach Byte am externen Datenausgang in der umgekehrten Reihenfolge relativ zu den im Testfall 3 gespeicherten Werten auftreten.	ja	19550-26650
6: Übergang zur internen Steuerung	Nach dem Setzen vom `EXT_CTRL_EN` auf low und dem Befehl `00000` soll die externe Steuerung deaktiviert werden und die interne aktiviert. U. a. bedeutet dies, dass nach der Erledigung des ersten Modul, der zweite Modul automatisch intern aktiviert wird. Das zweite Modul speichert die vom ersten Modul produzierten Werte.	ja	26550-32050

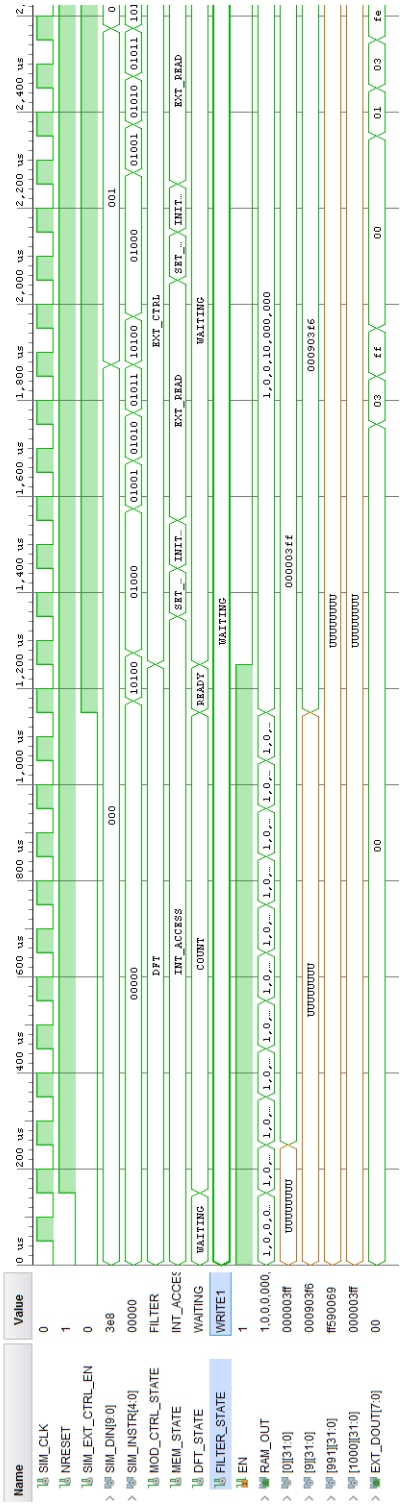

Abbildung B.6: Signalablauf-Diagramm der Simulation des Testsystems, 0-2400 µs

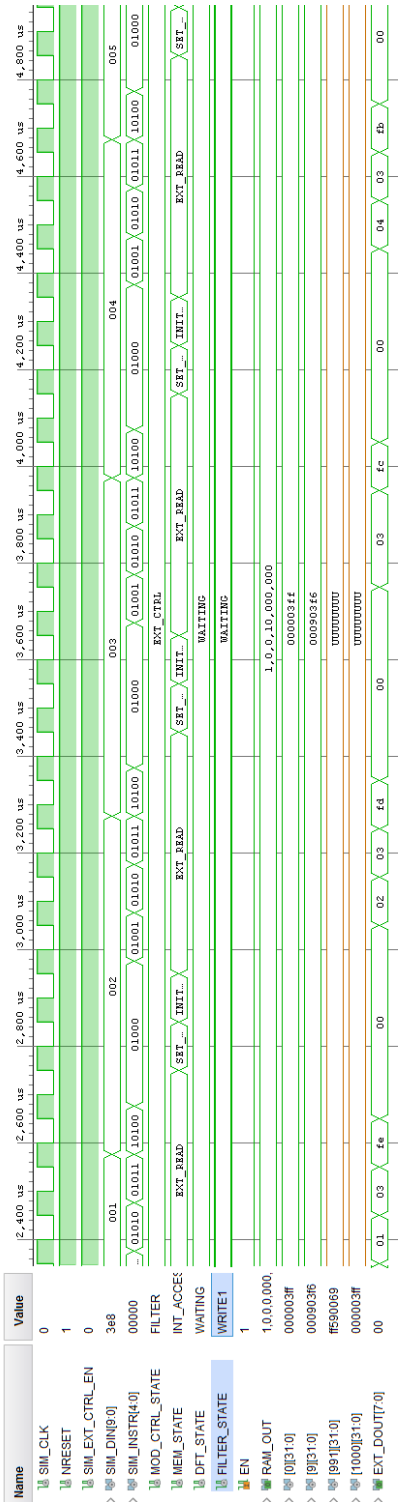

Abbildung B.7: Signalablauf-Diagramm der Simulation des Testsystems, 2400-4800 µs

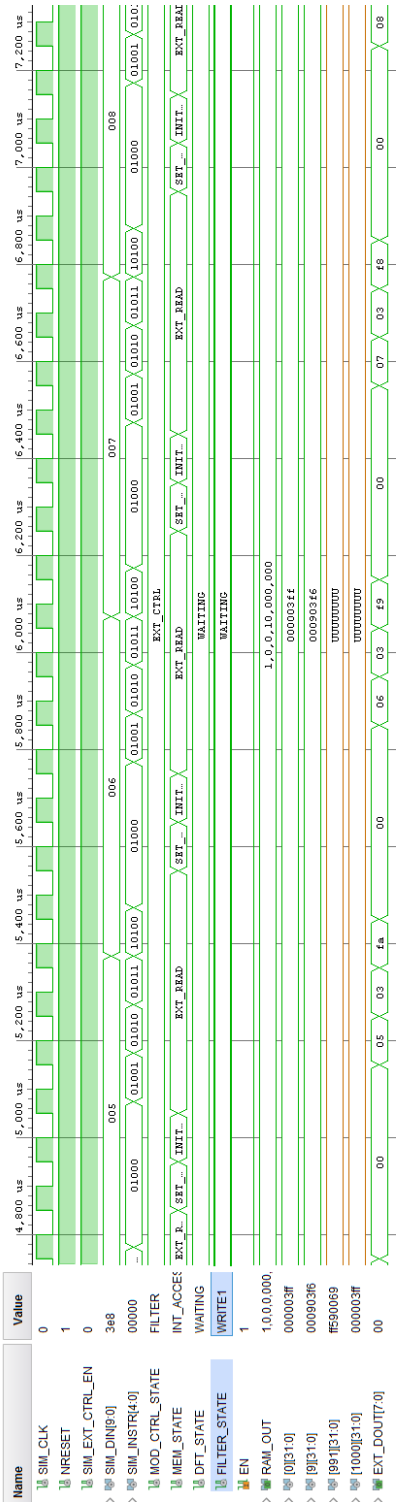

Abbildung B.8: Signalablauf-Diagramm der Simulation des Testsystems, 4800-7200 µs

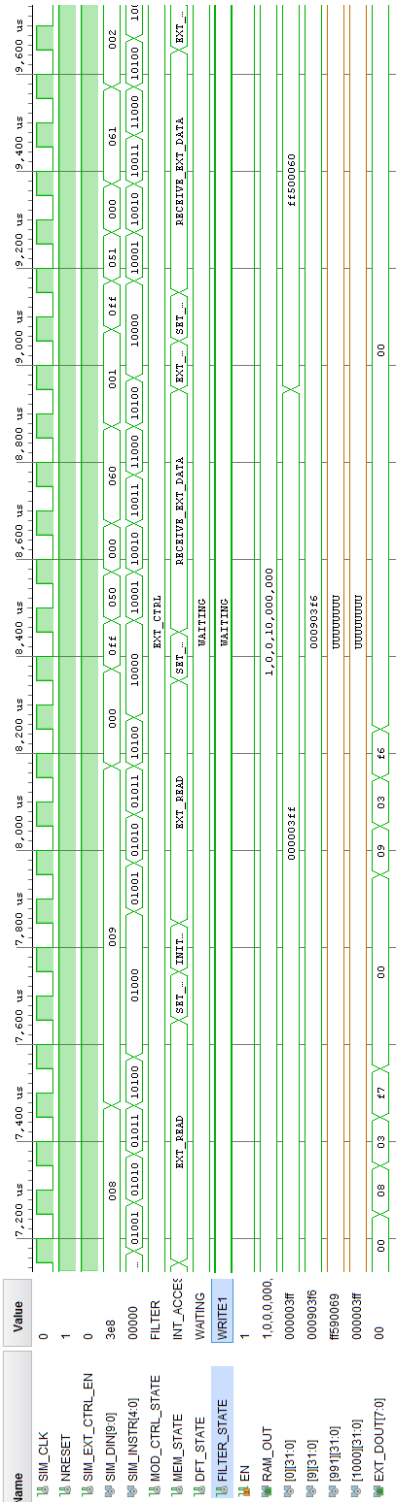

Abbildung B.9: Signalablauf-Diagramm der Simulation des Testsystems, 7200-9600 µs

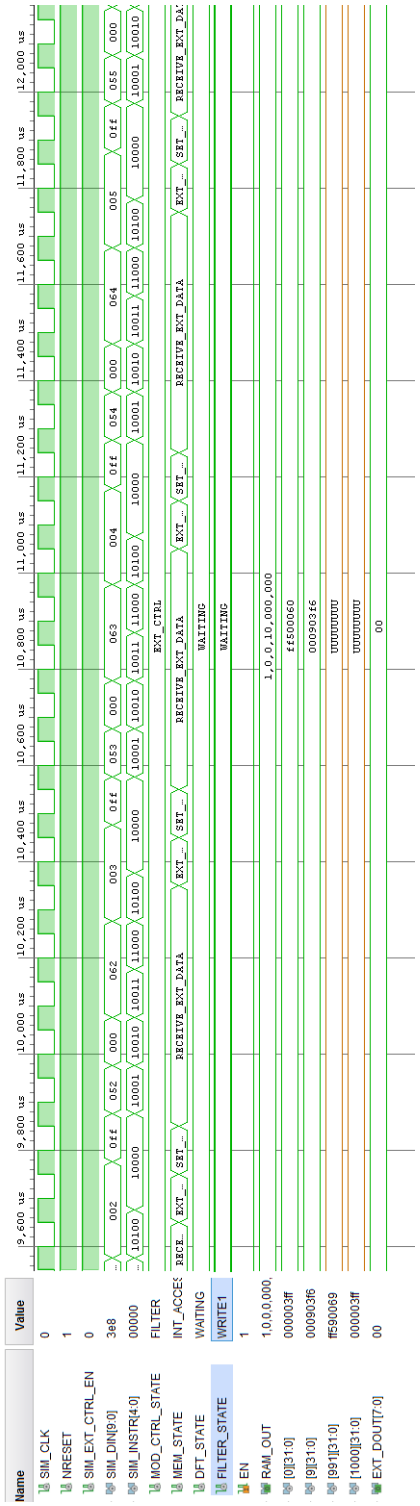

Abbildung B.10: Signalablauf-Diagramm der Simulation des Testsystems, 9600-12000 µs

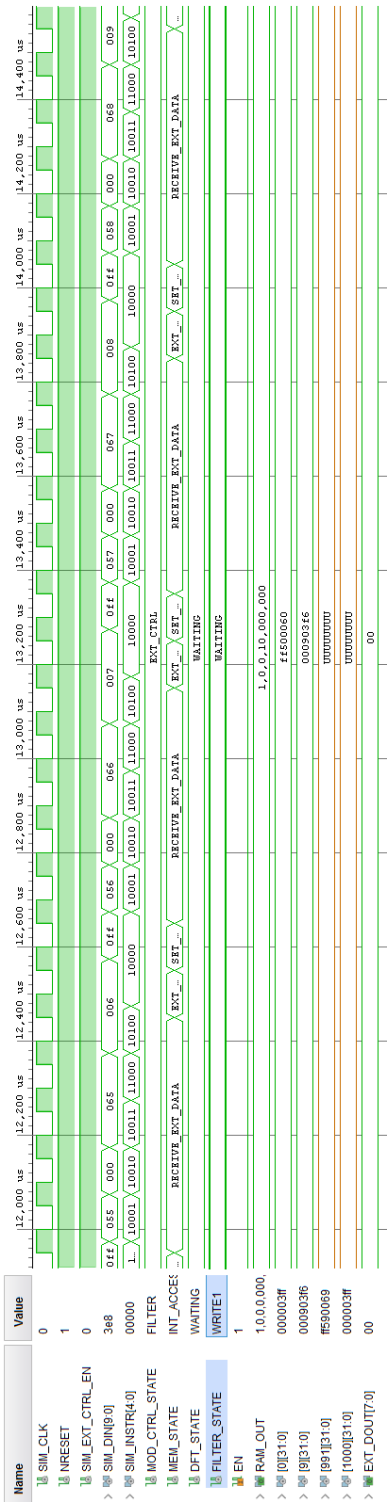

Abbildung B.11: Signalablauf-Diagramm der Simulation des Testsystems, 12000-14400 µs

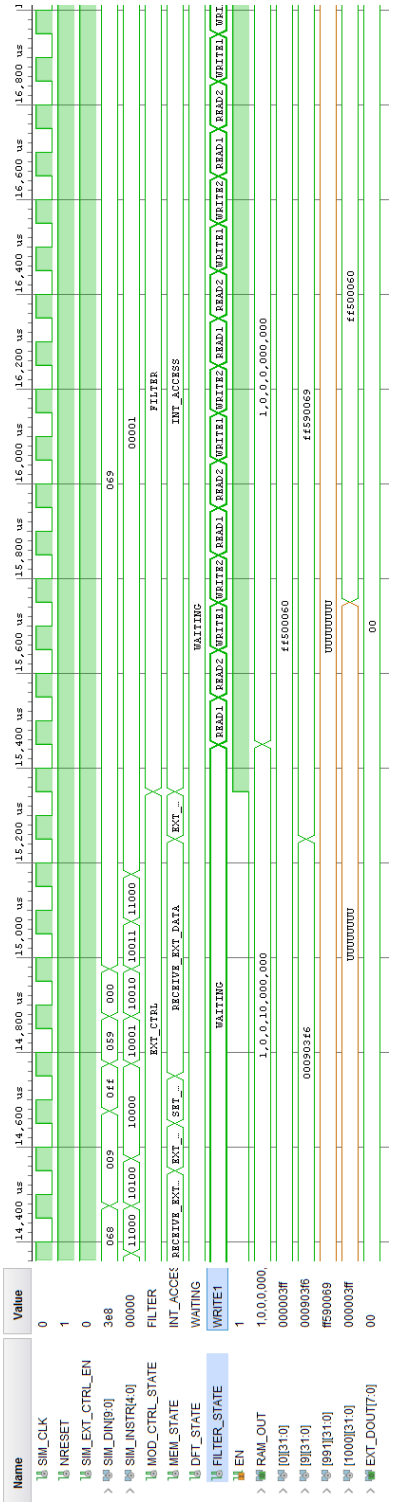

Abbildung B.12: Signalablauf-Diagramm der Simulation des Testsystems, 14400-16800 µs

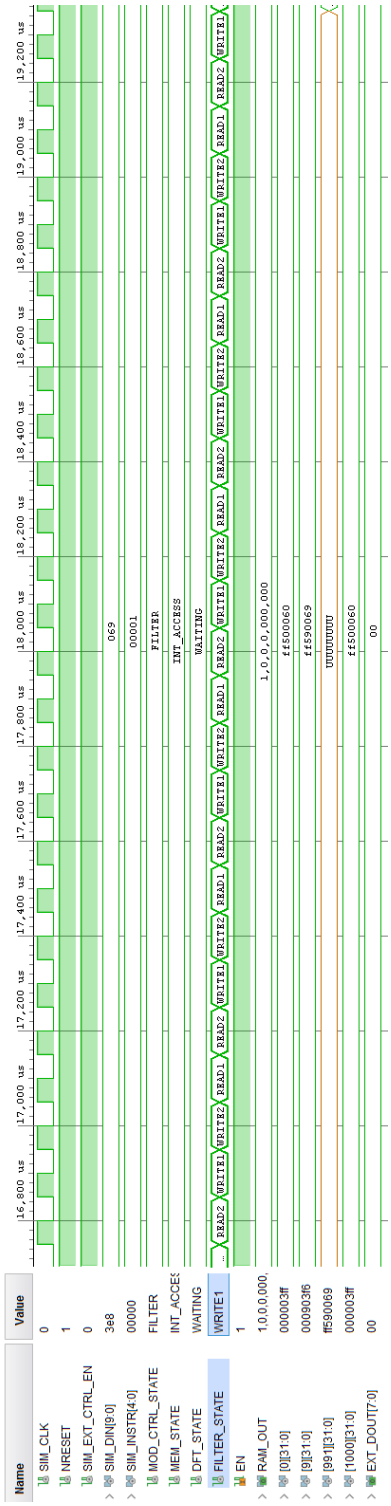

Abbildung B.13: Signalablauf-Diagramm der Simulation des Testsystems, 16800-19200 µs

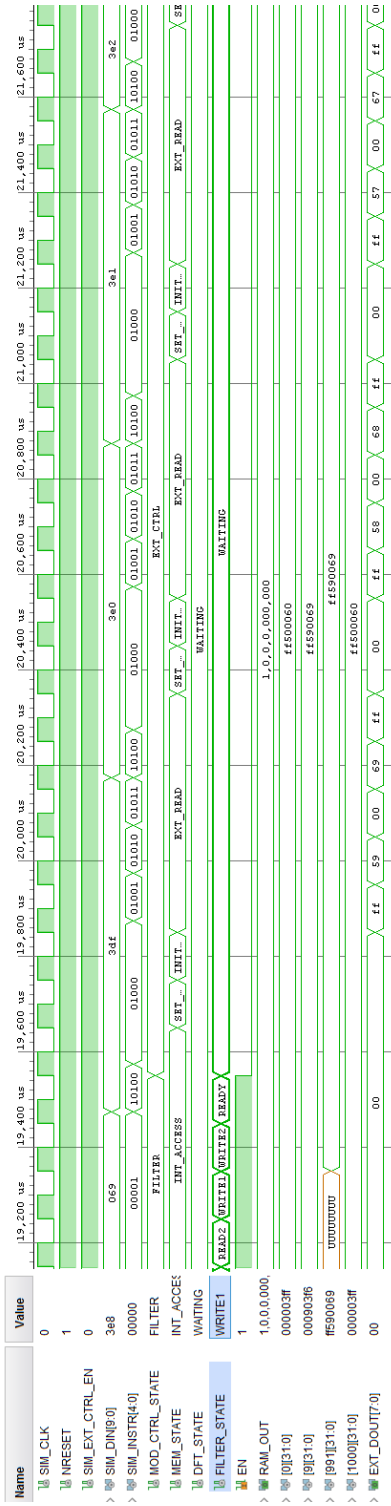

Abbildung B.14: Signalablauf-Diagramm der Simulation des Testsystems, 19200-21600 µs

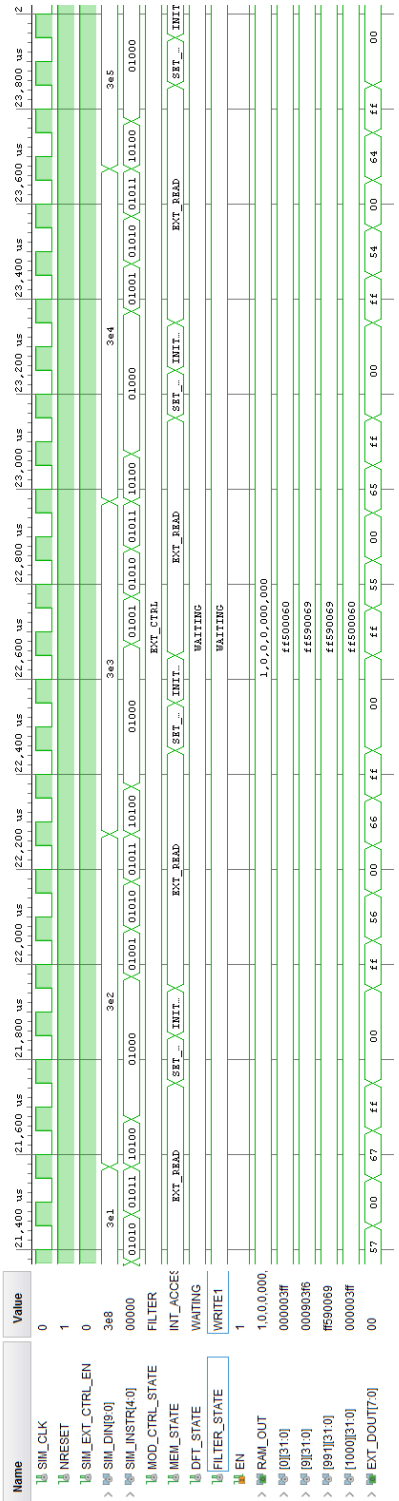

Abbildung B.15: Signalablauf-Diagramm der Simulation des Testsystems, 21400-23800 µs

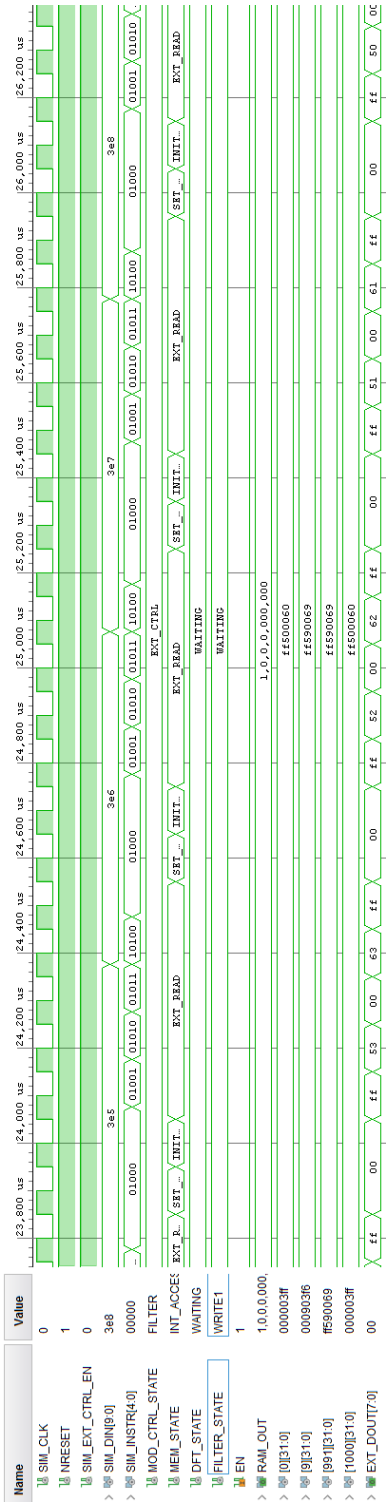

Abbildung B.16: Signalablauf-Diagramm der Simulation des Testsystems, 23800-26200 µs

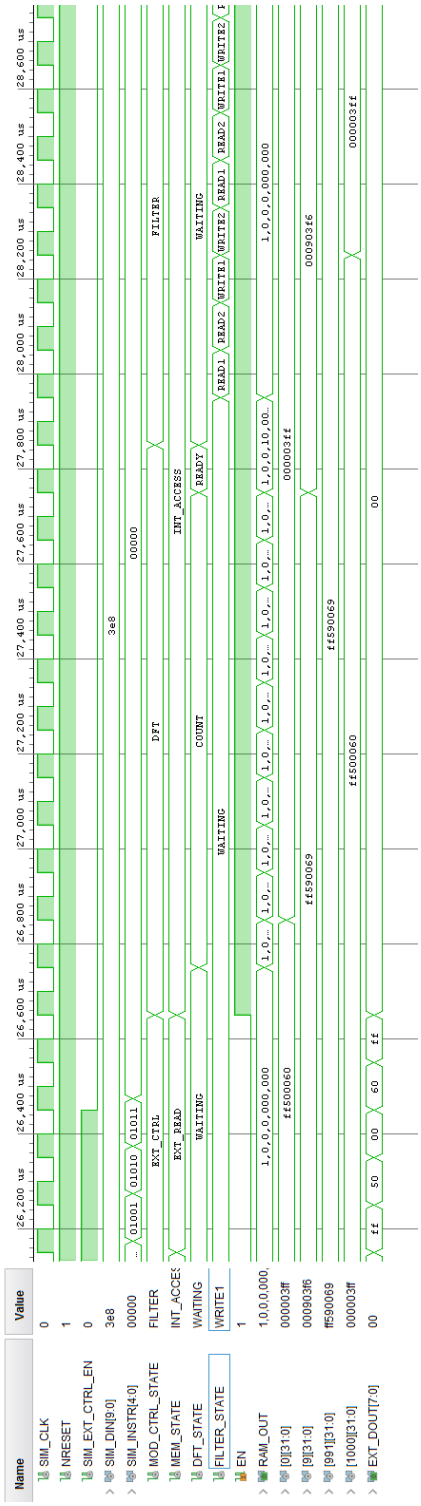

Abbildung B.17: Signalablauf-Diagramm der Simulation des Testsystems, 26200-28600 µs

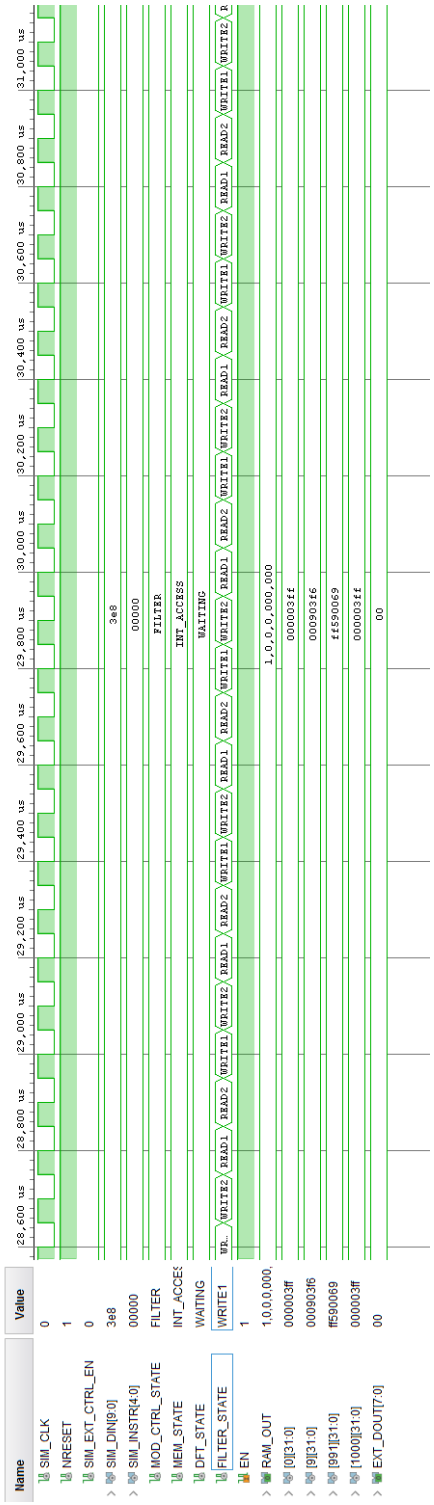

Abbildung B.18: Signalablauf-Diagramm der Simulation des Testsystems, 28600-31000 µs

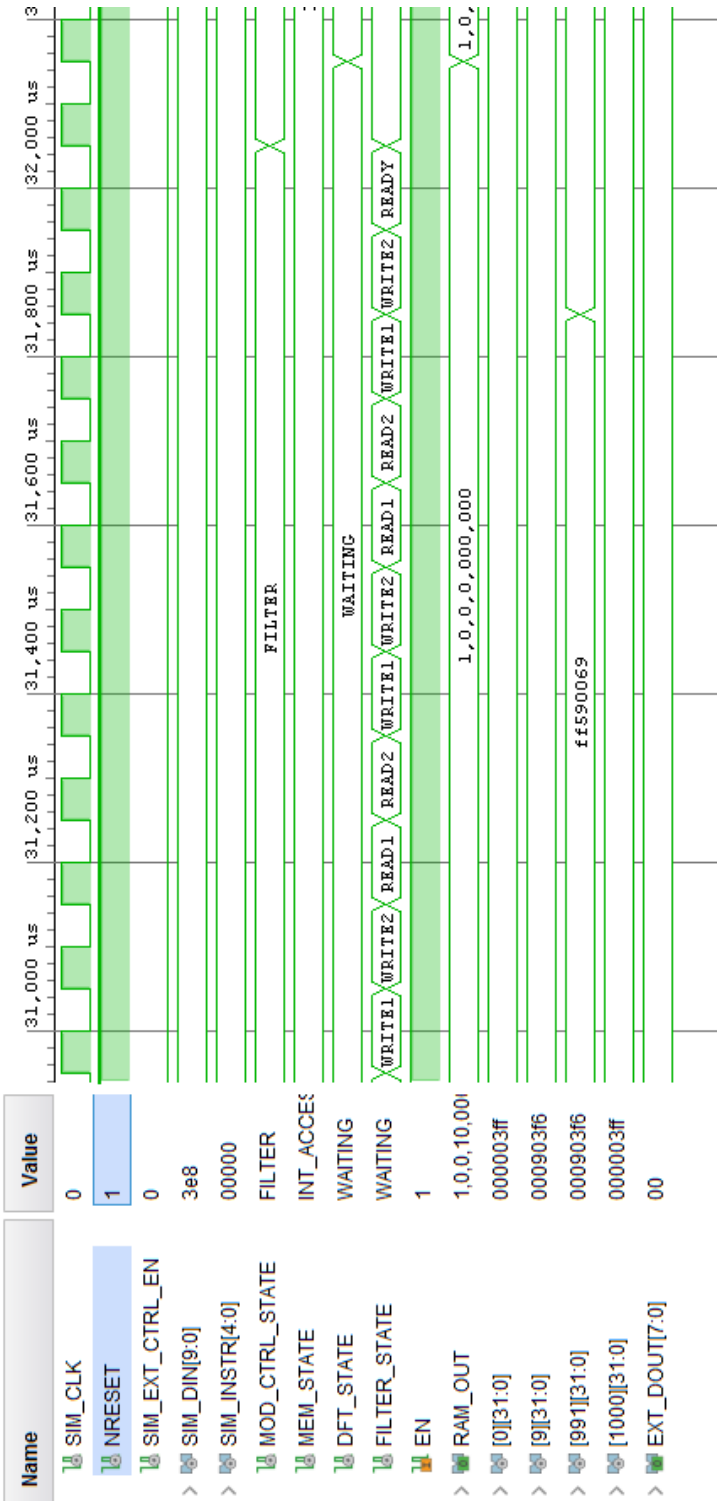

Abbildung B.19: Signalablauf-Diagramm der Simulation des Testsystems, 31000-32000 µs

B.1.4 Simulation von 2D-DFT

Die Simulation wurde aufgrund von rückständiger Unterstützung von VHDL-2008 durch Vivido-Simulator in Mentor Graphic ModelSim PE 10.4a durchgeführt. Glücklicherweise wurd ModelSim durch Vivado unterstützt, so dass man die Simulation in ModelSim von Vivado aus (bei entsprechender Einstellung) starten kann, was sie erheblich erleichtert.

Zuerst wurden die beiden Architekturen von dem ROM mit der 8×8 Twiddlefaktor-Matrix gleichzeitig instantiiert und simuliert. Sie müssen identisch auf Stimuli reagieren. Das Testbench ist in `twiddle_rom_sim.vhd` in sim_twiddle_rom. ModelSim vermag `sfixed`-Typen korrekt als Fixkommazahlen zu interpretieren und als Dezimalzahlen auf Signalablauf-Diagrammen darzuastellen.

Tabelle B.4: Testfälle für Twiddle-Faktoren-ROM

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
1: Read-Enable	Der Addressbus wurd auf 12 gesetzt, REN bleibt jedich auf '0'. Der Ausgang dard sich nicht ändern.	ja	0-150
2: Lesen aus dem ROM	Nun wird REN auf "high" gesetzt. Der Ausgang ändert sich auf den Twiddle-Faktor in der Zelle 12	ja	150-250
3: Lesen eines anderen Wertes	DEr Adressbus wird auf den Wert 23 gesetzt. Der Ausgang ändert sich auf den Twiddle-Faktor in der Zelle 23	ja	250-350
4: Auslesen der ganzen Twiddlefaktor-Matrix	Der ganze ROM wird konsekutiv durchgegangen. Der Ausgang gibt entsprechende Werte aus.	ja	250-6750

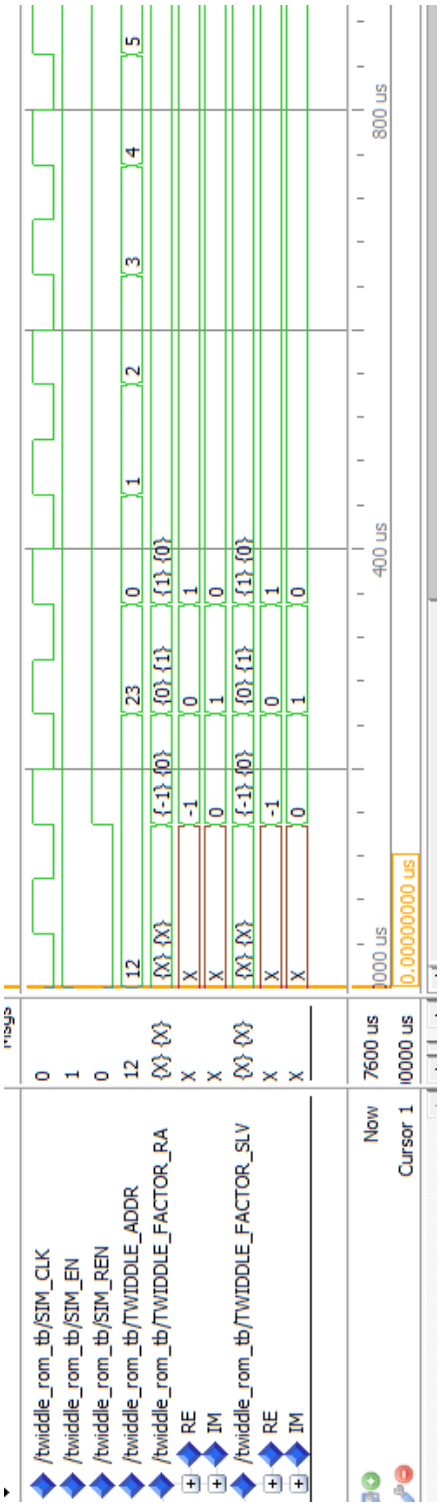

Abbildung B.20: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 0-850 µs

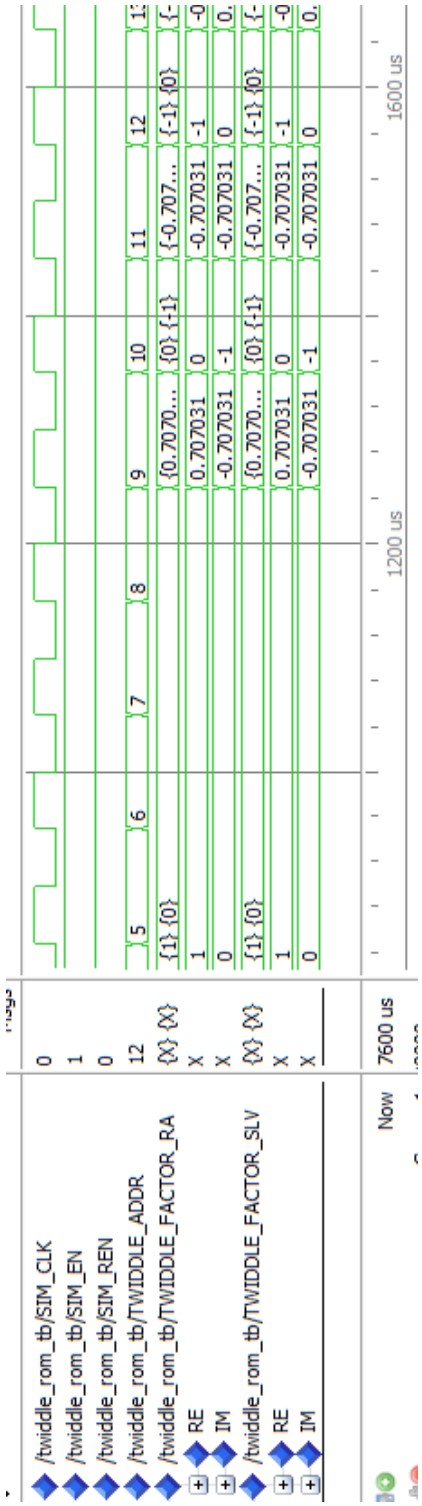

Abbildung B.21: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 850-1650 µs

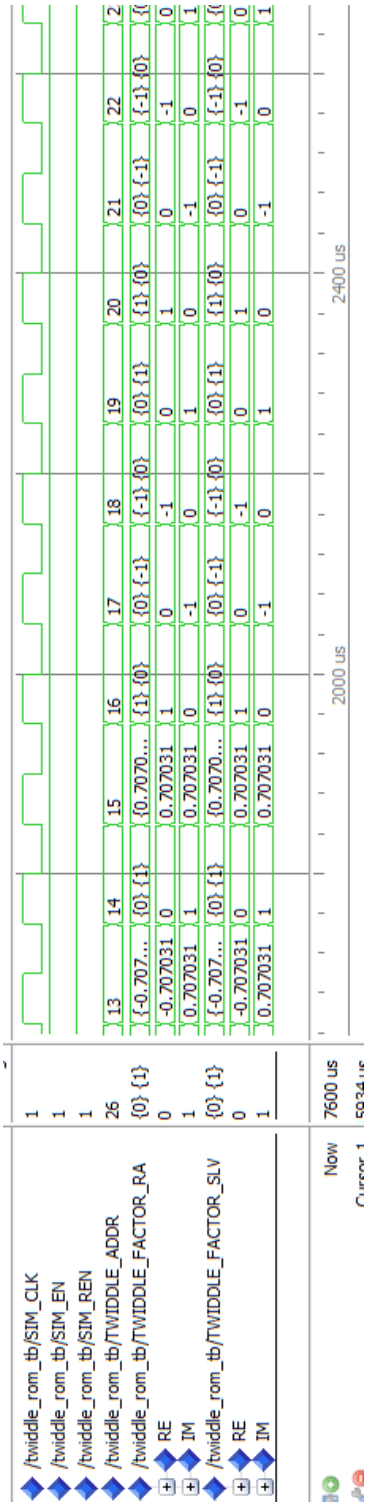

Abbildung B.22: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 1650-2450 µs

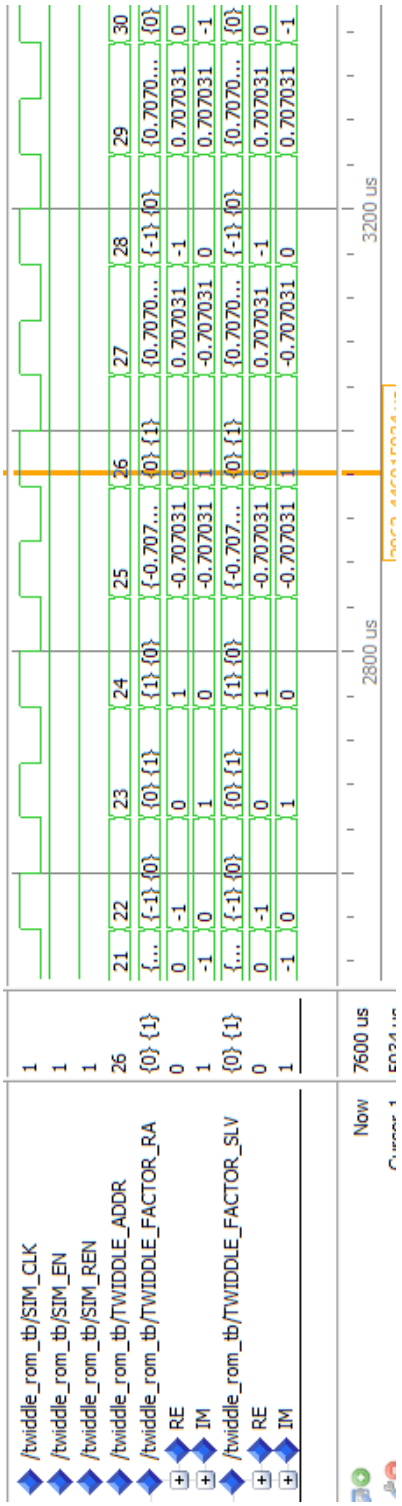

Abbildung B.23: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 2450-3250 µs

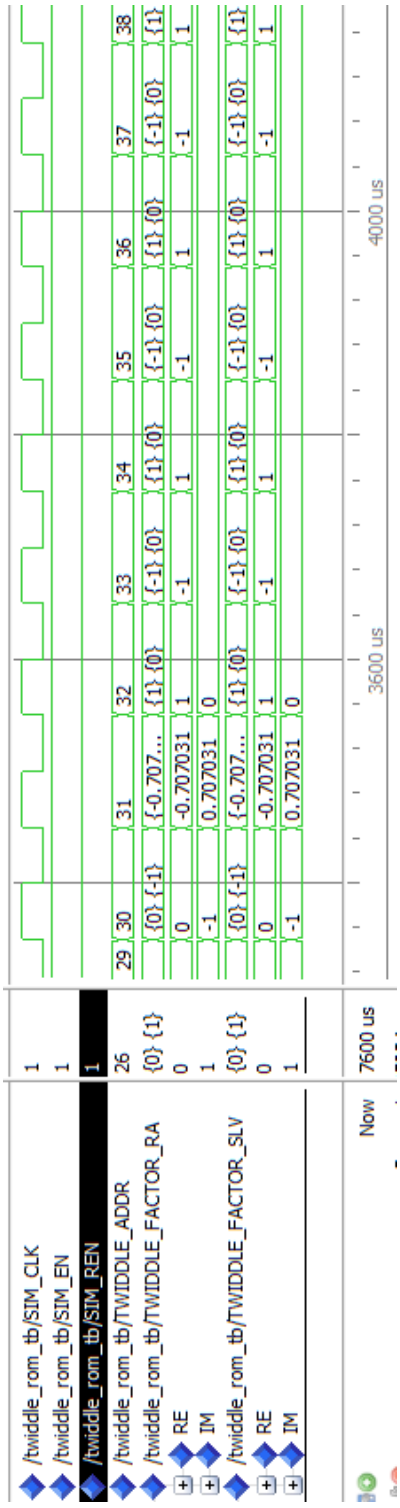

Abbildung B.24: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 3250-4050 µs

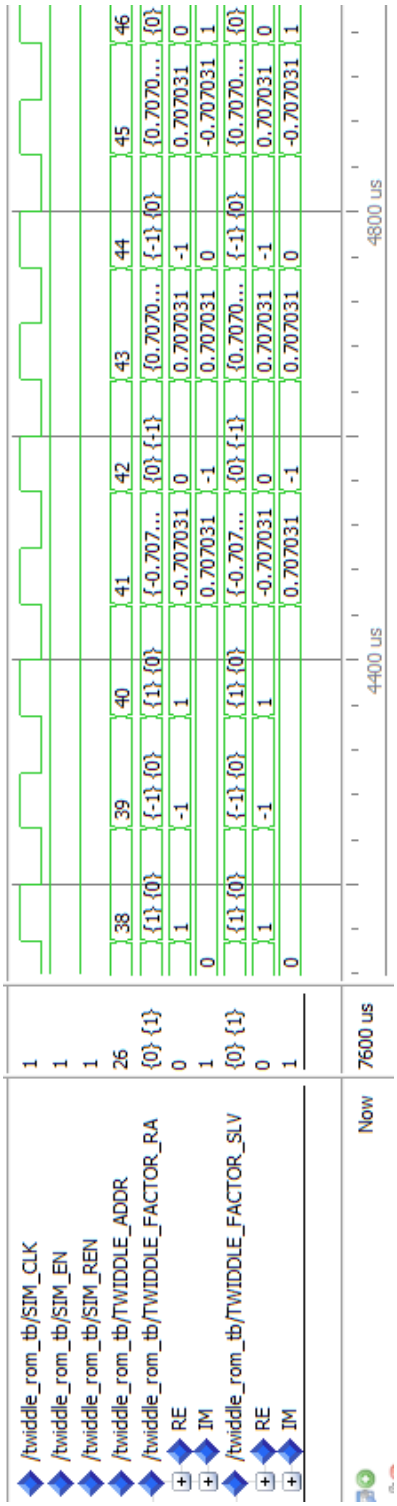

Abbildung B.25: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 4050-4850 µs

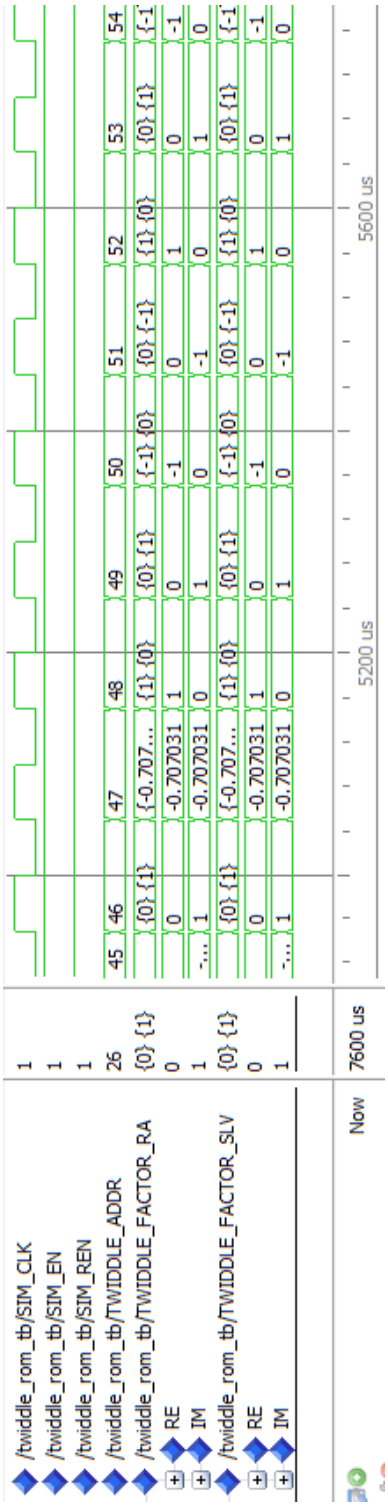

Abbildung B.26: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 4850-5650 µs

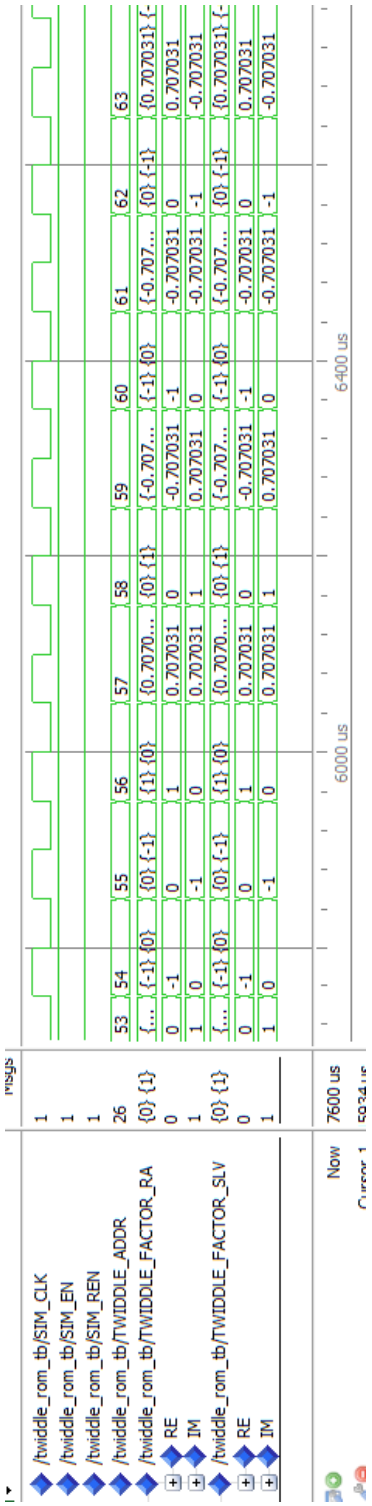

Abbildung B.27: Signalablauf-Diagramm der Simulation des Twiddle-Faktoren-ROMs, 5650-6750 µs

Die Einzelsimulation von `dft_2d` würde benötigen, dass auch alles Kontroll- und Eingangssignale imitiert werden sollten. Das ist sehr umständlich, deshalb wurde es innerhalb des Gesamtsystems simuliert. Am Anfang werden die Testdaten (simuliert) in die RAM geladen, weswegen die Simulation erst bei 45350 µs startet. Für die Verhaltenssimulation sind die konkrete Werte unwichtig. Es wurden von 1 aufsteigende Werte geladen, wodurch die RAM-Zellen effektiv durchnummeriert werden. So kann man in Simulation leicht erkennen, aus welche Zelle geladen wurde, was die Verifikation der Funktion von Speicheriteratoren sehr erleichtert. Darüber hinaus erkennt ModelSim die im Design vorhandene System-RAM, so dass ihre Inhalt in der Memory-Ansicht unmittelbar beobachtet werden kann.

Tabelle B.5: Testfälle für 2D-DFT

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Start der 1D-DFT	Mit dem Signal `START` vom Modulkontroller geht `dft_2d` in den Zustand `DFT1` über und verbleibt in diesem. Daraufhin wechselt `dft_mat_product` in den Zustand `READ`	ja	45350-45550
Umschalten von Adressbussen (“low”)	Wenn `SWITCH_SIDES` ’0’ ist, ist der linke Speicheriterator `LEFT_ADDR` an ROM angebunden, und der rechte `RIGHT_ADDR` an die RAM (`RAM_OUT.ADDR`).	ja	45350-46650
Speicheriteratoren	Der linke Speicheriterator `LEFT_ADDR` schreitet reihenweise voran, das bedeutet er wird um 1 inkrementiert. Der rechte Speicheriterator `RIGHT_ADDR` schreitet spaltenweise voran, das bedeutet er wird um 8 inkrementiert.	ja	45750-46550

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.5

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Akkumulator	Die Werte werden in dem Akkumulator ACCUM korrekt akkumuliert.	ja	45850-46550
Abbruch der Akkumulatorschleife	Wenn der linke Speicheriterator LEFT_ADDR das Ende der Reihe erreicht hat, wird die Akkumulierenschkeife abgebrochen und der Zustand ändert sich auf WRITE1.	ja	47250
Speichern der Ergebnis	Im Zustand WRITE1 wird Read-Enable REN abgeschaltet und Write-Enable RAM_OUT.WEN hochgesetzt. Der RAM-Adressebus RAM_OUT.ADDR wird auf den Ausgangsiterator OUT_ITER umgeschaltet. In WRITE2 wird dann der gerundete Wert aus dem Akkumulator in RAM platziert.	ja	47250-47450
Übergang zur nächsten Spalte	Der Spalteniterator COLUMN_ITER wird um 1 incrementiert. Der rechte Speicheriterator RIGHT_ADDR wird auf diesen Wert gesetzt. Der linke Speicheriterator LEFT_ADDR wird auf den Wert im Reiheniterator ROW_ITER zurückgesetzt.	ja	47450-47550
Ausgangsiterator	Der Ausgangsiterator OUT_ITER wird um 1 inkrementiert.	ja	47450

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.5

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Zurücksetzen vom Akkumulator	Der Akkumulator wird auf 0 zurückgesetzt	ja	47550-47650
Übergang zur nächsten Reihe	Wenn der Spalteniterator `COLUMN_ITER` die letzte Reihe erreicht hat, wird er zusammen mit dem rechten Speicheriterator `RIGHT_ADDR` auf die rechte Startadresse `RIGHT_START_ADDR` zurückgesetzt. Der Reiheniterator `ROW_ITER` wird um 8 inkrementiert und der linke Spalteniterator `LEFT_ADDR` wird auf diesen Wert gesetzt. Der Ausgangsiterator `OUT_ITER` wird auch um 1 inkrementiert.	ja	60650-60850
Ende der Matrizenmultiplikation	Wenn der Reiheniterator `ROW_ITER` die letzte Reihe und der Spalteniterator `COLUMN_ITER` die letzte Spalte errecht haben, wird die Berechnung beendet. Das Modul `dft_mat_product` wechlelt in den Zustand `READY` und `READY_OUT` (gleich wie `PRODUCT_READY`) wird für einen Takt hochgesetzt.	ja	167050-167250

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.5

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, μs**
Übergang von 1D-DFT zur 2D-DFT	Wenn PRODUCT_READY) hochgesetzt und dft_2d geht in den Zustand DFT2 über. Die Matrizenmultiplikation startet erneut mit anderen Werten von dem Reiheniterator ROW_ITER und Spalteniterator COLUMN_ITER. SWITCH_SIDES wird auf '1' gesetzt.	ja	167150-167450
Umschalten von Adressbussen ("high")	Wenn SWITCH_SIDES '1' ist, ist der linke Speicheriterator LEFT_ADDR an RAM (RAM_OUT.ADDR) angebunden, und der rechte RIGHT_ADDR an die ROM .	ja	167350-168000
1D-DFT Werte	An den RAM-Datenausgängen DATA_IN.RE und DATA_IN.IM erscheinen die zuvor gespeicherten Werte (47250-47450 μs) von 1D-DFT.	ja	167150-167450
Ende der Gesamtberechnung	Wenn dft_mat_product zum zweiten Mal READY_OUT ausgibt und in den Zustand IDLE übergeht, wechselt dft_2d in den Zustand READY und gibt seinerseits READY_OUT (gleich wie MOD_OUT) aus. Anschließend wechselt es zu IDLE.	ja	288950-289200

Abbildung B.28: Signalablauf-Diagramm der Simulation der 2D-DFT, 45350-46650 µs

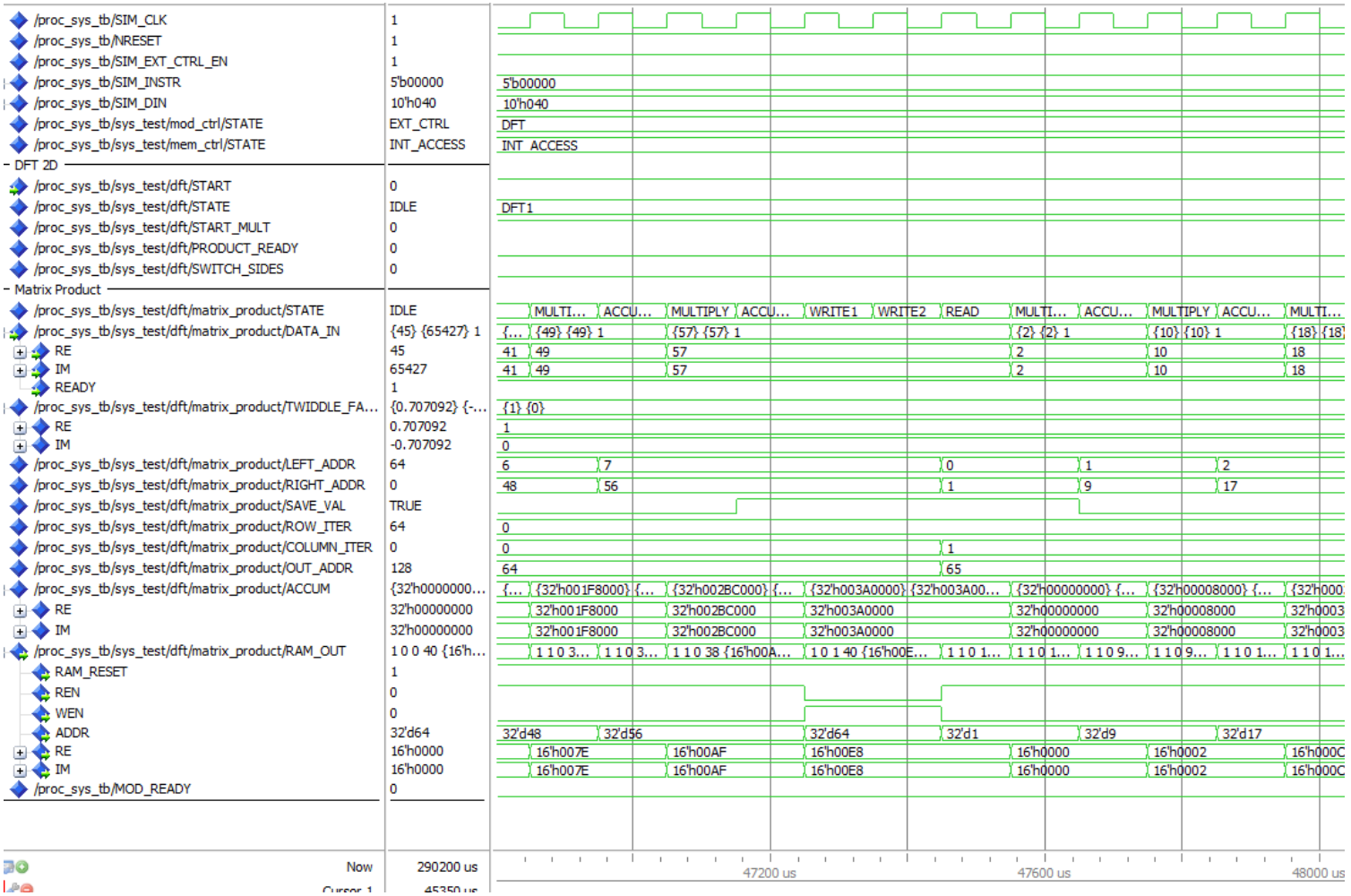

Abbildung B.29: Signalablauf-Diagramm der Simulation der 2D-DFT, 46850-48000 µs

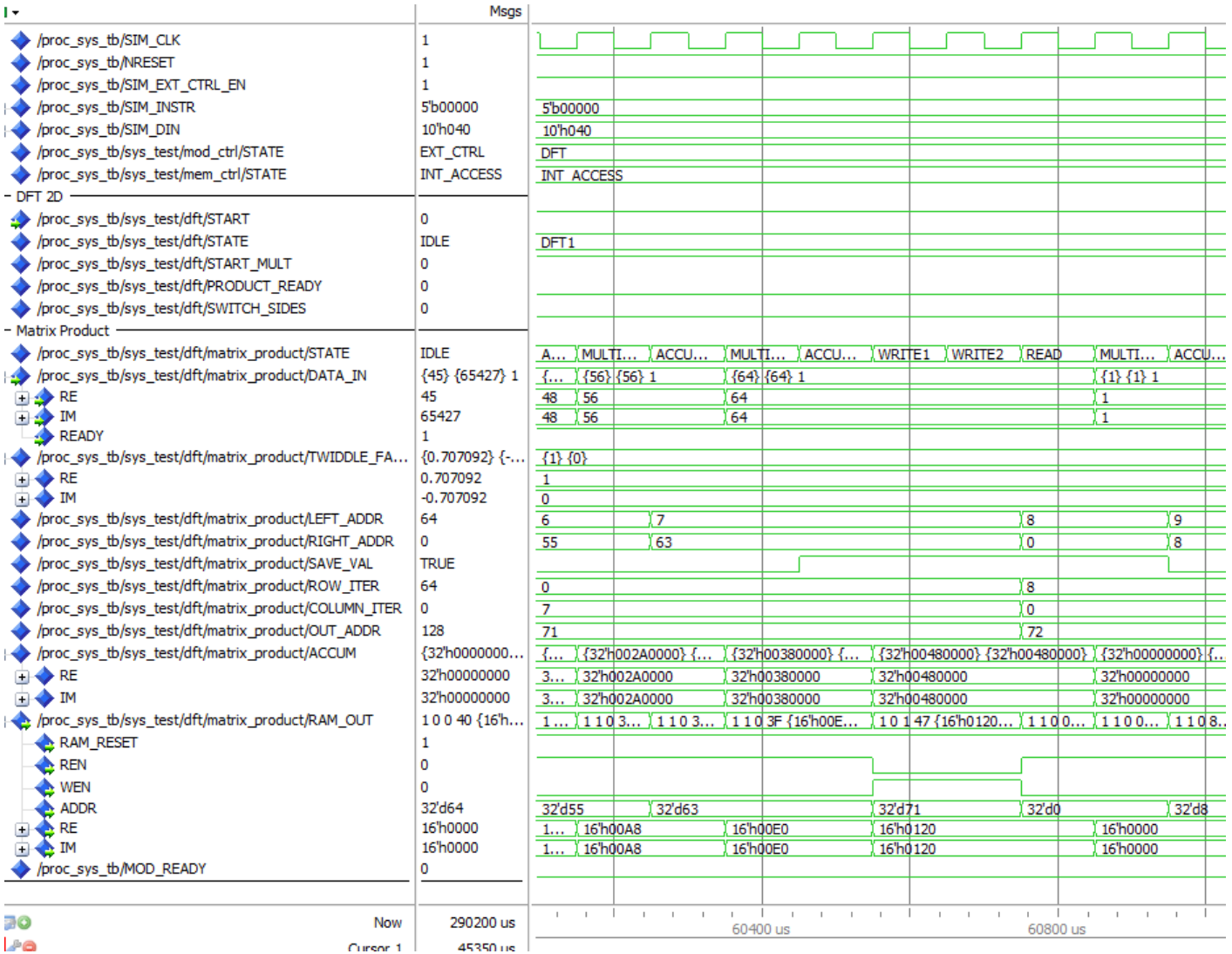

Abbildung B.30: Signalablauf-Diagramm der Simulation der 2D-DFT, 60150-61000 µs

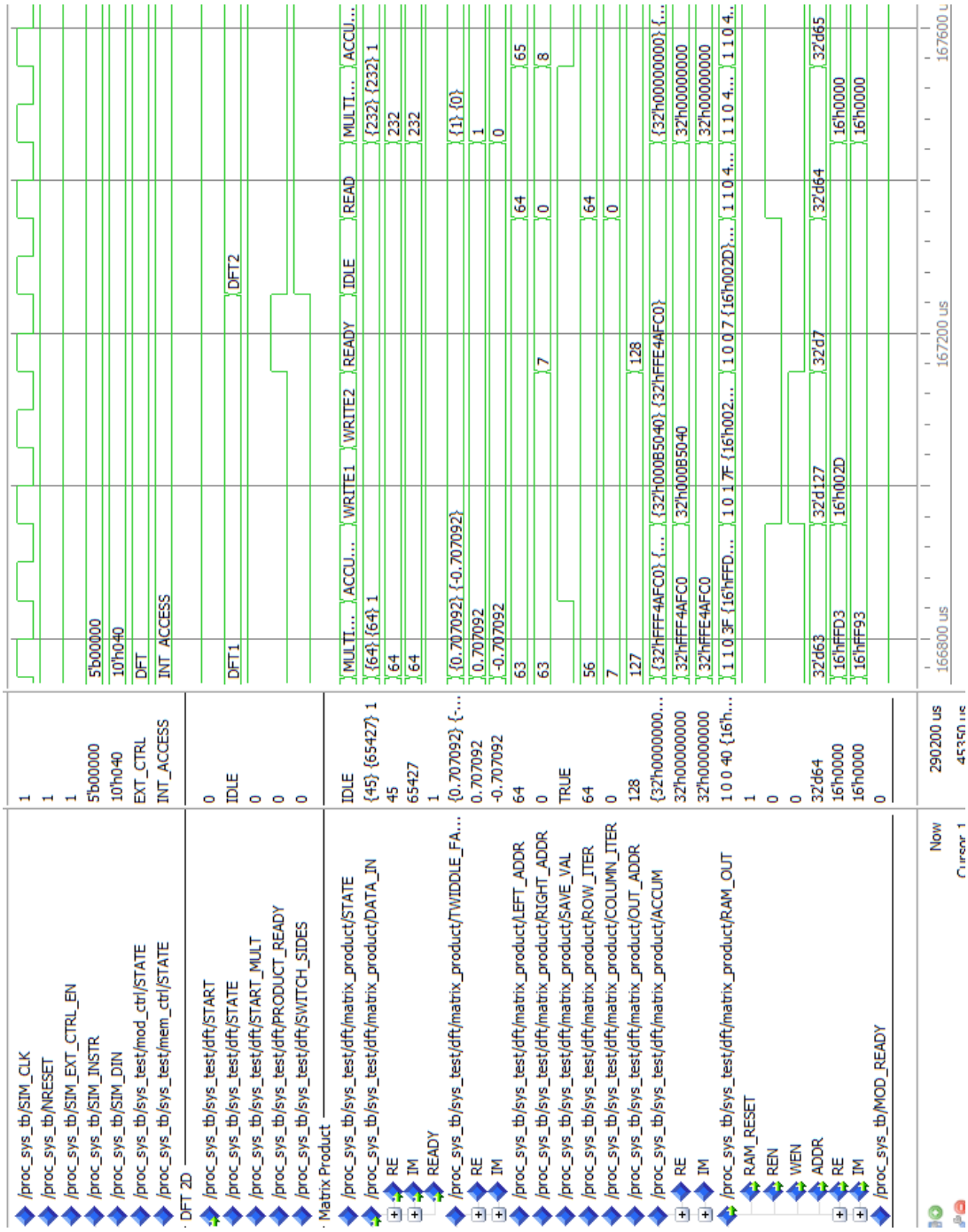

Abbildung B.31: Signalablauf-Diagramm der Simulation der 2D-DFT, 166750-167600 µs

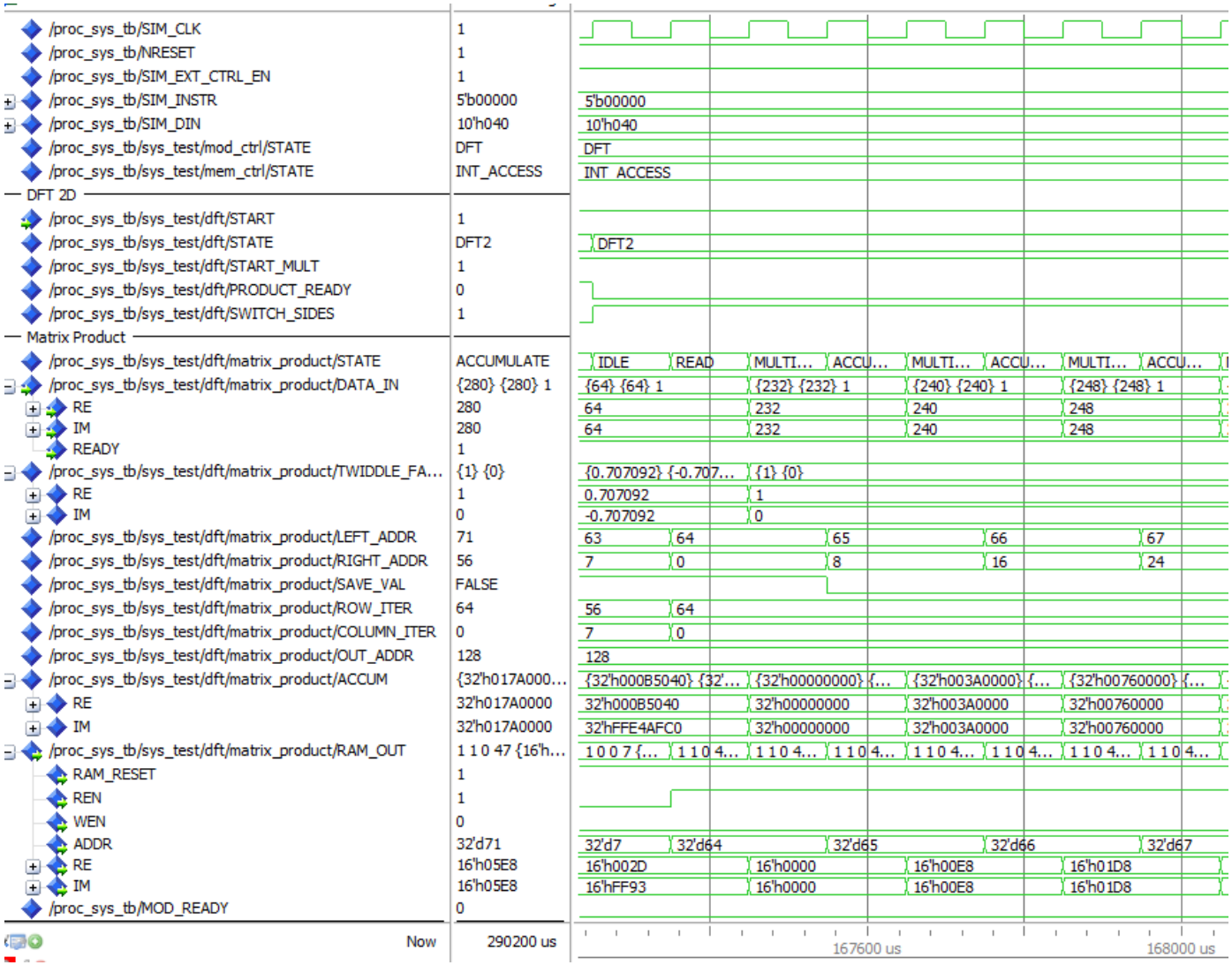

Abbildung B.32: Signalablauf-Diagramm der Simulation der 2D-DFT, 167250-168000 µs

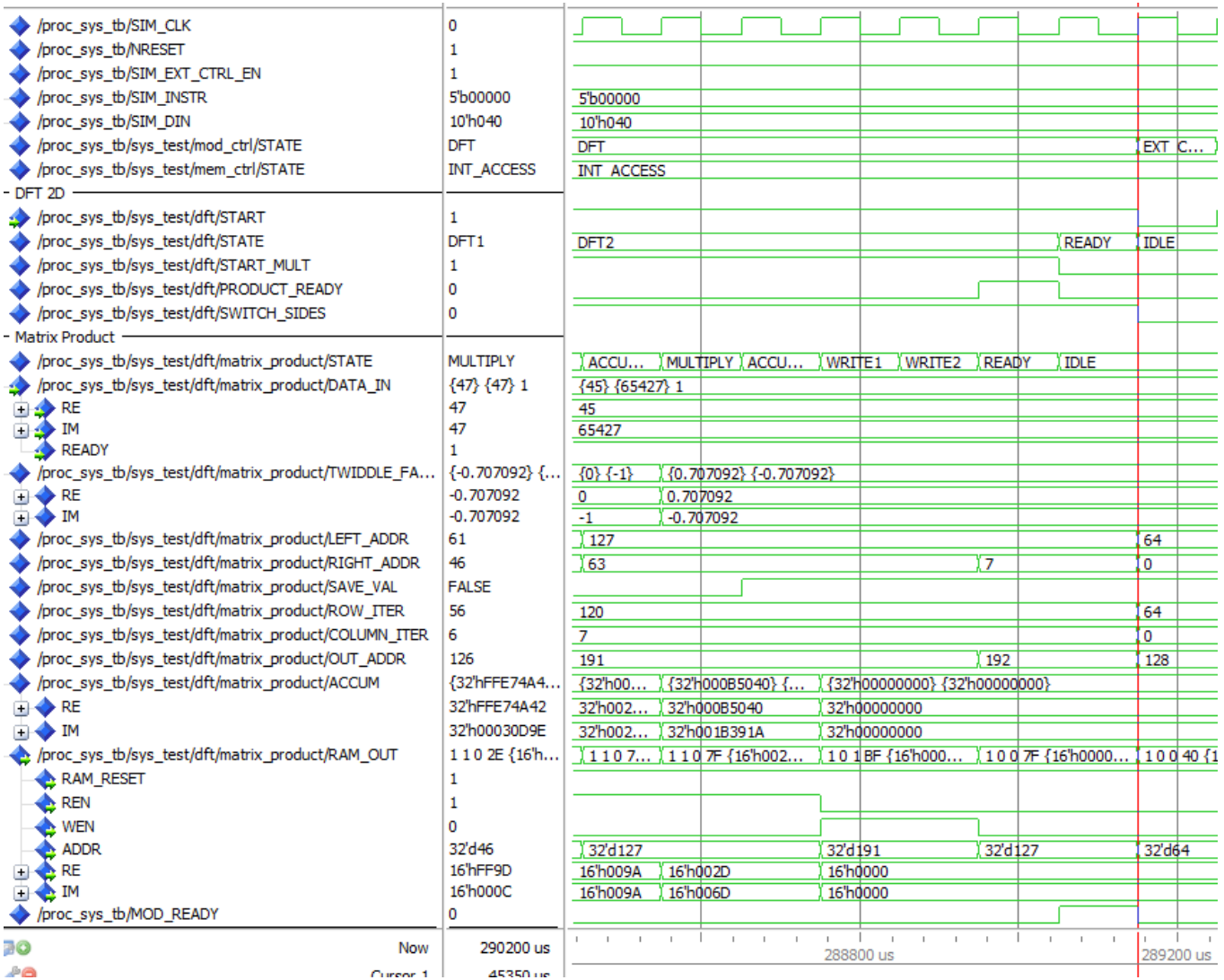

Abbildung B.33: Signalablauf-Diagramm der Simulation der 2D-DFT, 288450-289200 µs

B.1.5 Simulation von der Filterung um Ortsfrequenzbereich

Das Modul `filter_2d` wurde genau wie und aus denselben Gründin wie `dft_2d` als Teil des Gesamtsystems ebenfalss mit ModelSim simuliert. Es wurden zwar die beiden Architekturen simuliert, die Simulationen sind aber weitgehen gleich, deshab nur die von beiden komplexer `COEF_RAM` wird hier dokumentiert. Das Testbench befindet sich in der Datei `filter_behav_tb.vhd` aus Simulation-Set `sim_filter_behav`. In der Simulation werden zuerst Koeffizienten und Daten generiert und in die RAM geladen, deswegen fängt die eigentliche Simulation bei 333950 µs

Tabelle B.6: Testfälle für die `filter_2d.(COEF_RAM)`

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Initialisierung	Im Zustand `IDLE` wird `ADDRESS_ITER` auf `DFT2_START` (128) initialisiert	ja	333950-334050
Start der Filterung		ja	334050-334150
Laden der Filterkoeffizienten	Im Zustand `LOAD_FILTER_COEF` werden die Filterkoeffizienten in Registern `FILTER_COEF_RE` und `FILTER_COEF_IM` gespeichert. `RAM_OUT.ADDR` wird auf die Startadresse von 2D-DFT (128) gesetzt.	ja	334150-334250
Erste Multiplikationsphase	Im Zustand `MULT1` wird der Produkt der Realteilen in Zwischenregitern `REGS` geschrieben.	ja	334250-334350
Zweite Multiplikationsphase	Im Zustand `MULT2` die zweite Phase gemäß der Formel für komplxe Multiplikation wird korrekt ausgeführt und `REGS` bekommen Endergebnis	ja	334350-334450

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.6

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Speichern einer Ergebnis	In `WRITE1` werden `REN` auf '0' und `RAM_OUT.WEN` auf '1' gesetzt. `RAM_OUT.ADDR` wird auf die Startadresse des gefilterten Daten (192) gesetzt. Beim Wechsel zu `WRITE2` wird der Inhalt von `REGS` gerundet in die RAM geschrieben	ja	334450-334650
Übergang zur nächsten Iteration	Wenn `ADDRESS_ITER` kleiner als die Adresse des letzten Elements ist, wird es inkrementiert und das Modul wechselt in den Zustand `READ`.	ja	334550-334750
Ende der Berechnung	Wenn `ADDRESS_ITER` gleich der Adresse des letzten Elements ist (191), wechselt `filter_2d` in den Zustand `READY`. Dann wird `READY_OUT` hochgestzt, und das Modul wechselt zu `IDLE`.	ja	372350-372600

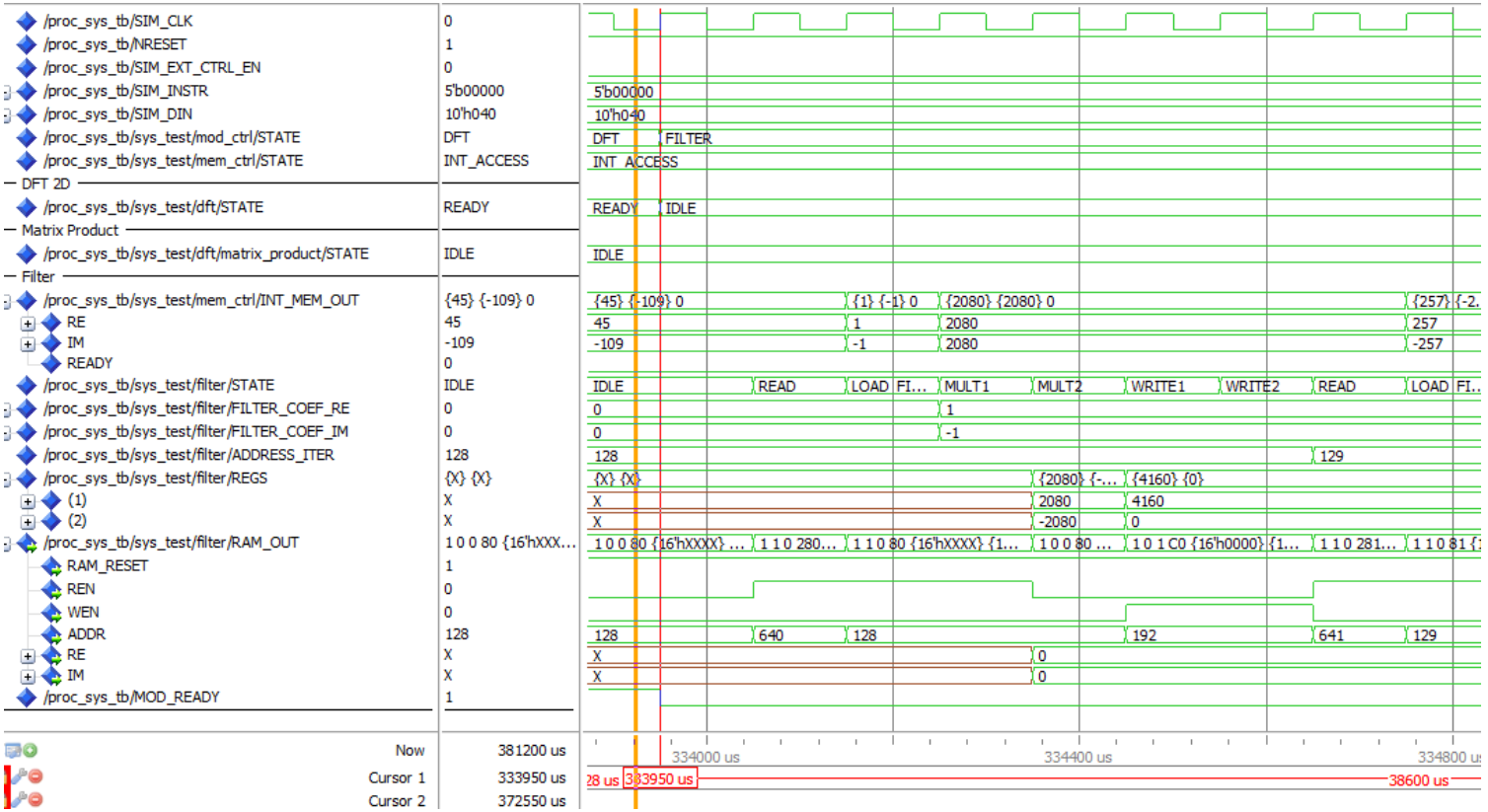

Abbildung B.34: Signalablauf-Diagramm der Simulation des 2D-Filters im Ortsbereich, 333950-334800 µs

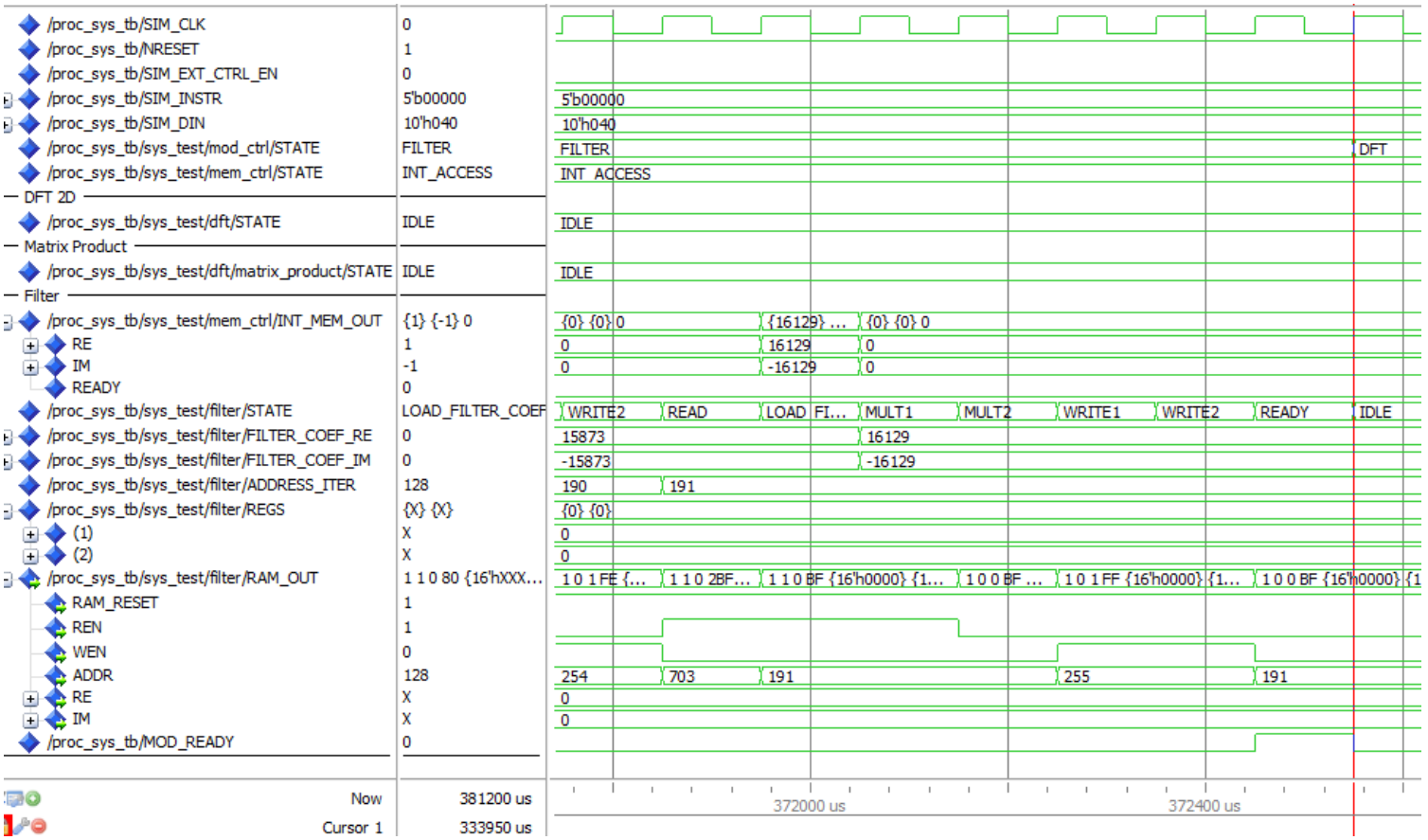

Abbildung B.35: Signalablauf-Diagramm der Simulation des 2D-Filters im Ortsbereich, 371750-372600 µs

B.1.6 Simulation von der Filterung über Faltung

Die beiden Architekturen wirden mit ModelSim als Teil des Gesamtsystems simuliert. Da die Simaltion ähnlich sind werden sie in Tabelle B.7 zusammen dargestelltm wo es gekennzeichnet wird, für welche Architektur der Testfall gilt. Die Simulation von der Architektur `CONV_ROM` befindet sich im Simulationset `sim_conv_behav` und von der Architektur `CONV_RAM` im Simulationset `sim_conv_ram_behav`

Tabelle B.7: Die wichtigste Testfälle für die `filter_conv`

Testfall	**Architektur**	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Initialisierung	beide	Im Zustand `IDLE` werden alle Iteratoren korrekt initialisiert.	ja	`CONV_RAM`: 79800-79850 `CONV_ROM`: 45400-45550
Start der Faltung	beide	Mit dem Signal `START` vom Modulkontroller geht `filter_conv` in den Zustand `READ` über. Dabei wird `REN` auf “high” gesetzt.	ja	`CONV_RAM`: 79850-79950 `CONV_ROM`: 45550-45650
Laden der Filterkoeffizienten	`CONV_RAM`	Im Zustand `LOAD_COEF` werden die Filterkoeffizienten im Register `COEF` gespeichert. `RAM_OUT.ADDR` wird auf die Startadresse von Daten (0) gesetzt.	ja	79950-80050
Akkumulieren	`COEF_RAM`	Nach dem Zustand `LOAD_COEF` werden die Speicheriteratoren `DATA_CURRENT_POS` und `KERNEL_CURRENT_POS` inkrementiert. Der Koeffizient `COEF` wird im Zustand `ACCUM` mit dem Dateneingang `RAM_IN` ausmultipliziert und akkumuliert.	ja	79950-80150

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.7

Testfall	Architektur	Testvorgang und zu erwartendes Verhalten	Bestanden?	Zeitintervall auf Signalablauf-Diagramm, µs
	COEF_ROM	Die Speicheriteratoren DATA_CURRENT_POS und KERNEL_CURRENT_POS im Zustand ACCUM werden inkrementiert. Gleichzeitig wird der Koeffizient COEF mit dem Dateneingang RAM_IN ausmultipliziert und akkumuliert	ja	45650-46000
Fertigstellung der Faltungswertes	COEF_RAM	Nach dem Erreichen durch Speicheriteratoren von den letzten überlappenden Reihe und Spalte wechselt filter_conv in den Zustand WRITE1 über	ja	83150
	COEF_ROM	Nach dem Erreichen durch Speicheriteratoren von den letzten überlappenden Reihe und Spalte wechselt filter_conv in den Zustand LAST_ACCUM, wo die letzte Multiplikation und Akkumulation ausgeführt wird. Anschließend geht das Modul in den Zustand WRITE1 über	ja	47050-47250

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.7

Testfall	**Architektur**	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Speichern einer Ergebnis	beide	In WRITE1 wirdRAM_OUT.WEN auf '1' gesetzt. RAM_OUT.ADDR wird auf den Ausgangsiterator OUT_POS gesetzt. Beim Wechsel zu WRITE2 wird der Inhalt von ACCUM_RE und ACCUM_IM gerundet in die RAM geschrieben	ja	CONV_RAM: 83150-83350 CONV_ROM: 47250-47450
Übergang zur nächsten Zentrierungsposition	beide	Falls noch nicht die letzte Spalte der Datenmatrix erreicht wurde, wird der Spalteniterator DATA_COL_CNT inkrementiert, anderfalls wird er auf 0 zurückgesetzt und der Reiheniterator DATA_ROW_CNT inkrementiert. Das Modul wechselt in den Zustand CALC_START.	ja	CONV_RAM: 83250-83450 CONV_ROM: 47350-47550
Berechnung der Startpositionen	beide	Im Zustand CALC_START werden DATA_ROW_START und KERNEL_ROW_START neu berechnet. Diese Werte werden auch den jeweiligen Speicheriteratoren zugewisen	ja	CONV_RAM:83350-83450 CONV_ROM: 47350-47550

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.7

Testfall	**Architektur**	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**	**Zeitintervall auf Signalablauf-Diagramm, µs**
Ende der Berechnung	beide	Falls der Reiheniterator und der Spalteniterator den letzten Wert errecht haben, wechselt `filter_conv` in den Zustand `READY`. Dann wird `READY_OUT` hochgestzt, und das Modul wechselt zu `IDLE`.	ja	`CONV_RAM`: 492450-492700 `CONV_ROM`: 264300-264800

Abbildung B.36: Signalablauf-Diagramm der Simulation der Architektur `CONV_RAM` von `filter_conv`, 79800-80400 µs

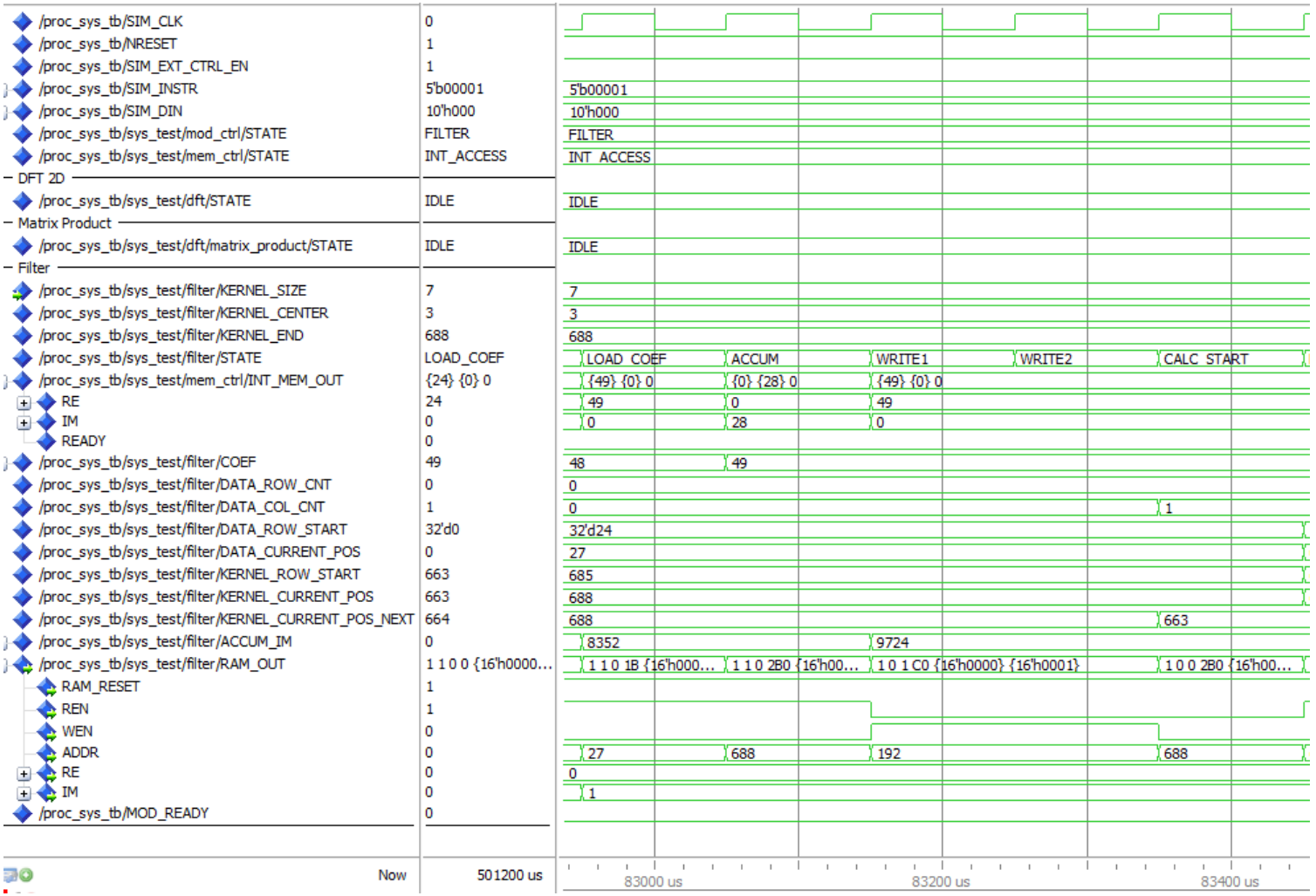

Abbildung B.37: Signalablauf-Diagramm der Simulation der Architektur `CONV_RAM` von `filter_conv`, 82950-83450 µs

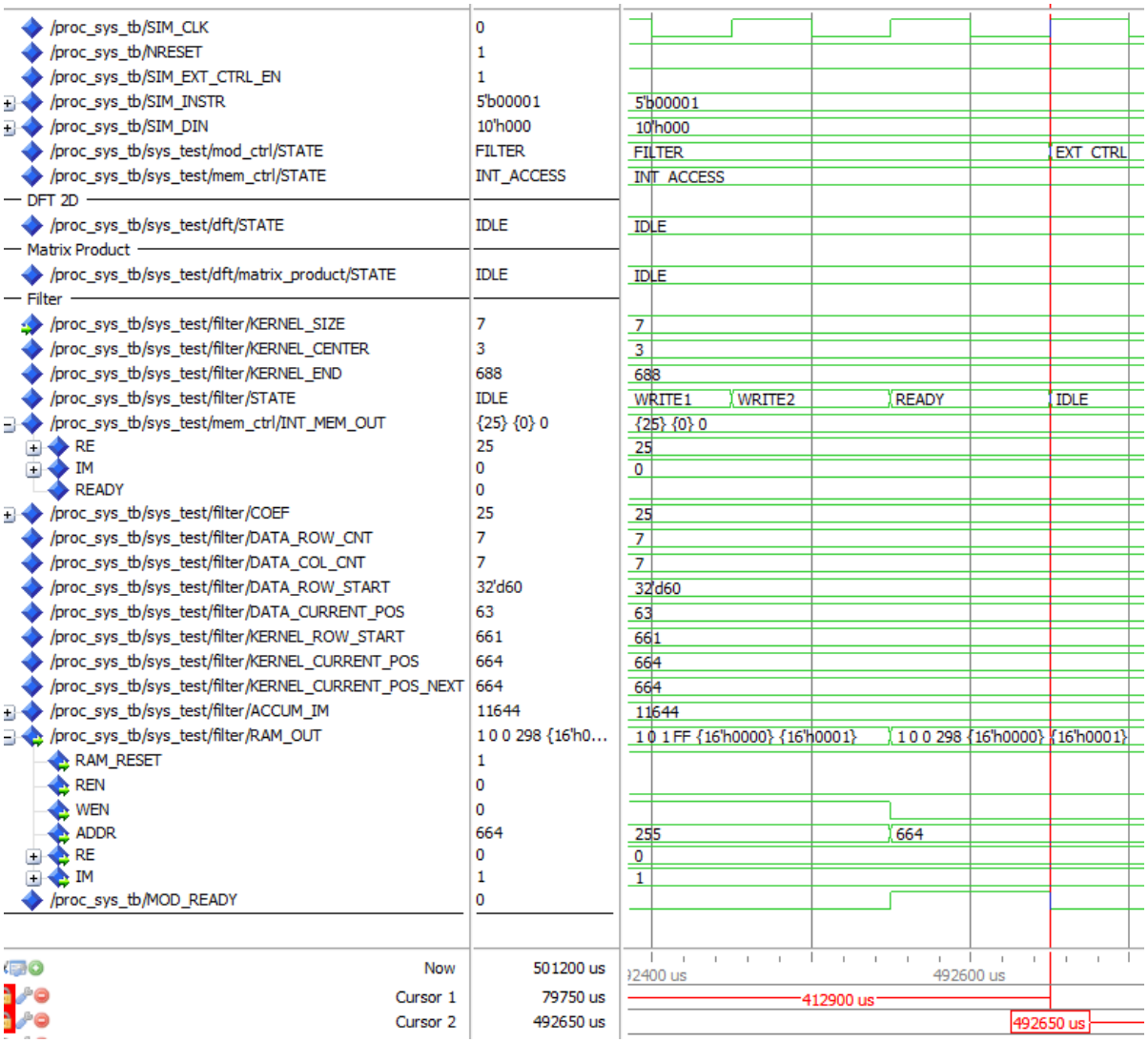

Abbildung B.38: Signalablauf-Diagramm der Simulation der Architektur `CONV_RAM` von `filter_conv`, 492400-492700 µs

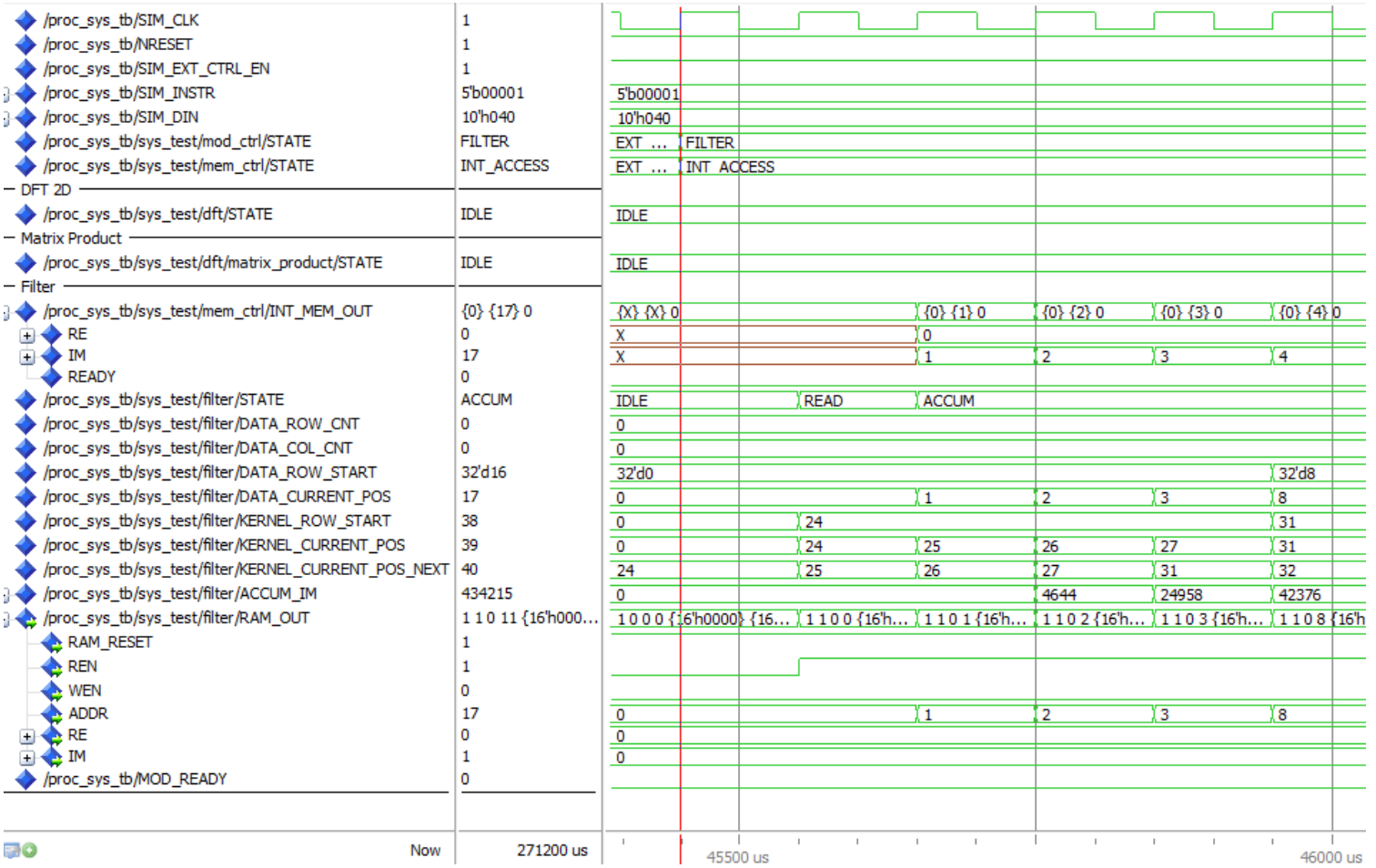

Abbildung B.39: Signalablauf-Diagramm der Simulation der Architektur `CONV_ROM` von `filter_conv`, 45340-46000 µs

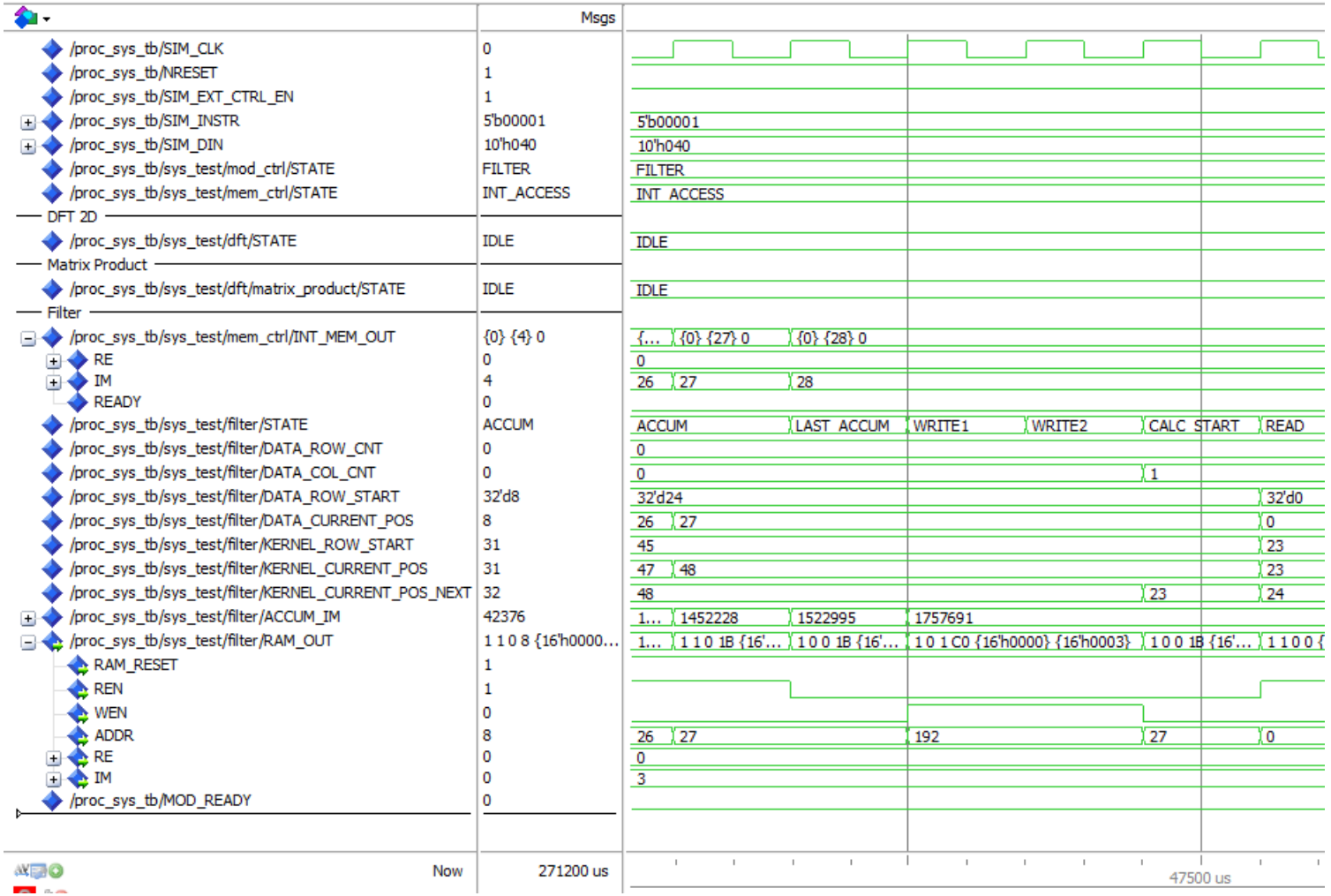

Abbildung B.40: Signalablauf-Diagramm der Simulation der Architektur `CONV_ROM` von `filter_conv`, 47150-47600 µs

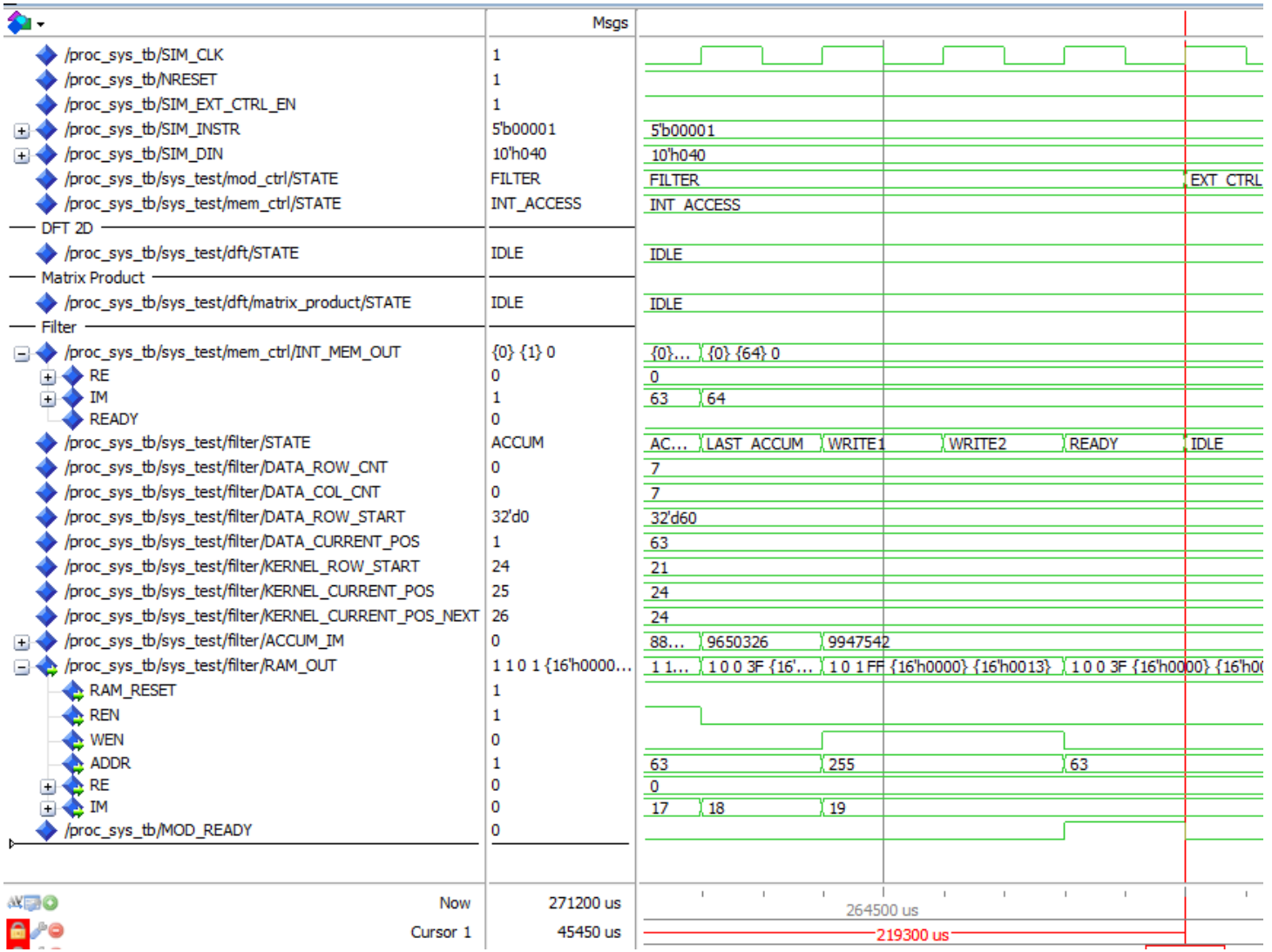

Abbildung B.41: Signalablauf-Diagramm der Simulation der Architektur `CONV_ROM` von `filter_conv`, 264300-264800 µs

B.2 Mikrocontroller

Die Mikrocontroller-Platine wurde in mehreren Schritten teilweise unter Einsatz eines Logikanalysators getestet.

Zunächst sollte es geprüft werden, ob die 5 verschobenen Drähte mit den beabsichtigten PMod Pins verbunden sind. Diesem Zweck dient der CodeComposer Projekt **test_gpio**. Die Pins wurden im Debugger schrittweise aktiviert und der Reaktion wurde auf dem an sie angebundenen Logikanalysator verfolgt. Der Effekt ist nur in Dynamik lässt sich nur sehr umständlich mit den stillen Bildern nachweisen. So wurde ein nicht eingeklemmter Pin gefunden und ausgebessert.

B.2.1 Test der GPIO Ausgänge

Dann wurden alle GPIO Ausgänge des Mikrocontrollers inklusive der Taktgenerierung im Projekt **MC_Output_test** getestet, ob sie korrekt konfiguriert sind (Funktion `initGPIO()`)und sich durch entsprechende Funktionen manipulieren lassen. Die Funktionalität wurde wieder mit dem Logikanalysator verifiziert. Die Testfälle sind in der Tabelle B.8 vorgestellt.

Tabelle B.8: Testfälle für GPIO-Ausgänge des Mikrocontrollers

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**
Taktsignal-Konfiguration (Funktion `startClock()`)	Der Taktsignal erscheint an dem Taktausgang	ja
Funktion `waitForPosedge()`	Funktion `waitForPosedge()` wartet auf definierte Anzahl der steigenden Flanke	ja
Test des GPIO-Datenausgangbusses bzw. Zedboard-Dateneingangbusses (Funktion `setZBInput()`)	Mithilfe der Funktion `setZBInput()` lässt sich ein Wert am Ausgangsdatenbus einstellen. Dann wird der eingestellte Wert bitweise negiert. Alle Bits des Busses sollen sich umschalten	ja

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.8

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Bestanden?**
Test des Befehlsbusses (Funktion `setInstr()`)	Mithilfe der Funktion `setInstr()` lässt sich ein Wert am Befehlsbus einstellen. Dann wird der eingestellte Wert bitweise negiert. Alle Bits des Busses sollen sich umschalten	ja
Resetausgang, Funktion `resetZB()`	Resetausgang lässt sich mithilfe der Funktion `resetZB()` manipulieren.	ja
Signal ext_ctrl_en, Funktion `extCtrlEn()`	Der Ausgang *ext_ctrl_en* lässt sich mithilfe der Funktion `extCtrlEn()` manipulieren.	ja

B.2.2 Test der Zedboard-Steuerung

Die restlichen Funktionen werden am sinnvollsten nicht einzeln, sondern im Rahmen der Steuerung des Zedboards durch den Mikrocontroller erprobt. Der Testablauf macht die Simulation in Unterkapitel ref zwecks der Vergleichbarkeit nach.

Tabelle B.9: Testfälle für Zedboard-Steuerung

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Kommentar**
Funktion `waitForReady()`	Nach dem Abschalten des Resets wird das erste Testmodul intern gesteuert aktiviert. Die Funktion `waitForReady()` wartet auf die Vollendung seiner Aufgabe.	Bestanden

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.9

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Kommentar**
Lesen des FPGA-RAMs mit `readRAM()`	Nach der Aktivierung der externen Steuerung wird Zedboard-RAM mit der Funktion `readRAM()` (Bereich [0;9]) in den lokalen Puffer gelesen. Die Funktion `printRAM()` zeigt den Inhalt dieses Bereichs. Es sollen die vom Testmodul produzierten Werte erscheinen, s. p. 11.	Fehler korrigiert: Die Funktion `readByte()` innerhalb `readRAM()` hat geklemmt, weil die Konfigurierung des Pinblocks N aus Versehen unterlassen wurde. Dies war der einzige Softwarefehler.
Schreiben in den RAM, externe Aktivierung des Moduls und Zurücklesen	Zuerst wird der lokale Puffer abwechselnd mit den von `0xff50` und `0x0060` aufsteigenden Werten befüllt. Dann werden diese Werte mit der Funktion `writeRAM()` in das FPGA-RAM übetragen. Der zweite Testmodul wird extern aktiviert und abgewartet. Anschließend werden die Daten aus dem Bereich [991;1000] zurückgelesen und angezeigt. Die Werte sollen in der umgekehrten Reihenfolge erscheinen.	Bestanden. Zusätzlich wurde der Inhalt des lokalen Puffers im Debugger kontrolliert.

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.9

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Kommentar**
Aktivierung der internen Steuerung, Abfrage des aktiven Moduls	Nach dem Reset wird die interne Steuerung startend mit dem ersten Testmodul aktiviert. Es wird auf die Vollendung der Arbeit von den beiden Modulen abgewartet, wobei das aktive Modul mit der Funktion `getModNum()` zwischendurch abgefragt wird. Anschließend werden die Daten aus dem Bereich [991;1000] zurückgelesen und angezeigt. Es sollen die vom ersten Modul produzierten Werte (p. 11) in der umgekehrten Reihenfolge auftreten	Bestanden

Die Ausgabe des Programms:

```
---Test1: internally generated values---
Cell 0: 0       3ff
Cell 1: 1       3fe
Cell 2: 2       3fd
Cell 3: 3       3fc
Cell 4: 4       3fb
Cell 5: 5       3fa
Cell 6: 6       3f9
Cell 7: 7       3f8
Cell 8: 8       3f7
Cell 9: 9       3f6

---Test2: externally written values after relocation---
```

```
Cell 991: ff59  69
Cell 992: ff58  68
Cell 993: ff57  67
Cell 994: ff56  66
Cell 995: ff55  65
Cell 996: ff54  64
Cell 997: ff53  63
Cell 998: ff52  62
Cell 999: ff51  61
Cell 1000: ff50 60

Active module: 0
Active module: 1
---Test3: After both internally controlled modules
finish their work---
Cell 991: 9     3f6
Cell 992: 8     3f7
Cell 993: 7     3f8
Cell 994: 6     3f9
Cell 995: 5     3fa
Cell 996: 4     3fb
Cell 997: 3     3fc
Cell 998: 2     3fd
Cell 999: 1     3fe
Cell 1000: 0    3ff
```

Im letzten Schritt sollten eigentlich die UART-Verbindung und die Befehlsbearbeitung geprüft werden. Dies lässt sich jedoch kaum sinnvoll durch den Mikrocontroller allein realisieren, so dass es im Anhang B.3.1 zusammen mit dem Gesamttestsystem getestet wurde.

B.3 Matlab

B.3.1 Gesamttest des Testsystems

Die Klasse `ProcSysProtocol` wurde zusammen mit der endgültigen Version der Mikrocontroller-Software (Projekt **Proc_Sys_Final**) im Rahmen des Gesamtsystemtests erprobt. Das Testskript in der Datei **test_proc_sys_protocol.m** folgt maßgeblich denselben Testfällen vom FPGA und vom Mikrocontroller in Tabelle B.3.

Tabelle B.10: Testfälle für Gesamttestsystem

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Kommentar**
Funktion `waitForReady()`	Das Zedboard in den Resetzustand mit `resetZB()` versetzt. Dann schaltet die Funktion `activateInternalControl()` die interne Steuerung startend mit dem ersten Testmodul und die Funktion `waitForCompletion()` wartet auf die Vollendung seiner Aufgabe. Nun wird Zedboard-RAM mit der Funktion `readZB()` (Bereich [0;9]) gelesen und dargestellt. Es sollen die vom Testmodul produzierten Werte erscheinen, s. p. 11.	Bestanden. Der Empfang und die Bearbeitung der Bedfehlen durch Mikrocontroller wurde zuerst im Debugger kontrolliert. Alle Funktionen sind Mitglieder der Klasse `ProcSysProtocol`

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.10

Testfall	**Testvorgang und zu erwartendes Verhalten**	**Kommentar**
Schreiben in den RAM, externe Aktivierung des Moduls und Zurücklesen	Zuerst werden die abwechselnden und von `0xff50` und `0x0060` aufsteigenden Werte mit der Funktion `writeZB()` an das FPGA übermittelt. Der zweite Testmodul wird mit der Funktion `activateModule()` extern aktiviert und abgewartet. Anschließend werden die Daten aus dem Bereich [991;1000] zurückgelesen und angezeigt. Die Werte sollen in der umgekehrten Reihenfolge erscheinen.	Bestanden.
Aktivierung der internen Steuerung	Nach dem Reset wird die interne Steuerung startend mit dem ersten Testmodul aktiviert. Es wird auf die Vollendung der Arbeit von den beiden Modulen abgewartet. Anschließend werden die Daten aus dem Bereich [991;1000] zurückgelesen und angezeigt. Es sollen die vom ersten Modul produzierten Werte (p. 11) in der umgekehrten Reihenfolge auftreten	Bestanden

Fortsetzung auf der nächsten Seite…

Fortsetzung der Tabelle B.10

Testfall	Testvorgang und zu erwartendes Verhalten	Kommentar
Schreiben und Lesen einer komplexen Matrix	Es wird eine ganze mit komplexen Zufallswerten befüllte quadratische Matrix mittels Funktion `writeMatrix()` in das FPGA-RAM geschrieben. Dann wird sie mit der Funktion `readMatrix` zurückgelesen. Die Werte sollen übereinstimmen.	Bestanden. Die konkreten Zahlen völlig irrelevant. Von Bedeutung ist nur, dass die Daten miteinander übereinstimmen.

Hier ist die Ausgabe des Skriptes:

```
Microcontroller found on port /dev/ttyS100
---Test1: internally generated values---
0       3ff
1       3fe
2       3fd
3       3fc
4       3fb
5       3fa
6       3f9
7       3f8
8       3f7
9       3f6

---Test2: Write values externally and read them back
(in reverse order) after relocation---
ff50    60
ff51    61
ff52    62
ff53    63
ff54    64
ff55    65
ff56    66
ff57    67
```

```
ff58    68
ff59    69

Active module id is 1

ff59    69
ff58    68
ff57    67
ff56    66
ff55    65
ff54    64
ff53    63
ff52    62
ff51    61
ff50    60

---Test3: After both internally controlled modules
finish their work---
9       3f6
8       3f7
7       3f8
6       3f9
5       3fa
4       3fb
3       3fc
2       3fd
1       3fe
0       3ff

Test4: write and read back complex matrix

data_out =

8×8 int16 matrix

Columns 1 through 4

1232 -   1122i   -1237 +    883i    -380 -
 705i    1632 -   1595i
```

```
-1465 +   150i   -1249 +   318i    1411 +  1240i    -290 -   380i
-89 +   1073i    -710 -    274i     472 +   2045i  -679 +   1573i
-997 -    625i    1557 +   1573i    -506 +  1970i     395 +    197i
-537 -    159i    -119 -    439i    1544 -  1528i    1646 -    537i
662 +    570i    -394 -   1315i    1166 -   1097i  827 -   1195i
-1354 +   1709i   -1314 +   546i    -144 -  1952i    -502 -    242i
-907 -   1387i    1920 +   507i    1286 +  440i     962 +   1868i

Columns 5 through 8

1860 -   1540i   -2010 +   758i   -2013 -  1411i   -1675 +   1045i
175 -    120i    1699 +   1677i    930 +   1306i  385 +    992i
164 +   1461i    584 +    454i    -598 +  511i   -1061 +   1356i
-774 -   1871i   -2043 +   1638i   1148 +  977i    1398 -   1407i
-1757 +    784i   -1924 -   1256i    -260 +  1249i    1463 -    175i
-1303 +   1961i   -1195 +   1042i    -260 -  1773i    1898 +    483i
-1668 -    888i    -185 -    630i   -1847 +  1846i     -46 +   1770i
-150 -   1501i   -1527 -    334i   -1845 -  10i   -1146 +   1372i


data_in =

8×8 int16 matrix

```

```
Columns 1 through 4

1232 -  1122i  -1237 +   883i   -380 -  705i  1632 -  1595i
-1465 +   150i  -1249 +   318i   1411 +  1240i   -290 -   380i
-89 +  1073i   -710 -   274i   472 +  2045i  -679 +  1573i
-997 -   625i   1557 +  1573i   -506 +  1970i   395 +   197i
-537 -   159i   -119 -   439i   1544 -  1528i  1646 -   537i
662 +   570i   -394 -  1315i  1166 -  1097i  827 -  1195i
-1354 +  1709i  -1314 +   546i   -144 -  1952i   -502 -   242i
-907 -  1387i   1920 +   507i   1286 +  440i   962 +  1868i

Columns 5 through 8

1860 -  1540i  -2010 +   758i  -2013 -  1411i  -1675 +  1045i
175 -   120i   1699 +  1677i   930 +  1306i  385 +   992i
164 +  1461i   584 +   454i   -598 +  511i  -1061 +  1356i
-774 -  1871i  -2043 +  1638i   1148 +  977i  1398 -  1407i
-1757 +   784i  -1924 -  1256i   -260 +  1249i  1463 -   175i
-1303 +  1961i  -1195 +  1042i   -260 -  1773i  1898 +   483i
-1668 -   888i   -185 -   630i  -1847 +  1846i   -46 +  1770i
-150 -  1501i  -1527 -   334i  -1845 -  10i  -1146 +  1372i

Input and output are equal: true
```

```
109
110 >>
```

B.3.2 Rechenversuche zur 2D-DFT

Tabelle B.11: Höchste zahlenmäßige Abweichungen bei unskalierten 2D-DFT bei jeweiligen Breiten der Bruchteile der Twiddle-Faktoren (Reihen) und der Ausgangsdaten (Spalten)

		9	11	13	15	17	19	21	23	25
2D-Sinus	8	0,005367	0,002339	0,002705	0,002431	0,002454	0,002455	0,002455	0,002455	0,002455
	10	0,005367	0,002339	0,002705	0,002431	0,002454	0,002455	0,002455	0,002455	0,002455
	12	0,005367	0,002339	0,002705	0,002431	0,002454	0,002455	0,002455	0,002455	0,002455
	14	0,003906	0,001046	0,000630	0,000386	0,000478	0,000470	0,000471	0,000471	0,000471
	16	0,003906	0,001046	0,000267	0,000055	0,000026	0,000026	0,000026	0,000025	0,000025
	18	0,003906	0,001046	0,000267	0,000055	0,000026	0,000026	0,000026	0,000025	0,000025
	20	0,003906	0,001046	0,000244	0,000069	0,000019	0,000004	0,000006	0,000006	0,000006
	22	0,003906	0,001046	0,000244	0,000055	0,000019	0,000004	0,000001	0,000001	0,000002
	24	0,003906	0,001046	0,000244	0,000055	0,000019	0,000002	0,000001	0,000000	0,000000
Viertel vom 2D-Sinus	8	0,003906	0,000977	0,000303	0,000273	0,000242	0,000227	0,000227	0,000227	0,000227
	10	0,003906	0,000977	0,000303	0,000273	0,000242	0,000227	0,000227	0,000227	0,000227
	12	0,003906	0,000977	0,000303	0,000273	0,000242	0,000227	0,000227	0,000227	0,000227
	14	0,003906	0,000958	0,000210	0,000065	0,000045	0,000044	0,000044	0,000044	0,000044
	16	0,003906	0,000958	0,000210	0,000058	0,000015	0,000004	0,000002	0,000002	0,000002
	18	0,003906	0,000958	0,000210	0,000058	0,000015	0,000004	0,000002	0,000002	0,000002
	20	0,003906	0,000958	0,000210	0,000065	0,000015	0,000004	0,000001	0,000001	0,000001
	22	0,003906	0,000958	0,000210	0,000065	0,000015	0,000004	0,000001	0,000000	0,000000
	24	0,003906	0,000958	0,000210	0,000065	0,000015	0,000004	0,000001	0,000000	0,000000
Alt. 1/-1	8	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	10	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	12	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	14	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	16	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	18	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	20	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	22	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
	24	0,001953	0,000488	0,000122	0,000031	0,000008	0,000002	0,000000	0,000000	0,000000
Zufallsmatrix	8	0,007807	0,004350	0,004091	0,004560	0,004353	0,005200	0,004355	0,003658	0,004651
	10	0,007807	0,004350	0,004091	0,004560	0,004353	0,005200	0,004355	0,003658	0,004651
	12	0,007807	0,004350	0,004091	0,004560	0,004353	0,005200	0,004355	0,003658	0,004651
	14	0,005531	0,001773	0,000795	0,000898	0,000836	0,000998	0,000836	0,000702	0,000893
	16	0,005538	0,001267	0,000360	0,000099	0,000052	0,000051	0,000045	0,000037	0,000047
	18	0,005538	0,001267	0,000360	0,000099	0,000052	0,000051	0,000045	0,000037	0,000047
	20	0,005538	0,001242	0,000388	0,000091	0,000025	0,000011	0,000011	0,000009	0,000012
	22	0,005538	0,001242	0,000388	0,000091	0,000022	0,000006	0,000004	0,000002	0,000003
	24	0,005538	0,001242	0,000388	0,000091	0,000023	0,000005	0,000002	0,000001	0,000001

Tabelle B.12: Höchste Abweichungen des Argumentes in Grad bei unskalierten 2D-DFT bei jeweiligen Breiten der Bruchteile der Twiddle-Faktoren (Reihen) und der Ausgangsdaten (Spalten)

		9	11	13	15	17	19	21	23	25
2D-Sinus	8	0,416903	0,320973	0,158533	0,125141	0,138768	0,117667	0,117667	0,117667	0,117667
	10	0,416903	0,320973	0,158533	0,125141	0,138768	0,117667	0,117667	0,117667	0,117667
	12	0,416903	0,320973	0,158533	0,125141	0,138768	0,117667	0,117667	0,117667	0,117667
	14	0,373906	0,119865	0,049442	0,023141	0,028420	0,022368	0,022698	0,022912	0,022582
	16	0,373906	0,119865	0,036788	0,008205	0,001379	0,001262	0,001455	0,001159	0,001188
	18	0,373906	0,119865	0,036788	0,008205	0,001379	0,001262	0,001455	0,001159	0,001188
	20	0,373906	0,119865	0,036788	0,008205	0,002424	0,000715	0,000332	0,000318	0,000301
	22	0,373906	0,119865	0,036788	0,008205	0,002424	0,000715	0,000195	0,000081	0,000089
	24	0,373906	0,119865	0,036788	0,008205	0,002424	0,000715	0,000095	0,000036	0,000023
Viertel vom 2D-Sinus	8	0,608403	0,165305	0,048509	0,027067	0,014467	0,015881	0,016286	0,016184	0,016195
	10	0,608403	0,165305	0,048509	0,027067	0,014467	0,015881	0,016286	0,016184	0,016195
	12	0,608403	0,165305	0,048509	0,027067	0,014467	0,015881	0,016286	0,016184	0,016195
	14	0,608403	0,165305	0,065167	0,015678	0,005283	0,002987	0,003005	0,003094	0,003113
	16	0,608403	0,108069	0,065167	0,012547	0,002722	0,000658	0,000171	0,000226	0,000165
	18	0,608403	0,108069	0,065167	0,012547	0,002722	0,000658	0,000171	0,000226	0,000165
	20	0,608403	0,108069	0,065167	0,019831	0,002722	0,000510	0,000122	0,000090	0,000033
	22	0,608403	0,108069	0,065167	0,019831	0,002722	0,000510	0,000208	0,000036	0,000020
	24	0,608403	0,108069	0,065167	0,019831	0,002722	0,000510	0,000208	0,000051	0,000014
Alternierende 1/-1	8	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	10	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	12	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	14	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	16	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	18	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	20	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	22	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	24	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
Zufallsmatrix	8	0,189276	0,073014	0,181085	359,859902	0,102964	0,110774	0,199188	0,143700	0,125757
	10	0,189276	0,073014	0,181085	359,859902	0,102964	0,110774	0,199188	0,143700	0,125757
	12	0,189276	0,073014	0,181085	359,859902	0,102964	0,110774	0,199188	0,143700	0,125757
	14	0,404232	0,074088	0,024821	0,030101	0,019675	0,021536	0,038222	0,027593	0,024151
	16	0,404232	0,040206	0,016422	0,004643	0,001468	0,001213	0,001888	0,001446	0,001271
	18	0,404232	0,040206	0,016422	0,004643	0,001468	0,001213	0,001888	0,001446	0,001271
	20	0,404232	0,040206	0,022618	0,002855	0,000778	0,000292	0,000600	0,000349	0,000320
	22	0,404232	0,040206	0,022618	0,002871	0,000917	0,000273	0,000198	0,000075	0,000084
	24	0,404232	0,040206	0,022618	0,002855	0,000917	0,000164	0,000171	0,000032	0,000019

Tabelle B.13: Höchste zahlenmäßige Abweichungen bei 2D-DFT mit herunterskalierten 1D-DFT bei jeweiligen Breiten der Bruchteile der Twiddle-Faktoren (Reihen) und der Ausgangsdaten (Spalten)

		8	10	12	14	16	18	20
2D-Sinus	8	0,008789	0,002930	0,000977	0,000292	0,000338	0,000304	0,000307
	10	0,008789	0,002930	0,000977	0,000292	0,000338	0,000304	0,000307
	12	0,008789	0,002930	0,000977	0,000292	0,000338	0,000304	0,000307
	14	0,008789	0,002930	0,000977	0,000131	0,000079	0,000048	0,000060
	16	0,008789	0,002930	0,000977	0,000131	0,000033	0,000007	0,000003
	18	0,008789	0,002930	0,000977	0,000131	0,000033	0,000007	0,000003
	20	0,008789	0,002930	0,000977	0,000131	0,000031	0,000009	0,000002
Viertel vom 2D-Sinus	8	0,009375	0,002635	0,000488	0,000122	0,000038	0,000034	0,000030
	10	0,009375	0,002635	0,000488	0,000122	0,000038	0,000034	0,000030
	12	0,009375	0,002635	0,000488	0,000122	0,000038	0,000034	0,000030
	14	0,009375	0,002635	0,000488	0,000120	0,000026	0,000008	0,000006
	16	0,009375	0,002635	0,000488	0,000120	0,000026	0,000007	0,000002
	18	0,009375	0,002635	0,000488	0,000120	0,000026	0,000007	0,000002
	20	0,009375	0,002635	0,000488	0,000120	0,000026	0,000008	0,000002
Alt. 1/-1	8	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
	10	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
	12	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
	14	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
	16	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
	18	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
	20	0,003906	0,000977	0,000244	0,000061	0,000015	0,000004	0,000001
Zufallsmatrix	8	0,010880	0,002930	0,000977	0,000591	0,000540	0,000520	0,000535
	10	0,010880	0,003023	0,000977	0,000591	0,000540	0,000520	0,000535
	12	0,010880	0,003023	0,000977	0,000591	0,000540	0,000520	0,000535
	14	0,010880	0,002930	0,000977	0,000182	0,000113	0,000098	0,000103
	16	0,010880	0,003220	0,000977	0,000182	0,000049	0,000013	0,000006
	18	0,010880	0,003220	0,000977	0,000182	0,000049	0,000013	0,000006
	20	0,010880	0,003220	0,000977	0,000182	0,000049	0,000011	0,000003

Tabelle B.14: Höchste Abweichungen des Argumentes in Grad bei 2D-DFT mit herunterskalierten 1D-DFT bei jeweiligen Breiten der Bruchteile der Twiddle-Faktoren (Reihen) und der Ausgangsdaten (Spalten)

		8	10	12	14	16	18	20
2D-Sinus	8	8,567765	1,645542	1,468801	0,320973	0,158533	0,125141	0,138768
	10	8,567765	1,645542	1,468801	0,320973	0,158533	0,125141	0,138768
	12	8,567765	1,645542	1,468801	0,320973	0,158533	0,125141	0,138768
	14	8,567765	1,645542	1,468801	0,119865	0,049442	0,023141	0,028420
	16	8,567765	1,645542	1,468801	0,119865	0,036788	0,008205	0,001379
	18	8,567765	1,645542	1,468801	0,119865	0,036788	0,008205	0,001379
	20	8,567765	1,645542	1,468801	0,119865	0,036788	0,008205	0,002424
Viertel vom 2D-Sinus	8	14,844590	12,994617	1,684684	0,165305	0,048509	0,027067	0,014467
	10	14,844590	12,994617	1,684684	0,165305	0,048509	0,027067	0,014467
	12	14,844590	12,994617	1,684684	0,165305	0,048509	0,027067	0,014467
	14	14,844590	12,994617	1,684684	0,165305	0,065167	0,015678	0,005283
	16	14,844590	12,994617	1,684684	0,108069	0,065167	0,012547	0,002722
	18	14,844590	12,994617	1,684684	0,108069	0,065167	0,012547	0,002722
	20	14,844590	12,994617	1,684684	0,108069	0,065167	0,019831	0,002722
Alt. 1/-1	8	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	10	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	12	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	14	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	16	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	18	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	20	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
Zufallsmatrix	8	359,865958	1,477984	0,416450	0,084042	0,137072	359,997104	0,098451
	10	359,865958	1,477984	0,416450	0,084042	0,137072	359,997104	0,098451
	12	359,865958	1,477984	0,416450	0,084042	0,137072	359,997104	0,098451
	14	359,865958	1,539270	0,416450	0,062258	0,043790	0,018591	0,018711
	16	359,865958	1,477984	0,416450	0,037173	0,027905	0,004395	0,001023
	18	359,865958	1,477984	0,416450	0,037173	0,027905	0,004395	0,001023
	20	359,865958	1,477984	0,416450	0,062258	0,027409	0,005257	0,000698

Tabelle B.15: Höchste zahlenmäßige Abweichungen bei 2D-DFT mit herunterskalierter Twiddlefaktor-Matrix bei jeweiligen Breiten der Bruchteile der Twiddle-Faktoren (Reihen) und der Ausgangsdaten (Spalten)

		8	10	12	14	16	18	20
2D-Sinus	8	0,002869	0,000535	0,000151	0,000054	0,000044	0,000037	0,000038
	10	0,002869	0,000535	0,000151	0,000054	0,000044	0,000037	0,000038
	12	0,002869	0,000535	0,000151	0,000054	0,000044	0,000037	0,000038
	14	0,002869	0,000535	0,000151	0,000035	0,000015	0,000006	0,000007
	16	0,002869	0,000535	0,000151	0,000035	0,000010	0,000003	0,000001
	18	0,002869	0,000535	0,000151	0,000035	0,000010	0,000003	0,000001
	20	0,002869	0,000535	0,000151	0,000035	0,000010	0,000003	0,000001
Viertel vom 2D-Sinus	8	0,002869	0,000702	0,000168	0,000039	0,000011	0,000005	0,000004
	10	0,002869	0,000702	0,000168	0,000039	0,000011	0,000005	0,000004
	12	0,002869	0,000702	0,000168	0,000039	0,000011	0,000005	0,000004
	14	0,002869	0,000702	0,000168	0,000039	0,000009	0,000002	0,000001
	16	0,002869	0,000702	0,000168	0,000036	0,000009	0,000002	0,000001
	18	0,002869	0,000702	0,000168	0,000036	0,000009	0,000002	0,000001
	20	0,002869	0,000702	0,000168	0,000036	0,000009	0,000002	0,000001
Alt. 1/-1	8	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	10	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	12	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	14	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	16	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	18	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	20	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
Zufallsmatrix	8	0,003013	0,000784	0,000229	0,000079	0,000067	0,000083	0,000068
	10	0,003013	0,000784	0,000229	0,000079	0,000067	0,000083	0,000068
	12	0,003013	0,000784	0,000229	0,000079	0,000067	0,000083	0,000068
	14	0,003013	0,000784	0,000229	0,000051	0,000020	0,000018	0,000014
	16	0,003013	0,000784	0,000229	0,000045	0,000012	0,000003	0,000001
	18	0,003013	0,000784	0,000229	0,000045	0,000012	0,000003	0,000001
	20	0,003013	0,000784	0,000229	0,000045	0,000012	0,000003	0,000001

Tabelle B.16: Höchste Abweichungen des Argumentes in Grad bei 2D-DFT mit herunterskalierter Twiddlefaktor-Matrix bei jeweiligen Breiten der Bruchteile der Twiddle-Faktoren (Reihen) und der Ausgangsdaten (Spalten)

		8	10	12	14	16	18	20
2D-Sinus	8	116,106963	5,305047	4,065051	1,249494	0,315987	0,098782	0,146243
	10	116,106963	5,305047	4,065051	1,249494	0,315987	0,098782	0,146243
	12	116,106963	5,305047	4,065051	1,249494	0,315987	0,098782	0,146243
	14	116,106963	5,305047	4,065051	1,249494	0,119865	0,049442	0,044256
	16	116,106963	5,305047	4,065051	1,249494	0,119865	0,070112	0,004778
	18	116,106963	5,305047	4,065051	1,249494	0,119865	0,070112	0,004778
	20	116,106963	5,305047	4,065051	1,249494	0,119865	0,070112	0,007866
Viertel vom 2D-Sinus	8	349,427819	57,994617	12,994617	5,440332	0,193486	0,040005	0,016570
	10	349,427819	57,994617	12,994617	5,440332	0,193486	0,040005	0,016570
	12	349,427819	57,994617	12,994617	5,440332	0,193486	0,040005	0,016570
	14	349,427819	57,994617	12,994617	5,440332	0,162522	0,040005	0,012291
	16	349,427819	57,994617	12,994617	5,440332	0,162522	0,040005	0,007476
	18	349,427819	57,994617	12,994617	5,440332	0,162522	0,040005	0,007476
	20	349,427819	57,994617	12,994617	5,440332	0,162522	0,040005	0,007476
Alt. 1/-1	8	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	10	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	12	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	14	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	16	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	18	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
	20	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000	0,000000
Zufallsmatrix	8	359,965964	359,977690	359,673133	359,957062	0,161447	0,111950	0,194550
	10	359,965964	359,977690	359,673133	359,957062	0,161447	0,111950	0,194550
	12	359,965964	359,977690	359,673133	359,957062	0,161447	0,111950	0,194550
	14	359,965964	359,977690	359,673133	359,957062	0,131195	0,039734	0,039967
	16	359,965964	359,977690	359,673133	359,957062	0,047724	0,010808	0,009591
	18	359,965964	359,977690	359,673133	359,957062	0,047724	0,010808	0,009591
	20	359,965964	359,977690	359,673133	359,957062	0,047724	0,009985	0,007856

B.3.3 Verifikation von 2D-DFT

B.3.3.1 Testergebnisse mit künstlich erzeugten Daten

Tabelle B.17: Testergebnisse auf FPGA mit künstlich erzeugten Testdaten.

Differenz zu:		FixedDFT		Matlab FFT	
Test		Zahlendif.	Winkeldif.	Zahlendif.	Winkeldif.
2D-Sinus	1D-DFT	0,001953	0,009683	0,001090	0,495137
	2D DFT	0,001953	0,054552	0,005367	0,416903
Viertelsinus	1D-DFT	0,000000	0,000000	0,000934	0,172578
	2D-DFT	0,000000	0,000000	0,003906	0,608403
Alt. 1/-1	1D-DFT	0,000000	0,000000	0,000000	0,000000
	2D-DFT	0,000000	0,000000	0,001953	0,000000
Zufallsmatrix	1D-DFT	0,001953	0,049081	0,001734	0,570458
	2D-DFT	0,001953	0,060544	0,007598	0,300435

B.3.3.2 Testergebnisse von unskalierter 8×8 16-Bit 2D-DFT

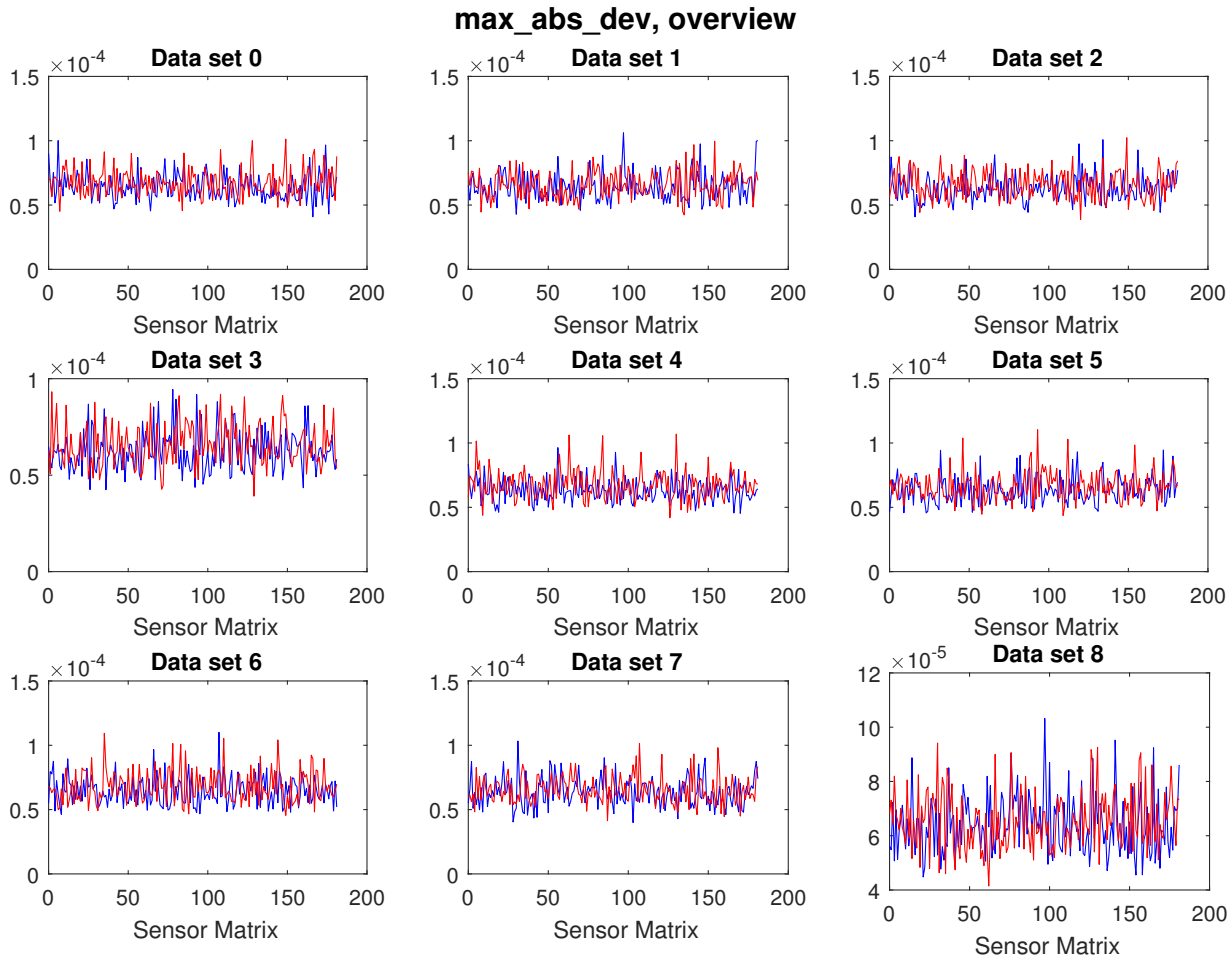

Abbildung B.42: Höchste zahlenmäßige Abweichungen bei der unskalierten 8×8 16-Bit 2D-DFT, alle Datensätze dargestellt. Auf die postive Aussteuerung (64) normiert.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

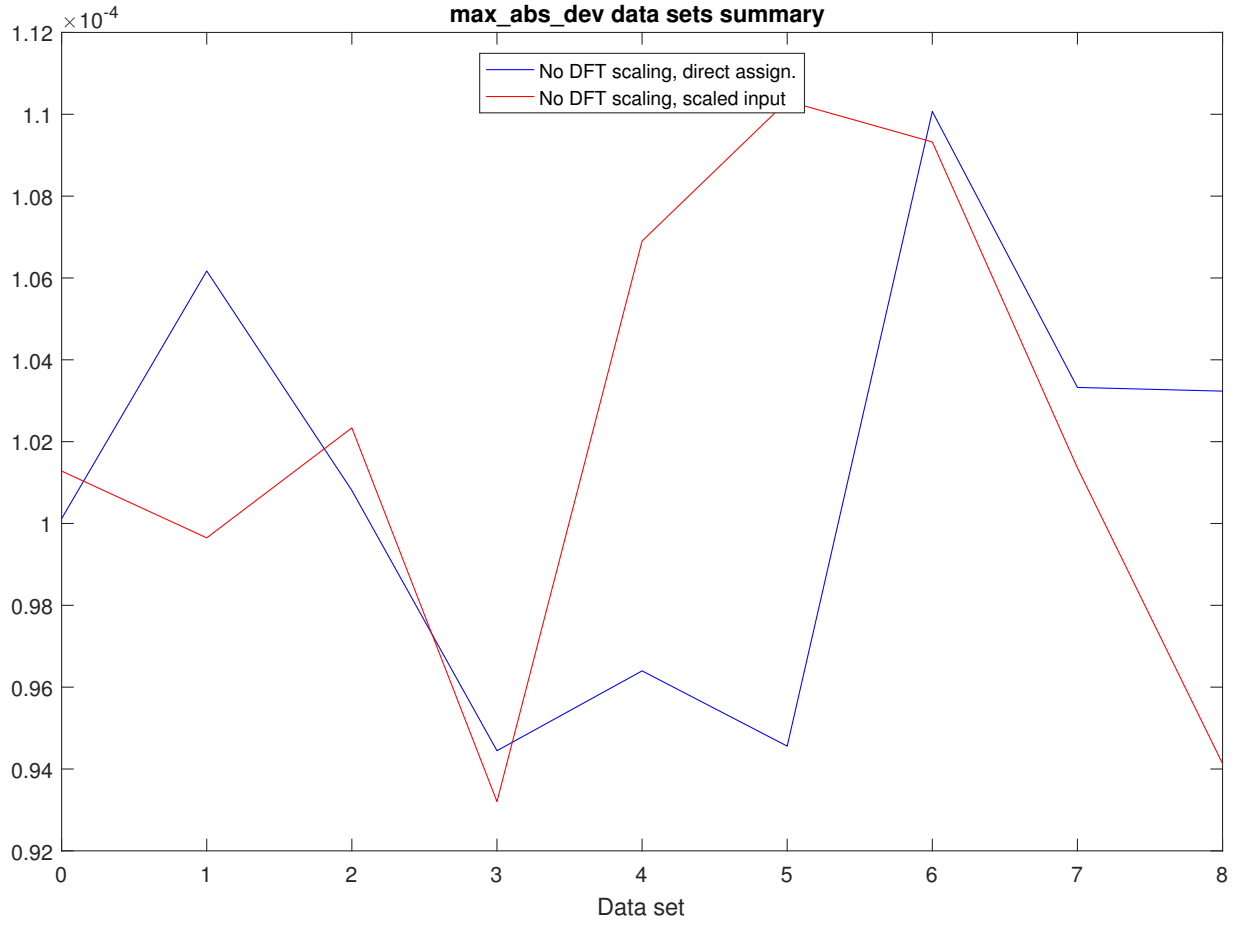

Abbildung B.43: Höchste zahlenmäßige Abweichungen bei der unskalierten 8×8 16-Bit 2D-DFT, Übersicht über Datenreihen Auf die postive Aussteuerung (64) normiert.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

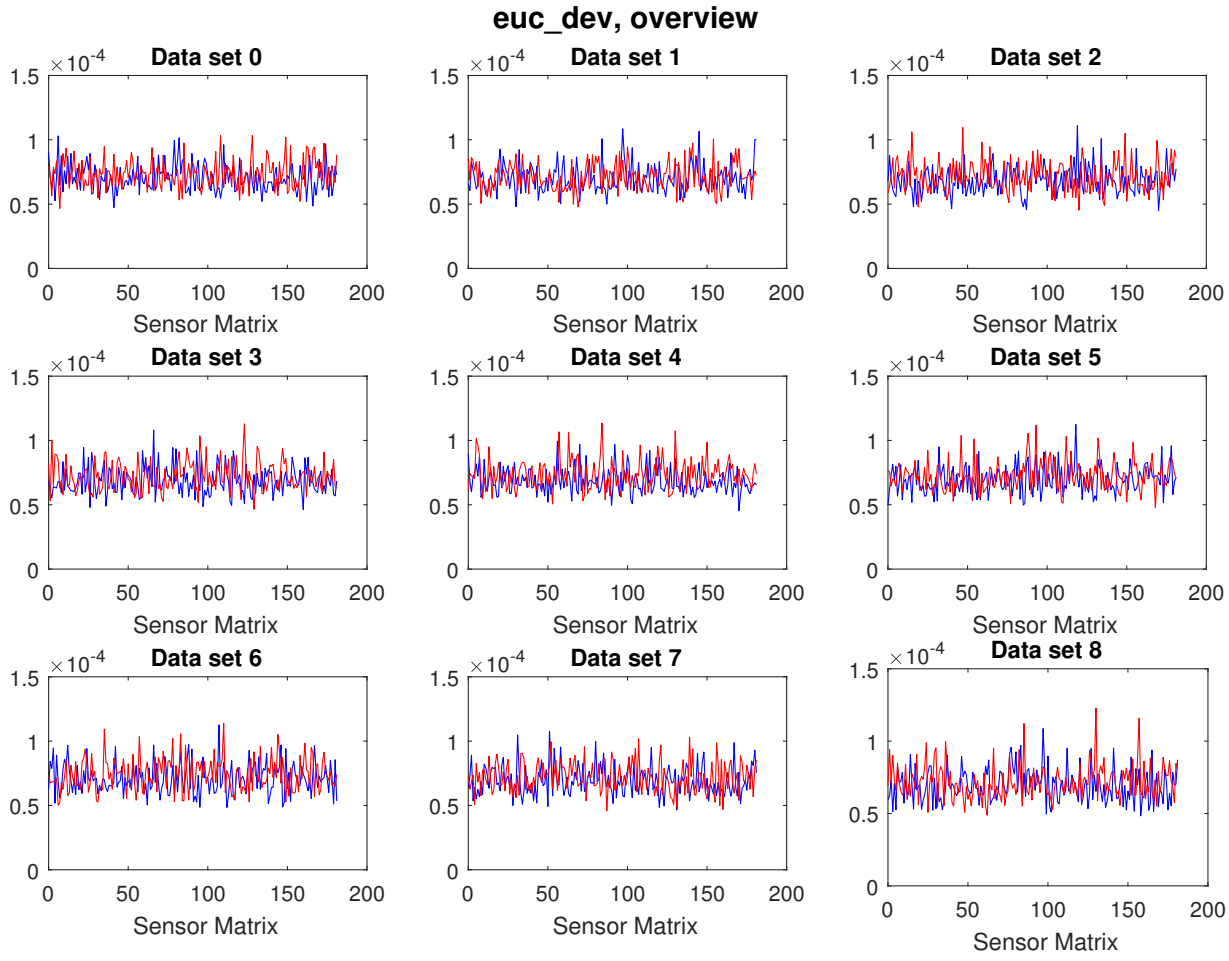

Abbildung B.44: Höchste euklidische Abweichungen bei der unskalierten 8×8 16-Bit 2D-DFT, alle Datensätze dargestellt. Auf die postive Aussteuerung (64) normiert.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

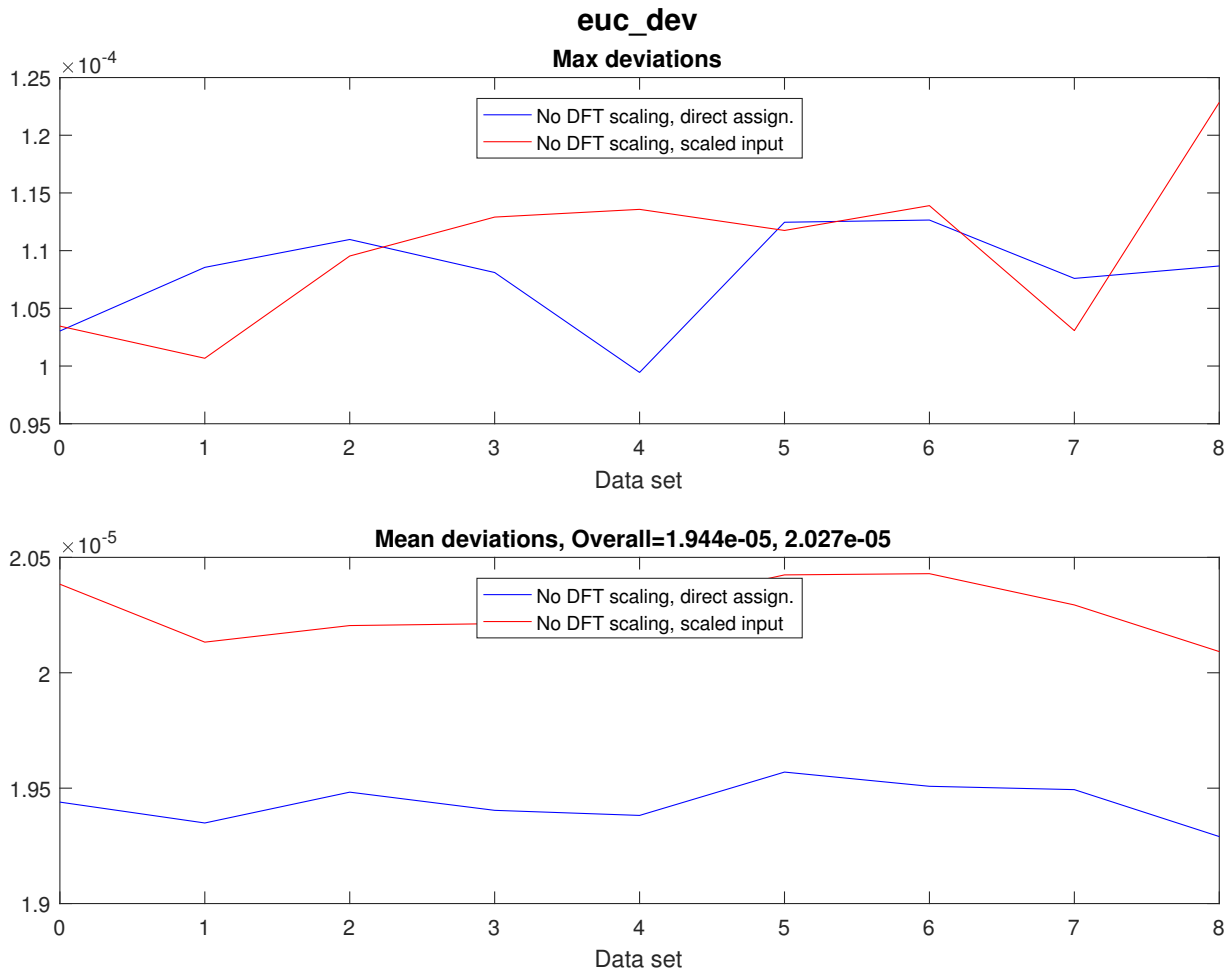

Abbildung B.45: Höchste und durchschnittliche euklidische Abweichung bei der unskalierten 8×8 16-Bit 2D-DFT, Übersicht über Datenreihen Auf die postive Aussteuerung (64) normiert.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

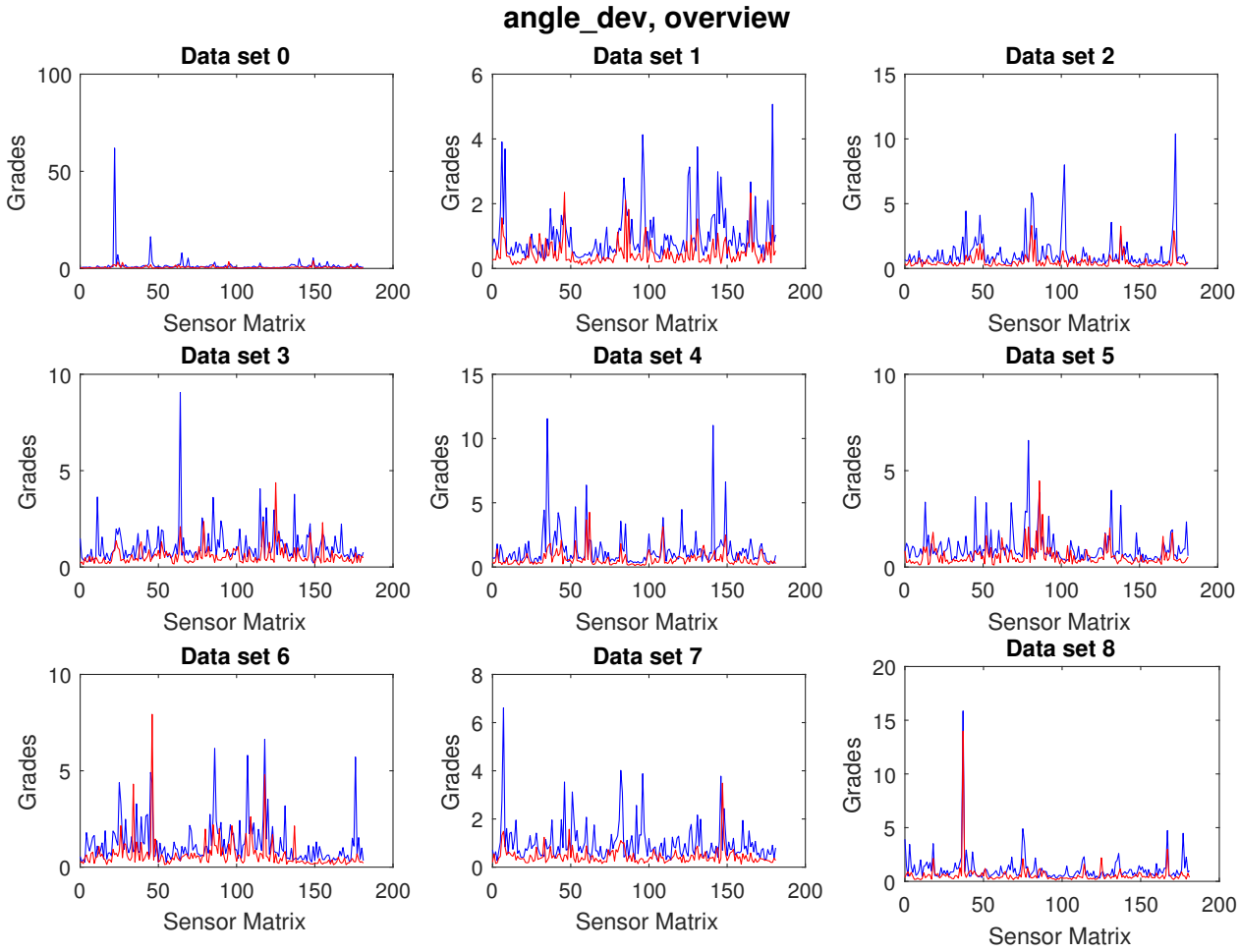

Abbildung B.46: Höchste Abweichungen des Argumentes bei der unskalierten 8×8 16-Bit 2D-DFT, nach der Korrektur mit `unwrap`.. Alle Datensätze sind dargestellt.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

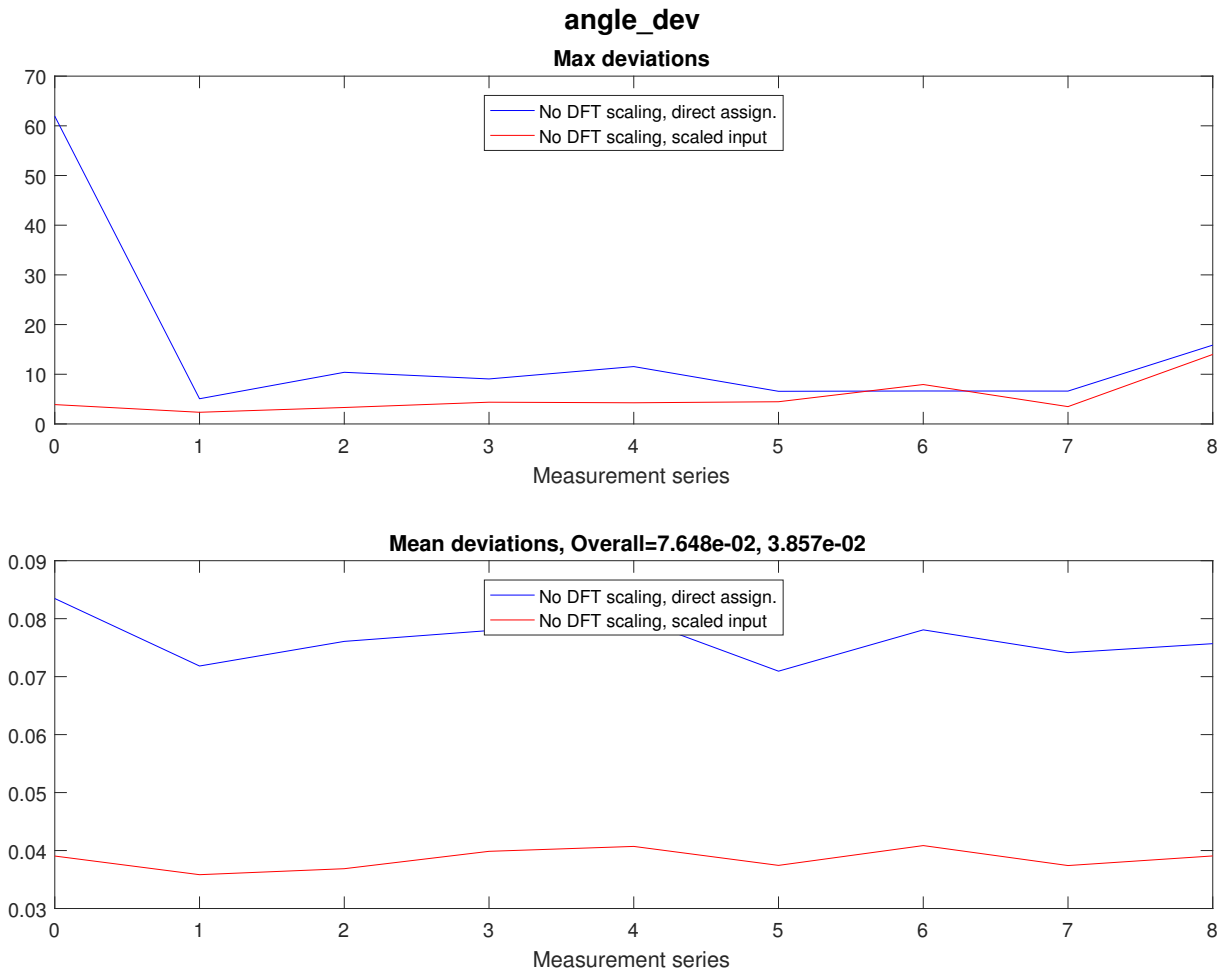

Abbildung B.47: Höchste und durchschnittliche Abweichungen des Argumentes bei der unskalierten 8×8 16-Bit 2D-DFT, nach der Korrektur mit `unwrap`.. Übersicht über Datenreihen
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

	Measurem	Pos in ...	Matlab FFT v...	Matlab ...	FPGA value	FPGA angle	Angle dif
1	1, 22	8, 4	-0.06621+0.07676i	130.7809	-0.06641+0.07227i	132.5805	1.7996
2	1, 23	8, 4	-0.01726-0.00673i	-158.6985	-0.01367-0.00586i	-156.8014	1.8971
3	1, 24	5, 8	-0.00880-0.06463i	-97.7490	-0.00781-0.06641i	-96.7098	1.0392
4	1, 25	5, 8	-0.01876-0.00628i	-161.4892	-0.01953-0.00781i	-158.1986	3.2907
5	1, 26	5, 8	0.03197-0.02655i	-39.7101	0.03125-0.02734i	-41.1859	-1.4758
6	1, 28	5, 8	0.03879+0.00404i	5.9489	0.03906+0.00195i	2.8624	-3.0865
7	1, 43	8, 2	-0.04637-0.09139i	-116.9027	-0.04883-0.08984i	-118.5231	-1.6205
8	1, 44	2, 3	0.02590+0.14443i	79.8344	0.02344+0.14648i	80.9097	1.0753
9	1, 46	8, 2	0.03253-0.02267i	-34.8759	0.03320-0.02148i	-32.9052	1.9706
10	1, 62	2, 2	-0.09419-0.02607i	-164.5278	-0.09570-0.02344i	-166.2392	-1.7114
11	1, 64	2, 8	-0.01912-0.05935i	-107.8568	-0.01953-0.05469i	-109.6538	-1.7971
12	1, 70	2, 5	-0.03617+0.03969i	132.3419	-0.03516+0.03711i	133.4518	1.1099
13	1, 86	2, 8	-0.07646-0.01408i	-169.5661	-0.07813-0.01172i	-171.4692	-1.9031
14	1, 94	8, 7	0.04082+0.21412i	79.2057	0.04492+0.21484i	78.1901	-1.0156
15	1, 96	8, 7	0.02464-0.03597i	-55.5883	0.02734-0.03516i	-52.1250	3.4633
16	1, 116	2, 5	-0.03436-0.16511i	-101.7570	-0.03711-0.16211i	-102.8937	-1.1367
17	1, 140	2, 7	0.16095-0.09434i	-30.3755	0.16406-0.09180i	-29.2281	1.1474
18	1, 148	8, 5	-0.10326-0.02480i	-166.4936	-0.10352-0.02734i	-165.2032	1.2904
19	1, 150	2, 5	0.01679+0.01710i	45.5286	0.01758+0.01563i	41.6335	-3.8951
20	1, 158	8, 7	-0.19185-0.08214i	-156.8215	-0.19336-0.07813i	-157.9993	-1.1778
21	1, 174	2, 1	-0.01709-0.06248i	-105.3009	-0.01953-0.06250i	-107.3540	-2.0531
22	1, 178	8, 2	0.12417-0.03874i	-17.3256	0.12305-0.03516i	-15.9454	1.3802
23	2, 7	8, 2	0.07854-0.05185i	-33.4299	0.07813-0.05469i	-34.9920	-1.5622
24	2, 8	8, 2	-0.05930-0.08189i	-125.9081	-0.05859-0.08398i	-124.9025	1.0056
25	2, 31	5, 8	0.05903-0.04351i	-36.3965	0.05859-0.04492i	-37.4762	-1.0797
26	2, 47	8, 1	0.09503+0.00770i	4.6325	0.09570+0.01172i	6.9811	2.3486
27	2, 81	2, 1	0.00976+0.09937i	84.3923	0.00781+0.09961i	85.5154	1.1230
28	2, 86	2, 8	-0.10044+0.06824i	145.8055	-0.09570+0.07031i	143.6955	-2.1100
29	2, 88	2, 8	0.05175+0.08654i	59.1200	0.04883+0.08789i	60.9454	1.8254
30	2, 99	2, 1	0.18407+0.05235i	15.8765	0.18359+0.05664i	17.1456	1.2691
31	2, 128	2, 2	-0.06928-0.11345i	-121.4116	-0.06641-0.11328i	-120.3791	1.0325
32	2, 132	8, 1	-0.05573-0.09128i	-121.4066	-0.05273-0.09180i	-119.8760	1.5306

Abbildung B.48: Werte von der unskalierten 8×8 16-Bit 2D-DFT mit Winkelabweichengen >1°, nach der Korrektur mit `unwrap`.

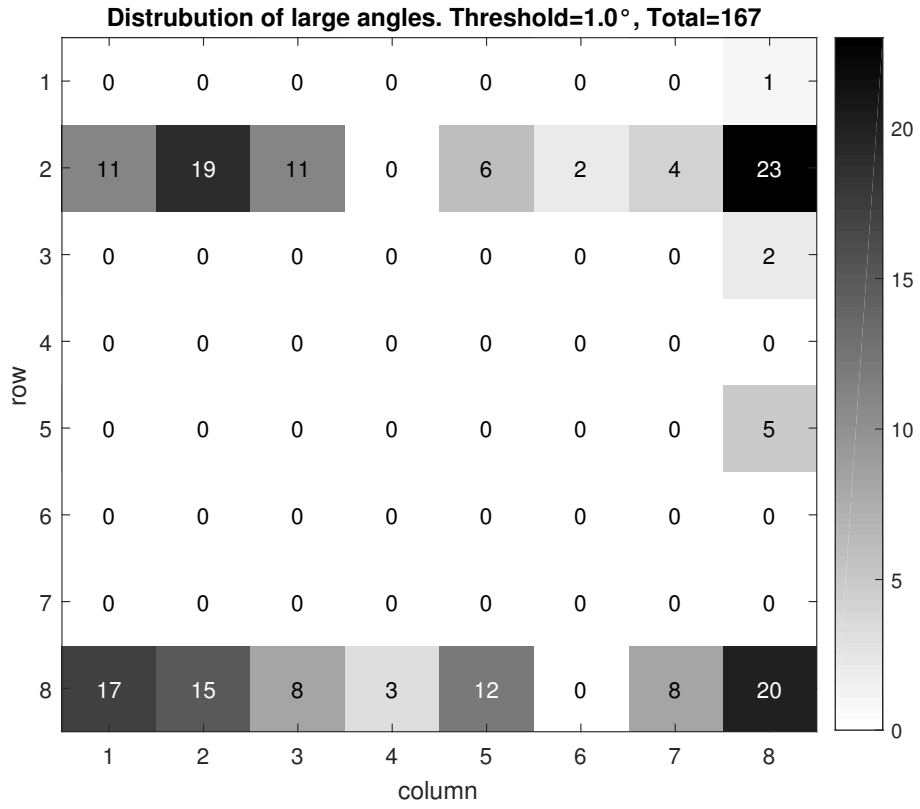

Abbildung B.49: Verteilung von Winkelabweichengen >1° in Ergebnissen vom unskalierten 8×8 16-Bit 2D-DFT., nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

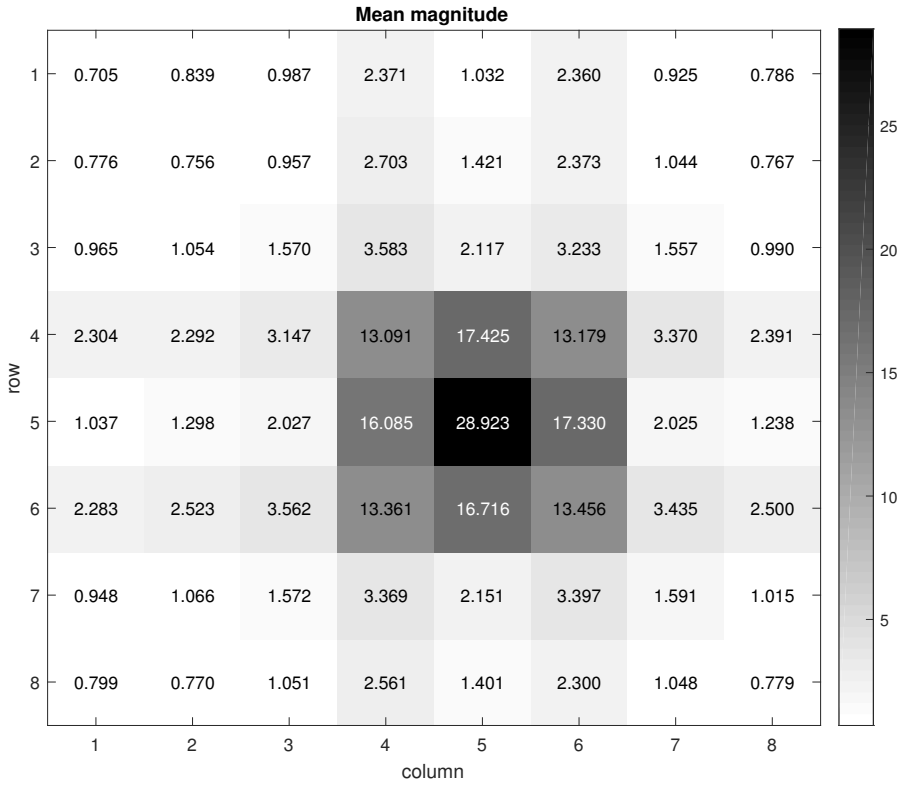

Abbildung B.50: Verteilung vom durchschnittlichen Betrag in Ergebnissen vom unskalierten 8×8 16-Bit 2D-DFT.

B.3.3.3 Testergebnisse von 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix

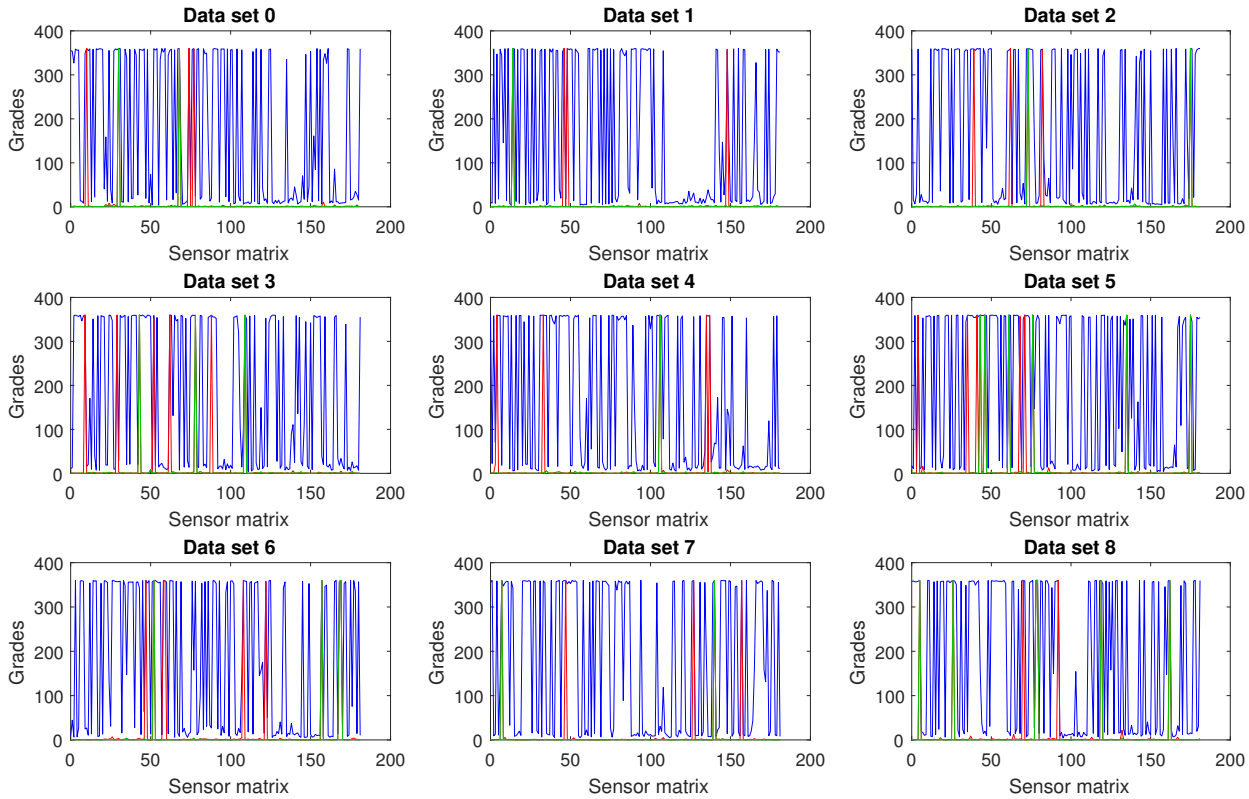

Abbildung B.51: Höchste und durchschnittliche Abweichungen des Argumentes bei der 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, alle Datenreihen
Blau: Daten ohne Shift.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

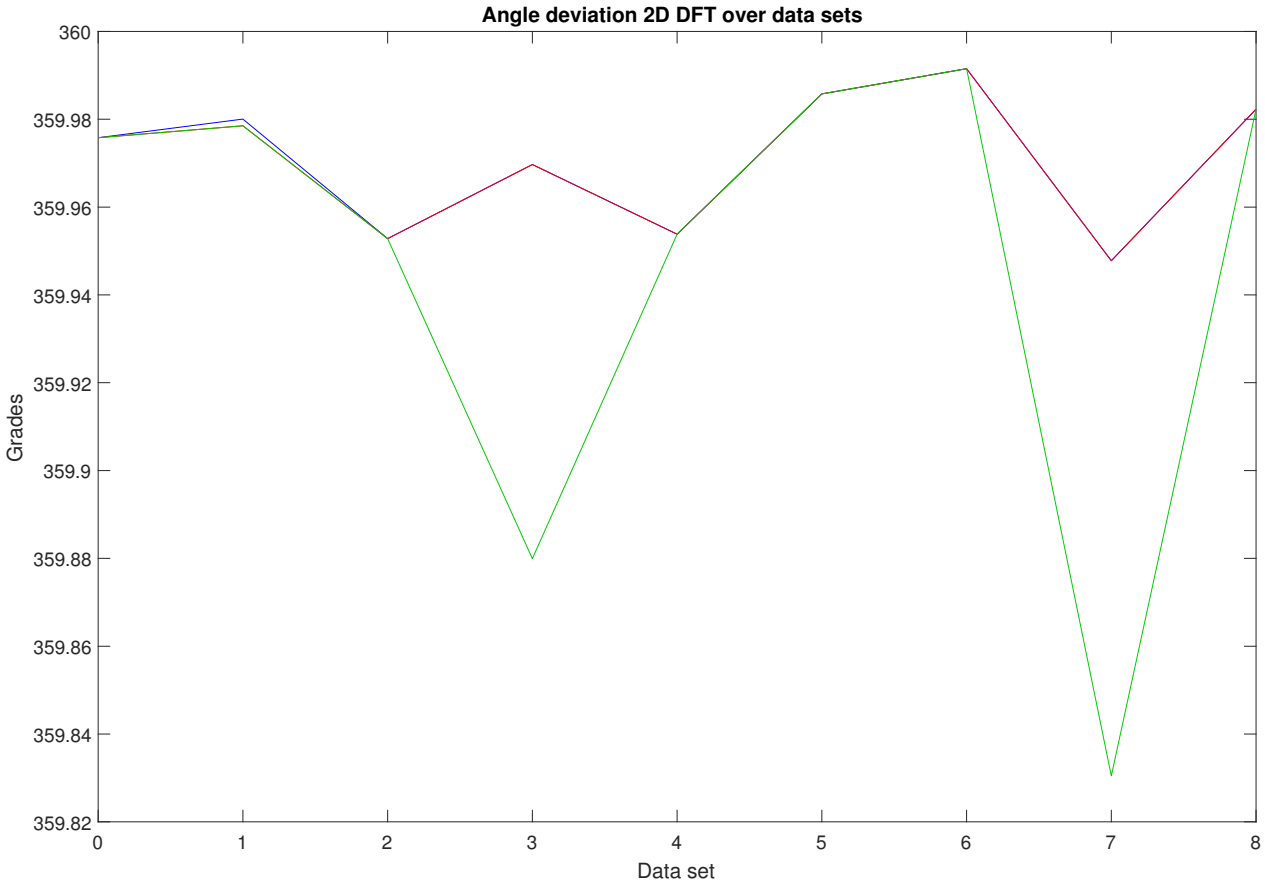

Abbildung B.52: Höchste und durchschnittliche Winkelabweichungenbei der 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen
Blau: Daten ohne Shift
Rot: um 4 Bits nach links verschobene Daten.
Grün: um 5 Bits nach links verschobene Daten.

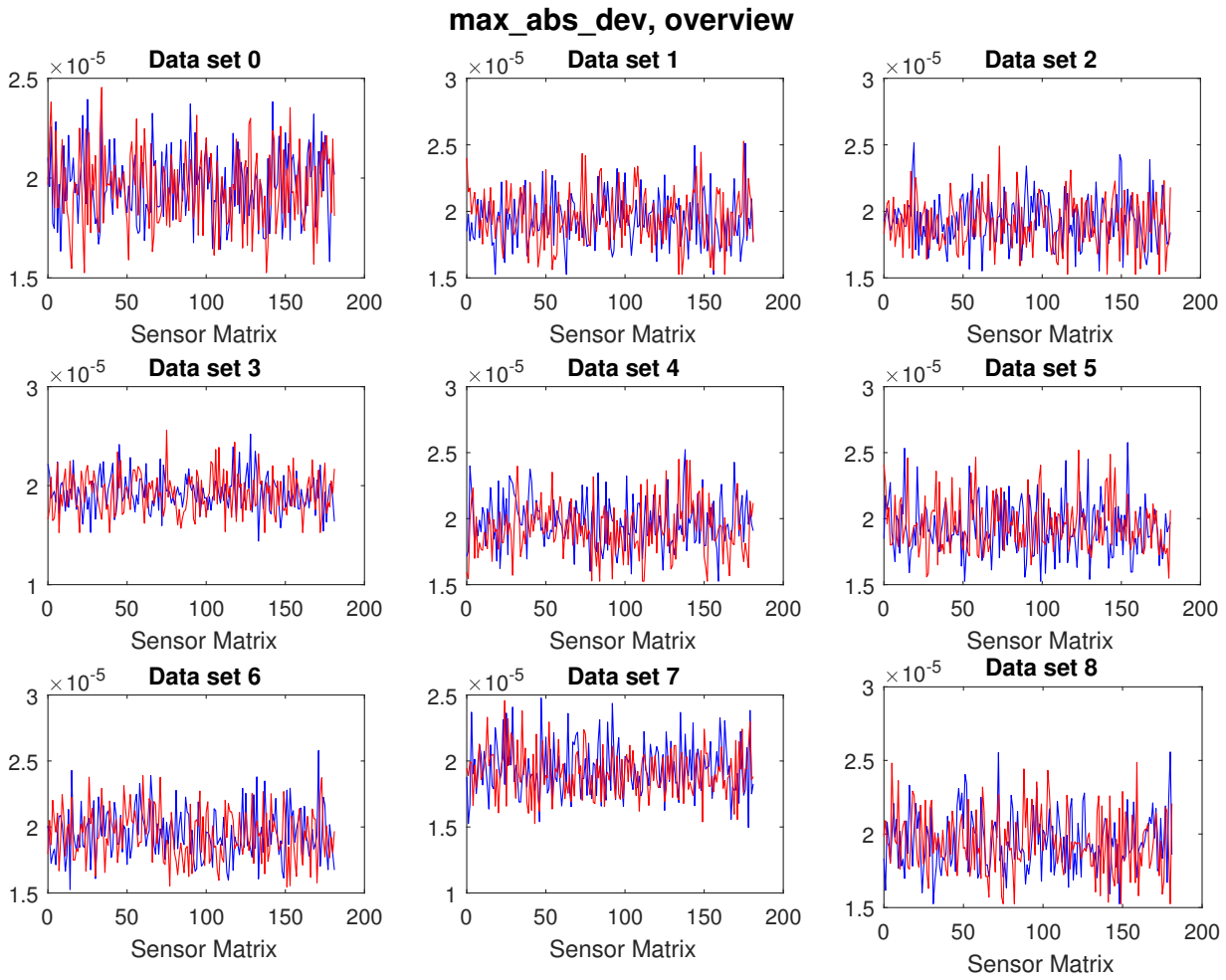

Abbildung B.53: Höchste zahlenmäßige Abweichungen bei der 8×8 16-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

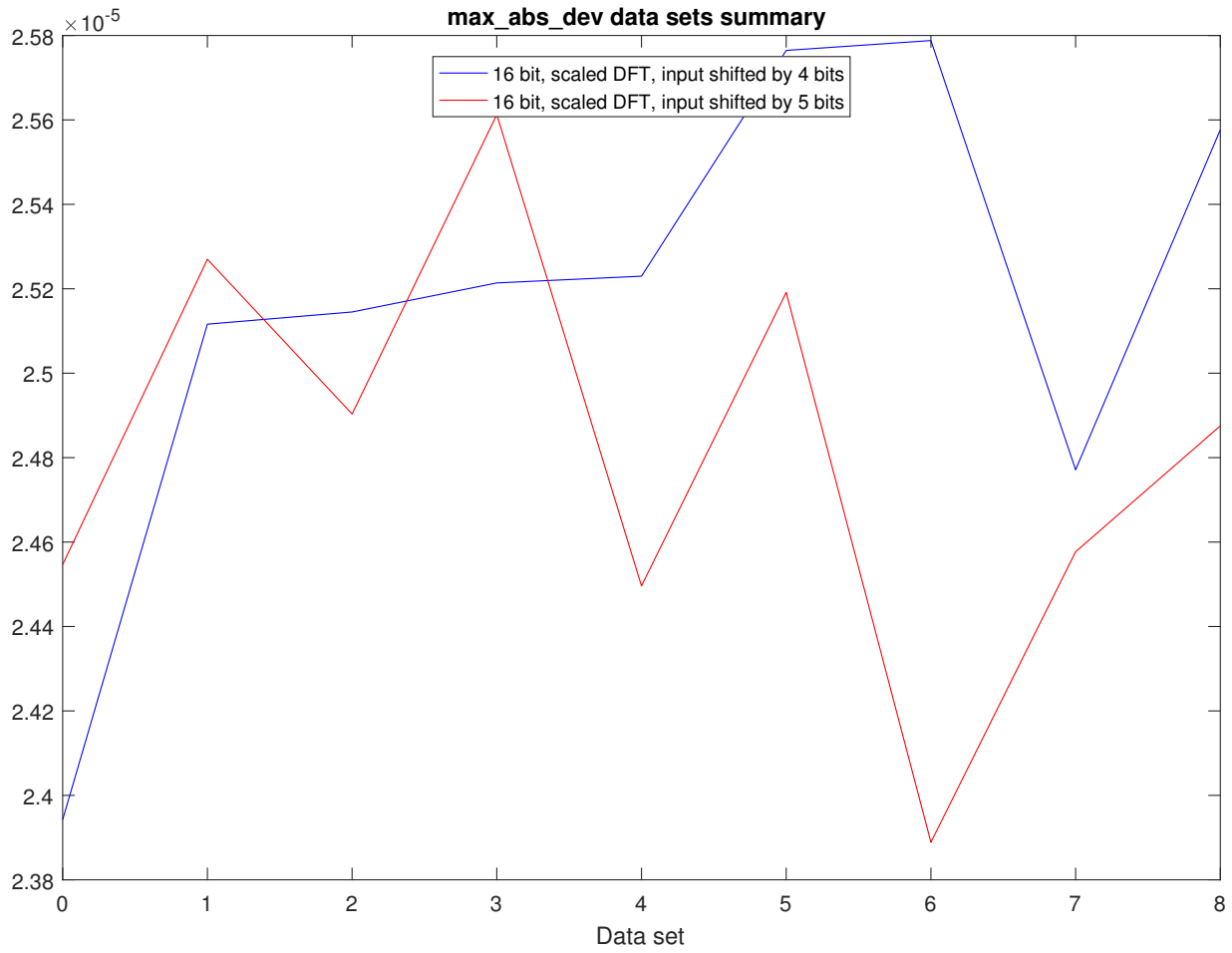

Abbildung B.54: Höchste zahlenmäßige Abweichungen bei der 8×8 16-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten

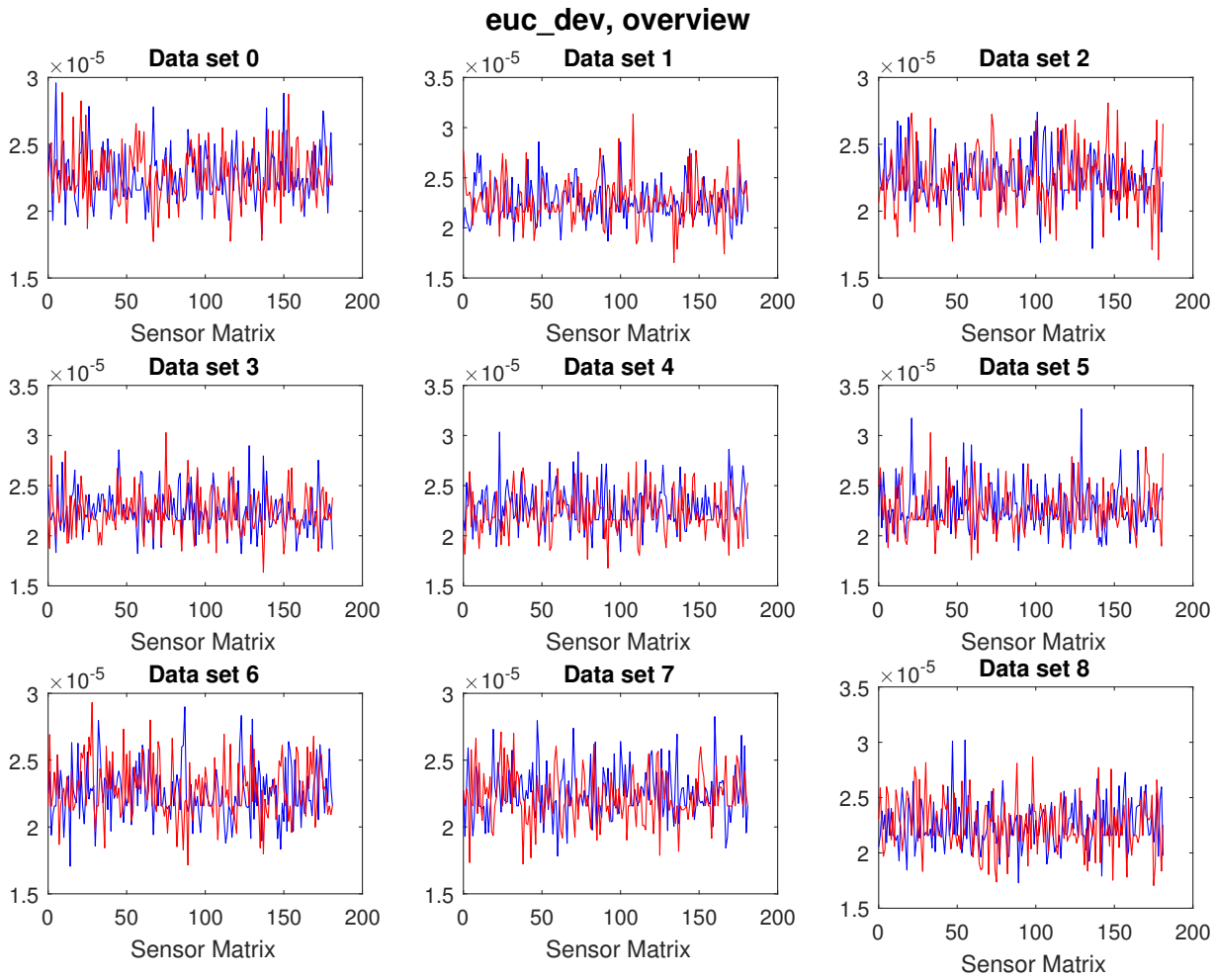

Abbildung B.55: Höchste euklidische Abweichungen bei der 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten

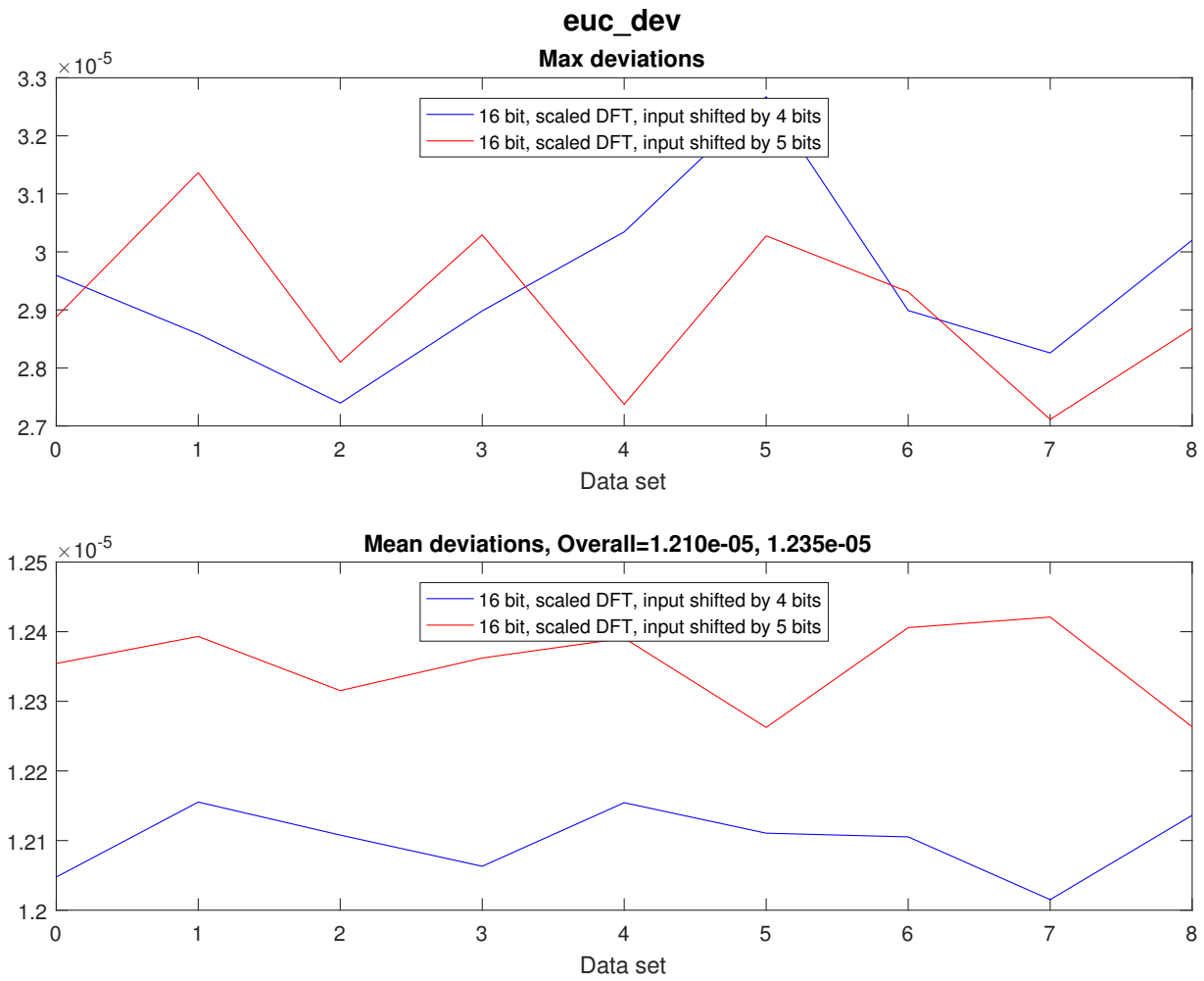

Abbildung B.56: Höchste und durchschnittliche euklidische Abweichung bei der 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

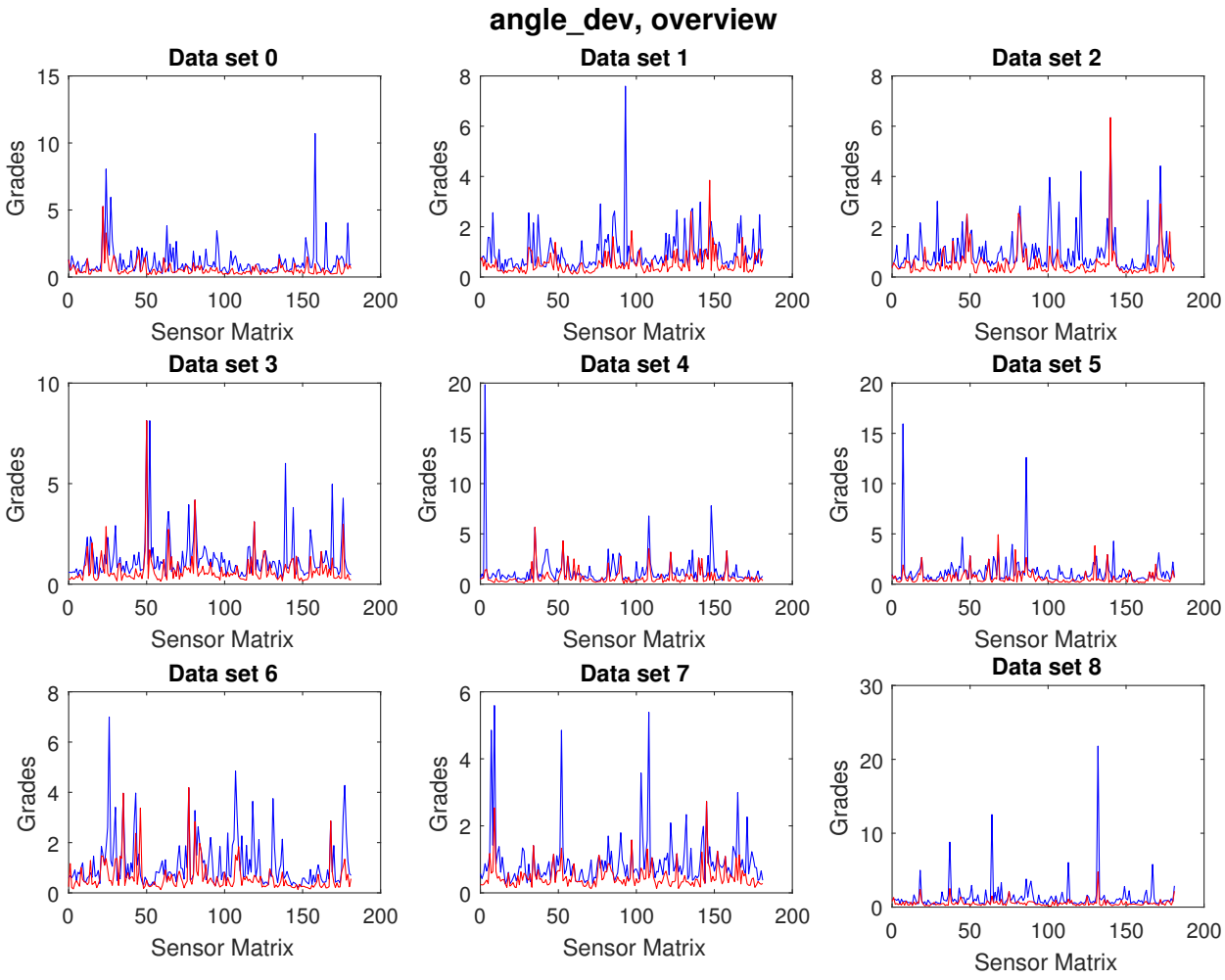

Abbildung B.57: Höchste Abweichungen des Argumentes bei der 8×8 16-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Alle Datensätze sind dargestellt.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

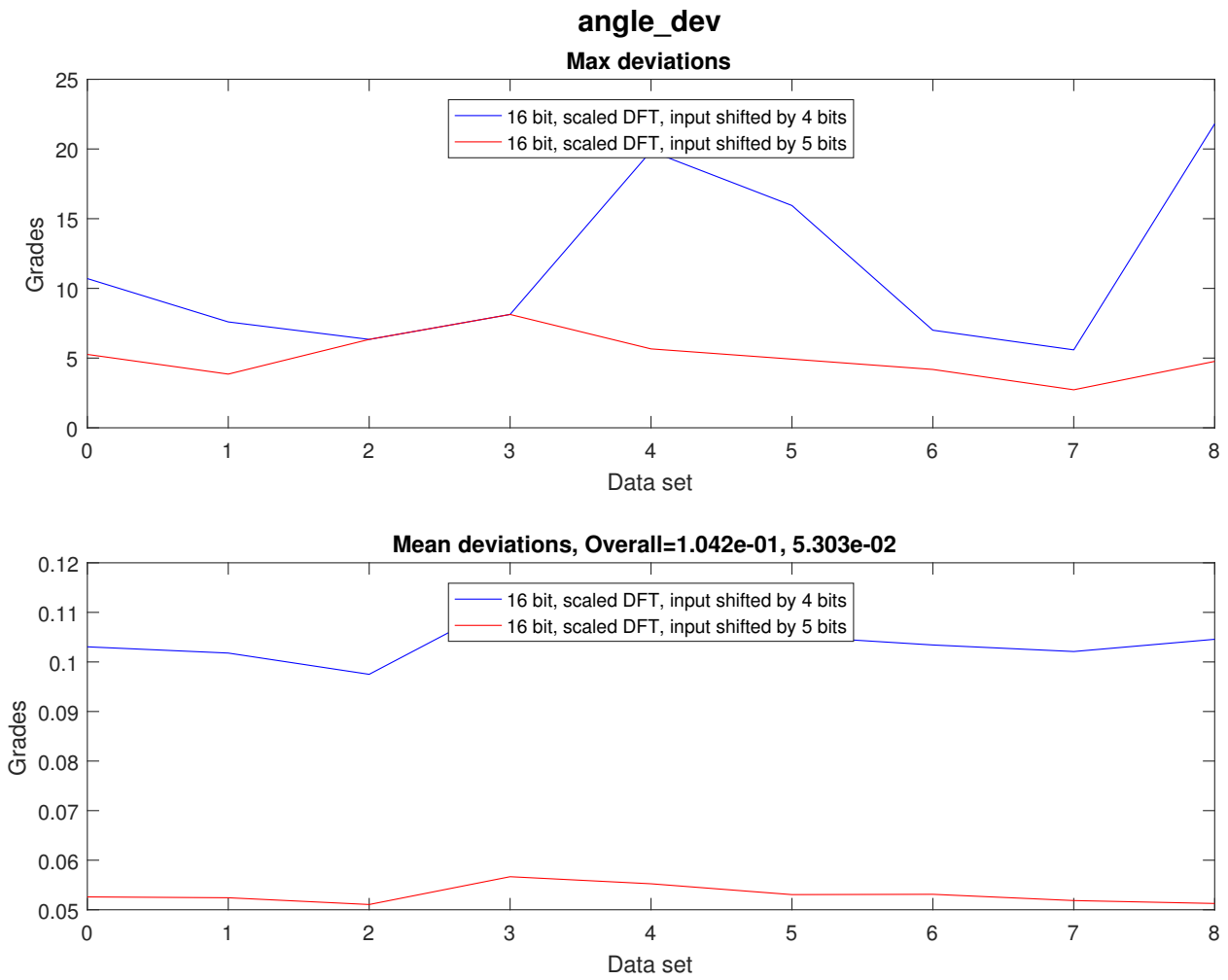

Abbildung B.58: Höchste und durchschnittliche Abweichungen des Argumentes bei der 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Übersicht über Datenreihen
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

	Measurem	Pos in ...	Matlab FFT v...	Matlab ...	FPGA value	FPGA angle	Angle dif
1	1, 1	7, 1	0.00040-0.00168i	-76.7014	0.00037-0.00171i	-77.9052	-1.2038
2	1, 1	1, 7	0.00119+0.00150i	51.4831	0.00122+0.00146i	50.1944	-1.2886
3	1, 13	5, 1	-0.00009-0.00137i	-93.8141	-0.00012-0.00134i	-95.1944	-1.3804
4	1, 23	8, 4	-0.00027-0.00011i	-158.6985	-0.00024-0.00012i	-153.4349	5.2635
5	1, 24	5, 8	-0.00014-0.00101i	-97.7490	-0.00012-0.00104i	-96.7098	1.0392
6	1, 25	5, 8	-0.00029-0.00010i	-161.4892	-0.00031-0.00012i	-158.1986	3.2907
7	1, 26	5, 8	0.00050-0.00041i	-39.7101	0.00049-0.00043i	-41.1859	-1.4758
8	1, 29	5, 8	0.00095+0.00004i	2.7166	0.00092+0.00006i	3.8141	1.0975
9	1, 30	5, 8	0.00046+0.00112i	67.6569	0.00049+0.00110i	66.0375	-1.6194
10	1, 31	5, 8	0.00099+0.00065i	33.3989	0.00098+0.00067i	34.5085	1.1096
11	1, 45	7, 7	-0.00058+0.00085i	124.1597	-0.00061+0.00085i	125.5377	1.3780
12	1, 46	8, 2	0.00051-0.00035i	-34.8759	0.00049-0.00037i	-36.8699	-1.9940
13	1, 49	1, 5	-0.00082-0.00085i	-133.9584	-0.00085-0.00085i	-135	-1.0416
14	1, 50	1, 5	-0.00021-0.00137i	-98.8418	-0.00024-0.00134i	-100.3048	-1.4630
15	1, 62	2, 2	-0.00147-0.00041i	-164.5278	-0.00146-0.00037i	-165.9638	-1.4360
16	1, 66	1, 3	-0.00009-0.00156i	-93.3665	-0.00012-0.00159i	-94.3987	-1.0322
17	1, 81	8, 7	-0.00138-0.00081i	-149.4516	-0.00140-0.00079i	-150.5241	-1.0725
18	1, 136	8, 2	-0.00137-0.00065i	-154.7776	-0.00134-0.00067i	-153.4349	1.3426
19	1, 146	5, 7	0.00034+0.00095i	70.4633	0.00037+0.00098i	69.4440	-1.0194
20	1, 154	6, 8	0.00009+0.00098i	84.9168	0.00006+0.00098i	86.4237	1.5069
21	1, 174	2, 1	-0.00027-0.00098i	-105.3009	-0.00024-0.00098i	-104.0362	1.2647
22	1, 179	1, 2	0.00117+0.00083i	35.3743	0.00116+0.00085i	36.3844	1.0100
23	1, 179	5, 3	0.00174+0.00003i	1.0051	0.00171+0.00000i	0	-1.0051
24	2, 32	5, 8	0.00113+0.00044i	21.4096	0.00116+0.00043i	20.2249	-1.1847
25	2, 33	5, 8	-0.00076+0.00145i	117.6589	-0.00073+0.00146i	116.5651	-1.0938
26	2, 34	5, 8	0.00039+0.00162i	76.4490	0.00037+0.00165i	77.4712	1.0222
27	2, 49	7, 7	0.00116-0.00034i	-16.1443	0.00116-0.00037i	-17.5256	-1.3812
28	2, 81	2, 1	0.00015+0.00155i	84.3923	0.00012+0.00153i	85.4261	1.0337
29	2, 83	8, 8	-0.00117-0.00137i	-130.5759	-0.00116-0.00140i	-129.5597	1.0162
30	2, 86	3, 3	0.00104+0.00021i	11.6336	0.00104+0.00024i	13.2405	1.6069
31	2, 98	2, 1	0.00045+0.00104i	66.6427	0.00049+0.00104i	64.7989	-1.8438
32	2, 103	1, 8	0.00095+0.00010i	6.0527	0.00098+0.00012i	7.1250	1.0723

Abbildung B.59: Werte von der 8×8 16-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix mit Winkelabweichengen >1°, nach der Korrektur mit `unwrap`.

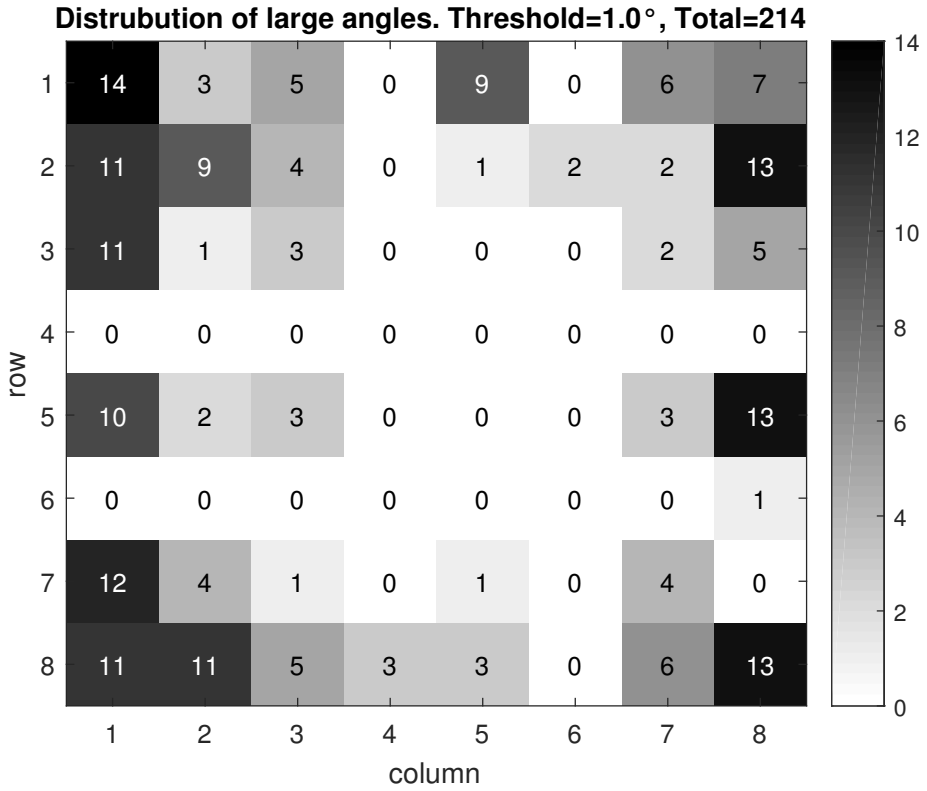

Abbildung B.60: Verteilung von Winkelabweichengen >1° in Ergebnissen vom 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

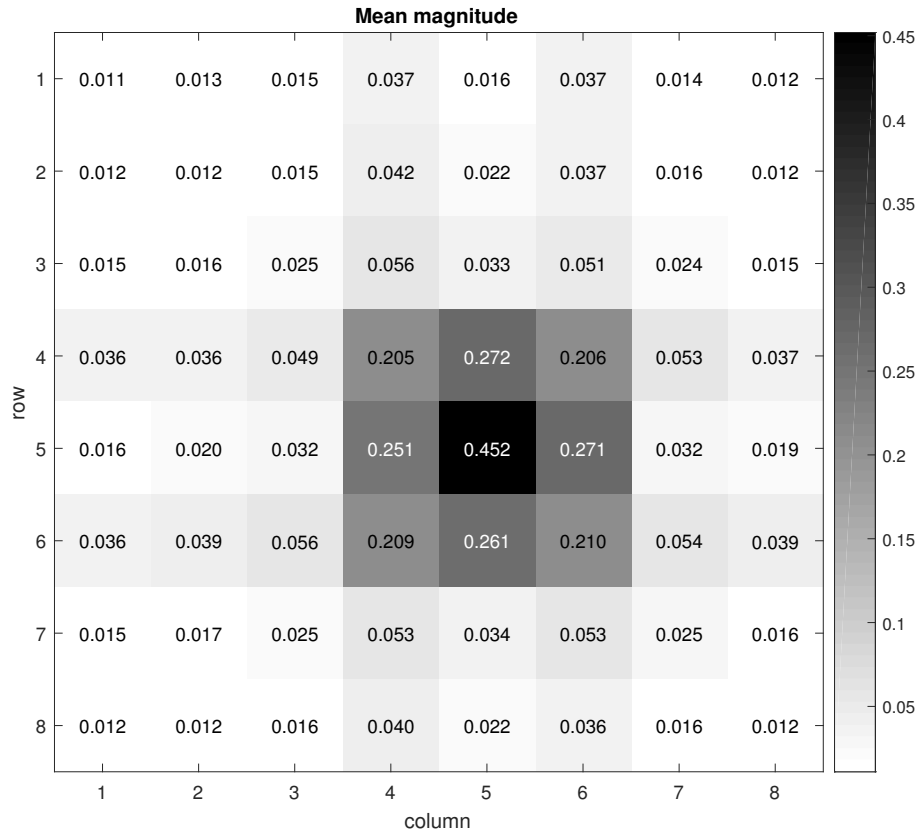

Abbildung B.61: Verteilung vom durchschnittlichen Betrag in Ergebnissen vom 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix.

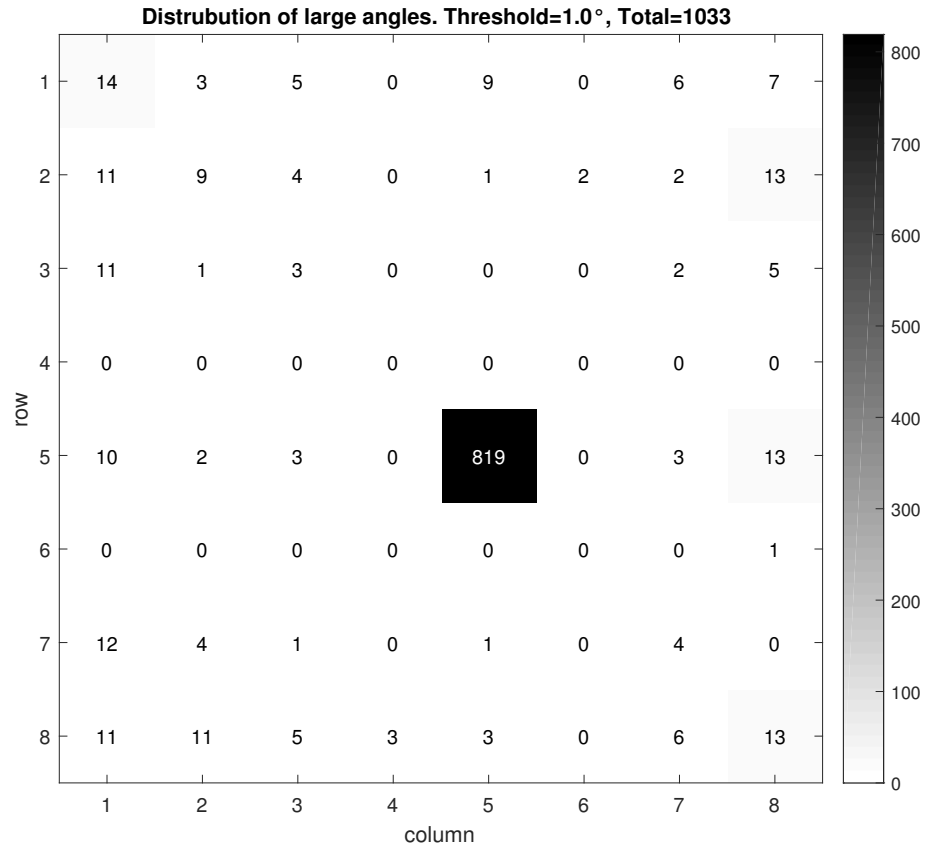

Abbildung B.62: Verteilung von Winkelabweichengen >1° in Ergebnissen vom 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix und mittelwertfreien Sensordaten, nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

B.3.3.4 Testergebnisse von 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix

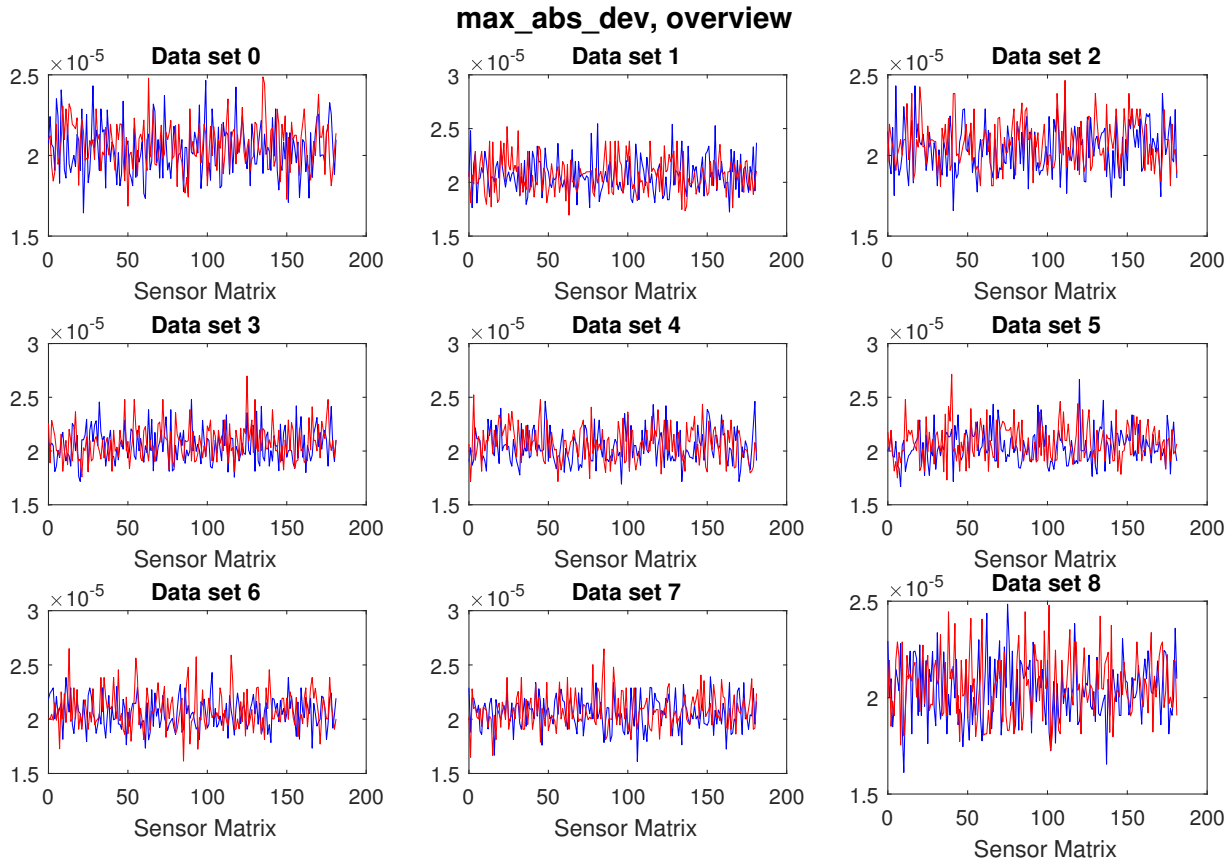

Abbildung B.63: Höchste zahlenmäßige Abweichungen bei der 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.
Blau: durch 2 geteilte und gerundete Daten.
Rot: nicht dividierte (skalierten) Daten.

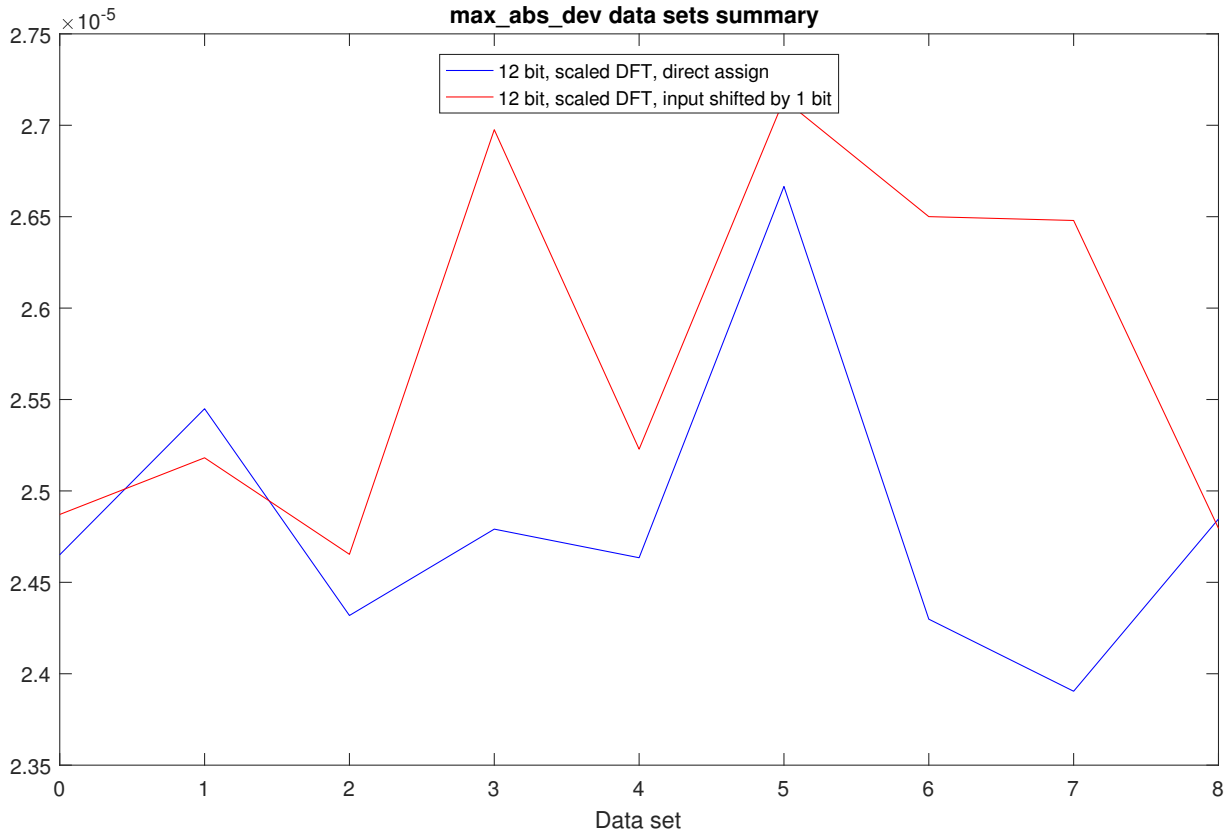

Abbildung B.64: Höchste zahlenmäßige Abweichungen bei der 8×8 12-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten

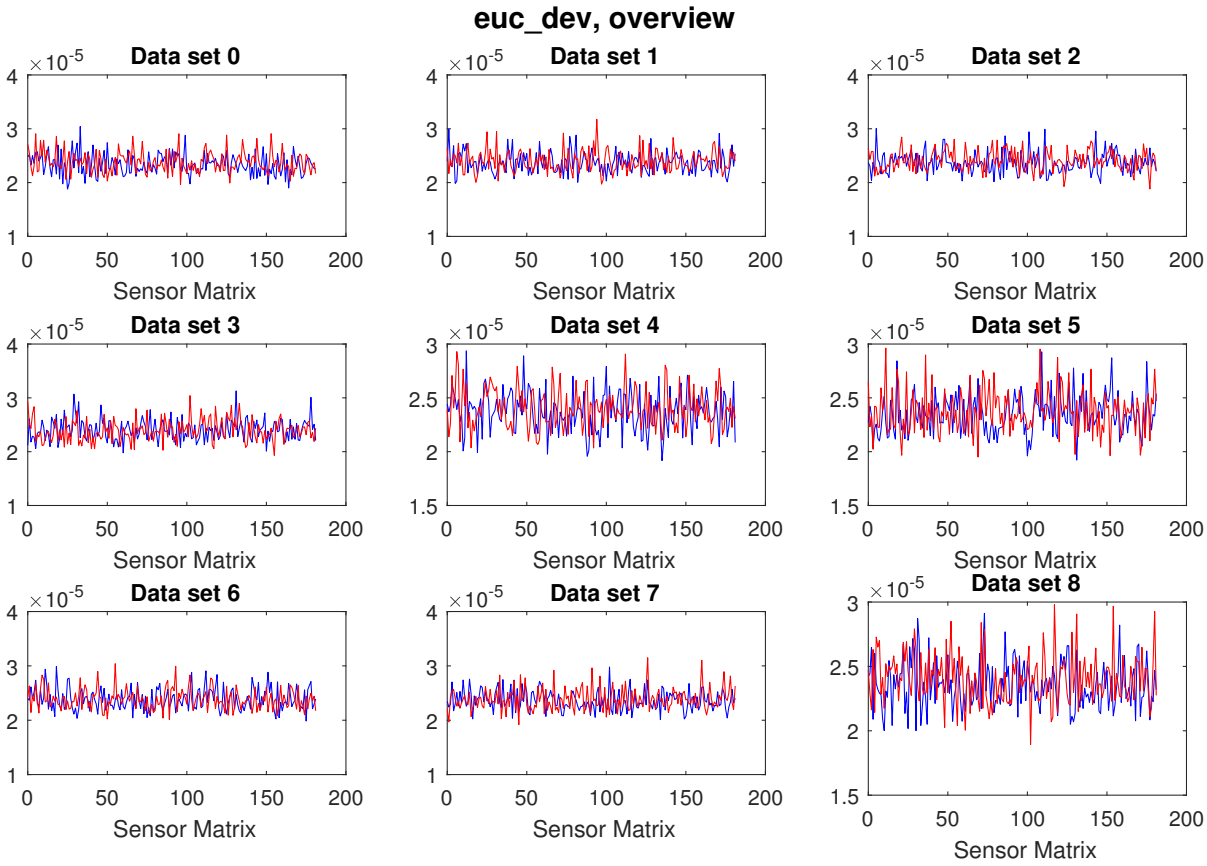

Abbildung B.65: Höchste euklidische Abweichungen bei der 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten

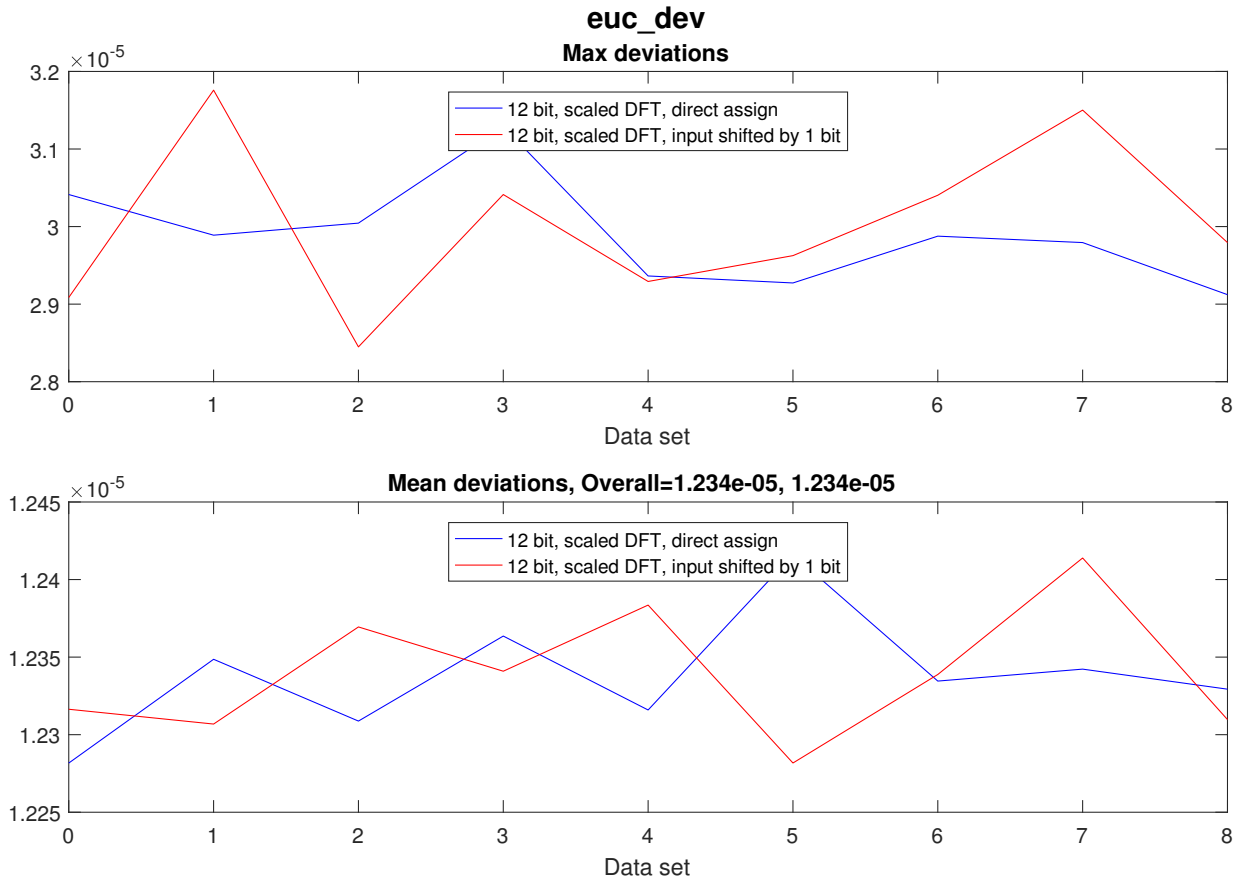

Abbildung B.66: Höchste und durchschnittliche euklidische Abweichung bei der 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

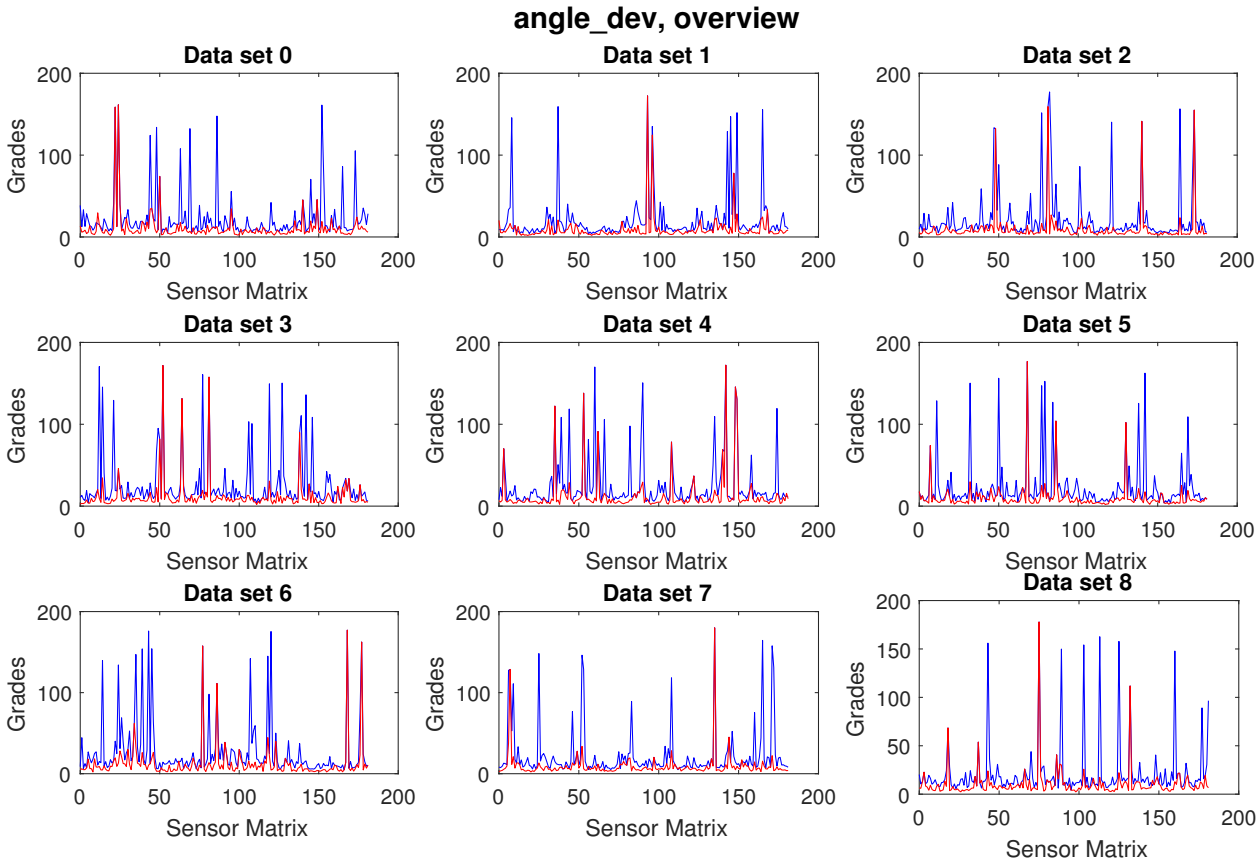

Abbildung B.67: Höchste Abweichungen des Argumentes bei der 8×8 12-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Alle Datensätze sind dargestellt.
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

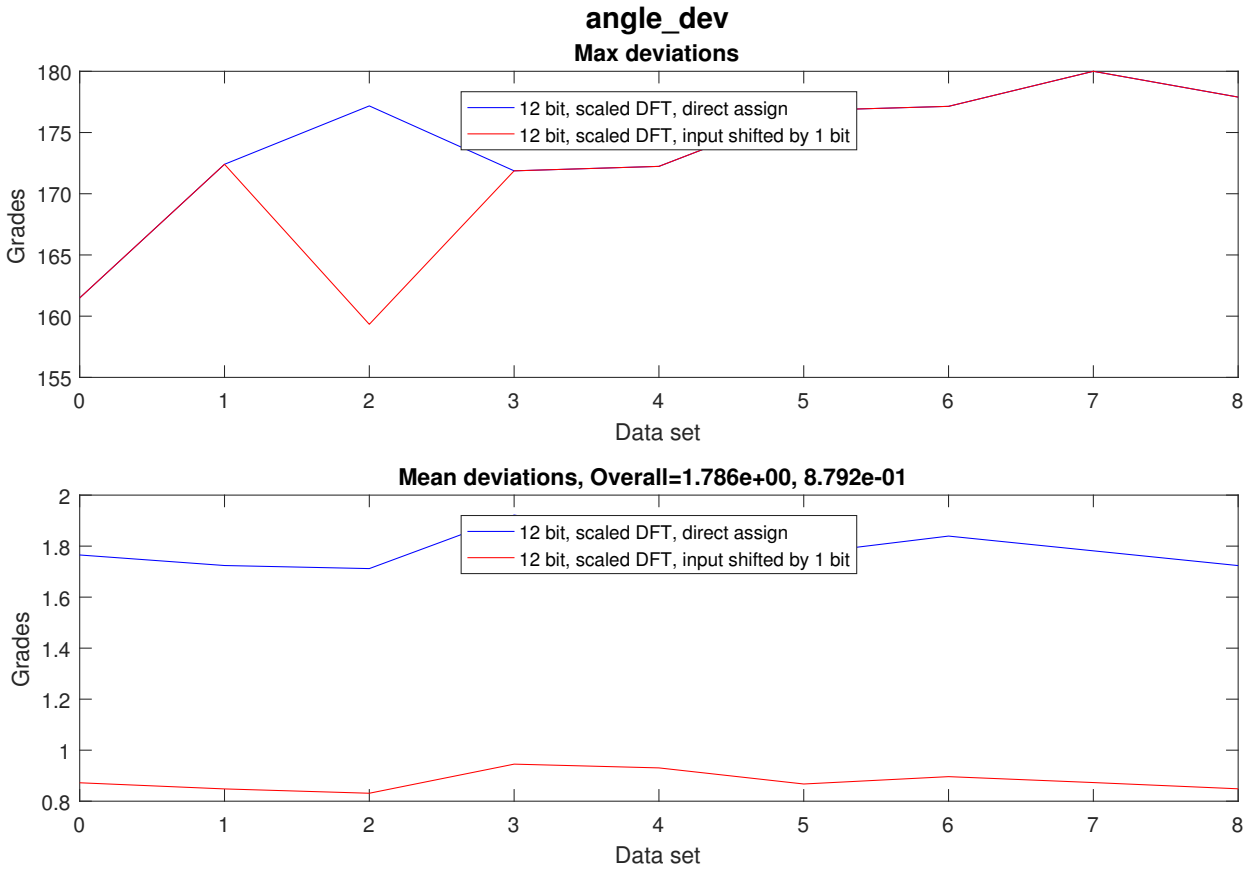

Abbildung B.68: Höchste und durchschnittliche Abweichungen des Argumentes bei der 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Übersicht über Datenreihen
Blau: um 4 Bits nach links verschobene Daten.
Rot: um 5 Bits nach links verschobene Daten.

	Measurem	Pos in ...	Matlab FFT v...	Matlab ...	FPGA value	FPGA angle	Angle dif
1	1, 1	1, 1	0.00641+0.01721i	69.5777	0.00586+0.01758i	71.5651	1.9874
2	1, 1	7, 1	0.00040-0.00168i	-76.7014	0.00000-0.00195i	-90	-13.2986
3	1, 1	1, 2	-0.00235-0.00279i	-130.0428	-0.00195-0.00293i	-123.6901	6.3527
4	1, 1	3, 2	-0.01411+0.01347i	136.3148	-0.01367+0.01367i	135	-1.3148
5	1, 1	7, 2	0.00043+0.02116i	88.8483	0.00000+0.02148i	90	1.1517
6	1, 1	8, 2	-0.00604+0.00221i	159.8824	-0.00586+0.00195i	161.5651	1.6827
7	1, 1	2, 3	-0.01707+0.00032i	178.9172	-0.01758+0.00000i	180	1.0828
8	1, 1	3, 3	-0.00134-0.00873i	-98.7462	-0.00098-0.00879i	-96.3402	2.4060
9	1, 1	7, 3	0.02026+0.01025i	26.8405	0.02051+0.00977i	25.4633	-1.3772
10	1, 1	8, 3	-0.01110+0.01730i	122.6822	-0.01074+0.01758i	121.4296	-1.2526
11	1, 1	1, 5	-0.01239-0.00336i	-164.8405	-0.01172-0.00391i	-161.5651	3.2754
12	1, 1	5, 6	-0.01375-0.00722i	-152.2931	-0.01367-0.00684i	-153.4349	-1.1418
13	1, 1	1, 7	0.00119+0.00150i	51.4831	0.00098+0.00195i	63.4349	11.9519
14	1, 1	1, 8	0.00243+0.00984i	76.1101	0.00293+0.00977i	73.3008	-2.8094
15	1, 1	2, 8	0.00400-0.00649i	-58.3424	0.00391-0.00684i	-60.2551	-1.9127
16	1, 1	7, 8	0.01522+0.00535i	19.3502	0.01563+0.00488i	17.3540	-1.9962
17	1, 1	8, 8	0.00062+0.01785i	88.0125	0.00098+0.01758i	86.8202	-1.1923
18	1, 2	1, 1	0.00146+0.01987i	85.7831	0.00195+0.01953i	84.2894	-1.4937
19	1, 2	2, 1	-0.01370+0.00648i	154.6682	-0.01367+0.00684i	153.4349	-1.2332
20	1, 2	3, 1	-0.01294-0.00143i	-173.6746	-0.01270-0.00195i	-171.2538	2.4208
21	1, 2	5, 1	0.00470+0.00797i	59.4578	0.00488+0.00781i	57.9946	-1.4631
22	1, 2	7, 1	-0.00201-0.00467i	-113.3340	-0.00195-0.00488i	-111.8014	1.5326
23	1, 2	8, 1	0.01230+0.00679i	28.9050	0.01172+0.00684i	30.2564	1.3514
24	1, 2	1, 2	-0.00037-0.00364i	-95.8229	0.00000-0.00391i	-90	5.8229
25	1, 2	2, 2	0.00244+0.01497i	80.7585	0.00293+0.01465i	78.6901	-2.0684
26	1, 2	8, 2	-0.00620+0.00295i	154.5308	-0.00586+0.00293i	153.4349	-1.0958
27	1, 2	2, 3	-0.01730+0.00056i	178.1313	-0.01758+0.00098i	176.8202	-1.3112
28	1, 2	1, 7	0.00076+0.00342i	77.4170	0.00098+0.00391i	75.9638	-1.4533
29	1, 2	5, 7	0.01315+0.00787i	30.9051	0.01367+0.00781i	29.7449	-1.1602
30	1, 2	7, 7	0.02011+0.00549i	15.2772	0.01953+0.00586i	16.6992	1.4221
31	1, 2	1, 8	0.00079+0.00801i	84.3509	0.00098+0.00781i	82.8750	-1.4759
32	1, 2	5, 8	-0.00361-0.01166i	-107.1842	-0.00391-0.01172i	-108.4349	-1.2508

Abbildung B.69: Werte von der 8×8 12-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix mit Winkelabweichengen >1°, nach der Korrektur mit `unwrap`.

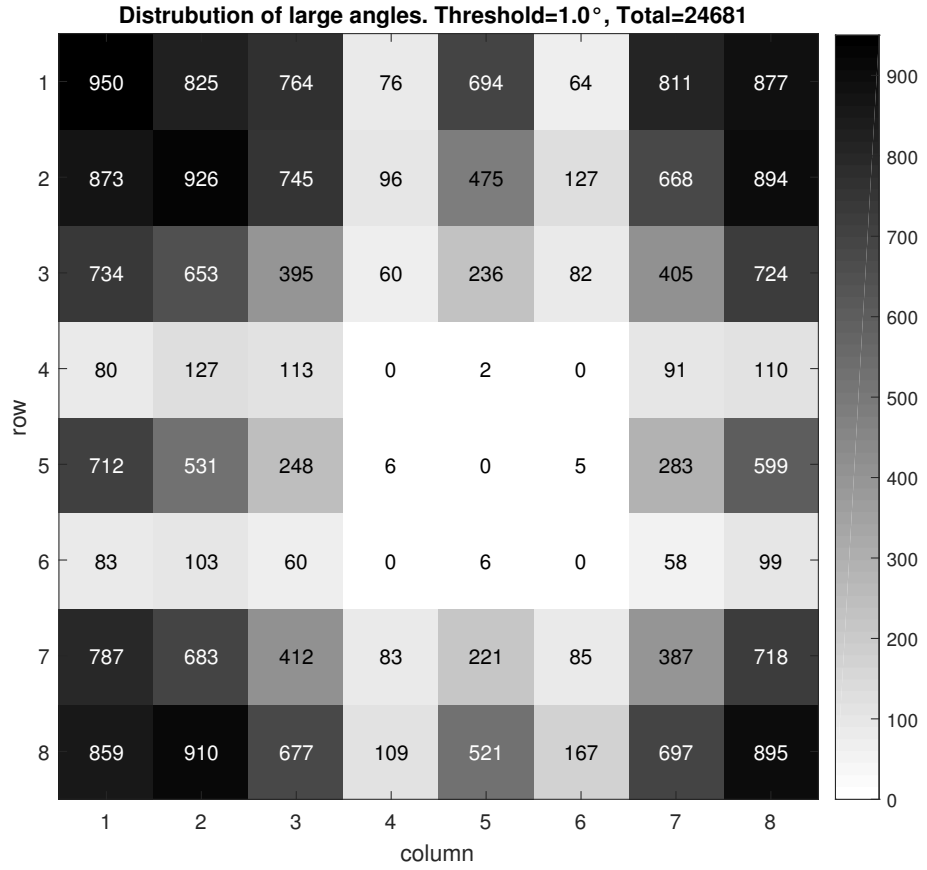

Abbildung B.70: Verteilung von Winkelabweichengen >1° in Ergebnissen vom 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

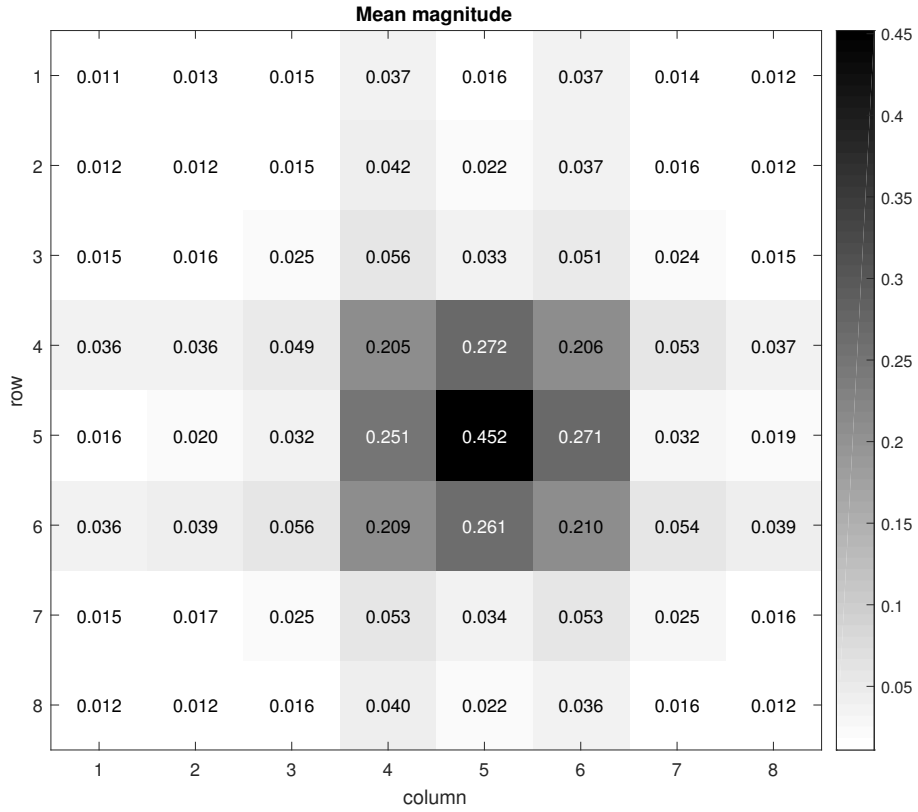

Abbildung B.71: Verteilung vom durchschnittlichen Betrag in Ergebnissen vom 8×8 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix.

B.3.3.5 Testergebnisse von 8×8 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix

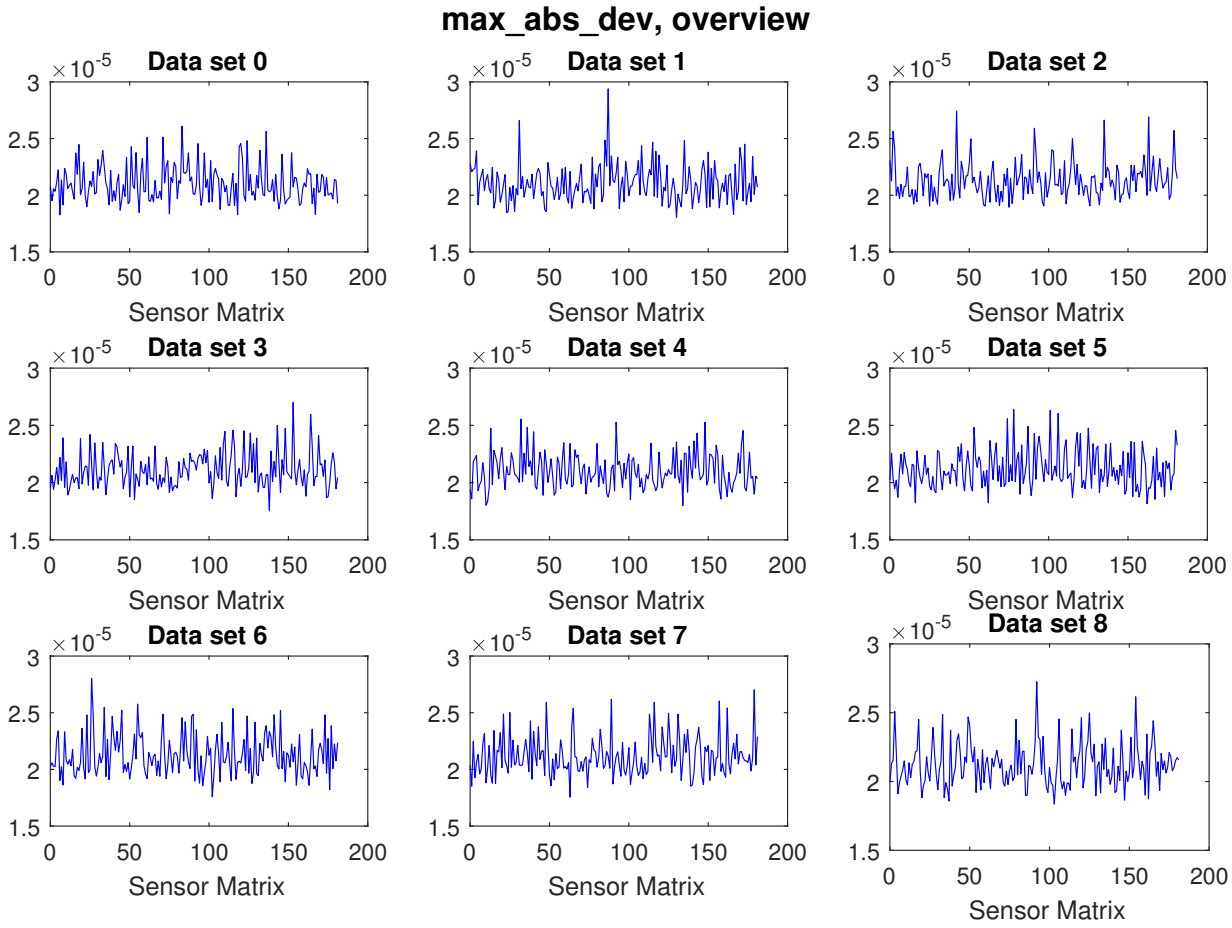

Abbildung B.72: Höchste zahlenmäßige Abweichungen bei der 15×15 16-Bit 2D-DFT mit der heruntekalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.

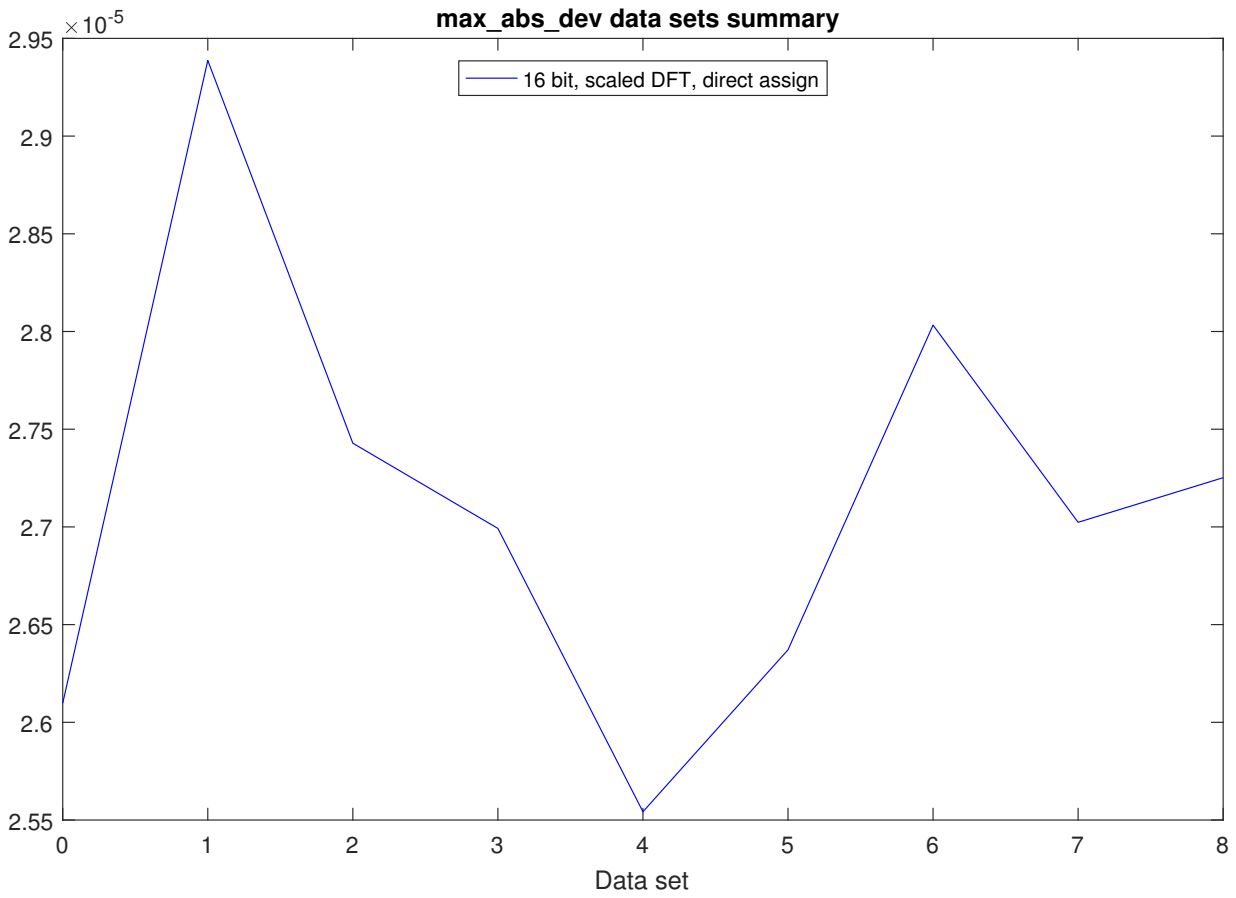

Abbildung B.73: Höchste zahlenmäßige Abweichungen bei der 15×15 16-Bit 2D-DFT mit der heruntekalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.

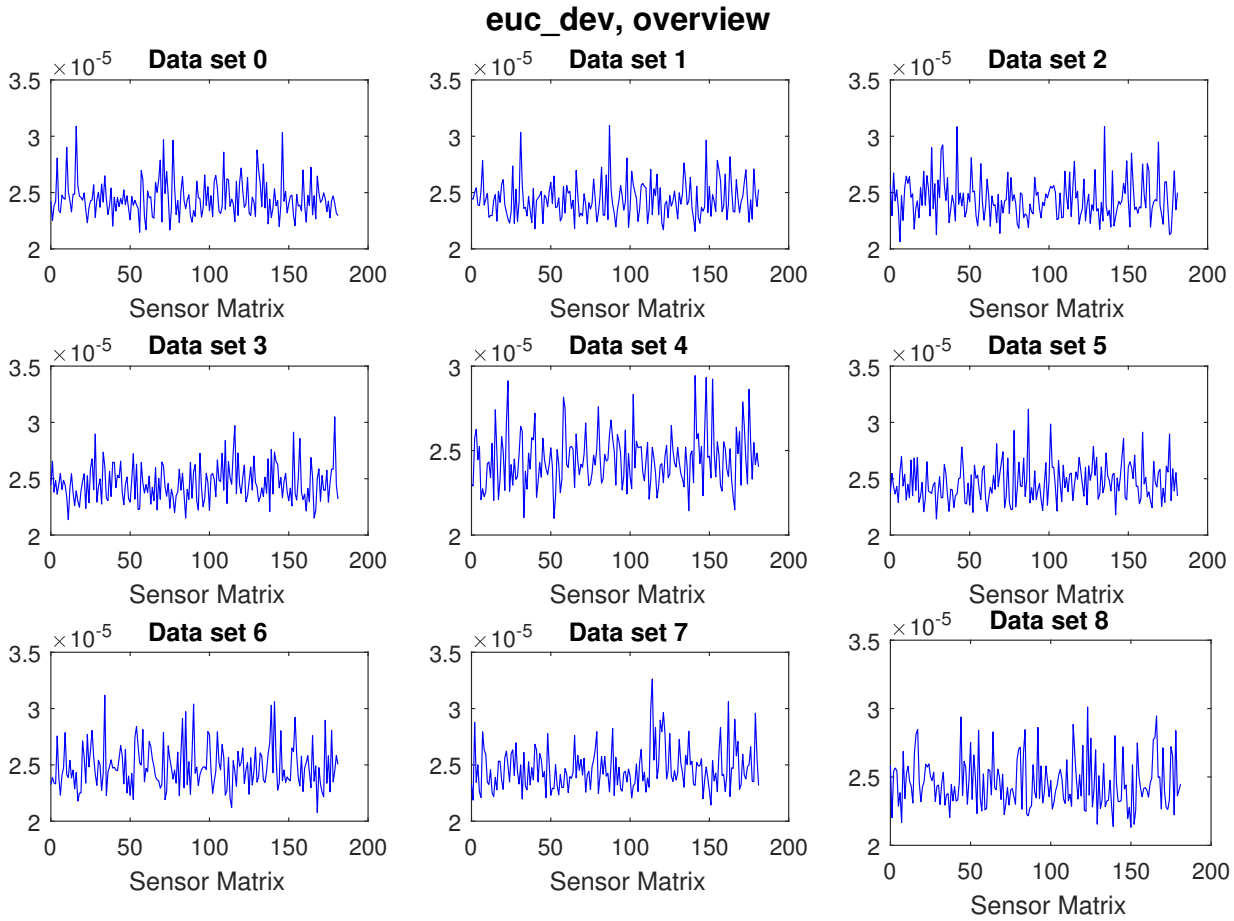

Abbildung B.74: Höchste euklidische Abweichungen bei der 15×15 16-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.

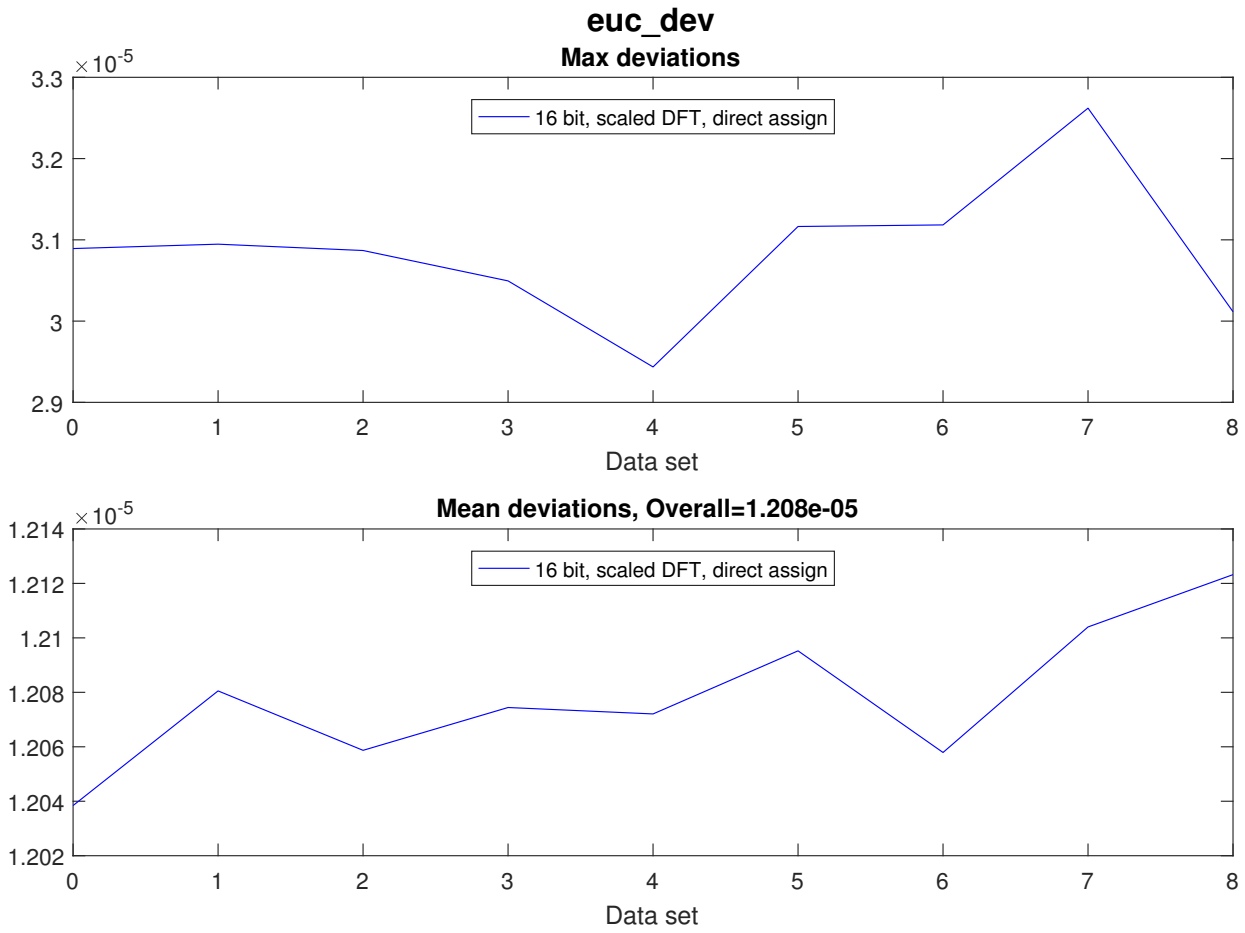

Abbildung B.75: Höchste und durchschnittliche euklidische Abweichung bei der 15×15 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.

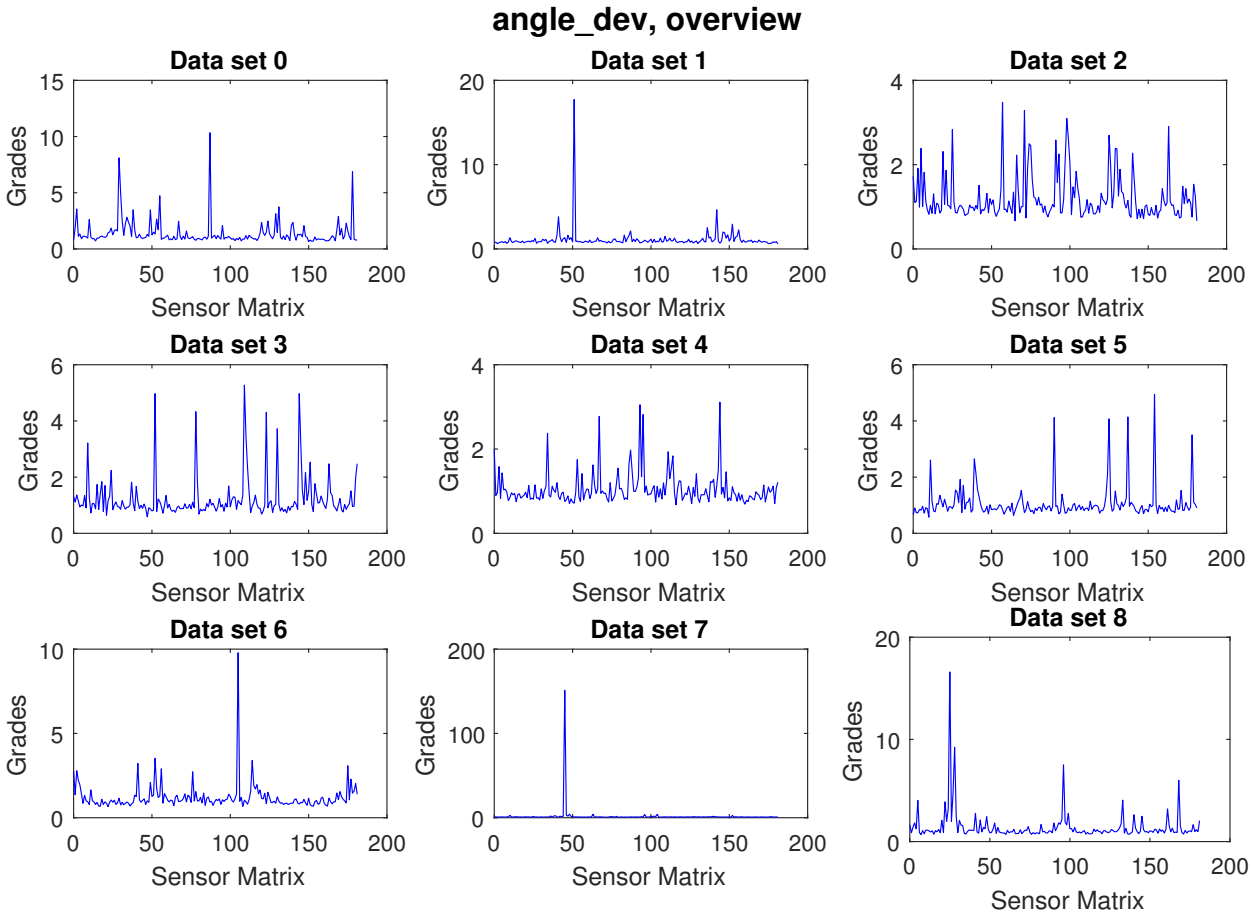

Abbildung B.76: Höchste Abweichungen des Argumentes bei der 15×15 16-Bit 2D-DFT mit der heruntergeskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Alle Datensätze sind dargestellt.

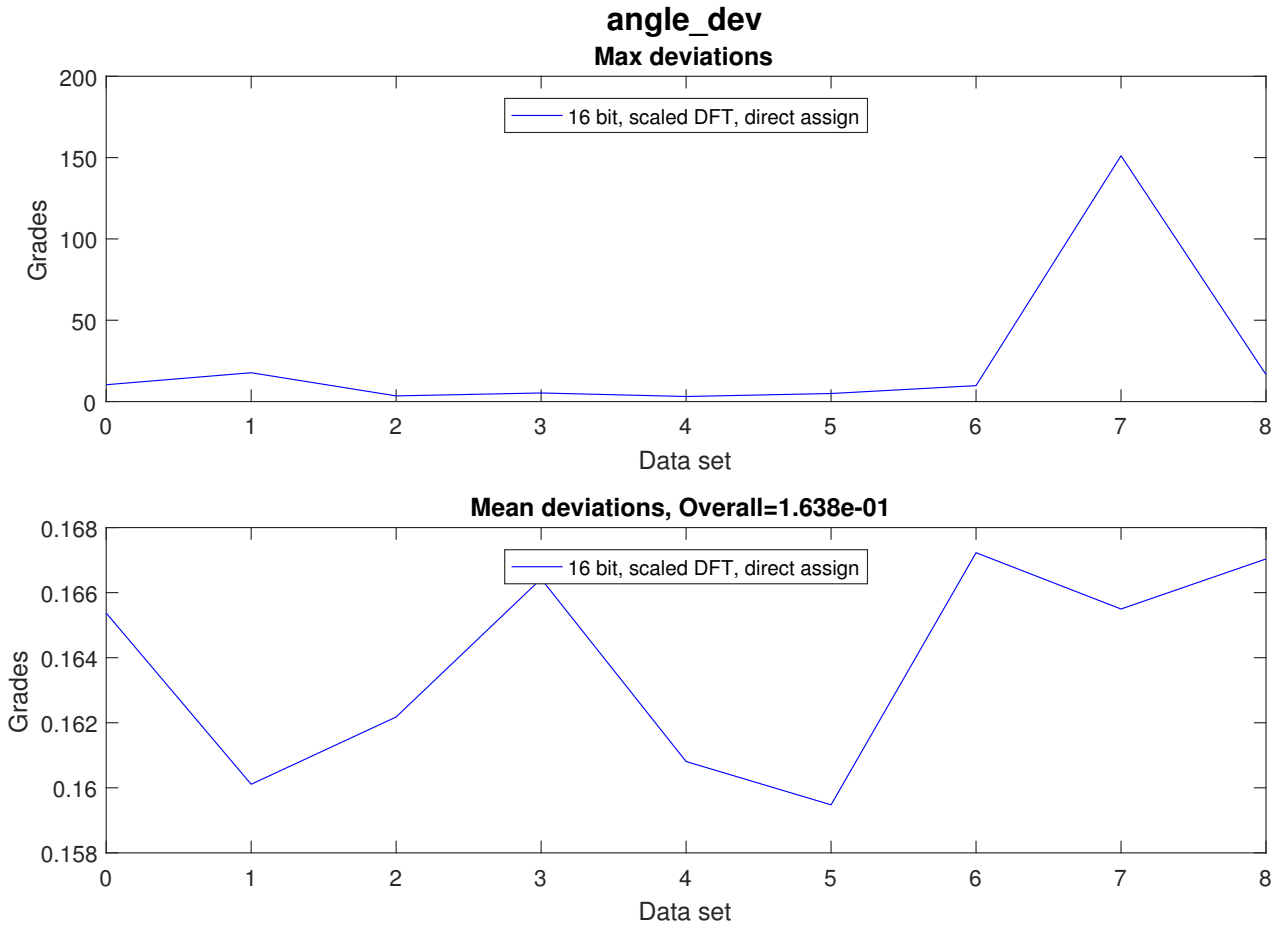

Abbildung B.77: Höchste und durchschnittliche Abweichungen des Argumentes bei der 15×15 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`. Übersicht über Datenreihen

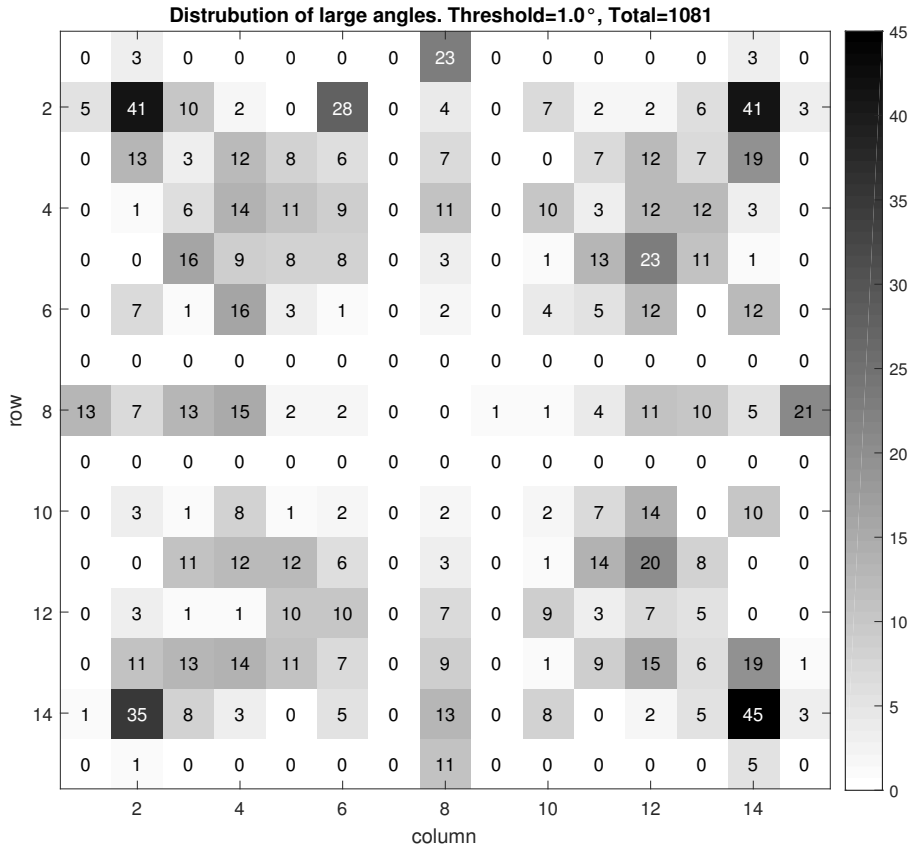

Abbildung B.78: Verteilung von Winkelabweichengen >1° in Ergebnissen von der 15×15 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

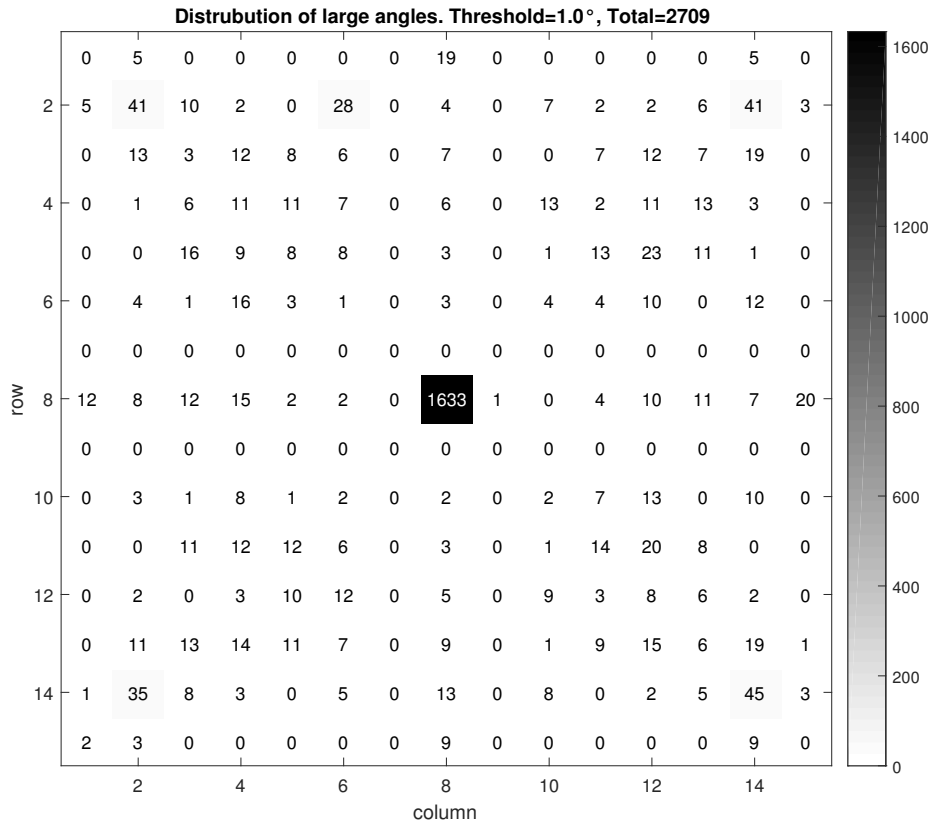

Abbildung B.79: Verteilung von Winkelabweichengen >1° in Ergebnissen von der 15×15 16-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix und mittelwertfreien Sensordaten, nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

B.3.3.6 Testergebnisse von 15×15 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix

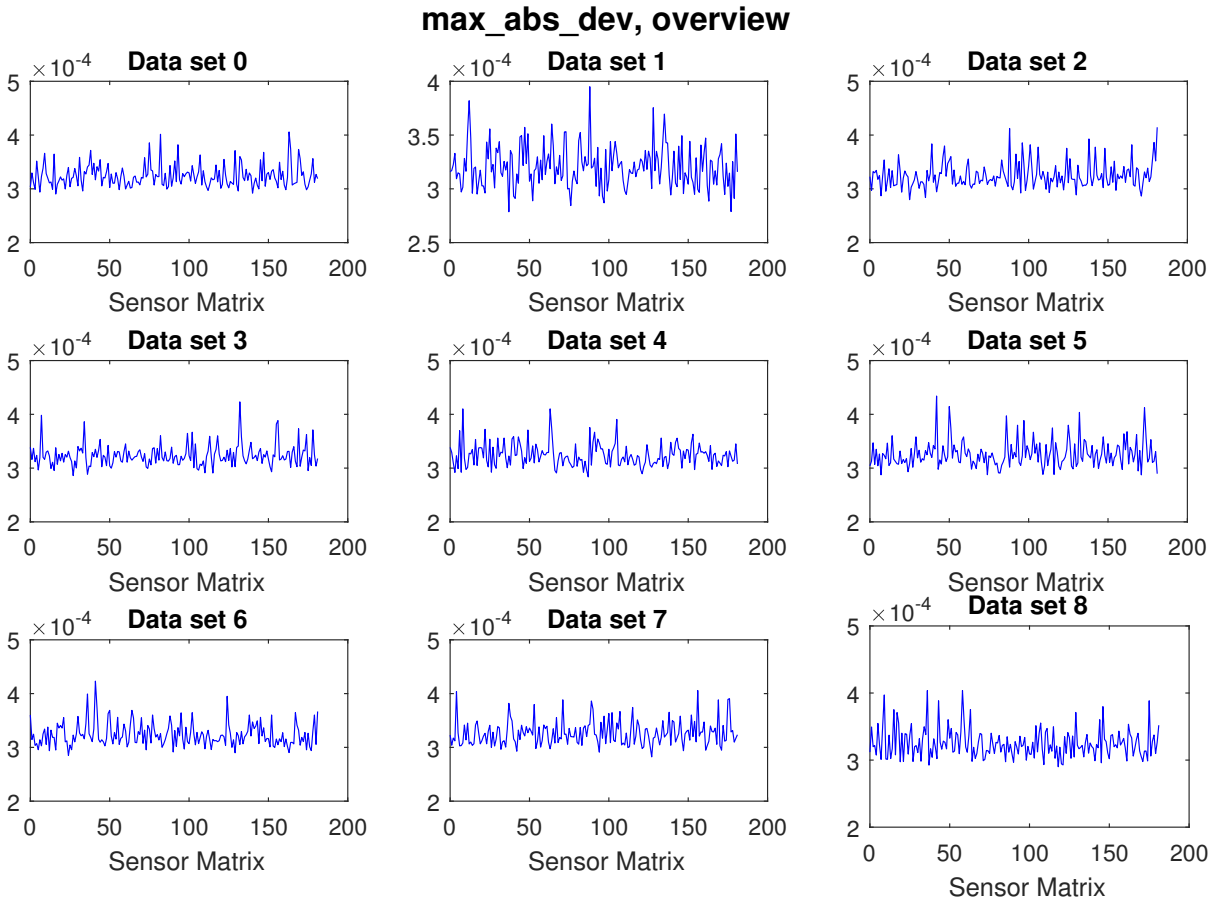

Abbildung B.80: Höchste zahlenmäßige Abweichungen bei der 15×15 12-Bit 2D-DFT mit der heruntergeskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.

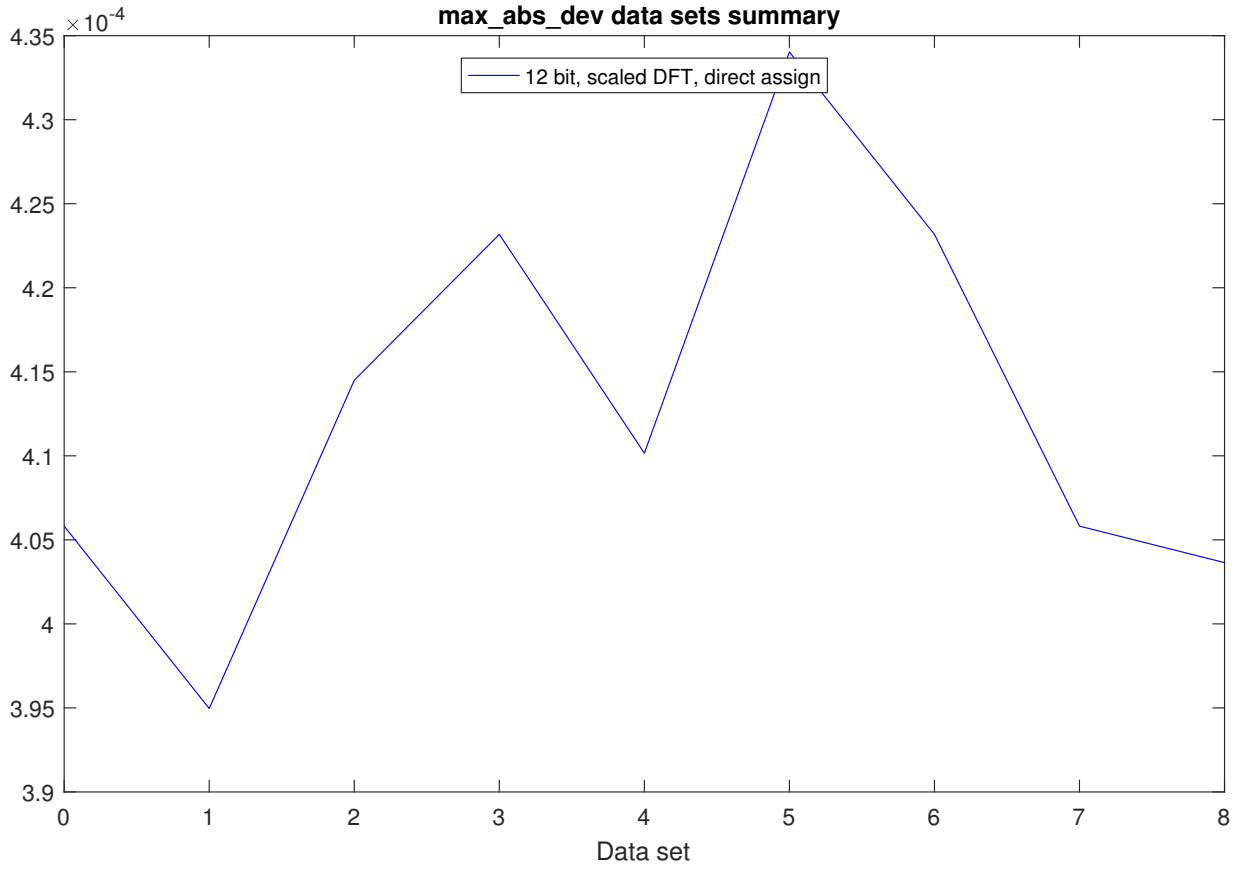

Abbildung B.81: Höchste zahlenmäßige Abweichungen bei der 15×15 12-Bit 2D-DFT mit der heruntergeskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.

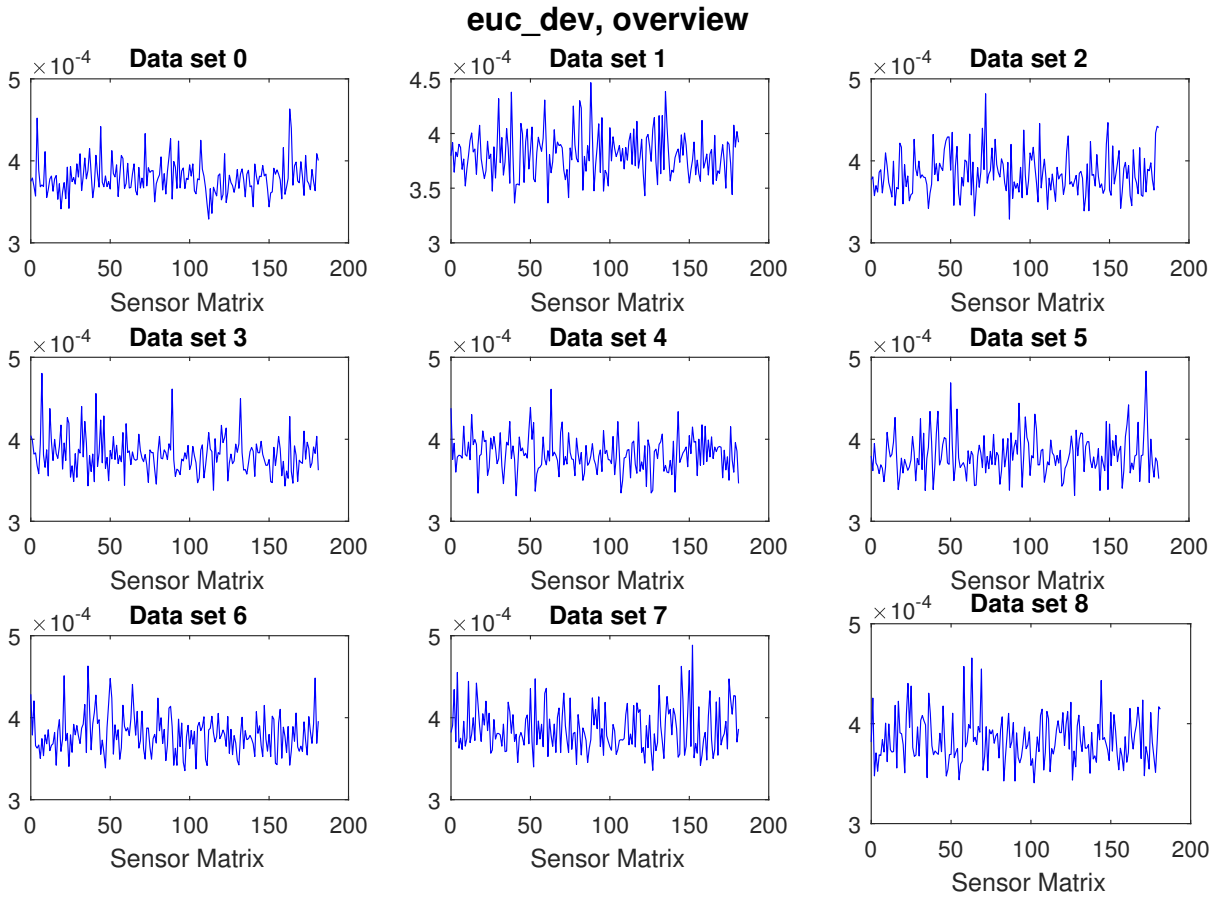

Abbildung B.82: Höchste euklidische Abweichungen bei der 15×15 12-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, alle Datensätze dargestellt. Auf die postive Aussteuerung (2) normiert.

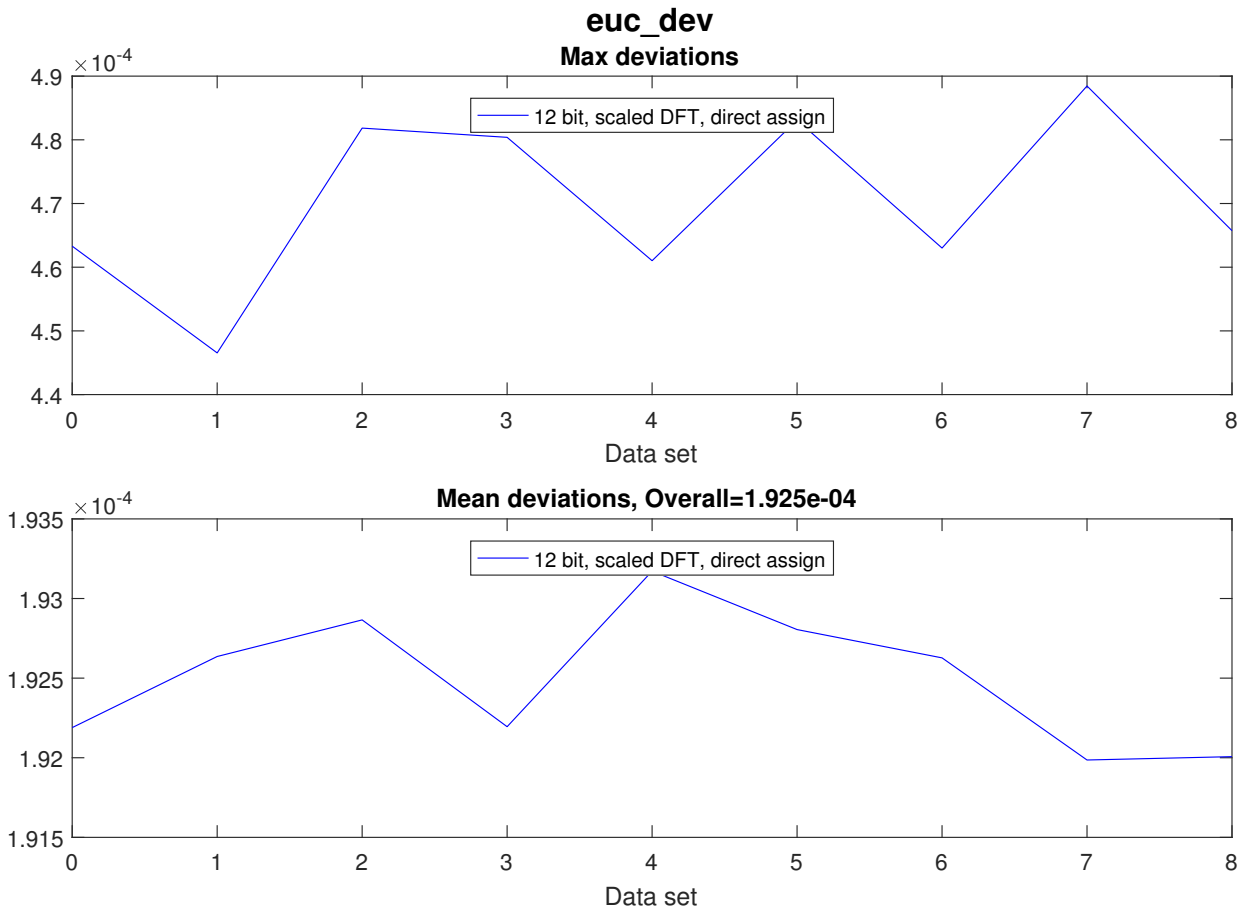

Abbildung B.83: Höchste und durchschnittliche euklidische Abweichung bei der 15×15 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, Übersicht über Datenreihen Auf die postive Aussteuerung (2) normiert.

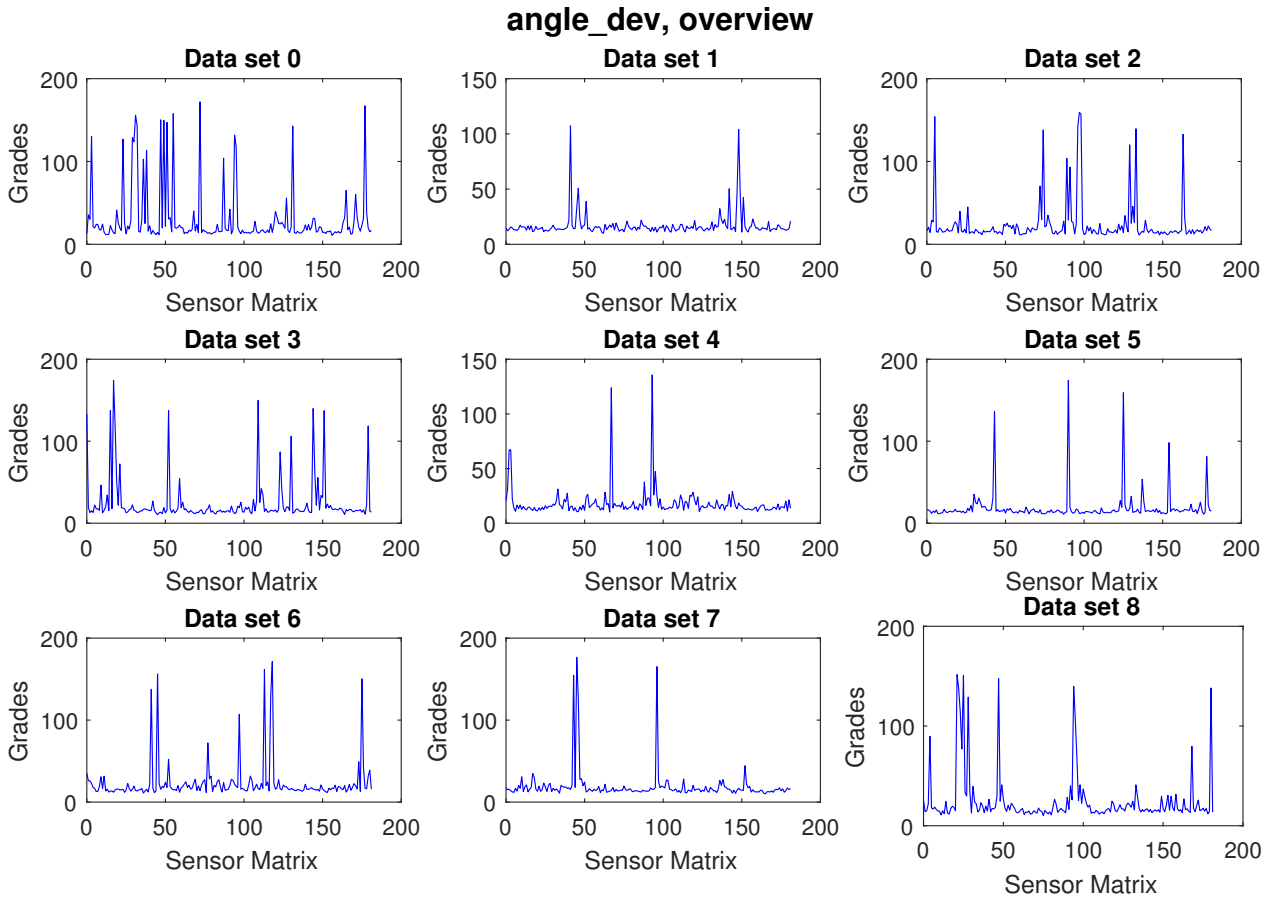

Abbildung B.84: Höchste Abweichungen des Argumentes bei der 15×15 12-Bit 2D-DFT mit der heruntеskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Alle Datensätze sind dargestellt.

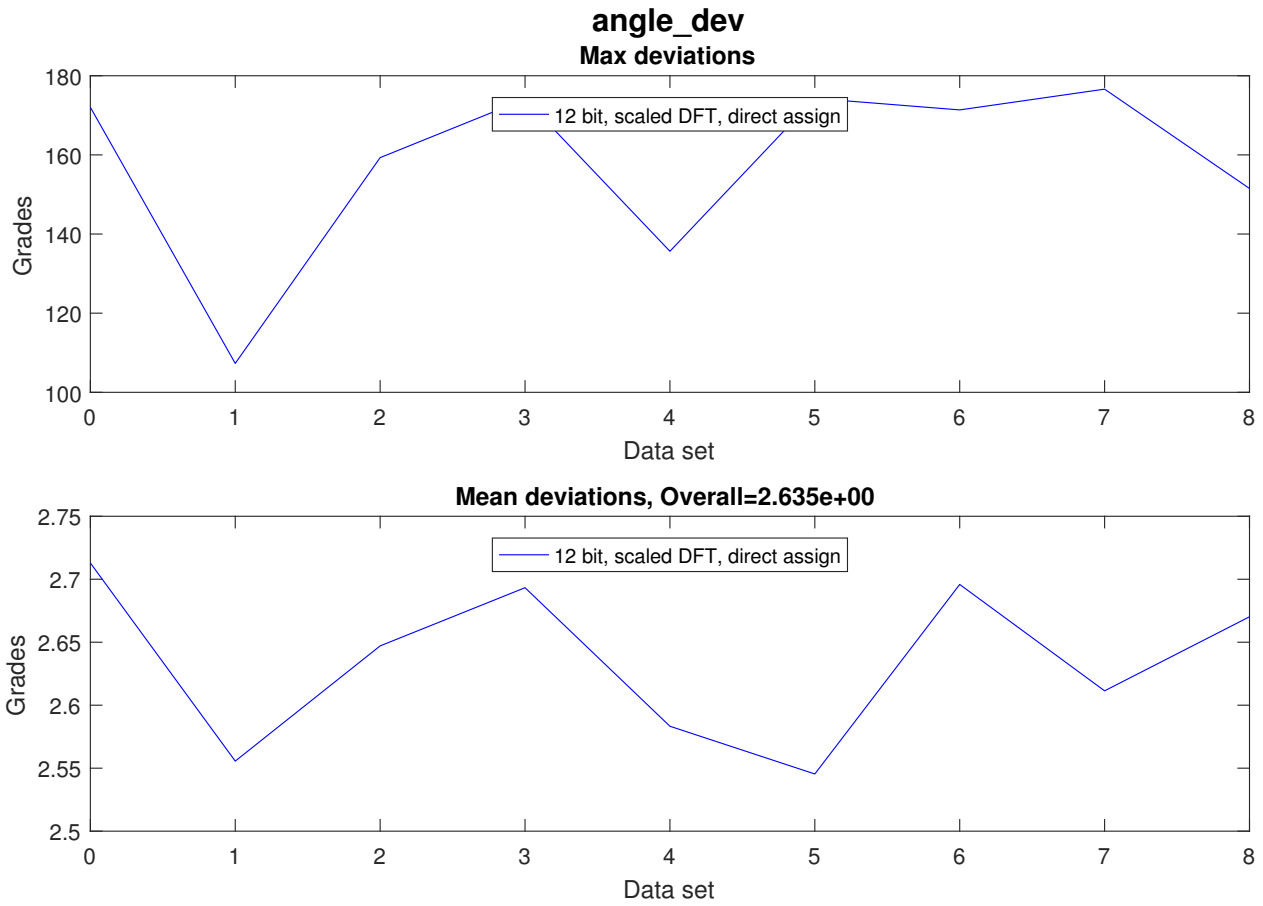

Abbildung B.85: Höchste und durchschnittliche Abweichungen des Argumentes bei der 15×15 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`.. Übersicht über Datenreihen

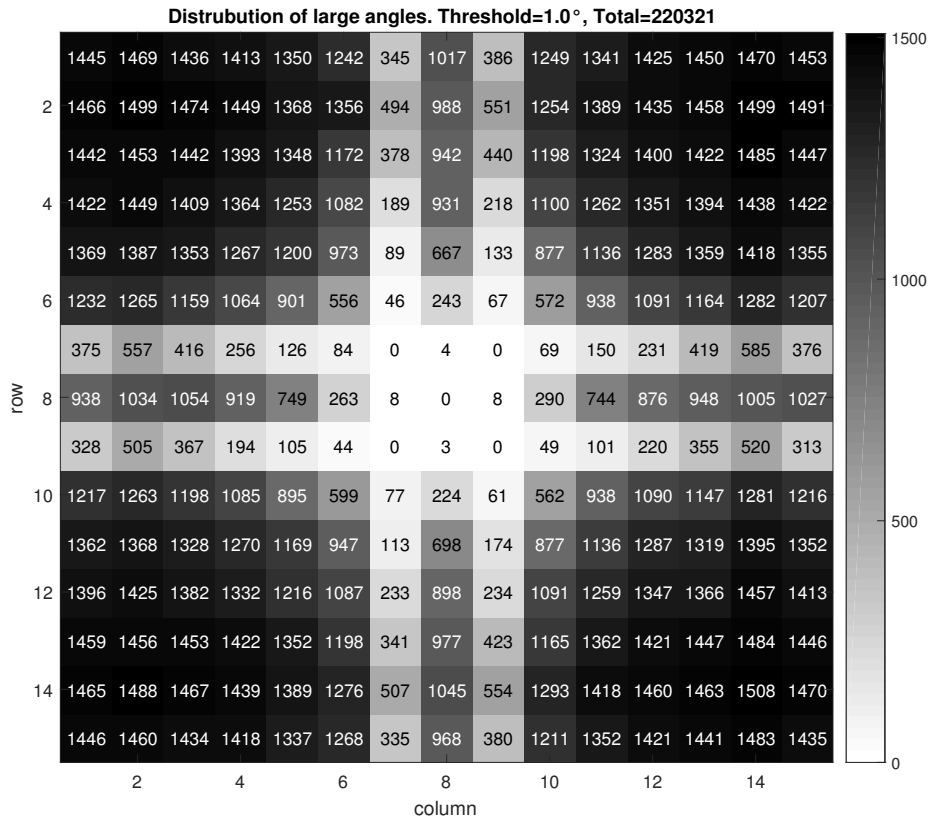

Abbildung B.86: Verteilung von Winkelabweichengen >1° in Ergebnissen von der 15×15 12-Bit 2D-DFT mit der herunteskalierten Twiddlefaktor-Matrix, nach der Korrektur mit `unwrap`. Die Mitte der Matrix entspricht der niedrigen Ortsfrequenzen.

Anhang C

Verschiedenes

C.1 Pinbelegeng der Mikrocontroller GPIO Pins und Zedboard PMod

Tabelle C.1: Pinbelegung von PMod und GPIO Pins auf Pinleiste X8.

Flachbandkabelverbindung	Pin auf der Pinleiste	µC GPIO	Pmod	Signal
1	2	PE4	JA1	EXT_DIN(4)
2	3	PC4	JA7	EXT_DIN(6)
3	13	PE0	JA2	EXT_DIN(0)
4	4	PC5	JA8	EXT_DIN(7)
5	14	PE1	JA3	EXT_DIN(1)
6	5	PC6	JA9	EXT_DIN(8)
7	15	PE2	JA4	EXT_DIN(2)
8	6	PE5	JA10	EXT_DIN(5)
9	16	PE3	JB1	EXT_DIN(3)
10	17	PD7	JB7	READY_OUT
11	8	PC7	JB2	EXT_DIN(9)
12	18	PA6	JB8	MOD_OUT(2)
13	9	PB2	JB3	EXT_INSTR(1)
14	19	PM4	JB9	EXT_DOUT(4)
15	10	PB3	JB4	EXT_INSTR(2)
16	20	PM5	JB10	EXT_DOUT(5)

Tabelle C.2: Pinbelegung von PMod und GPIO Pins auf Pinleiste X6.

Flachbandkabelverbindung	Pin auf der Pinleiste	µC GPIO	Pmod	Signal
1	3	PP0	JC1_P	EXT_RESET
2	13	PB4	JC3_P	EXT_INSTR(3)
3	4	PP1	JC1_N	EXT_CTRL_EN
4	14	PB5	JC3_N	EXT_INSTR(4)
5	5	PD4	JC2_P	reserviert
6	15	PK0	JC4_P	EXT_DOUT(0)
7	6	PD5	JC2_N	reserviert
8	16	PK1	JC4_N	EXT_DOUT(1)
9	7	PQ0	JD1_P	CLK
10	17	PK2	JD3_P	EXT_DOUT(2)
11	8	PP4	JD1_N	EXT_INSTR(0)
12	18	PK3	JD3_N	EXT_DOUT(3)
13	9	PN5	JD2_P	EXT_DOUT(7)
14	19	PA4	JD4_P	MOD_OUT(0)
15	10	PN4	JD2_N	EXT_DOUT(6)
16	20	PA5	JD4_N	MOD_OUT(1)

C.2 Hinweise zur Projekteinstellungen in Code Composer Studio

1. Beim erstellen des Projekts muss “Stellaris In-Circuit Debug Interface” unter “Connection” ausgewählt werden.
2. Es muss erläutert werden, wie die Quellcode-Dateien mit relativen Pfadnamen hinzugefügt werden können. Dies ermöglicht die Pfadunabhängige Ausführung, setzt jedoch voraus, dass die hinzugefügte Dateien ihre Stelle relativ zum Projektordner stets behalten.
   - Die Header-Dateien werden am besten mit dem ganzen Ordner hinzugefügt. Diese Einstellung befindet sich in Projekteigenschaften→CCS Build→ARM Compiler→Include Options.
     a) Im oberen Feld muss das grüne Symbol geklickt werden.

b) Dann im Dialogfenster “Variables...” klicken.

c) In der Liste entweder `PROJECT_LOC` oder `PARENT_LOC` auswählen und unten “Extend...” klicken.

d) Im Pfadbaum den Ordner auswählen.

e) “OK” klicken, bis alle Dialogfenster geschlossen werden.

- Die .c-Dateien werden am besten direkt im Projekt hinzugefügt, damit sie mitcompiliert werden. Es ist der Übersichtlichkeit halber empfehlenswert, dafür einen separaten logischen Ordner im Projekt einzurichten.

  a) Am einfachsten lassen sich die .c-Dateien per Drag and Drop einfügen. Dazu in einem Dateimanager alle hinzufügende Dateien markieren und in das Projekt (und ggf. den eingerichtetn Ordnere)im CodeComposer Project Explorer herüberziehen.

  b) Im angezeigten Dialog sicherstellen, dass das Radiobutton “Link files” ausgewählt ist und die Checkbox “ Create link relative to” gesetzt ist.

  c) Im Dropdown-Liste entweder “`PARENT_LOC`” oder “`PROJECT_LOC`” auswählen und “OK” anklicken.

3. Es müssen die TivaWare-Treiber wie in 2 beschrieben hinzufügt werden. Der Ordner “TivaWare” muss der Liste der Header-Dateien angefügt werden. Dann die dazugehörigen .c-Dateien aus dem Ordner `TivaWare/driverlib`.

4. Die entwickelten Funktionen befinden sich im Ordner `proc_sys_lib`, was erleichtert sie von mehreren Projekten gleichzeitig zu verwenden. Sie müssen auch wie in 2 beschrieben hinzufügt werden.

5. In der Dropdown-Liste “C Dialect” in Language Options “C99” auswählen.

6. In Voreinstellungen akzeptiert der Kompilator die implizite Deklaration von Funktionen. Es wird lediglich eine Warnung ausgegeben. Dies hat Potenzial üble Bugs zu produzieren. Daher muss die Zahl “225” in Projekteigenschaften→CCS Build→ARM Compiler→Advanced Options→Diagnostic Options aus dem unteren Bereich entfernt und dem

oberen Bereich hinzugefügt werden, damit es als ein Fehler behandelt würde.

7. Die Ausgabefunktionen wie `printf` brauchen die dynamische Speicherzuweisung. Das dynamische Heap ist jedoch standardmäßig abgeschaltet. Will man von diesen Funktionen Gebrauch machen, muss in Projekteigenschaften→ARM Linker→Basic Options “Heap size” auf einen größeren Wert (z. B. 512) gesetzt wird.

8. In Release-Konfiguration sollte...

   a) ...der obere Kasten in Projekteigenschaften→CCS Build→ARM Compiler→Predefined Symbols um `NDEBUG` ergänzt werden. Dies setzt die Debug-Assertionen aus.

   b) ...“Optimisation level” in Projekteigenschaften→CCS Build→ARM Compiler→Optimization auf 3 oder 4 gesetzt werden. Der größte Vorteil ist dass dann die unbenutzte Code-Abschnitte herausgeschnitten werden.

Anhang D

Inhalt des DVDs

Struktur des beigefügten DVDs:

Stammverzeichnis: Diese Arbeit, sowie einige Vorgängerarbeiten

`Cadence\logs`: Die Log-Dateien aus der Synthese mit Genus und Layout mit Innovus

`CodeComposer`: Workspace für TI Code Composer Studio. Enthält jegliche Software Projekte für Mikrocontroller, sowie Treiber und Dokumentation zum Mikrocontroller.

`doxygen`: Doxygen-Dokumentation für Mikrocontroller (Unterordner `mc`) und FPGA (Unterordner `vhdl`)

- `Matlab\proc_sys`: jegliche für diese Arbeit geschriebene Matlab-Skripte und Funktionen. Der Unterdner `testlogs` enthält die Untersuchungsergebnisse.
- `Vivado`: VHDL Code als fertige Vivado-Projekte. Zusätzlich enthält einige nützliche Dokumentation zum FPGA.

Selbstständigkeitserklärung

Hiermit versichere ich, Andrey Luzhanskij, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22 (6) PStO ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 18. Dezember 2019	
Ort, Datum	Unterschrift